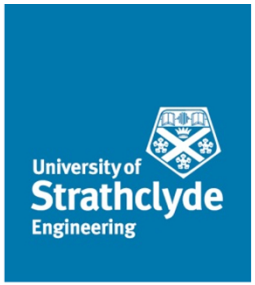


Introduction to Assembly Language

James Irvine

DEPARTMENT OF ELECTRONIC & ELECTRICAL ENGINEERING

Assembler

- Individual machine instructions written in ‘English’
- Instruction (mnemonic) followed by operands
- Varies with type of microcontroller (tens to hundreds of instructions)

Instruction Format

- Assembler instructions generally conform to:

```
<mnemonic> <size>   <operand>,<operand>
```

- Size may not be relevant in some processors
  - In the MSP430 may be B(yte), W(ord) (default) (or A(ddress))
- Operands are often source and destination
  - BUT not necessarily in that order (like in the PIC ☹)
  - The MSP430 follows the sensible ordering

Instruction Types

- Data Processing
  - Arithmetic & Logic instructions
- Data Storage
  - Memory instructions
- Data Movement
  - I/O instructions
- Control
  - Test & Branch instructions

- Going to look at addressing modes
- Where the actual values for the instruction come from
- They may not be required at all, give the value, give the value where the operand is (in effect, that's a variable), or it might give the location where the location of the value can be found

Addressing

- Inherent
  - The address is implied by the instruction
- Immediate
  - The data is given as an operand
  - Used for constants
- Direct
  - The address of the data is given as an operand (often with an offset)
  - Used for variables
- Indirect
  - The address where the address can be found is an operand
  - Used for indexed variables, like arrays

Inherent Addressing

- Also called ‘implicit’ addressing
- The address is ‘inherent’ – a characteristic of the instruction
- Example

```
CLRC                    //Clear Carry Flag
```

  - Note – no parameter to the instruction
  - Actually, on the MSP430 CLRC is an emulated instruction
    - CLRC → BIC #1,SR which uses immediate addressing!
    - As a RISC processor, the MSP430 doesn’t have inherent addressing

Immediate Addressing

- Also called ‘literal’ addressing
- The instruction operates using the value is given in the instruction
  - On the MSP430, this means at the following address @PC+
- Convention is to show this by #<value>
- Example

```
MOV.B #0, &P1OUT          //Set P1OUT to 0
```

Direct Addressing

- Also called ‘absolute’ addressing
- The instruction operates using the address given as a parameter to the instruction
- Convention is to show this by <value>
  - BUT the MSP430 uses ‘&’ to show absolute addressing
- Example

```
MOV.B R4, &P1OUT          //Set P1OUT to the value
                          //contained in R4
```

Indirect Addressing

- The address where the address of the value can be found is given as a parameter to the instruction
- Convention is to show this by (<value>)
  - BUT the MSP430 uses '@' to show indirect addressing
  - MSP430 only allows indirect addressing using registers
- Example

```
MOV.B @R4, &P1OUT          //Set P1OUT to the value
                           //contained in the location
                           //contained in R4
```

Indexed Addressing

- A variation of indirect addressing
- The address where the value can be found is by the parameter to the instruction plus an index register
- Convention is to show this by <value>(<index_reg>)
  - The MSP430 uses this convention ☺
- Example

```
MOV.B 2(R4), &P1OUT        //Set P1OUT to the value
                           //contained in the location
                           //contained in (R4 + 2)
```

Indexed Addressing

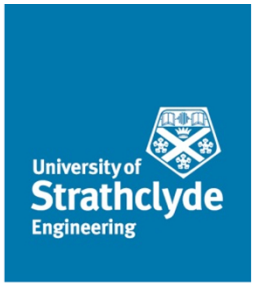
- Very useful for arrays

```
numbers:
        ds16 5                          //Reserve 5 words (10 bytes)

        mov.w #0,&total;                //Initialize total
        mov.w #10,R9;                   //Use R9 as array index
sum_loop:
        decd.w R9                       //Update array pointer
        jn more_code                    //If finished, continue
        add.w numbers(R9),&total
        jmp sum_loop                    //Loop for next value
more_code:
```

100	CLR 101
101	
102	MOV #2,103
103	
104	MOV 105,106
105	
106	
107	MOV (109), 108
108	
109	10B
10A	1
10B	2
10C	3

Relative Addressing



- Addresses are often given as relative addresses
- Address given as difference from current location rather than absolute address
- Allows code to be relocated by the linker

- Most processors have relative branch addresses
  - Linking is hard without it!
- Some processes have relative addresses for data
  - MSP430 has this – terms it ‘symbolic’ address
  - Direct addressing defaults to relative to PC, use ‘&’ for absolute

MSP430 Address Modes

- Immediate
  - 8 or 16 depending on instruction – specify with '#'
- Register Direct
- Indexed
  - x(Rn) - operand is in memory at address R*n*+*x*
- Symbolic
  - Special case of direct, address is added to PC so relative
  - This is the default
- Absolute
  - The address is given in the instruction – specify with '&'
- Register Indirect
  - @Rn
  - Special case with autoincrement by 1 or 2 (@Rn+)

MSP430 Addressing Modes

University of Strathclyde Engineering

Mode	Syntax	operand
Immediate	#N	N is a constant stored following the instruction
Register	Rn	Register contents.
Symbolic	ADDR	At address PC+ADDR, where ADDR is stored following the instruction
Absolute	&ADDR	At address ADDR stored following the instruction
Indexed	X(Rn)	Value at address X+Rx where X is stored following the instruction (and x forms part of the instruction)
Indirect Register	@Rn	At address pointed to by register Rn
Indirect Autoincrement	@Rn+	Rn contains address of operand; afterwards, the contents of Rn are incremented

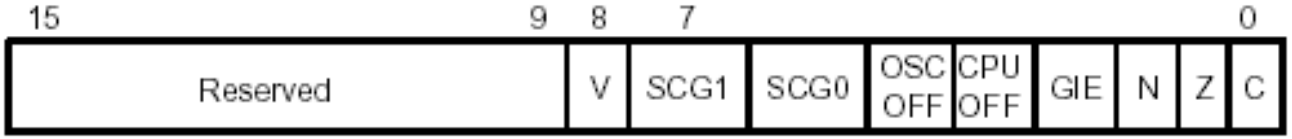
- As a RISC processor, the MSP430 has only a few addressing modes but uses them very consistently
- For other processors, different instructions may have different possible addressing modes
- Often better to try things out, and then change the code if the compiler complains a mode isn't available for that instruction

Registers

- 16 16 bit registers
- 12 general purpose registers (R4 – R15)
- Other four registers have special functions

- R0 is the program counter
- R1 is the stack pointer
- R2 is the status register
- R3 has common constants

Status Register



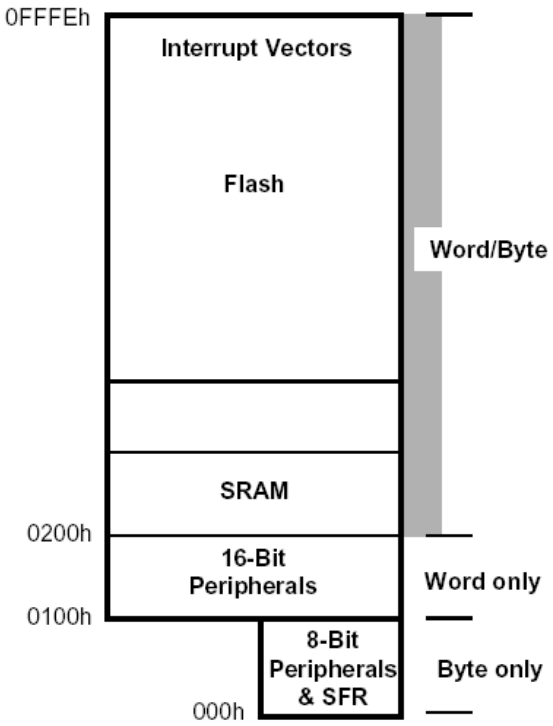
- C (Carry flag) SR(0)
- Z (Zero flag) SR(1)
- N (Negative flag) SR(2)
- GIE (Global interrupt enable) SR(3)
- CPUOff (Switches off CPU) SR(4)
- OSCOff (Switches off oscillator) SR(5)
- SCG1, SCG0 (Clock on/off) SR(7), SR(6)
- V (Overflow bit) SR(8)

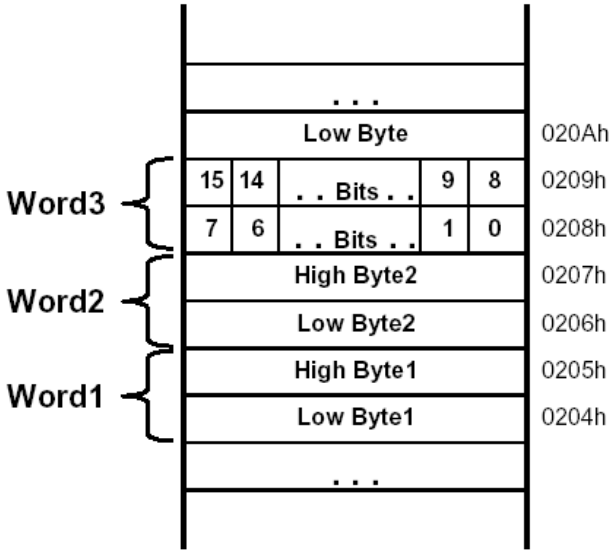
Status Bits

Bit	Description	
V	Overflow bit. This bit is set when the result of an arithmetic operation overflows the signed-variable range.	
	`ADD(.B),ADDC(.B)`	Set when: Positive + Positive = Negative Negative + Negative = Positive, otherwise reset
	`SUB(.B),SUBC(.B),CMP(.B)`	Set when: Positive – Negative = Negative Negative – Positive = Positive, otherwise reset
SCG1	System clock generator 1. This bit, when set, turns off the SMCLK.	
SCG0	System clock generator 0. This bit, when set, turns off the DCO dc generator, if DCOCLK is not used for MCLK or SMCLK.	
OSCOFF	Oscillator Off. This bit, when set, turns off the LFXT1 crystal oscillator, when LFXT1CLK is not use for MCLK or SMCLK	
CPUOFF	CPU off. This bit, when set, turns off the CPU.	
GIE	General interrupt enable. This bit, when set, enables maskable interrupts. When reset, all maskable interrupts are disabled.	
N	Negative bit. This bit is set when the result of a byte or word operation is negative and cleared when the result is not negative.	
	Word operation:	N is set to the value of bit 15 of the result
	Byte operation:	N is set to the value of bit 7 of the result
Z	Zero bit. This bit is set when the result of a byte or word operation is 0 and cleared when the result is not 0.	
C	Carry bit. This bit is set when the result of a byte or word operation produced a carry and cleared when no carry occurred.	

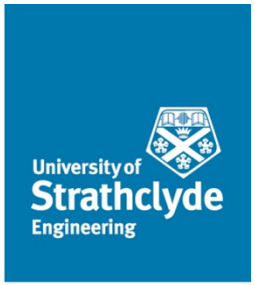
Memory

- Little endian (LSB of words stored first)





Assembler Statements



[name] [mnemonic] [operands] [comment]

- [name] is a label to which the program can jump (optional)
- [mnemonic] is the instruction itself
- [operands] (if required by the instruction)
- [comment] (optional, but very necessary to explain what you are trying to do - NOT what the instruction does, as that will be obvious)

MSP430 Assembler

University of Strathclyde Engineering

- Development system moves code blocks into RAM for execution
- Therefore, you must use RSEGs (Relocatable Segments) for your code (rather than .ORGs)
- Many different types, but you only need:

Contents	CCS Naming	IAR Naming
Program Code	.text	.RSEG CODE
Variables (uninitialized) in RAM	.bss	.RSEG DATA16_N
Constants (in ROM/flash)	.data	.RSEG CONST or DATA16_C
Stack	.stack	.RSEG CSTACK
Interrupt vectors	.int00 - .int14	.COMMON INTVEC
Reset vector	.reset	.RSEG RESET

IAR ‘Empty’ asm file

```
      #include "msp430.h"                ; #define controlled include file

         NAME    main                    ; module name

         PUBLIC  main                    ; make the main label vissible
                                         ; outside this module
         ORG     0FFFEh
         DC16    init                    ; set reset vector to 'init' label

         RSEG    CSTACK                  ; pre-declaration of segment
         RSEG    CODE                    ; place program in 'CODE' segment

init:    MOV     #SFE(CSTACK), SP        ; set up stack

main:    NOP                             ; main program
         MOV.W   #WDTPW+WDTHOLD,&WDTCTL  ; Stop watchdog timer
         JMP $                           ; jump to current location '$'
                                         ; (endless loop)
         END
```

CCS ‘Empty’ asm file

```
;-------------------------------------------------------------------------------
; MSP430 Assembler Code Template for use with TI Code Composer Studio
;
;
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

;-------------------------------------------------------------------------------
            .def    RESET                   ; 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

;-------------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  RESET
```

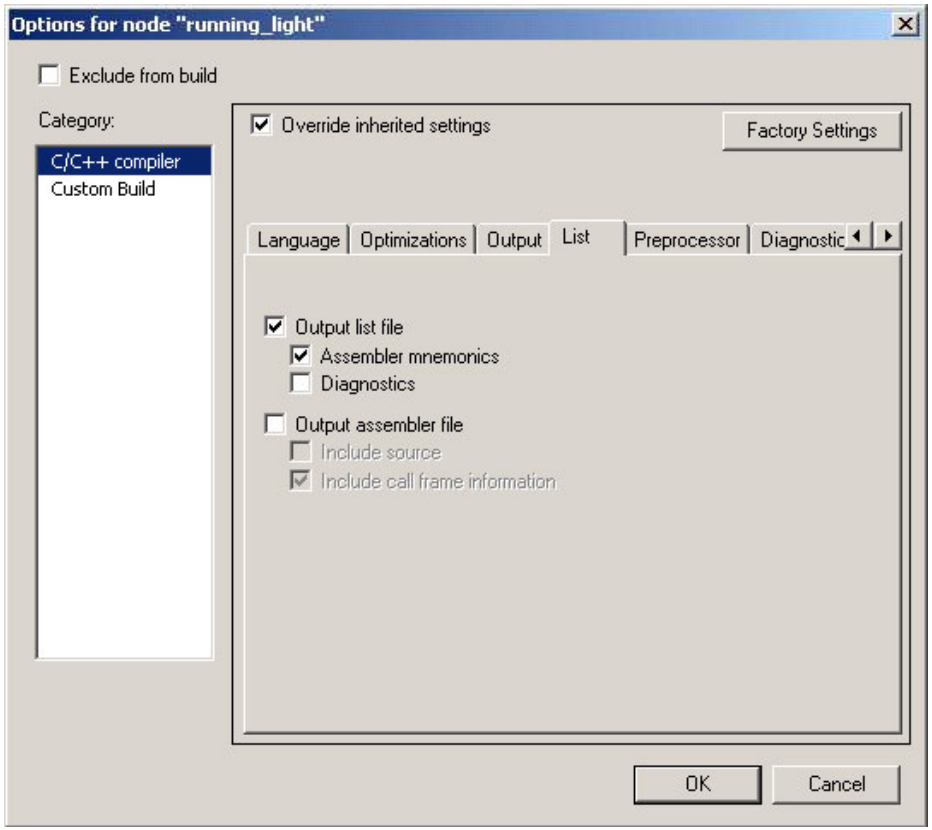
Simple C program

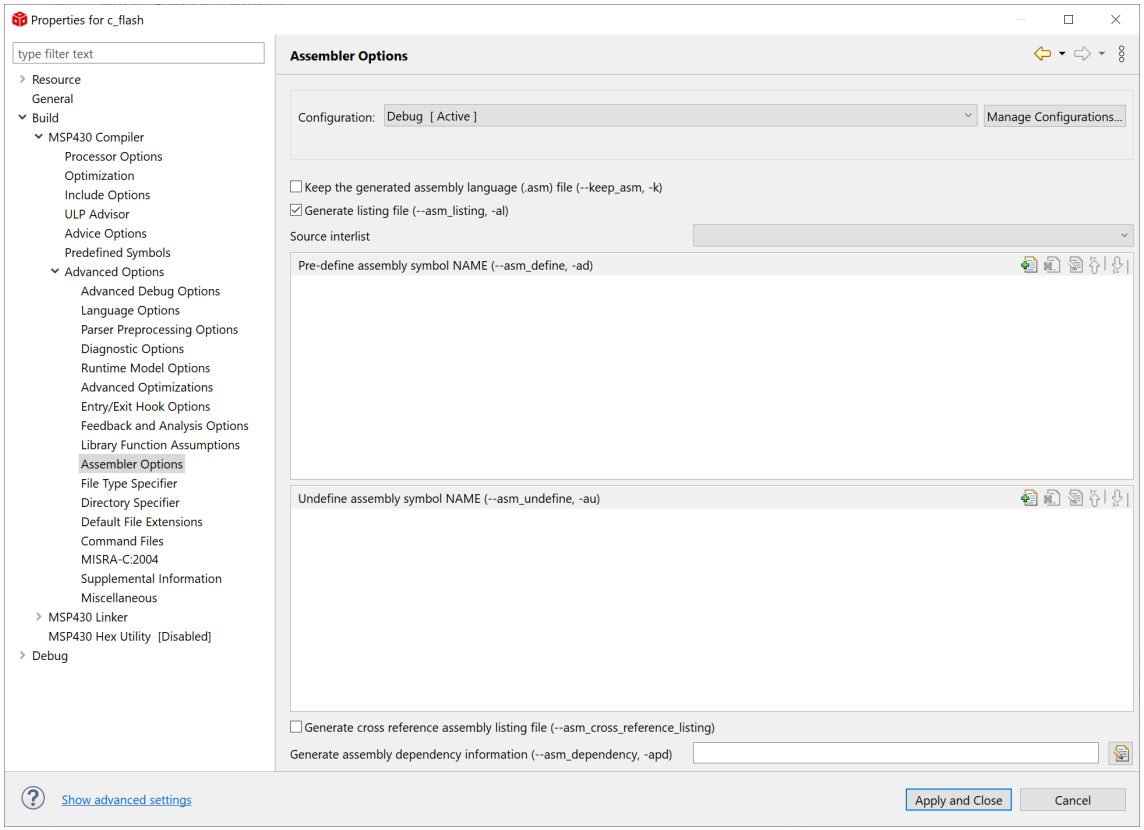
```
#include <io430.h>

int main( void )
{
  int counter;
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;

  P1DIR = 0x01;
  while (1)
  {
    if (counter++ < 0) P1OUT = 0x00;
    else P1OUT = 0x01;
  }
}
```

Getting an Assembler Listing





IAR GOTCHA

- The educational version of the IAR software includes a full version of the IDE and assembler, but a cut down version (‘Kickstart’) of the compiler
- The Kickstart version of the compiler won’t produce an assembler file, but will produce a listing file with assembler mnemonics, so
- Don’t tick ‘Output assembler file’
  - 10 second scripting test – can you produce an assembler file from the listing file?

The equivalent IAR assembler (I)

```
#include "msp430.h"                    ; #define controlled include file

        NAME    main                   ; module name

        PUBLIC  main                   ; make the main label vissible
                                       ; outside this module
        ORG     0FFFEh
        DC16    init                   ; set reset vector to 'init' label

        RSEG    DATA16_N
counter:
        DS16    1                      ; Counter value for loop (16 bit)

        RSEG    CSTACK                 ; pre-declaration of segment
        RSEG    CODE                   ; place program in 'CODE' segment
```

The equivalent IAR assembler (II)

```
init:   MOV     #SFE(CSTACK), SP         ; set up stack

main:   NOP                              ; main program
        MOV.W   #WDTPW+WDTHOLD,&WDTCTL   ; Stop watchdog timer
        MOV.B   #0x1, P1DIR              ; Set DDR for P1.0 output
        MOV.W   #0, counter              ; Initialise counter
coreloop:
        ADD.W   #1, counter
        CMP.W   #0, counter
        JN counter_neg                   ; Is counter negative?
        MOV.B   #0, P1OUT                ; If not, turn on the LED
        JMP coreloop                     ; and repeat
counter_neg:
        MOV.B   #1, P1OUT                ; Otherwise turn the LED off
        JMP coreloop                     ; and repeat
        NOP                              ; NOP at end due to pipelining

        END
```

The equivalent IAR assembler (II)

```
init:   MOV     #SFE(CSTACK), SP          ; set up stack

main:   NOP                               ; main program
        MOV.W   #WDTPW+WDTHOLD,&WDTCTL    ; Stop watchdog timer
        MOV.B   #0x1, P1DIR               ; Set DDR for P1.0 output
        MOV.W   #0, counter               ; Initialise counter
coreloop:
        ADD.W   #1, counter
        CMP.W   #0, counter
        JN counter_neg                    ; Is counter negative?
        MOV.B   #0, P1OUT                 ; If not, turn on the LED
        JMP coreloop                      ; and repeat
counter_neg:
        MOV.B   #1, P1OUT                 ; Otherwise turn the LED off
        JMP coreloop                      ; and repeat
        NOP                               ; NOP at end due to pipelining

        END
```

The equivalent IAR assembler (II)

```
init:   MOV     #SFE(CSTACK), SP          ; set up stack

main:   NOP                               ; main program
        MOV.W   #WDTPW+WDTHOLD,&WDTCTL    ; Stop watchdog timer
        MOV.B   #0x1, P1DIR               ; Set DDR for P1.0 output
        MOV.W   #0, counter               ; Initialise counter
coreloop:
        INC.W   counter
        JN counter_neg                    ; Is incremented counter negative?
        MOV.B   #0, P1OUT                 ; If not, turn on the LED
        JMP coreloop                      ; and repeat
counter_neg:
        MOV.B   #1, P1OUT                 ; Otherwise turn the LED off
        JMP coreloop                      ; and repeat
        NOP                               ; NOP at end due to pipelining

        END
```

The equivalent CCS assembler (I)

```
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

;-------------------------------------------------------------------------------
            .def    RESET                   ; 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.

            .bss counter,2,2                ; Counter value for loop (16 bit)

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

The equivalent CCS assembler (II)

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

            MOV.B   #0x1, P1DIR             ; Set DDR for P1.0 output
            MOV.W   #0, counter             ; Initialise counter
coreloop:
            ADD.W   #1, counter
            CMP.W   #0, counter
            JN counter_neg                  ; Is counter negative?
            MOV.B   #0, P1OUT               ; If not, turn on the LED
            JMP coreloop                    ; and repeat
counter_neg:
            MOV.B   #1, P1OUT               ; Otherwise turn the LED off
            JMP coreloop                    ; and repeat
            NOP                             ; NOP at end due to pipelining

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

;-------------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  RESET
```

The equivalent CCS assembler (II)

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

            MOV.B   #0x1, P1DIR             ; Set DDR for P1.0 output
            MOV.W   #0, counter             ; Initialise counter
coreloop:
            ADD.W   #1, counter
            CMP.W   #0, counter
            JN counter_neg                  ; Is counter negative?
            MOV.B   #0, P1OUT               ; If not, turn on the LED
            JMP coreloop                    ; and repeat
counter_neg:
            MOV.B   #1, P1OUT               ; Otherwise turn the LED off
            JMP coreloop                    ; and repeat
            NOP                             ; NOP at end due to pipelining

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

;-------------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  RESET
```

The equivalent CCS assembler (II)

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

            MOV.B   #0x1, P1DIR             ; Set DDR for P1.0 output
            MOV.W   #0, counter             ; Initialise counter
coreloop:
            INC.W   counter
            JN counter_neg                  ; Is counter negative?
            MOV.B   #0, P1OUT               ; If not, turn on the LED
            JMP coreloop                    ; and repeat
counter_neg:
            MOV.B   #1, P1OUT               ; Otherwise turn the LED off
            JMP coreloop                    ; and repeat
            NOP                             ; NOP at end due to pipelining

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

;-------------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
```

MSP430 Programming

- Uses the same development system as C
- Alternatives like gcc available

For assembler:

- RISC architecture – only 27 instructions
- Emulation gives 51 assembler instructions
  - E.g., dec Rx for sub #1, Rx
- Byte (.B) or Word (.W) options
  - **Defaults to .W**
- Details in user guide

Mnemonic		Description		V	N	Z	C
ADC(.B)†	dst	Add C to destination	dst + C → dst	*	*	*	*
ADD(.B)	src,dst	Add source to destination	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	Add source and C to destination	src + dst + C → dst	*	*	*	*
AND(.B)	src,dst	AND source and destination	src .and. dst → dst	0	*	*	*
BIC(.B)	src,dst	Clear bits in destination	.not.src .and. dst → dst	–	–	–	–
BIS(.B)	src,dst	Set bits in destination	src .or. dst → dst	–	–	–	–
BIT(.B)	src,dst	Test bits in destination	src .and. dst	0	*	*	*
BR†	dst	Branch to destination	dst → PC	–	–	–	–
CALL	dst	Call destination	PC+2 → stack, dst → PC	–	–	–	–
CLR(.B)†	dst	Clear destination	0 → dst	–	–	–	–
CLRC†		Clear C	0 → C	–	–	–	0
CLRN†		Clear N	0 → N	–	0	–	–
CLRZ†		Clear Z	0 → Z	–	–	0	–
CMP(.B)	src,dst	Compare source and destination	dst – src	*	*	*	*
DADC(.B)†	dst	Add C decimally to destination	dst + C → dst (decimally)	*	*	*	*
DADD(.B)	src,dst	Add source and C decimally to dst.	src + dst + C → dst (decimally)	*	*	*	*
DEC(.B)†	dst	Decrement destination	dst – 1 → dst	*	*	*	*
DECD(.B)†	dst	Double-decrement destination	dst – 2 → dst	*	*	*	*
DINT†		Disable interrupts	0 → GIE	–	–	–	–
EINT†		Enable interrupts	1 → GIE	–	–	–	–
INC(.B)†	dst	Increment destination	dst +1 → dst	*	*	*	*
INCD(.B)†	dst	Double-increment destination	dst+2 → dst	*	*	*	*
INV(.B)†	dst	Invert destination	.not.dst → dst	*	*	*	*
JC/JHS	label	Jump if C set/Jump if higher or same		–	–	–	–
JEQ/JZ	label	Jump if equal/Jump if Z set		–	–	–	–
JGE	label	Jump if greater or equal		–	–	–	–

JL	label	Jump if less		–	–	–	–
JMP	label	Jump	PC + 2 x offset → PC	–	–	–	–
JN	label	Jump if N set		–	–	–	–
JNC/JLO	label	Jump if C not set/Jump if lower		–	–	–	–
JNE/JNZ	label	Jump if not equal/Jump if Z not set		–	–	–	–
MOV(.B)	src,dst	Move source to destination	src → dst	–	–	–	–
NOP†		No operation		–	–	–	–
POP(.B)†	dst	Pop item from stack to destination	@SP → dst, SP+2 → SP	–	–	–	–
PUSH(.B)	src	Push source onto stack	SP – 2 → SP, src → @SP	–	–	–	–
RET†		Return from subroutine	@SP → PC, SP + 2 → SP	–	–	–	–
RETI		Return from interrupt		*	*	*	*
RLA(.B)†	dst	Rotate left arithmetically		*	*	*	*
RLC(.B)†	dst	Rotate left through C		*	*	*	*
RRA(.B)	dst	Rotate right arithmetically		0	*	*	*
RRC(.B)	dst	Rotate right through C		*	*	*	*
SBC(.B)†	dst	Subtract not(C) from destination	dst + 0FFFFh + C → dst	*	*	*	*
SETC†		Set C	1 → C	–	–	–	1
SETN†		Set N	1 → N	–	1	–	–
SETZ†		Set Z	1 → C	–	–	1	–
SUB(.B)	src,dst	Subtract source from destination	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	Subtract source and not(C) from dst.	dst + .not.src + C → dst	*	*	*	*
SWPB	dst	Swap bytes		–	–	–	–
SXT	dst	Extend sign		0	*	*	*
TST(.B)†	dst	Test destination	dst + 0FFFFh + 1	0	*	*	1
XOR(.B)	src,dst	Exclusive OR source and destination	src .xor. dst → dst	*	*	*	*

† Emulated Instruction

Writing code

- Even with a RISC chip …
- Don’t try to learn all the instructions!
- 20% of the instructions form 80% of the code
- Most common are:
  - MOV - To move variables between registers and memory, or with immediate data to initialise variables or set up peripherals
  - CMP - to compare values
  - J<condition> - to branch on the outcome of a condition
  - BIS, BIC - to set or clear a bit
  - and sundry arithmetic and logical instructions

Writing code

- Start with detailed psuedocode (try to use only the instructions on the previous slide)
- If you have no idea how to perform a function, write it in C and look at the assembler the compiler produces
- Try your code through the assembler, and worry about addressing modes, etc, when it complains

University of
Strathclyde
Glasgow

DEPARTMENT OF ELECTRONIC & ELECTRICAL ENGINEERING