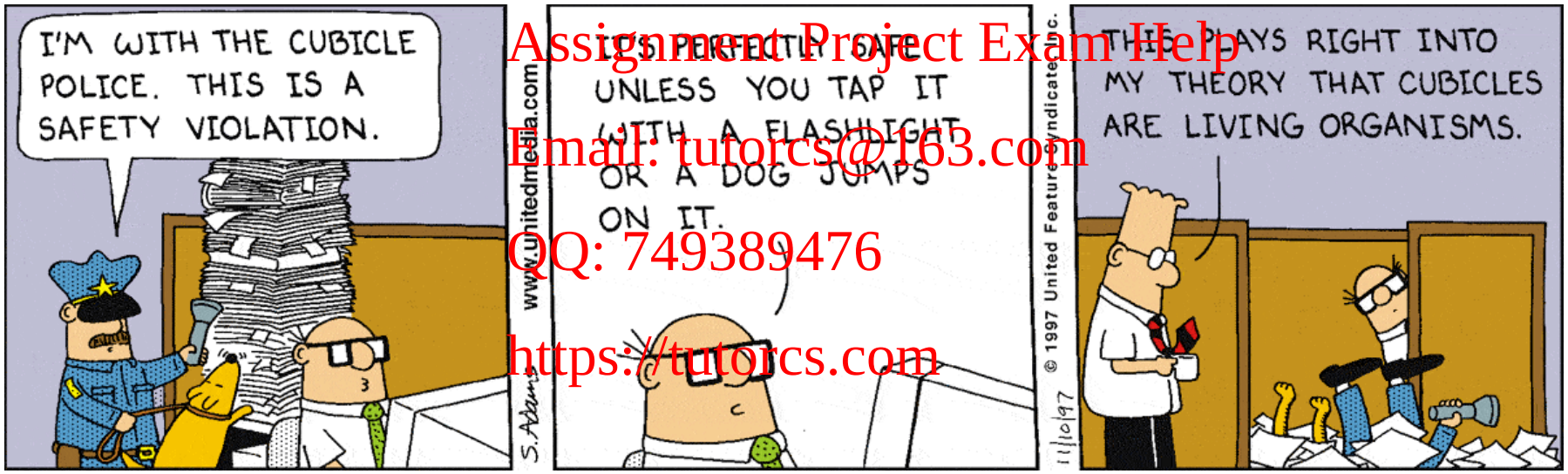
Lecture 15

More Subroutines

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



October 26, 2022

Last Time: The Stack

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

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

tine calls and interrupts use the stack 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

⇒ dynamic data allocation

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

← **Stack** starts here

0x2400 – not in RAM

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

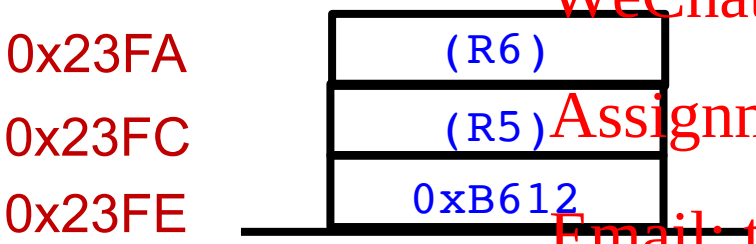
0x2400

Stack – Adding and Removing Data

We add data to the top of stack

we remove data from the top of stack

```
push.w   src
pop.w    dst
```




```
push.w   #0xB612
push.w   R5
push.w   R6

pop.w    R5      R5 <- (R6)
pop.w    R6      R6 <- (R5)
pop.w    R7      R7 <- 0xB612
```


Distinguish between

- removing data from stack (**pop**) and reading from the stack (using **mov**)
- adding data to stack (**push**) and writing to the stack (using **mov**)

Using Core Reg's as Local Variables

Subroutines need "local variables" – i.e., space to manipulate data

e.g., for counters, indices etc.

Often a subroutine is not s  are used by the calling program modify core registers (except for output)


```
;-------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------

            mov.w   #8, R4              ; init index, counting down 8 to 0

next:       mov.w   x_array(R4), R5     ; R5 = x
            mov.w   y_array(R4), R6     ; R6 = y

            call    #is_divisible       ; check if x divides 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
;-------------------------------------------------------------------
```

Using Core Reg's as Local Variables

Subroutines need "local variables" – i.e., space to manipulate data

e.g., for counters, indices etc.

Often a subroutine is not  modify core registers (except for output)

s are used by the calling program

Solution: Subroutine preserves core registers on the stack – via **`push.w`**

And recovers them from the stack before returning – via **`pop.w`**

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 every subroutine cleans up after itself: whatever it pushes onto stack it has to pop back !!!

Using Core Reg's as Local Variables

What is the issue with this subroutine?

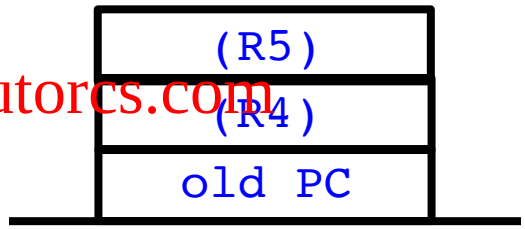
```
Wreak_havoc:
            push.w  R4
            push.w  R5

            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
```

From least to worst:

- R4, R5 can be modified
- **Memory leak** of 2 words – i.e., 2 words left behind on the stack when returning from `***`
- When returning from `***` the PC is restored from the Top of Stack (TOS)

`(R5) -> PC !!!!`

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



Stack

with `push R5`

with `push R4`

with `call #W...`

Using Core Reg's as Local Variables

A subroutine can return from multiple points

No matter what the point **must 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
```

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

            tst.w   R6
            jne     Continue

            jmp     Ret_here

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

Ret_here:   pop.w   R5
            pop.w   R4

            ret
```

Infinitely better !!!

Using Core Reg's as Local Variables

A subroutine can return from multiple points

No matter what the point **must clean up the stack**

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

            tst.w   R6
            jne     Continue

            jmp     Ret_here

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

Ret_here:   pop.w   R5
            pop.w   R4

            ret
```

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

            tst.w   R6
            jeq     Ret_here

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

Ret_here:   pop.w   R5
            pop.w   R4

            ret
```

Even better !!!

Using Core Reg's as Local Variables

A subroutine can return from multiple points

No matter what the point **must clean up the stack**

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

            tst.w   R6
            jeq     Ret_here

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

Ret_here:   pop.w   R5
            pop.w   R4

            ret
```



```
Wreak_havoc:
            tst.w   R6
            jne     Continue

            ret

Continue:   push.w  R4
            push.w  R5

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

            pop.w   R5
            pop.w   R4

            ret
```

Yet another alternative

Today's Coding Task

Task: implement binary long multiplication in assembly

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

Logic Instructions in MSP430

Bitwise AND

```
and.w   src, dst        ; dst = src & dst
```

Bitwise XOR

```
xor.w   src, dst        ; dst = src ^ dst
```

Bit Set

```
bis.w   src, dst        ; dst = src | dst
```

Bit Clear

```
bic.w   src, dst        ; dst = src & ~dst
```

Status Bits unchanged

Bitwise Test

```
bit.w   src, dst        ; sets C bit according to
                        ; src & dst – dst unchanged
```

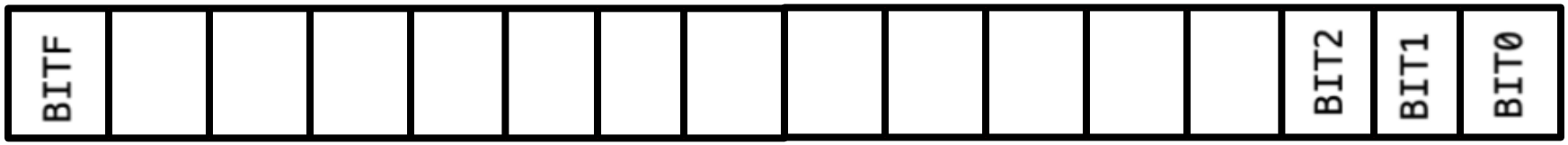
Invert Bits

```
inv.w   dst             ; dst = ~dst
```

Bitmasks

We will use **bitmasks** for setting, clearing or testing bits defined in "msp430fr69891.h"

```
#define BIT0                (0x0001)
#define BIT1                (0x0002)
#define BIT2                (0x0004)
#define BIT3                (0x0008)
#define BIT4                (0x0010)
#define BIT5                (0x0020)
#define BIT6                (0x0040)
#define BIT7                (0x0080)
#define BIT8                (0x0100)
#define BIT9                (0x0200)
#define BITA                (0x0400)
#define BITB                (0x0800)
#define BITC                (0x1000)
#define BITD                (0x2000)
#define BITE                (0x4000)
#define BITF                (0x8000)
```

Binary Long Multiplication


```
    1101   <-
  x 1011   <-
  ------
    1101
   1101  <<
  0000  <<
+ 1101  <<
  ------
 10001111
```

BIT0 of y is 1 — add x

BIT1 of y is 1 — shift left x, add x

BIT2 of y is 0 — shift left x, no add

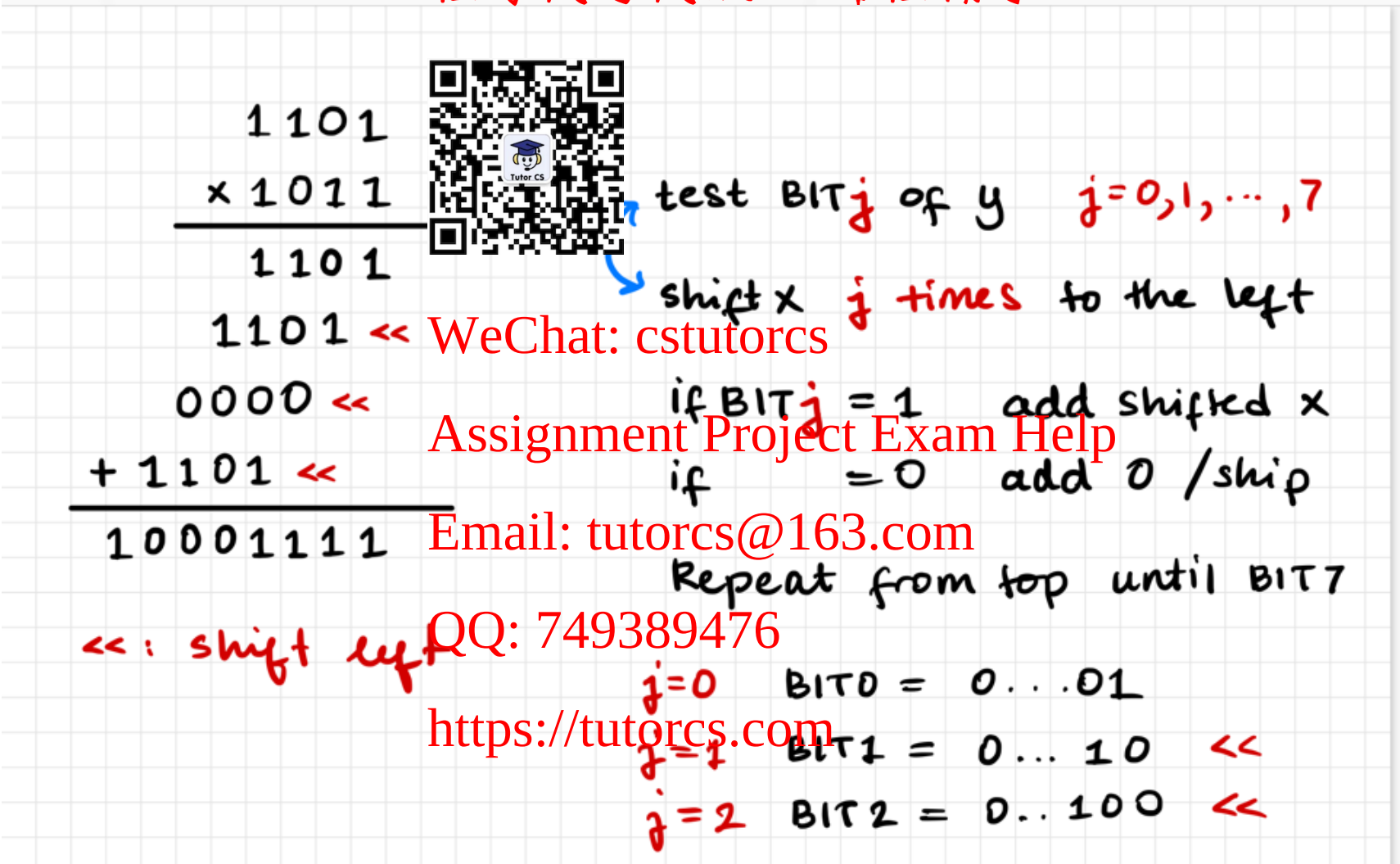
BIT3 of y is 1 — shift left x, add x

<< : 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



Binary Long Multiplication

How do we test 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

x in R5 y in R6

product in R12 init. R12=0

Variables we need

- counter to count through bits j=0,1,...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

rla.w R11

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*

n-bit number

x n-bit number

2n-bit number

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 counter 0, 1, ... 7

R11 = 0x01 bit mask

Repeat : test jth bit of R5 bit.w R11, R5

if bit is 1 i.e., C = 1

R12 += R6

bit counter ++

left shift bit mask rla.w R11

left shift y in R6

if bit counter ≤ 7

repeat

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

Binary Long Multiplication

Pseudocode at the level of regs and instructions

Init : R12 = 0 (accumulator)

R10 = 0 (counter) 0, 1, ... 7

R11 = 0x0080 (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 bit mask — rla.w R11

left shift y in R6 — rla.w R6

if bit counter ≤ 7

repeat

Binary Long Multiplication v.1


```
;
x_times_y:

; Save afftected core registers to stack - You can add this part last once you
; know which registere
            push.w  R6
            push.w  R1
            push.w  R1

            clr.w   R12             ; R12 will accumulate R5*R6
            clr.w   R10             ; R10 will index bits j = 0, 1, ..., 7
            mov.w   #BIT0, R11      ; R11 has the bitmask to use with tst.w

check_next_bit:
            bit.w   R11, R5         ; Is the jth bit 1?
            jnc     prep_next_bit   ; If not prepare for checking next bit

            add.w   R6, R12         ; Bit j is 1, add

prep_next_bit:
            rla.w   R11             ; Prepare next bitmask
            rla.w   R6              ; Prepare shifted version of R6
            inc.w   R10             ; increase bit index
            cmp.w   #8, R10         ; Are we done with all bits?
            jlo     check_next_bit

; Restore saved core registers from stack
; Watch the order and make sure not to leave anything behind
            pop.w   R11
            pop.w   R10
            pop.w   R6

            ret
```

Binary Long Multiplication v.1.1


```
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
```


How you count through the eight bits does not matter

Here bits are counted 8, 7, …, 2, 1