Real Time Embedded Systems
Worksheet 1: Assembly Language Familiarisation Exercises

The processor we are using in the lab is the Freescale Coldfire+, which is a derivative of the 68000. The simulator can be d[illegible].easy68k.com. It is more advanced than the generic device used to introduc[illegible]ture. Its main features are summarised below.

Wordlength

The simple processor u[illegible]rried out operations on 8-bit data values. By default, the Coldfire processor work[illegible]lthough it can be set to use 8 or 32-bit values if desired. For now, we will work a[illegible]any data value occupies two consecutive byte locations in memory. For example, an item of data at address 3000 (hex) actually occupies addresses 3000H and 3001H.

Specification of hexadecimal values

By default the assembler for this processor assumes that all numbers are in decimal. To specify a hexadecimal number, it must be prefixed with '$'. E.g. 4000H would be written $4000.

Registers

This processor has 8 registers, called 'data registers', numbered D0 .. D7, e.g.

add	$1000,d1	Add the 16-bit value in memory location 1000H - 1001H to D1, leaving the result in D1.

Operand locations

An instruction can act on values held in:

a register and a memory location, e.g.

sub	$1000,d0	Subtracts the 16-bit value in memory location 1000H - 1001H from the value in D0, leaving the result in D0.

two registers, e.g.

move	d2,d3	Moves the 16-bit value in D2 to D3.

a constant value and a register: the constant is denoted by '#', e.g.

move	#2,d4	Moves the value 2, ***not*** the value held in memory location 2, to D4.
add	#$000A,d5	Adds the hexadecimal value 0AH (decimal 10) to D5.

The result should always be returned to a register, so the right-hand operand is always D0 .. D7. The only exception is with a move instruction, e.g.

move	d1,$2000	Moves the 16-bit value in D1 to memory location 2000H - 2001H.

Flags

A zero flag is set true if an instruction returns a zero result, or false if not. There are a few other flags that will be introduced as they are needed.

Instructions

The most commonly used instructions are listed here, with examples of their use.

	move	$2000,d0	[illegible] tents of a 16-bit value from memory locations 2000H - [illegible] a register D0. Sets zero flag true if the value moved is [illegible] n sets it false.
	add	$200A,d1	[illegible] t value from locations 200AH - 200BH into D1. Sets the [illegible] e if the result of the addition is zero, otherwise sets it [illegible]
	sub	d2,d3	Subtracts the 16-bit value in D2 from the value in D3, leaving the result in D3. Sets zero flag true if the result is zero, or false if not.
	beq	*name*	'Branch if equal': Goes to another instruction if the zero flag was set true by the previous instruction. The instruction branched to is identified by the name given in the instruction. If the zero flag was left at false by the previous instruction, then the instruction does nothing and control passes to the next instruction in sequence.
	bne	*name*	'Branch if not equal': Behaves as BEQ except that the logic is reversed. It branches if the zero flag was *not* true, that is, if it was false, and does nothing if it *was* true.
	bra	*name*	Branches to the named instruction unconditionally.

There are two instructions for defining storage.

name	ds	1	Defines memory for one 16-bit value and gives it the name specified. The assembler will allocate the actual memory location. E.g.
x	ds	1	Define storage for a 16-bit value, and call it x.
name	dc	*value*	Define 16-bit constant, and give it the name and value shown. E.g.
k6	dc	6	Define a 16-bit constant with the value 6, and call it k6.

Other instructions will be introduced in the questions as they are needed.

Note, in all cases, that instructions should be laid out in such a way that column 1 at the far left is only used for the name of an instruction or data item.

Address Registers

There are also 8 address registers, numbered A0 .. A7. These allow the address on which an instruction acts to be computed and changed during the execution of the programme, instead of being fixed in the instruction code. E.g.

move	$1000,a0	Loads address 1000H into A0
move	(a0), d0	Moves 16 bits from memory location addressed by A0 (1000H) to D0
add	#2,a0	Adds 2 to A0, which becomes 1002H
move	(a0),d1	Moves 16 bits from location 1002H to D1

The address registers may also be used in offset addressing, e.g. to access different elements in a data structure. Suppose that there is an array of data records. The array is called *recary*, and each record within it is called *rec*. Each record contains three elements: *item1*, *item 2* and *item3*. Each element is 2 bytes long, so each record is 6 bytes.

recary					
rec [0]			*rec [1]*		
item 1	*item 2*		*item 1*	*item 2*	*item 3*
0	2		0	2	4

Storage is defined for 8

recary ds 24 Define storage for 24 (8 x 3) 16-bit values

The record itself is defined as follows, using 'equate' directives that assign the value to the name. (There are better ways to do this, but this is the most straightforward).


```
rec     equ   0   record
item1   equ   0   item 1 is located at the start of the record (i.e. zero bytes from it)
item2   equ   2   item 2 is located 2 bytes from the start
item3   equ   4   item 3 is located 4 bytes from the start
reclen  equ   6   the length of the record is 6 bytes
```


We place the address of the array in A0. A0 now 'points' to the first record, *rec [0]*.

move #recary,a0 move the value *recary* (i.e. the address of *recary*) to A0

recary					
rec [0]			*rec [1]*		
item 1	*item 2*	*item 3*	*item 1*	*item 2*	*item 3*
0	2	4	0	2	4

A0

Elements within it may then be addressed as follows.

```
move  item1(a0),d0   move item1 (in the memory location at 0 bytes from A0) into D0
move  item2(a0),d1   move item2 (2 bytes from A0) into D1
move  item3(a0),d2   move item3 (4 bytes from A0) into D2
```

If we now increase the value in A0 by the record length *reclen*, which was defined as 6, A0 will now contain the address of the next record, *rec [1]*. The three instructions above will then access data from the second record in the array.

```
add   #reclen, a0
```

recary					
rec [0]			*rec [1]*		
item 1	*item 2*	*item 3*	*item 1*	*item 2*	*item 3*
0	2	4	0	2	4

A0

Practical Work

You will now need to familiarise yourself with the 'integrated development environment', which includes an editor, assembler and simulator, and is accessed from the EECE program menu.

To start you off, questio [illegible]

1.
Using two MOVE instru [illegible] tents of one 16-bit variable from address 2000H to 2002H.

Here is the answer. Not [illegible] and 'SIMHALT' macro, which are needed in all the questions. Remember to type it so that each line is indented from column 1.

```
        org         $1000          ;'Origin': puts the program at location 1000H in memory
        move        $2000,d0       ;Moves 16-bit value from memory location 2000H to D0
        move        d0,$2002       ;Moves same value back from D0 to location 2002H
        SIMHALT                    ;Ends the program and returns to the operating system
```


Test the program as follows. Insert a 16-bit value into location 2000H. Recall that a 16-bit number is represented by *four* hexadecimal digits, so the value will actually occupy locations 2000H and 2001H. Now run the program, and check to see that the same value has now been copied to location 2002H (i.e. 2002H and 2003H). Single-step through the program, observing the changing values in data register 0 and memory location 2002H.

Now reset the program, and place a 16-bit value of 0000H into location 2000H (that is, locations 2000H - 2001H). Single-step through it again, and observe that the zero flag, labelled 'Z', goes to 1 (true) after the first move is executed because the value moved to D0 is indeed zero. Reset the program again, and place a non-zero value in 2000H. Single stepping again, the zero flag should this time stay at 0 (false).

2.
Add the 16-bit values at location 2000H and 2002H, and place the result at location 2004H. Both values should be positive. (How do you recognise a positive number expressed in hexadecimal?) Again, test the program by inserting values into 2000H and 2002H, and single-stepping through it.

Now make one of the numbers negative, and test the program again. Check the result carefully.

3.
Three 16-bit variables are located at 2000H, 2002H, and 2004H. Write a programme that compares 2000H and 2002H, then sets 2004H to 1 if the first two values were equal, or 0 if they were not. Initialise 2002H and 2004H, run the programme, and check the result in 2004H.

As this is a conditional program, you will need to use the conditional branch instruction described in the lecture. Remember that you will need to identify the point that you want the program to branch to by giving it a name, (which must be typed so as to start in column 1).

4.
Repeat question 1, this time putting a negative value in location 2000H, and single step through it. Apart from the zero flag, there is also a *negative flag*, labelled 'N'. Observe how this is set when an instruction produces a negative result.

5.
After each instruction, the negative flag is set true if the instruction returned a negative result, and false if the result was positive. A result of zero is regarded as positive (can you explain why?).

There are two more conditional branch instructions that test this flag:

bmi	*name*	[illegible]s' which branches if the N flag is 1 (true)
bpl	*name*	[illegible] which branches if the N flag is 0 (false)

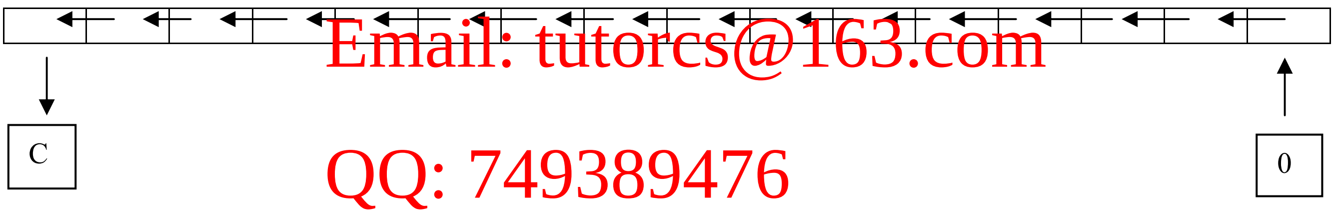
Determine which is the [illegible] 16-bit values stored at locations 2000H and 2002H. Store the larger value a[illegible]

6.
Count the number of 1's in a 16-bit word held at location 2000H. Store the result at 2004H. There are different ways to do this, but one method is to use the instruction 'logical shift left':


```
lsl     #1,d0
```

This moves every bit in D0 one place to the left, as illustrated here:

D0



The empty bit at the right hand side (the least significant bit, or LSB) is set to 0. The bit that gets pushed out at the left hand side (most significant bit, MSB) is moved into the *carry flag*, labelled 'C' in the simulator. You will then need one of the following:

bcs	*name*	'Branch if carry set' which branches if the C flag is true
bcc	*name*	'Branch if carry clear' which branches if the C flag is false

7.
If a register is shifted left by one place, as above, what happens to the binary value that it contains? What about shifting right? If a 16-bit variable *x* is located at 2000H, write a program that places *x*/2 at 2002H and *x*/4 at 2004H. Does this work if *x* might be negative? Look up the 'arithmetic shift right' (ASR) instruction.

8.
An array of 8 values, each 16 bits, is held at location2000H. Using address registers, copy it to location 2010H.

9.
A linked list is a data structure consisting of an array of records. Each record contains one or more data items, and a pointer to the next item. For example, a list might contain several 4-byte records, each of which holds a 2-byte integer data value and a 2-byte pointer, as shown here, with the last pointer being set to zero. The list starts at address 2000H.

rec [0]		rec [1]		[illegible]		rec [3]		rec [4]		rec [5]	
val	2004	val	[illegible]	[illegible]	200C	val	2010	val	2014	val	0000
2000	2002	2004	[illegible]	[illegible]	200A	200C	200E	2010	2012	2014	2016

Sort the list so that the v[illegible]ing order. Do not move the location of each record, but instead, adjust the poin[illegible] the new address of the start of the list.

For example, if the initial state of the list is as follows,

rec [0]		rec [1]		rec [2]		rec [3]		rec [4]		rec [5]	
4	2004	7	2008	3	200C	1	2010	9	2014	2	0000
2000	2002	2004	2006	2008	200A	200C	200E	2010	2012	2014	2016

then the sorted result should be as follows.

New start of list is at 200CH.

rec [0]		rec [1]		rec [2]		rec [3]		rec [4]		rec [5]	
4	2004	7	2010	3	2000	1	2014	9	0000	2	2008
2000	2002	2004	2006	2008	200A	200C	200E	2010	2012	2014	2016

Now write an additional programme that takes the sorted list and its start address, and transfers each value to a new list consisting of the 2-byte values in their sorted order.

Answers

2.

```
        org         $1000          ;'Origin': puts the program at location 1000H in memory
        move        $2000,d0       ;M[illegible] 16-bit value from memory location 2000H to D0
        add         $[illegible]   [illegible]16-bit value from 2002H into D0, producing sum
        move        d[illegible]   [illegible] back from D0 to location 2004H
        SIMHALT                    [illegible]he program and returns to the operating system
```

3.

```
        org         $[illegible]
        move        $[illegible]   [illegible]2000H to D0
        sub         $2002,d0       ;subtract 2002H from D0, Z becomes true if values equal
        beq         equal          ;branch if Z true to 'equal'
        move        #0,d0          ;move value 0 to D0
        bra         end            ;branch to 'end'
equal   move        #1,d0          ;move value 1 to D0
end     move        d0,$2004       ;move D0 to 2004H
        SIMHALT
```


Alternatively, you could do this by defining constant values 0 and 1:

```
        org         $1000
        move        $2000,d0       ;move 2000H to D0
        sub         $2002,d0       ;subtract 2002H from D0, Z becomes true if values equal
        beq         equal          ;branch if Z true to 'equal'
        move        k0,d0          ;move value 0 to D0
        bra         end            ;branch to 'end'
equal   move        k1,d0          ;move value 1 to D0
end     move        d0,$2004       ;move D0 to 2004H
        SIMHALT

k0      dc      0                  ;set up constant with value 0
k1      dc      1                  ;set up constant with value 1
```


5.

```
        org         $1000
        move        $2000,d0       ;move 2000H (1st value) to D0
        sub         $2002,d0       ;subtract 2002H (2nd value) from D0, N set true if 2nd > 1st
        bmi         sec            ;branch if N true (2nd > 1st) to 'sec'
        move        $2000,d0       ;1st value greater, so move it to D0
        bra         end            ;branch to 'end'
sec     move        $2002,d0       ;second greater, so move it to D0
end     move        d0,$2004       ;move D0 to 2004H
        SIMHALT
```

6.

```
        org         $1000
        move        $2000,d0        ;move value under test to D0
        move        #0,d1           ;bit count in D1, initialised to 0
        move        #16,d2          ;loop counter in D2, initialised to 16
loop    lsl         #[illegible]    [illegible]0 left one place
        bcc         n[illegible]    [illegible]ifted out not = 1, branch to 'not1'
        add         #[illegible]    [illegible]d bit = 1, so increment D1
not1    sub         #[illegible]    [illegible]ment loop counter D2
        bne         lo[illegible]   [illegible] if not zero to loop
        move        d[illegible]    [illegible]bit count (D1) at 2004H
        SIMHALT
```

7.

If a binary value is shifted left one place, it is multiplied by 2, and divided by 2 if shifted right. However, if the value is signed, then this may not work because the most significant bit (the far left-hand bit), which represents the sign, will have the neighbouring value shifted into it and may therefore change. This would be incorrect because doubling or halving a value ought not to change its sign. Therefore use the arithmetic shift instruction, which maintains the sign bit whilst shifting all the other bits.

```
        org         $1000
        move        $2000,d0        ;Move 16-bit value from memory location 2000H to D0
        asr         #1,d0           ;Shift right, halving value
        move        d0,$2002        ;Move  back from D0 to location 2004H
        asr         #1,d0           ;Shift right, halving value again
        move        d0,$2002        ;Move  back from D0 to location 2004H
        SIMHALT
```

8.

```
        org         $1000
        move        #$2000,a0       ;Initialise A0 to point to 2000H
        move        #$2010,a1       ;  and A1 to 2010H
        move        #8,d0           ;Initialise loop counter
                                    ;Repeat
loop    move        (a0), d1        ;  Move from (A0) to (A1)
        move        d1,(a1)
        add         #2,a0           ;  Increment address registers
        add         #2,a1
        sub         #1,d0           ;  Decrement loop counter
        bne         loop            ;Until loop counter = 0
```