HND 1 Computer Engineering II
Introduction to Microprocessors

School of Electronic, Communication and Electrical Engineering

1 Introduction

1.1 Example assembler code

```
c:\meta\68000.tab/
        LONGS
ram     EQU     $0400000
codeorg EQU     $400400
ramt8   EQU     $403FFF
ramt32  EQU     $40FFFF

lights  EQU     $E00183
ddr     EQU     $E00187

        SEGMENT word at 00400400 'ram'

        move.b  #%11111111,ddr

        move.b  #%00000001,D0
restart move.b  #7,D1
left    move.b  D0,lights
        bsr     delay
        lsl.b   #1,D0
        sub.b   #1,D1
        bne     left

        move.b  #7,D1
right   move.b  D0,lights
        bsr     delay
        lsr.b   #1,D0
        sub.b   #1,D1
        bne     right

        bra     restart

delay   move.l  #16000,D2
dloop   sub.l   #1,D2
        bne     dloop
        rts

        end
```

Notes: Words appearing in the leftmost column are *labels*. A label will be converted to an address when the assembler is run.

move.b means move a byte from the left operand to the right operand. Operands can be addresses, numbers (prefixed with #) or *registers*. The 68000 microprocessor has 8 data registers, D0–D7. Numbers can be specified in *hexadecimal* ($), *decimal* or *binary* (%).

EQU sets up a macro, e.g., `ddr EQU $E00187` is similar to `#define ddr 0xE00187` in C.

lsl.b and **lsr.b** mean (logical) shift the byte left and right. `lsl.b #1,t` is just like `t=t<<1` in C.

bne stands for 'branch if not equal' to zero.

bra stands for 'branch always'. Like 'goto'.

bsr is 'branch to subroutine' and is the way you can call a function.

rts is 'return from subroutine'.

sub.b means subtract from a byte (8 bits)

sub.l means subtract from a long word (32 bits).

See if you can work out what this program should do; write comments and draw arrows on the program to help. There are 8 LEDs connected to the output `lights`. Don't worry about the `ddr` or any of the program above `move.b #%00000001,D0`.

1.2 Thoughts

Why on earth would anyone want to run a program like this?

2 Number bases revision

2.1 Decimal

Base ten (denary) is commonly used for counting. There are ten digits (0–9). After 9 we say that we have one group of ten and no more (we write a 1, then a 0: i.e., 10). Then comes one group of ten and one more (11). Eventually two groups of ten and no more (20), ..., etc.. For example, 175 is one group of [ten groups of ten] and seven [groups of ten] and 5 [groups of units].

To multiply by ten, you just shift all the numbers left one, and pop a 0 on the right in the 'hole'.

2.2 Binary

Computers don't like tens. It's easier to say a voltage is either high or low than to say a voltage is x tenths of the way up a scale. So computers use base 2 (binary) with two digits (0, 1). After 1 we say we have one group of two and no more (10_2). Then comes one group of two and one more (11_2). After that comes one group of [two groups of two] and no [groups of two] and no [groups of units] ($100_2 = 4_{10}$). For example, 1101_2 is one group of [two groups of [two groups of [two groups of two]]] and 1 group of [two groups of [two groups of two]] and no groups of [two groups of two] and one [group of units]. It gets silly.

To multiply by two, you just shift all the numbers left one, and pop a 0 on the right in the 'hole'.

2.3 Hexadecimal

Computer programmers don't like twos. The numbers turn out far too long, e.g, $80_{10} = 1010000_2$. Hexadecimal is base 16. There are sixteen digits (0, 1, ..., 9, A, B, C, D, E, F). $2C3_{16}$ is two groups of [sixteen groups of sixteen] and C_{16} (twelve) [groups of sixteen] and three [groups of units].

To multiply by sixteen, you just shift all the numbers left one, and pop a 0 on the right in the 'hole'.

2.4 Base Conversion

Saying 'groups of' is tedious. We know that two groups of two gives us four. Two groups of four gives eight and so on. When adding together all the bunches of groups, what we're doing is multiplying the digit by its group value and finding the total for all the digits. We can work out what the examples are in denary now.

$$1101_2 = 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 13_{10}$$

$$2C3_{16} = 2 \times 256 + 12 \times 16 + 3 \times 1 = 707_{10}$$

Where 256 is 16 groups of 16. In practise you won't need to convert bases very often. Nearly all modern calculators do conversions and the computer can too. Use the base that's most convenient.

You probably did something like this at school

$$830 = 8 \times 100 + 3 \times 10 + 0 \times 1$$

and you can see that 100 is ten groups of ten.

2.5 Why hexadecimal?

Two groups of eight is sixteen. So, sixteen is two groups of [two groups of [two groups of two]]. This means that *one* hexadecimal digit can represent *four* binary digits. Said again, four binary digits represent a number from 0 to 15, and one hexadecimal digit also represents a number from 0 to 15.

At the bottom of the page, write down, in binary, the numbers from 0 to 15. Here are the first five: 0000, 0001, 0010, 0011, 0100. If you write them so they line up in a column you'll see a pattern (hence the leading 0s). Next to each binary number, write its value in hexadecimal. Now read on.

To convert hexadecimal to binary, simply look up each hex digit and copy down the binary equivalent for it. For example ...

$$6D3_{16} \Rightarrow 0110\,1101\,0011$$

To convert from binary to hexadecimal, do the same in reverse: gather up groups of four digits (from the right) and change them to their hex form.

$$11\,0101\,1111 \Rightarrow 35F_{16}$$

Converting binary $\Leftrightarrow$ hexadecimal is much easier than converting either base to base 10. That's why programmers use base 16.

Note: You may also come across octal (base 8). In octal each digit represents 3 binary digits.

2.6 Thoughts

How do you do long addition in decimal? How would it work out in binary? Is long division easiest in binary, hex or decimal?

3 Binary arithmetic

3.1 Two's complement

If we use a fixed number of digits to count with we eventually 'wrap around' back to the beginning. For example, with years, at the end of the century the year number will go '97, '98, '99, '00, '01....

Using the two digit notation, add 99 years to '45. Now add 98 years to '45. The result appears as if 99 was the same as −1, and 98 the same as −2. We can say that the year number x behaves like the number $(x - 100)$ when working in only two digits.

Because computers only work to a finite number of bits, this negative number scheme also works in binary. One particular convention is called two's complement. For convenience we shall assume a 4-bit computer is being used (they do exist, but aren't common). In two's complement, all numbers with the top bit (sign bit) set are assumed negative.

		1000	−8
0111	7	1001	−7
0110	6	1010	−6
0101	5	1011	−5
0100	4	1100	−4
0011	3	1101	−3
0010	2	1110	−2
0001	1	1111	−1
0000	0		

Check out a few values. What's $3+(-3)$? What's $2+(-6)$? Things start going strange with, say, $4+5 = -5$, but if you subtract 5 again, you do get the 4 back.

You may have noticed a pattern. To convert a number to its negative, invert all the bits and add 1. E.g.,

$$-4 = 1100 \xrightarrow{\text{invert}} 0011 \xrightarrow{+1} 0100 = 4$$

One of the remarkable properties of two's complement is that it keeps track of the sign of the numbers during addition, subtraction, multiplication and division. There is a slight complication though.

Suppose we divide a negative two's complement number by 2. This is equivalent to shifting the number right once.

$$\frac{6}{2} = 1010 >> 1 = 0101 = 5$$

This wasn't the answer we were looking for ($\frac{-6}{2} = -3$). The problem comes when the left most binary digit is (almost arbitrarily) zeroed. $-6 >> 1$ should be -3 (1101). We can correct this problem by extending the sign bit. (The sign bit is the most significant bit.) When we shift the number right, the old sign bit should be copied to the new sign bit.

Fortunately most microprocessors know this and helpfully provide two types of shift left and right instructions. Here are the 68000 microprocessor codes for shifting.

lsl	logical shift left
lsr	logical shift right
asl	arithmetic shift left
asr	arithmetic shift right

The difference between the logical and arithmetic versions is that the arithmetic shifts extend the sign bits, so that negative numbers can be shifted around. In effect lsr/lsl is the unsigned version, and asr/asl the signed version. (As if that wasn't enough, there are also ror/rol (rotate) instructions, which copy the bit that fell off one end back on the other end.)

3.2 Example multiplication

$$5 \times -13 = -65$$

We shall use 8 bits, two's complement. To verify the −13, add $13 (= 1101_2)$ to it. The product $-13 \times 5 = -65$ is on the right.

$$\begin{array}{r} 0000\,0101 \\ \underline{\times 1111\,0011} \\ 0000\,0101 \\ 0000\,1010 \\ 0101\,0000 \\ 1010\,0000 \\ 0100\,0000 \\ \underline{+1000\,0000} \\ 1011\,1111 \end{array} \qquad \begin{array}{r} 1111\,0011 \\ \underline{\times 0000\,0101} \\ 1111\,0011 \\ \underline{+1100\,1100} \\ 1011\,1111 \end{array}$$

Note that the 8-bit system was carefully chosen so that no values 'overflowed'. When multiplying numbers, you may have noticed that if you multiply a 2 digit number by a 3 digit number you get a 4 or 5 digit answer. In general, if you multiply an n-digit number by an m-digit number you get a $(n+m)$-digit result (the first digit may be a zero). The same is true for binary. 5 $(= 101_2)$ is a 3-bit (3-digit) number while −13 is a 5-bit number when the sign bit has been allowed for. Hence we need at least an 8-bit result. Handy hint: don't let binary numbers overflow unless you're *really* sure what you're doing.

3.3 68000 arithmetic operations

Here are the arithmetic operations that the 68000 microprocessor knows how to do by itself. The first operand is called the 'source' operand, and the second is called the 'destination'. The source and destination are combined together and stored in the destination operand.

add	adds two numbers
sub	subtracts the source from the destination
mul	multiply
div	divides the destination by the source
neg	negates a number (swaps the sign of a number)
move	does nothing: just moves the source to the destination
clr	makes the destination zero

`neg` and `clr` are slightly special because they only take a destination operand.

There are examples of `add` and `sub` in the example programs. When you use these commands, you will need a suffix to tell the 68000 whether it should operate using bytes (e.g., `add.b #3,D0`) or words (16-bits) or long words (32-bits) (e.g., `sub.l #$C0000,D3`).

There is a shorthand in most of the books. For example, written under `add` you might see an entry like this...

ADD	(Destination) + (Source) $\longrightarrow$ Destination

It simply means that the destination is loaded with the source + destination.

3.4 68000 logical operations

These operators are known as logical bitwise operators. They operate on all bits at once in parallel. The source and destination work just like the arithmetic operations.

and	logical and
or	logical or
eor	logical exclusive-or
not	logical not

The `not` here also takes just a destination operand.

Here's another example of a cryptic shorthand description...

EOR	(Destination) $\oplus$ (Source) $\longrightarrow$ Destination

3.5 Example program

This week's example program is a simple game. The row of 8 switches and LEDs are set up as before with each switch next to an LED. One of the LEDs will flash quickly. You then have to flick the switch corresponding to that LED. The flash time gets progressively shorter until you flick the wrong switch.

Note the subroutine `waitforswitch` which sits doing nothing until a switch is flicked. The `eor.b` ensures that the loop will drop out only when a switch is *changed*. (`beq` means branch if equal (to zero).)

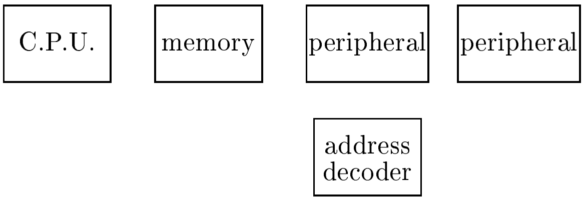
Remember that `D0` etc. are like variables in C.

There is another new command: `bset`. This little beastie sets the source bit of the destination. For example, if D0 is zero, then `bset.l #2,D0` will make D0 equal to 8. (Bits are numbered from zero, starting from the right hand side.)

```
c:\meta\68000.tab/
        LONGS
        SEGMENT word at 00400400 'ram'

switch  EQU     $E00181
sddr    EQU     $E00185
lights  EQU     $E00183
lddr    EQU     $E00187

      move.b  #%00000000,sddr
      move.b  #%11111111,lddr

      clr.l   D7              ; D7 is 'random' number
start move.l  #$70000,D5      ; D5 is the delay time

loop  clr.b   D1              ; D1 is lights output
      move.b  D7,D0           ; D7 is 'random' number
      and.b   #%111,D0        ; take bottom 3 bits only
      bset.b  D0,D1           ; set bit D0 of the lights
      move.b  D1,lights       ; finally set the lights

      move.l  D5,D6           ; move delay value into D6
      bsr     delay           ; delay is given by D6
      move.b  #0,lights       ; turn all lights off

      bsr     waitforswitch ; returns D0=switch
      cmp.b   D1,D0           ; is D1=D0?
      bne     fail            ; no!  game over

      move.l  #$70000,D6
      bsr     delay           ; take a short break

      move.l  D5,D6           ; set delay to 7/8 of last
      asr.l   #3,D6           ; delay by subtracting 1/8
      sub.l   D6,D5           ; so D5=D5-(D5>>3)

      bra     loop            ; and have another go...

fail  move.l  #60000,D6       ; you have failed!
      move.l  #10,D5
fl1   move.b  #$AA,lights     ; flash lights nastily
      bsr     delay
      move.b  #$55,lights
      bsr     delay
      sub.l   #1,D5
      bne     fl1
      bra     start

delay move.l  D6,D0
dloop sub.l   #1,D0
      bne     dloop
      rts

waitforswitch
      move.b  switch,D4
wl1   move.b  switch,D0
      add.b   #1,D7           ; make D7 'random'
      eor.b   D4,D0           ; mask unchanged switches
      beq     wl1
      rts

      end
```

4 Computer architecture

4.1 Basic structure



C.P.U. controls the system. Nearly every signal gets goes through the C.P.U.[1].

memory: stores numbers for later retrieval. There are two main forms of memory: ROM[2] and RAM[3]. The C.P.U. can store and retrieve numbers from RAM but can only retrieve numbers from ROM. ROM is used for permanent storage. (Note that memory has its own internal address decoder.)

peripheral: there can be lots of peripherals in a single computer. They include things like keyboards, floppy disk drives, serial ports, network ports, etc.. Sometimes the screen is a peripheral, but more often the screen appears as a large block of memory. The peripherals are often referred to as I/O[4]. Anything that the C.P.U. can access that isn't memory is normally a peripheral.

address decoder: decodes which addresses are to be sent to which peripheral. Some peripherals may need more than one address.

In some systems the peripherals have their own address bus, separate from the memory address bus. The 8086 family of processors (used in I.B.M. compatible machines) use separate signals from the C.P.U. to tell the devices whether the address bus is currently a peripheral or memory address bus. The 68000 family of processors map the memory and peripherals into a single address bus.

4.2 The hardware

In addition to the address bus and data bus, there are control signals to say what the C.P.U. is doing. For instance, there is a R/$\overline{\text{W}}$ signal to say whether the signals on the data bus are going to (Read) or from (Write) the C.P.U.. The CK (clock) signal allows the processor to time all its functions.

The major peripherals are usually (but not always) large ICs[5] or sets or ICs. They sit on the data bus (and sometimes part of the address bus too). The address decoder will inform the peripheral when it's wanted, and the R/$\overline{\text{W}}$ signal tells it whether it should put data on the data bus, or read the data from the data bus.

So called *glue logic* (simple logic gates) is often used to convert the C.P.U.'s control signals into the peripheral's control signals (i.e., glue them together).

4.3 Programmer's model

Often the programmer doesn't want to know what the hardware is (because it will changes as the designers get new bright ideas). As long as the software 'sees' the same machine, no matter what the hardware is this week, the programs will still work. The system that programs see is referred to as the programmer's model.

4.4 68000 programmer's model

The 68000 programmer sees an environment like this.

- The address bus is 24 bits wide (so it spans from 0 to 16 MB). Not all of this is filled with memory. The peripherals also live in this space.
- The C.P.U. clock is 8MHz, but each instruction takes around 8 clock cycles to complete.
- There are 8 registers, D0–D7, that can be accessed quickly. They are all 32 bits wide.
- There are 7 registers for storing addresses, A0–A6.
- Register A7 is used as a stack pointer.
- The *status register* contains a set of flags (single bits) that store some of the results from the previous operation. These flags include

  zero: set if the last result was zero

  negative: set if the last result was negative

  carry: set if the remaining carry bit was 1

  overflow: if the last operation caused a number to overflow the length of the register then this bit will be set

- Operations can be performed on…

[1]Central Processing Unit
[2]Read Only Memory
[3]Random Access Memory — confusing notation becase ROM is also random access. Random access means that any part can be accessed, as opposed to sequential access devices.
[4]Input/Output
[5]Integrated Circuits: chips

bytes (8-bits) suffix `.b`

words (16-bits) suffix `.w`

long words (32-bits) suffix `.l`

- Operations can act on (with example)...

numbers: `move.l #12,D0`
moves the number 12 into D0

registers: `add.l D3,D2`
adds D3 to D2

addresses: `move.l 12,D0`
moves the contents of address location 12 into D0

addresses in registers: `move.l (A3),D0`
moves the contents of the address location given by A3 into D0

There are other *addressing modes* that are variations on these.

4.5 80x86 programmer's model

The 68000 model is quite simple and straightforward. Certain other systems aren't as nicely made. The following is a description of the 80x86 programmer's model. If you're interested in programming I.B.M.-compatible machines then this may be interesting, otherwise skip it.

The forerunner to the 8086 could access 64 KBytes of address space (it had a 16-bit address bus). All the instructions were designed so that the 16-bits of address fitted in nicely. The 8086 was designed to access 1 MByte (20-bit address bus) but still be binary compatible with the old processor (the same code can run on the different processors). Thus it came to pass that the segmented memory architecture was built. Whenever a piece of memory was accessed, it was designated to a *segment* (64 KB) of memory. The different address registers could each use a different segment. The final memory address was made by

$$\text{final address} = 16 \times \text{segment} + \text{offset}$$

The segment and offsets are 16-bit numbers, giving the $16 \times 64 = 1024$ KB total address space.

The 80286 allowed a flat (i.e., not segmented) memory model with a 24-bit (16 MByte) address bus. This new space was called *extended* memory (XMS[6]). Later processors (80386 onwards) have 32-bit registers so that much more memory can be accessed. But because the code needed to access this extra memory was not compatible with the 8086 (which DOS and Windows, etc., were originally written for) most code is still limited to 1 MB. The processor can switch itself into 'protected mode' to access the extra memory.

As a stopgap, a system called *expanded* memory (EMS[7]) was invented. This allowed 16 KB blocks of memory to be 'mapped' into the 1 MB address range. By changing which 16 KB block of memory was mapped in, programs could use more memory than normal — up to several megabytes.

All this was developed by the Intel Corporation (the company that makes the 80x86 family of processors). I.B.M. also made some memory models for their IBM/PC, PC/XT and PC/AT range of microcomputers.

I.B.M. placed the ROM memory between 768 KB and 1 MB, and placed screen memory between 640 KB and 768 KB. Hence programs were limited to the lowest 640 KB of memory (conventional memory). Not all the ROM area was used (a lot of it was set aside for the ROMs on expansion cards) so people started using the spare address space. Expanded memory was placed here and other bits were filled with RAM by the memory manager[8]. These little bits of memory are called the Upper Memory Blocks (UMB).

Following this? It gets worse.

There is a bug in the 80286 that was soon exploited. If you accessed the piece of memory at segment:offset FFFF:0010 (i.e., just off the top of the 1MB address space) then the 8086 would read location 0, but the 80286 accessed memory above 1 MB. This gave an extra 64 KB (less 16 bytes) of usable address space. This was called the HMA[9] (yet another TLA[10]). But different machines connected the A20 line[11] differently, so there is often an A20 configuration option in the machine set-up.

Finally, peripheral memory uses a different address space to normal memory. Peripherals have 16-bits (64 KB) of space, but I.B.M. only decoded the lowest 10 bits, leaving 1 KB of I/O space.

There are 4 main 16-bit registers in the 8086 (AX, BX, CX and DX). There are also some 16-bit address pointers (SI, DI, SP and BP) and the 16-bit segment registers (CS, DS, SS and ES). Unfortunately some instructions are limited to using certain registers. Fortunately though, the instructions, addressing modes and status flags of the 8086 are very similar to the 68000 counterparts.

[6] Extented Memory System

[7] Expanded Memory System; sometimes called LIM EMS after Lotus, Intel, Microsoft who wrote the specification.

[8] What's a memory manager? It's like an address decoder for memory. It maps bits of memory from one place to another place by changing its address. A memory manager is built into the 80386 and later processors.

[9] High Memory Area

[10] Three Letter Acronym

[11] Address bus line 20 — you need 20 bits to span 1 MB.

5 Program structure

5.1 Branches

Branches are the machine code equivalent of goto. They are the only type of flow control in a 68000 program: there are no while {...} or for {...} constructions. The branches are conditional, that is to say the instruction will either branch to its pointer, or it won't.

The conditions under which the branch will work depend on the flags ('carry', 'overflow', 'equal to zero' and 'negative'). It's worth knowing some of these conditions, so here are the useful ones....

`beq`	branch if equal [to zero]
`bne`	branch if not equal [to zero]
`bgt`	branch if greater than
`bge`	branch if greater than or equal
`blt`	branch if less than
`ble`	branch if less than or equal
`bpl`	branch if positive (plus)
`bmi`	branch if negative (minus)
`bra`	branch always (unconditional)

The branches work very well with the compare (`cmp`) instruction. For example,

```
        clr.l   D0      ; sets D0 to zero
loop    add.l   #1,D0
        cmp.l   #5000,D0
        blt     toolow
        cmp.l   #70000,D0
        bgt     toohigh
        move.b  #0,lights
        bra     loop
toohigh move.b  #red,lights
        bra     loop
toolow  move.b  #green,lights
        bra     loop
```

This short snippet will put the green light on for a short while, then no lights, then the red light for a long time.

Each branch tends to associate itself with the previous instruction. (The `cmp` instruction sets the flags, but so does `move`, `clr`, `add`, etc..)

The equivalent C program is something like

```
x=0;
do {
    x++;
    if (x<5000)  {*lights=red;   continue;}
    if (x>70000) {*lights=green; continue;}
    *lights=0;
    } while(1); /* forever */
```

5.2 Subroutines

Function calls can be used. The `bsr` (branch to subroutine) mnemonic is used for this. However, you can't pass any parameters about. Of course, you can arrange for a subroutine to take an argument in, say, D0, and return the result in D5 though.

Every function call must end with an `rts`, just like C functions must end in `return`.

```
        move.l #1,D0
loop    bsr    incr
        cmp.l  #10,D5
        bne    loop
        bra    somewhere

incr    add.l  D0,D5
        rts
```

5.3 A note on jumps

The 68000 allows two types of branching: `bra` and `jmp`. It also allows two types of subroutine calling: `bsr` and `jsr`. The `j` versions are called jumps. The difference between the two is subtle and to do with the encoding of the instruction. A branch hops to a *relative* address but a jump hops to an *absolute* address. A branch might be coded, 'branch to the address 5 bytes up from here' whereas a jump would say 'jump to address $400465'. In self-contained programs there's not much difference between the two forms.

5.4 The stack

The 68000 processor doesn't have any of its own memory (apart from the registers) and you may have noticed that if you call a subroutine, then somehow the program flow has got to get back to where it was called from. This means that the return address must be stored somewhere.

If one subroutine calls another subroutine, the first return address mustn't be overwritten by the second. So each level of subroutine must store its return address in a different place.

This is done by a mechanism called the *stack* (also called a LIFO — last in, first out). It's called a stack because it works like a stack of paper.

Every time a subroutine is called, the return address is put onto the stack (in the paper version, you write the address on a piece of paper and put it on top of the pile). When the processor executes an `rts` instruction it reads the return address from the stack (you pick up the top piece of paper). This way the C.P.U. never gets confused about where it is and where it came from.

The stack is actually a piece of RAM. The address register A7 points to the address that's currently the top of the stack. When something goes onto the stack, the size of the data is subtracted from A7 (so it points to a new piece of memory) then the data is written to the address in A7. To read stuff back from the stack you read from A7, then add the length of the data to A7 so it points to the previous value in the stack.

The stack can be handy for other things too. Suppose you are using all the registers and you need another one for your splendid new subroutine. It would be nice to store the value of a register, then use the register for your own purposes, and read the value back when you've finished with it.

Here's how it's done. Use

`move.l` *value*`,-(A7)`

to store a value on the stack and

`move.l (A7)+,`*destination*

to get it back again. The `-(A7)` means 'decrement A7 by the length of the operand then use the address in A7'. Compare this with the normal (e.g.) `(A2)` which means 'use the address in A2'. The `(A7)+` means 'use the address in A7, then add the length of the operand to A7'. In these cases the length of the operand is 4 (because `.l` means long word: 4 bytes).

Here's an example delay subroutine that leaves all the registers unchanged.

```
delay   move.l  D0,-(A7)   ;store D0 on stack
        move.l  #10000,D0
dloop   sub.l   #1,D0
        bpl     dloop
        move.l  (A7)+,D0   ;retrieve D0
        rts
```

Be extremely careful about using the stack in subroutines: always remember to take off whatever you put on, otherwise the `rts` instruction will take your stored values off and they'll be wrong.

Other processors have special instructions for manipulating the stack. For instance, the 80x86 family use the `push` instruction to push stuff onto the stack, and `pop` to pop values back off.

5.5 'Meta' directives

Meta is the name of the assembler and development system that's used in the University. The assembler cuts down on the amount of work you have to do to write a program (although sometimes it doesn't seem that way). If you look back through past programs, you will see things like...

```
c:\meta\68000.tab/
        LONGS
ram     EQU     $0400000
lights  EQU     $E00183
red     EQU     128

label   move.b  #red,lights
        end
```

Everything above is a directive — they don't produce any code — apart from the move.b which is an instruction to be assembled. A lot of this can be ignored, but you should know these.

The `c:\meta\68000.tab/` is the filename of the rules for assembling code for the 68000 processor.

The `EQU` tells the assembler that one word should be replaced with another. For instance, wherever the assembler sees the word `lights` it will replace it with `$E00183` (in this case, the address of some LEDs). This makes programs easier to read.

The `label` is a bit like the `EQU` but wherever the assembler sees `label` it will replace it with the address of the instruction following the label's definition.

Finally, `end` marks the end of the file. Meta is quite insistent about it.

5.6 Editing programs

Program development is done on an I.B.M. compatible machine and the programs are run on a 68000 target board.

The Meta system includes a program editor and also a terminal program to talk (electronically) to the target 68000 computer. (Terminal means end: the terminal program is at one end of the cable connecting the computers, and the target board is at the other.)

Once your program has been written, there are three steps to perform before you can run it.

Your assembly code must be assembled into machine code (numbers), so once you've saved your file you must run the assembler on it.

Then your binary masterpiece must be joined up with any other neccessary bits. (The process is called linking, but since most programs are only one file, you won't need to link any files. Very large programs might be made of hundreds of smaller files.)

Finally the finished machine code must be transferred from the development computer to the target board where it can be run.

Fortunately, because these three stages are very common, there's a special program (called make) that does all these in one go. You will have to give make a file to tell it *how* to do all the above though.

6 Peripherals

6.1 Parallel digital I/O

As far as computers are concerned, parallel digital I/O (sometimes just called parallel I/O) is about as simple as it gets. Parallel means working at the same time and I/O stands for input output. You can make a parallel output simply by connecting the computer's data bus to a latch, and the output from an address decoder to the latch's trigger. When the address decoder activates the latch, the latch remembers the value on the data bus (the value that the processor has written) and copies it to its own outputs.

A parallel output gives you several independent digital output lines. Each line corresponds to a bit of data. The lines can either be high (normally 5 V) or low (0 V).

A parallel input gives you several independent digital input lines. When the computer reads the port, each line will correspond to a bit in the result.

The parallel I/O on the Kaybug system is more flexible. (For the full gory details, find a '6522 Data Sheet'. It's, erm, sophisticated.) The I/O pins can be either inputs or outputs. In fact they are normally inputs, but it you write a 1 to a bit in the port's *data direction register* (`DDR`) then the corresponding port bit will become an output.

There are two 8-bit ports on each 6522 chip — PORTA and PORTB. Each port has its own data direction register — DDRA and DDRB. In the dual V.I.A. box there are two 6522s, giving four ports: PORTA1, PORTB1, PORTA2, PORTB2. (V.I.A. stands for versatile interface adapter) and there is a connector for each 6522 chip.

Suppose there were LEDs connected to bits 0, 2 and 4 of PORTA2 and buttons connected to bits 1, 3 and 5. The LEDs are outputs but the switches are inputs. Nothing is connected to bits 6 and 7. This code fragment will wait for the first button to be pressed then light the second LED. Note the value put into `DDRA2`.

```
      move.b   #%00010101,DDRA2
      move.b   #0,PORTA2
loop  btst.b   #1,PORTA2
      bne      loop
      move.b   #%00000100,PORTA2
```

Notes: `btst` tests a numbered bit, `bne` is branch if not equal (to zero). Bit 0 is the right-most digit in a binary number. This is simply so that the value of each bit is 2^n where n is the bit number. A value of 0 was written to PORTA2 just to clear any stray value that may be left in it.

6.2 Analogue I/O

An analogue voltage can take any value. Since computers only understand integers (a consequence of bits being only 1 or 0, never, say, 0.2) there must be a conversion somewhere.

The conversion of an analogue voltage into a digital value that the computer can understand is quite involved. Fortunately there are chips that can do the job straight off. ADC (analogue to digital conversion) will be covered next year.

Digital to analogue conversion is easier. A DAC (digital to analogue converter) can be made from a parallel output and a handful of resistors. In an 8-bit DAC the most significant bit is 'worth' 128. The next bit down is worth 64, and so on. So to make a DAC, the digital outputs only have to be scaled to give proportional voltages.

There are chips to do this process as well. They are useful because the resistor tolerences are already sorted out.

6.3 Serial I/O

Serial means one after the other. On a serial port each bit to be transmitted is written onto a single wire, one bit after the other. For sending bytes, this saves on 7 bits of wire, but it is normally slower than parallel I/O. Also, serial ports tend to communicate only with other serial ports — it isn't easy to interface hardware directly to them.

Although there are two serial ports on the Kaybug machine, programming them is not easy. There are subroutines in the Kaybug ROM to manipulate the serial ports, so find out about them if serial I/O is absolutely neccessary.

6.4 The P.I.T.

There is a useful peripheral on the Kaybug board called the parallel interface and timer (P.I.T.). It contains some parallel I/O and a timer.

The timer simply counts down from a pre-loaded value at 250 kHz. The timer itself is 24 bits wide, giving a maximum value of just over 16 million, so counting down at 250 kHz gives a maximum time of about a minute. The timer can be used to time things — it works like a stopwatch. The C.P.U. tells the timer when to start counting down, and when to stop, using the *timer control register* (`TCR`). The value left in its *counter* (`CNTR`) can then be read back by the program.

Unfortunately the timer chip is connected to the data bus by only 8 bits. Since the timer is 24 bits wide, it has to be split up into three 8-bit bytes (high, medium and low). Reading back the value of the timer into a (long) register is fairly simple, though tedious.

```
clr.l   D4
move.b  CNTRH,D4
lsl.l   #8,D4
move.b  CNTRM,D4
lsl.l   #8,D4
move.b  CNTRL,D4
```

Here, D4 is first cleared, then the high byte of the timer (`CNTRH`) is moved into D4. Then the whole of D4 is shifted left (`lsl`) by 8 bits to make room for the timer's next byte.

The counter starts counting down from the value in the *counter preload register* (`CPR`) which is also 24 bits wide, so it has a high, medium and low byte too.

6.5 An echo

```
        ; set up DDRs and CPR
        move.b  #0,CPRH
        move.b  #0,CPRM
        move.b  #10,CPRL
        move.b  #$FF,dacddr
        move.b  #$00,adcddr
        move.b  #$FF,ledddr
        move.b  #$00,swiddr
        ; starts the timer
        move.b  #st,TCR
        ; setup the ADC strobes
        move.b  #%10100000,PCR1

        ; this is the memory buffer
        move.l  #memstrt,A2
        move.l  #memend,A1
        ; A0 is the mem counter
        move.l  A2,A0

        ; the main loop
loop    move.b  (A0),D0
        bsr     putsample
        bsr     lights
        bsr     delay
        bsr     getsample
        move.b  D0,(A0)+
        cmp.l   A0,A1
        bne     loop
        move.l  A2,A0
        bra     loop

putsample ; outputs from D0
        move.b  D0,dac
        rts

getsample ; inputs to D0
        move.b  adc,D0   ; read value
        move.b  D0,adc   ; start conversion
        rts

lights  ; flashes the lights from D0
        ; find average signal level in D5
        move.l  D5,D6
        lsr.l   #4,D6
        sub.l   D6,D5
        add.l   D0,D5
        ; then convert it for lights
        clr.l   D7
        move.b  D0,D7
        sub.b   D6,D7
        bpl     pos
        neg.b   D7
pos     move.b  #1,D6
lloop   lsl.l   #1,D6
        lsr.l   #1,D7
        bne     lloop
        sub.b   #1,D6
        move.b  D6,leds
        rts

delay   ; delays by value of switches
        move.b  TSR,D7
        beq     delay
        ; stop timer
        move.b  #0,TCR
        ; move switch val to CPR
        move.b  switch,CPRL
        ; restart timer
        move.b  #st,TCR
        rts
```

7 Example program

7.1 One morning...

boss: I'm building a 100-lane Eight-pin Bowling Megaplex in my back garden. I want you to design the controlling computers.
you: Urk.
boss: I want a prototype by the end of the day.
you: Eek!

7.2 8-pin bowling

8-pin bowling is a simplified (and fictitious) version of 10-pin bowling. The game is made of 8 frames. In each frame you get two balls to try to knock over as many of the 8 pins (skittles) as possible. To simplify programming, there are no 'strikes', 'scores', extra frames or uncomfortable shoes.

7.3 The prototype

The subroutine `pstring` prints the program status back to the computer terminal. The score-so-far is also printed every frame.

The switch box is used instead of pins, and when a switch is flipped, it's like a pin has been knocked over. The lights on the switch box show which pins are standing and which pins have just been knocked over. A stepper motor pretends to be a pin-setting-up machine.

7.4 The program

```
c:\meta\68000.tab/
        LONGS
ram     EQU     $0400000
codeorg EQU     $400400
ramt8   EQU     $403FFF
ramt32  EQU     $40FFFF
        ; VIA register addresses
DDRA1   EQU     $E00187
DDRB1   EQU     $E00185
PORTA1  EQU     $E00183
PORTB1  EQU     $E00181
DDRA2   EQU     $E001A7
DDRB2   EQU     $E001A5
PORTA2  EQU     $E001A3
PORTB2  EQU     $E001A1
;*****************************************************
        SEGMENT word at 00400400 'ram'

; useful aliases for post addresses
MOTOR   EQU     PORTA2
LEDS    EQU     PORTA1
SWITCH  EQU     PORTB1

        move.b  #%11,DDRA2  ; stepper motor
        move.b  #0,MOTOR
        move.b  #255,DDRA1  ; LEDs
        move.b  #0,LEDS
        move.b  #0,DDRB1    ; switches

        move.l  #welcome,A6 ; print welcome message
        bsr     pstring
        bsr     resetpins
        move.l  #scores,A0  ; this is the score line
        move.b  #1,frame

loop    bsr     playframe

        move.b  #32,(A0)+   ; add a space to the score
        move.l  #line,A6    ; and print it
        bsr     pstring

        add.b   #1,frame
        cmp.b   #8,frame
        ble     loop
        bsr     newgame
        bra     loop

playframe ; plays one frame of 8-pin bowling
        ; A0 points store line position
        move.l  #throw1,A6
        bsr     pstring
        bsr     readpins    ; returns pins in D0
        bsr     countpins   ; returns count in D1
        add.b   #48,D1      ; change to ASCII digit
        move.b  D1,prognum
        move.l  #progress,A6
        bsr     pstring
        bsr     flashpins   ; flashes LEDs in D0
        move.b  D1,(A0)+    ; store result in scores

        move.l  #throw2,A6  ; same again for 2nd throw
        bsr     pstring
        bsr     readpins
        bsr     countpins
        add.b   #48,D1
        move.b  D1,prognum
        move.l  #progress,A6
        bsr     pstring
        bsr     flashpins
        move.b  D1,(A0)+

        bsr     resetpins
        rts

readpins ; waits for a ball to come
        ; then reads the pins knocked over
        move.l  D1,-(A7)
        ; wait for a switch to be changed
        move.b  SWITCH,D0
rploop  move.b  SWITCH,D1
        eor.b   D0,D1
        beq     rploop

        ; wait for switches to stop changing
        bsr     delay
        ; read switches
        move.b  SWITCH,D1
        eor.b   D1,D0
        move.l  (A7)+,D1

        ; only standing pins can be knocked over
        and.b   standing,D0
        ; bit in D0 is set if pin just toppled...
        ; so update standing
        eor.b   D0,standing
        rts
```

```
countpins ; count how many bits are set in D0 into D1
        move.l  D0,-(A7)    ; it gets corrupted
        move.l  D2,-(A7)    ; a counter for all bits

        clr.l   D2
        move.b  D0,D2
        move.b  D2,D0       ; clear top bits of D0

        move.b  #8,D2
        move.b  #0,D1
cploop  cmp.l   #0,D0
        beq     cploope     ; finish if no bits left
        lsr.l   #1,D0       ; LSB goes into carry flag
        bcc     cploop
        add.b   #1,D1
        bra     cploop
cploope move.l  (A7)+,D2
        move.l  (A7)+,D0
        rts

flashpins ; flash the LEDs from D0
        ; pins standing are lit
        ; pins knocked over are flashed 5 times
        ; pins not standing are dark
        move.l  D0,-(A7)
        move.l  D1,-(A7)
        ; have to flash standing pins & pins toppled
        ; then just standing pins
        or.b    standing,D0
        move.b  #5,D1
fploop  move.b  D0,LEDS
        bsr     delay
        move.b  standing,LEDS
        bsr     delay
        sub.b   #1,D1
        bne     fploop
        move.l  (A7)+,D1
        move.l  (A7)+,D0
        rts

resetpins ; make the pins all stand up
        ; rattle the motor a bit for show
        move.l  D0,-(A7)
        move.l  #30,D0
        bsr     stepmotor
        move.l  #-30,D0
        bsr     stepmotor
        ; reset all pins to standing
        move.b  #%11111111,standing
        ; LEDs all on
        move.b  standing,LEDS
        move.l  (A7)+,D0
        rts

stepmotor ; step the motor D0 places
        ; D0=0 on exit
        move.l  D1,-(A7)
        move.b  #0,D1       ; D1 is the direction bit
        cmp.l   #0,D0       ; negative values step
        bpl     smloop      ; the motor backwards
        neg.l   D0          ; make it positive
        move.b  #2,D1       ; change the direction bit
smloop  eor.b   #1,D1
        move.b  D1,MOTOR
        bsr     tdelay
        eor.b   #1,D1       ; steps on falling edge
        move.b  D1,MOTOR
        bsr     tdelay
        sub.l   #1,D0
        bpl     smloop
        move.l  (A7)+,D1
        rts

newgame ; resets the scores
        move.l  #line,A6
        bsr     pstring
        move.l  #final,A6
        bsr     pstring     ; print game over message
        move.l  #blankscore,A0
        move.l  #scores,A1
ngloop  move.b  (A0)+,D0
        move.b  D0,(A1)+
        cmp.b   #0,D0
        bne     ngloop
        rts

delay   ; short ~0.2s delay
        move.l  D0,-(A7)
        move.l  #$20000,D0
dloop   sub.l   #1,D0
        bpl     dloop
        move.l  (A7)+,D0
        rts

tdelay  ; tiny ~0.01s delay
        move.l  D0,-(A7)
        move.l  #$1000,D0
        bra     dloop

ldelay  ; long ~1s delay
        move.l  D0,-(A7)
        move.l  #$70000,D0
        bra     dloop

pstring ; prints string pointed to by A6 then newline
        ; D0 is corrupted by call, so preserve it
        move.l  D0,-(A7)
        trap    #11
        dc.w    5           ; pdata1
        trap    #11
        dc.w    12          ; crlf
        move.l  (A7)+,D0
        rts

line     dc.b    "score "
scores   dc.b    "-- -- -- -- -- -- -- -- ",0
blankscore
         dc.b    "-- -- -- -- -- -- -- -- ",0

final    dc.b    "That is the final score.  "
         dc.b    "New game started.",0
welcome  dc.b    "Welcome to 8-pin bowling!  "
         dc.b    "Enjoy your game.",0

throw1   dc.b    "Throw your first ball!",0
throw2   dc.b    "Throw your second ball!",0

progress dc.b    "You knocked over "
prognum  dc.b    "0 pins",0

         ; standing is which pins (bits) are still
         ; standing up
standing dc.b    %00000000  ; 1 for standing
frame    dc.b    0          ; current frame number
         end
```