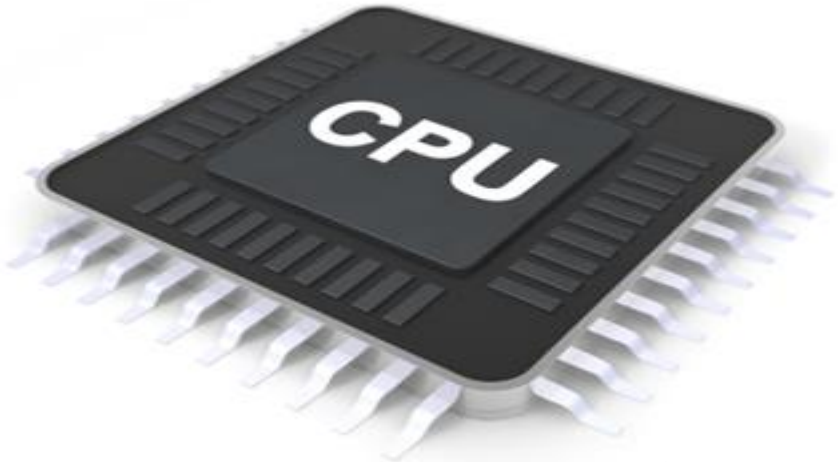

Data Definition and Transfer

ENEE 3582

Microp

.CSEG .DSEG .ESEG .ORG

- .CSEG: directive to define the start of the code segment
  - Mega2560 code is in the flash memory (256KB)
  - Memory organized is words (not bytes)
- .DSEG: directive to define the start of the data segment
  - Mega2560 data is in the SRAM (8KB internal)
- .ESEG: directive to define the start of the EEPROM segment
  - Mega2560 EEPROM (64KB internal)
- .ORG: directive used to define the starting address (origin) for data/code
  - For program memory the address is x2
  - Example:

```
.CSEG
.ORG 0x00100          //all code that will appear next starts at 0x00200
```

.EQU .SET .DEF

- .EQU
  - Equate a symbol to an expression.
  - The symbol becomes a name for that expression
  - Usage: .EQU symbol = expression
  - Can’t re-equate symbol to a different expression
- .SET
  - Set a symbol to an expression.
  - Symbols can be re-Set to a different expression
- .DEF
  - Give a register a symbolic name
  - Usage: .DEF symbol = R#
- .UNDEF
  - Undo a .DEF Give a register a symbolic name

Examples: EQU, SET, DEF

```
.EQU addr = 0x23      //addr is a name for the number 0x23
.EQU addr = addr+1    //illegal after 1st EQU
.SET foo  = 0x114     //foo is a name for the number 0x114
.SET foo  = foo + 1   //foo is now a symbol for 0x115
```

Program Memory vs RAM

- Program Memory
  - Where the program will be stored
    - `.CSEG`
  - Flash memory
    - ROM type that can be erased
  - Used to store constants (aka immediates) that the program needs
    - Can’t be used to store variables
- RAM
  - Where temporary values/variables can be stored
    - `.DSEG`
  - Can’t be initialized with values before program runs

Program Memory vs RAM: Usage

- Use PM to supply program with values
  - Use directives to define these values: `.DB`, `.DW` , `.DD` , `.DQ`
  - Examples:

```
        .CSEG
Bval    .DB   5
```

- Use RAM to store new values the programs generates
  - Use directives to reserve RAM space: `BYTE`
  - Examples:

```
        .DSEG
var1    BYTE  10     ;reserves 10 bytes for var1
```

Defining Constants in CSEG

- `.CSEG`: program memory
- `.DB` Define constant byte(s)
  - Single values or array of values
  - Range: 0 to 255 (U), -128 to 127 (S)
- `.DW`: Define constant word(s)
- `.DD`: Define constant double-word(s)
- `.DQ`: Define constant quad-word(s)
- Can use any number base
  - Example:

```
.CSEG
consts: .DB 0, 255, 0b01010101, -128, 0xaa
```

Little Endian Format

- Used to store data that is greater than a byte
  - word, doubleword, quadword
- Little Endian Format:
  - lowest significant BYTE is stored first,
  - highest significant BYTE stored last
- Example: `0x12345678` is stored as `0x78`, `0x56`, `0x34`, `0x12`

Example: Little Endian

```
            .CSEG
arrB:       .DB     0x12, 0x34, 0x56
arrW:       .DW     0x1234, 0x5678, 0x9ABC
arrD:       .DD     0x12345678, 0x9ABCDEF0
arrQ:       .DQ     0x123456789ABCDEF0
```

```
0x000000   12 34 56 00 34 12 78 56 bc 9a 78 56 34 12 f0 de
0x000010   bc 9a f0 de bc 9a 78 56 34 12 ff ff ff ff ff ff
```

00 is inserted to make addresses multiples of 2 (CSEG)

Function to Access Data in Words, Doubles, Quads

- Available from Microchip Studio
- LOW(): get the lowest significant byte
- HIGH(): get the highest significant byte
  - Highest byte for a word
  - Same as BYTE2()
- BYTE2(): get the 2nd lowest significant byte
- BYTE3(): get the 3rd lowest significant byte
- BYTE4(): get the 4th lowest significant byte
- LWRD(): get the lowest significant word
- HWRD(): get the highest significant word

Example of Functions

```
.EQU        val1  = 0x12345678
            LDI R16, HIGH(val1)
            LDI R16, LOW(val1)
            LDI R16, BYTE2(val1)
            LDI R16, BYTE3(val1)
            LDI R16, BYTE4(val1)

wval1:      .dW HWRD(val1)
wval2:      .dw LWRD(val1)
```

Reg-Reg Transport: MOV, MOVW, SWAP

- MOV
  - Copy byte from a source reg (Rs) into destination reg (Rd)
  - Syntax: `MOV Rd, Rs`
  - Example:
```
MOV R16, R0              ;Copy r0 to r16
```
- MOVW
  - Copy word from 2 adjacent source regs into 2 adjacent destination regs
  - Syntax: `MOVW Rd+1:Rd, Rs+1:Rs`
  - Example:
```
MOVW R16:R15, R1:R0      ;Copy r0,r1 to r15,r16
```
- SWAP
  - Swap upper nibble (4 bits) with lower nibble
  - Syntax: `SWAP Rd`
  - Example:
```
SWAP R16            ;Assume Rd=0b11110000.
                    ;SWAP:  Rd=0b00001111
```

Load/Store

- All memory access is done through load/store instructions
  - RISC style
  - Load: read from mem to regs
  - Store: write regs into mem
  - Load/store operands: index regs, regs
  - Index regs (X,Y, Z) used to access mem
  - Regs (R#) used to hold data

LDI

- Loads an **immediate**
- Constant is loaded into a destination reg
- Destination reg can be R16 to R31
- Constant range is byte (0 to 255, -128 to 127)
- Syntax:
```
LDI Rd, K          ;K is a constant val
```
- Example:
```
LDI R16, 12        ;R16=12
```

LDS

- **Direct** load
- A byte is loaded from memory into a register
- address memory using a constant K
  - Can also be a symbolic constant
- K is a 16-bit value (word)
- Destination reg can be `R0` to `R31`
- Used with **SRAM**
- Syntax: `LDS Rd, Address`
- Example: `LDS R16,0x1234  ;R16→Memory[0x1234]`

LD

- **Indirect** Load from memory
- Load using an index reg (Ri) into a destination reg
- Used with **SRAM**
- Rix can be X, Y, or Z
- Can **post-increment** after load, or **pre-decrement** Rix before a load
- Destination reg can be `R0` to `R31`
- Syntax:

```
LD Rd, Rix
LD Rd, Rix+      ;load then increment Rix
LD Rd, -Rix      ;decrement Rix then load
```

- Examples:

```
LD R16,X         ;R16←Memory[X]
LD R16, X+       ;R16←Memory[X] then X=X+1
LD R16, -X       ;X=X-1 then R16←Memory[X]
```

LDD

- **Indirect with displacement** Load
- Address = index reg (Rix) + constant displacement (D)
- D is 6 bits: 0 to 63
- Used with **SRAM** only
- Rix can be Y, or Z
- Syntax: `LDD Rd, Rix+D` `;D is a constant`
- Examples: `LDD R16, Z+5` `;R16→Memory[Z+5]`

LPM

- **Indirect** Load from **Program Memory** (aka flash)
- Using an Z as index register
- Can **post-increment** after load (Z+)
  - Can't pre-decrement Z
- Destination reg can be R0 to R31
- Syntax:

```
LPM   Rd, Z      ;Rd←Memory[Z]
LPM   Rd, Z+     ;Rd←Memory[Z] then increment  Z
```

STS

- **Direct** store
- Store a byte from a source reg (Rs) to memory using a direct address (K)
- Address (K) is provided as an immediate (constant)
  - Can be a symbolic constant
- K is 16 bit value (word)
- Source reg can be `R0` to `R31`
- Used with **SRAM**
- Syntax:
```
STS K, Rs          ;K=address constant vak
```
- Example:
```
STS 0x1234, R1     ;Memory[0x1234] = R1
```

ST

- **Indirect** store
- Use an index reg (Ri) to store a source reg
- Used with **SRAM**
- Rix can be X, Y, or Z
  - Can **post-increment** after load, or **pre-decrement** Rix before a load
- Syntax:

```
ST Rix , Rs
ST Rix+, Rs      ;store then increment Rix
ST –Rix, Rs      ;decrement then store
```

- Examples:

```
ST X ,R1         ;Memory[X]←R1
ST X+,R1         ;Memory[X]←R1 then X=X+1
ST -X,R1         ;X=X-1 then Memory[X]←R1
```

STD

- **Indirect with displacement** store
- Store using an index reg (Rix) + constant displacement (D)
- D is a 6 bit value (0 to 53)
- Used with **SRAM** only
- Rix can be Y, or Z
- Source reg can be `R0` to `R31`
- Syntax: `STD Rix+D,Rd ;D is a constant`
- Example: `STD Z+5, R1 ;Memory[X+5]←R1`

SPM

- **Indirect** store to **Program Memory** (aka flash)
- Erases pages from flash
- Using an Z as index register
- Can **post-increment** after load (Z+)
  - -Z not supported

XCH

- **Indirect** data mode
- Uses Z only as index register
- Operates on **SRAM** only
- Exchanges the value stored in register and the value pointed to by Z
- Reg can be `R0` to `R31`
- Syntax: `XCH Z,Rd`
- Example: `XCH Z,R16  ;Memory[Z]←R16, R16←Memory[Z]`

Arrays Length

- The length of an array is the number of elements (values) in it
- The symbolic name of the array is the starting address of the array
- To determine length use:

```
array_len: .DB (array_len - array)*2/type
```

  - *2 for CSEG
  - type = 1, 2, 4, 8 for byte, word, doubleword, quadword
  - Length definition must follow array IMMEDIATELY

Example: Array Length

```
                .CSEG
arrB:           .DB     1,2,3,4,5,6
arrB_len:       .DB     (arrB_len – arrB)*2          ;
arrW:           .DW     1,2,3,4,5,6
arrW_len:       .DB     (arrW_len – arrW)*2/2
arrD:           .DD     1,2,3,4,5,6
arrD_len:       .DB     (arrD_len – arrD)*2/4        ;
arr_doh:        .DB     (arr_doh – arrB)*2
```

```
0x000000   01 02 03 04 05 06 06 00 01 00 02 00 03 00 04 00
0x000010   05 00 06 00 06 00 01 00 00 00 02 00 00 00 03 00
0x000020   00 00 04 00 00 00 05 00 00 00 06 00 00 00 06 00
0x000030   30 00 ff ff ff ff ff ff ff ff ff ff ff ff ff ff
```