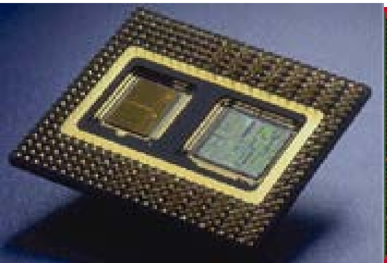
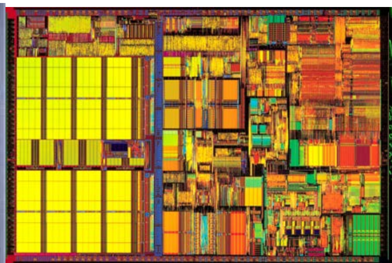
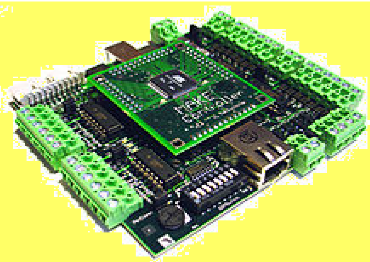
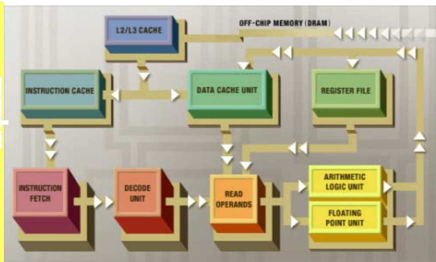
Lecture 9
Subroutines

Bryan Ackland
Department of Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ 07030

Adapted from HCS12/9S12 An Introduction to Software and Hardware Interfacing Han-Way Huang, 2010

Structured Programming

- When developing a large, complex program, desirable to hierarchically partition code into small functions that have:
  - single entry point
  - well defined interface (input parameters, results)
  - well defined, consistent functionality
  - minimal side effects
  - especially important in assembly language programming
- Well structured programming makes the code
  - easy to read & document
  - easy to verify and debug
  - easy to maintain
- In assembly language, we hierarchically partition the code into small functions using subroutine call

Subroutine

- A subroutine is a sequence of instructions that can be called from many places in a program
  - allows same operation to be performed on different parameters
- When a subroutine is called, the processor saves the return address (address of next instruction after the call)
- When a subroutine has completed it uses this return address to resume execution of the calling code

```
start:          •         ; start of main program
                •
          jsr   subroutine_x
                •
end:            •         ; end of main program
                •
                •
subroutine_x:   •         ; start of subroutine
                •
                •
          return          ; end of subroutine
```

Subroutine Instructions

- There are a number of instructions that support subroutine call and return. We will consider only two:

Mnemonic	Function	Description
jsr <sub>	jump to subroutine	Address of subroutine <sub> can be specified using extended, indexed or indexed indirect mode - anywhere in 64kB address space. But it is usually specified using a label. The return address (i.e. address of next sequential instruction after the subroutine call) is automatically PUSH'd on to the stack
rts	return from subroutine	PULL's return address from stack and loads it into PC. Calling program resumes at the return address.

- Note that a main program must set up the stack (set the SP to suitable memory address) before calling a subroutine

Subroutine Example:

- Write a subroutine that determines the length of a string (in bytes), not including the NULL termination. A pointer to the string is passed (to the subroutine) in register X and the string length is returned (to the main program) in accumulator A

```
strlen:   clra              ;initialize character count to 0
slp:      tst     1,x+      ;test for NULL
          beq     done
          inca              ;increment count
          bra     slp
done:     rts               ;return to caller
```

- Subroutine can be called as follows:

```
          lds     #$5000          ; set up stack (once at beginning of program)
               :
          ldx     #test_string    ;load parameter into X
          jsr     strlen          ;execute subroutine
          staa    length          ;subroutine returns here
```

Subroutine Data Issues

- jsr and rts instructions deal with program flow and ensure we return correctly to calling program.

- Programmer must also deal with:
  - passing parameters to subroutine
    - string pointer in X in previous example
  - retrieving results from subroutine
    - character count in A in previous example
  - allocating local data storage space for subroutine operations
    - not required in previous example
  - saving data stored in registers used by subroutine
    - not required in previous example

Subroutine Issues: Parameters & Results

- **Parameter Passing:**
  - **Use registers:** Convenient when there are only a few parameters to be passed
  - **Use global memory:** Accessible to both caller and callee. Good structured programming practice limits passing of global variables. Limits ability to make subroutines re-entrant.
  - **Use stack:** Parameters are pushed on to stack before subroutine is called. Stack must be cleaned up after subroutine has executed.
- **Result Returning:**
  - **Use registers:** Convenient when only a few bytes to be returned
  - **Use global memory:** Accessible to both caller and callee. Same concern about use of global variables
  - **Use stack:** Caller allocates space on stack before making subroutine call
  - **Use pointer parameters:** Caller passes pointer to variables that need to be modified by the subroutine

Subroutine Issues: Parameters & Results

- When a program “calls” a subroutine, the caller and the subroutine must agree on how parameters will be passed to the subroutine and how results will be returned to the caller

- We sometimes say that there is a “contract” between the calling program and the subroutine which defines how parameters and results will be passed.

Subroutine Issues: Saving Registers

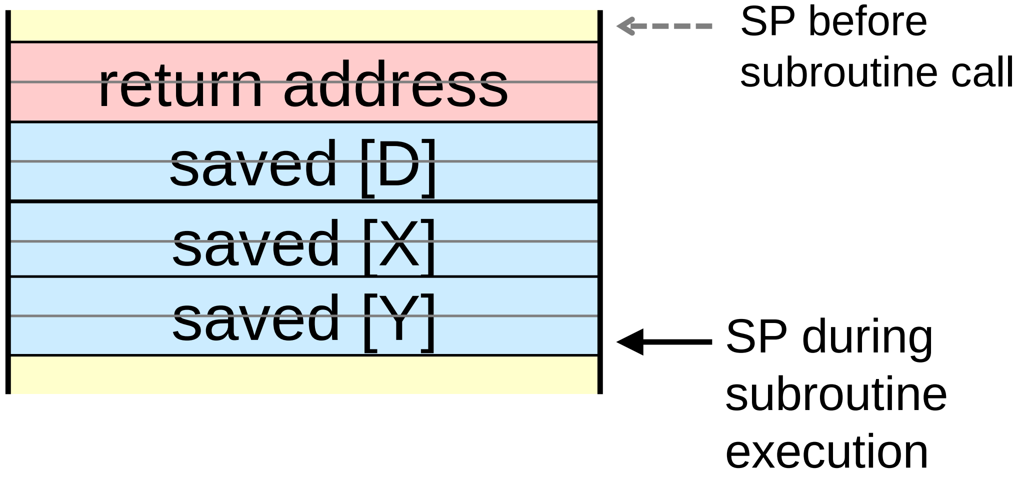
- Subroutine may use some CPU registers that are being used by caller
- Best practice is to make no assumptions about which registers are being used by caller
  - makes subroutine useful in broader arrays of apps.

- All registers used by subroutine (except those used for passing parameters or results) should be saved to stack
  - registers must be restored before returning to caller
  - registers are pulled off stack in reverse order
  - For example if a subroutine uses D, X and Y:

```
sub:    pshd
        pshx
        pshy
         •
         •
        puly
        pulx
        puld
        rts
```

Saving Registers on Stack

```
sub:    pshd
        pshx
        pshy
         •
         •
        puly
        pulx
        puld
        rts
```



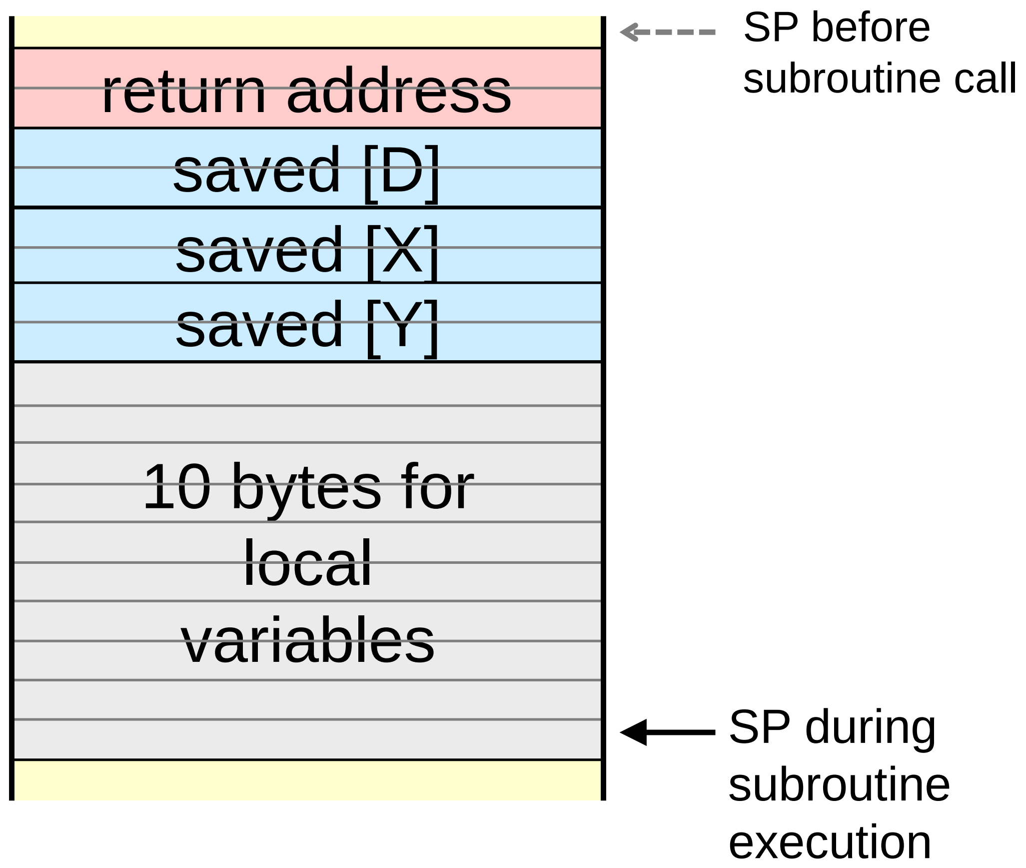
Subroutine Issues: Local Variables

- Subroutine may need local variables to complete operation
  - beyond that provided by register storage
- Not wise to use global variables
  - local variables should be limited in scope to subroutine
  - not available to caller once subroutine has returned
- Local variables should be allocated on stack
- **leas** instruction can be used to allocate and de-allocate space on stack, e.g:

```
leas    -10,sp      ;SP ← [SP] – 10
    •               ;effectively allocates10 bytes to top of stack
    •
leas    10,sp       ;SP ← [SP] + 10
                    ;de-allocates 10 bytes from top of stack
```

Local variables on Stack

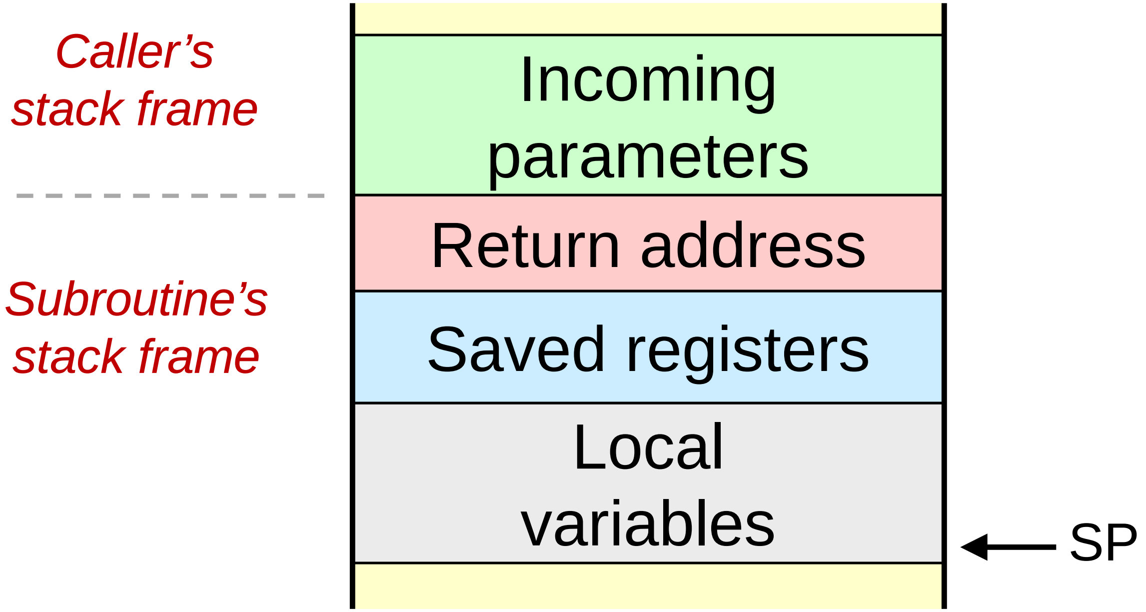
```
sub:    pshd
        pshx
        pshy
        leas    -10, SP
          •
          •
        leas    10, SP
        puly
        pulx
        puld
        rts
```



- Note that during the execution of the subroutine, the SP can be used as an index register to access local variables

Stack Frame

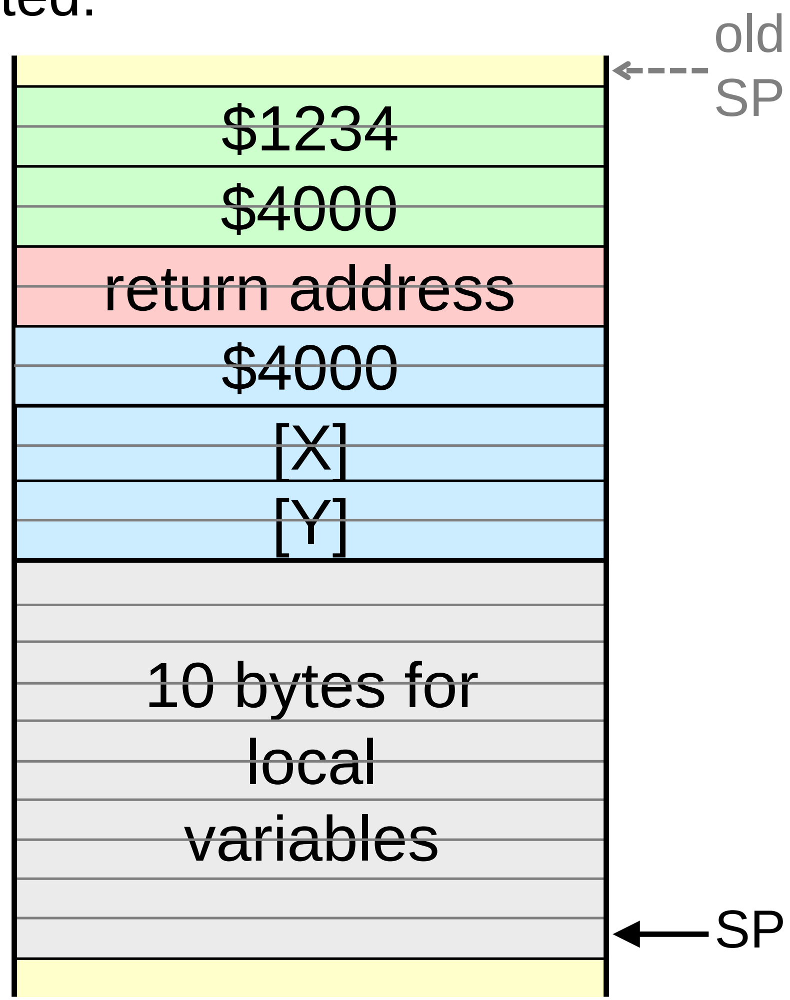
- Stack is used heavily in subroutine calls
- Stack may hold parameters, return address, saved registers and local variables
- Stack frame for a subroutine is sometimes called activation record
- All parameters and variables can be accessed within the subroutine using SP as an index register



Stack Frame Example

- Draw the stack frame for the following program segment after the leas instruction is executed:

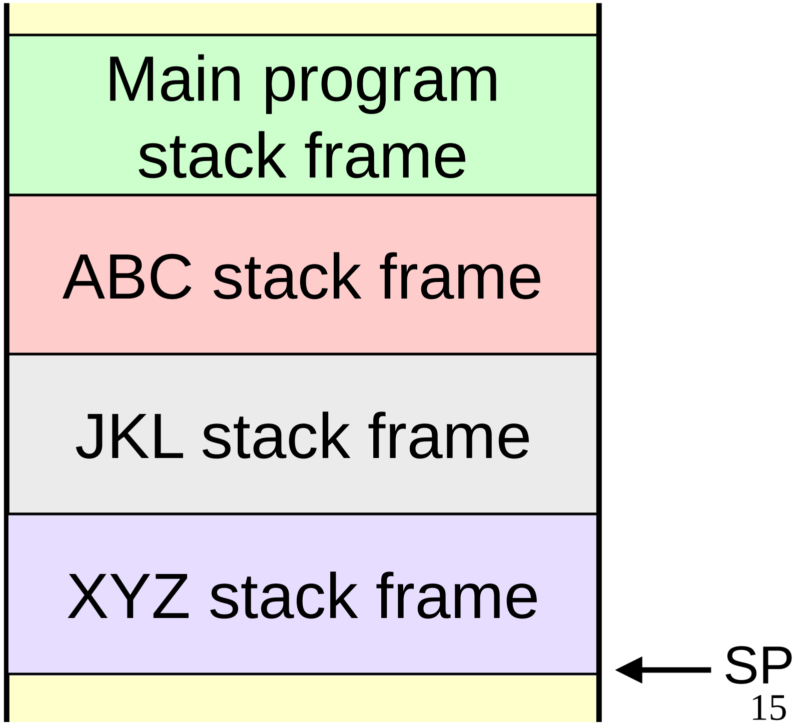
```
          ldd     #$1234      ;param1
          pshd
          ldd     #$4000      ;param2
          pshd
          jsr     sub_xyz
          …
          …
sub_xyz:  pshd                ;save regs
          pshx
          pshy
          leas    –10,sp      ;space for
          …                   ;local variables
```



Stack History

- The stack tells a story of the history of subroutine calls that got us to the current state of the program
- Suppose main program calls subroutine ABC. Subroutine ABC then calls subroutine JKL which, in turn calls subroutine XYZ
- While XYZ is running, stack will look like:

In a large program in which there is a complex hierarchy of subroutine calls, the stack will advance (downwards) and retreat (upwards) as subroutines are called (and returned)

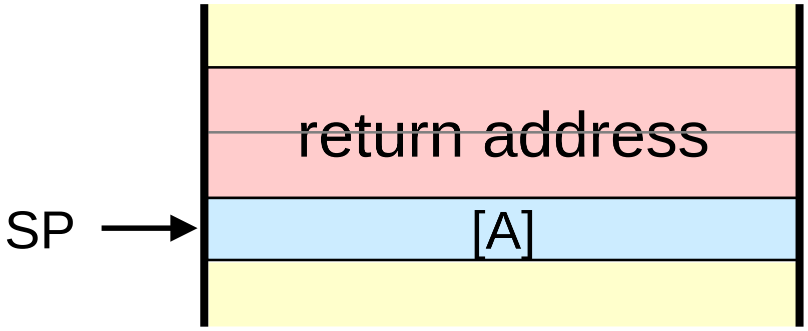


Example: Subroutine with saved registers

Write a subroutine that counts the number of negative values in an array of 8-bit signed integers. A pointer to the array is passed in Y. The number of elements in the array is passed in B. The answer should be returned in B (overwriting the total number of elements). Save any registers used (other then Y and B).

Solution: Use Y as a data pointer, B as a loop counter and accumulate count of negative values in A. Will need to save accumulator A

Stack Frame:



Counting Negative Values Example (cont.)

```
cnt_neg:  psha                ; save A accumulator
          clra                ; set count = 0
nloop:    tst     1, Y+       ; test sign of data
          bpl     skip        ; if positive do nothing
          inca                ; increment count
skip:     dbne    B, nloop    ; done yet?
          tfr     A, B        ; yes, place result in B
          pula                ; restore A accumulator
          rts
```

Counting Negative Values Example (cont.)

- To use this subroutine:

```
        ORG     $1000
array:  dc.b    4, 15, -87, 44, -3, -29, 33, 104
result: ds.b    1
        ORG     $4000
        lds     #$5000          ; set up stack pointer
        ldy     #array          ; set Y to point to array
        ldab    #8              ; no. of array elements in B
        jsr     cnt_neg         ; call subroutine to count neg values
        stab    result          ; result in B
        bgnd
```

Example: Subroutine with local variables

Write a subroutine that converts a decimal ascii string to a 16-bit signed binary number and leaves result in D. A pointer to the string is passed in X. If error (non-decimal digit) is detected, set X=0. Save any registers used (other then X and D).

Solution: Use local variables sign, number and temp

Step1: initialize sign=number=0

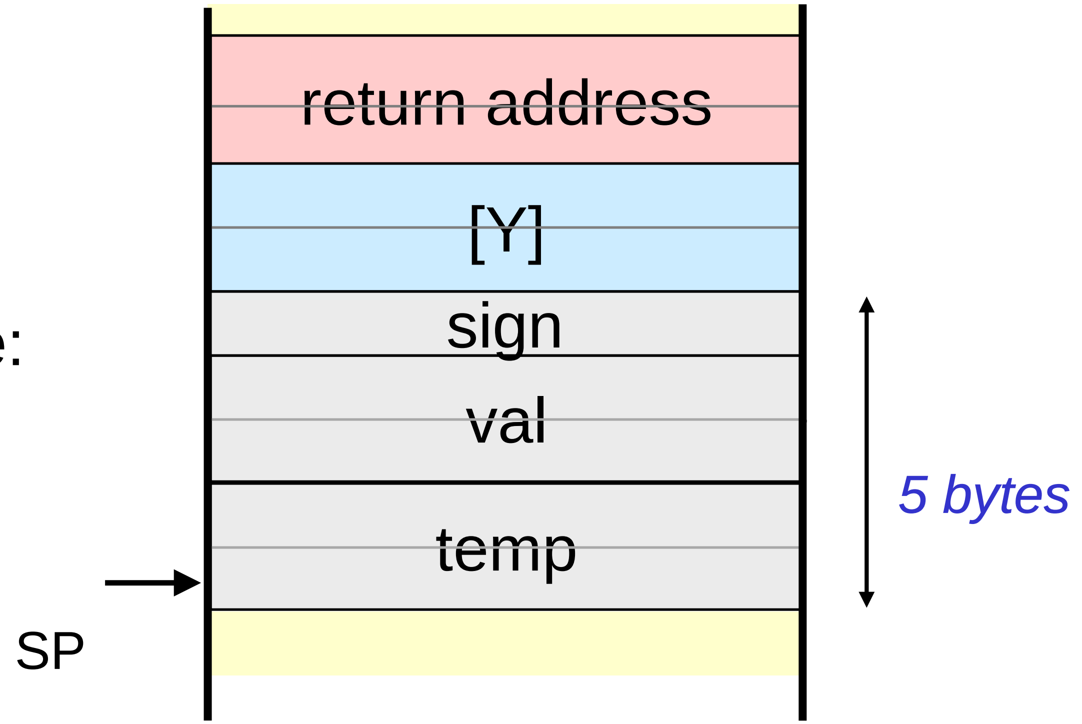
Step2: if m[ptr] is '–', then sign=1, increment ptr

Step3: if m[ptr] is NULL, go to step 4
else if m[ptr] is not decimal digit (0-9) then set X=0 and return
else number = number X 10 + (m[ptr] - $30)
increment ptr, go to step 3

Step4: if sign=1, number = twos complement (number)
set D=number and return

Ascii to Binary Example (cont.)

- First define stack frame:




```
minus:    equu    $2D
temp:     equ     0        ;stack offset of temp
val:      equ     2        ;stack offset of val
sign      equ     4        ;stack offset of sign
```

Ascii to Binary Example (cont.)

```
dec2bin:  pshy                      ;save Y register
          leas    –5, SP            ;allocate 5 bytes for local storage
          movw    #0, val, SP       ;initialize value to 0
          clr     sign, SP          ;initialize sign to 0
          ldaa    0, X              ;get first character
          cmpa    #minus            ;is first character a ‘–’ sign?
          bne     dloop
          inc     sign, SP          ;set sign=1
          inx                       ;update pointer
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
dloop:    ldab    1, X+             ;is next character a NULL?
          lbeq    done              ;we are at end of string
          cmpb    #$30              ;is character less than ascii 0?
          blo     cherr
          cmpb    #$39              ;is character greater than ascii 9?
          bhi     cherr
          subb    #$30              ;convert digit to binary
          clra                      ;set A=0 to make 16-bit quantity
          std     temp, SP          ;temporarily store 16-bit value of this digit
```

Ascii to Binary Example (cont.)

```
          ldd       val, SP       ; current accumulated value
          ldy       #10
          emul                    ; Y:D = [Y] x [D] (16x16 mult)
          addd      temp, SP      ; add current digit value
          std       val, SP       ; save the new value
          bra       dloop         ; move on to next digit
cherr:    ldx       #0            ; set the error condition
          bra       dealloc
done:     ldd       val, SP       ; put result in D
          tst       sign, SP      ; check sign intended sign of result
          beq       dealloc       ; if number is positive, we’re done
          ldd       #0            ; if number is negative…
          subd      val, SP       ; form twos complement in D
dealloc:  leas      6, SP         ;de-allocate local variables
          puly                    ;restore Y register
          rts                     ;return to caller
```

Ascii to Binary Example (cont.)

- To use this subroutine:

```
            ORG     $800
dec_str:    dc.b    "–4723",0
result:     ds.w    1
            ORG     $4000
            lds     #$6000
            ldx     #dec_str
            jsr     dec2bin
            std     result
            bgnd
```

Subroutine to return mean

- Write a subroutine to return the mean (average) of two 16-bit signed quantities. The two parameters are passed to the subroutine by loading their value on the stack immediately prior to the subroutine call. The result should be placed in register X. Save and restore any registers used by the subroutine.
- A calling sequence may look like:

```
ldx     valA
pshx
ldx     valB
pshx
jsr     average
stx     mean
```

Programmed Delay

- Sometimes we may want the microprocessor to wait for a specified period of time before executing next operation.
- Many HCS12 instructions execute in a predetermined number of clock cycles
  - i.e. for a given clock frequency take a known fixed time to execute
- We measure instruction execution time in terms of bus clock (E-clock) cycles
  - bus frequency is half that of PLL clock
- Create a known time delay in two steps:
  1. Select a sequence of instructions that takes known time to execute
  2. Repeat instruction sequence a number of times to generate required delay
- For example, sequence on following slide takes 40 E-cycles to execute

40 E-cycle delay loop

```
tloop:  psha                ;2 E-cycles
        pula                ;3 E-cycles
        psha
        pula
        psha
        pula
        psha
        pula
        psha
        pula
        psha
        pula
        psha
        pula
        nop                 ;1 E-cycle
        nop                 ;1 E-cycle
        dbne    x, tloop    ;3 E-cycles
```

- **psha** and **pula** are stack instructions
- Lab. EVB has E-Clock = 8 MHz, each E-clock period is 125 ns.
- Each iteration through loop takes 5 μs
- By entering loop with X initialized to $20{,}000_{10}$, we create a delay of 100 ms.
- Longer delays can be created by nesting this loop within a second (outer) loop that repeats the 100 ms sequence a specified number of times

Example: Time Delay Subroutine

- This routine delays by a multiple of 100 ms (assuming a 8 MHz E-clock).
- The multiple is passed as a parameter in register Y. xloop is the “inner” 100ms timing loop. yloop is the outer parameterized loop.

```
delayby100ms:
        pshx            ; save X
yloop:  ldx     #20000  ;2 E-cycles
xloop:  psha            ;2 E-cycles
        pula            ;3 E-cycles
        psha
        pula
        psha
        pula
        psha
        pula
```

```
        psha
        pula
        psha
        pula
        psha
        pula
        nop             ;1 E-cycles
        nop             ;1 E-cycles
        dbne    x, xloop ;3 E-cycles
        dbne    y, yloop ;3 E-cycles
        pulx            ;restore X
        rts
```