538 Lecture Notes Week 3

Answers to last week's questions

1 Write code so that the least significant bit of Accumulator A is cleared, the most significant set and the other 6 bits inverted. For example, 0x7f would become 0x80 or 0b01011100 would become 0b10100010.

ANSWER:

```
            oraa #$81  ;set lsb and msb to 1
            eora #$7f  ;invert other bits and lsb
```

2 Convert the following pseudo-C into assembler:

```
byte bytes[] = {2, 7, 1, 8, 3, 1, 4, 8};
int i = 0
int sum = 0
byte * bp = bytes; //same as bp = &bytes[0]
while(bp <= bytes+7) {
      sum = sum + (*bp)*2;
      bp = bp + 2;
}
```

ANSWER:

- We answer this by writing a subroutine.
- *Notes:*
  - The value calculated is two times the sum of the array elements with even indices.
  - The multiplication by 2 is done with a left shift which is more efficient than an explicit multiply.
  - An optimizing compiler (or a better programmer) would have added up the array elements and then multiplied the total by 2 instead of performing the multiplication on each element.

```
bytes       dc.b 2, 7, 1, 8, 3, 1, 4, 8

Start:
           jsr AddTwiceArray  ;expect A = 0x14 (20 decimal)
           swi

AddTwiceArray:
            clra ;use AccA for sum
            ldx #bytes
loop_while_1:
            cpx #bytes+7
            bhi done_while_1
            ldab 2,x+
            lslb  ;multiply by 2
            aba   ;add B to A
            bra loop_while_1
 done_while_1:
            rts
```

3 Assuming that the NZV and C bits are all zero, show how they evolve with each instruction:

```
          org $3000
          ldaa #0f
          oraa $3000 ;Note: some disassembly required!
```

ANSWER:

Instruction	Acc A (hex)	N Z V C
Initial values	?	0 0 0 0
ldaa #$0f	0f	0 0 0 0
ora $3000	8f	1 0 0 0

4 Translate the following pseudo-C to assembler.

```
unsigned byte p, q;

while(p >= 128) {
  if(q < 128) {
    p--;
  } else {
    p = p + 2;
```

```
    }
}
```

ANSWER:

```
pq:         ;while p>= 128
            tst p
            blo endWhilepq
            ;if q  < 128
            tst q
            bhs elsepq
            dec p
            bra pq
elsepq:
            ldaa p
            adda #2
            staa p
            bra pq
endWhilepq:
```

5 Translate the following C code into assembler:

```
byte p, q, w;
w = p + q + w;
if (w == 0)
      w++;
```

ANSWER:

```
            ; byte p, q, w;
p ds.b 1
q ds 1
w ds 1

;           w = p + q + w;
            ldaa p
            adda q
            adda w
            staa w
            ;if (w == 0)
               ;w++;
            bne doneq5
            inc w
doneq5: ....
```

Increment/Decrement Instructions

- Many registers can be incremented or decremented by 1 with a single instruction.
- For example: `inx, iny, dex, dey, inca, incb, ins, des`
- Memory locations can also be incremented/decremented directly:

```
inc a,x  ;increment memory location A+X
```

Multiplication/Division

- Unlike addition/subtraction, different hardware is needed to multiply (divide) signed and unsigned numbers.
- To multiply, use:
  - `mul ;8bit unsigned multiply A*B => D`
  - `emul ;16-bit unsigned multiply D*Y => D:Y (32 bit result)`
  - `emuls ;16-bit signed multiply D*Y => D:Y`
- To divide, use:
  - `ediv ;32-bit by 16-bit unsigned Y:D/X => Y (quot), D (rem)`
  - `edivs;32-bit by 16-bit signed Y:D/X => Y (quot), D (rem)`
  - `fdiv ;16 by 16 bit unsigned fractional divide`
  - `idiv ;16 by 16-bit unsigned: D/X => X (quot), D (rem)`
  - `idivs;16 by 16-bit signed: D/X => X (quot), D (rem)`

Logical and shift/rotate operations

AND, OR, EXCLUSIVE OR, NOT

- The bitwise logical OR, AND and EXCLUSIVE OR, NOT are performed with the `and, or, eor, com` instructions. For example:

```
 ldaa #%01011100 ;load A with 0b01011100 = 0x5c
anda #$f1   ;A = 0b01011100 AND 0b11110001 = 0b01010000
ora #%00010010 ;A = 0b01010000 OR 0b00010010 = 0b01010011
eroa #$f0       ;A now 0b10100011
```

```
        coma            ;A now 0b01011100
```

Setting/Clearing Bits

- Bits in memory can be set with the *Bit Set* (`bset`) instruction. For example:

```
   movb #$55,$1000  ;Location 0x1000 now 0b01010101
bset $1000,%00001010 ;Location $1000 now 0x5f
```

- Similarly, selected bits can be cleared with the *Bit Clear* (`bclr`) instruction.
- Bits in the CCR can be set or cleared with the `orcc` and `orcc` instructions. The most common patterns have alias assembly language instructions. For example:

```
orcc #1  ;Sets least significant (C) bit of CCR
         ;"sec" is identical
andcc #$fe ;Clears C bit of CCR; "clc" is identical
```

Shifting/Rotating

- A bit pattern can be shifted left or right by one position.
- The bit shifted out is put into the C bit of the CCR.
- The bit shifted in for a left shift is always a zero.
- The bit shifted in for a right shift is either:
  - A Zero for a *logical* shift.
  - The sign bit for an *arithmetic* shift.
- A left or right rotate instruction a 9-bit quantity with the C bit as the ninth bit.
- Examples:

```
     ldaa #$10
     lsla  ; shift A left: $20=>A; 0 => C
     rola  ;rotate C-A 1 bit left: $41 => A; 0 => C
;The following combination shifts D left by 1 bit:
     lslb
     rola
```

If..else.., Loops, Conditional branches

- The general rule for translating an *if (condition) then...else... endif* is:
    - Label the first instruction of the else clause (eg: else: and the first statement following the else clause.
    - Branch if the condition is *false* to the else label.
    - Encode the then block and end it with an unconditional branch around the "else" block.
    - For example, to translate:

```
unsigned byte p, q, r;
if (p == 0) {
    p++;
} else {
    q++;
}
r = p + q;
```

    - The assembly language equivalent is:

```
          p  ds 1
          q  ds 1
          r  ds 1

          ldaa p
          bne else  ;branch on the opposite of ==
                 ;then clause continues here
          inc p;
          bra endif  ;branch around the else block
else:     inc q
endif:    ldaa p
          adda q
          staa r
```

The Compare Instructions

- It is very common for the condition controlling a loop or if statement to require comparing two numbers.
- If we subtract two numbers the result can be zero, positive or negative and hence we we can determine if the first number is smaller, bigger or the same as the second.
- The *compare* instruction performs this subtraction but discards the result but it does set the condition code bits so that the relative sizes of the two numbers can be determined.
- The compare instructions include `cmpa, cmpb, cpd, cpx, cpy,cps.`
- Often you want to compare against zero to determine if something is positive, negative or zero.

This can be done efficiently with a *Test* instruction (`tsta, tstb, tst`).

- It is essential that the programmer know whether the numbers are to be interpreted as signed or unsigned as different conditional branches are required as summarized in the table below:

Math	Signed	English	Unsigned	English
==	beq	Branch if Equal	beq	Branch if Equal
!=	bne	Branch if Not Equal	bne	Branch if Not Equal
>	bgt	Branch if Greater	bhi	Branch if Higher
>=	bge	Branch if Greater or Equal	bhs	Branch if Higher or Same
<	blt	Branch if Less Than	blo	Branch if Lower Than
<=	ble	Branch if Less or Equal	bls	Branch if Lower or Same

- Some examples of using compare instructions and the proper conditional branches are given below:

Pseudo C	Assembler

`signed byte p, q;`
`unsigned byte w, z;`
 	
`while (p > q) {`
`  if (w <= z) {`
`  w++;`
`  }`
`  p--;`
`}`
	`P ds 1`
	`q ds 1`
	`w ds 1`
	`z ds 1`
	
	`loop:`
	`  ldaa p`
	`  cmpa q`
	`  ble endif2`
	`  ldaa w`
	`  cmpa z`
	`  bhi endif1`
	`  inc w`
	`endif1: dec p`
	`  bra loop`
	`endif2:`

`#define N 10`
`signed byte ar[N];`
`byte * ar_ptr = &ar[0];`
`signed byte total = 0;`
 	
`while(ar_ptr < ar + N) {`
	`N  equ 10`
	`ar  ds  N`
	`total ds 1`
	
	`  clra`
	`  ldx #ar`

```
    total += *ar_ptr++;
// total += ar[i]; //*(ar+i)
}
```

```
loop:
        cpx #ar+N
        bhs done
        adda 0,x
        bvs oops
        inx
        bra loop

done:   sta total
     ;program continues here

oops:  ;handle overflow
```

```
#define N 10
signed byte ar[N];
byte * ar_ptr = &ar[0];
signed int16 total = 0;

while(ar_ptr < ar + N) {
    total += *ar_ptr++;
}
```

```
N equ 10
signed byte ar[N];
byte * ar_ptr = ar;
total ds 2
       ldx #ar
       ldd #0
loop:
       cpx ar+N
       bhs done
       addb 1,x+
       adca #0
       bra loop
done:  std total
```

How to interpret the Instruction Set Reference

LDAA #*opr8i*
LDAA *opr8a*
LDAA *opr16a*
LDAA *oprx0_xysp*
LDAA *oprx9,xysp*
LDAA *oprx16,xysp*
LDAA [D,xysp]
LDAA [*oprx16,xysp*]
	(M) ⇒ A
	Load Accumulator A
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			86 ii
			96 dd
			B6 hh ll
			A6 xb
			A6 xb ff
			A6 xb ee ff
			A6 xb
			A6 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					----	ΔΔ0-

Figure 1: Instruction Set Reference for ldaa instruction

- The Instruction Set Reference (Appendix A of the text and available here) contains a wealth of information presented in a compact fashion. (You will be provided a copy of this during the midterm and final; it is imperative that you learn how to use this reference.)
- The figure above shows the entry for the `ldaa` instruction. The first column gives the assembly

language syntax for its use in 8 different addressing modes. The Machine Coding column (4th column) gives the machine language in hexadecimal.

- The Access Detail column indicates how the bus is used. For our purposes, the number ofletters is all that matters for now: this is the number of bus cycles required to fetch and execute the instruction. For example, in immediate addressing mode, 1 bus cycle is required while 3 bus cycles are required in the extended addressing mode.
- The last column (NZVC) indicates if and how these 4 CCR bits are affected by the instruction.
- In the case of `ldaa`, the N bit is set to 0 or 1 depending on the value loaded. Similarly, the Z bit is set to 1 if zero is loaded and set to 0 otherwise. The V bit is cleared to 0 unconditionally and the C bit is unaffected.

Subroutines

- A subroutine (like a function in C) performs a clearly defined computational task.
- The subroutine should be clearly described in English so that a programmer knows how to use it and what it does. A programmer using a subroutine should not have to look at the source code to see what it does or how it is implemented.
- A subroutine is easier and safer to use if it has no *side effects* unrelated to the subroutine's function.
- An avoidable side-effect is a change to register values. To avoid this, push any registers that are modified at the beginning of the subroutine and restore them (pulling them back from the stack) before returning.

Example: Cube (unsigned) subroutine

- Suppose we wish to write a subroutine to calculate n^3 . Simple, eh? After all, in C it would be one line: `return n*n*n;`
- Additional details are required. Let's suppose:
  - *n* is unsigned;
  - *n* is 8 bits;
  - The cube of an 8-bit unsigned number could require 24 bits for the result;
  - Let's assume (arbitrarily): we are only interested in results that fit within 16 bits.
- The largest integer whose cube fits into 16 bits is 40 ($40^3 = 64{,}000$)
- Before trying to implement the subroutine, we give it a name and write documentation which

should include details about how "n" is passed to the subrutine and how to retrieve the result:

```
;ucube8to16
;
;Calculate n^3 where n is an 8-bit unsigned nummber <= 40
;If the result does not fit in 16 bits, 0xffff is returned
;and the Carry bit is set.
;Otherwise the result is returned in Acc D
;and the Carry bit is cleared.
;
;Side effects:  None
;
;Parameters:
;AccA : n, the value to cube
;
;Return:
;n^3 in AccD
```

- Writing the documentation first clarifies what the programmer wishes to achieve.
- The name chosen for the subroutine – `ucube8to15` – is meant to suggest that it calculates the cube of an 8 bit number and returns a 16 bit result.
- (One can imagine other subroutine names such as `scube8to32, ucube8to32`, etc.)
- Note that the documentation indicates no side effects. Consequently the programmer should ensure that any registers (except for D and CCR) have unchanged values when the subroutine returns.
- Once the documentation is done, we can write test code. It is often a good idea to write test code even before you write the implementation. This helps you ensure that the documenation is clear and complete.
- Here is the test code:

```
            ldaa #5
            jsr ucube8to16  ;D should be 125 0x00fd
            bcs oops
            cpd #5*5*5
            bne oops
            ldaa #45
            jsr ucube8to16  ;D should be 0xffff
            bcc oops
            cpd #$ffff
            bne oops
            clra
            jsr ucube8to16  ;D should be 0
            bcs oops
            cpd #0
```

```
            bne oops
done:       bra done
oops:       bra oops
```

We can now look at one possible implementation:

```
ucube8to16:
            cmpa #40
            bls ucube_continue
            ldd #$ffff
            sec    ;set the Carry bit
            rts

ucube_continue:  ;n^3 fits, calculate it
            pshy      ;Save Y since we use it below
            psha
            tfr a,b
            mul       ;D now n^2
            tfr d,y   ;Y now n^2
            ldab 0,sp
            clra      ;D now n
            emul      ;n^3 now in Y:D
                      ;but we know that Y is zero
            ins       ;Increment SP so it points
                      ;to pushed Y value
            puly      ;Restore Y to its value on entry
            clc       ;Clear the carry bit.
            rts
```

An Alternative Implementation

- Using the identical documentation (or application programmer's interface – API) we could have written a *table lookup* version as follows:

```
cubeTable: fdb 0; cube of 0
           fdb 1; cube of 1
           fdb 8; cube of 2
           fdb 27;cube of 3
           ;....other cubes
           fdb 64000 ;cube of 40

ucube8to16:
```

```
            cmpa #40
            bls continue
            ldd #$ffff
            sec
            rts

 continue:
            pshx
            ldx #cubeTable
            asla
            ldd a,x
            pulx
            clc
            rts
```

- This implementation would be significantly faster but would require extra memory (82 bytes) for the table of cubes.

A Strings Library

Characters and Strings

- Characters are encoded as binary bit patterns.
- The most common encodings are ASCII (American Standard Code for Information Interchange), a 7-bit code, and Unicode, a 16-bit code. (The first 128 Unicode characters are identical to ASCII).
- ASCII is limited to the English alphabet; Unicode can be used for many languages and symbols. (For example, the characters π, ש, א, or € can be encoded in Unicode but not ASCII.)
- Nonetheless, we use ASCII in this course and hence a byte is sufficient to encode a single character.
- A memory dump also displays the character encoded *for those bytes that correspond to a printable ASCII character*. Note, however, that most bytes do *not* encode a printable character. These are indicated as a '.' (dot). For example,

```
4000 30 20 33 41 40 42 80 00 43 64       0 3A@B..Cd
```

- The character column shows that 0x33 is the ascii code for the character '3', 0x64 is the code for 'd', etc.
- A *string* is a set of characters stored in sequential memory locations.
- The end of the string is signalled using the C convention with the non-printable NULL character

(0x00)

- Thus the number of bytes required for a string is one more than the number of characters. (The extra byte is needed for the *null terminator*.
- The following memory dump describes 2 strings:

```
4000  41 42 00 63 64 65 00 00        AB.cde..
```

- The string "AB" starts at location 0x1000, the string "cde" starts at 0x1003 and the *empty string ""* starts (and ends) at 0x1007
- These strings could be specified in assembler with the following directives:

```
      org $4000
Abstr fcc 'AB'
      fcb 0   ;null terminator
cde   fcc 'cde'
      fcb 0
empty fcb 0
```

- We now document and write a set of general purpose string subroutines similar to those found in the standard C string library. The subroutines are:
  - `strlenASM`: Determine the length of a string (number of characters not including the null terminator). Similar to C's *strlen* function.
  - `strcpyASM`: Copy a string elsewhere. Similar to C's *strcpy* function.
  - `strcatASM`: Concatenate one string to another.
  - `strrevASM`: Reverse a string.
  - `strprtASM`: Print a string.

- Here are the subroutines:

```
;Name: strlenASM
 ;Description:  Determines the number of characters in
 ;              a null-terminated string
 ;Side effects: CCR modified
 ;Parameters: X - Address of string
 ;Returns: Acc D - number of characters in the string

strlenASM:
            pshx
loopStrLen:
            tst 1,x+
            bne loopStrLen
            dex
            tfr x,d
```

```
            subd 0,sp
            pulx
            rts

 ;Name: strrevASM
 ;Description:  Reverses a null-terminated string.
 ;Side effects: CCR modified
 ;Parameters: X - Address of string to reverse
 ;            Y - Address of where to place reversed string.
 ;Note: X and Y can be the same.
 ;Returns: Nothing

strrevASM:
            psha
            pshb
            pshx
            pshy

            clrb  ;B maintains num of chars in string
pushLoopStrrev:    ;Push the chars onto the Stack
            incb
            ldaa 1,x+
            psha
            bne pushLoopStrrev
            pula
            decb

popLoopStrrev:
               ;Pop the chars off the stack
               ;They are popped in the reverse order
               ;they were pushed.
            pula
            staa 1,y+
            decb
            bne popLoopStrrev
            clr 0,y ;Add null terminator
            ;Restore registers
            pula
            puly
            pulx
            pulb
            pula
            rts

 ;Name: strcpyASM
```

```
 ;Description:  Copies a null-terminated string.
 ;Side effects: CCR modified
 ;Parameters: X - Address of string to copy
 ;            Y - Address of where to place string copy.
 ;Note: X and Y cannot overlap
 ;Returns: Nothing

strcpyASM:
           psha
           pshx
           pshy
loopStrcpy:
           ldaa 1,x+
           staa 1,y+
           bne loopStrcpy
           puly
           pulx
           pula
           rts

 ;Name: strcatASM
 ;Description:  Concatenates one string to another
 ;Side effects: CCR modified
 ;Parameters: X - Address of string concatenated to.
 ;            Y - Address of string to concatenate to first.
 ;Note: If X == Y, the string will be doubled.
 ;Returns: Nothing
strcatASM:
           psha
           pshx
           pshy
             ;Move to end of first string
loopStrCat1:
           tst 1,x+
           bne loopStrCat1
           dex

           exg x,y
           jsr strcpyASM

           puly
           pulx
           pula
           rts
```

```
   ;Name: strprtASM
   ;Description:  Prints a null-terminated string.
   ;Side effects: CCR modified
   ;Parameters: X - Address of string to print.
   ;            Y - Address of subroutine to output a char.
   ;                This subroutine must print the char
   ;                in AccA.
   ;Returns: Nothing

   strprtASM:
              psha
              pshx

   loopStrprt:
             ldaa 1,x+
             beq doneStrprt
             jsr 0,y
             bra loopStrprt

  doneStrprt:
              pulx
              pula
              rts
```

- The following code is used to test them:

```
              ORG RAMStart
   ; Insert here your data definition.
  stringLen1 fcc 'A'
              dc.b 0

  stringLen0 dc.b 0

  stringLen5 fcc 'Hello'
             dc.b 0

  freeSpace  ds 256

  stringLen550 fcc
'0123456789012345678901234567890123456789012345678901234567890123456789'
               fcc
'0123456789012345678901234567890123456789012345678901234567890123456789'
               fcc
'0123456789012345678901234567890123456789012345678901234567890123456789'
               fcc
```

```
'0123456789012345678901234567890123456789012345678901234567890123456789'
                fcc
'0123456789012345678901234567890123456789012345678901234567890123456789'
                fcc
'0123456789012345678901234567890123456789012345678901234567890123456789'
                fcc
'0123456789012345678901234567890123456789012345678901234567890123456789'
                fcc
'0123456789012345678901234567890123456789012345678901234567890123456789'
                fcc
'0123456789012345678901234567890123456789012345678901234567890123456789'
                fcc
'0123456789012345678901234567890123456789012345678901234567890123456789'
                dc.b 0  ;null terminator



  ; code section
                ORG   ROMStart


  Entry:
  _Startup:
                LDS   #RAMEnd+1       ; initialize the stack pointer

  ; Test strlenASM subroutine
                ldx #stringLen1
                jsr strlenASM  ;Acc D should be 0x0001 upon return

                ldx #stringLen0
                jsr strlenASM  ;Acc D should be 0x0000 upon return

                ldx #stringLen5
                jsr strlenASM  ;Acc D should be 0x0005 upon return

                ldx #stringLen550
                jsr strlenASM  ;Acc D should be 0x0226 (= 550
decimal) upon return


  ; Test strcpyASM subroutine
                ;Copy "Hello"
                ldx #stringLen5
```

```
            ldy #freeSpace
            jsr strcpyASM

; Test strcatASM subroutine
            ldx #freeSpace
            clr 0,x
            ldy #stringLen1
            jsr strcatASM
            jsr strcatASM
            ldy #stringLen5
            jsr strcatASM

 ; Test strrevASM subroutine
            ;Test in-place reverse
            ldx #stringLen5  ;"Hello"
            tfr x,y
            jsr strrevASM   ;X should point to "olleH"



            swi
```

Parallel Ports

Memory Mapped Devices

- The Address, Data and Control buses interconnect the CPU with memory (both volatile—RAM—and non-volatile—EEPROM).
- Peripheral Devices can also be connected to these buses.
- These devices have internal registers that then become memory mapped.
- For example, if a device with 4 internal registers--regA, regB, regC and regD—is memory mapped to 0x8100—0x8103—then its regA would be mapped to 0x8100, regB to 0x8101, etc.
- Device registers typically include one or more data registers as well as registers for *control* (to specify optional actions of the device) or *status* (to monitor the device's state).
- Typical devices include:
    - Parallel ports.
    - Serial communication ports connected to external signals.

- Timer chips to provide precise timing without the need for software delays and to generate or measure external waveforms.
- Graphics processors (but not in the hcs12)
- Ethernet or WiFi controllers (but not in hcs12)

hcs12 Parallel Ports

- There are several general purpose input/output paralell ports in the hcs12.
- There are some differences in various models of the hcs12.
- The most commonly used ports are named A, B, E, K, H,J,M P, S and T.
- These ports are 8-bits wide (although Ports A and B can be combined into a single 16-bit port).
- Each bit in the port corresponds to a pin on the hcs12 chip and can be configured as either an input or output bit.
- The direction of each bit is controlled (and set by the programmer) by the corresponding *Data Direction Register* (DDR). A '1' in the DDR programs that bit as output while a '0' configures it as input.
- Suppose we have 4 switches connected to the most significant bits of Port H and 4 LEDs connected to the 4 least significant bits. We configure this as follows:

```
ldaa #$0f
staa DDRH
```

- We could then, for example, read the switches as a 4-bit binary number, add 1 to that number and display the result in the LEDs as follows:

```
ldaa PTH  ;Upper 4 bits are switch values
lsra      ;Shift them down to lower 4 bits
lsra
lsra
lsra
inca      ;add 1 to the switch value
staa PTH  ;Light up the corresponding LEDs
```

- But what are the addresses of the Port H data and direction registers?
- The Port H data register is mapped to 0x0260 and its direction register is mapped to 0x0262.
- Port H is used on the lab board to connect to 8 LEDs (the LED bar).
- Information about Port H can be found in the file `mc9s12dg128.inc` (automatically included by Codewarrior) as shown below:

```
;*** PTH - Port H I/O Register; 0x00000260 ***
PTH:            equ   $00000260                    ;*** PTH - Port H I/O Register; 0x00000260 ***
; bit numbers for usage in BCLR, BSET, BRCLR and BRSET
```

```
   PTH_PTH0:        equ   0                        ; Port H Bit 0
   PTH_PTH1:        equ   1                        ; Port H Bit 1
   PTH_PTH2:        equ   2                        ; Port H Bit 2
   PTH_PTH3:        equ   3                        ; Port H Bit 3
   PTH_PTH4:        equ   4                        ; Port H Bit 4
   PTH_PTH5:        equ   5                        ; Port H Bit 5
   PTH_PTH6:        equ   6                        ; Port H Bit 6
   PTH_PTH7:        equ   7                        ; Port H Bit 7
   ; bit position masks
   mPTH_PTH0:       equ   %00000001
   mPTH_PTH1:       equ   %00000010
   mPTH_PTH2:       equ   %00000100
   mPTH_PTH3:       equ   %00001000
   mPTH_PTH4:       equ   %00010000
   mPTH_PTH5:       equ   %00100000
   mPTH_PTH6:       equ   %01000000
   mPTH_PTH7:       equ   %10000000


   ;*** PTIH - Port H Input Register; 0x00000261 ***
   PTIH:            equ   $00000261                ;*** PTIH - Port H Input Register;
0x00000261 ***
   ; bit numbers for usage in BCLR, BSET, BRCLR and BRSET
   PTIH_PTIH0:      equ   0                        ; Port H Bit 0
   PTIH_PTIH1:      equ   1                        ; Port H Bit 1
   PTIH_PTIH2:      equ   2                        ; Port H Bit 2
   PTIH_PTIH3:      equ   3                        ; Port H Bit 3
   PTIH_PTIH4:      equ   4                        ; Port H Bit 4
   PTIH_PTIH5:      equ   5                        ; Port H Bit 5
   PTIH_PTIH6:      equ   6                        ; Port H Bit 6
   PTIH_PTIH7:      equ   7                        ; Port H Bit 7
   ; bit position masks
   mPTIH_PTIH0:     equ   %00000001
   mPTIH_PTIH1:     equ   %00000010
   mPTIH_PTIH2:     equ   %00000100
   mPTIH_PTIH3:     equ   %00001000
   mPTIH_PTIH4:     equ   %00010000
   mPTIH_PTIH5:     equ   %00100000
   mPTIH_PTIH6:     equ   %01000000
   mPTIH_PTIH7:     equ   %10000000


   ;*** DDRH - Port H Data Direction Register; 0x00000262 ***
   DDRH:            equ   $00000262                ;*** DDRH - Port H Data Direction
```

```
Register; 0x00000262 ***
   ; bit numbers for usage in BCLR, BSET, BRCLR and BRSET
   DDRH_DDRH0:      equ   0                    ; Data Direction Port H Bit 0
   DDRH_DDRH1:      equ   1                    ; Data Direction Port H Bit 1
   DDRH_DDRH2:      equ   2                    ; Data Direction Port H Bit 2
   DDRH_DDRH3:      equ   3                    ; Data Direction Port H Bit 3
   DDRH_DDRH4:      equ   4                    ; Data Direction Port H Bit 4
   DDRH_DDRH5:      equ   5                    ; Data Direction Port H Bit 5
   DDRH_DDRH6:      equ   6                    ; Data Direction Port H Bit 6
   DDRH_DDRH7:      equ   7                    ; Data Direction Port H Bit 7
   ; bit position masks
   mDDRH_DDRH0:      equ   %00000001
   mDDRH_DDRH1:      equ   %00000010
   mDDRH_DDRH2:      equ   %00000100
   mDDRH_DDRH3:      equ   %00001000
   mDDRH_DDRH4:      equ   %00010000
   mDDRH_DDRH5:      equ   %00100000
   mDDRH_DDRH6:      equ   %01000000
   mDDRH_DDRH7:      equ   %10000000
```

- Without writing any program at all, we can directly light up LEDs by writing directly to memory when Codewarrior is in the Serial mode (i.e. connected to the actual hcs12 board in the lab) as follows:

```
>wb $0262 $ff    configure it as output port
>wb $0260 $f0    turn on upper 4 LEDs
```

- We could also implement a visual binary counter with the following program:

```
      ldaa #$ff
      staa DDRH  ;configure as output
      clra
loop:
      staa PTH          ;light up next binary number
      jsr delay1second  ;software delay loop one second
      inca
      bra loop
```

Notes on Lab 2

- To discuss in class.

Questions

1. The loop in the strcpyASM subroutine uses:

```
ldaa 1,x+
staa 1,y+
bne loopStrcpy
```

It is suggested that the "load/store" instructions could be replaced with a single instruction `movb 1,x+,1,y+` which is a legal instruction. How many bytes are required to encode the load/store instructions and how many clock cycles are required to execute them? How many bytes and clock cycles does the "movb" instruction take? Which method do you think is better? Explain. (*Warning:* This is a tricky question!)

2. How would you write a subroutine `ucube8to8` (i.e. calculate the cube of an 8 bit number where the result fits into 8 bits.)
3. Write subroutines `ucube8to32` (8-bit unsigned cube to 32 bit result) and `scube8to16` (signed 8-bit cube to 16-bit result).
4. Determine the address of Port P and its Direction register. Configure the Port so that the most significant bit is an input and the other bits are output. Assume that the 7 outputs are connected to LEDs and that the input is connected to a (debounced) switch. Initially, turn off all the LEDs. Then write a loop so that whenever the switch changes the binary number displayed by the LEDs increments by 1.
5. Write a subroutine `strcmpASM` that compares strings pointed to by X and Y. The CCR should indicate the result of the comparison. If the two strings are identical, the Z bit should be set; if the string pointed to by X would appear before the one pointed to by Y, the C bit should be set; otherwise both the C and Z bits should be clear. The subroutine should have no side-effects (i.e. all registers except CCR should be unchanged when the subroutine returns.)

Vocabulary

New terms are in **bold**.

Memory Dump	The standard way to display the contents of a block of memory in hexadecimal.
CPU	The Central Processor Unit, the computer's heart.
Bus	A set of parellel wires of digital signals
Address Bus	It indicates the address involved in a bus operation
Data Bus	The data being transferred during a bus cycle.
Control Bus	Signals to time and define the type of bus cycle.

IDE	"Integrated Development Environment". Includes editors, compilers, assemblers, disassemblers, etc. We use *CodeWarrior* in this course. In your Java course, you used *Netbeans*. *Eclipse* is another IDE you may use in other courses.
Read Bus Cycle	Data transferred to the CPU.
Write Bus Cycle	Data transferred from the CPU to memory or a memory-mapped device.
Idle Bus Cycle	Bus inactive
Assembler	Software to translate from symbolic assembler to machine code
Disassembler	Software to translate from machine code to symbolic assembler.
Assembler directive	A line of assembler code that gives information to the assembler software and does not correspond to a machine instruction.
Program Counter	A register containing the address of the next instruction.
Stack Pointer	A register containing the address of the top of stack.
Condition Code Register	Contains bits indicating various conditions (such as whether the last number added was zero, negative, generated a carry, etc.) Also called the *Status Register* in some machines (such as Intel processors).
Index Register	A register mainly used as a pointer. It's size equals the width of the Address Bus.
Arithmetic Shift	Only applies to a right shift where the sign bit is "shifted in" from the right maintaining the sign of the shifted value.
Indexed Addressing	Accessing the contents of memory whose address is calculated by adding an offset (usually zero) to an index register.
Indirect Indexed Addressing	Using indexed addressing to obtain another pointer in memory and then dereferencing the location it points to.
Overflow (V) bit	Set when the result of an addition/subtraction will not fit in the number of bits used.
Effective Address	The address that will be used in indexed addressing. i.e. the index register + offset.
Addressing Mode	The way an operand is specified.
Inherent Addressing	The operand is inherent in the instruction itself.
Immediate Addressing	The operand is part of the instruction (a specific field) and no further memory reference is required.
PC-Relative Addressing	The operand is an offset relative to the current value of the PC.
Subroutine	Similar to a C function. Parameters, if any, can be passed in registers or on the Stack.
Side effect	A change in the CPUs state (such as a register) unrelated to the

	subroutine's function.
Parameter	An argument passed to a subroutine. They are usually passed in registers or on the stack.
Direction Register	A control register associated with a parallel port that configures individual bits to be inputs or outputs.
Memory Mapped Device	A peripheral device whose internal registers are mapped to specific memory locations.