Lecture 3
Assembly Language

EE579
Advanced Microcontroller Applications
Dr James Irvine, EEE

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 M16C may be B(yte), W(ord), L(ong)
- Operands are often source and destination
  - BUT not necessarily in that order (like in the PIC :-(
  - The M16C follows the sensible ordering

Instruction Types

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

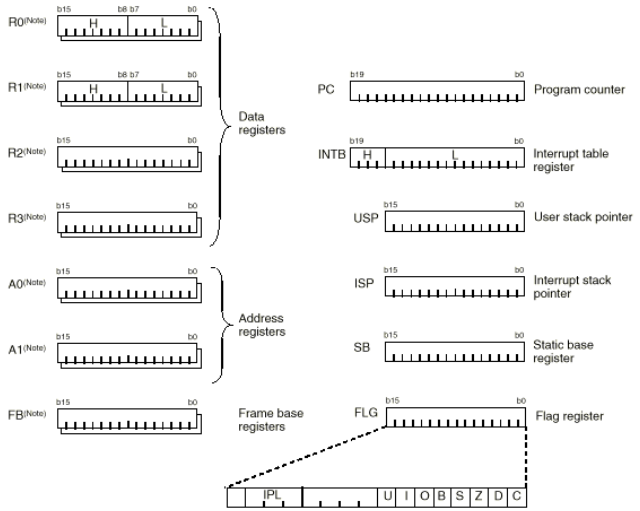
Addressing

- Inherent
  - The address is inherent in the instruction, like ‘Shift Right Accumulator’
- Immediate
  - The data is given as an operand
  - Often given by ‘#’ before the operand
- Direct
  - The address of the data is given as an operand
- Indirect
  - The address where the address can be found is an operand

Relative Addressing

- Branch 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
- Some processors has both, in which case:
  - Jumps are usually absolute
  - Branches are usually relative

M16C Registers



Note: These registers consist of two register banks.

M16C Data Registers

- Data registers (R0, R1, R2, and R3)
  - 16 bits registers,used primarily for transfer and arithmetic/logic operations.
  - R0 and R1 each can be used as separate 8-bit data registers, high-order bits as (R0H/R1H), and low-order bits as (R0L/R1L)
  - In some instructions, registers R2 and R0, as well as R3 and R1 can use as 32-bit data registers (R2R0/R3R1).
- Address registers (A0 and A1)
  - 16 bits registers; can be used as data registers or for address register indirect addressing and address register relative addressing
  - In some instructions, registers A1 and A0 can be combined for use as a 32-bit address register (A1A0)

M16C Registers

- Interrupt table register (INTB)
  - 20 bits, holds the start of an interrupt vector table.
- Stack pointer (USP/ISP)
  - Each configured with 16 bits. One in use selected by a stack pointer select flag
- Frame base register (FB)
  - 16 bits; used for FB relative addressing.
- Static base register (SB)
  - 16 bits; used for SB relative addressing.
- Program counter (PC)
  - 20 bits

Flag register (FLG)

- 11 bits, each one is used as a flag

Bit 0: Carry flag (C flag)

Set to '1' on carry from arithmetic operation

Bit 1: Debug flag (D flag)

When '1' enables a single-step interrupt after each instruction.

Bit 2: Zero flag (Z flag)

Set to '1' when the result of an arithmetic operation is zero

Bit 3: Sign flag (S flag)

'1' when result of arithmetic operation is negative

Bit 4: Register bank select flag (B flag)

Chooses a register bank : bank 0 when 0, bank 1 when 1.

Bit 5: Overflow flag (O flag)

Set to '1' when an arithmetic operation overflows

Flag register (FLG)

Bit 6: Interrupt enable flag (I flag)

Enables maskable interrupts when '1'

Bit 7: Stack pointer select flag (U flag)

Interrupt stack pointer (ISP) is selected when this flag is "0" ; user stack pointer (USP) is selected when this flag is "1".

Automatically cleared to "0" when a hardware interrupt is acknowledged or an INT instruction of software interrupt Nos. 0 to 31 is executed.

Bits 12 to 14: Processor interrupt priority level (IPL)

Holds the current processor interrupt priority level

Only interrupts with a higher level will be enabled

Bit 15: Reserved area

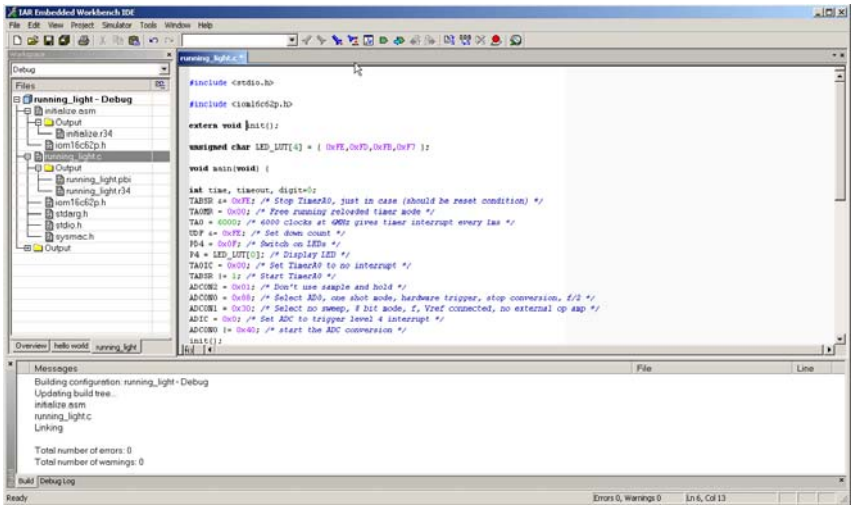
Assembler Statements

[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)

Simple Assembler Program

```
// Include the register definitions
#include <iom16c62p.h>


      .NAME main
      .PUBLIC main

      .RSEG CODE
main:
      FSET I              // enable interrupts for debugging
      MOV.B #0x0F, PD4    // data direction registers
      MOV.B #0x0A, P4     /* c comment style – turn on 2 LEDs */
      JMP main            ; assembler comment style – loop forever

      .END main
```

EE579 Advanced Microcontroller Applications

Init() Assember code

```
#include <iom16c62p.h>


      .NAME initialise
      .PUBLIC init
      .RSEG CODE
init  BCLR 0, TABSR     // Stop TimerA0, just in case (should be reset condit'n)
      MOV.B #0, TA0MR   // Free running reloaded timer mode
      MOV.W #6000, TA0  // 6000 clocks at 6MHz gives timer interrupt every 1ms
      BCLR 0, UDF       // Set down count
      MOV.B #0x0F, PD4  // Select LHS digit

      RTS

      .END
```

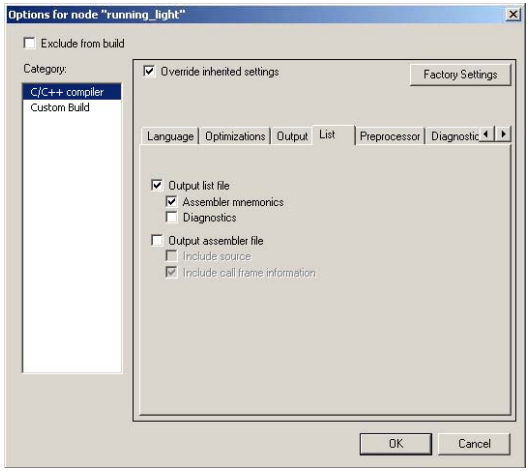
EE579 Advanced Microcontroller Applications

Solution (Part 1) with Assembler Module

```
#include <iom16c62p.h>

extern void init();

const unsigned char seg_LUT[10] = { /* P0:- 7 segment character Look Up Table */
  /* 0,   1,   2,   3,   4,   5,   6,   7,   8,   9 */
  0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x98 };

void main(void) {
  int time, timeout, digit;

  init();               // call assembler routines to initialise

  P4      = LED_LUT[0];  /* Display digit */
  TA0IC   = 0x00;        /* Set TimerA0 to no interrupt */
  TABSR  |= 1;           /* Start TimerA0 */
  ADCON2  = 0x01;        /* Don't use sample and hold */
  ADCON0  = 0x88;        /* Select AD0, one shot mode, hardware trigger, stop conversion, f/2 */
  ADCON1  = 0x20;        /* Select no sweep, 8 bit mode, f/2, Vref connected, no external op amp */
  ADIC    = 0x04;        /* Set ADC to trigger level 4 interrupt */
  ADCON0 |= 0x40;        /* start the ADC conversion */
```

EE579 Advanced Microcontroller Applications

Getting an Assembler Listing



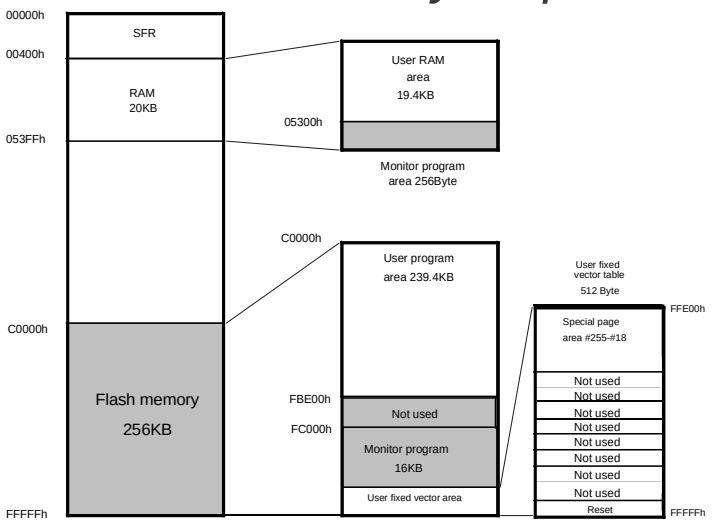
GOTCHA

- 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?

M16C Assembler

- 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:
  - .RSEG CODE (for code, located in ROM)
  - .RSEG DATA16_N (for variables, located in RAM)
  - .RSEG DATA16_C (for constants, located in ROM)

M16C Memory Map



WARNING!

- CODE and DATA16_C segments (along with other ROM segments) are located initially at C0000h
- Access requires far pointers (20 bit address)
- Normal addressing won't work - use LDE (LoaD from EXtra far data area) instruction to access memory

Register Definitions

- Depending on the assembler, registers may be built in or you may have to define them
- Registers built in for the IAR compiler
  - R0-R3, A0-1, FB, SB, SP, ISP, USP, FLG, INTB
- SFR not built in for the IAR compiler
  - Define using an include file for the chip
  - IAR uses the same include file for c and assembler – makes life simple but not a common approach
  - File has two blocks – one for c and one for assembler – with conditional 'compilation'
  - Register names are case insensitive for the IAR compiler

Number Definitions

- Different assemblers have different conventions
- WARNING – some assemblers (not IAR) use hex by default
- IAR understands the following
  - Binary 1010b, b'1010'
  - Octal 1234q, q'1234'
  - Decimal 1234, -1, d'1234'
  - Hexadecimal 0FFFFh, 0xFFFF, h'FFFF'

  The RADIX assembler directive changes the default number base

Conditional Assembly

- Most assemblers allow 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

M16C Instructions

- 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
  - BSET, BCLR, BTST - to set, clear or test 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

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
R0L/R0	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	
R1L/R2	X	X	X	X	X	X	X				X	X	X			
R1H/R3	X	X	X	X	X	X	X				X	X	X			
An	X	X	X	X	X	X	X			X	X	X	X			
[An]	X	X	X	X	X	X	X				X	X	X			
dsp:8[An]	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			
dsp:16[An]	X	X	X	X	X	X	X				X	X	X			
dsp:16[SB]	X	X	X	X	X	X	X				X	X	X			
abs16	X	X	X	X	X	X	X			X	X	X	X			
#IMM8		X		X		X	X	X	X		X	X	X	X	X	X
#IMM16		X		X		X	X	X	X		X	X	X	X	X	
#IMM20																
#IMM					X											

	EXTS	INC	INT	JMPI*1	JMPS	JSRI*1	JSRS	LDC*1	LDE*1	LDINTB	LDIPL	MOV*1	MOVA	MOV Dir	MUL	MULU	NEG	NOT	OR
R0L/R0	X	3		X		X		X	X			X	X	X	X	X	X	X	X
R0H/R1		4		X		X		X	X			X	X	X	X	X	X	X	X
R1L/R2	2			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
An		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
dsp:8[An]	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
dsp:16[An]	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
abs16	X	X		X		X		X	X			X	X	X	X	X	X	X	X
#IMM8					X		X					X			X	X			X
#IMM16								X				X			X	X			X
#IMM20										X									
#IMM			X								X								

	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
R0H/R1	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
R1H/R3	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
[An]	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
dsp:8[SB/FB]	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
dsp:16[SB]	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
#IMM16									X										X	X		X
#IMM20																						
#IMM								X		X	X	X										

Transfer

MOV	Transfer
MOVA	Transfer effective address
MOVDir	Transfer 4-bit data
POP	Restore register/memory
POPM	Restore multiple registers
PUSH	Save register/memory/immediate data
PUSHA	Save effective address
PUSHM	Save multiple registers
LDE	Transfer from extended data area
STE	Transfer to extended data area
STNZ	Conditional transfer
STZ	Conditional transfer
STZX	Conditional transfer
XCHG	Exchange

Bit Manipulation

BAND	Logically AND bits
BCLR	Clear bit
BM Cnd	Conditional bit transfer
BNAND	Logically AND inverted bits
BNOR	Logically OR inverted bits
BNOT	Invert bit
BNTST	Test inverted bit
BNXOR	Exclusive OR inverted bits
BOR	Logically OR bits
BSET	Set bit
BTST	Test bit
BTSTC	Test bit & clear
BTSTS	Test bit & set
BXOR	Exclusive OR bits

Shift

ROLC	Rotate left with carry
RORC	Rotate right with carry
ROT	Rotate
SHA	Shift arithmetic
SHL	Shift logical

Arithmetic

ABS	Absolute value
ADC	Add with carry
ADCF	Add carry flag
ADD	Add without carry
CMP	Compare
DADC	Decimal add with carry
DADD	Decimal add without carry
DEC	Decrement
DIV	Signed divide
DIVU	Unsigned divide
DIVX	Singed divide
DSBB	Decimal subtract with borrow
DSUB	Decimal subtract without borrow
EXTS	Extend sign

Arithmetic

INC	Increment
MUL	Signed multiply
MULU	Unsigned multiply
NEG	Two’s complement
RMPA	Calculate sum-of-products
SBB	Subtract with borrow
SUB	Subtract without borrow

Logical

AND	Logical AND
NOT	Invert all bits
OR	Logical OR
TST	Test
XOR	Exclusive OR