Stack and Subroutines
Stack

• A user-defined area of RAM
• Used for:
  – Temporary data storage
  – Subroutine calls and parameter passing
• SP points to *last used* location (top element)
  – SP (stack pointer register)
  – **LIFO** data structure ("last in, first out")

- Main operations:
  – “Push” data onto stack (like putting a plate on top of a stack of plates)
  – “Pull” data off stack

- SP fills low addresses
Machine Code Stack Instructions

• There are push and pull instructions for each register
  – “push” means: decrement the SP and store a byte on the stack
  – “pull” means: read a byte from the stack and increment SP
  – Of course, pushing and pulling a 16-bit register means that you store or read two bytes (and increment or decrement SP by 2)

• Example instructions

  psha  | decrement the SP by 1, then store A to memory at that location

  pula  | load A from memory at location pointed to by SP, then increment SP by 1

  pshx  | decrement the SP by 2, then store X to the two bytes of memory starting at that location

  pulx  | load X from the two bytes of memory starting at the location pointed to by SP, then increment SP by 2
Example

- Assume SP contains $4000, A contains $23, X contains $4567

\[
\begin{array}{|c|}
\hline
Initial \hspace{2cm} after \ psha \hspace{2cm} after \ pshx \hspace{2cm} after \ pulx \hspace{2cm} after \ pula \\
\hline
A \hspace{1cm} $23 \hspace{1cm} A \hspace{1cm} $23 \hspace{1cm} A \hspace{1cm} $23 \hspace{1cm} A \hspace{1cm} $23 \\
X \hspace{1cm} $4567 \hspace{1cm} X \hspace{1cm} $4567 \hspace{1cm} X \hspace{1cm} $4567 \hspace{1cm} X \hspace{1cm} $4567 \\
SP \hspace{1cm} $4000 \hspace{1cm} SP \hspace{1cm} $3FFF \hspace{1cm} SP \hspace{1cm} $3FFD \hspace{1cm} SP \hspace{1cm} $4000 \\
\hline
4000 \hspace{2cm} 4000 \hspace{2cm} 4000 \hspace{2cm} 4000 \hspace{2cm} 4000 \\
3FFF \hspace{2cm} 3FFF \hspace{2cm} 3FFF \hspace{2cm} 3FFF \\
\hline
3 FF D  \hspace{2cm} 3 FF D  \hspace{2cm} 45 \hspace{2cm} 3 FF D  \hspace{2cm} 45 \\
3 FF E  \hspace{2cm} 3 FF E  \hspace{2cm} 67 \hspace{2cm} 3 FF E  \hspace{2cm} 67 \\
3 FF F  \hspace{2cm} 3 FF F  \hspace{2cm} 23 \hspace{2cm} 3 FF F  \hspace{2cm} 23 \\
4000 \hspace{2cm} 4000 \hspace{2cm} 4000 \hspace{2cm} 4000 \\
\hline
\end{array}
\]

What if you did the “pulls” in reverse order (i.e., pula, pulx)?
More on Stack Instructions

• You are supposed to pull off values in the reverse order that you pushed them on
• Should only access stack using push and pull instructions

• Every program should initially load the stack pointer register with the “lds” instruction
  – Example:
    
    lds #RAMEND+1 ; load SP with the address RAMEND+1

• This makes SP point to a location where there is no physical memory! Why is this not a problem?
Use of stack for temporary data storage

- Sometimes you use a variable inside a function, but don’t need it anywhere else
- C example

```c
int sumTable(int table[], int N)
{
    int i, sum = 0;
    for (i=0; i<N; i++)
        sum += table[i];
    return sum;
}
```

- C stores these temporary variables \((i, \text{sum})\) on the stack
Global vs. Local Variables

• Global variables
  – Are defined outside “main”
  – Can be accessed from any function
  – Are stored in fixed locations in RAM

• Local variables
  – Are defined inside “main” or other functions
  – Only accessible within that function
  – Are stored on the stack
  – They disappear when function exits
Example

#include <hidef.h>    /* common defines and macros */
#include "derivative.h"    /* derivative-specific definitions */

int global;       // a global variable

void main(void) {
    unsigned char c;       // a local variable

    EnableInterrupts;

    global = 1;
    c = 0;

    for(;;) {
        global++;
        c++;

        _FEED_COP();    /* feeds the dog */
    } /* loop forever */
}
Variable "global" is stored in location 0x900

Variable "c" is stored on the stack
Subroutines and Functions

• A re-usable piece of code, logically separate and independent

• Example
  – A subroutine to delay 1 msec
  – More compact than replicating that module wherever you need it

• Machine code instructions for subroutines:
  – jsr <opr>
    • “jump to subroutine”
    • Actions:
      – Push PC (current program counter) onto stack; this is the address of the instruction following the jsr instruction
      – Then load PC with operand (which is the starting address of the subroutine)
  – rts
    • “return from subroutine”
    • Action:
      – Pull PC from stack; this means the next instruction will be the one that followed the jsr instruction
Example

- **Main program (loaded at $C000)**
  
  ```
  lds #$4000
  jsr sub1
  inca
  ```

- **Subroutine (loaded at $C800)**
  
  ```
  sub1 psha
  :
  :  --- Some code that uses A
  :  --- Restore the value of A before returning
  pula
  rts
  ```

- **What if you forgot to do the pula before rts?**

```asm
lds #$4000
jsr sub1
inca
```
C Function Example

#include <hidef.h>     /* common defines and macros */
#include "derivative.h" /* derivative-specific definitions */

/* Return the absolute value of the input number. */
char myAbsoluteValue(char x)
{
    if (x < 0)
        return -x;
    else
        return x;
}

void main(void)
{
    char m, n;

    EnableInterrupts;

    m = -3;
    n = myAbsoluteValue(m);

    for(;;) ;  /* loop forever */
}
5: char myAbsoluteValue(char x)
6: {
7:     if (x < 0)
8:         return -x;
9:     else
10:        return x;
11: }

Subroutine
- Assumes that the input parameter is in register B
- The result is also returned in B

12:

13:

14: void main(void)
15: {

Main program

Allocate space for m,n on stack

Store value of m on stack

Store value of n on stack

16:     char m, n;
17:
18:     EnableInterrupts;
19:     CLI
20:     m = -3;

21:     n = myAbsoluteValue(m);
22:     BSR myAbsoluteValue

23:     for(;;) ; /* loop forever */
24: }

Look at contents of stack when subroutine is called
Summary / Questions

• The “stack” is a user-defined area of RAM. The “stack pointer” is a CPU register that points to (pick one) __________ location of the stack
  – (a) next empty
  – (b) last filled

• Name some machine code instructions that use the stack

• As the stack fills, does it grow down or up?