Lecture 14

Subroutines and the Stack

Assignment Pipeline

Will post a graded anonymous survey: **Mid-Semester Class Feedback**

Quiz #5 is in the making: Can you compute 160^{77} (mod 221) ?
i.e.,
What is the remainder when you divide 160^{77} by 221?

Can you compute this using 16 bits? ???
How big is 160^{77}?

$160^{77} > 128^{77} = (2^7)^{77} = 2^{539}$ $160^{77} \sim 2^{564}$ **Fun!**

Will post Quiz #5 over the weekend **due Wednesday October 17**
You will have a full week even without counting the Autumn Break

No office hours on Tuesday October 10

Last Time: Subroutine Calling Sequence

Sequence o events after

```
call #div_by_16
```

- Address of next instruction is **saved on the stack**
- This will be the **return address**

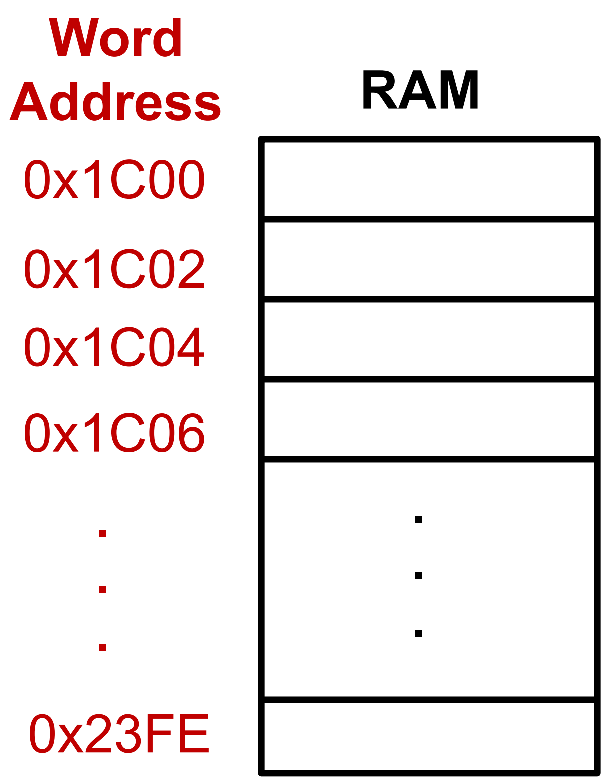
- The address of the subroutine is loaded into the PC
- The subroutine is executed

- With `ret`, the return address is **restored from the stack** into PC
- Execution continues from this point in the calling function

```
;-------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------
            mov.w   #LENGTH-2, R4

read_nxt:   mov.w   array_1(R4), R5
            call    #div_by_16
ret_addr:   mov.w   R5, array_2(R4)

            decd.w  R4
            jhs     read_nxt

main:       jmp     main
            nop

;-------------------------------------------------------------
; Subroutine: div_by_16
; Input:    16-bit signed number in R5 -- mod:
; Output:   16-bit signed number in R5 -- R5 =
;-------------------------------------------------------------
div_by_16:  rra.w   R5              ; R5 <-- R5/2
            rra.w   R5              ; R5 <-- R5/2
            rra.w   R5              ; R5 <-- R5/2
            rra.w   R5              ; R5 <-- R5/2
            ret
```

Static vs. Dynamic Allocation

So far we have used the RAM for storing program data initialized or reserved at compilation time – using compiler directives `.word` `.byte` `.space`



Assembler directives allocate data at the **top of the RAM**
e.g.,

```
            .data
            .retain
            .retainrefs

array_1:    .word   1, 2, 3, 4, 5, 6, 7, 8, 9,
array_2:    .space  24
```

This allocation is **static** – done at compile time and does *not* change during runtime

⇒ **Static allocation**

The Stack

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

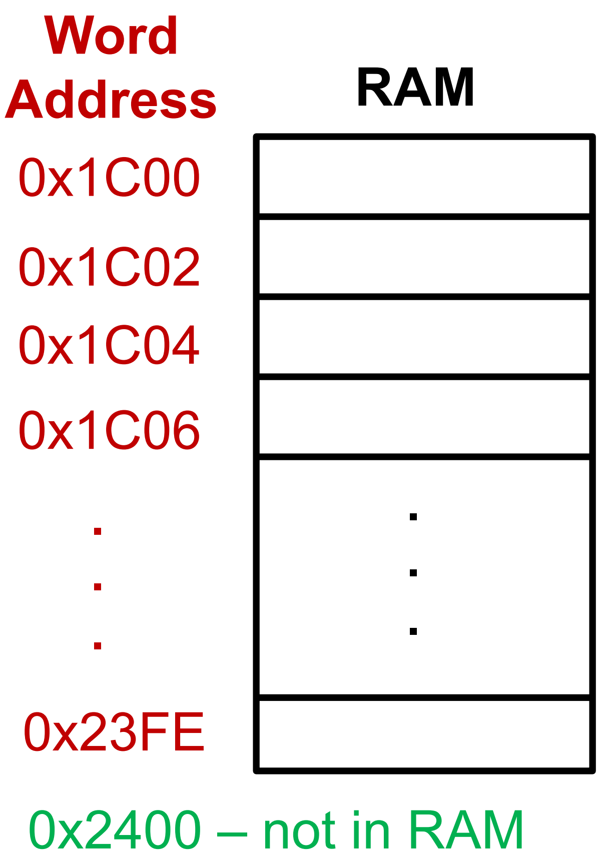
managed using `SP`, `push` and `pop`

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

The stack enables **dynamic data allocation**



← **Stack** starts here

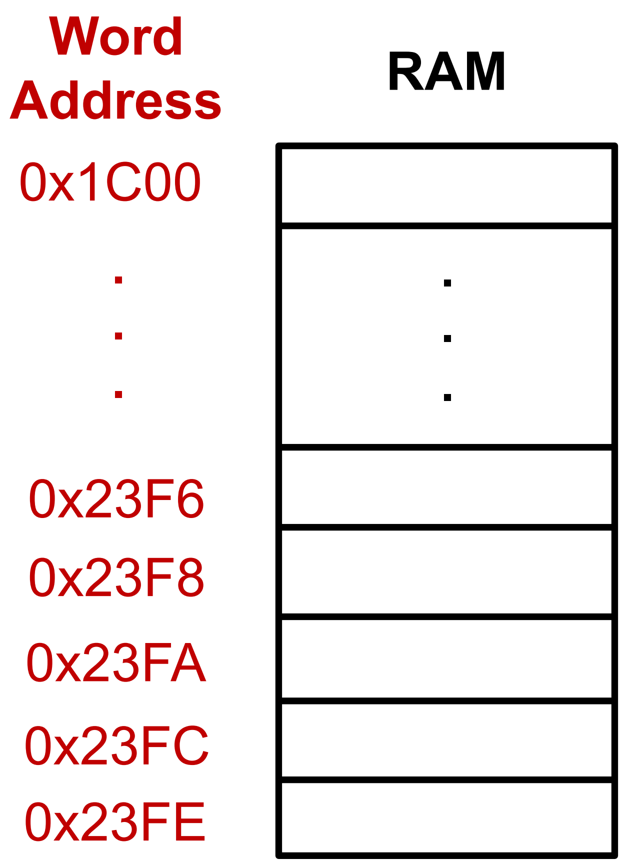
```
mov.w    #__STACK_END,SP      ; Initialize stackpointer
```

0x2400

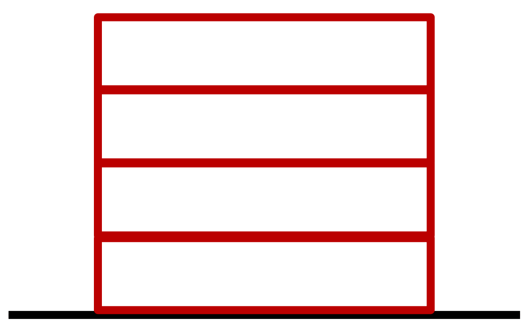
The Stack

The **stack** starts empty and is managed dynamically during runtime

i.e., we can add new data to the stack and remove it



Always add to the top and remove from the top



Top of Stack – Stack Pointer (SP)

New elements are added onto the top of stack and removed from there

To manage the stack, we **only** need to know the address of the **top of stack**

Core register R1 is dedicated for this task: **Stack Pointer (SP)**

At the beginning of each program the stack pointer is initialized

```
RESET       mov.w   #__STACK_END,SP         ; Initialize stackpointer
                    #0x2400
```

0x23F8

0x23FA

0x23FC

0x23FE

0x2400 – not in RAM !

The stack pointer is always aligned with **even addresses**

`SP = 0x2400`

← `SP` points here

Saving/Restoring Registers using the Stack

Often subroutine contracts have restrictions on using core registers

```
;-------------------------------------------------------------------
; Subroutine: x_Times_y
; Inputs: unsigned 8-bit number x in R5 -- returned unchanged
;         unsigned 8-bit number y in R6 -- returned unchanged
;
; Output: unsigned 16-bit number in R12 -- R12 = R5 * R6
;
; All other core registers in R4-R15 unchanged
;-------------------------------------------------------------------
```

We will use the stack to save core registers at the beginning of a subroutine and restore them before returning

```
x_times_y:

            push    R5
            push    R6

            ; Compute R5*R6

            pop     R6
            pop     R5

            ret
```

Match the number `push` and `pop`!

If you leave anything behind in stack program will crash !!

Mind the order of `push` and `pop`!

Stack is last-in first-out

Stack – Adding and Removing Data

To add data onto the stack we use **push.w** src

To remove data from the stack **pop.w** dst

only the **.w** versions to make it easy!

Address **RAM**

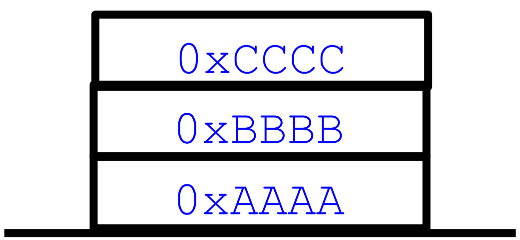
0x23F6

0x23F8

0x23FA

0x23FC

0x23FE




```
push.w    #0xAAAA
push.w    #0xBBBB
push.w    #0xCCCC

pop.w     R4
pop.w     R5
pop.w     R6
```

The stack is a **last-in first-out** data structure: the last element that is added onto the stack (i.e., **push**ed) is the first element removed (i.e., **pop**ped)

Today's Coding Task

Task: Write a subroutine that does multiplication

```
;--------------------------------------------------------------------
; Subroutine: x_Times_y
; Inputs: unsigned 8-bit number x in R5 -- returned unchanged
;         unsigned 8-bit number y in R6 -- returned unchanged
;
; Output: unsigned 16-bit number in R12 -- R12 = R5 * R6
;
; All other core registers in R4-R15 unchanged
;--------------------------------------------------------------------
```

How do we do this?

One Idea

Multiplication is repeated addition

Multiplying the number `x` (in `R5`) by the number `y` (in `R6`) …

… is the same as adding the number x y times

$$\underbrace{\texttt{x + x + ... + x}}_{\texttt{y times}}$$

Start with `R12` = 0

Make a simple loop:

add `R5` to `R12`

decrease `R6` to account for one addition

Repeat until `R6` hits zero

Good code is ________ . **defect-free** ✓ **spaghetti-free** ✓ **efficient** ✗

Binary Long Multiplication

1101 ← x		
× 1011 ← y		
1101	BIT0 of y is 1	add x
1101 <<	BIT1 of y is 1	shift left x, add x
0000 <<	BIT2 of y is 0	shift left x, no add
+ 1101 <<	BIT3 of y is 1	shift left x, add x
10001111		

<< : shift left

No need to do multiplication – all we need is

- test bits: is a bit 0 or 1
- shift left
- add

Binary Long Multiplication

```
      1101   ← x
    × 1011   ← y
    ------
      1101
     1101 <<
    0000 <<
 + 1101 <<
 ---------
  10001111
```

<< : shift left

test BIT j of y   j = 0, 1, ..., 7

shift x j times to the left

if BIT j = 1   add shifted x

if = 0   add 0 / skip

Repeat from top until BIT7

j=0   BIT0 = 0...01

j=1   BIT1 = 0... 10   <<

j=2   BIT2 = 0.. 100   <<

Binary Long Multiplication

How do we test BIT$_j$ say of R5 ?

$j=0$ bit.w #BIT0, R5

$j=1$ bit.w #BIT1, R5

$j=2$ bit.w #BIT2, R5

⋮

How can we do this in a loop ?

Observe: BIT0 = 00...001

BIT1 = 00...010 << rla.w

BIT2 = 00...100 << rla.w

Binary Long Multiplication

Putting everything together

Variables we have   x in R5   y in R6

product in R12   init. R12 = 0

Variables we need

- counter to count through bits $j = 0, 1, \ldots 7$

  use R10   init. R10 = 0

  increase by 1 to next bit

- bitmask to check bits

  use R11   init. R11 = BIT0

  shift left for next bit

Binary Long Multiplication

It seems we have an algorithm that works and
that we can implement using our limited instruction set

Correct Result = Correct Algorithm + *Correct Handling of Numbers*

$$\begin{array}{r} \text{n-bit number} \\ \times\quad \text{n-bit number} \\ \hline \text{2n-bit number} \end{array}$$

We have to always watch for overflow!

```
;-------------------------------------------------------------------------------
; 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
;
```

Binary Long Multiplication

Pseudocode at the level of regs and instructions

Init : R12 = 0 accumulator
R10 = 0 bit counter 0, 1, ... 7
R11 = BIT0 bit mask

Repeat : test jth bit of R5 bit.w R11, R5
if bit is 1 i.e., C = 1
R12 += R6

Prep. for next bit
bit counter ++
left shift bitmask rla.w R11
left shift y in R6 rla.w R6
if bit counter ≤ 7
repeat

One last thing: Save reg's on stack and restore