Chapter 8 – Stacks
Topics to Cover…

- The Stack
- Subroutines
- Subroutine Linkage
- Saving Registers
- Stack Operations
- Activation Records
  - Example 8.1: Activation Records
- Recursive Subroutines
- Interrupt Stack Usage
Levels of Transformation

Problems

Algorithms

Language (Program)

Machine (ISA) Architecture

Microarchitecture

Circuits

Devices

Programmable

Computer Specific

Manufacturer Specific
Stacks

- Stacks are the fundamental data structure of computers today.
- A stack is a Last In, First Out (LIFO) abstract data structure.
- A true stack is a restricted data structure with two fundamental operations, namely push and pop.
- Elements are removed from a stack in the reverse order of their addition.
- Memory stacks are used for random access of local variables.
MSP430 Stack

- Hardware support for stack
  - Register R1 – Stack Pointer (SP)
  - Initialized to highest address of available RAM
    - MSP430G2553 → 0x0400 (512 bytes)
    - MSP430F2274 → 0x0600 (1k bytes)
  - Stack grows down towards lower memory addresses.

- Initialize stack pointer at beginning of program

```
STACK .equ 0x0400  ; top of stack

start: mov.w #STACK,SP  ; initialize stack pointer
```
The MSP430 stack is a word structure
- Elements of the stack are 16-bit words.
- The LSB of the Stack Pointer (SP) is always 0.
- The SP points to the last word added to the stack (TOS).

The stack pointer is used by
- **PUSH** – put a value on the stack
- **POP** – retrieve a value off the stack
- **CALL** – put a return address on the stack
- **RET** – retrieve a return address off the stack
- **RETI** – retrieve a return address and status register off the stack
- Interrupts – put a return address and status register on the stack
Computer Memory – Up or Down?

The Stack

memory addresses:

- x0000
- xFFFF

Up

Down

1996
1995
1982
1998
1996
Implementing Stacks in Memory

Unlike a coin stack, in a memory stack, the data does not move in memory, just the pointer to the top of stack.

- Push #0x0018
- Push #0x0025
- Push #0x0058
- Pop R15: #58 -> R15
- Push #0036
Quiz 8.1

1. What is the value of the stack pointer after the second call to delay?

2. Is there a problem with the program?

```assembly
start:   mov.w   #0x0400,SP
         mov.w   #WDTPW+WDTHOLD,&WDTCTL
         bis.b   #0x01,&P1DIR ; P1.0 as output

mainloop: bis.b   #0x01,&P1OUT ; turn on LED
          push    #1000
          call    #delay
          bic.b   #0x01,&P1OUT ; turn off led
          call    #delay
          jmp     mainloop

delay:   mov.w   2(sp),r15 ; get delay counter

delaylp2: dec.w   r15 ; delay over?
           jnz     delaylp2
           ret

.sect    ".reset"
.word    start
.end
```
Subroutines

- A subroutine is a program *fragment* that performs some useful function.
  - Subroutines help to organize a program.
  - Subroutines should have strong *cohesion* – perform only one specific task.
  - Subroutines should be loosely *coupled* – interfaced only through parameters (where possible) and be independent of the remaining code.
  - Subroutines keep the program smaller
    - Smaller programs are easier to maintain.
    - Reduces development costs while increasing reliability.
    - Fewer bugs – copying code repeats bugs.
  - Subroutines are often collected into libraries.
The Call / Return Mechanism

(a) Without subroutines

X
A
Y
A
Z
A
W

(b) With subroutines

X
Call
Y
Call
Z
Call
W

Smaller programs.
Easier to maintain.
Reduces development costs.
Increased reliability.
Fewer bugs due to copying code.
More library friendly.

Faster programs.
Less overhead.
A subroutine is “called” in assembly using the MSP430 CALL instruction.

The address of the next instruction after the subroutine call is saved by the processor on the stack.

Parameters are passed to the subroutine in registers and/or on the stack.

Local variables are created on the stack at the beginning of the subroutine and popped from the stack just before returning from the subroutine.

At the end of a subroutine, a RET instruction “pops” the top value from the stack into the program counter.
Stack Operations

- Single operand instructions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH(.B or .W) src</td>
<td>SP-2→SP, src→@SP</td>
<td>Push byte/word source on stack</td>
</tr>
<tr>
<td>CALL dst</td>
<td>dst→tmp ,SP-2→SP, PC→@SP, tmp→PC</td>
<td>Subroutine call to destination</td>
</tr>
<tr>
<td>RETI</td>
<td>TOS→SR, SP+2→SP, TOS→PC, SP+2→SP</td>
<td>Return from interrupt</td>
</tr>
</tbody>
</table>

- Emulated instructions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Emulation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RET</td>
<td>@SP→PC SP+2→SP</td>
<td>MOV @SP+,PC</td>
<td>Return from subroutine</td>
</tr>
<tr>
<td>POP(.B or .W) dst</td>
<td>@SP→temp SP+2→SP temp→dst</td>
<td>MOV(.B or .W) @SP+,dst</td>
<td>Pop byte/word from stack to destination</td>
</tr>
</tbody>
</table>
Call Instruction

\[ \text{call } \#\text{func } ; \ M(--\text{sp}) = \text{PC}; \ \text{PC} = M(\text{func}) \]
# Subroutine Call

<table>
<thead>
<tr>
<th>CALL</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>CALL dst</td>
</tr>
<tr>
<td>Operation</td>
<td>dst → tmp</td>
</tr>
<tr>
<td></td>
<td>(SP–2) → SP</td>
</tr>
<tr>
<td></td>
<td>PC → @SP</td>
</tr>
<tr>
<td></td>
<td>tmp → PC</td>
</tr>
</tbody>
</table>

| Description | A subroutine call is made to an address anywhere in the 64K address space. All addressing modes can be used. The return address (the address of the following instruction) is stored on the stack. The call instruction is a word instruction. |

| Status Bits | Status bits are not affected. |
| Example     |                              |
CALL Examples

- CALL #EXEC ; Call on label EXEC or immediate address (e.g. #0A4h)
  ; @PC+ → tmp, SP−2 → SP, PC → @SP, tmp → PC

- CALL EXEC ; Call on the address contained in EXEC
  ; X(PC)→tmp, PC+2→PC, SP−2→SP, PC→@SP, tmp→PC

- CALL &EXEC ; Call on the address contained in absolute address EXEC
  ; X(0)→tmp, PC+2→PC, SP−2→SP, PC→@SP, tmp→PC

- CALL R5 ; Call on the address contained in R5
  ; R5→tmp, SP−2→SP, PC→@SP, tmp→PC

- CALL @R5 ; Call on the address contained in the word pointed to by R5
  ; @R5→tmp, SP−2→SP, PC→@SP, tmp→PC

- CALL @R5+ ; Call on the address contained in the word pointed to by R5
  ; and increment pointer in R5.
  ; @R5+→tmp, SP−2→SP, PC→@SP, tmp→PC

- CALL X(R5) ; Call on the address contained in the address pointed to by
  ; R5 + X (e.g. table with address starting at X)
  ; X can be an address or a label
  ; X(R5)→tmp, PC+2→PC, SP−2→SP, PC→@SP, tmp→PC
Caution…

- The destination of branches and calls is used indirectly, and this means the content of the destination is used as the address.

- Errors occur often when confusing symbolic and absolute modes:
  - CALL MAIN ; Subroutine’s address is stored in MAIN
  - CALL #MAIN ; Subroutine starts at address MAIN

- The real behavior is easily seen when looking to the branch instruction. It is an emulated instruction using the MOV instruction:
  - BR MAIN ; Emulated instruction BR
  - MOV MAIN, PC ; Emulation by MOV instruction

- The addressing for the CALL instruction is exactly the same as for the BR instruction.
Return Instruction

```
ret ; mov.w @(sp+),PC
```

Diagram showing the flow of instructions and data between memory, the CPU, and registers.
Return from Subroutine

- **RET**
  - Return from subroutine
- **Syntax**
  - RET
- **Operation**
  - \( @SP \rightarrow PC \)
  - \( SP + 2 \rightarrow SP \)
- **Emulation**
  - MOV @SP+,PC
- **Description**
  - The return address pushed onto the stack by a CALL instruction is moved to the program counter. The program continues at the code address following the subroutine call.
- **Status Bits**
  - Status bits are not affected.
- **Example**
Quiz 8.2

1. What is wrong (if anything) with the following code?

2. How many times will delay be called for each loop?

3. How long will myDelay delay?

```assembly
loop:   call   #myDelay
        jmp   loop
myDelay: mov.w  #0, r15
         call   #delay
         call   #delay
delay:  sub.w  #1, r15
        jne   delay
        ret
```
Saving and Restoring Registers

- **Called routine -- “callee-save”**
  - At beginning of subroutine, save all registers that will be altered (unless a register is used to return a value to the calling program or is a scratch register!)
  - Before returning, restore saved registers in reverse order.
  - Or, avoid using registers altogether.

- **Calling routine -- “caller-save”**
  - If registers need to be preserved across subroutine calls, the calling program would save those registers before calling routine and restore upon returning from routine.
  - Obviously, avoiding the use of registers altogether would be considered caller-safe.

  **Values are saved by storing them in memory, preferably on the stack.**
Caller-Save vs. Callee-Save

- **Caller-Save**
  - Save Registers
  - Call subroutine
  - Restore Registers

- **Callee-Save**
  - Save Registers
  - Call subroutine
  - Restore Registers
  - Subroutine
Stack Operations

### Single operand instructions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH (.B or .W) src</td>
<td>SP-2→SP, src→@SP</td>
<td>Push byte/word source on stack</td>
</tr>
<tr>
<td>CALL dst</td>
<td>dst→tmp ,SP-2→SP, PC→@SP, tmp→PC</td>
<td>Subroutine call to destination</td>
</tr>
<tr>
<td>RETI</td>
<td>TOS→SR, SP+2→SP, TOS→PC, SP+2→SP</td>
<td>Return from interrupt</td>
</tr>
</tbody>
</table>

### Emulated instructions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Emulation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RET</td>
<td>@SP→PC SP+2→SP</td>
<td>MOV @SP+,PC</td>
<td>Return from subroutine</td>
</tr>
<tr>
<td>POP (.B or .W) dst</td>
<td>@SP→temp SP+2→SP temp→dst</td>
<td>MOV(.B or .W) @SP+,dst</td>
<td>Pop byte/word from stack to destination</td>
</tr>
</tbody>
</table>
Push Instruction

\[ \text{push.w cnt ; M(--sp) = M(cnt)} \]

BYU CS/ECEn 124

Chapter 8 - Stacks
Push Operand

- **PUSH**
  - Push word or byte onto stack
- **Syntax**
  - PUSH{.W or .B} src
- **Operation**
  - SP – 2 → SP
  - src → @SP
- **Description**
  - The stack pointer is decremented by two, then the source operand is moved to the RAM word addressed by the stack pointer (TOS).
- **Status Bits**
  - Status bits are not affected.
- **Example**
  - PUSH SR ; save SR
  - PUSH R8 ; save R8
  - PUSH.B &TCDAT ; save data at address
  - ; TCDAT onto stack

**Note:** The system stack pointer (SP) is always decremented by two, independent of the byte suffix.
Pop Operand

- **POP**
  - Pop word or byte from stack to destination
- **Syntax**
  - POP{.W or .B} dst
- **Operation**
  - @SP -> temp
  - SP + 2 -> SP
  - temp -> dst
- **Emulation**
  - MOV{.W or .B} @SP+,dst
- **Description**
  - The stack location pointed to by the stack pointer (TOS) is moved to the destination. The stack pointer is incremented by two afterwards.
- **Status Bits**
  - Status bits are not affected.
- **Example**
  - POP R7 ; Restore R7
  - POP.B LEO ; The low byte of the stack is moved to LEO.

Note: The system stack pointer (SP) is always incremented by two, independent of the byte suffix.
Stack Operations

Subroutine Linkage

0xf820: ...
0xf822: call #subroutine
0xf826: ...

subroutine:
0xf852: push r15
0xf854: push r14
...
0xf882: pop r14
0xf884: pop r15
0xf886: ret

Unprotected!
A subroutine is *activated* when called and an *activation record* is allocated (pushed) on the stack.

An *activation record* is a template of the relative positions of local variables on the stack as defined by the subroutine.

- Return address
- Memory for local subroutine variables
- Parameters passed to subroutine from caller
- Saved registers used in subroutine (callee-save)

A new activation record is created on the stack for each invocation of a subroutine or function.

A *frame pointer* indicates the start of the activation record.

When the subroutine ends and returns control to the caller, the activation record is discarded (popped).
Example 8.1: Activation Record

Delay Activation Record:
4(SP) = delay count
2(SP) = return address
0(SP) = r15

Stack:
2(SP) = delay count
0(SP) = return address

Stack:
0(SP) = return address

Stack:
(EMPTY)
Quiz 8.3

Change the following code to use a callee-save, loosely coupled, cohesive subroutine.

```assembly
.cdecls C,"msp430.h"
.text
.start:    mov.w #0x0400,SP
            mov.w #WDTPW+WDTHOLD,&WDTCTL
            bis.b #0x01,&P1DIR ; P1.0 as output
.mainloop:    bis.b #0x01,&P1OUT ; turn on LED
            mov.w #10000,r15 ; delay counter
            delaylp1:    dec.w r15 ; delay over?
                         jnz delaylp1 ; n
                         bic.b #0x01,&P1OUT ; turn off led
                         mov.w #0,r15 ; delay counter
                         delaylp2:    dec.w r15 ; delay over?
                                        jnz delaylp2 ; n
                                        mov.w #0,r15 ; delay counter
                         delaylp3:    dec.w r15 ; delay over?
                                        jnz delaylp3 ; n
                                        jmp mainloop ; y, toggle led
            .sect ".reset"
            .word start ; reset vector
            .word start ; start address
.end
```
Recursive Subroutine

- A subroutine that makes a call to itself is said to be a recursive subroutine.
- Recursion allows direct implementation of functions defined by mathematical induction and recursive divide and conquer algorithms
  - Factorial, Fibonacci, summation, data analysis
  - Tree traversal, binary search
- Recursion solves a big problem by solving one or more smaller problems, and using the solutions of the smaller problems, to solve the bigger problem.
- Reduces duplication of code.
- MUST USE STACK!
Interrupts

- Execution of a program normally proceeds predictably, with *interrupts* being the exception.
- An *interrupt* is an asynchronous signal indicating something needs attention.
  - Some event has occurred
  - Some event has completed
- The processing of an interrupt subroutine uses the stack.
  - Processor stops with it is doing,
  - stores enough information on the stack to later resume,
  - executes an *interrupt service routine* (ISR),
  - restores saved information from stack (*RETI*),
  - and then resumes execution at the point where the processor was executing before the interrupt.
Interrupt Stack

Prior to Interrupt

- **SP**
- **PC**
  - `add.w r4,r7`
  - `jnc $+4`
  - `add.w #1,r6`
  - `add.w r5,r6`

Interrupt (hardware)

- Program Counter pushed on stack
- Status Register pushed on stack
- Interrupt vector moved to PC
- Further interrupts disabled
- Interrupt flag cleared

Execute Interrupt Service Routine (ISR)

- **PC**
  - `xor.b #1,&P1OUT`
  - `reti`

Return from Interrupt (**reti**)

- **SP**
- **PC**
  - `add.w r4,r7`
  - `jnc $+4`
  - `add.w #1,r6`
  - `add.w r5,r6`