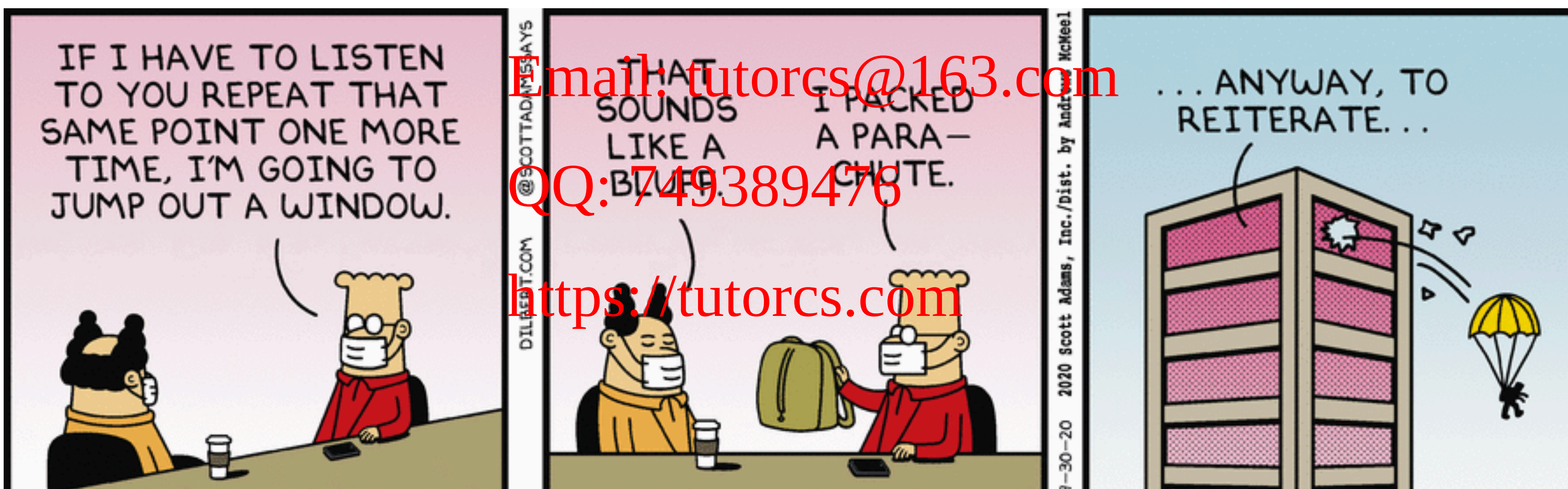
Lecture 9

Status Register, Conditional Jumps & Flow Control



Announcements

Midterm 1 will be posted next Wednesday February 15

dnesday February 22 before class

- You will write code to c specific task on an array
- I will ask for one more onditions

What you need to know? Everything until the end of Lecture 11

- Instructions and addressing modes, array addressing
- Conditional jump instructions
- Flow control" Loops and if statements in assembly (start)

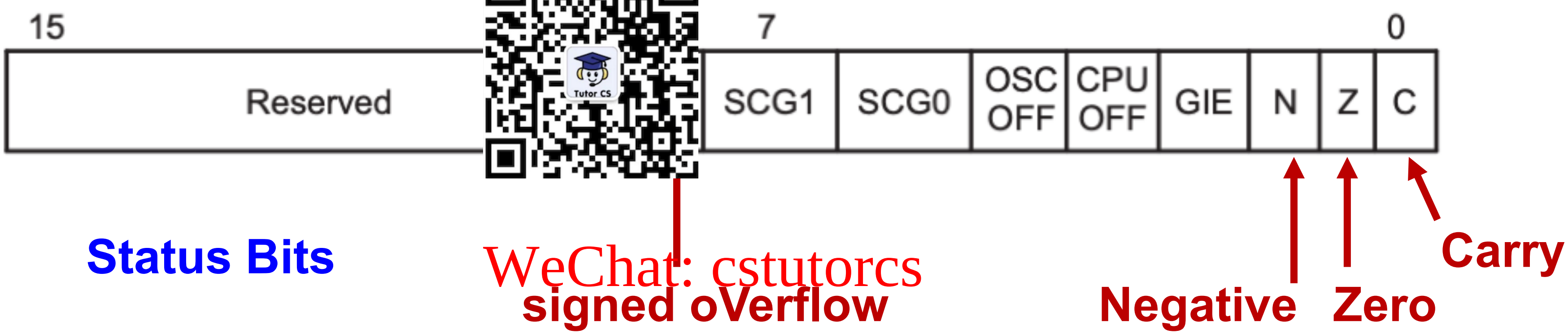
Practice opportunity: **Quiz 4 will be posted later today**

due Wednesday February 15 before class

- Office hours: Tuesdays 1 pm – 3 pm Dreese Lab 259

Status Register SR/R2

The core register R2 has a special function: **Status Register SR**



The C, Z, N, V flags are set/cleared after arithmetic and logic operations

not after move

Zero is set when the result of an operation is 0

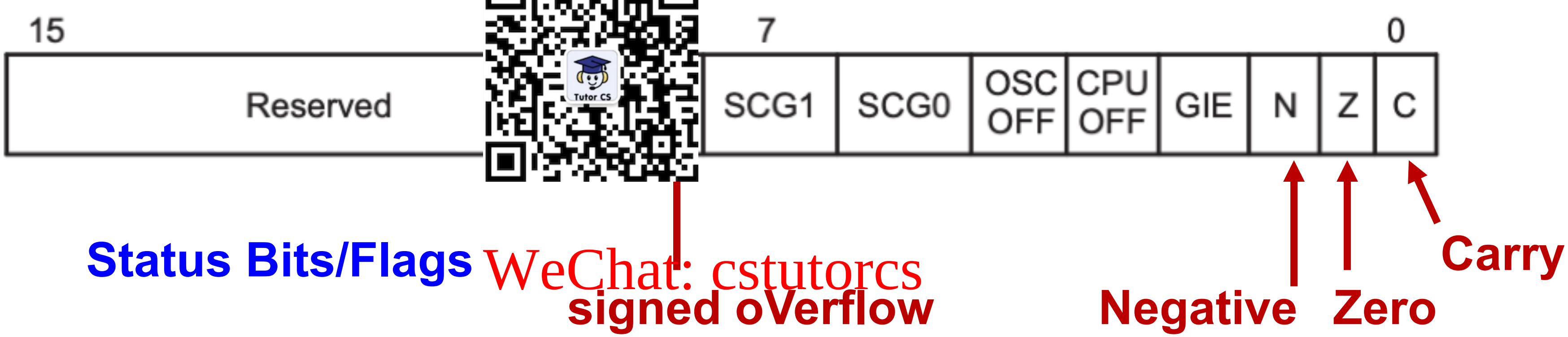
cleared when the result is not 0

Negative is set when the result of an operation is negative

cleared when the result is positive

Status Register SR/R2

The core register R2 has a special function: **Status Register SR**



Carry is set when the result of an operation produces a **carry/borrow**
cleared when no **carry/borrow** occurs

Carry: overflow into 9th or 17th bit !

signed oVerflow is set when the result of an arithmetic operation overflows the signed-variable range

Basic Arithmetic Instructions

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

```
add.w    src,  dst          dst += source
```

The **sub** instruction subtracts the *source* from the *destination*

```
sub.w    src,  dst          dst -= source
```


There are multiple instructions with **one** operand


```
inc.w    dst                dst++
dec.w    dst                dst--

incd.w   dst                dst += 2
decd.w   dst                dst -= 2
```


All these instructions **modify** the destination and set the status bits in SR

Example

The **zero** bit is set when the result of an arithmetic or logic operation is zero

e.g.:

```
sub.w   src, dst
```

$$Z = \begin{cases} 1 & \texttt{if (src == dst)} \\ 0 & \texttt{if (src != dst)} \end{cases}$$

This is *similar* to `if (src == dst)`

We can check the **zero bit and decide on the program flow**

- If the zero bit is set, we know that `src == dst`
- If the zero bit is not set, we know that `src != dst`

(There is an instruction to check if a bit is set or not: **`bit.w`**)

Instead: We use the **correct conditional jump to control the program flow**

Comparison Only

Sometimes we want to set the status bits **without changing the value of the destination**

```
cmp.w   src, dst
```

This instruction sets the status bits according to the outcome of `(dst – src)`

But it does not change the destination!

There is a special version

```
tst.w   dst          same as      cmp.w    0, dst
```

- does **not** change the value of `dst`
- only sets status bits according to operation `(dst – 0)`

Then we use a **conditional jump to control the program flow**

Jump Instructions

Jumps can be **unconditional** or **conditional**

Unconditional jump `jmp` jump to the given label

e.g.

```
Loop:      jmp   Loop
```


Syntax

```
jmp   label
```

jump to label unconditionally

Effect: Program execution continues from instruction marked with `label` which can be before or after `jmp` instruction

`jmp` does not encode the absolute address of the label, but a relative offset within ~ +/- 1 KiB

PC is updated by (PC + offset)

offset > 0 if label is after jmp
offset < 0 if label is before jmp

Conditional Jump Instructions

There are two overlapping sets of conditional jump instructions

- named after the status fter an arithmetic/logic operation

or

- based on an explicit c instruction `cmp.w src, dst`

Conditional jump instructions named after status bits

`jc`	`label`	jump to label **if** carry set (i.e., C = 1)
`jnc`	`label`	jump to label **if** carry not set (i.e., C = 0)
`jn`	`label`	jump to label **if** negative (i.e., N = 1)
`jz`	`label`	jump to label **if** zero (i.e., Z = 1)
`jnz`	`label`	jump to label **if** nonzero (i.e., Z = 0)

Conditional Jump Instructions

Conditional jump instructions based on explicit comparison

`cmp.w`	`src, dst`	`status bits based on dst-src`
`tst.w`	`dst`	`lated instruction cmp.w #0, dst`

`jeq label`	jump **if** equal	`jz label`
`jne label`	jump **if** not equal	`jnz label`
`jhs label`	jump **if** higher or same – unsigned	`jc label`
`jlo label`	jump **if** lower – unsigned	`jnc label`
`jge label`	jump **if** greater or equal – signed	
`jl label`	jump **if** less than – signed	

or

`jlt label`	jump **if** less than – signed

Which Unconditional Jump to Use?

All you care is whether two values are **equal or not** `cmp.w src, dst`

`jeq`

You want to check for **or**[illegible]**e., >= or <** `cmp.w src, dst`

- with **signed values**

`jge` `jl`

- with **unsigned values**

`jhs` `jlo`

You care whether one value (e.g. result of operation or `tst.w dst`) is zero, nonzero, negative

`jz` `jnz` `jn`

You are working with the carry bit (e.g., `bit.w`) `jc` `jnc`

Instructions and Status Bits

	Instruction	Operands	Operation	V	N	Z	C
*	ADC(.B)	dst	dst + C → dst	x	x	x	x
	ADD(.B)	s[illegible]	[illegible] → dst	x	x	x	x
	ADDC(.B)	s[illegible]	[illegible] + C → dst	x	x	x	x
	AND(.B)	s[illegible]	[illegible] dst → dst	0	x	x	x
	BIC(.B)	s[illegible]	[illegible] and. dst → dst	-	-	-	-
	BIS(.B)	s[illegible]	[illegible] st → dst	-	-	-	-
	BIT(.B)	s[illegible]	[illegible] dst	0	x	x	x
*	BR	d[illegible]	[illegible]	-	-	-	-
	CALL	dst	PC+2 → stack, dst → PC	-	-	-	-
*	CLR(.B)	dst	Clear destination	-	-	-	-
*	CLRC		Clear carry bit	-	-	-	0
*	CLRN		Clear negative bit	-	0	-	-
*	CLRZ		Clear zero bit	-	-	0	-
	CMP(.B)	src,dst	dst - src	x	x	x	x
*	DADC(.B)	dst	dst + C → dst (decimal)	x	x	x	x
	DADD(.B)	src,dst	src + dst + C → dst (decimal)	x	x	x	x
*	DEC(.B)	dst	dst - 1 → dst	x	x	x	x
*	DECD(.B)	dst	dst - 2 → dst	x	x	x	x
*	DINT		Disable interrupt	-	-	-	-
*	EINT		Enable interrupt	-	-	-	-
*	INC(.B)	dst	Increment destination, dst +1 → dst	x	x	x	x
*	INCD(.B)	dst	Double-Increment destination, dst+2→dst	x	x	x	x
*	INV(.B)	dst	Invert destination	x	x	x	x
	JC/JHS	Label	Jump to Label if Carry-bit is set	-	-	-	-
	JEQ/JZ	Label	Jump to Label if Zero-bit is set	-	-	-	-
	JGE	Label	Jump to Label if (N .XOR. V) = 0	-	-	-	-
	JL	Label	Jump to Label if (N .XOR. V) = 1	-	-	-	-
	JMP	Label	Jump to Label unconditionally	-	-	-	-
	JN	Label	Jump to Label if Negative-bit is set	-	-	-	-

Legend:
- 0 Status bit always cleared
- 1 Status bit always set
- x Status bit cleared or set on results
- \- Status bit not affected
- * Emulated Instructions

Instructions and Status Bits

	Mnemonic	Operands	Description	V	N	Z	C
	JNC/JLO	Label	Jump to Label if Carry-bit is reset	-	-	-	-
	JNE/JNZ	Label	Jump to Label if Zero-bit is reset	-	-	-	-
	MOV(.B)	src,dst	src → dst	-	-	-	-
*	NOP		No operation	-	-	-	-
*	POP(.B)	dst	Item from stack, SP+2 → SP	-	-	-	-
	PUSH(.B)	src	SP-2 → SP, src → @SP	-	-	-	-
	RETI		Return from interrupt TOS → SR, SP + 2 → SP TOS → PC, SP + 2 → SZP	x	x	x	x
*	RET		Return from subroutine TOS → PC, SP + 2 → SP	-	-	-	-
*	RLA(.B)	dst	Rotate left arithmetically	x	x	x	x
*	RLC(.B)	dst	Rotate left through carry	x	x	x	x
	RRA(.B)	dst	MSB → MSB ....LSB → C	0	x	x	x
	RRC(.B)	dst	C → MSB ....LSB → C	x	x	x	x
*	SBC(.B)	dst	Subtract carry from destination	x	x	x	x
*	SETC		Set carry bit	-	-	-	1
*	SETN		Set negative bit	-	1	-	-
*	SETZ		Set zero bit	-	-	1	-
	SUB(.B)	src,dst	dst + .not.src + 1 → dst	x	x	x	x
	SUBC(.B)	src,dst	dst + .not.src + C → dst	x	x	x	x
	SWPB	dst	swap bytes	-	-	-	-
	SXT	dst	Bit7 → Bit8 .... Bit15	0	x	x	x
*	TST(.B)	dst	Test destination	x	x	x	x
	XOR(.B)	src,dst	src .xor. dst → dst	x	x	x	x

Legend:
- 0 The Status Bit is cleared
- 1 The Status Bit is set
- x The Status Bit is affected
- \- The Status Bit is not affected
- * Emulated Instructions

A Simple Loop

Add all numbers from 1 to 100 Answer 100*50.5 = 5050

16-bit unsigned integer

```
clr.w   R5        ; Initialize accumulator R5 = 0
mov.w   #1, R4    ; R4 = 1 1st value

add.w   R4, R5    ; R5 += R4 and R4 = 1
inc.w   R4        ; R4++

add.w   R4, R5    ; R5 += R4 and R4 = 2
inc.w   R4
...

add.w   R4, R5    ; R5 += R4 and R4 = 100
inc.w   R4
```

Repeat 100 times

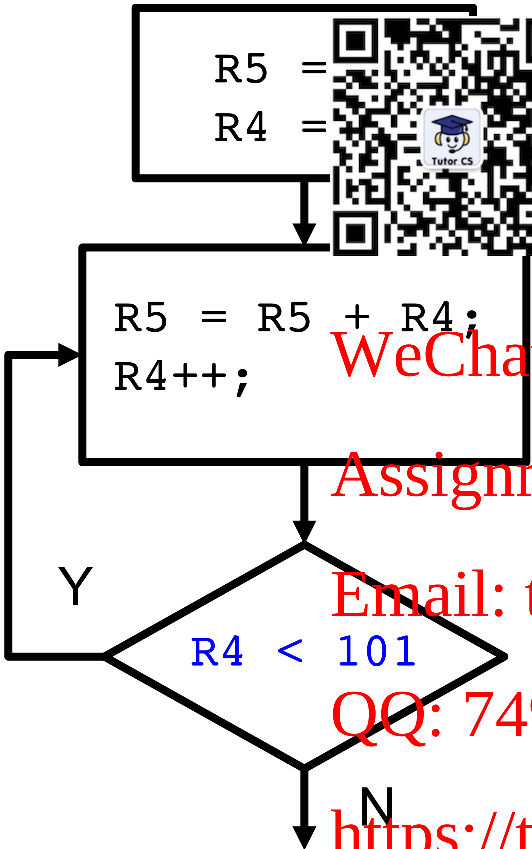
As long as R4 <= 100

A Simple Loop – Flowchart

Init:

Repeat:

Done:




```
Init:     clr.w   R5
          mov.w   #1, R4

Repeat:   add.w   R4, R5
          inc.w   R4

          cmp.w   #101, R4
          jlo     Repeat

Done: when R4 == 101 we are
done
```