Program Structures

EE3376
Local Variables

- **Temporary Information Storage**
  - used by subroutines
  - used by interrupt service routines
  - used for short while in main thread

- **Allocated, Used, De-allocated**

- **Used only by the thread that created it**

- **Implemented in stack or in registers**

- **Examples:**
  - loop counter
  - temporary swap storage
  - intermediate result in larger calculation
Local Variables

• On stack
  - Push and Pop are provided in hardware for allocate/de-allocate
  - Efficient use of memory – de-allocation frees memory for later use
  - Code using local stacked variables is re-locatable
  - Code is reentrant as each instance has its own stack frame
  - Virtual unlimited amount of storage space
  - Limited scope – can not be seen by other threads (protected)

• In registers
  - Fastest data storage location in system
  - However limited amount – precious resource in microcontroller
Local Variables – swap example

; ****************************************************************************
; SWAP – subroutine that swaps contents of two registers
; - uses local variable implemented in R7 accumulator
; ****************************************************************************

swap:  push.w R7
       mov.w R5, R7
       mov.w R6, R5
       mov.w R7, R6
       pop.w R7
       ret
Local Variables – swap example

; SWAP – subroutine that swaps contents of two memory
; - uses local variable implemented with the stack

swap:   add.w -2, SP
        mov.w R5, 0(SP)
        mov.w R6, R5
        mov.w @SP, R6
        add.w 2, SP
        ret
Global Variables

- Permanent Data Storage
- Stored in memory
- Accessed with extended addressing
- Shared between threads

Should not be used:
  - for subroutines due to re-entrant requirements
  - temporary data due to wasted memory space
long int time;

void main () {
    long int temp;
    time = 0;
    temp = times_two(time);
}

long int times_two(long int x) {
    long int p;
    p = 2 * x;
    return(p);
}
The Stack in Detail

- Hardware assist for LIFO data structure
- Used for
  - local variables
  - parameter passing to subroutines
  - saving program counter in subroutines
  - saving all registers during interrupt handling
Stack Areas

Section of RAM

initial SP = 0x0280
Space never used

current SP

Non-RAM at 0x01FF

Allocated Stack

Free Stack

Total Stack Area
Stack Rules

1. Same number of pushes as pulls
   - within the total running life of the program
   - within a subroutine
   - within an interrupt handler

2. Accesses should remain in Stack area
   - causes either stack underflow or overflow
   - example: pull from an empty stack

3. Reads/Writes can not be done in free area
   - new stack entry should be allocated before use
Stack rule 3 violation

```assembly
mov.w  #0, R5
mov.w  R5, -2(SP); use of free space - illegal
```

Subsequent interrupt will overwrite cleared value and main program will be unaware.
Stack rule 3 violation corrected

```
mov.w   #0, R5
push    R5 ; allocate then use space
```

Now interrupt will start writing registers in location after pushed value.
some_function:
; bind index for pressure is 0
; bind index for temperature is 2
  add.w  -4, SP ; allocate space for both
  mov.w  #20, 0(SP) ; initialize pressure
  mov.w  #25, 2(SP) ; initialize temperature

use variables here

  add.w  4, SP ; deallocate space
  ret

NOTE: push and pop both allocate and initialize in one instruction.
Stack Frames

- Alternative approach to local variables
- Requires the use of stack frame pointer
  - frame pointer cannot be used for other purpose
- Requires that the SFP is saved/restored in subroutines called during life of stack frame
Stack Frame illustrated

Initial SP: 0x0300
Current SP

Section of RAM

Previous stack data
Stack Frame Allocation

Initial SP
0x0300

FSP

SP

Section of RAM

Previous stack data

mov.w SP, R7 ;create FSP
add.w -4, SP ;allocate
Stack Frame illustrated - use

initial SP 0x0300
FSP
SP

Section of RAM

speed .set -2
accel .set -4
mov.w #10, accel(R7)
mov.w accel(R7), R8
add.w #20, R8
mov.w R8, speed(R7)

acceleration
speed
Stack Frame illustrated – de-allocate

mov.w R7, SP

initial SP 0x0300

FSP = SP

Section of RAM
unsigned short calc (void) {
    unsigned short sum;
    unsigned short n;

    sum = 0;
    for (n=100; n>0; n--){
        sum = sum + n;
    }
    return sum;
}
Stack Frame Example

; binding and initialization

sum .set -2 ; bind index for sum
n .set -4 ; bind index for n

calc:
push R7 ; save old R7
mov.w SP, R7
add.w -4, SP
mov.w #0, sum(R7); initialize sum
mov.w #100, n(R7) ; initialize n
Stack Frame Example

; main section of subroutine

loop:  mov.w n(R7), R8    ; R8 = index n
       add.w sum(R7), R8 ; R8 = sum + n
       mov.w R8, sum(R7) ; sum = sum + n
       mov.w n(R7), R8    ; R8 = n
       add.w -1, R8       ; decrement R8
       mov.w R8, n(R7)    ; store in local var
       bne loop

       mov.w R8, sum(R7) ; store final value in D
                           ; as output parameter
Stack Frame Example

; de-allocation, restore and return

mov.w R7, SP
pull.w R7
ret
Parameter Passing in Assembly

- Parameter passing is communication to subroutines – both to and from
- Avoid using globals as parameters
  - if a subroutine calls itself and acts on a global var then corruption is possible
  - if unavoidable, make the access atomic
- Using registers is the easiest method
  - but registers are limited and precious
- Using the stack is the most versatile method
Subroutine to Wait (using registers)

;-----------------------------------------------
; Subroutine DELAY, Delay = (R8) * msec, no registers modified
; Call sequence (Parameters passed by register – Y in this case)
;    mov.w    #0x0010, R8
;    call    delay
;-----------------------------------------------
delay:  push   R7
starty: mov.w   #0x1000, R7 ; start R8 loop and set R7 loop 1 msec
startx: dec.w   R7 ; start R7 loop
             jne  startx
             dec.w  R8
             jne  starty
             pop   R7
ret

*****************************************************************************
Subroutine to Wait (using stack)

; Call sequence (Parameters passed by stack)
;   add.w -1, SP ; allocate space for parameter
;   mov.b #0x0010, SP
;   call delay
;   add.w 1, SP; allocate space for parameter

;------------------------------------------------------

delay:
   push R8 ; save existing value of X
   push R7
   mov.b 6(SP), R8 ; get input parameter

starty:
   mov.w #1000, R7 ; start y loop and set x loop to 1 msec

startx:
   add.w -1, R7 ; start x loop
   jne startx
   add.w -1, R8 ;
   jne starty
   pop R7
   pop R8
   ret
Parameter Passing in C (refresher)

- **Call by reference**
  - pass pointer to subroutine
  - subroutine can change the actual variable
  - even when data is large, reference is only an address – fast

- **Call by value**
  - pass copy of value to subroutine
  - changing copy does not change actual variable
  - safer as calling thread can place result in variable
  - slower if data is large
short T; // global variable

void main (void) { // main thread
    short V; // local var for main
    T = 5; //
    V = 6; //
    T = sub ( V ); // call by reference
    return(0);
}

short sub (short U) { // U and V are local variables
    short V;
    V = U + 1;
    return ( V );
}

Compiled C Program
Recursion Example

; recursive implementation of factorial
; register R5 is used as an input parameter n
; register R5 is used as output parameter fact(n)
; corrupts R6

```
fact    cmp.w   #1,   R5
        jeq    done
        push.w R5
        dec.w  R5
        call   fact
        pop.w  R6
        call   mul ; R5 = R5 * R6 (potential overflow)
done    ret
```
Finite State Machines

- in EE2369 – how to design a FSM as hardware
  - fast and small (much)
  - difficult to design, fabricate and to update
- in EE3376 – how to design a FSM in software
  - slower and larger, but very flexible

- Examples
  - elevator
  - vending machine
  - traffic light
  - sequence detector
Sequence Detector Example

- Very common interview question for design jobs
- One digital input and one digital output
  - input represents sequential list of 1’s and 0’s
  - output asserts if specific sequence is detected
- In this example, the prescribed sequence is 101

\[
\begin{align*}
  x &= 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \\
  z &= 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 
\end{align*}
\]
State graph for mealy detector

\[
\begin{align*}
X &= 0 0 1 1 0 1 1 0 0 1 0 1 0 1 0 0 \\
Z &= 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0
\end{align*}
\]
### State Table for sequence detector

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State x = 0</th>
<th>Next State x = 1</th>
<th>Output x = 0</th>
<th>Output x = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>s0</td>
<td>s1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s1</td>
<td>s2</td>
<td>s1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s2</td>
<td>s0</td>
<td>s1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

![State Diagram](image)
### Transition Table for sequence detector

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x = 0$</td>
<td>$x = 1$</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>01</td>
</tr>
</tbody>
</table>

You only need this for hardware design. In software, you’ll stop at the state table and then start the software.
Final Design of Mealy Sequence Detector

\[
\begin{array}{c|cc}
ab & 0 & 1 \\
\hline
00 & 0 & 0 \\
01 & 1 & 0 \\
11 & \times & \times \\
10 & 0 & 0 \\
\end{array}
\]

\[
X + = X' B
\]

\[
\begin{array}{c|cc}
ab & 0 & 1 \\
\hline
00 & 0 & 1 \\
01 & 0 & 1 \\
11 & \times & \times \\
10 & 0 & 1 \\
\end{array}
\]

\[
B^+ = X
\]

\[
\begin{array}{c|cc}
ab & 0 & 1 \\
\hline
00 & 0 & 1 \\
01 & 0 & 1 \\
11 & \times & \times \\
10 & 0 & 1 \\
\end{array}
\]

\[
Z^+ = XA
\]
State graph for Moore detector

Same FSM implemented as Moore slower and more states but is more robust in hardware implementation.

\[ X = 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \]

\[ Z = 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \]
Software State Machines

Write code that inspects inputs and saves a state variable and responds to outputs based on state and inputs.

Examples:

Elevator Control – State variable is which floor and current motion. Inputs are buttons in and out of elevator. Output is elevator motor and door control.

Stop Light – State is which color in which direction and for how long. Inputs could include pressure sensor. Output is lights.

Drink dispenser – State is the current credit in coins. Input includes coin monitoring and drink selection. Output is drink release.

Not an example: Temperature control. No state required. What is the current temperature versus target. Control heater without knowledge of the past.