More Assembly Language

James Irvine

DEPARTMENT OF ELECTRONIC & ELECTRICAL ENGINEERING

University of Strathclyde Engineering

Register Definitions

- Depending on the assembler, registers may be built in or you may have to define them
- Including ‘msp430.h’ and defining processing in project options includes the relevant processor definitions
  - In CCS, the pre-processor isn’t available
  - Use `.cdecls C,LIST, "msp430.h"`

BUT

- Everything defined for the C compiler
- Some missing (like PM5CTL0 for the 4133) for assembler
- Include #defines yourself from data sheet in a header (IAR)
- Use `SYMBOL .set VALUE` in CCS (or `.equ` – equate)

Number Definitions

- Different assemblers have different conventions
- WARNING – some older assemblers use hex by default
- CCS and IAR use the following:
  - Binary 1010b (CCS/IAR), 0b1010 (CCS), b'1010' (IAR)
  - Octal 1234q (CCS/IAR), 01234 (CCS), q'1234' (IAR)
  - Decimal 1234, -1, (CCS/IAR), d'1234‘ (IAR)
  - Hexadecimal 0FFFFh, 0xFFFF, (CCS/IAR), h'FFFF' (IAR)
- While not available for C, binary is very helpful for initalising registers

Constants in CCS

- CCS has a number of directives defining C types
  - `.byte`
  - `.char`
  - `.string`
  - `.int`
  - `.long`
  - `.float`
  - `.double`
  - `.long`
- Use with a label to initialise variables
  - E.g., `Offset: .double -2.0e25`

Conditional Assembly

- CCS uses assembler directives
  - `.if` *condition* marks the beginning of a conditional block and assembles code if the .if condition is true.
  - [`.elseif` *condition*] marks a block of code to be assembled if the .if condition is false & the .elseif condition is true.
  - `.else` marks a block of code to be assembled if the .if condition is false & any .elseif conditions are false.
  - `.endif` marks the end of a conditional block and terminates the block
- IAR allows a C pre-processor syntax for conditional assembly

```
#if (DEBUGGING > 2)
...
#else
...
#endif

#ifdef DEBUG
...
#endif
```

IAR C-style Preprocessor

#define	Assigns a value to a label
#elif	Introduces a new condition in a #if…#endif block
#else	Assembles instructions if a condition is false
#endif	Ends a #if, #ifdef, or #ifndef block
#error	Generates an error
#if	Assembles instructions if a condition is true
#ifdef	Assembles instructions if a symbol is defined
#ifndef	Assembles instructions if a symbol is undefined
#include	Includes a file
#message	Generates a message on standard output
#undef	Undefines a label

IAR Modules

- IAR Assembler allows for modules
- Program modules
```
NAME <module_name>
ENDMOD
```
- Library modules
```
MODULE <module_name>
ENDMOD
```
- Last module in a file ends with **`END`**
- Program modules are always linked
- Library modules are only linked if a public symbol is referenced by other code

MSP430 Programming

- Uses the same IAR 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

Software Manual

3.4.6.7 BIT

BIT[.W] Test bits in destination

BIT.B Test bits in destination

Syntax `BIT src,dst or BIT.W src,dst`

Operation src .AND. dst

Description The source and destination operands are logically ANDed. The result affects only the status bits. The source and destination operands are not affected.

Status Bits
N: Set if MSB of result is set, reset otherwise
Z: Set if result is zero, reset otherwise
C: Set if result is not zero, reset otherwise (.NOT. Zero)
V: Reset

Mode Bits OSCOFF, CPUOFF, and GIE are not affected.

Example If bit 9 of R8 is set, a branch is taken to label TOM.

```
BIT    #0200h,R8    ; bit 9 of R8 set?
JNZ    TOM          ; Yes, branch to TOM
...                 ; No, proceed
```

Example If bit 3 of R8 is set, a branch is taken to label TOM.

```
BIT.B   #8,R8
JC      TOM
```

Example A serial communication receive bit (RCV) is tested. Because the carry bit is equal to the state of the tested bit while using the BIT instruction to test a single bit, the carry bit is used by the subsequent instruction; the read information is shifted into register RECBUF.

```
;
; Serial communication with LSB is shifted first:
                        ; xxxx   xxxx   xxxx   xxxx
BIT.B   #RCV,RCCTL      ; Bit info into carry
RRC     RECBUF          ; Carry -> MSB of RECBUF
                        ; cxxx    xxxx
......                  ; repeat previous two instructions
......                  ; 8 times
                        ; cccc   cccc
                        ; ^         ^
                        ; MSB       LSB
; Serial communication with MSB shifted first:
BIT.B   #RCV,RCCTL      ; Bit info into carry
RLC.B   RECBUF          ; Carry -> LSB of RECBUF
                        ; xxxx   xxxc
......                  ; repeat previous two instructions
......                  ; 8 times
                        ; cccc   cccc
                        ; |
                        ; MSB       LSB
```

Software Manual

- Detailed description of each instruction
  - Syntax (including size of operands)
  - ‘Operation’ (what it does as an equation)
  - ‘Description’ (what it does in words)
  - Status bits - What condition flags are changed
  - Mode bits – What CPU mode flags are changed
  - Examples
- In separate table of all instructions
  - Length of instruction
  - Number of cycles to complete
  - Available address modes

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

Working with Assembler

- Start with detailed psuedocode
- Possible instructions are on the slides of the last lecture
- If you have no idea how to perform a function, write it in C and look at the assembler the compiler produces for ideas
- Try your code through the assembler, and worry about addressing modes, etc, when it complains

Things to consider…

- Programming assembler is time-consuming and error-prone, so use it only when required

- When something has to be fast
- When something has to be small
- When something has to last a precise time
  - You have complete control of the processor, and know the number of cycles taken for each operation
- When you've no other choice
  - But even PICs have C compilers…

Variables

- In assembler, everything is just a number
- Variables are labelled memory locations
- CCS
  - In a code (`.text`) segment
  - `.bss symbol, size in bytes[, alignment]`
- IAR
  - Define a data segment
    - `.RSEG DATA16_N`
  - Label a line with the name of the variable
  - Reserve the correct number of bytes
    - DS8 (Define space 8 bits) for bytes (i.e. c chars)
    - DS16 for words (c ints)
    - DS32 for longs (32 bits, c long)

Variables

- CCS

```
.text
.bss time,2,2                  ;Block Started by Symbol
.bss digit,1                   ;Better Save Space ☺

main: <code starts here...>
```

- IAR

```
.RSEG DATA16_N
time  DS16 1                   ;Define Storage
digit DS8  1

            .RSEG CODE
main: <code starts here...>
```

Simple Ways to Get Started

- Write the program in C
- Add an assembler module to the project
- Write a routine in assembler
- Call it from C
  - Remember to declare it as extern in the C module
  - Declare it as public in assember

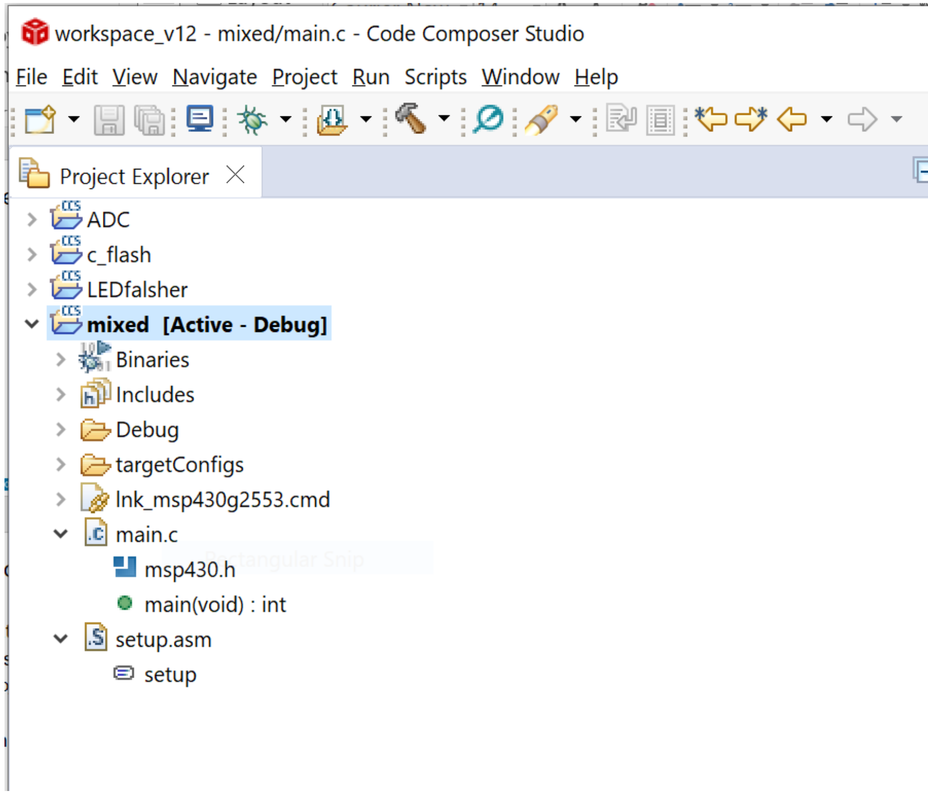
Simple C program

```c
#include <msp430.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;
  }
}
```

Add some assembler

- Create a new assembler file setup.asm in the project directory
- Add setup.asm to the project
  - Project → Add files
- Start with an empty file, make sure it compiles and runs
- Then start moving over code



Simple C program

```
#include "io430.h"

extern void setup();

int main( void )
{
  int counter;
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  setup();
  P1DIR = 0x01;
  while (1)
  {
    if (counter++ < 0) P1OUT = 0x00;
    else P1OUT = 0x01;
  }
}
```

Initial asm routine file

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

;-------------------------------------------------------------------------------
            .def    setup                   ; 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.

setup:      NOP                             ; do nothing
            RET                             ; then return


            .end
```

Start transferring code

University of Strathclyde Engineering

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

;-------------------------------------------------------------------------------
            .def    setup                   ; 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.

setup:      MOV.B   #0x1, P1DIR             ; Set DDR for P1.0 output
            RET                             ; then return


            .end
```

Updated C program

```
#include "io430.h"

extern void setup();

int main( void )
{
  int counter;
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  setup();
  while (1)
  {
    if (counter++ < 0) P1OUT = 0x00;
    else P1OUT = 0x01;
  }
}
```

More transferring

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

;-------------------------------------------------------------------------------
            .def    setup               ; Export program entry-point to
                                        ; make it known to linker.
            .def    flash               ; and make flash visible outside

;-------------------------------------------------------------------------------
            .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.

            .ref counter                ; bring in the counter variable from c

setup:      MOV.B   #0x1, P1DIR         ; Set DDR for P1.0 output
            RET                         ; then return


flash:      NOP                         ; Do nothing
            RET                         ; then return

            .end
```

Updated C program

```
#include <msp430.h>

extern void setup();
extern void flash();

int counter;          // counter now a global variable so the assembler can see it

int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  setup();
  while (1)
  {
    flash();
    if (counter++ < 0) P1OUT = 0x00;
    else P1OUT = 0x01;
  }
}
```

Move the functionality

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

;-------------------------------------------------------------------------------
            .def    setup                         ; Export program entry-point to
                                                  ; make it known to linker.
            .def    flash                         ; and make flash visible outside

;-------------------------------------------------------------------------------
            .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.

            .ref counter                          ; bring in the counter variable from c

setup:      MOV.B   #0x1, P1DIR                   ; Set DDR for P1.0 output
            RET                                   ; then return


flash:      INC.W counter
            JN led_on                             ; is incremented counter negative?
            MOV.B   #0x1,P1OUT                    ; if not, turn LED off
            RET                                   ; then return
led_on:     MOV.B   #0x0,P1OUT                    ; if so, turn it on
            RET                                   ; then return

            .end
```

Updated C program

```
#include "io430.h"

extern void setup();
extern void flash();
int counter;

int main( void )
{
// Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  setup();
  while (1)
  {
    flash();
  }
}
```

Fast code

- Use registers for frequently accessed variables
  - But remember helpful comments to avoid confusion
- Loop down to zero
  - Decrement and comparison in one, saving an instruction
- Use custom instructions when available
  - For example, SBJNZ - Subtract & jump if not zero
  - If you have a complex instruction set processor, use them!
  - However, MSP430 is RISC
- Multiply/divide by factors of two when possible
  - Shifts are faster than additions

Interrupts

- To use interrupts with assembler, you must:
  - Define an interrupt routine
  - Define the interrupt vector to point to the routine
  - Set up the interrupt vector table
  - Enable interrupts

Define an interrupt routine

- Routine defined as any normal subroutine
  – Put it in a code segment (.text or .RSEG CODE)
- Give the entry point a label (so you can set up the vector)
- Finish with an RETI (return from interrupt)
- If you alter any registers, push them on the stack at the start and pop them off the stack before returning

Set up the interrupt vector

- Put the address of the interrupt routine in the vector table
  - Get the address of the vector from the programming manual
  - CCS defines a section for each interrupt vector

```
.sect    ".int05"                 ; ADC10 Vector
.short   ADC10_ISR                ;
```

  - IAR
    - Reference the common segment called INTVEC

```
.COMMON INTVEC
```

    - Move to the vector address using `.ORG <address>`
    - `DC16 <interrupt_routine_label>`

Enable interrupts

- Programme your interrupt generating device (set up the interrupt control registers, etc)
- Enable interrupts
  - NOP ; NOP before enabling interrupts - an MSP430 requirement
  - BIS.W #GIE,SR
- MSP requires a NOP first (due to pipelining)
  - Ensures processor not half way through an instruction
- If you are not doing anything else, switch to a low power mode to stop the CPU
  - The interrupt will restart it

Interrupt timing

- The MSP430 isn't that fast…
- Interrupts take 6 clock cycles to start running ISR code
  – Plus the time to complete the current instruction, at most 6 more
- RETI takes 5 cycles
- Plus the time to execute the ISR code

- Default DCO speed is 1MHz
  – Means anything more than 50kHz through interrupts is hard
  – DCO can be increased to 16MHz if you need more speed

```
.cdecls C,LIST,  "msp430.h"
;-------------------------------------------------------------------------------
            .def    RESET                           ; Export program entry-point to
                                                    ; make it known to linker.

;-------------------------------------------------------------------------------
            .text

            .bss ADC_Result,2                       ; variable to store ADC value

;-------------------------------------------------------------------------------
RESET       mov.w   #0280h,SP                       ; Initialize stackpointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL          ; Stop WDT
            BIS.B  #0x1, &P1DIR                     ; Set P1.0/LED to output direction
            BIC.B  #0x1, &P1OUT                     ; P1.0 LED off
            MOV.W  #INCH_1, &ADC10CTL1
            BIS.B  #02h,&ADC10AE0                   ; P1.1 ADC10 option select
            MOV.W  #ADC10SHT_2+ADC10ON+ADC10IE, &ADC10CTL0  ; ADCON, 16x, enable int.
            JMP    mainloop

delay:
            MOV.W  #5000, R15                       ; Delay for 5000 - set initial counter
delayloop:
            SUB.W  #1, R15                          ; Decrement loop counter
            JC     delayloop                        ; If not yet zero, loop

mainloop:
            BIS.W   #ENC+ADC10SC,&ADC10CTL0         ; Start sampling/conversion
            BIS.W   #CPUOFF+GIE,SR                  ; LPM0, ADC10_ISR will force exit
            CMP.W  #0x1ff, &ADC_Result              ; Will be here after RETI as CPU restarts
            JC     LEDon                            ; If > 0x1FF, light LED
            BIC.B  #0x1, &P1OUT                     ; Otherwise switch it off
            JMP    delay
LEDon:
            BIS.B  #0x1, &P1OUT
            JMP    delay
            NOP
```

```
;=============== Interrupt routine for ADC

Int_ADC:                                                        ; Interrupt routine for ADC
            MOV.W   &ADC10MEM, &ADC_Result                      ; Store ADC result
            BIC.W   #CPUOFF,0(SP)                               ; Exit LPM0 on reti
            RETI

;=============== Interrupt Vectors

            .sect   ".reset"                ; MSP430 RESET Vector
            .short  RESET                   ;
            .sect   ".int05"                ; ADC10 Vector
            .short  Int_ADC               ;
            .end
```

```
#include "msp430.h"
            NAME main
            PUBLIC main

            ORG     0FFFEh
            DC16    init                                ; set reset vector to 'init' label

            RSEG DATA16_N
ADC_Result:
            DS16 1                                      ; variable to store ADC value

;=============== Interrupt routine for ADC
            RSEG CODE
Int_ADC:                                                ; Interrupt routine for ADC
            MOV.W   &ADC10MEM, &ADC_Result              ; Store ADC result
            BIC.W   #CPUOFF,0(SP)                       ; Exit LPM0 on reti
            RETI

;=============== Interrupt vectors
            COMMON INTVEC
            ORG ADC10_VECTOR                            ; Interrupt vector for ADC
            DC16 Int_ADC                                ; Point to the interrupt routine

;=============== Declare the stack to the linker
            RSEG CSTACK
```

```
;=============== Main code block
            RSEG CODE
init:       MOV     #SFE(CSTACK), SP                        ; set up stack

main:
            MOV.W   #WDTPW+WDTHOLD,&WDTCTL                  ; Stop watchdog timer
                                                            ; Configure GPIO
            BIS.B   #0x1, &P1DIR                            ; Set P1.0/LED to output direction
            BIC.B   #0x1, &P1OUT                            ; P1.0 LED off
            MOV.W   #INCH_1, &ADC10CTL1
            BIS.B   #02h,&ADC10AE0                          ; P1.1 ADC10 option select
            MOV.W   #ADC10SHT_2+ADC10ON+ADC10IE, &ADC10CTL0 ; ADCON, 16x, enable int.
            JMP     mainloop

delay:
            MOV.W   #5000, R15                              ; Delay for 5000 - set initial counter
delayloop:
            SUB.W   #1, R15                                 ; Decrement loop counter
            JC      delayloop                               ; If not yet zero, loop

mainloop:
            BIS.W    #ENC+ADC10SC,&ADC10CTL0                ; Start sampling/conversion
            BIS.W    #CPUOFF+GIE,SR                         ; LPM0, ADC10_ISR will force exit
            CMP.W   #0x1ff, &ADC_Result                     ; Will be here after RETI as CPU restarts
            JC      LEDon                                   ; If > 0x1FF, light LED
            BIC.B   #0x1, &P1OUT                            ; Otherwise switch it off
            JMP     delay
LEDon:
            BIS.B   #0x1, &P1OUT
            JMP     delay
            NOP
            END
```

Passing Parameters

- C functions use the stack for passing parameters
- To write parameterised assembler functions
  - Write a stub in C, where all the parameters are used

```
int myfunction(int a, int b, int c) {
  return (a+b+c);
}
```

  - Compile it with the relevant compiler
    - CCS and IAR pass parameters ***in different ways***!
  - Use the stack code from the assembler
  - Use pointers to variables if you need more than one return value
- Or, just use global variables…
  - Microcontroller stacks are small, and globals make planning easier
  - Assembler is messy anyway!

University of Strathclyde Glasgow