Lecture 5
Advanced Assembly Language

EE579
Advanced Microcontroller Applications
Dr James Irvine, EEE

Working with Assembler

- Start with <u>detailed</u> 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

M16C Address Modes

- M16C implements all 4 standard addressing modes
  - Inherent for relevant instructions
  - Immediate for 8, 16 and 20 bit values
  - Direct for most registers and memory locations, except most instructions can only address first 64k of memory (address is limited to 16 bits)
  - Indirect using the address registers A0 and A1
  - Useful ‘indirect with fixed displacement’ mode
- Details are given in Section 2 of the M16C Programming Manual (on the web site)

M16C Address Modes

- Immediate
  - 8, 16 or 20 bit depending on instruction
- Register Direct
  - R0L, R0H, R0, R1L, R1H, R1, R2,R3, An
- Absolute
  - Only 16 bit (I.e. 0000 - FFFFh) for most instructions
- Register Indirect
  - [An]
- Address register relative
  - 8 or 16 bit displacement (written displacement[An])
- SB/FB relative
  - 8 or 16 bit displacement from SB, or 8 from FB

Available Modes

- Not all modes are available for all instructions
- Permitted modes are:

	ABS	ADC	ADCF	ADD*1	ADJNZ*1	AND	CMP	DADC	DADD	DEC	DIV	DIVU	DIVX	DSBB	DSUB	ENTER	EXTS	INC	INT	JMPI*1	JMPS	JSRI*1	JSRS	LDC*1	LDE*1	LDINTB	LDIPL	MOV*1	MOVA
R0L/R0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	3		X		X		X	X			X	X
R0H/R1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X			4		X		X		X	X			X	X
R1L/R2	X	X	X	X	X	X	X				X	X	X				2			X		X		X	X			X	X
R1H/R3	X	X	X	X	X	X	X				X	X	X							X		X		X	X			X	X
An	X	X	X	X	X	X	X			X	X	X	X					X		X		X		X	X			X	X
[An]	X	X	X	X	X	X	X				X	X	X				X			X		X		X	X			X	
dsp:8[An]	X	X	X	X	X	X	X				X	X	X				X			X		X		X	X			X	X
dsp:8[SB/FB]	X	X	X	X	X	X	X			X	X	X	X				X	X		X		X		X	X			X	X
dsp:16[An]	X	X	X	X	X	X	X				X	X	X				X							X	X			X	X
dsp:16[SB]	X	X	X	X	X	X	X				X	X	X				X			X		X		X	X			X	X
abs16	X	X	X	X	X	X	X			X	X	X	X				X	X		X		X		X	X			X	X
#IMM8		X		X		X	X	X	X		X	X	X	X	X	X					X		X					X	
#IMM16		X		X		X	X	X	X		X	X	X	X	X									X				X	
#IMM20																										X			
#IMM					X														X								X		

	MOV Dir	MUL	MULU	NEG	NOT	OR	POP	POPM*1	PUSH	PUSHA	PUSHM*	ROLC	RORC	ROT	SBB	SBJNZ*1	SHA*1	SHL*1	STC*1	STCTX*1	STE*1	STNZ	STZ	STZX	SUB	TST	XCHG	XOR
R0L/R0	X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X
R0H/R1	X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X
R1L/R2	X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X		X				X	X	X	X
R1H/R3	X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X		X				X	X	X	X
An		X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X		X				X	X	X	X
[An]	X	X	X	X	X	X	X		X			X	X	X	X	X	X	X	X		X				X	X	X	X
dsp:8[An]	X	X	X	X	X	X	X		X	X		X	X	X	X	X	X	X	X		X				X	X	X	X
dsp:8[SB/FB]	X	X	X	X	X	X	X		X	X		X	X	X	X	X	X	X	X		X				X	X	X	X
dsp:16[An]	X	X	X	X	X	X	X		X	X		X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X
dsp:16[SB]	X	X	X	X	X	X	X		X	X		X	X	X	X	X	X	X	X		X				X	X	X	X
abs16	X	X	X	X	X	X	X		X	X		X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X
#IMM8		X	X			X									X							X	X	X	X	X		X
#IMM16		X	X			X									X										X	X		X
#IMM20																												
#IMM														X		X	X	X										

Software Manual

ADC *Add with carry* **ADdition with Carry** **ADC**

[Syntax]
ADC.size src,dest
B , W

[Instruction Code/Number of Cycles]
Page= 140

[Operation]
dest ← src + dest + C

[Function]
- This instruction adds *dest*, *src*, and C flag together and stores the result in *dest*.
- If *dest* is an A0 or A1 when the size specifier (.size) you selected is (.B), *src* is zero-expanded to perform calculation in 16 bits. If *src* is an A0 or A1, operation is performed on the eight low-order bits of the A0 or A1.

[Selectable src/dest]

src				dest			
R0L/R0	R0H/R1	R1L/R2	R1H/R3	R0L/R0	R0H/R1	R1L/R2	R1H/R3
A0/A0*1	A1/A1*1	[A0]	[A1]	A0/A0*1	A1/A1*1	[A0]	[A1]
dsp:8[A0]	dsp:8[A1]	dsp:8[SB]	dsp:8[FB]	dsp:8[A0]	dsp:8[A1]	dsp:8[SB]	dsp:8[FB]
dsp:16[A0]	dsp:16[A1]	dsp:16[SB]	abs16	dsp:16[A0]	dsp:16[A1]	dsp:16[SB]	abs16
~~dsp:20[A0]~~	~~dsp:20[A1]~~	~~abs20~~	#IMM	~~dsp:20[A0]~~	~~dsp:20[A1]~~	~~abs20~~	
~~R2R0~~	~~R3R1~~	~~A1A0~~		~~R2R0~~	~~R3R1~~	~~A1A0~~	

*1 If you specify (.B) for the size specifier (.size), you cannot choose A0 or A1 for *src* and *dest* simultaneously.

[Flag Change]

Flag	U	I	O	B	S	Z	D	C
Change	—	—	O	—	O	O	—	O

Conditions
O : The flag is set when a signed operation resulted in exceeding +32767 (.W) or –32768 (.W) or +127 (.B) or –128 (.B); otherwise cleared.
S : The flag is set when the operation resulted in MSB = 1; otherwise cleared.
Z : The flag is set when the operation resulted in 0; otherwise cleared.
C : The flag is set when an unsigned operation resulted in exceeding +65535 (.W) or +255 (.B); otherwise cleared.

[Description Example]

```
ADC.B   #2,R0L
ADC.W   A0,R0
ADC.B   A0,R0L        ; A0's 8 low-order bits and R0L are added.
ADC.B   R0L,A0        ; R0L is zero-expanded and added with A0.
```


Software Manual

- Detailed description of each instruction
  - Syntax (including size of operands)
  - Number of cycles to complete
  - ‘Operation’ (what it does as an equation)
  - ‘Function’ (what it does in words)
  - Options for the source and destination (e.g. address modes)
  - What condition flags are changed
  - Examples

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…

Segments

BITVARS	Bit variables.
CODE	**The program code**
CSTACK	The stack used by C or Embedded C++ programs.
CSTART	The startup code.
DATA16_HEAP	Heap data used by malloc and free. Used by CLib and DLib
FAR_HEAP	Heap used by malloc and free in DLib
DATA20_HEAP	Heap used by malloc and free in DLib
x_AC	Non-initialized located const objects
x_AN	Non-initialized located non-const objects
x_C	**Constant data, including string literals**
x_I	Initialized data
x_ID	Data that is copied to x_I by cstartup.
x_N	**Uninitialized data**
x_Z	zero initialized data

Where x can be one of:
DATA13 (Range: 0-0x1FFF)
DATA16 (Range: 0-0xFFFF, except DATA16_ID)
DATA20 (Range: 0-0xFFFFF)
FAR (Range: 0-0xFFFFF)

DIFUNCT	Pointers to code, typically EC++ constructors
FLIST	Jump table for __tiny_func functions.
INTVEC	**Contains reset and interrupt vectors**.
INTVEC1	Contains the fixed reset and interrupt vectors.
ISTACK	The stack used by interrupts and exceptions.

Variables

- In assembler, everything is just a number
- Variables are labelled memory locations
- 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
  - DS16 for words
  - DS32 for longs (32 bits)

Variables

```
      .RSEG DATA16_N
time  DS16 1
digit DS8  1

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

- Constants can have the same approach
  - Use DC8, DC16 and DC32
  - Put them in a .RSEG CONST
- CONST segment normally in flash, but not for the monitor, so ‘constants’ not initialised

Sample Program

```
#include <iom16c62p.h>

      .NAME main
      .PUBLIC main

      .RSEG DATA16_N
msec  DS16 1                 //number of milliseconds

      .RSEG CODE
main:
      FSET I                 // enable interrupts for debugging
      MOV.B #0x0F, PD4       // data direction registers
      MOV.B #0x0A, P4        // turn on 2 LEDs
      MOV.B #0, TA0MR        // Free running reloaded timer mode
      MOV.W #6000, TA0       // 6000 clicks at 6Mhz = 1ms
      BCLR 0, UDF            // Set down count
      MOV.W #0, msec         // Initialise millisecond count
      BSET 0, TABSR          // Start the timer
mainloop:
      BTST 3, TA0IC          // loop until TA0 times out
      JZ mainloop
      BCLR 3, TA0IC
      ADD.W #1, msec         // A millisecond has passed, so count it
      CMP.W #1000, msec      // Have we reached a second yet?
      JNZ mainloop           // If not, loop
      XOR.B #0x0F, P4        // Second is up, so flip LEDs
      MOV.W #0, msec         // Reset counter
      JMP mainloop           // Start again
      .END main
```

BTSTC

```
#include <iom16c62p.h>

      .NAME main
      .PUBLIC main

      .RSEG DATA16_N
msec  DS16 1                //number of milliseconds

      .RSEG CODE
main:
      FSET I                // enable interrupts for debugging
      MOV.B #0x0F, PD4      // data direction registers
      MOV.B #0x0A, P4       // turn on 2 LEDs
      MOV.B #0, TA0MR       // Free running reloaded timer mode
      MOV.W #6000, TA0      // 6000 clicks at 6Mhz = 1ms
      BCLR 0, UDF           // Set down count
      MOV.W #0, msec        // Initialise millisecond count
      BSET 0, TABSR         // Start the timer
mainloop:
      BTSTC 3, TA0IC        // loop until TA0 times out
      JZ mainloop
      ADD.W #1, msec        // A millisecond has passed, so count it
      CMP.W #1000, msec     // Have we reached a second yet?
      JNZ mainloop          // If not, loop
      XOR.B #0x0F, P4       // Second is up, so flip LEDs
      MOV.W #0, msec        // Reset counter
      JMP mainloop          // Start again
      .END main
```


Count down

```
#include <iom16c62p.h>

      .NAME main
      .PUBLIC main

      .RSEG DATA16_N
msec  DS16 1                //number of milliseconds

      .RSEG CODE
main:
      FSET I                // enable interrupts for debugging
      MOV.B #0x0F, PD4      // data direction registers
      MOV.B #0x0A, P4       // turn on 2 LEDs
      MOV.B #0, TA0MR       // Free running reloaded timer mode
      MOV.W #6000, TA0      // 6000 clicks at 6Mhz = 1ms
      BCLR 0, UDF           // Set down count
      MOV.W #1000, msec     // 1000 ms = 1 second
      BSET 0, TABSR         // Start the timer
mainloop:
      BTSTC 3, TA0IC        // loop until TA0 times out
      JZ mainloop
      SUB.W #1, msec        // A millisecond has passed, so count it
      JNZ mainloop          // If not 1 second yet, loop
      XOR.B #0x0F, P4       // Second is up, so flip LEDs
      MOV.W #1000, msec     // Reset counter for 1 second
      JMP mainloop          // Start again
      .END main
```


SBJNZ

```
#include <iom16c62p.h>

      .NAME main
      .PUBLIC main

      .RSEG DATA16_N
msec  DS16 1                //number of milliseconds

      .RSEG CODE
main:
      FSET I                // enable interrupts for debugging
      MOV.B #0x0F, PD4      // data direction registers
      MOV.B #0x0A, P4       // turn on 2 LEDs
      MOV.B #0, TA0MR       // Free running reloaded timer mode
      MOV.W #6000, TA0      // 6000 clicks at 6Mhz = 1ms
      BCLR 0, UDF           // Set down count
      MOV.W #1000, msec     // 1000 ms = 1 second
      BSET 0, TABSR         // Start the timer
mainloop:
      BTSTC 3, TA0IC        // loop until TA0 times out
      JZ mainloop
      SBJNZ.W #1, msec, mainloop        // If not 1 second yet, loop
      XOR.B #0x0F, P4       // Second is up, so flip LEDs
      MOV.W #1000, msec     // Reset counter for 1 second
      JMP mainloop          // Start again
      .END main
```


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
- Better still – SBJNZ
  - Subtract & jump if not zero
  - If you have a complex instruction set processor, use them!
- Multiply/divide by factors of two when possible
  - Shifts are faster than additions

Look Up Tables

2 options:

- Have the code in RAM
  - Simpler addressing (RAM is <16bit limit, so standard addressing modes can be used)
  - Values have to be initiallised
- Have the code in ROM
  - Have to use LDE instructions (ROM is > 16 bit limit)
  - Can't use other instructions (ADD, etc) directly – have to move the data first
  - No initialisation required

RAM based code

```
  .RSEG DATA16_N
RAMseg_LUT:
  .BLKB 10                //reserve 10 bytes for LUT

  .RSEG CODE

  MOV #0xC9, RAMseg_LUT    //initialise the LUT
  MOV #0xA0, RAMseg_LUT+1

etc
```


ROM based code

```
 .RSEG CODE

seg_LUT:
  .BYTE 0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8, 0x80, 0x98
  ;      0     1     2     3     4     5     6     7     8     9
  ; Digit values

<other code>

  LDE.B seg_LUT, P0         // Do something with the LUT value
                            // Only LDE instruction can be used
```


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 (.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
  - Reference the common segment called INTVEC
    - .COMMON INTVEC
  - Move to the vector address using .ORG <address>
  - DC24 <interrupt_routine_label>
- Set up the vector table
  - Put LDINTB #sfb(INTVEC) early in your code (*before* enabling interrupts)

Enable interrupts

- Programme your interrupt generating device (set up the interrupt control registers, etc)
- Enable interrupts
  - FSET I
- BEWARE! (http://documentation.renesas.com/eng/products/mpumcu/tu/mc850204.pdf)
  - The M16C uses pipelining – more than one instruction is being processed at one time
  - FSET I is processed quickly, so if you do it immediately after an instruction to clear an interrupt flag (in an interrupt control register), the interrupt flag will be set – and an interrupt generated – *before* the ICR gets cleared
  - Work around – Add a dummy move instruction between an ICR move and FSET I


```
#include <iom16c62p.h>

  .NAME main
  .PUBLIC main

  .RSEG DATA16_N
msec BLKW 1                             ;number of milliseconds
flag BLKB 1                             ;flag to indicate interrupt

;=============== Interrupt routine for timer TA0
  .RSEG CODE
Int_TimerA0:                          ;Interrupt routine for TA0
    MOV.B #0x1, flag                  ;Set flag to indicate timeout
    REIT

;=============== Interrupt vectors
  .COMMON INTVEC
                                        ; Defaults to starting at 0
    ORG 84                              ; Move origin to 84, the interrupt vector for TA0
    DC24 Int_TimerA0                    ; Point to the interrupt routine

;=============== Main code block
  .RSEG CODE

main:
    LDINTB #sfb(INTVEC)                 ; Point to the interrupt vector table
    FSET I
    MOV.B #0x0F, PD4                    ; data direction registers
    MOV.B #0x0A, P4                     ; turn on 2 LEDs */
    MOV.B #0, TA0MR                     ; Free running reloaded timer mode
    MOV.W #6000, TA0                    ; 6000 clicks at 6Mhz = 1ms
    BCLR 0, UDF                         ; Set down count
    MOV.W #1000, msec                   ; 1000 ms = 1 second
    MOV.B #4,TA0IC                      ; Level 4 interrupt from TA0
    BSET 0, TABSR                       ; Start the timer

mainloop:
    CMP.B #0, flag                      ; Wait for interrupt to set flag
    JZ mainloop
    MOV.B #0, flag                      ; Reset timer timed-out flag
    SBJNZ.W #1, msec, mainloop          ; Decrement time count, and loop if not zero
    XOR.B #0x0F,P4                      ; Flip LEDs
    MOV.W #1000, msec                   ; Reset to 1 second
    JMP mainloop                        ; Start all over again
  .END
```