Lecture 14

The Stack

Assignment Project Exam Help

https://tutorcs.com

WeChat: cstutorcs

Last Time: Subroutine Calling Sequence

Sequence o events after

call #div_by_16

- Current PC 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
```

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

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`**

Word Address

RAM

Word Address	RAM
0x1C00	
0x1C02	
0x1C04	
0x1C06	
. . .	. . .
0x23FE	

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** – it does *not* change during runtime

⇒ **Static allocation**

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

The Stack

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

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

Word Address

RAM

0x1C00

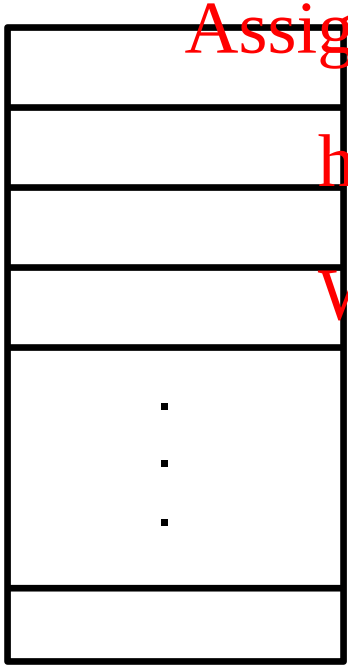
0x1C02

0x1C04

0x1C06

.
.
.

0x23FE



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

The stack enables **dynamic data allocation**

← **Stack** starts here

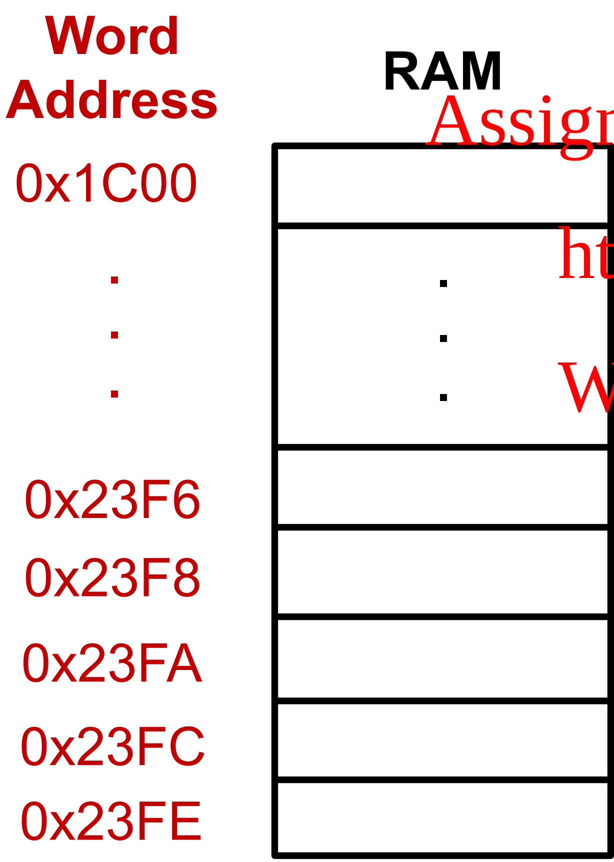
```
mov.w    #__STACK_END,SP     ; Initialize stackpointer
```

0x2400

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

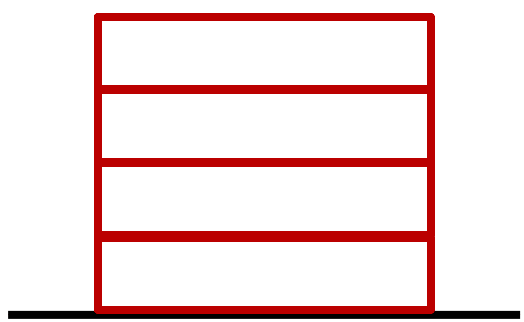
The Stack

The **stack** starts empty and is managed dynamically during runtime
i.e., we can add new data to the stack and remove it



Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

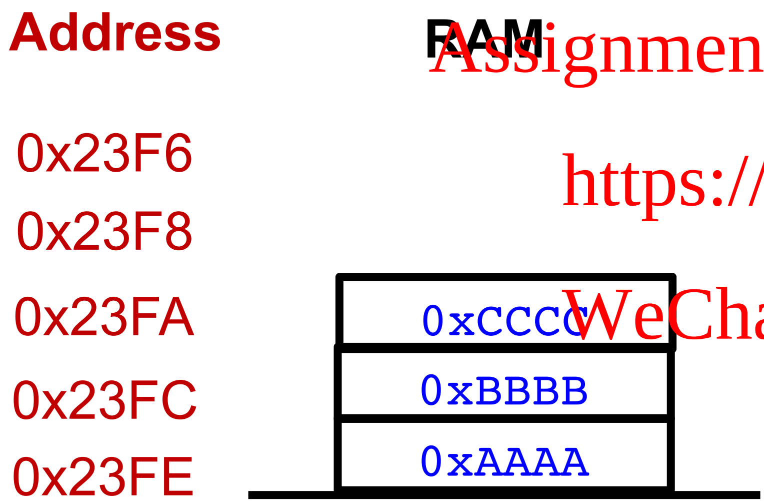
Always add to the top and remove from the top



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



Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

```
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)

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)**

Assignment Project Exam Help

At the beginning of each program the stack pointer is initialized

https://tutorcs.com

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

WeChat: cstutorcs

0x23F8

0x23FA

0x23FC

0x23FE

0x2400 – not in RAM !

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

`SP = 0x2400`

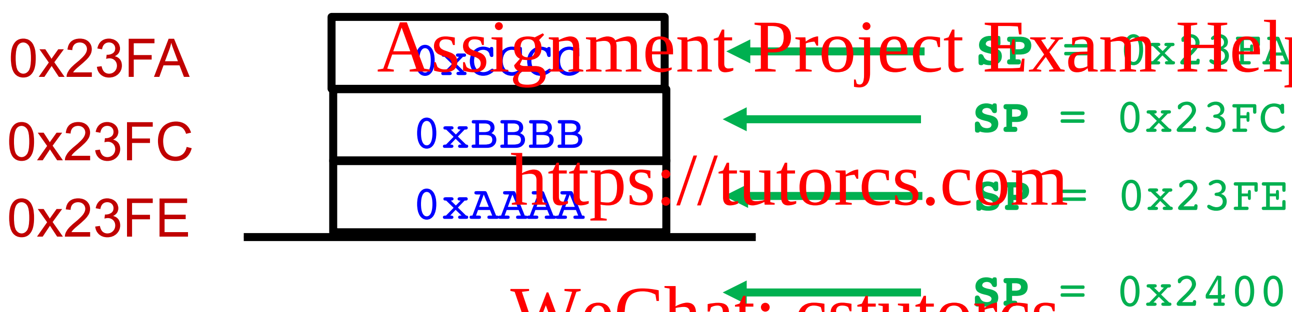
`SP` points here

Top of Stack – Stack Pointer (SP)

The stack pointer SP is **decremented** as we `push` elements onto stack

incremented as we `pop` elements from stack

The SP operates using a **pre-decrement, post-increment** scheme



Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs

```
push.w    #0xAAAA        SP = 0x23FE
push.w    #0xBBBB        SP = 0x23FC
push.w    #0xCCCC        SP = 0x23FA

pop.w     R4             SP = 0x23FC        R4 = 0xCCCC
pop.w     R5             SP = 0x23FE        R5 = 0xBBBB
pop.w     R6             SP = 0x2400        R6 = 0xAAAA
```

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

Assignment Project Exam Help

https://tutorcs.com

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

WeChat: cstutorcs

```
x_times_y:

            push    R5
            push    R6

            ; Compute R5*R6 by repeatedly adding R5 -- R6 times

            pop     R6
            pop     R5

            ret
```

mind the order of push and pop!

Not so efficient x_times_y

```
;-------------------------------------------------------------------------------
; 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
; Does not access any addressed memory
;-------------------------------------------------------------------------------
x_times_y:
; Compute R5*R6 by repeatedly adding R5 -- R6 times

            push    R6              ; save R6 on stack

            clr.w   R12             ; start with R12=0 before adding

check_R6:   tst.w   R6              ; R6 could be zero to start with, check before 1st add
            jz      ret_from_x_times_y

            add.w   R5, R12         ; R6 not zero, continue adding R5
            dec.w   R6              ; account for added R5 by decreasing R6
            jnz     check_R6

ret_from_x_times_y:
            pop     R6          ; restore R6 from stack
            ret
```

Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs