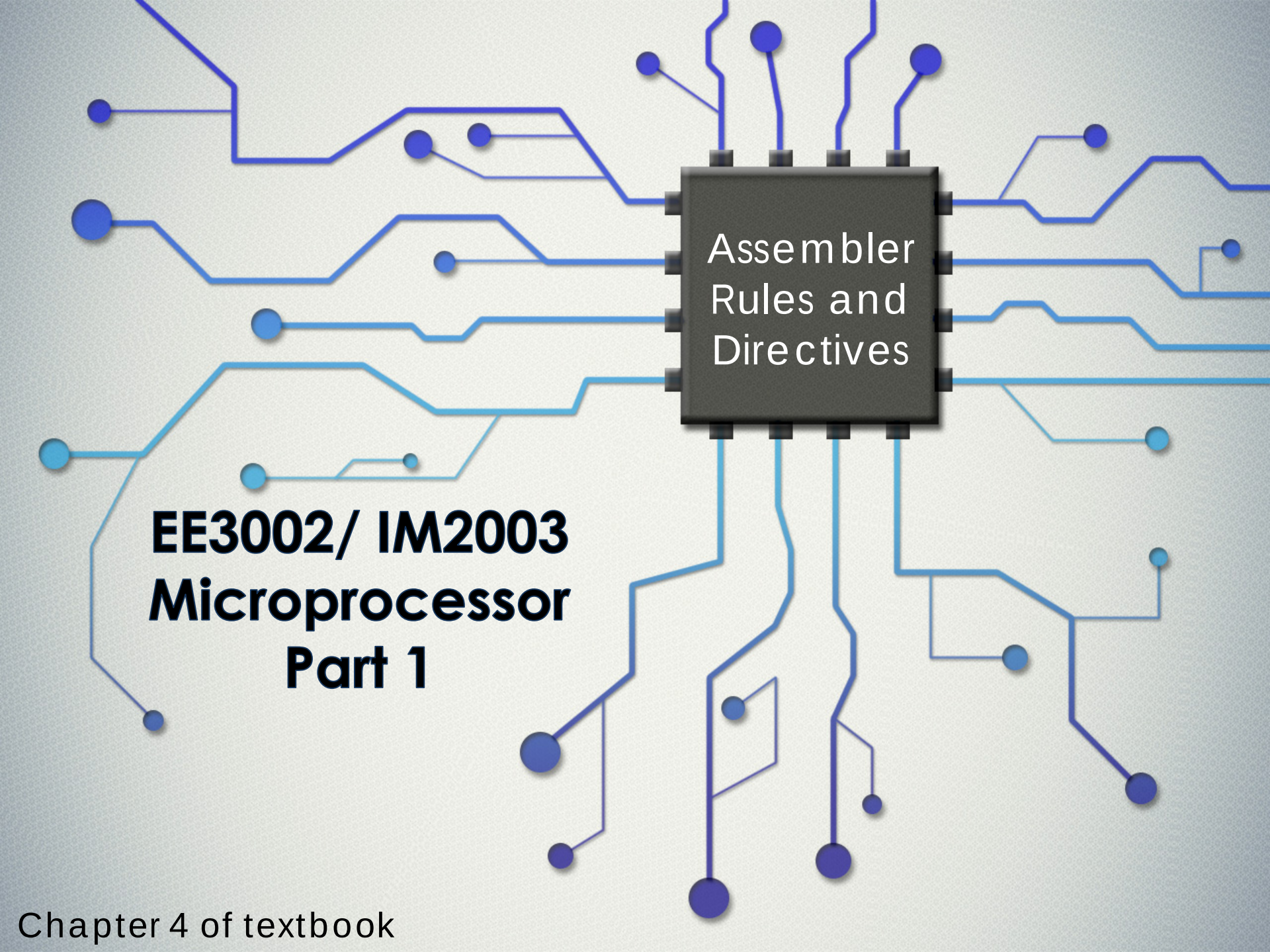

EE3002/ IM2003 Microprocessor Part 1

Chapter 4 of textbook

General Form

- General form of source lines in your assembly file is

  {label} {instruction | directive | pseudo-instruction} {;comment}

  where each field in braces is optional.
- When all fields are absent, it become a blank line. Blank lines are sometimes used to make a program more readable.
- The instructions, directives and pseudo-instructions must be preceded by a white space or tab, even in the absence of a label.

Labels

- Labels are names chosen to represent an address somewhere in memory and they will be eventually be translated into a numeric value by the linker.
- A label name can only defined once in your code, multiple labels of the same name is not allowed.
- Label name cannot start with a number .e.g. `1app` is not allowed.
- Labels must start on the left.

```
    AREA Prog2,  CODE,  READONLY
    ENTRY
    MOV  r6, #10     ; load 10 into r6
    MOV  r4, r6      ;  copy n into a temp register
loop  SUBS  r4, r4, #1 ; decrement next multiplier
    MULNE   r7, r6, r4   ;perform multiply
    MOV     r6, r7
    BNE     loop   ;go again if not complete
stop  B     stop   ;stop program
    END
```

Comment lines

- The first semicolon on a line indicates the beginning of a comment.
- A well documented assembly language is important as assembly language programs can be hard to understand. Normally a well written assembly language program will have around 50% comment lines.
- Don't state the obvious, for example in the previous slide.

```
MOV  r6, #10    ; load 10 into r6
```

```
        AREA Factorial,  CODE,  READONLY
    ; This program calculates the factorial of a number, n
    ; Factorial (n) = n(n-1)(n-2)... 1
        ENTRY
        MOV    r6, #10 ; r6 will contain the input n = 10
        MOV    r4, r6          ; copy n into r4 which also
    ; serves as a multiplier
loop    SUBS       r4, r4, #1  ; decrement next multiplier
        MULNE  r7, r6, r4      ;perform multiply
        MOV        r6, r7      ; save product in r6
        BNE            loop    ;go again if not complete
stop    B              stop    ;stop program, final result in r6
        END
```

Constant Values

- Constants can be Expressed as Numeric Values or Character Strings
  - Decimal: 1324
  - Hexadecimal: 0x3DE2 (32-bit value, zero-padded)
  - General: n_xxxx (n is base in [2,9], xxx is digit string)
  - Character: 'V' (enclosed in single quote)
  - String: "Hello world!\n"
- Control Characters Specified as in C Language
- Single Quote:' \' '
- Dollar Sign or Double Quote: Use Two in a row "$$" is a SINGLE Dollar Sign; " "" " is a single Double Quote

Predefined Register Names

- r0-r15 or R0-R15
- a1-a4 (argument, result, or scratch registers, synonyms for r0 to r3)
- sp or SP (stack pointer, r13)
- lr or LR (link register, r14)
- pc or PC (program counter, r15)
- cpsr or CPSR (current program status register)
- spsr or SPSR (saved program status register)

Some Common Directives

Directive	Comment
AREA	Defines Block of data or code
RN	Equates a Register with a name
EQU	Equates a Symbol to a Numeric Constant
ENTRY	Declares an Entry Point to a Program
DCB	Allocates one or more Bytes of memory. It also specifies initial runtime contents of memory.

Some Common Directives

Directive	Comment
DCW	Allocates one or more Halfwords (16 bits) of memory. It also specifies initial runtime contents of memory.
DCD	Allocates one or more Words (32bits) of memory. It also specifies initial runtime contents of memory.
ALIGN	Aligns data or code to a specific boundary
SPACE	Reserves a zeroed block of memory of a certain size.
LTORG	Assigns starting point of a literal pool.
END	Designates end of source file

AREA-Define a Block of Data or Code

- The AREA directive instructs the assembler to begin a new code or data section.
- There must be at least one AREA directive in an assembly module.
- Normal to use separate sections for code and data.
- Large programs can be divided into several code sections.
- Large independent data sets, such as tables, should also be placed in separate sections.
- Syntax is

AREA *sectionname* {, attr} {, attr}

- Where *sectionname* is name of the section and any name can be chosen as long as it does not start with a numeric digit.

Attributes for AREA

- CODE : The section contains machine instructions. READONLY is the default
- DATA : The section contains data. READWRITE is the default.
- READONLY: This indicates that this section should not be written to and there it can be placed in read-only memory.
- READWRITE: This section can be read from and written to and it must therefore be placed in read-write memory.

REGISTER NAME DEFINITION

- **RN** directive defines a register name for a specified register.
- Syntax is

  name RN *expr*

  where *name* is the name to be assigned to the register. The parameter *expr* is a number from 0 to 15.
- Examples

```
coeff1   RN 8   ; coefficient 1
dest  RN 0   ; register 0 holds the pointer to
             ;  destination matrix
```

EQUATE A SYMBOL TO A NUMERIC CONSTANT

- The EQU directive gives a symbolic name to a numeric constant, a register-relative value, or a program-relative value.
- Syntax : *name* EQU *expr* {,*type*}

  where parameter *type* is optional and can be any one of the following:

  CODE16, CODE32, DATA

Example

```
SRAM_BASE  EQU  0x04000000  ; assigns SRAM
           ; base address
abc   EQU   2   ; assigns the value 2 to the
    ;symbol abc
xyzEQU   label+8  ; assigns the address
   ;(label+8) to the symbol xyz
fig EQU   0x1C, CODE32  ; assigns the          ;absolute
   address 0x1C to the symbol      ;fig, and marks it as
   code
```

ENTRY

- The ENTRY directive declares an entry point to a program.
- You must specify at least one ENTRY point for a program.
- If no ENTRY exists, a warning is generated at link time.
- You must not use more than one entry directive in a single source file and not every source file has to have an ENTRY directive.

Allocate Memory and Specify Contents

- The DCB directive allocates one or more bytes of memory and defines the initial contents of the memory.
- The syntax is

  {*label*} DCB *expr* {,*expr*} …

  where expr is either a numeric expression that evaluates to an integer in the range of -128 to 255 or a quoted string
- Examples

```
max_mark DCB 100      ; define max_mark to be 100
marks DCB 10, 20, 25, 33 ; define 4 bytes with values of 10,
                         ; 20, 25 and 33
C_string DCB “C_string”, 0 ; ARM assembly strings are not null-
;terminated, hence a 0 is added behind to form a null-
terminated ;string like in C language.
```

DCW

- DCW directive allocates one or more halfwords of memory aligned on two byte boundaries and defines the initial contents.
- The syntax is

  {*label*} DCW{U} *expr* {,*expr*} …

  where expr is either a numeric expression that evaluates to an integer in the range of -32768 to 65535
- DCWU is same as DCW except that there is no memory alignment
- Examples

  coeff DCW 0xFE37, 0x8ECC ; defines 2 halfwords

DCD

- DCD directive allocates one or more words of memory aligned on 4-byte boundaries and defines the initial contents.
- The syntax is

  {*label*} DCD{U} *expr* {,*expr*} ...
- DCDU is same as DCD except that there is no memory alignment
- Examples

```
data1 DCD 1, 5, 20 ; Defines 3 words containing
                   ; decimal values 1, 5, and 20
data2 DCD mem06 + 4 ; Defines 1 word containing 4+
                    ; the address of label mem06
```

ALIGN

- The ALIGN directive aligns the current location to a specified boundary by padding with zeros.
- The syntax is

ALIGN {*expr* {,*offset*}}

where *expr* is a numeric expression evaluating to any power of two from 2^0 to 2^{31}.

- ALIGN itself without any *expr* sets the current location to the next word (4-byte) boundary. This is the most common usage of this directive.
- Proper alignment can speed up the efficiency of the program in some cases

SPACE

- The SPACE directive reserves a zeroed block of memory.
- The syntax is

  {*label*} SPACE *expr*

where *expr* evaluates to the number of zeroed bytes to reserve.

- Example

```
data1 SPACE 255 ; defines 255 bytes of zeroed ;
    storage
```

END

- The END directive informs the assembler that it reached the end of a source file.
- Every assembly language source file must terminate with END on a line by itself

Macros

- Macros allow a programmer to build definitions of functions or operations once and then call this operation by name throughout the code, saving some writing time.
- Macros are NOT the same as subroutine call, since the macros definitions are substituted during assembly time.
- Compare Macros with Subroutines at a later date when subroutines are covered.

Macro definition

MACRO

{*$label*} *macroname*{*$cond*}

{*$parameter*{,*$parameter*}...}

; code

MEND

where *$label* is a parameter that is substituted with a label,

$cond is a special parameter which contain a condition code.

$parameter is substituted when the macro is invoked.

Macro example (definition)

```
    MACRO
    ; macro defintion:
    ; vara = 8* (varb + varc + 6)
$Label_1    AddMul  $vara, $varb, $varc
$Label_1
    ADD $vara, $varb, $varc  ; add two terms
    ADD $vara, $vara, #6     ; add 6 to the sum
    MOV $vara, $vara, LSL #3    ; multiply by 8
    MEND
```

Invoking the Macro

- The macro can be invoked as many times as you wish in the source file. For e.g.

```
;invoke the macro
CSet1   AddMul      r0, r1, r2
```

- Assembler will make the necessary substitutions and your listing will become

```
;invoke the macro
CSet1
    ADD     r0, r1, r2
    ADD     r0, r0, #6
    MOV     r0, r0, LSL #3
```

Assembler Operators

- Primitive operations can be performed on data during assembly process
- A:MOD:B A modulo B
- A:ROL:B Rotate A left by B bits
- A:ROR:B Rotate A right by B bits
- A:SHL:B Shift A left by B bits
- A:SHR:B Shift A right by B bits
- A+B Add A to B
- A-B Subtract B from A
- A:AND:B Bitwise AND of A and B
- A:EOR:B Bitwise Exclusive OR of A and B
- A:OR:B Bitwise OR of A and B

Examples

- ORR r1, r1, #1:SHL:3 ;set bit 3 of r1

  ; 1 shifted left three times is binary 1000
- DCD (0x8321:SHL:4):OR:2

; is equivalent to DCD 0x83212

- MOV r0, #((1:SHL:14) :OR: (1:SHL:12))

; is equivalent to

MOV r0, #0x5000

Example Workings

DCD (0x8321:SHL:4):OR:2

Hexadecimal

Binary

8	3	2	1
1000	0011	0010	0001

Shift left by 4 bits

```
                                    0000
Or  00000000000000000010
-----------------------------------------
    10000011001000010010
      8    3    2    1    2
```

Example Workings

MOV r0, #((1:SHL:14) :OR: (1:SHL:12))

(1:SHL:14) => shifting 1 14 bits to the left

In binary notation

=> 100 0000 0000 0000

(1:SHL:12) => 001 0000 0000 0000 OR

101 0000 0000 0000

Hexadecimal 5 0 0 0

Readability

- Compare the 3 ways of writing.
- Using binary, which bits are set?

```
MOV r0,
#2_00000000000000001010000000000000
```

- Using hexadecimal

```
MOV r0, #0x5000
```

- Using shift and OR operations

```
MOV r0, #((1:SHL:14) :OR: (1:SHL:12))
```

Literals

- Literals are considered to be fixed values.
- Two types of literals, string literals and numeric literals.
- abc **DCB** “This is a string” ;string literal
- Examples of numeric literals
  - MOV r9, #0xCF
  - MOV r3, #’Q’
  - Addr DCD 2_1100101 ; binary constant

Summary

- Learn how to document your program properly
- Learn what are assembler directives
- Learn how to use Macros
- Learn how to use assembler operators