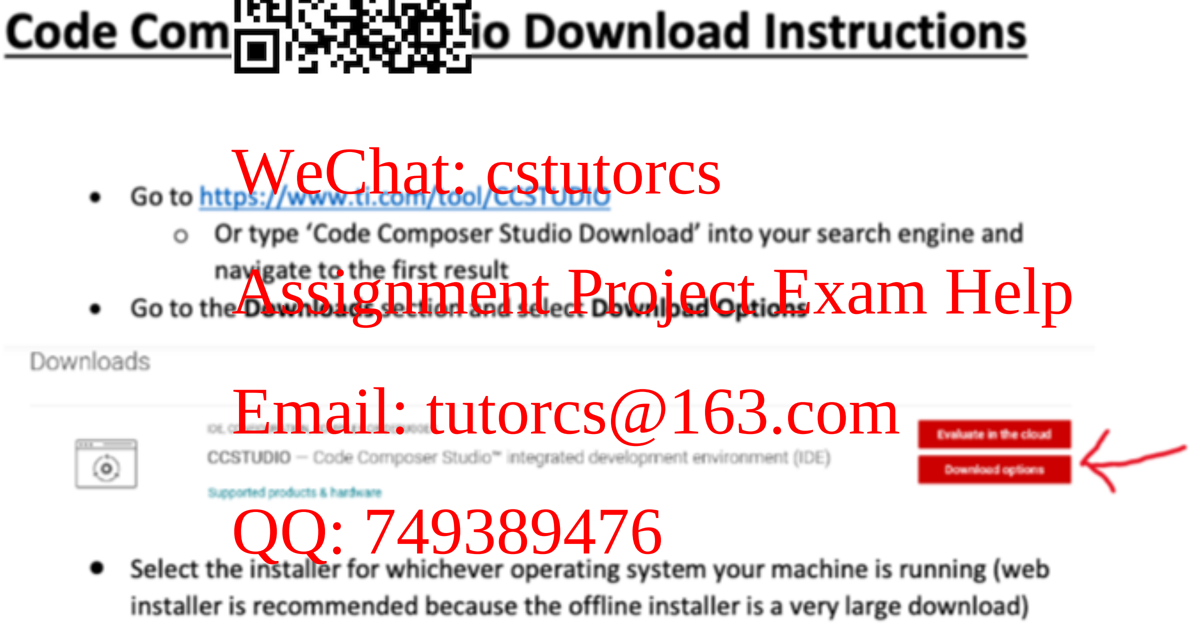
Lecture 6

First Instructions

Code Composer Studio

Starting today, we will need CCS – not necessarily in class but for all assignments from now on

Posted a guide on how to [illegible]S – Find it under Resources in Carmen



Code Com[illegible]io Download Instructions

- Go to https://www.ti.com/tool/CCSTUDIO
  - Or type 'Code Composer Studio Download' into your search engine and navigate to the first result
- Go to the Downloads section and select Download Options

Downloads

[illegible]

CCSTUDIO – Code Composer Studio™ integrated development environment (IDE)

Supported products & hardware

- Select the installer for whichever operating system your machine is running (web installer is recommended because the offline installer is a very large download)

CCS does not work on tablets (iPad etc.)

You can use ECE Windows Computing Labs – all machines have CCS https://ets.osu.edu/labs

Code Composer Studio

Code Composer Studio is an **Integrated Development Environment (IDE)**

We will use CCS to

- Write/edit assembly language programs
- Compile/build assembly language programs into machine language code
- Load the code into the FRAM of the MCU
- Debug the program as it runs in the CPU
  - Single step the program
  - Set breakpoints
  - View memory locations: core registers, RAM, FRAM etc.

Assembly Language

A typical line of assembly language has four parts

```
StopWDT:    mov.w   #W          HOLD,&WDTCTL   ;Stop watchdog timer
```

Label   **Operation**   **Operands**   **Comment**

Label: starts in the first column and may be followed by a colon (:)   optional

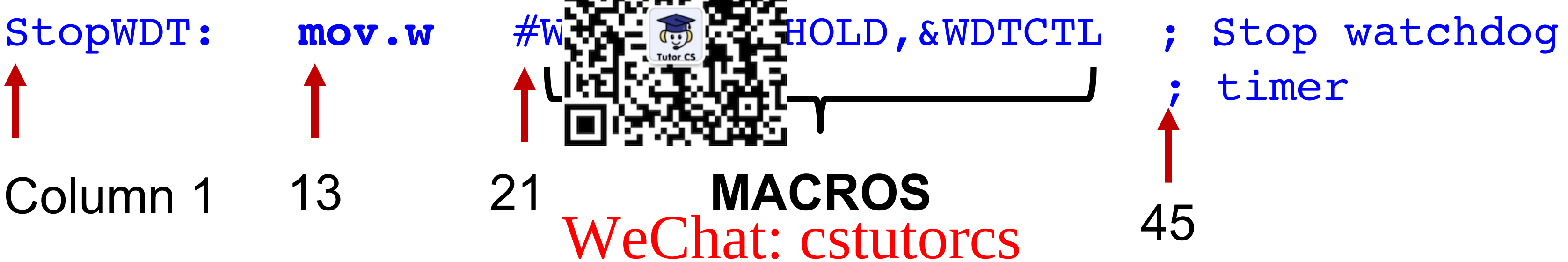
Operation: **instruction** for CPU or **assembler directive**

Operands: data needed for the operation, nature depends on the operation

Comment: the rest of the line, following a semicolon (;)
assembler skips comments, but critical for the programmer

Coding Style Guidelines

TI recommended coding style guidelines for assembly are



```
StopWDT:    mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog
                                            ; timer
```

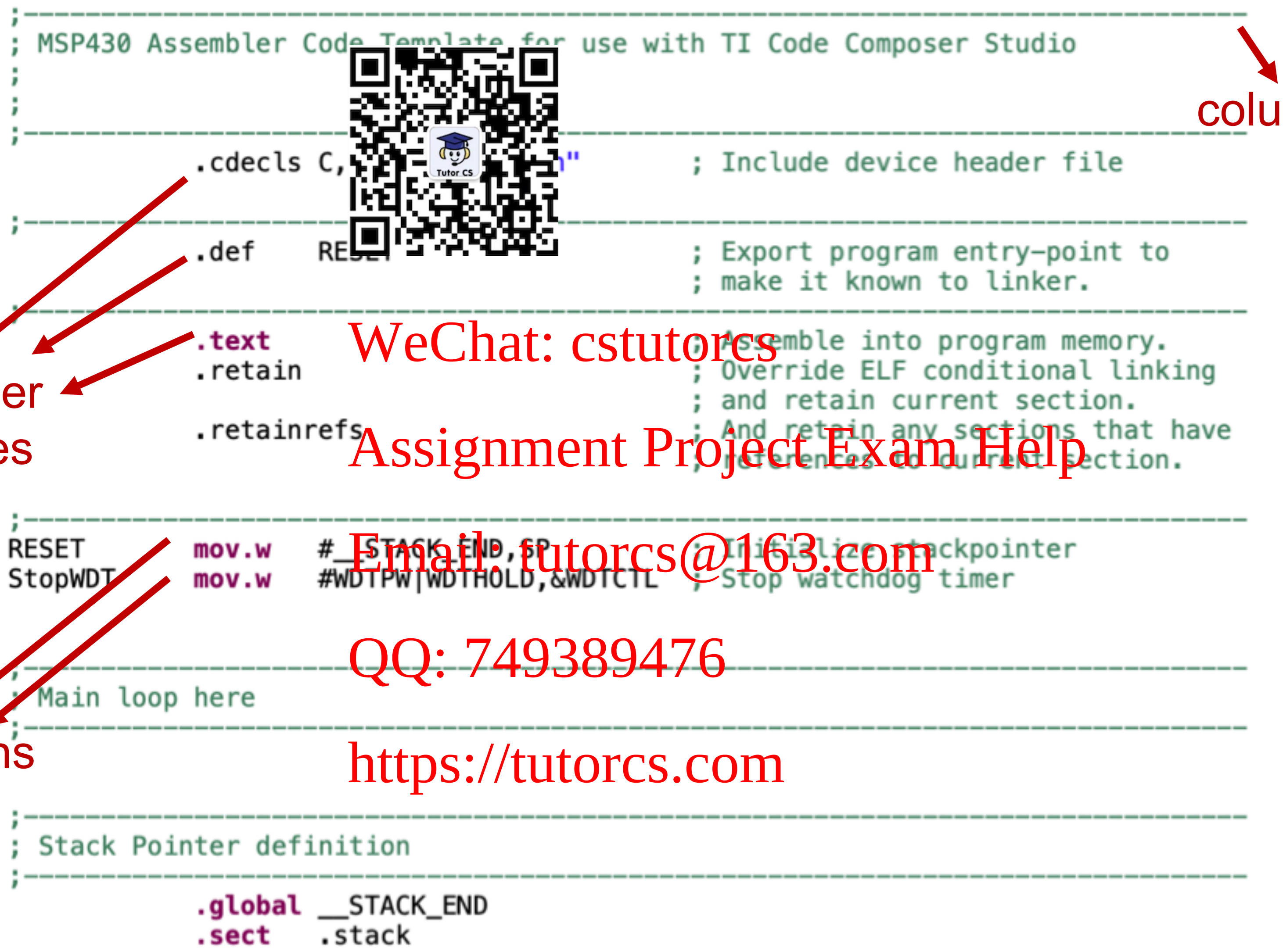
1. No line should exceed 80 characters
2. **Use macros provided in the MSP430 header file**
3. **Labels start in column 1** (and are 10 characters or fewer) **Critical!**
4. **Operators start in column 13**
5. **Operands start in column 21**

Please follow

6. Comments start in column 45, the first word is capitalized
7. For multiline comments, additional lines are not capitalized

Comments are very important, but format can be more flexible

MSP430 Assembler Code Template



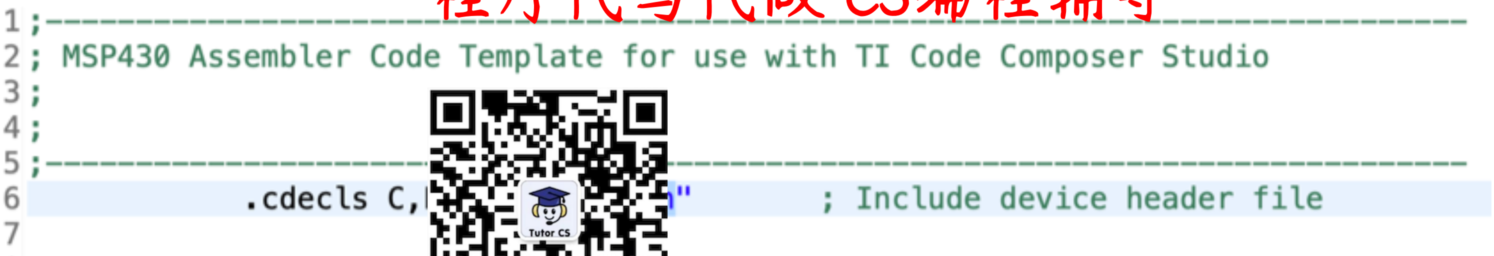
```
;-------------------------------------------------------------------------------
; MSP430 Assembler Code Template for use with TI Code Composer Studio
;
;
;-------------------------------------------------------------------------------
            .cdecls C,[illegible]"      ; Include device header file

;-------------------------------------------------------------------------------
            .def    RE[illegible]               ; Export program entry-point to
                                            ; make it known to linker.
;-------------------------------------------------------------------------------
            .text                           ; Assemble into program memory.
            .retain                         ; Override ELF conditional linking
                                            ; and retain current section.
            .retainrefs                     ; And retain any sections that have
                                            ; references to current section.

;-------------------------------------------------------------------------------
RESET       mov.w   #__STACK_END,SP         ; Initialize stackpointer
StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer


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


;-------------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack
```

column 80

assembler directives

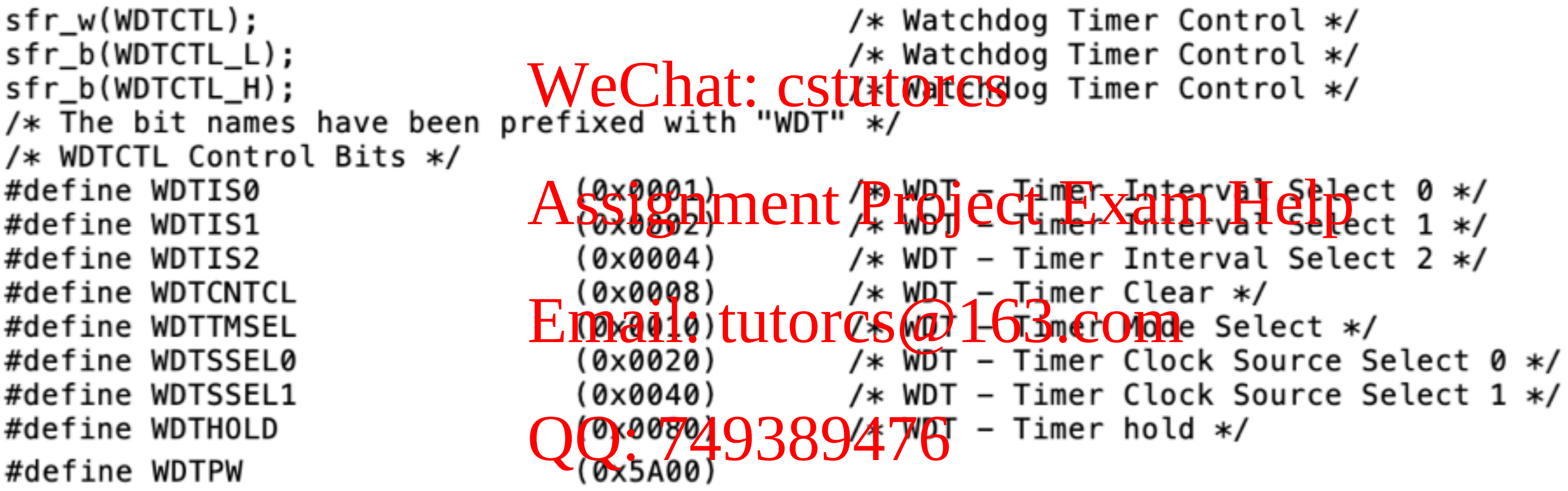
instructions

MSP430 Header File - Macros



```
;-------------------------------------------------------------------------------
; MSP430 Assembler Code Template for use with TI Code Composer Studio
;
;
;-------------------------------------------------------------------------------
            .cdecls C,[illegible]"          ; Include device header file

```

Search for “**msp430fr698**[illegible]our computer



```
sfr_w(WDTCTL);                              /* Watchdog Timer Control */
sfr_b(WDTCTL_L);                            /* Watchdog Timer Control */
sfr_b(WDTCTL_H);                            /* Watchdog Timer Control */
/* The bit names have been prefixed with "WDT" */
/* WDTCTL Control Bits */
#define WDTIS0              (0x0001)        /* WDT - Timer Interval Select 0 */
#define WDTIS1              (0x0002)        /* WDT - Timer Interval Select 1 */
#define WDTIS2              (0x0004)        /* WDT - Timer Interval Select 2 */
#define WDTCNTCL            (0x0008)        /* WDT - Timer Clear */
#define WDTTMSEL            (0x0010)        /* WDT - Timer Mode Select */
#define WDTSSEL0            (0x0020)        /* WDT - Timer Clock Source Select 0 */
#define WDTSSEL1            (0x0040)        /* WDT - Timer Clock Source Select 1 */
#define WDTHOLD             (0x0080)        /* WDT - Timer hold */

#define WDTPW               (0x5A00)
```

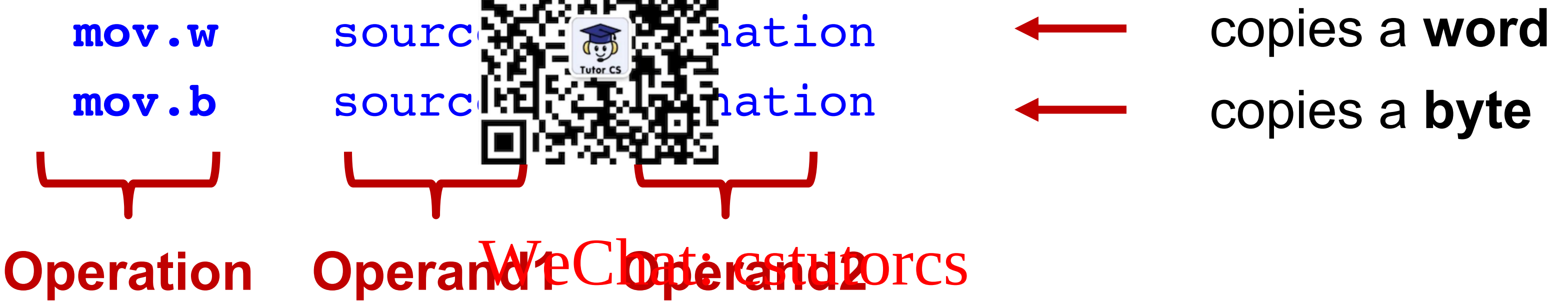
```
mov.w   #WDTPW|WDTHOLD,&WDTCTL  ;Stop watchdog timer
```

MACROS

| is logic OR

The Move Instruction

The **move instruction** *copies* a byte or word specified in the source to the destination



```
mov.w     sourc      ation
mov.b     sourc      ation
```

The source is not affected by this operation – copy rather than move

If the suffix (i.e., `.w` or `.b`) is omitted, the default is `mov.w`

Best practice is to always explicitly specify the suffix

There are several options to specify the `source` and `destination`

⇒ **Addressing modes**

Addressing Modes

At this point we will look at three modes – more will follow

- **Immediate data** using [illegible]

  The value (byte [illegible] given after #

- **Absolute address** usi[illegible]

  The absolute address of the byte or word is given after &

- **Register mode** using R

  The source/destination is one of the core registers R0 – R15

e.g.:

```
mov.w   #WDTPW|WDTHOLD,   &WDTCTL
```

Immediate

#0x5A80

Absolute

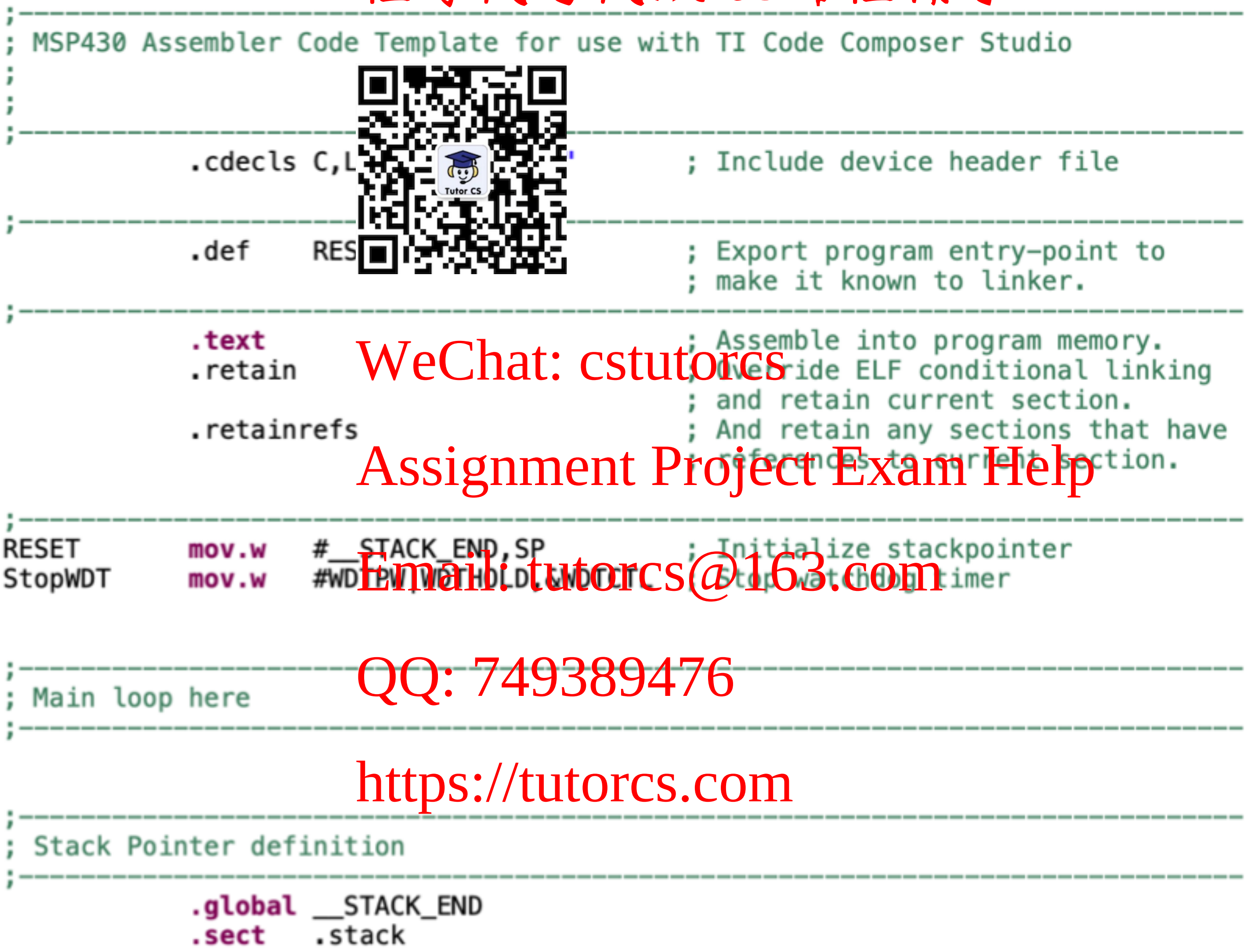
&0x051C

address of watchdog control register

Addressing Modes

e.g.:

mov



```
;-------------------------------------------------------------------------------
; MSP430 Assembler Code Template for use with TI Code Composer Studio
;
;
;-------------------------------------------------------------------------------
            .cdecls C,L                        ; Include device header file

;-------------------------------------------------------------------------------
            .def    RES                        ; Export program entry-point to
                                               ; make it known to linker.
;-------------------------------------------------------------------------------
            .text                              ; Assemble into program memory.
            .retain                            ; Override ELF conditional linking
                                               ; and retain current section.
            .retainrefs                        ; And retain any sections that have
                                               ; references to current section.

;-------------------------------------------------------------------------------
RESET       mov.w   #__STACK_END,SP            ; Initialize stackpointer
StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL     ; Stop watchdog timer


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


;-------------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack
```

er

mov

r R4

R4

mov

of core

of R5

R5

Addressing Modes

```
mov.w           ,   destination
mov.b           ,   destination
```

Immediate #
Absolute &
Register R

Absolute &
Register R

Immediate values can be **binary**, **hexadecimal** or **decimal**

```
mov.b   #00001000b, R4
mov.b   #0x8, R5
mov.b   #16, R6
```

0x0008	R4
0x0008	R5
0x0010	R6

Addressing Modes

e.g.:

```
mov.w   R4, R5
```

copy contents of R4 to R5

```
mov.w   R4, &0x1C00
```

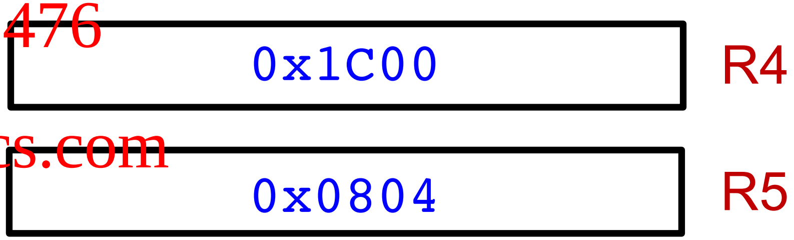
copy contents of R4 to memory location with address 0x1C00

```
mov.w   #0x0804, &0x1C00
```

copy value 0x0804 to memory location with address 0x1C00

Popquiz

```
mov.w   #0x0804, &0x1C00

mov.w   #0x1C00, R4

mov.w   &0x1C00, R5
```



Basic Arithmetic: add

The **`add`** instruction adds the *source* to the *destination*

`destination += source`

```
mov.w             destination
mov.b    source,  destination
```

Immediate #
Absolute &
Register R

Absolute &
Register R

Basic Arithmetic: `rra`

The **`rra`** instruction shifts all bits one position to the right and fills the void by replicating the most signif

`rra` is short for roll right a

and corresponds to dividing a signed number by 2

The **`rra`** instruction takes only one operand

```
rra.w     dst
rra.b     dst
```

Absolute & Register R

Division by a Power of Two

To divide a **signed number by 2^m**

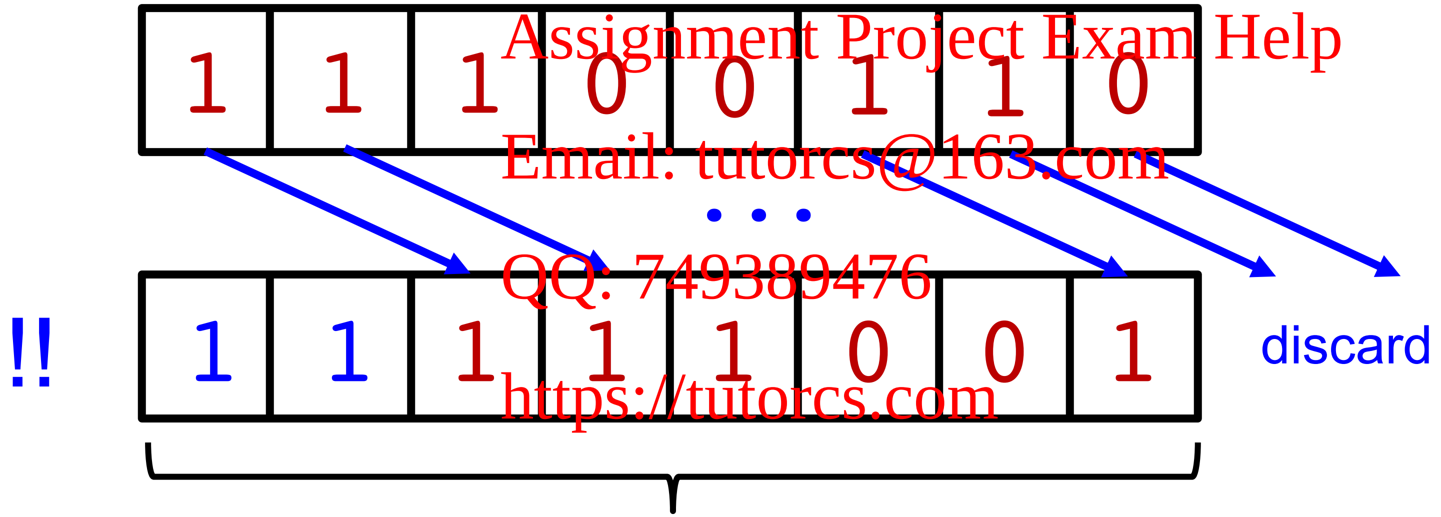
- **Shift m-bits to the rig** **with the most significant bit** **"sign bit"**
- The answer will not be e are discarding the fractional part

king the floor function: $\lfloor -6.5 \rfloor = -7$

e.g.: –26 ÷ 4

4 = 2^2 ⇒ Shift 2 bits to the right

```
rra.b     dst
rra.b     dst
```



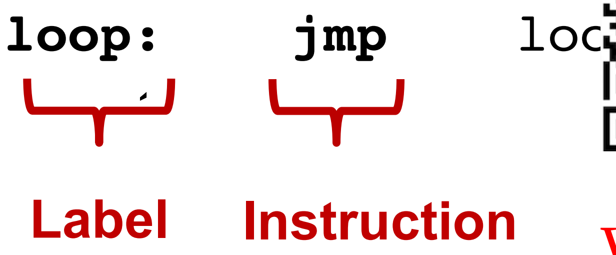
1	1	1	0	0	1	1	0

1	1	1	1	1	0	0	1

249 in decimal ⇒ **- 7**

The Infinite Loop

At the end of (almost) every program we write we will add an infinite loop



Unconditionally jump to the label loop and continue to execute instructions from that point on

Prevents the program counter (PC) from proceeding to the next word written in FRAM and executing the random data in there

The compiler gives a warning when the last instruction of a program is a jump

Hence, we will follow with a `nop` – an operation that does nothing

Until next notice, every program we write will end with

```
loop:     jmp     loop
          nop
```

You can use a different label name