ecture 7

Firs structions II

Last Time

Five instructions

```
mov.w      src,
add.w      src,
rra.w      dst
jmp        label
nop
```

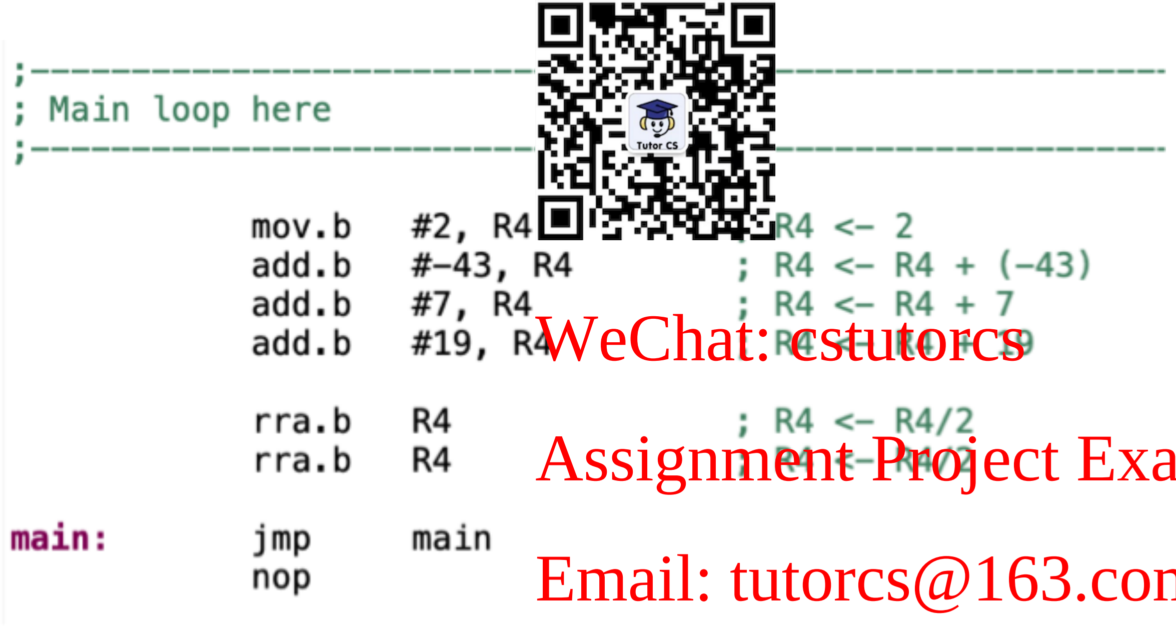
These instructions also have a byte version

Three addressing modes

- **Immediate data:** `src` is the value given after **#**
- **Absolute address:** the address of the `src` or `dst` is given after **&**
- **Register mode:** `src` or `dst` is one of the core registers **R**0 – **R**15

First Code

First task: Find the average value of the set of numbers {2, -43, 7, 19}



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

            mov.b   #2, R4                  ; R4 <- 2
            add.b   #-43, R4                ; R4 <- R4 + (-43)
            add.b   #7, R4                  ; R4 <- R4 + 7
            add.b   #19, R4                 ; R4 <- R4 + 19

            rra.b   R4                      ; R4 <- R4/2
            rra.b   R4                      ; R4 <- R4/2

main:       jmp     main
            nop
```

Today we will redo this

- Introducing assembler directives
- *Variables* and *arrays*
- More addressing modes

Assembler Directives

Assembler directives supply program data and control the assembly process

We will use them to

- Assemble code and data into specified sections

```
.data     ; Everything after this goes to RAM
.text     ; Everything after this goes to FRAM
```

- Reserve space in memory (initialized to zero)

```
.space 6   ; Reserve 6 bytes of space
```

- Initialize memory to desired values

```
.word 0xB, 0xC     ; initialize words
.byte -1, 5, 3     ; initialize bytes
```

- Define global variables

```
array:      .word 0x1, 0x2, 0x3, 0x4
```

- Define symbolic constants – no memory reserved

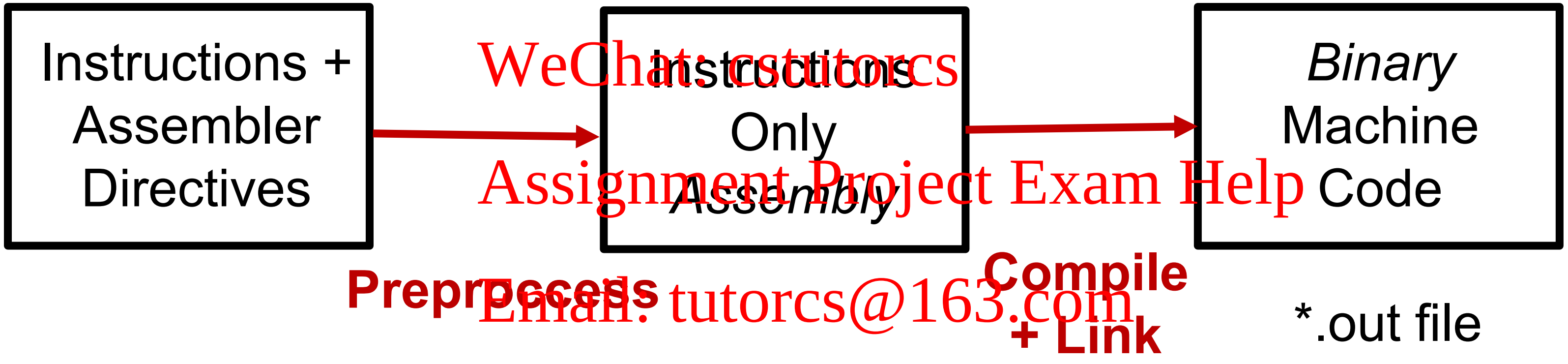
```
scon:       .set 4
```

Assembly to Machine Code

The hammer icon 🔨 on CCS initiates the **build** of the code

Build = Preprocess + Compile + Link

Simplified Picture

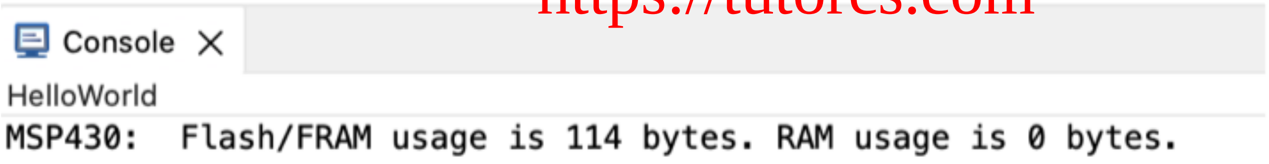


Instructions + Assembler Directives
Instructions Only *Assembly*
Binary Machine Code
Preprocess
Compile + Link
*.out file

The bug icon 🐞 uploads the binary machine code to the FRAM and also initiates memory in RAM and FRAM (per preprocessor directions)

Assembly to Machine Code

Assembly Code		Machine Code	Address of Instruction
mov.w	#__STACK_END, SP	4031 2400	0x4400
mov.w	#WDTPW|WDTHOLD, &WDTCTL	40B2 5A80 015C	0x4404
mov.b	#2, R4	4364	0x440a
add.b	#-43, R4	5074 FFD5	0x440c
add.b	#7, R4	5074 0007	0x4410
add.b	#19, R4	5074 0013	0x4414
rra.b	R4	1144	0x4418
rra.b	R4	1144	0x441a
jmp	main	3FFF	0x441c
nop		4303	0x441e



```
HelloWorld
MSP430:  Flash/FRAM usage is 114 bytes. RAM usage is 0 bytes.
```

Memory usage reported after code upload

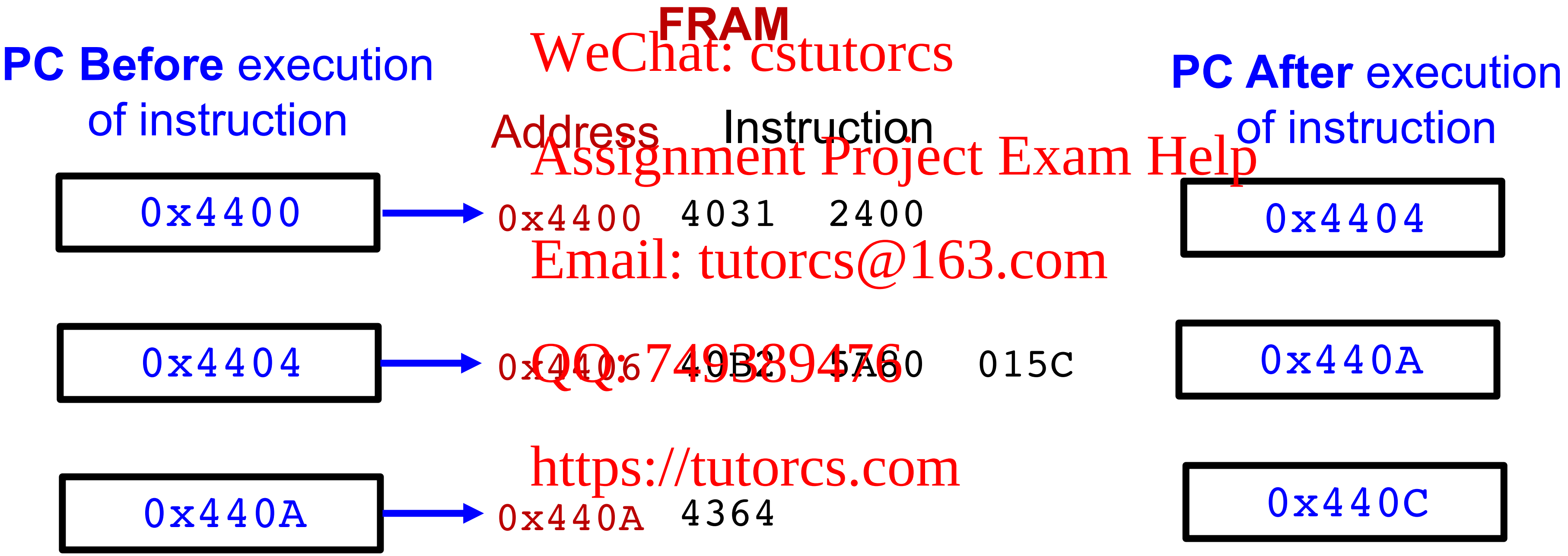
The Program Counter R0/PC

The core register R0 is the **Program Counter PC**

The **program counter p** **e next instruction to be executed**

i.e.,

when we look into the PC /e see the address of the next instruction



PC Before execution of instruction	FRAM Address	FRAM Instruction	**PC After** execution of instruction
0x4400	0x4400	4031 2400	0x4404
0x4404	0x4406	40B2 5A80 015C	0x440A
0x440A	0x440A	4364	0x440C

Variables in MSP430 Assembly

We will use assembler directives to reserve and initialize data in memory

We will use labels to nam[illegible]s and use **absolute address mode (&)**

or **symbolic address mode**

Task: Define word variables x = 5 and y = 8 in RAM and reserve space for word variable sum

```
        .data
x:      .word    5
y:      .word    8
sum:    .space   2
```

A label is simply a name for an address

```
x  =  0x1C00
y  =  0x1C02
sum  =  0x1C04
```

Symbolic address mode

```
mov.w   x, R4
add.w   y, R4
mov.w   R4, sum
```

Task: Add x and y and store in sum

Arrays in MSP430 Assembly

There is no actual array construct in assembly

We will emulate arrays us bler directives and labels

```
                array1          +2      array1+4          address

array1: .word   0x0100,   0x0200,   0x0300
```

```
                array2    array2+1    array2+2           address

array2: .byte   0x01,       0x02,       0x03
```

We will have to be careful with byte and word arrays

Indexed Mode of Addressing

Syntax of **indexed mode**

```
array1: .word   0x0100, 0x0200,   0x0300

        mov.w   array1(R4), R5
```

e.g.:

```
        mov.w   #2, R4
        mov.w   array1(R4), R5
```

same as

```
        mov.w   &array1+2, R5
```

Indexed Mode and **Byte** Arrays

Rewrite our previous example using indexed mode

```
array2: .byte   0x10,             0x30

        mov.b   &arra
        add.b   &array2+1, R5
        add.b   &array2+2, R5
```

```
array2: .byte   0x10, 0x20, 0x30

        mov.w   #0, R4            ; R4 = 0 will be the index
        mov.b   array2(R4), R5    ; R5 = array2[R4]
        inc.w   R4                ; R4++
        add.b   array2(R4), R5    ; R5 += array2[R4]
        inc.w   R4                ; R4++
        add.b   array2(R4), R5    ; R5 += array2[R4]
```

Indexed Mode and **Word** Arrays

```
array1: .word   0x0100,  0x0200,  0x0300

        mov.w   &arra
        add.w   &arra
        add.w   &arra
```

```
array1: .word   0x0100,  0x0200,  0x0300

        mov.w   #0, R4                ; R4 = 0 will be the index
        mov.w   array2(R4), R5        ; R5 = array2[R4]
        inc.w   R4                    ; R4++
        inc.w   R4                    ; R4++
        add.w   array2(R4), R5        ; R5 += array2[R4]
        inc.w   R4                    ; R4++
        inc.w   R4                    ; R4++
        add.w   array2(R4), R5        ; R5 += array2[R4]
```