STACK & SUBROUTINES

I. Concepts
- A subroutine is a unit of code that performs a well-defined sub-function of an overall program.
- One purpose of subroutines is to support modular code development and organization, i.e. division of problem complexity; ex: automobile fuel injection system. Vis:

```
Main
\[\begin{array}{c}
\text{Sub-1} \\
\text{ex: read sensors} \\
\text{Sub-2} \\
\text{ex: do calculations} \\
\text{Sub-3} \\
\text{ex: control actuators}
\end{array}\]
```

- Another purpose is to support code reuse; ex: a square root calculation routine.
- To implement a subroutine mechanism, the processor will require use of a stack.
- Stack: a special area of RAM reserved for use by the processor to temporarily save register values for later retrieval.

II. The Stack
- A stack is a data structure with a top and bottom, similar to an array except that data can only be put into or taken from the stack on one end; hence referred to as a LIFO structure. An analogy here is a stack of plates at the cafeteria where plates are added to and taken from the stack one-at-a-time.
- Adding a new element onto the top of the stack is called a push, removing an element is called a pull or pop. This is typically done with push/pull instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSHA</td>
<td>push content of AccA to stack</td>
</tr>
<tr>
<td>PULA</td>
<td>pull from stack to A</td>
</tr>
<tr>
<td>PSHB</td>
<td>push AccB to stack</td>
</tr>
<tr>
<td>PULB</td>
<td>pull from stack to B</td>
</tr>
<tr>
<td>PSHX</td>
<td>push X to stack (lo byte first, hi byte second)</td>
</tr>
<tr>
<td>PULX</td>
<td>pull X from stack (hi byte first, lo byte second)</td>
</tr>
<tr>
<td>PSHY</td>
<td>push Y to stack (lo, hi)</td>
</tr>
<tr>
<td>PULY</td>
<td>pull Y from stack (hi, lo)</td>
</tr>
</tbody>
</table>

- Processor architectures include a special register that is used as a pointer to the Top of Stack (TOS). So instead of moving all the elements in the stack to make room for a new element (highly inefficient), the processor simply modifies the value of the pointer register (highly efficient). The HC11’s SP register serves as its TOS pointer.
- To preserve integrity of the stack, the HC11 automatically decrements the SP after each byte is pushed to the stack and automatically increments the SP before each byte is pulled. This is referred to post-decrement / pre-increment behavior. Thus, we can say that the HC11’s SP always points to the next available location. Other processors might be the other way ‘round (pre-decrement / post-increment).
Before using the stack, the SP must be initialized by “pointing” it to a valid area of RAM. On the HC11, this is done with a load stack pointer (LDS) instruction. Note: evaluation boards running a monitor such as BUFFALO typically do this for its own use; in lab we use BUFFALO’s stack.

Ex: analyze the following code sequence and determine the final values of the accumulators and SP; also track stack contents during execution.

1 * Initialize SP register to valid RAM
2 LDS #$1FFF
3 * load accumulators with sample values
4 LDAA #$22
5 LDAB #$33
6 * store data from accumulators to stack
7 PSHA
8 PSHB
9 * retrieve data values from stack
10 PULA
11 PULB
12 * end program
13 SWI

Stack Usage Tips:

a. always ensure stack operations are balanced! (# pushes = # pulls)
b. observe byte ordering when performing 16-bit transfers!

Thought question: what would happen if the above tips were overlooked?

III. Subroutines

In order to call a subroutine, the CPU must remember where the subroutine is being called from for proper return at the end of the subroutine. This is easily accomplished by pushing the PC to the stack before transferring the subroutine. The HC11 implements this via the branch to subroutine (BSR) or jump to subroutine (JSR) instruction. BSR uses relative addressing while JSR uses direct, extended or indexed. In a program, the BSR or JSR mnemonic is followed by the starting address of the subroutine. Ex:

BSR $150 ;branch to subroutine starting at $150 (must be within 127 bytes reach)
JSR $FFB8 ;calls “OUTA” subroutine within BUFFALO (can be anywhere in memory map)

To return from a subroutine, the CPU must execute a return from subroutine (RTS) instruction, which simply retrieves (pulls) the PC from the top 2 bytes of the stack. RTS uses inherent addressing.

So, the subroutine mechanism on the HC11 may be summarized as follows:

1. the current PC value (next opcode address following the BSR/JSR, called the return address) is pushed onto the stack
2. the address of the subroutine as specified by the instruction is loaded into the PC
3. subroutine execution proceeds as normal until the required RTS at the end of the subroutine is encountered
4. the RTS causes the previously saved PC value to be pulled from the stack and returned to the PC (thus returning to the instruction following the BSR/JSR)

[need example here, but there is a BUFFALO example below]
Subroutine Tips:

a. **always** call a subroutine with a BSR or JSR instruction, **never** BRA or JMP to a subroutine!

b. **always** end a subroutine with a RTS instruction, **never** BRA or JMP out of a subroutine!

c. there is no such thing as a “jsr immediate”, so **never** use ‘#’ with a JSR!

d. **make sure** the SP points to valid stack area in RAM before calling subroutines!

e. subroutine calls may be **nested** as long as adequate stack space exists!

Thought question: what would happen if the above tips were overlooked?

IV. BUFFALO Subroutines

- Besides writing your own subroutines, resident monitors such as BUFFALO may also provide useful subroutines that may be called by any user program.

- Ex: subroutine “OUTA” which begins at $FFB8 outputs the ASCII character code in AccA to the terminal/user. This could be used to output the 2-letter message “Hi” as follows:

```
1     LDA A #$48 ; load A with ASCII 'H'
2     JSR $FFB8 ; output to user
3     LDA A #$69 ; load A with ASCII 'i'
4     JSR $FFB8 ; output to user
5     SWI
```

- The following table summarizes the useful subroutines provided in the BUFFALO monitor. ref: “BUFFALO Monitor” (PDF), pg. 25.

<table>
<thead>
<tr>
<th>Subroutine Name</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPCASE</td>
<td>$FFA0</td>
<td>If character in accumulator A is lower case alpha, convert to upper case.</td>
</tr>
<tr>
<td>WCHEK</td>
<td>$FFA3</td>
<td>Test character in accumulator A and return with Z bit set if character is whitespace (space, comma, tab).</td>
</tr>
<tr>
<td>DCHEK</td>
<td>$FFA6</td>
<td>Test character in accumulator A and return with Z bit set if character is delimiter (carriage return or whitespace).</td>
</tr>
<tr>
<td>INIT</td>
<td>$FFA9</td>
<td>Initialize I/O device.</td>
</tr>
<tr>
<td>INPUT</td>
<td>$FFAC</td>
<td>Read I/O device.</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>$FFAF</td>
<td>Write I/O device.</td>
</tr>
<tr>
<td>OUTLHLF</td>
<td>$FFB2</td>
<td>Convert left nibble of accumulator A contents to ASCII and output to terminal port.</td>
</tr>
<tr>
<td>OUTRHLF</td>
<td>$FFB5</td>
<td>Convert right nibble of accumulator or A contents to ASCII and output to terminal port.</td>
</tr>
<tr>
<td>OUTA</td>
<td>$FFB8</td>
<td>Output accumulator A as ASCII character.</td>
</tr>
<tr>
<td>OUT1BYT</td>
<td>$FFB8</td>
<td>Convert binary byte at address in index register X to two ASCII characters and output. Returns address in index register X pointing to next byte.</td>
</tr>
<tr>
<td>OUT1BSP</td>
<td>$FFBE</td>
<td>Convert binary byte at address in index register X to two ASCII characters and output followed by a space. Returns address in index register.</td>
</tr>
<tr>
<td>OUT2BSP</td>
<td>$FFC1</td>
<td>Convert two consecutive binary bytes starting at address in index register X to four ASCII characters and output followed by a space. Returns address in index register X pointing to next byte.</td>
</tr>
<tr>
<td>OUTCRLF</td>
<td>$FFC4</td>
<td>Output ASCII carriage return followed by a line feed.</td>
</tr>
<tr>
<td>OUTSTRG</td>
<td>$FFC7</td>
<td>Output string of ASCII bytes pointed to by address in index register X until character is an End Of Transmission ($04).</td>
</tr>
<tr>
<td>OUTSTRG0</td>
<td>$FFCA</td>
<td>Same as OUTSTRG except leading carriage return and line feed is skipped.</td>
</tr>
<tr>
<td>INCHAR</td>
<td>$FFCD</td>
<td>Input ASCII character to accumulator A and echo back. This routine loops until character is actually received.</td>
</tr>
</tbody>
</table>