“I’ve traveled the length and breadth of this country and talked with the best people, and I can assure you that data processing is a fad that won’t last out the year.”
Editor in charge of business books for Prentice Hall, 1957

3.3.5. Subroutines and the stack
classical definition of the stack
• push saves data on the top of the stack,
• pull removes data from the top of the stack
• stack implements last in first out (LIFO) behavior

many uses of the stack
• temporary calculations
• subroutine (function) return addresses
• subroutine (function) parameters
• local variables

Initially, the stack is empty

lds #$0C00 for MC68HC812A4 (book)
lds #$4000 for 9S12C32 (class, lab)
ldaa #1
psha ; push 1 on the stack

ldaa #2
psha ; push 2 on the stack
ldaa #3
psha ; push 3 on the stack

At this point if one were to pull from the stack execute **pula**, the 3 would be returned
the 2 would now be on the top of the stack
The push and pull instructions

- **psha**: push Register A on the stack
- **pshb**: push Register B on the stack
- **pshx**: push Register X on the stack
- **pshy**: push Register Y on the stack
- **des**: $S = S - 1$ (reserve space)
- **pula**: pull from stack into A
- **pulb**: pull from stack into B
- **pulx**: pull from stack into X
- **puly**: pull from stack into Y
- **ins**: $S = S + 1$ (discard top of stack)

The following are important instructions

- **tsx**: transfer $S$ to X
- **tsy**: transfer $S$ to Y
- **txs**: transfer X to $S$
- **tys**: transfer Y to $S$

we use the term **subroutine** all functions or procedures
- whether or not they return a value
• develop modular software
• called by either bsr or jsr
• subroutine returns using rts

```
org $4000
main lds #$4000 ; initialize stack
clra
loop bsr sub    ; branch to subroutine
bra loop

; Purpose: increment a number
; Input:  RegA, range 0 to 255
; Output: RegA=Input+1
; Errors: Will overflow if input is 255
sub inca        ; adds one to Input
rts

org $fff6
fdb main
```

Program 3.1. Simple program showing how to use the bsr and rts instructions to implement a subroutine.

We begin the study by looking at the listing file.

```
$4000                           org  $4000
$4000 CF4000 [2]( 0){OP   }main lds #$4000
$4003 87     [1]( 2){O    }     clra
$4004 0702   [4]( 3){PPPS }loop bsr sub
$4006 20FC   [3]( 7){PPP  }     bra  loop
$4008 42     [1](10){O    }sub  inca
```
Execute the \texttt{lds} instruction

\textbf{Opcode fetch} \hspace{1cm} \textbf{Operand fetch}

\begin{tabular}{ll}
\texttt{lds} & \texttt{R 0x4000 0xCF} from EEPROM \\
\texttt{R 0x4001 0x40} from EEPROM \\
\texttt{R 0x4002 0x00} from EEPROM
\end{tabular}

Execute the \texttt{clra} instruction

\textbf{Opcode fetch} \hspace{1cm} \textbf{Operand fetch}

\begin{tabular}{ll}
\texttt{clra} & \texttt{R 0x4003 0x87} from EEPROM
\end{tabular}

Execute the \texttt{bsr} instruction

\textbf{Opcode fetch} \hspace{1cm} \textbf{Operand fetch} \hspace{1cm} \textbf{Stack store lsbW} \hspace{1cm} \textbf{Stack store msbW}

\begin{tabular}{llll}
\texttt{bsr} & \texttt{R 0x4004 0x07} from EEPROM & \texttt{R 0x4005 0x02} from EEPROM & \texttt{0x3FFF 0x06} to RAM & \texttt{0x3FFE 0x40} to RAM
\end{tabular}

\textbf{Figure 3.18. The stack before and after execution of the \texttt{bsr} instruction.}

Execute the \texttt{inca} instruction

\textbf{Opcode fetch} \hspace{1cm} \textbf{Operand fetch}

\begin{tabular}{ll}
\texttt{inca} & \texttt{R 0x4008 0x42} from EEPROM
\end{tabular}
Execute the \texttt{rts} instruction

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{stack.png}
\caption{The stack before and after execution of the \texttt{rts} instruction.}
\end{figure}

3.3.6. Branch operations

\begin{verbatim}
bcc  place  go if C=0
bc\text{\textbar}s  place  go if C=1
beq  place  go if Z=1
bne  place  go if Z=0
bmi  place  go if N=1
bpl  place  go if N=0
bvc  place  go if V=0
bvs  place  go if V=1
bra  place  go always
brn  place  go never
jmp  place  go always, ext addr
\end{verbatim}

must follow a subtract compare or test instruction, such as

\begin{verbatim}
subs  subb  sbca  sbcb  subd
cba  cmpa  cmpb  cpd
tsta  tstb  tst
\end{verbatim}
signed branches, branch if
   \textbf{bge} \textit{place} \quad \text{greater than or equal to}
   \begin{align*}
   &\text{if} \quad (N^\wedge V)=0 \\
   &\text{i.e.,} \quad (~N\cdot V+N\cdot ~V)=0
   \end{align*}
   \begin{align*}
   &\textbf{bgt} \textit{place} \quad \text{greater than} \\
   &\text{if} \quad (Z+N^\wedge V)=0 \\
   &\text{i.e.,} \quad (Z+~N\cdot V+N\cdot ~V)=0
   \end{align*}
   \begin{align*}
   &\textbf{ble} \textit{place} \quad \text{less than or equal to} \\
   &\text{if} \quad (Z+N^\wedge V)=1 \\
   &\text{i.e.,} \quad (Z+~N\cdot V+N\cdot ~V)=1
   \end{align*}
   \begin{align*}
   &\textbf{blt} \textit{place} \quad \text{less than} \\
   &\text{if} \quad (N^\wedge V)=1 \\
   &\text{i.e.,} \quad (~N\cdot V+N\cdot ~V)=1
   \end{align*}

unsigned branches, branch if
   \begin{align*}
   &\textbf{bhs} \textit{place} \quad \text{greater than or equal to} \\
   &\text{if} \ C=0, \text{ same as } \textbf{bcc} \\
   &\textbf{bhi} \textit{place} \quad \text{greater than} \\
   &\text{if} \ C+Z=0 \\
   &\textbf{blo} \textit{place} \quad \text{less than} \\
   &\text{if} \ C=1, \text{ same as } \textbf{bcs} \\
   &\textbf{bls} \textit{place} \quad \text{less than or equal to} \\
   &\text{if} \ C+Z=1
   \end{align*}

it is important to know
\begin{itemize}
   \item \textbf{precision} (e.g., 8-bit, 16-bit)
   \item \textbf{format} (e.g., unsigned, signed)
\end{itemize}

It takes three steps
\begin{enumerate}
   \item read the first value into a register
\end{enumerate}
2. compare the first value with the second value
3. conditional branch

When testing for equal or not equal
• doesn’t matter whether signed or unsigned
• still matters if 8-bit or 16-bit
• doesn’t matter about load and compare order

8-bit if–then compare to zero examples
Assume \( G \) is an 8-bit global variable, signed or unsigned

<table>
<thead>
<tr>
<th>C code</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(G == 0){</td>
<td>\textcolor{blue}{\textbf{ldaa G}}</td>
</tr>
<tr>
<td></td>
<td>\textcolor{blue}{\textbf{bne next}}</td>
</tr>
<tr>
<td></td>
<td>\textcolor{blue}{\textbf{next}}</td>
</tr>
<tr>
<td>if(G != 0){</td>
<td>\textcolor{blue}{\textbf{ldaa G}}</td>
</tr>
<tr>
<td></td>
<td>\textcolor{blue}{\textbf{beq next}}</td>
</tr>
<tr>
<td></td>
<td>\textcolor{blue}{\textbf{next}}</td>
</tr>
</tbody>
</table>

8-bit if–then signed compare to zero examples
Assume \( sG \) is a signed 8-bit global variable

<table>
<thead>
<tr>
<th>C code</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(sG &gt;= 0){</td>
<td>\textcolor{blue}{\textbf{ldaa sG}}</td>
</tr>
<tr>
<td></td>
<td>\textcolor{blue}{\textbf{bmi next}}</td>
</tr>
<tr>
<td></td>
<td>\textcolor{blue}{\textbf{next}}</td>
</tr>
<tr>
<td>if(sG &lt; 0){</td>
<td>\textcolor{blue}{\textbf{ldaa sG}}</td>
</tr>
<tr>
<td></td>
<td>\textcolor{blue}{\textbf{bpl next}}</td>
</tr>
<tr>
<td></td>
<td>\textcolor{blue}{\textbf{next}}</td>
</tr>
</tbody>
</table>
8-bit if–then compare examples
Assume \( G_1 \ G_2 \) are 8-bit global variables, signed or unsigned

<table>
<thead>
<tr>
<th>C code</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(G2 == G1){</td>
<td>1daa G2</td>
</tr>
<tr>
<td>???</td>
<td>cmpa G1</td>
</tr>
<tr>
<td>bne next</td>
<td>???</td>
</tr>
<tr>
<td>}</td>
<td>next</td>
</tr>
<tr>
<td>if(G2 != G1){</td>
<td>1daa G2</td>
</tr>
<tr>
<td>???</td>
<td>cmpa G1</td>
</tr>
<tr>
<td>beq next</td>
<td>???</td>
</tr>
<tr>
<td>}</td>
<td>next</td>
</tr>
</tbody>
</table>

Compare 8-bit versus 16-bit conditionals
Assume \( G_1 \ G_2 \) are 8-bit global variables, signed or unsigned
Assume \( H_1 \ H_2 \) are 16-bit global variables, signed or unsigned

<table>
<thead>
<tr>
<th>C code</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(G2 == G1){</td>
<td>1dab G2</td>
</tr>
<tr>
<td>???</td>
<td>subb G1</td>
</tr>
<tr>
<td>bne next</td>
<td>???</td>
</tr>
<tr>
<td>}</td>
<td>next</td>
</tr>
<tr>
<td>if(H2 == H1){</td>
<td>1dy H2</td>
</tr>
<tr>
<td>???</td>
<td>cpy H1</td>
</tr>
<tr>
<td>bne next</td>
<td>???</td>
</tr>
</tbody>
</table>
Common error. It is an error to use an 8-bit comparison to test two 16-bit values.

it is important to know

- **precision** (e.g., 8-bit, 16-bit)
- **format** (e.g., unsigned, signed)
  - unsigned, **bhi blo bhs** and **bls**
  - signed, **bgt bls bge** and **ble**

It takes three steps

1. read the first value into a register
2. compare the first value with the second value
3. conditional branch

**Compare the four possible inequalities**

Assume uG is a unsigned 8-bit global variable

<table>
<thead>
<tr>
<th>C code</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(uG &gt; 5){</td>
<td></td>
</tr>
<tr>
<td>?? ?? ?</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>ldau uG</td>
<td></td>
</tr>
<tr>
<td>cmpa #5</td>
<td></td>
</tr>
<tr>
<td>bls next</td>
<td></td>
</tr>
<tr>
<td>?? ?? ??</td>
<td></td>
</tr>
</tbody>
</table>

| if(uG >= 5){           |                                   |
|     ?? ?? ?            |                                   |
| }                      |                                   |
| ldau uG                 |                                   |
| cmpa #5                 |                                   |
| blo next                |                                   |
| ?? ?? ??               |                                   |

| if(uG < 5){            |                                   |
|     ?? ?? ?            |                                   |
| }                      |                                   |
| ldau uG                 |                                   |
| cmpa #5                 |                                   |
| bhs next                |                                   |
| ?? ?? ??               |                                   |

| if(uG <= 5){           |                                   |
|     ?? ?? ??           |                                   |
| }                      |                                   |
| ldau uG                 |                                   |

Jonathan W. Valvano
Compare signed versus unsigned conditionals
Assume \texttt{uG} is an unsigned 8-bit global variable
Assume \texttt{sG} is a signed 8-bit global variable

<table>
<thead>
<tr>
<th>C code</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>\begin{verbatim} if(uG &gt;= 5){} \ ???? } \end{verbatim}</td>
<td>\begin{verbatim} ldaa uG \ cmpa #5 \ bhi next \ ??? ?? \ next \end{verbatim}</td>
</tr>
<tr>
<td>\begin{verbatim} if(sG &gt;= 5){} \ ???? } \end{verbatim}</td>
<td>\begin{verbatim} ldaa sG \ cmpa #5 \ blt next \ ??? ?? \ next \end{verbatim}</td>
</tr>
</tbody>
</table>

Compare 8-bit versus 16-bit conditionals
Assume \texttt{uG1} and \texttt{uG2} are unsigned 8-bit variables
Assume \texttt{uH1} and \texttt{uH2} are unsigned 16-bit variables

<table>
<thead>
<tr>
<th>C code</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>\begin{verbatim} if(uG2 &gt;= uG1){} \ ???? } \end{verbatim}</td>
<td>\begin{verbatim} ldaa uG2 \ cmpa uG1 \ blo next \ ??? ?? \end{verbatim}</td>
</tr>
</tbody>
</table>
if (uH2 >= uH1) {
  ???
}

ldd  uH2
cpd  uH1
blo  next
???
next

while (uG < 5) {
  ???
}

clr  uG
loop  ldax  uG
cmpa  #5
bhs  next ; stop when uG>=5
???  ; body of while loop
inc  uG
bra  loop

next
do{
    ?? ?? ?
}
while(uG < 5);

loop ?? ?? ? ; body of while loop
    ldaa uG
    cmpa #5
    blo  loop ; stop when G2<=G1
loop ?? ?? ? ; body of while loop
    ldaa sG
    cmpa #5
    blt  loop ; stop when G2<=G1
loop ?? ?? ? ; body of while loop
    ldd uH
    cpd #5
    blo  loop ; stop when G2<=G1
while(1){?? ?? ?
}

loop ?? ?? ?? ; body of while loop
    bra  loop