Assembly Language Programming

EE3376
Moving Up Levels of Abstraction

Problems
Algorithms
Language
Machine (ISA) Architecture
Microarchitecture
Circuits
Devices

Assembly code
Machine code
MSP430 Architecture
Logic gates, multiplexers, memory, etc.
Transistors

Adapted from notes from BYU ECE124
High Level vs. Assembly

- **High Level Languages**
  - More programmer friendly
  - More ISA independent
  - Each high-level statement translates to several instructions in the ISA of the computer

- **Assembly Languages**
  - Lower level, closer to ISA
  - Very ISA-dependent
  - Each instruction specifies a single ISA instruction
  - Makes low level programming more user friendly
  - More efficient code

Adapted from notes from BYU ECE124
Assembler Syntax

- Each assembly line begins with either a label, a blank (tab), an asterisk, or a semicolon.
- Each line has four fields:
  
  `{label[:]} mnemonic {operand list} {;comment}`

- Some line examples are:

  ```
  .sect "sysmem" ; data space
  var1 .word 2 ; variable var1 declaration
  .text ; program space
  loop: mov #COUNT,r5 ; get counter
  .end ; end of program
  ```

Adapted from notes from BYU ECE124
Symbols / Labels

- **Symbols**
  - Symbols are used as labels, constants, and substitution values
  - Symbols are stored in a symbol table
  - A symbol name
    - is a string of up to 200 alphanumeric characters (A-Z, a-z, 0-9, $, and _)
    - cannot contain embedded blanks
    - first character cannot be a number
    - case sensitive
  - Symbols used as labels become symbolic addresses that are associated with locations in the program

- **Label Field**
  - Labels are symbols
  - Labels must begin in column 1.
  - A label can optionally be followed by a colon
  - The value of a label is the current value of the Location Counter (address within program)
  - A label on a line by itself is a valid statement
  - Labels used locally within a file must be unique.
Mnemonics / Operands

- **Mnemonic Field**
  - The mnemonic field follows the label field.
  - The mnemonic field cannot start in column 1; if it does, it is interpreted as a label.
  - The mnemonic field contains one of the following items:
    - MSP430 instruction mnemonic (ie. ADD, MOV, JMP)
    - Assembler directive (ie. .data, .list, .equ)
    - Macro directive (ie. .macro, .var, .mexit)
    - Macro call

- **Operand Field**
  - The operand field follows the mnemonic field and contains one or more operands.
  - The operand field is not required for all instructions or directives.
  - An operand may consist of:
    - Symbols
    - Constants
    - Expressions (combination of constants and symbols)
  - Operands are separated with commas
Assembler Directives

- Assembly directives are used to specify:
  - Starting addresses for programs
  - Starting values for memory locations
  - Specify the end of program text.
;;*******************************************************************************
;; CS/ECEn 124 Lab 4 - morse.asm: Student Code
;;*******************************************************************************
.cdecls C, LIST, "msp430x22x4.h" ; include C header
COUNT .equ 2000

.data
.cntr,2 ; ISR counter

.text
Program reset
RESET:
  mov.w #0x0280,SP ; Initialize stack pointer
  mov.w #WDT_MDLY_0_5,&WDTCTL ; Set Watchdog interval to ~0.5ms
  mov.b #WDTIE,&IE1 ; Enable WDT interrupt
  bis.b #0x01,&P1DIR ; P1.0 output
  bis.b #0x20,&P4DIR ; P4.0 output
  mov.w #COUNT,&cnt ; initialize counter
  bis.w #LPM0+GIE,SR ; Enter LPM0 w/ interrupt
  jmp $ ; Loop forever; interrupts do all

; Watchdog Timer interrupt service routine
;
WDT_ISR:
  xor.b #0x20,&P4OUT ; pulse buzzer
  dec.w &cnt ; decrement counter
  jne WDT_exit ; initialize counter
  mov.w #COUNT,&cnt ; toggle P1.0
  xor.b #0x01,&P1OUT
WDT_exit:
  reti ; return from interrupt

.sect ".int10" ; MSP430 RESET Vector
.word WDT_ISR ; Watchdog ISR
.sect ".reset" ; MSP430 RESET Vector
.word RESET ; Power Up ISR
.end

Adapted from notes from BYU ECE124
### Common Assembler Directives

<table>
<thead>
<tr>
<th>Mnemonic and Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.bss symbol, size in bytes[, alignment]</code></td>
<td>Reserves size bytes in the .bss (uninitialized data) section</td>
</tr>
<tr>
<td><code>.sect &quot;section name&quot;</code></td>
<td>Assembles into a named (initialized) section</td>
</tr>
<tr>
<td><code>.text</code></td>
<td>Assembles into the .text (executable code) section</td>
</tr>
<tr>
<td><code>.byte value_1[, ..., value_n]</code></td>
<td>Initializes one or more successive bytes in the current section</td>
</tr>
<tr>
<td><code>.string &quot;string_1&quot;[, ..., &quot;string_n&quot;]</code></td>
<td>Initializes one or more text strings</td>
</tr>
<tr>
<td><code>.word value_1[, ..., value_n]</code></td>
<td>Initializes one or more 16-bit integers</td>
</tr>
<tr>
<td><code>.align [size in bytes]</code></td>
<td>Aligns the LC on a boundary specified by size in bytes; must be a power of 2; defaults to word (2 byte)</td>
</tr>
<tr>
<td><code>.def symbol_1[, ..., symbol_n]</code></td>
<td>Identifies one or more symbols that are defined in current module and that can be used in other modules</td>
</tr>
<tr>
<td><code>.include [&quot;filename&quot;]</code></td>
<td>Includes source statements from another file</td>
</tr>
<tr>
<td><code>.ref symbol_1[, ..., symbol_n]</code></td>
<td>Identifies one or more symbols used in the current module that are defined in another module</td>
</tr>
<tr>
<td><code>symbol .equ value</code></td>
<td>Equates value with symbol</td>
</tr>
<tr>
<td><code>symbol .set value</code></td>
<td>Equates value with symbol</td>
</tr>
<tr>
<td><code>.cdecls [options,&quot; filename&quot;]</code></td>
<td>Share C headers between C and assembly code</td>
</tr>
<tr>
<td><code>.end</code></td>
<td>Ends program</td>
</tr>
</tbody>
</table>

Adapted from n.+.s from BYU ECE124
CCS Window – C/C++ Perspective

1-click project debug and Programming view

Project View
- List of Projects

Code Window
- Breakpoints
- Syntax highlighting

Console Build information

Problems View
- Information
- Warnings
- Errors

Adapted from notes from BYU ECE124
A line in a listing file has four fields:

- Field 1: contains the source code line counter
- Field 2: contains the section program counter
- Field 3: contains the object code
- Field 4: contains the original source statement.

---

**Adapted from notes from BYU ECE124**
Compilation

The assembly language stage is often skipped…

- **Algorithm**
  - by hand

- **C-language program**
  - \( c = a + b; \)

- **Assembly language program**
  - ADD r4, r5

- **Machine language programs**
  - 0100 0100 0000 0101

Compiler often directly generates machine code.

However, low-level assembly language is often used for programming directly.

We will start from assembly language but use high-level C language to help understand it.
MSP 430 Micro-Architecture

Information:
- Memory Address Register
- Program Counter
- Address Bus
- Source Operand
- Destination Operand
- Memory
- Port 1 Output
- Data Bus
- Condition Codes
- Instruction Register
- Arithmetic Logic Unit

Adapted from notes from BYU ECE124
MSP 430 Data Storage

- The MSP430 CPU has 64KB memory space and 16 registers for data storage

- **R0 (PC) – Program Counter**
  - This register always points to the next instruction to be fetched

- **R1 (SP) – Stack Pointer**
  - The MSP430 CPU stores the return address of routines or interrupts on the stack
  - User programs store local data on the stack

- **R2 (SR/CG1) – Status Register**
  - The status of the MSP430 CPU is defined by a set of bits contained in register R2

Adapted from notes from BYU ECE124
R2 (SR/CG1), R3 (CG2) – Constant Generators

- Six different constants commonly used in programming can be generated using the registers R2 and R3, without adding a 16-bit extension word of code to the instruction

<table>
<thead>
<tr>
<th>Register</th>
<th>As</th>
<th>Constant</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>00</td>
<td>-</td>
<td>Register mode</td>
</tr>
<tr>
<td>R2</td>
<td>01</td>
<td>(0)</td>
<td>Absolute mode</td>
</tr>
<tr>
<td>R2</td>
<td>10</td>
<td>00004h</td>
<td>+4, bit processing</td>
</tr>
<tr>
<td>R2</td>
<td>11</td>
<td>00008h</td>
<td>+8, bit processing</td>
</tr>
<tr>
<td>R3</td>
<td>00</td>
<td>00000h</td>
<td>0, word processing</td>
</tr>
<tr>
<td>R3</td>
<td>01</td>
<td>00001h</td>
<td>+1</td>
</tr>
<tr>
<td>R3</td>
<td>10</td>
<td>00002h</td>
<td>+2, bit processing</td>
</tr>
<tr>
<td>R3</td>
<td>11</td>
<td>0FFFFh</td>
<td>-1, word processing</td>
</tr>
</tbody>
</table>

Adapted from notes from BYU ECE124
MSP 430 Registers

- **R4-R15 – General Purpose registers**
  - The general purpose registers R4 to R15 can be used as data registers, data pointers and indices.
  - They can be accessed either as a byte or as a word.
  - Instruction formats support byte or word accesses.
  - The status bits of the CPU in the SR are updated after the execution of a register instruction.

Adapted from notes from BYU ECE124
MSP430G2553 Memory Map
Format I: 12 Double Operand Instructions

Double operand instructions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic instructions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD(.B or .W) src,dst</td>
<td>src+dst→dst</td>
<td>Add source to destination</td>
</tr>
<tr>
<td>ADDC(.B or .W) src,dst</td>
<td>src+dst+C→dst</td>
<td>Add source and carry to destination</td>
</tr>
<tr>
<td>DADD(.B or .W) src,dst</td>
<td>src+dst+C→dst (dec)</td>
<td>Decimal add source and carry to destination</td>
</tr>
<tr>
<td>SUB(.B or .W) src,dst</td>
<td>dst+.not.src+1→dst</td>
<td>Subtract source from destination</td>
</tr>
<tr>
<td>SUBC(.B or .W) src,dst</td>
<td>dst+.not.src+C→dst</td>
<td>Subtract source and not carry from destination</td>
</tr>
<tr>
<td><strong>Logical and register control instructions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND(.B or .W) src,dst</td>
<td>src.and.dst→dst</td>
<td>AND source with destination</td>
</tr>
<tr>
<td>BIC(.B or .W) src,dst</td>
<td>.not.src.and.dst→dst</td>
<td>Clear bits in destination</td>
</tr>
<tr>
<td>BIS(.B or .W) src,dst</td>
<td>src.or.dst→dst</td>
<td>Set bits in destination</td>
</tr>
<tr>
<td>BIT(.B or .W) src,dst</td>
<td>src.and.dst</td>
<td>Test bits in destination</td>
</tr>
<tr>
<td>XOR(.B or .W) src,dst</td>
<td>src.xor.dst→dst</td>
<td>XOR source with destination</td>
</tr>
<tr>
<td><strong>Data instructions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP(.B or .W) src,dst</td>
<td>dst-src</td>
<td>Compare source to destination</td>
</tr>
<tr>
<td>MOV(.B or .W) src,dst</td>
<td>src→dst</td>
<td>Move source to destination</td>
</tr>
</tbody>
</table>

Adapted from notes from BYU ECE124
Examples

MOV.w #0x08,R5 ; move source to destination
    ; assign a hexadecimal value 0x08 to Register R5

AND.w #0x00,R6 ; bitwise AND source with destination
    ; whatever value in R6 is ANDed with 0 -> R6=0

ADD.w #0x03,R6 ; add source to destination
    ; R6 = R6+3 = 0+3 = 3

SUB.w R6, R5 ; subtract source from destination
    ; R5 = R5-R6 = R5+(Not R6)+1 = 8-3 = 5

XOR.w R6, R5 ; bitwise XOR source with destination
    ; R5 = 0011 XOR 0101 = 0110 = 6

BIC.w #0x03, R5 ; clear bits in destination
    ; (Not 0011) AND 0110 = 1100 AND 0110 = 0100 = 4

BIS.w #0x08, R5 ; set bits in destination
    ; 1000 OR 0100 = 1100 = 12

BIT.w #0x08, R5 ; test bits in destination
    ; 1000 AND 1100 = 1000 -> Bit 3 is not zero

CMP.w R6, R5 ; compare source to destination
    ; R5-R6 = 12-6 = 6 greater than 0, so R5 > R6

Adapted from notes from BYU ECE124
## Format II: 7 Single Operand Instructions

### Single operand instructions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR (.B or .W) dst</td>
<td>MSB→MSB→…LSB→C</td>
<td>Logical and register control instructions</td>
</tr>
<tr>
<td>RRA (.B or .W) dst</td>
<td>MSB→MSB→…LSB→C</td>
<td>Roll destination right</td>
</tr>
<tr>
<td>RRC (.B or .W) dst</td>
<td>C→MSB→…LSB→C</td>
<td>Roll destination right through carry</td>
</tr>
<tr>
<td>SWPB (.B or .W) dst</td>
<td>Swap bytes</td>
<td>Swap bytes in destination</td>
</tr>
<tr>
<td>SXT dst</td>
<td>bit 7→bit 8…bit 15</td>
<td>Sign extend destination</td>
</tr>
<tr>
<td>PUSH (.B or .W) src</td>
<td>SP-2→SP, src→@SP</td>
<td>Push source on stack</td>
</tr>
<tr>
<td>CALL (.B or .W) dst</td>
<td>SP-2→SP, PC+2→@SP, dst→PC</td>
<td>Subroutine call to destination</td>
</tr>
<tr>
<td>RETI</td>
<td>@SP+→SR, @SP+→SP</td>
<td>Return from interrupt</td>
</tr>
</tbody>
</table>

Adapted from notes from BYU ECE124
Examples

```plaintext
MOV.w  #0xF009,R5 ; move source to destination
               ; assign a hexadecimal value 0x08 to Register R5

RRA.w  R5      ; Roll destination right and send LSB to Carry
               ; 1111 0000 0000 1001 -> 1111 1000 0000 0100  C=1

RRC.w  R5      ; Roll destination right through Carry
               ; 1111 1000 0000 0100 -> 1111 1100 0000 0010  C=0

SWPB.w R5     ; subtract source from destination
               ; 1111 1100 0000 0010 -> 0000 0010 1111 1100

SXT       R5   ; sign extension
               ; 1111 1100 -> 1111 1111 1111 1100
               ; (bit 7 is 1) bits 8~15 are all set to 1
```

PUSH, CALL, and RETI will be discussed later

Adapted from notes from BYU ECE124
Jump Instruction Format

- Jump instructions are used to direct program flow to another part of the program.
- The condition on which a jump occurs depends on the Condition field consisting of 3 bits:
  - JNE/JNZ: jump if not equal
  - JEQ/JZ: jump if equal
  - JNC/JLO: jump if carry flag equal to zero
  - JC/JHS: jump if carry flag equal to one
  - JN: jump if negative (N = 1)
  - JGE: jump if greater than or equal (N = V)
  - JL: jump if lower (less) (N ≠ V)
  - JMP: unconditional jump (no condition check)
Examples

MOV.w #0x05,R5 ; move source to destination
; assign a hexadecimal value 0x05 to Register R5

MOV.w #0x03,R6 ; move source to destination
; assign a hexadecimal value 0x03 to Register R6

CMP.w R6, R5 ; compare source to destination
; R5-R6 = 5-3 = 2 greater than 0, so R5 > R6

JNE somewhere ; jump if not equal
; The program will jump to “somewhere” because R5 ≠ R6
Emulated Instructions

- In addition to the 27 instructions of the CPU there are 24 emulated instructions.
- The CPU coding is unique.
- The emulated instructions make reading and writing code easier, but do not have their own op-codes.
- Emulated instructions are replaced automatically by CPU instructions by the assembler.
- There are no penalties for using emulated instructions.

Adapted from notes from BYU ECE124
# Emulated Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Emulation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic instructions</strong></td>
<td><strong>ADC(.B or .W) dst</strong></td>
<td>dst+C → dst</td>
<td>ADDC(.B or .W) #0,dst</td>
</tr>
<tr>
<td></td>
<td><strong>DADC(.B or .W) dst</strong></td>
<td>dst+C → dst (decimally)</td>
<td>DADD(.B or .W) #0,dst</td>
</tr>
<tr>
<td></td>
<td><strong>DEC(.B or .W) dst</strong></td>
<td>dst-1 → dst</td>
<td>SUB(.B or .W) #1,dst</td>
</tr>
<tr>
<td></td>
<td><strong>DECD(.B or .W) dst</strong></td>
<td>dst-2 → dst</td>
<td>SUB(.B or .W) #2,dst</td>
</tr>
<tr>
<td></td>
<td><strong>INC(.B or .W) dst</strong></td>
<td>dst+1 → dst</td>
<td>ADD(.B or .W) #1,dst</td>
</tr>
<tr>
<td></td>
<td><strong>INCD(.B or .W) dst</strong></td>
<td>dst+2 → dst</td>
<td>ADD(.B or .W) #2,dst</td>
</tr>
<tr>
<td></td>
<td><strong>SBC(.B or .W) dst</strong></td>
<td>dst+0FFFFh+C → dst</td>
<td>SUBC(.B or .W) #0,dst</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dst+0FFh → dst</td>
<td></td>
</tr>
</tbody>
</table>
### Emulated Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Emulation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical and register control instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INV(.B or .W) dst</td>
<td>.NOT.dst→dst</td>
<td>XOR(.B or .W) #0(FF)FFh,dst</td>
<td>Invert bits in destination</td>
</tr>
<tr>
<td>RLA(.B or .W) dst</td>
<td>C←MSB←MSB-1 LSB+1←LSB←0</td>
<td>ADD(.B or .W) dst,dst</td>
<td>Rotate left arithmetically (multiplied by 2)</td>
</tr>
<tr>
<td>RLC(.B or .W) dst</td>
<td>C←MSB←MSB-1 LSB+1←LSB←C</td>
<td>ADDC(.B or .W) dst,dst</td>
<td>Rotate left through carry</td>
</tr>
<tr>
<td>Program flow control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BR dst</td>
<td>dst→PC</td>
<td>MOV dst,PC</td>
<td>Branch to destination</td>
</tr>
<tr>
<td>DINT</td>
<td>0→GIE</td>
<td>BIC #8,SR</td>
<td>Disable (general) interrupts</td>
</tr>
<tr>
<td>EINT</td>
<td>1→GIE</td>
<td>BIS #8,SR</td>
<td>Enable (general) interrupts</td>
</tr>
<tr>
<td>NOP</td>
<td>None</td>
<td>MOV #0,R3</td>
<td>No operation</td>
</tr>
<tr>
<td>RET</td>
<td>@SP→PC</td>
<td>MOV @SP+,PC</td>
<td>Return from subroutine</td>
</tr>
</tbody>
</table>

Adapted from notes from BYU ECE124
### Emulated Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Emulation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data instructions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLR(.B or .W) dst</td>
<td>0→dst</td>
<td>MOV(.B or .W) #0,dst</td>
<td>Clear destination</td>
</tr>
<tr>
<td>CLRC</td>
<td>0→C</td>
<td>BIC #1,SR</td>
<td>Clear carry flag</td>
</tr>
<tr>
<td>CLRN</td>
<td>0→N</td>
<td>BIC #4,SR</td>
<td>Clear negative flag</td>
</tr>
<tr>
<td>CLRZ</td>
<td>0→Z</td>
<td>BIC #2,SR</td>
<td>Clear zero flag</td>
</tr>
<tr>
<td>POP(.B or .W) dst</td>
<td>@SP→temp SP+2→SP temp→dst</td>
<td>MOV(.B or .W) @SP +,dst</td>
<td>Pop byte/word from stack to destination</td>
</tr>
<tr>
<td>SETC</td>
<td>1→C</td>
<td>BIS #1,SR</td>
<td>Set carry flag</td>
</tr>
<tr>
<td>SETN</td>
<td>1→N</td>
<td>BIS #4,SR</td>
<td>Set negative flag</td>
</tr>
<tr>
<td>SETZ</td>
<td>1→Z</td>
<td>BIS #2,SR</td>
<td>Set zero flag</td>
</tr>
<tr>
<td>TST(.B or .W) dst</td>
<td>dst + 0FFFFh + 1</td>
<td>CMP(.B or .W) #0,dst</td>
<td>Test destination</td>
</tr>
<tr>
<td></td>
<td>dst + 0FFh + 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adapted from notes from BYU ECE124
Example: Emulated Instructions

- Emulated instructions are replaced automatically by CPU instructions by the assembler
- Could be replaced differently
- Clear the contents of register R5:
  \[\text{CLR R5} = \text{MOV.W} \#0, \text{R5}\]
- Increment the content of register R5:
  \[\text{INC R5} = \text{ADD.W} \#1, \text{R5}\]
- Decrement the content of register R5:
  \[\text{DEC R5} = \text{SUB.W} \#1, \text{R5}\]
Example: Emulated Instructions

- Decrement by two the contents of register R5:
  \[
  \text{DECD R5} = \text{SUB.W} \ #2, \ R5
  \]

- Do not carry out any operation:
  \[
  \text{NOP} = \text{MOV.W} \ R3, \ R3
  \]

- Add the carry flag to the register R5:
  \[
  \text{ADC R5} = \text{ADC.W} \ #0, \ R5
  \]
Source Addressing Modes

- The MSP430 has four basic modes for the source address:
  - Rs - Register
  - x(Rs) - Indexed Register
  - @Rs - Register Indirect (not for destination)
  - @Rs+ - Indirect Auto-increment (not for destination)

- In combination with registers R0-R3, three additional source addressing modes are available:
  - label - PC Relative, x(PC)
  - &label – Absolute, x(SR)
  - #n – Immediate, @PC+ (not for destination)
Destination Addressing Modes

- There are two basic modes for the destination address:
  - \texttt{Rd} - Register
  - \texttt{x(Rd)} - Indexed Register

- In combination with registers R0/R2, two additional destination addressing modes are available:
  - \texttt{label} - PC Relative, \texttt{x(PC)}
  - \texttt{&label} – Absolute, \texttt{x(SR)}

Adapted from notes from BYU ECE124
Register Mode (Rn)

- The most straightforward addressing mode and is available for both source and destination
  - Example:
    \[ \text{mov.w r5,r6} \quad ; \text{move word from r5 to r6} \]
- The registers are specified in the instruction; no further data is needed
- Also the fastest mode and does not require an addition cycle
- Byte instructions use only the lower byte, but clear the upper byte when writing
Indexed Mode x(Rn)

- The address is formed by adding a constant (index) to the contents of a CPU register
  - Example:
    
    ```
    mov.b 3(r5),r6  
    ; move byte from
    ;   M(3\text{\_}10+r5) to r6
    ```

- Indexed addressing can be used for source and/or destination, value in r5 is unchanged.

- The index is located in the memory word following the instruction and requires an additional memory cycle

- There is no restriction on the address for a byte, but words must lie on even addresses
Symbolic Mode (PC Relative)

- The address is formed by adding a constant (index) to the program counter (PC)
  - **Example:** (mov.w x(PC), r6 where x=Cnt-PC)
    
    ```
    mov.w Cnt, r6 ; move word
    ; M(Cnt) or M(x+PC) to r6
    ```

- The PC relative index is calculated by the assembler
- Produces position-independent code, but rarely used in the MSP430 because absolute addressing can reach all memory addresses
- Note: this is NOT an appropriate mode of addressing when referencing fixed locations in memory such as the special function registers (SFR’s)
Absolute Mode (&label)

- The address is formed directly from a constant (index) and specified by preceding a label with an ampersand (&)
  - Example: (mov.w x(SR), r6 where 0 is used for SR)
    
    \[
    \text{mov.w } \&\text{Cnt}, \text{r6} \quad ; \quad \text{move word} \\
    ; \quad \text{M(Cnt) to r6}
    \]

- Same as indexed mode with the base register value of 0 (by using the status register SR as the base register)
- The absolute address is stored in the memory word following the instruction and requires an additional cycle
- Note: this is the preferred mode of addressing when referencing fixed locations in memory such as the special function registers (SFR’s)
Indirect Register Mode (@Rn)

- The address of the operand is formed from the contents of the specified register
  - Example:
    
    ```
    mov.w @r5,r6 ; move word
    ; M(r5) to r6
    ```

- Only available for source operands
- Same as indexed mode with index equal to 0, but does not require an additional instruction word
- The value of the indirect register is unchanged

<table>
<thead>
<tr>
<th>Op-code</th>
<th>S-reg</th>
<th>Ad</th>
<th>b/w</th>
<th>As</th>
<th>D-reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Adapted from notes from BYU ECE124
Indirect Autoincrement Mode (@Rn+)

- The address of the operand is formed from the contents of the specified register and afterwards, the register is automatically incremented by 1 if a byte is fetched or by 2 if a word is fetched.
  - Example:
    
    ```
    mov.w @r5+,r6 ; move word
    ; M(r5) to r6
    ; increment r5 by 2
    ```

- Only available for source operands.
- Usually called post-increment addressing.
- **Note:** All operations on the first address are fully completed before the second address is evaluated.

---

<table>
<thead>
<tr>
<th>Op-code</th>
<th>S-reg</th>
<th>Ad</th>
<th>b/w</th>
<th>As</th>
<th>D-reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0</td>
<td>0 1 0 1</td>
<td>0 0</td>
<td>1 1</td>
<td>0 1</td>
<td>1 0</td>
</tr>
</tbody>
</table>

Adapted from notes from BYU ECE124
Immediate Mode (#n)

- The operand is an immediate value
  - Example (mov.w @PC+, r6)
    
    \[
    \text{mov.w } \#100, \text{r6} \quad ; \quad 100 \rightarrow \text{r6}
    \]

- The immediate value is located in the memory word following the instruction
- Only available for source operands
- The immediate mode of addressing is a special case of auto-increment addressing that uses the program counter (PC) as the source register.
- The PC is automatically incremented after the instruction is fetched; hence points to the following word

<table>
<thead>
<tr>
<th>Op-code</th>
<th>S-reg</th>
<th>Ad</th>
<th>b/w</th>
<th>As</th>
<th>D-reg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adapted from notes from BYU ECE124
Examples - Source

```assembly
mov.w R5, R6 ; move the content of R5 (0010) to R6
mov.w @R7, R8 ; use the content of R7 (9000) as the address to move data (000F) to R8
mov.w &0x9004, R6 ; go to the absolute address (9004) to move data (0011) to R6
mov.w 2(R7), R9 ; use the content of R7 (9000) as the base address and offset it by 2 (9002)
                 ; to move data (0010) to R9
mov.w &0x9006, R5 ; go to the absolute address (9006) to move data (0012) to R5
mov.w @R7+, R6 ; use the content of R7 (9000) as the address to move data (000F) to R6 and
                 ; then increment R7 by 2 (one word is 2 bytes)
mov.w #0x9000, R7 ; move the immediate value (9000) to R7
mov.w label, R7 ; move the data (0012) in the memory space represented by “label” to R7
```

Adapted from notes from BYU ECE124
Examples - Destination

<table>
<thead>
<tr>
<th>registers</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>R5: 0010</td>
<td>x9000: 000F</td>
</tr>
<tr>
<td>R6: 0020</td>
<td>x9002: 0010</td>
</tr>
<tr>
<td>R7: 9000</td>
<td>x9004: 0011</td>
</tr>
<tr>
<td>R8: 9004</td>
<td>x9006: 0012</td>
</tr>
<tr>
<td>R9: 9006</td>
<td>x9008: 0013</td>
</tr>
</tbody>
</table>

- **mov.w R5, R6**: move the content of R5 (0010) to R6
- **mov.w R5, 4(R7)**: use the content of R7 (9000) as the base address and offset it by 4 (9004) and move the content of R5 (0010) to the effective address (9004)
- **mov.w R6, label**: move the content of R6 (0012 moved from R5) to the memory space represented by “label”
- **mov.w &0x9004, &0x9008**: go to the absolute address (9004) to move data (0011) to the memory location of the absolute address (9008)
- **mov.w R6, &label**: move the content of R6 (0012 moved from R5) to the memory space represented by the absolute address “label”

Adapted from notes from BYU ECE124
High Level vs. Assembly

- **High Level Languages**
  - More programmer friendly
  - More ISA independent
  - Each high-level statement translates to several instructions in the ISA of the computer

- **Assembly Languages**
  - Lower level, closer to ISA
  - Very ISA-dependent
  - Each instruction specifies a single ISA instruction
  - Makes low level programming more user friendly
  - More efficient code

Adapted from notes from BYU ECE124
The first assembly program

; MSP430 Micro-Architecture Simulator Code;
; Description:
; Display an incrementing counter in the simulator LEDs.
;      MSP430
;            ----------------
;            |                 |
;            |             P1.0|-->RED LED
;            |             P1.1|-->YELLOW LED
;            |             P1.2|-->BLUE LED
;            |             P1.3|-->GREEN LED
;*******************************************************************************
.cdecls C,LIST,  "msp430x22x4.h"

.text                            ; Program reset
.reset                            ; MSP430 RESET Vector
 MOV.W   $0300,SP               ; Initialize stack pointer
 MOV.W   WDTPW+WDTHOLD,&WDTCTL   ; Stop WDT
 BIS.B   $0f,&P1DIR             ; Set P1.0-3 output
 MOV.W   $0,r14
 Mainloop:   MOV.B   r14,&P1OUT               ; output P1.0-3
 ADD.W   $000f,r14              ; mask counter
 Wait:      MOV.W   Delay,r15                ; Delay to R15
 Push    r15
 Dec.w   0(sp)                    ; decrement delay counter
 JNZ     L1                       ; delay over?
 JMP     Mainloop                 ; repeat
 Delay:      .word   2
 .sect   ".reset"
 .short   RESET
 .end
What does it do?

Reset:

Move an immediate value #0x0300 into stack pointer register SP (R1)

mov.w #0x0300,SP               ; Initialize stack pointer

Move an immediate value that is the OR’ed result of WDTPW and WDTHOLD (both are predefined memory addresses) into a memory location at the absolute address WDTCTL (also a predefined address)

mov.w #WDTPW+WDTHOLD,&WDTCTL ; Stop WDT

Set the lowest 4 bits of the value stored in the absolute address location P1DIR to all ‘1’. This is to set Pins 0~3 of Port 1 to be for output controls (turn LEDs on or off)

bis.b #0x0f,&P1DIR             ; Set P1.0-3 output

mov.w #0,r14

Mainloop:

Move an immediate value 0 to register #14. 0 is actually created by hardware

mov.b r14,&P1OUT               ; output P1.0-3

inc.w r14

and.w #0x000f,r14              ; mask counter

mov.w Delay,r15                ; Delay to R15

push r15

L1:

Decrement the value stored on top of the stack

dec.w 0(sp)                    ; decrement delay counter

if the previous instruction result is NOT zero, then jump to location labeled L1, else continue to the next instruction

jnz L1

mov.w @sp+,r15                 ; y

jmp Mainloop                   ; repeat

Delay:

Move the value stored in register #14 to the absolute address location PIOUT (predefined for Port 1 output register)

.word 2                         ; word directive assigned a work value 2 to Delay

Move the value in r15 onto the stack

inc.w r14

Move the value stored on top of the stack to r15 and then increment SP by 2

And the value in register #14 with 0x000F (0000 0000 0000 1111) to keep only the lowest 4 bits in r14.

L1: Unconditionally (always) jump to location labeled Mainloop

mov.w #0x000f,&r14              ; Mask counter

mov.w Delay,r15

push r15

Decrement the value stored on top of the stack
What does it do?

RESET:
- `mov.w #0x0300,SP` ; Initialize stack pointer
- `mov.w #WDTPW+WDTHOLD,&WDTCTL` ; Stop WDT
- `bis.b #0x0f,&P1DIR` ; Set P1.0-3 output
- `mov.w #0,r14`

Mainloop: `mov.b r14,&P1OUT` ; output P1.0-3
- `inc.w r14`
- `and.w #0x000f,r14` ; mask counter
Wait:
- `mov.w Delay,r15` ; Delay to R15
- `push r15`

L1:
- `dec.w 0(sp)` ; decrement delay counter
- `jnz L1` ; delay over?
- `mov.w @sp+,r15` ; y
- `jmp Mainloop` ; repeat

Delay: `.word 2`

Adapted from notes from BYU ECE124
Three Basic Constructs

Sequential

Conditional

Iterative

Adapted from notes from BYU ECE124
if (buzzerON == 1)
{
pulse_buzzer();
turn_on_LED();
}
else
{
turn_off_LED();
}

Could be different addressing modes

Adapted from notes from BYU ECE124
while Translation

while_loop:
  bit.w #1,R4
  jnz while_done
  call #LED_ON
  call #delay
  call #LED_OFF
  call #delay
  jmp while_loop

while_done:

#define TRUE 1

while (TRUE)
{
  LED_ON();
  delay();
  LED_OFF();
  delay();
}

Adapted from notes from BYU ECE124
for-loop Translation

```
.bss i,2 ; int i;

mov.w #0,&i ; for(i=0; i<10; i++)
fl_ck: cmp.w #10,&i   ; { 
jge for_done ;
call #do_dot ;   do_dot();
call #delay ;   delay();
call #do_dash ;   do_dash();
call #delay ;   delay();
add.w #1,&i ;
jmp fl_ck ;
for_done: ;
```

Adapted from notes from BYU ECE124
Better for-loop Translation

```c
.bss i,2 ; int i;

mov.w #0,&i ; for(i=10; i>0; i--)
fl_ck: call #do_dot ; {  do_dot();
call #delay ;  delay();
call #do_dash ;  do_dash();
call #delay ;  delay();
dec.w &i ;
jnz  fl_ck ; }
for_done: ;
```

Adapted from notes from BYU ECE124
switch/case Translation

- switch/case

```assembly
cmp.w #DOT,&myByte ; switch (myByte)
jne sw_01 ; {
call #do_dot ; case DOT:
jmp sw_end ; do_dot(); break;

sw_01:
    cmp.w #DASH,&myByte ; case DASH:
jne sw_end ; do_dash();
call #do_dash ; break;
jmp sw_end ;
}

sw_end: ;
```

Adapted from notes from BYU ECE124
# Cycles Per Instruction...

<table>
<thead>
<tr>
<th>Src</th>
<th>Dst</th>
<th>Cycles</th>
<th>Length</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>MOV R5,R8</td>
</tr>
<tr>
<td></td>
<td>@Rm</td>
<td>2</td>
<td>1</td>
<td>MOV R5,@R6</td>
</tr>
<tr>
<td></td>
<td>x(Rm)</td>
<td>4</td>
<td>2</td>
<td>ADD R5,4(R6)</td>
</tr>
<tr>
<td></td>
<td>EDE</td>
<td>4</td>
<td>2</td>
<td>XOR R8,EDE</td>
</tr>
<tr>
<td></td>
<td>&amp;EDE</td>
<td>4</td>
<td>2</td>
<td>MOV R5,&amp;EDE</td>
</tr>
<tr>
<td>#n</td>
<td>x(Rm)</td>
<td>5</td>
<td>3</td>
<td>MOV #100,TAB(R8)</td>
</tr>
<tr>
<td>&amp;TONI</td>
<td>&amp;EDE</td>
<td>6</td>
<td>3</td>
<td>MOV &amp;TONI,&amp;EDE</td>
</tr>
</tbody>
</table>

See “How To Determine Cycles Per Instruction...” in Blinky Lab instructions.

Adapted from notes from BYU ECE124
Include Symbolic & Absolute Address

(c) One operand (Format II)

\[
\begin{align*}
\text{rra.w Rs} & : \\
\text{fetch instruction} &,
\end{align*}
\]

Adapted from notes from BYU ECE124
Instruction Timing

```
mainloop:    xor.b  #0x01,&P1OUT            ; toggle P1.0
            mov.w  #0,r15                  ; use R15 as delay counter
delayloop:  dec.w  r15                     ; delay over?
            jnz   delayloop                 ; n
            jmp   mainloop                ; y, toggle led
```

4 cycles
1 cycle
1 cycle
2 cycles
2 cycles