Lecture 15

More on Subroutines

Moral of the Day: **Clean up the Stack.**

Solutions to Midterm #1

Task: Write a short assembly program that finds the maximum value in an array and the number of times this value appears

```
            mov.w   #0x8000, max_value      ; init max_value = -infinity
            clr.w   max_count
            clr.w   R5                      ; R5 is the index 0, 2, ..., LENGTH-2

compare_to_max:
            cmp.w   max_value, array(R5)    ; compare current element to max
            jl      next_element            ; if smaller than max, proceed to next

            jeq     same_max                ; if equal to max, just update count

new_max:                                    ; here it is larger than max
            mov.w   array(R5), max_value    ; update max
            mov.w   #1, max_count           ; restart count
            jmp     next_element

same_max:
            inc.w   max_count               ; encountered same max, update count

next_element:
            incd.w  R5                      ; index to next word
            cmp.w   #LENGTH, R5             ; last index to process is LENGTH-2
            jne     compare_to_max

done:       jmp     done
            nop
```

Solutions to Midterm #1

Task: Write a short assembly program that finds the maximum value in an array and the number of times this value appears

Can reduce code by 2 lines with little tricks ...

```
            mov.w   #0x8000, max_value       ; init max_value = -infinity
            clr.w   max_count
            mov.w   #LENGTH-2, R5            ; R5 is the index LENGTH-2, ..., 2, 0

compare_to_max:
            cmp.w   max_value, array(R5)     ; compare current element to max
            jl      next_element             ; if smaller than max, proceed to next

            jeq     same_max                 ; if equal to max, just update count

new_max:                                     ; here it is larger than max
            mov.w   #0, max_count            ; restart count

same_max:
            mov.w   array(R5), max_value     ; update max
            inc.w   max_count                ; encountered same max, update count

next_element:
            decd.w  R5                       ; index to next word
            jhs     compare_to_max

done:       jmp     done
            nop
```

Not so Good Solutions

Good assembly code is

- correct
- efficient
- well-structured: no spaghetti
- easy to read and test

Not so good solutions to Midterm #1:

- Why do two loops when you can do only one? ???
- Spaghetti

```
loop:
        cmp.w   R6, R8
        jn      end
        jmp     next_element
retel:
        jmp     compare_to_max
retmax:
        jn      new_max
        jeq     same_max
        jmp     loop
```

Anytime you have back-to-back jumps

Ask yourself: Do I really need back-to-back jumps?

Sometimes two jumps are necessary

(Almost) never three or more jumps

Not so Good Solutions

- Milder forms of spaghetti

```
; Add your code here
;Use the labels given below (alphabetical or
; You can add more labels if really necessar
; that the task can be done in ~15 lines
        mov.w #0, R5                 ; R5 to 0
        mov.w array(R5), &max_value   ; max_V
        mov.w #0, &max_count        ;max_count t
done:
        cmp.w #26, R5           ;Compare R5 with
        jge  loop                   ;R5 <= #26,
compare_to_max:
        cmp.w array(R5), &max_value ;compare
        jeq same_max                ; jump to sa

        cmp.w array(R5), &max_value   ; Compa
        jge new_max               ; Jump to new_m
        jmp next_element     ; Jump to nex_ele
next_element:
        incd.w R5          ; Inc R5 to the next
        jmp done

new_max:
        mov.w array(R5), &max_value   ; updat
        mov.w #0, &max_count        ; Reset max_
        incd.w max_count          ; Increment m
        jmp next_element          ; Jump to ne
same_max:
        incd.w max_count         ; Increment ma
        jmp next_element
loop:
        nop
```

```
                mov.w   #0x8000, max_value
                clr.w   max_count
                clr.w   R5

compare_to_max:
                cmp.w   max_value, array(R5)
                jl      next_element

                jeq     same_max

new_max:
                mov.w   array(R5), max_value
                mov.w   #1, max_count
                jmp     next_element

same_max:
                inc.w   max_count

next_element:
                incd.w  R5
                cmp.w   #LENGTH, R5
                jne     compare_to_max

done:           jmp     done
                nop
```

Not so Good Solutions

Do you see what can go wrong here?

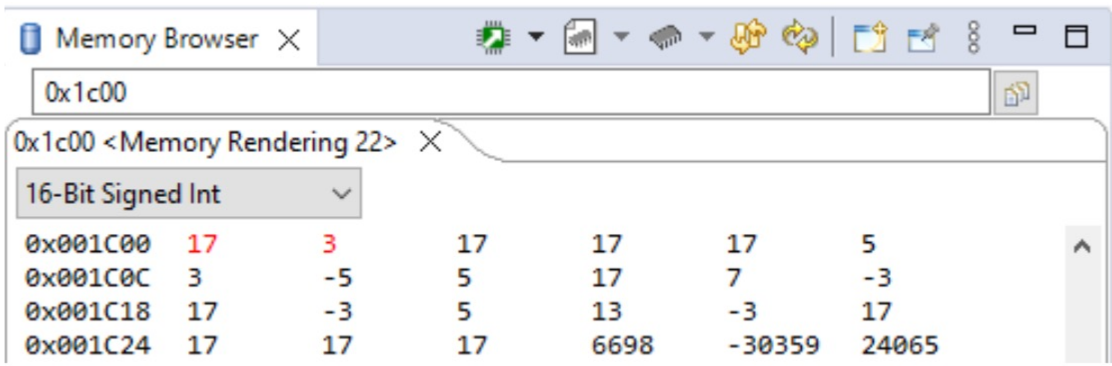
First element counted here

First element counted again

```
            mov.w   #0, R4                  ;Moving the value of
            mov.w   array(R4), max_value    ;Moving or setting t
            mov.w   #1, max_count           ;Moving the value 1

compare_to_max:
            cmp.w   max_value, array(R4)    ;Comparing if R5 is
            jeq     same_max                ;Jump if equal
            jge     new_max                 ;ump if greater or e

next_element:
            incd.w  R4                      ;R4++...proceed to t
            cmp.w   #LENGTH,    R4          ;Compare R4 to the l
            jne     compare_to_max          ;Jump if not equal
            jmp     done
new_max:
            mov.w   array(R4),  max_value   ;Moving the new larg
            mov.w   #1, max_count           ;Set max_count equal
            jmp     next_element            ;Jump to the nect el

same_max:
            inc.w   max_count               ;Increment max_count
            jmp     next_element            ;Jump to the next el

done:       jmp     done
            nop
```

Importance of Testing Code

You write and run some code, see the desired results $\Rightarrow$ A promising start



Testing **IS NOT** running the code you wrote and checking the results

Goal is to make sure that the code works correctly

- for ANY given array
- for ANY value of the max. element
- for ANY location of the max. element & ANY number of max. elements

Need to create **test cases** to confirm correct behavior for corner cases

- first and/or last element in array is maximum
- max. element is negative

Quiz #5

Posted to Carmen -- **Due Friday October 20**

No need to work on it over the break!

Task 1: Write a subroutine `mod` that finds the remainder after division

the modulus operator `%`

Task 2: Write a subroutine `mod_exp` that does modular exponentiation

Subroutine `mod_exp` will depend on subroutine `mod`

=> You need to test your subroutine `mod` **thoroughly**

Test cases to check everything that could go wrong

```
0 % 23 = 0                56 % 7 = 0
31 % 41 = 31              313 % 37000 = 313
35002 % 350 = 2           60000 % 45000 = 15000
```

Clarifying a Confusion

What way do comparisons work?

```
cmp.w    max_value, array(R5)
jl       next_element
```

No need to memorize -- just know what compare does

```
cmp.w    src, dst
```

Sets the status bits according to the result of `dst-src`

Here, we jump when

```
array(R5) - max_value < 0
```

i.e.,

```
array(R5) < max_value
```

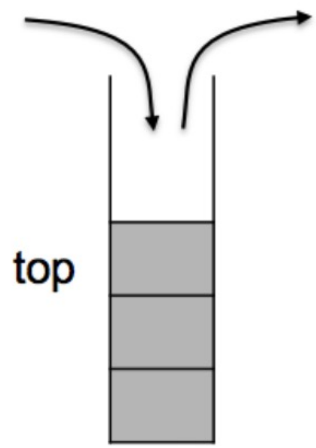
Make sure to use the correct set of jumps: **`jhs & jlo`** unsigned

or **`jge & jl`** signed

A *Corny* Joke

What happens when you push corn onto the stack?

corn

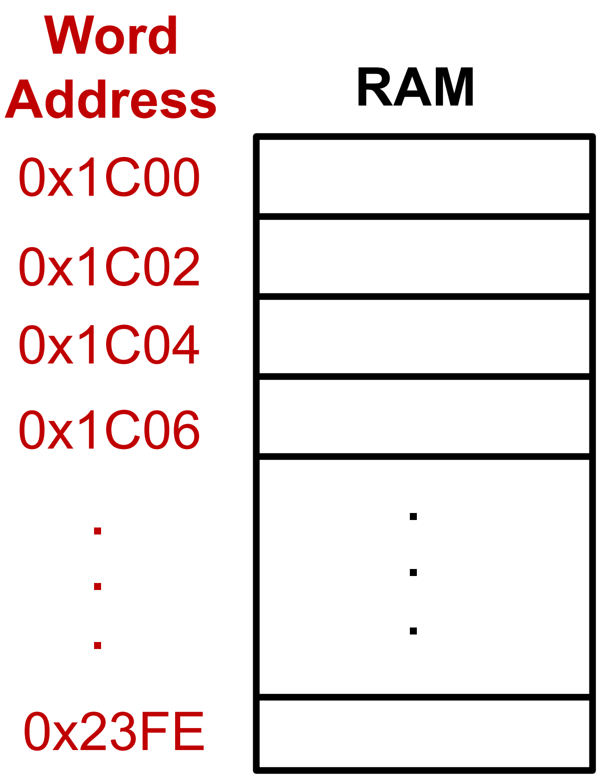


stack

Last Time: The Stack

The **stack** is a data structure that is managed at the end of the RAM

managed using **`SP = R1`**, **`push`** and **`pop`**



0x2400 – not in RAM

Subroutine calls and interrupts use the stack to save critical registers (`PC` and `SR`) before execution and restore these with `ret`/`reti`

We can use the stack to save/restore additional registers (R4 – R15) during subroutine calls and interrupts

We can create variables during runtime without initializing/reserving them at compile time
$\Rightarrow$ dynamic data allocation

← **Stack** starts here

```
mov.w    #__STACK_END,SP      ; Initialize stackpointer
```

0x2400

Last Time: `x_times_y`

We wrote a subroutine to multiply two numbers in R5 and R6 (both <= 255) and return the result in R12

```
;-------------------------------------------------------------------------------
; Subroutine: x_times_y
; Inputs: unsigned byte x in R5 -- returned unchanged
;         unsigned byte y in R6 -- returned unchanged
;
; Output: unsigned number in R12 -- R12 = R5 * R6
;
; This time implement the long multiplication algorithm
;
; All other core registers in R4-R15 unchanged
;-------------------------------------------------------------------------------
x_times_y:
```

Main idea was to

- test the bits of one of the numbers (`R5`) one by one using `bit.w`
- add a left shifted version of the other number (`R6`) if a bit is 1

Binary Long Multiplication v.1

```
x_Times_y:
            push.w  R6
            push.w  R10
            push.w  R11

            clr.w   R12

            mov.w   #8, R10                 ; R10 will count through the bits
            mov.w   #BIT0, R11              ; R11 will mark the bits

Repeat:     bit.w   R11, R5
            jnc     Next_bit

            add.w   R6, R12

Next_bit:   rla.w   R6
            rla.w   R11
            dec.w   R10
            jne     Repeat

            pop.w   R11
            pop.w   R10
            pop.w   R6

            ret
```

Binary Long Multiplication v.2

Alternative way of sequentially testing bits

`rra.w  R5`

`shift/roll right arithmetic`

right most bit in R5 → C status bit

use `jc` or `jnc` to control the flow

Does not require a bitmask to test the bits of `R5`

However, unlike bit test `bit.w`, roll right arithmetic `rra.w` **modifies `R5`**

No big deal! We know how to fix it.

Makes a great practice problem!

Binary Long Multiplication v.2

```
x_Times_y:
            push.w  R5
            push.w  R6
            push.w  R10

            clr.w   R12

            mov.w   #8, R10                 ; R10 will count through the bits

Repeat2:    rra.w   R5
            jnc     Next_bit2

            add.w   R6, R12

Next_bit2:  rla.w   R6
            dec.w   R10
            jne     Repeat2

            pop.w   R10
            pop.w   R6
            pop.w   R5

            ret
```

Using Core Reg's in Subroutines

Even if a subroutine is not allowed to modify any core register (contract!) it can still use them

- At the beginning of the subroutine **`push`** affected registers onto stack
- Use them during the subroutine
- Before returning from subroutine **`pop`** all registers that have been pushed

e.g.,

Binary Long Multiplication v.1

```
x_Times_y:
            push.w  R6
            push.w  R10
            push.w  R11

            clr.w   R12

            mov.w   #8, R10                 ; R10 will count through the bits
            mov.w   #BIT0, R11              ; R11 will mark the bits

Repeat:     bit.w   R11, R5
            jnc     Next_bit

            add.w   R6, R12

Next_bit:   rla.w   R6
            rla.w   R11
            dec.w   R10
            jne     Repeat

            pop.w   R11
            pop.w   R10
            pop.w   R6

            ret
```

Using Core Reg's in Subroutines

Even if a subroutine is not allowed to modify any core register (contract!) it can still use them

- At the beginning of the subroutine **`push`** affected registers onto stack
- Use them during the subroutine
- Before returning from subroutine **`pop`** all registers that have been pushed

e.g.,

```
x_Times_y:

            clr.w     R12
            push      R6

Test4Ret:   tst.w     R6
            jne       Add_more

            pop       R6
            ret

Add_more:   add.w     R5, R12
            dec.w     R6
            jmp       Test4Ret
            nop
```

It is critical that a subroutine always cleans up the stack: whatever it pushes onto stack it has to pop back !!!

Using Core Reg's in Subroutines

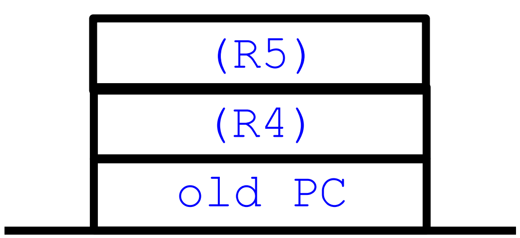
If a subroutine does not clean up the stack

```
Wreak_havoc:
            push.w  R4
            push.w  R5

            tst.w   R6
            jne     Continue

    ***     ret

Continue:   ; do some stuff here
            ; do more stuff
            ; even more stuff
            ; when done

            pop.w   R5
            pop.w   R4

            ret
```

- Memory leak of 2 words – i.e., 2 words left behind on the stack when returning from `***`

- When returning from `***` the PC is restored incorrectly
  `(R4) -> PC !!!!`

Program execution will continue from a random location in memory – **BAD**



with `push R5`

with `push R4`

with `call #W...`

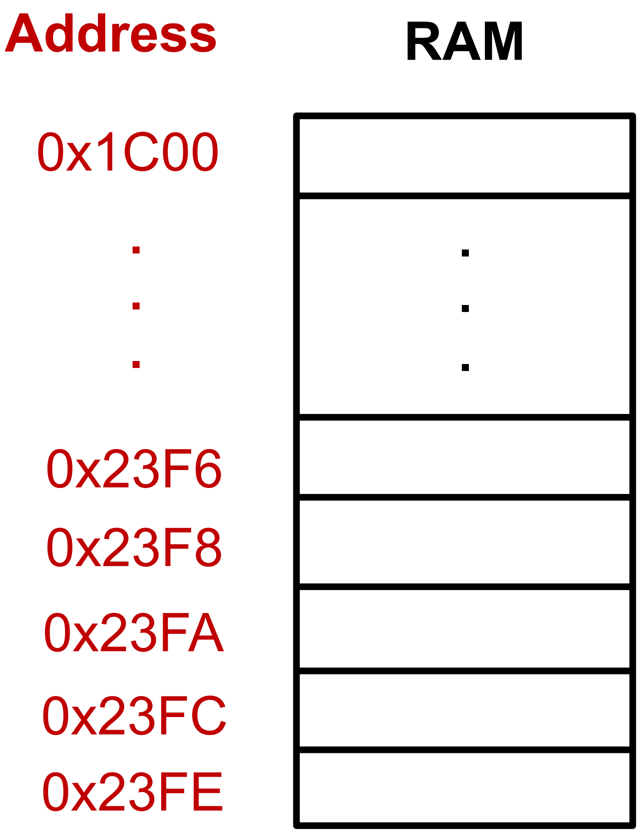
Stack

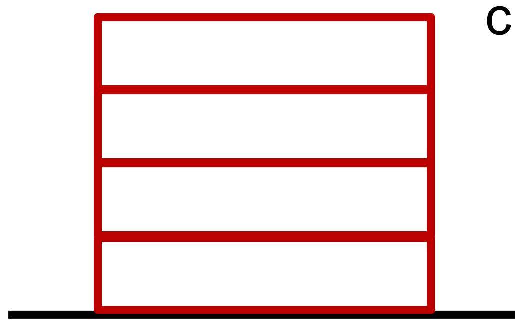
Size of the Stack

How much data can we push onto the stack?

Depends on how much data we have allocated at the top of RAM

using compiler directives





If we put too much data onto the stack it will start to overwrite the data allocated at the top of RAM

$\Rightarrow$ **Stack overflow**

When stack pointer crosses top of RAM

$\Rightarrow$ **Crash**

Avoid at all cost!