Notes - Unit 4

ASSEMBLY LANGUAGE PROGRAMMING

We refer to the HCS12 CPU Reference Manual Rev. 4.0 for a comprehensive list of Assembly Instructions. We will be referring to sections in this Reference Manual. Also, refer to the MC9S12DG256 memory map found on the Dragon12-Light User Manual to find about the 12KB RAM for Data, the 16KB Fixed Flash for Instructions, etc.

CONDITION CODE REGISTER (CCR)

Many instructions (especially branch instructions) use the bits in the Condition Code Register (CCR). In particular, the status bits reflect the result of a CPU operation:

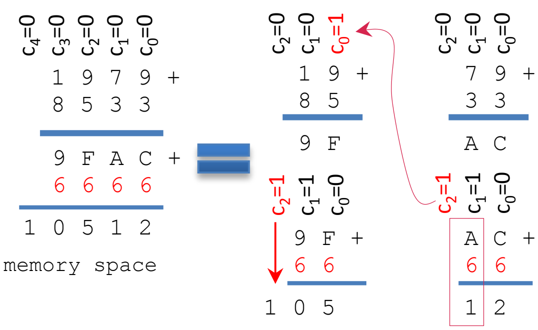
S	X	H	I	N	Z	V	C

- **C**: This bit is set (C←1) whenever a carry occurs during addition or a borrow occurs during subtractions.
  - Addition: The carry bit is the *carry out* bit whether we treat the operands as unsigned or signed numbers.
  - Subtraction: The carry bit is actually a *borrow out* bit. A borrow out bit is only valid when the operands are unsigned. Thus, for a subtraction operation, the C bit is obtained by treating the operands as unsigned numbers.
- **V**: This bit is set (V←1) when there is an overflow in a 2's complement operation, i.e., operands are treated as signed numbers. For an n-bit result, $V=c_n \oplus c_{n-1}$. Note that we can always treat operands as unsigned, but the V bit will be invalid. When using certain instructions (like divide), the overflow bit V has different rules.
  - For example, for the SBA instruction: `[A] ← [A]-[B]`. The subtraction assumes that the operands are unsigned, and as such the C bit will tell us whether there is a borrow. However, the V bit will be incorrect.
- **N**: This bit is set (N←1) when the result of an operation is a negative number. This bit is obtained by treating the result as a signed number. For an n-bit result R, the status bit N is equal to the MSB ($N = R_{n-1}$).
- **Z**: This bit is set (Z←1) when the result of an operation is 0.
- **H**: This bit is set (H←1) when there is a carry from the bit 3 of the accumulator A, i.e., $c_4=1$.

ARITHMETIC OPERATIONS

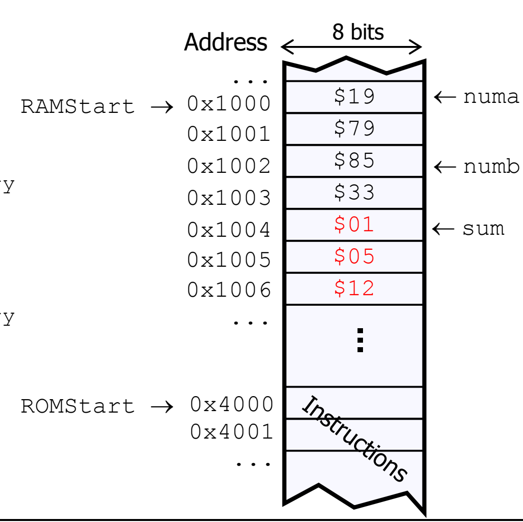
Tables 5.4, 5.5, and 5.10 of the HCS12 CPU Reference Manual list the available arithmetic operations.

EXAMPLE: Multi-precision BCD Addition: Add two BCD numbers, where each BCD number has 4 digits.



ASM Code: `unit4a.asm`

```
; Include derivative-specific definitions
        INCLUDE 'derivative.inc'
ROMStart    EQU $4000  ; ROMStart ← $4000
nbytes EQU 3  ; constant (does not occupy space in memory)

; variable/data section
   ORG RAMStart ; Originate data at address RAMStart($1000)

; Variables definition: Data stored in the RAM section of the memory space
; Debug: Data appears in the Memory Window at Address $1000
numa dc.w $1979; 1 word reserved for the first number
numb dc.w $8533; 1 word reserved for the second number
sum ds.b nbytes; 'nbytes' bytes reserved for the final BCD sum

; code section
   ORG   ROMStart ; Originate Instructions at address ROMStart
   ; Debug -> Assembly Window: Instructions start at ROMStart
   ldaa numa+1          ; A ← [numa+1]
   adda numb+1          ; A ← [A] + [numb+1]
   daa                  ; A: BCD adjust. It may introduce a carry
   staa sum+nbytes-1    ; m[sum+nbytes-1] ← A

   ldaa numa            ; A ← [numa]
   adca numb            ; A ← [A] + [numb] + C
   daa                  ; A: BCD adjust. It may introduce a carry
   staa sum+nbytes-2    ; m[sum+nbytes-2] ← A

   ldaa #0              ; A ← 0
   adca #0              ; A ← [A] + 0 + C
   staa sum             ; m[sum] ← A
```



MULTIPLICATION

- For the multiplication instructions (`emul, emuls, mul`), be aware of how the operands are treated (signed or unsigned). Note that when multiplying numbers, the number of bits of the result is the sum of the number bits of the multiplicands.

Examples:

- Unsigned multiplication of two 16-bit operands.
  We can use `emul`: Unsigned multiplication of `[D]` and `[Y]`. The 32-bit result is stored in `Y:D`
```
ldd #$FA34 = 64052
ldy #$012B = 299
emul       ; 64052×299 = 19151548 = $01243ABC. Y ← $0124, D ← $3ABC
```

- Signed multiplication of two 16-bit operands.
  We can use `emuls`: Signed multiplication of `[D]` and `[Y]`. The 32-bit result is stored in `Y:D`
```
ldd #$FA34 = -1484
ldy #$012B = 299
emuls      ; -1484×299 = -443716 = $FFF93ABC. Y ← $FFF9, D ← $3ABC
```

- Unsigned multiplication of two 8-bit operands.
  We can use `mul`: Unsigned multiplication of `[A]` and `[B]`. The 16-bit result is stored in `D`
```
ldab #$91 = 145
ldaa #$F2 = 242
mul        ; 145×242 = 35090 = $8912. D ← $8912
```

DIVISION:

- For the division operations (`ediv, edivs, fdiv, idiv, idivs`) be aware of how the operands are treated (signed, unsigned) and how many bits are specified for each operand.

- 32 by 16 bit divide (`ediv, edivs`): Dividend `Y:D`. Divisor: `X`. The quotient (stored in Y) and the remainder (stored in D) are 16-bits wide. The quotient, however, might require more than 16 bits for its proper result (e.g. unsigned `FFFFFFFF/0001`). In this case, the overflow bit is set. If a division by zero is attempted, the C bit is set, and the contents of `D` and `Y` do not change.

Examples:

- Unsigned division:
```
ldy #$0033
ldd #$1B89 ; Dividend: $00331B89 = 3349385
ldx #$E24A ; Divisor: $E24A = 57930
ediv       ; 3349385/57930: Y(quotient) ← 57 = $0039, D(Remainder) ← 47375 = $46F1
```

- Signed division:
```
ldy #$FFF5 ;
ldd #$02EA ; Dividend: $FFF502EA = -720150
ldx #$0653 ; Divisor: $0653 = 1619
edivs      ; -720150/1619: Y(quotient) ← -445 = $FE43, D(Remainder) ← 305 = $0131
```

- 16 by 16 bit divide (`idiv, idivs`). Dividend: D, Divisor: X. Quotient (in X) and Remainder (in D) are 16-bits wide. Here, the 16 bits are enough for all possible cases. If a division by zero is attempted, C is set, Quotient is `$FFFF` and remainder indeterminate.
  - ✓ `idivs` instruction: We have the case `$8000/$FFFF=-32768/-1=32768`, which requires 17 bits in 2's complement representation). Here, the V bit is set.

- `fdiv` instruction: Unsigned Fractional divide (16 bits by 16 bits). Dividend: D, Divisor: X. Quotient (in X) and Remainder (in D) are 16-bits wide. This is useful for division when numbers are represented in fixed point arithmetic.

IMPLEMENTING LOOPS

Loops make use of the Branch Instructions along with Compare and Test, Loop Primitive, and Decrementing / Incrementing Instructions. The Branch Instructions can be classified by the type of condition required to branch:

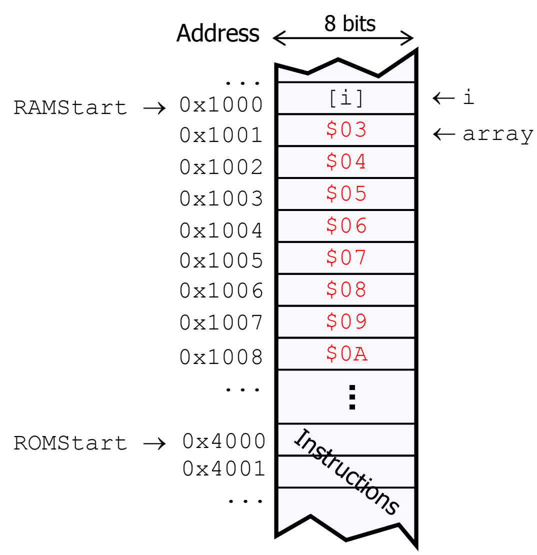
- **Unconditional branches:** The branch is always taken (e.g.: `bra next`)
- **Simple branches:** These instructions take a look at a particular bit in the Condition Code Register (`CCR`) to determine whether to branch.
  For example: `beq next` ; If `Z=1`, it branches.
  For example: `bvs next` ; If `V=1` (overflow in 2's complement), it branches.
- **Unsigned branches**: These instructions treat the previous operation as between unsigned numbers and as such look for a specific combination of CCR bits.
  For example: `bls next` ; If in the previous operation, the first operand was lower than or equal to the second operand, then this means that `C+Z=1` (this is only true if the operands are treated as unsigned).
- **Signed branches**: These instructions treat the previous operation as between unsigned numbers and as such look for a specific combination of CCR bits.
  For example: `bge next` ; If in the previous operation, the first operand was greater than or equal to the second operand, then this means that `N⊕V=0` (this is only true if the operands are treated as signed).

EXAMPLE: Store the numbers from `na` to `nb` in a memory array. This can be implemented as a *for loop* :

```
for i = na to nb do
    array[i] ← i
end
```

ASM Code: `unit4b.asm`

```
; Include derivative-specific definitions
       INCLUDE 'derivative.inc'
ROMStart    EQU  $4000  ; ROMStart ← $4000
na EQU 3
nb EQU 10

; variable/data section
       ORG RAMStart ; Originate data at address RAMStart

; Variables Definition:
i ds.b 1; 1 byte is reserved for the index. Range: 0 to 255
array ds.b nb-na+1; nb-na+1 bytes reserved for the final sum

; code section
       ORG   ROMStart
       movb #na,i   ; m[i] ← na
       ldx #array   ; X ← array

loop   movb i, 0,X ; m[X] ← [i]
       ldaa i       ; A ← [i]
       cmpa #nb
       beq next
       inx
       inc i        ; m[i] ← [i]+1
       bra loop

next   bra next ; infinite loop
```



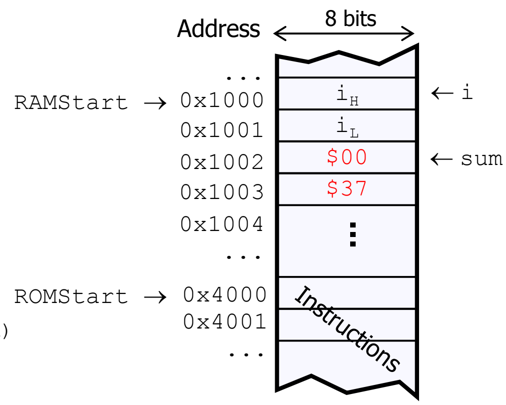
- The definition of the constants `na` and `nb` allows us to have a generic code.
- Note that `X` is used as an index. Initially `X ←$1001`. Then the memory contents pointed to by `X` store the value of '`i`'. As `i` increases, so does the address stored in `X`.
- At every iteration, the register `A` holds the value of `i` so that it can be compared with `nb`.
- The variable '`array`' is defined as having '`nb-na+1`' bytes.
- The variable '`i`' occupies 1 byte, i.e., `na` and `nb` are limited to 0 to 255.
- The instruction `beq` branches if the result of the previous operation is 0. What the instruction actually checks is whether the bit `Z` of the `CCR` is 1.
- The instruction `inc` increments the contents of a memory address. Note that only the 8-bit data of the memory address is incremented, so this instruction does not work with variables defined as having more than one byte.
- The last instruction `bra next` is an unconditional branch. By always branching to itself, it enters into an infinite loop.

EXAMPLE: Add the numbers from `na` to `nb`. This can be implemented as a *for loop*:

```
sum = 0
for i = na to nb
        sum ← sum + i
end
```

ASM Code: `unit4c.asm`

```
; Include derivative-specific definitions
        INCLUDE 'derivative.inc'
ROMStart    EQU  $4000   ; ROMStart ← $4000
na EQU 1
nb EQU 10

; variable/data section
        ORG RAMStart ; Originate data at address RAMStart
; Variables Definition:
i ds.w 1; 1 word is reserved for the index
sum ds.w 1; 1 word is reserved for the final sum

; code section
        ORG   ROMStart ; Originate data at address ROMStart.
        ldx #na         ; X ← na
        ldd #na         ; D ← na

loop:   cpx #nb         ; X = nb? (X-nb: only CCR bits are modified)
        beq next        ; Z = 1? (i.e., X=i?)
        inx
        stx i
        addd i          ; D ← [D] + [i]
        bra loop
next    std sum         ; m[sum] ← [D]
```



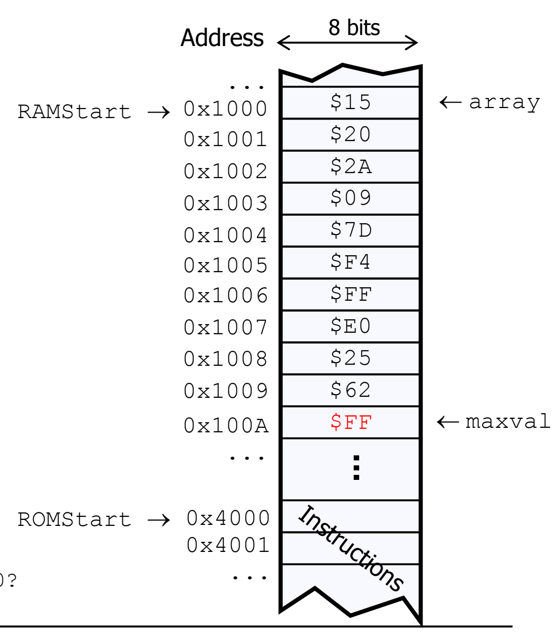
- `'i'` is defined as 1 word (16 bits, range: 0 to 65535). To operate `'i'`, we must use instructions that operate with 16 bits.
- The variable `'sum'` occupies 16 bits (one word). This is where the final result will be stored. The most efficient way to accumulate the summation is to store it in register `D` and use the `addd` instruction. In the example, the sum of the numbers 1 to 10 is: `1+2+3+4+5+6+7+8+9+10=55=$0037`.

EXAMPLE: Given 10 consecutive positive numbers (1 byte) in memory, find the maximum value.

```
maxval ← array[0]
i=1;
while (i < N) do
        if maxval < array[i] then
                maxval ← array[i]
        end
        i ← i + 1
end
```

ASM Code: `unit4d.asm`

```
; Include derivative-specific definitions
        INCLUDE 'derivative.inc'
ROMStart    EQU  $4000   ; ROMStart <- $4000
N EQU 10

; variable/data section
            ORG RAMStart ; Originate data at address RAMStart
; variables definition:
array dc.b 21,32,42,9,125,244,255,224,37,98; positive numbers
maxval ds.b 1 ; 1 byte is reserved for the maximum value

; code section
            ORG   ROMStart ; Originate data at address ROMStart
            movb array, maxval ; m[maxval] ← [array]
            ldx #(array + 1); X ← array+1
            ldab #N          ; B ← N
loop:       ldaa maxval      ; A ← [maxval]
            cmpa 0,X
            bhs next         ; A >= [$0+X]?
            movb 0,X,maxval ; m[maxval] ← [$0+[X]]
next        inx
            dbne B,loop ; loop primitive instruction. B ← B-1 = 0?
```

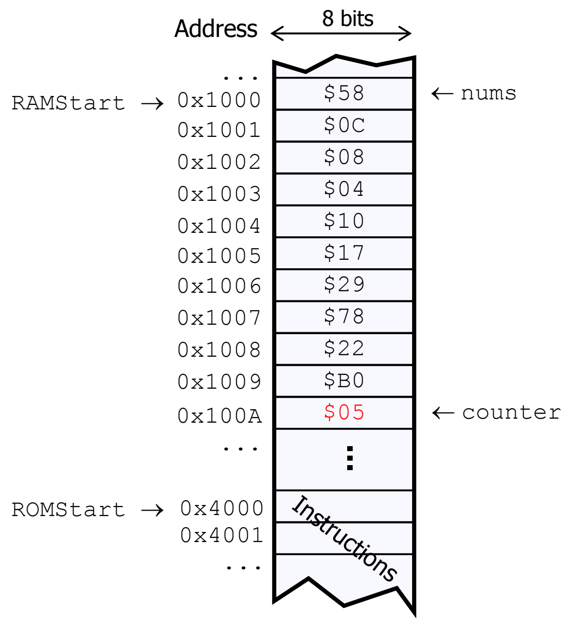


- Assumption: We are working with positive numbers. Thus, we used `bhs` instead of `bge`. `bge` makes the comparison treating the numbers as signed, while `bhs` makes the comparison treating the numbers as unsigned. Since we are working with positive numbers, we must use `bhs`. If we were treating the bytes as unsinged, `bge` must be used instead.

EXAMPLE: Given 10 numbers, count the number of elements that are divisible by 8.
We count the numbers whose 3 LSBs are zero.

ASM Code: `unit4e.asm`

```
; Include derivative-specific definitions
        INCLUDE 'derivative.inc'
ROMStart    EQU  $4000  ; ROMStart <- $4000
N EQU 10

; variable/data section
      ORG RAMStart ; Originate data at address RAMStart
; variables definition:
nums dc.b 88,12,8,4,16,23,41,120,34,176;
counter ds.b 1 ; 1 byte is reserved for the count

; code section
            ORG   ROMStart
            clr counter
            ldx #nums  ; X ← nums
            ldab #N    ; B ← N
loop:       brclr 0,X, $07, yes
            bra nno
yes:        inc counter
nno:        inx
            dbne B, loop ; B ← B-1 = 0?
forever:    bra forever
```



Shift and Rotate Instructions

Logical shift: The input bit is 0, and the output bit goes to C flag.
Arithmetic shift: The output bit goes to C flag. The input bit is zero if left shift. The input bit is the MSB if right shift

Example: `A = $9A, B = $CE, m[$1F] = $B7`

	Result	C		Result	C		Result	C		Result	C
lsla	A = $34	1	lsra	A=$4D	0	asla	A = $34	1	asra	A = $CD	0
lslb	B = $9C	1	lsrb	B=$67	0	aslb	B = $9C	1	asrb	B = $E7	0
lsld	D = $359C	1	lsrd	D=$4D67	0	asld	D = $359C	1			
lsl $1F	m[$1F]=$6E	1	lsr $1F	m[$1F]=$5B	1	asl $1F	m[$1F]=$6E	1	asr $1F	m[$1F]=$DB	1

Rotate: We can rotate to the left or to the right. The trick is that these instructions use whatever it is on the carry bit

Example: `A = $9A, B = $CE, m[$1000] = $B7`

C = 0				C = 1			
	Result		Result		Result		Result
rola	A=$34,C=1	rora	A=$4D,C=0	rola	A=$35,C=1	rora	A=$CD,C=0
rolb	B=$9C,C=1	rorb	B=$67,C=0	rolb	B=$9D,C=1	rorb	B=$E7,C=0
rol $1000	m=$6E,C=1	ror $1000	m=$5B,C=1	rol $1000	m=$6F,C=1	ror $1000	m=$DB,C=1

Boolean Instructions

Example: `D = $BE45, m[$1000] = $C3`

	Result		Result		Result		Result
anda $1000	A = $82	oraa $1000	A = $FF	eora $1000	A = $7D	com $1000	m[$1000] = $3C
andb $1000	B = $41	orab $1000	B = $C7	eorb $1000	B = $86	neg $1000	m[$1000] = $3D

Bit Test and Manipulate Instructions

Take a look at HCS12 CPU Reference Manual Rev. 4.0 for detailed information on the instructions (allowed addressing modes, bytes per instruction, etc).

Examples:

- `bclr 0,X,$42`: Clears bit 1 and bit 6 (`$42 = 0100 0010`) of the memory contents pointed by `0+[X]`
- `bset 0,Y,$45`: Sets bit 1, bit 3, bit 7 (`$85: 1000 0101`) of the memory contents pointed by `0+[Y]`
- `bita 0,X`: Performs `A AND [0+[X]]`. Updates Z if result in 0, and N if result is negative. V is set to zero.

PROGRAM EXECUTION TIME

The execution time is the sum of bus cycles a series of instructions takes. For each instruction, this information is found on the Access Detail column of the Instruction Glossary in the HCS12 CPU Reference Manual. For example, `psha` takes 2 cycles (the Access Detail column has 2 letters, `pula` takes 3 cycles, and `nop` takes one cycles. This is very useful to create time delays. In particular, `psha` followed by `pula` takes 5 cycles and does nothing.

Example:

- We want to generate a 50 ms delay on a Dragon12-Light Board with a 25 MHz bus clock.
- This can be accomplished by a loop. Each iteration of the loop takes n cycles. If we loop for $ntimes$, then we have that our delay was $ntimes \times n\ cycles \equiv ntimes \times n \times clock\ period\ seconds$.
- $ntimes \times n \times \frac{1}{25\times10^6} = 50 \times \frac{1}{10^3} \rightarrow ntimes \times n = 125 \times 10^4$
- As a counter, $ntimes$ can be stored in a 16-bit register (largest one). This means that $ntimes \leq 65535$. Then, a good number would be 50000. This results in $n = \frac{125\times10^4}{50000} = 25$. So, we need to have a loop with 25 bus cycles.

<table>
<tr><th>Using just nops</th><th>More efficient code (fewer instructions)</th></tr>
<tr><td><pre>ldx #50000
loop: nop   ; 1 cycle
       nop   ; 1 cycle
       nop   ; 1 cycle
       ...
       nop   ; 1 cycle
       dbne X, loop ; 3 cycles</pre></td><td><pre>ldx #50000
loop: psha   ; 2 cycles
       pula   ; 3 cycles
       psha   ; 2 cycles
       pula   ; 3 cycles
       psha   ; 2 cycles
       pula   ; 3 cycles
       psha   ; 2 cycles
       pula   ; 3 cycles
       nop   ; 1 cycle
       nop   ; 1 cycle
       dbne X, loop ; 3 cycles</pre></td></tr>
</table>

MULTIPLY-ACCUMULATE INSTRUCTION

- `emacs <opr>`: multiplies the 16 bits pointed by X and the 16 bits pointed by Y and add the result to the memory operand (4 bytes). Stores the result in the same address. It is a signed operation, i.e. multiplication is carried out assuming that the numbers are in 2's complement.

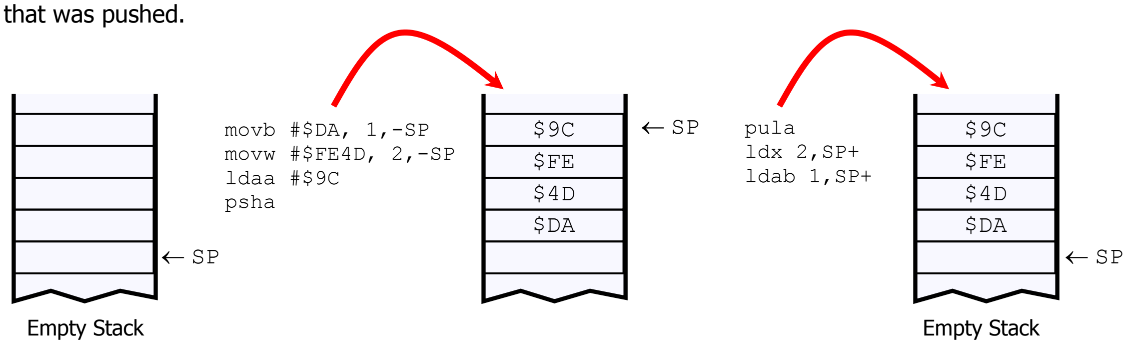
Example: $p \times w^2 + q \times w + r = (p \times w + q) \times w + r$. In the following code, we use `p=125, w=103, q=3452, r=2134`.

ASM Code: `unit4f.asm`

```
; Include derivative-specific definitions
        INCLUDE 'derivative.inc'
ROMStart    EQU  $4000  ; absolute address to place my code/constant data
            ORG RAMStart ; Start data at $1000
p dc.w 125    ; 1 word reserved (only use 1 byte: -128 to 127 to avoid overflow).
w dc.w 103    ; 1 word reserved (only use 1 byte: -128 to 127 to avoid overflow).
q dc.w 3452   ; 1 word reserved (only use 15 bits: -2^14 to 2^14-1 to avoid overflow).
r ds.w 2      ; 2 words reserved (only use 31 bits: -2^30 to 2^30-1 to avoid overflow)
result ds.w 2 ; 2 word reserved for the 32-bit result
k ds.w 1      ; 1 word reserved for intermediate 16-bit result

; code section
            ORG   ROMStart
Entry:
_Startup:   LDS   #RAMEnd+1       ; initialize the stack pointer. SP <- $3FFF+1

mainLoop:   movw  #0,r
            movw  #2134, r+2      ; r <- 2134

            ldx #p
            ldy #w
            movw #0,result
            movw q,result+2       ; result <- q

            emacs result          ; result <- p*w + result

            movw result+2, k      ; k <- p*w+q
            ldx #k
            movw r, result
            movw r+2, result+2    ; result <- r

            emacs result; ; result <- k*w + result
```

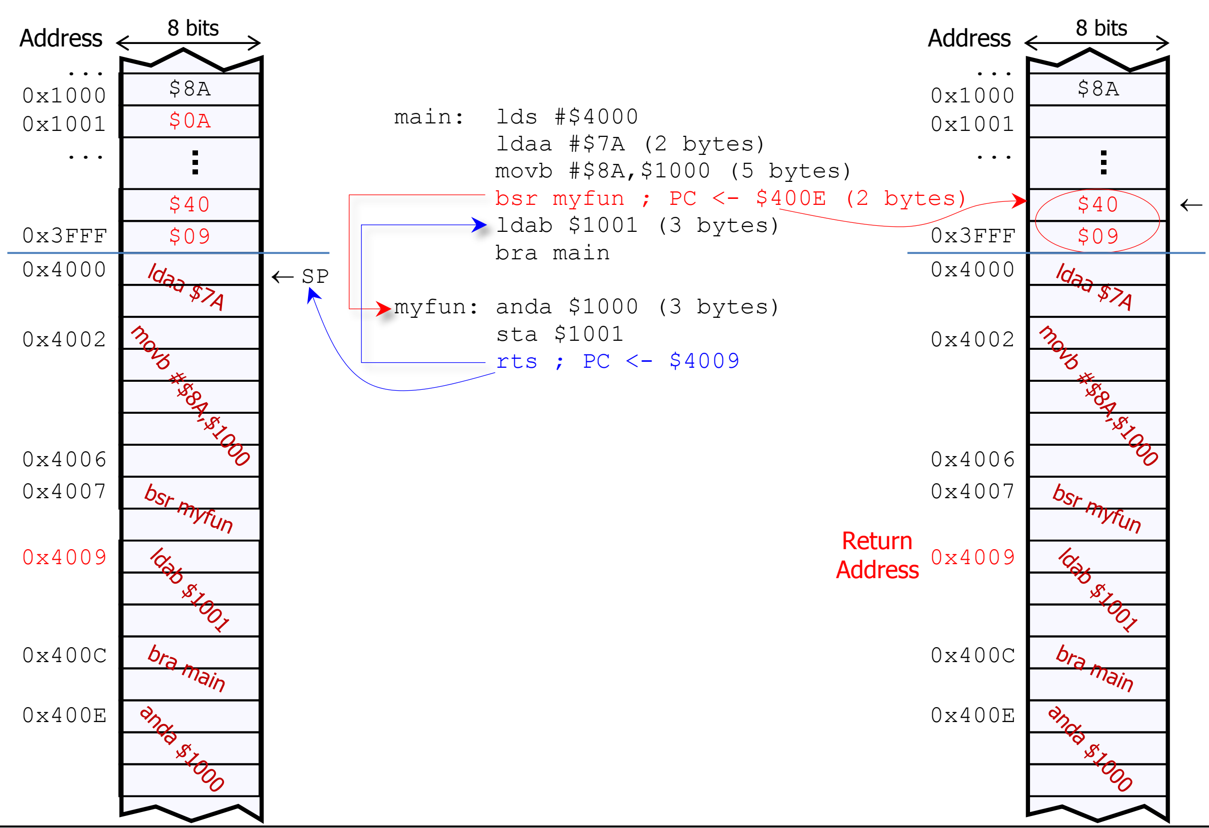
SUBROUTINES

- A subroutine is a sequence of instructions that can be called from different places in a program.
- Once the subroutine is finished, it returns to the instruction immediately following the call instruction (bsr, jsr, call). To accomplish this, when the subroutine call instruction is executed, the address of the next instruction (called *Return Address*) is stored in the Stack. When the subroutine is finished, the return instruction (rts, rtc) grabs the *Return Address* from the Stack and places it on the Program Counter so that we continue with the execution of the instruction located at the *Return Address*.
- The Stack is a LIFO structure. Values are pushed onto the Stack. Values are pulled from the Stack starting with the last value that was pushed.



EXAMPLE:

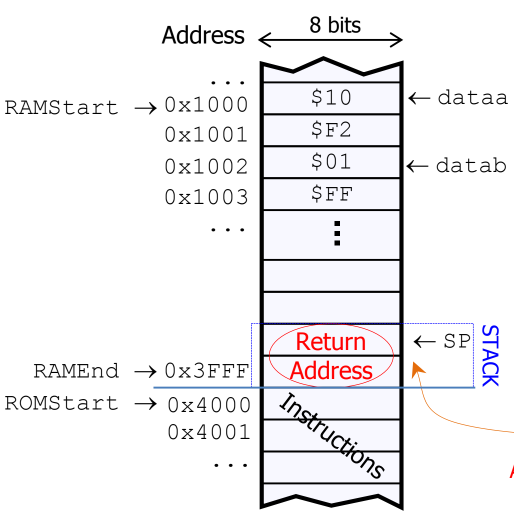
- Here, the subroutine called '`myfun`' performs the bitwise AND operator of register A with a memory position. The result is stored at another memory position.
- Note how the Stack Pointer (SP) is given the value $4000 (which is the address where the instruction start). The idea is that the Stack must grow from the last allowable data memory (`$1000` to `$3FFF`). So, when we want to add data to the Stack, we first decrease the value of SP and then store the data.
- The memory figure on the left is the state of the memory after the `bsr myfun` instruction has been executed (and right before entering the subroutine). This instruction does the following:
  - ✓ `SP ← SP – 2`. Then, the Return Address (`0x4009`) is stored on `m[SP]` and `m[SP+1]`.
  - ✓ `PC ← PC + offset`. This becomes: `PC ← Address of 'myfun' = $400E`
- The memory figure on the right is the state of the memory after the rts instruction has been executed. This instruction does the following:
  - ✓ `SP ← SP + 2`. This makes the Stack effectively disappear.
  - ✓ `PC ← PC + offset`. This becomes: `PC ← Return Address = $4009`.

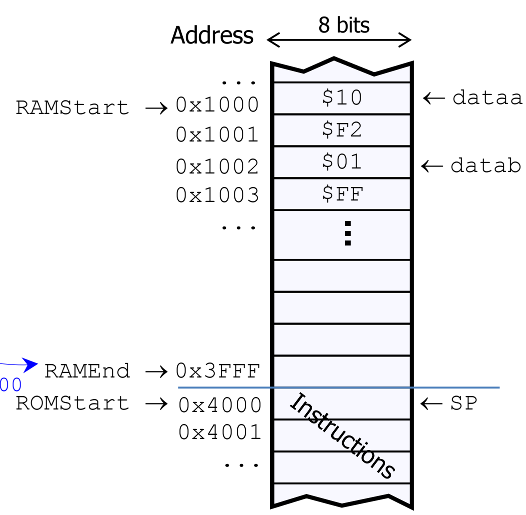


EXAMPLE: Applying the 2's complement operation to a 16-bit value

- In this example, the subroutine '`my_neg`' computes the 2's complement of a 16-bit value (or changes the sign of a 16-bit value). The input value is given in D. The output value is returned in Y.
- The subroutine employs the register D to perform the operation. However, the main routine uses D after the subroutine is completed. Therefore, the subroutine has to push the value of D into the Stack before modifying it. Before exiting, the subroutine pulls the value of D from the Stack to restore D to its original value.
- In the code below, we use `pusha` and `pushb` to store the contents of D (note that `pushd` could have also been used). We also use `pula` and `pulb` to restore the contents of D (`puld` could have also been used).
- Note that during the execution of the subroutine, the Stack stores A, B, and the Return Address.
- In general, we can pass parameters to a Subroutine via: Registers, Stack (parameters are pushed on the Stack before the subroutine is called), and Global Memory. We can also get the results from the Subroutine on: Registers, Stack (the main routine requires to create space in the Stack before the subroutine call), and Global Memory.

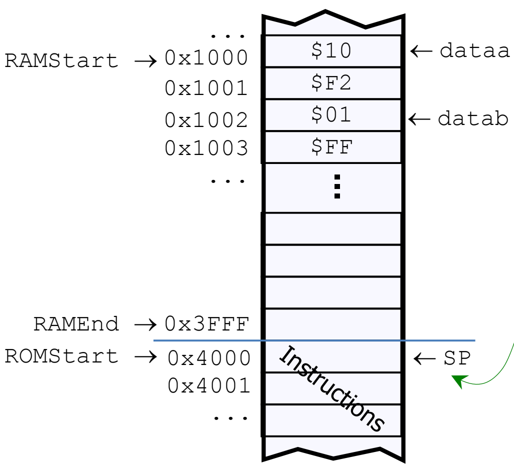
- `lds #RAMEnd+1`:
  In CodeWarrior with the Dragon12-Light Board, the Stack Pointer is assigned the value of `#RAMEnd+1 = $3FFF+1 = $4000`. At this point, the Stack is empty, and we are not supposed to write on `$4000`, since this is the starting address of the Instructions (`ROMStart = $4000`). Every time we push a value onto the Stack, we use `pusha, pushb, pushc, pushd, bsr, jsr`. These instructions decrease the value of the Stack Pointer, and then store the contents on `m[SP]`. If we use `movw, movb` to store data on the Stack (this is useful when we have to input parameters and we are out of registers), we have to make sure to first decrease the value of the Stack Pointer.
- The value of `SP=$4000` means that the Stack is empty. By using this value of `$4000`, we are placing the Stack at the last memory addresses of the RAM section (`$1000` to `$3FFF`).

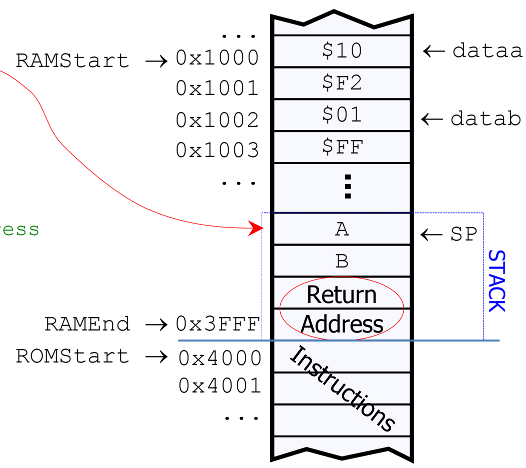





```
INCLUDE 'derivative.inc'
ROMStart    EQU  $4000

; variable/data section
            ORG RAMStart
; variables definition:
dataa dc.w $10F2;
datab dc.w 511;

; code section
            ORG   ROMStart
Entry:
_Startup:   lds #RAMEnd+1 ; SP←$4000
            ldd dataa
            bsr my_neg
            addd datab
            sty 0,D

; ===== Subroutine ============
my_neg:     pshb
            psha
            coma
            comb
            addd #1
            leay 0,D
            pula
            pulb
            rts ; PC <- Return Address
```

Return Address





EXAMPLE: Fibonacci sequence

$F_0 = 0, F_1 = 1$ (starting point). $F_n = F_{n-1} + F_{n-2}$

- The following algorithm gets F(counter) and places this result in variable 'FiboRes'.
- A Subroutine called '`CalcFibo`' computes the Fibonacci number. The Subroutine gets the input argument in register X; the subroutine provides the result in D. Since D is a 16-bit register, the maximum positive number we can store is 65535. Since `F(24) = 46368` and `F(25) = 75025`, we can only compute up to `F(24)`.
- The subroutine modifies the contents of X. As a precautionary measure, the subroutine pushes the value of X on the stack at the beginning of the subroutine and pulls the value of X at the end of the subroutine (so that X keeps its proper value).

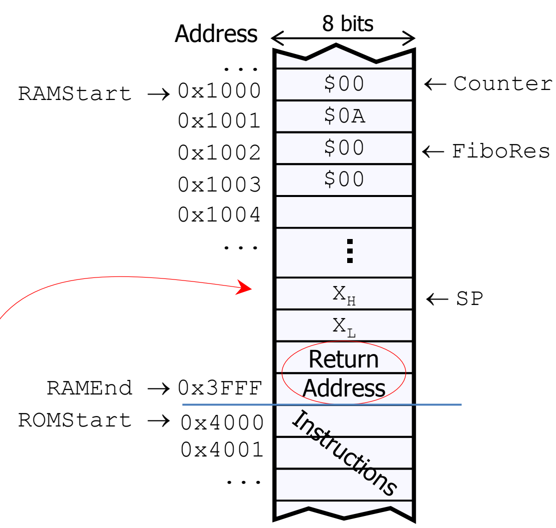
Algorithm:

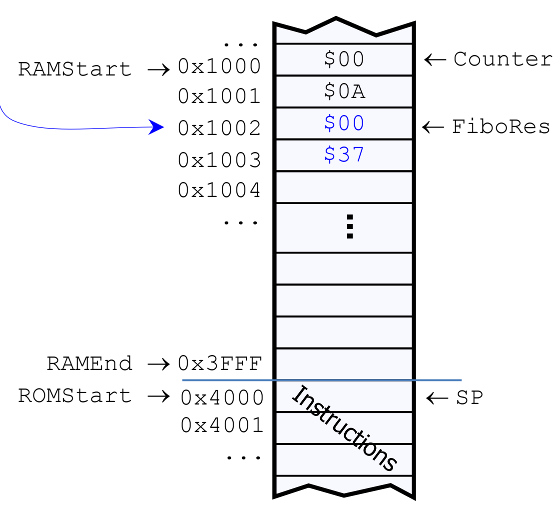
Main Routine	**CalcFibo** SubRoutine. Input Parameter X. Result stored in D

X ← counter
if X > 24 then
   restart
end
 	
Go to Subroutine CalcFibo
 	
FiboRes ← D
restart
	push X
	Y ← 0, D ← 1 (Y stores F_{n-2}, D stores F_{n-1})
	X ← X-1
	if X = 0 then
	   exit CalcFibo routine
	end
	while X ≠ 0
	   Y ← D+Y
	   exchange D and Y
	   X ← X-1
	end
	pull X

ASM Code: `unit4g.asm`

```
; Include derivative-specific definitions
                INCLUDE 'derivative.inc'

ROMStart    EQU  $4000
; variable/data section
            ORG RAMStart ; Start data at $1000
 ; Insert here your data definition.
Counter     DC.W 10 ; Fibonacci number to compute. Counter >= 1
FiboRes     DS.W 1 ; 1 word (16 bits) reserved for the result


; code section
            ORG   ROMStart

Entry:
_Startup:   LDS   #RAMEnd+1     ; SP <- $3FFF+1 (initialize SP)

mainLoop:   LDX   Counter       ; X contains counter
            CPX   #24
            bhi   mainLoop      ; Fibo(25) causes overflow!

            BSR   CalcFibo
            STD   FiboRes       ; store result
            BRA   mainLoop      ; restart.


; ==========================================================
; Subroutines
; ==========================================================
CalcFibo:  ; Function to calculate fibonacci numbers.
            pshx
            LDY   #$00          ; Fn-2
            LDD   #$01          ; Fn-1
            DBEQ  X,FiboDone    ; If X was 1, we're done
FiboLoop:
            LEAY  D,Y           ; Y ← D+Y
            EXG   D,Y           ; exchange D and Y
            DBNE  X,FiboLoop
FiboDone:   pulx
            RTS                 ; result in D
```





STACK FRAME:
It is an orderly way to store data on the Stack:

- Incoming Parameters: They are usually provided as registers. But if we run out of registers, we can always place incoming parameters on the Stack.
- Return Address: These two bytes are always stored on the Stack.
- Saved Registers: If we need to use registers inside the subroutine, a good practice is to push the registers' values at the beginning of the subroutine. When the subroutine is finished, the register recover their original values. This way, the registers are never modified after a subroutine has been executed.
- Local Variables: We might need memory for particular operations inside the subroutine. If we just want to use this memory while the subroutine is executing, we use memory from the stack. This is called Local memory, because it is not available once we exit the subroutine.

Example:

```
; Include derivative-specific definitions
               INCLUDE 'derivative.inc'

ROMStart    EQU  $4000  ; absolute address to place my code/constant data

; variable/data section

            ORG RAMStart ; Start data at $1000
 ; Insert here your data definition.
val1 dc.b #$A2 ; $039D
val2 dc.b #$91 ; $00F2
val3 dc.b #$34 ; $00F5

; code section
            ORG   ROMStart

Entry:
_Startup:   LDS   #RAMEnd+1    ; initialize the stack pointer. SP <- $3FFF+1

mainLoop:   ; ................
            movb val1,1,-SP
            movb val2,1,-SP
            movb val3,1,-SP

            bsr myfun

            leas 3,SP

            ; More instructions
            ; ..........

forever:    bra forever
; ====================================
; Subroutines
; ====================================

myfun:      psha
            pshb

            ; Allocating 4 bytes for local variables
            leas -4,SP; SP <- SP-4


            ; Instructions that use local variables, registers,
            ; and input parameters provided in the Stack
            ; ................
            ; ................

            leas 4,SP; SP <- SP+4 ; De-allocating Local Memory
            pulb
            pula
            rts
```