Lecture 3
MSP430 Assembly Language Instructions

Texas Instruments Incorporated
University of Beira Interior (PT)

Pedro Dinis Gaspar, António Espírito Santo, Bruno Ribeiro, Humberto Santos
University of Beira Interior, Electromechanical Engineering Department
www.msp430.ubi.pt

Contents

Introduction

- The following section introduces some fundamentals of assembly language programming using the MSP430 family of microcontrollers;

- Rather than to make an exhaustive study of programming methodologies and techniques, the intention is to focus on the aspects of assembly language programming relevant to the MSP430 family;

- The examples are based on the MSP430 CPU;

UBI

System status flags (1/7)

- **Modification of bits:**
  - The state of one or more bits of a value can be changed by the bit clear (`BIC`) and bit set (`BIS`) instructions, as described below;
  - The `BIC` instruction clears one or more bits of the destination operand. This is carried out by inverting the source value then performing a logical `&` (`AND`) operation with the destination. Therefore, if any bit of the source is one, then the corresponding bit of the destination will be cleared to zero.

```
BIC source,destination or BIC.W source,destination
BIC.B source,destination
```

System status flags (2/7)

- **Modification of bits (continued):**

  - For example, there is the bit clear instruction `BIC`:

```
BIC    #0x000C,R5
```

  - This clears bits 2 and 3 of register R5, leaving the remaining bits unchanged.

  - There is also the bit set (`BIS`) instruction:

```
BIS source,destination or BIS.W source,destination
BIS.B source,destination
```

System status flags (3/7)

- **Modification of bits (continued):**

  `BIS source,destination or BIS.W source,destination`

  `BIS.B source,destination`

  - This sets one or more bits of the destination using a similar procedure to the previous instruction;
  - The instruction performs a logical `|` (`OR`) between the contents of the source and destination.
  - For example, the instruction:

```
   BIS   #0x000C,R5
```

System status flags (4/7)

❑ **Modification of bits (continued):**

```
BIS    #0x000C,R5
```

- Sets bits 2 and 3 of register R5, leaving the remaining bits unchanged;
- It is recommended that whenever it is necessary to create control flags, these are located in the least significant nibble of a word;
- In this case, the CPU constant generator can generate the constants necessary (1, 2, 4, 8) for bit operations;
- The code produced is more compact and therefore the execution will be faster.

System status flags (5/7)

- **CPU status bits modification:**

  - The CPU contains a set of flags in the Status Register (SR) that reflect the status of the CPU operation, for example that the previous instruction has produced a carry (C) or an overflow (V).

  - It is also possible to change the status of these flags directly through the execution of emulated instructions, which use the `BIC` and `BIS` instructions described above.

System status flags (6/7)

- **Directly changing the CPU status flags:**

  - The following instructions clear the CPU status flags (C, N and Z):

```
CLRC; clears carry flag (C). Emulated by BIC #1,SR
CLRN; clears negative flag (N). Emulated by BIC #4,SR
CLRZ; clears the zero flag (Z). Emulated by BIC #2,SR
```

  - The following instructions set the CPU status flags (C, N and Z):

```
SETC; set the carry flag (C). Emulated by BIS #1,SR
SETN; set the negative flag (N). Emulated by BIS #4,SR
SETZ; set the zero flag (Z). Emulated by BIS #2,SR
```

System status flags (7/7)

- **Enable/disable interrupts:**
  - Two other instructions allow the flag that enables or disables the interrupts to be changed.
  - The global interrupt enable `GIE` flag of the register `SR` may be cleared to disable interrupts:

```
DINT; Disable interrupts. (emulated by BIC #8,SR)
```

  - Or the `GIE` may be set to enable interrupts:

```
EINT; Enable interrupts (emulated by BIS #8,SR)
```

UBI

Arithmetic and logic operations (1/31)

- **Addition and Subtraction:**

  - The MSP430 CPU has instructions that perform addition and subtraction operations, with and without input of the carry flag (C).

  - It is also possible to perform addition operations of values represented in binary coded decimal (BCD) format.

  - The CPU has an instruction set to efficiently perform addition and subtraction operations;

Arithmetic and logic operations (2/31)

- **Addition operation:**
  - There are three different instructions to carry out addition operations;
  - The addition of two values is performed by the instruction:

```
ADD source,destination or ADD.W source,destination
ADD.B source,destination
```

  - The addition of two values, also taking into consideration (= adding) the carry bit (C), is performed by the instruction:

```
ADDC source,destination or ADDC.W source,destination
ADDC.B source,destination
```

Arithmetic and logic operations (3/31)

❑ **Addition operation (continued):**

- The addition of the carry bit (C) to a value is provided by instruction:

```
ADC destination or ADC.W destination
ADC.B destination
```

- The CPU status flags are updated to reflect the operation result.

Arithmetic and logic operations (4/31)

- **Addition operation (continued):**
  - For example, two 32-bit values are represented by the combination of registers R5:R4 and R7:R6, where the format is most significant word:least significant word;
  - The addition operation must propagate the carry from the addition of the least significant register words (R4 and R6) to the addition of the most significant words (R5 and R7).
  - Two 32-bit values are added in the example presented below:

```
MOV    #0x1234,R5 ; operand 1 most significant word
MOV    #0xABCD,R4 ; operand 1 least significant word

MOV    #0x1234,R7  ; operand 2 most significant word
MOV    #0xABCD,R6 ; operand 2 least significant word

ADD    R4,R6 ; add least significant words
ADDC   R5,R7 ; add most significant words with carry
```

Arithmetic and logic operations (5/31)

- **Addition operation (continued):**

  - The code begins by loading the values into the registers to be added, 0x1234ABCD in R5:R4 and 0x1234ABCD in R7:R6;
  - The operation continues by adding the two least significant words 0xABCD and 0xABCD in registers R4 and R6;
  - The addition may change the carry bit (C), and this must be taken into account during the addition of the two most significant words;
  - The result is placed in the structure formed by the registers R7:R6.

Arithmetic and logic operations (6/31)

- **Subtraction operation:**

  - There are three instructions to perform subtraction operations.

  - The subtraction of two values is performed by the instruction:

    ```
    SUB source,destination or SUB.W source,destination
    SUB.B source,destination  ( dst+.not.src+1→dst )
    ```

  - The subtraction of two values, taking into consideration the carry bit (C), is performed by the instruction:

    ```
    SUBC source,destination or SUBC.W source,destination
    SUBC.B source,destination  ( dst+.not.src+C→dst )
    ```

Arithmetic and logic operations (7/31)

❑ **Subtraction operation (continued):**

- The subtraction of destination taking into consideration the carry bit (C) is provided by the instruction:

  ```
  SBC destination
  or SBC.W destination ( dst+0FFFFh+C→dst )
  SBC.B destination ( dst+0FFh+C→dst )
  ```

- The CPU status flags are updated to reflect the result of the operation;

- The borrow is treated as a .NOT. carry: The carry is set to 1 if NO borrow, and reset if borrow.

UBI

Arithmetic and logic operations (8/31)

- **Subtraction operation (continued):**
  - For example, two 32-bit values are represented by the combination of registers R5:R4 and R7:R6, where the format is most significant word:least significant word;
  - The subtraction operation of these values must propagate the carry (C) from the subtraction of the least significant words (R4 and R6) to the most significant words (R5 to R7);
  - Two 32-bit values are subtracted in the example presented below:

```
MOV     #0xABCD,R5    ; load operand 1 in R5:R4
MOV     #0x1234,R4

MOV     #0x0000,R7    ; load operand 2 in R7:R6
MOV     #0x1234,R6

SUB     R4,R6  ; subtract least significant words
SUBC    R5,R7  ; subtract most significant words with carry
```

Arithmetic and logic operations (9/31)

- **Subtraction operation (continued):**
  - The code begins by loading the values in the registers to be subtracted, 0xABCD1234 into R5:R4 and 0x00001234 into R7:R6;
  - Next, the next operation is to subtract the two least significant words;
  - The result of the subtraction affects the carry bit (C), which must be taken into account during the subtraction of the two most significant words.

Arithmetic and logic operations (10/31)

- **BCD format addition:**
  - The CPU supports addition operations for values represented in binary coded decimal (BCD) format. There are two instructions to perform addition operations in this format:

```
DADD source,destination or DADD.W source,destination
DADD.B source,destination
```

  - The addition of the carry bit (C) to a BCD value is provided by instruction:

```
DADC destination or DADC.W destination
DADC.B destination
```

  - The CPU status flags are updated to reflect the result of the operation.

Arithmetic and logic operations (11/31)

- **BCD format addition (continued):**
  - For example, two 32-bit BCD values are represented by the combination of registers R5:R4 and R7:R6, where the format is most significant word:least significant word;
  - The addition of these values must propagate the carry from the addition of the least significant words (R4 and R6) to the addition of the most significant words (R5 and R7);
  - Two 32-bit BCD values are added in the example below:

```
MOV     #0x1234,R5   ; operand 1 most significant word
MOV     #0x5678,R4   ; operand 1 least significant word
MOV     #0x1234,R7   ; operand 2 most significant word
MOV     #0x5678,R6   ; operand 2 least significant word
DADD    R4,R6   ; add least significant words with carry
DADC    R5,R7   ; add most significant words
```

UBI

Arithmetic and logic operations (12/31)

- **BCD format addition (continued):**
  - The code begins by loading the BCD values into the registers to be added;
  - Next, the least significant words are added together;
  - The result of the addition generates a carry, which must be added together with the two most significant words.

UBI

Arithmetic and logic operations (13/31)

- **Sign extension:**
  - The CPU supports 8-bit and 16-bit operations;
  - Therefore, the CPU needs to be able to extend the sign of a 8-bit value to a 16-bit format;
  - The extension of the sign bit is produced by the instruction:

```
SXT    destination
```

  - For example, the sign extension of the value contained in R5 is:

```
MOV.B   #0xFC,R5 ;  place the value 0xFC in R5
SXT     R5       ;  sign byte extend. R5 = 0xFFFC
```

Arithmetic and logic operations (14/31)

- **Increment and decrement operations:**
  - There are several operations that need to increment or decrement a value, e.g. control of the number of code iterations (`for` or `while` loops), access to memory blocks using pointers, etc;
  - Therefore, there are four emulated instructions based on the `ADD` instruction, which facilitate the implementation of these operations;
  - To increment a value by one:

```
INC      destination or INC.W destination
INC.B    destination
```

Arithmetic and logic operations (15/31)

❑ Increment and decrement operations (continued):

- Similarly, to decrement a value by one:

```
DEC      destination or DEC.W destination
DEC.B    destination
```

- In the following example, the value placed in register R5 is initially incremented and then decremented:

```
   MOV.B #0x00,R5 ; move 0x00 to register R5
   INC   R5       ; increment R5 by one. R5 = 0x0001
   DEC   R5       ; decrement R5 by one. R5 = 0x0000
```

Arithmetic and logic operations (16/31)

- **Increment and decrement operations (continued):**
  - The ability of the CPU to address 16-bit values requires the ability to increment or decrement the address pointer by two;
  - The following instruction is used to increment a value by two:

```
INCD   destination or INCD.W destination
INCD.B destination
```

  - Similarly, to decrement a value by two, the following instruction is used:

```
DECD   destination or DECD.W destination
DECD.B destination
```

UBI

Arithmetic and logic operations (17/31)

- **Increment and decrement operations (continued):**

  - In the following example, the value stored in register R5 is initially incremented by two and then decremented by two:

```
MOV.B #0x00,R5     ; move 0x00 to the register R5
INCD  R5       ; Increment R5 by two. R5 = 0x0002
DECD  R5       ; Decrement R5 by two. R5 = 0x0000
```

Arithmetic and logic operations (18/31)

- **Logic operations - Logic instructions:**

  - The CPU performs a set of logical operations through the operations `AND` (logical and), `XOR` (exclusive OR) and `INV` (invert). The CPU status flags are updated to reflect the result of the operation;

  - The `AND` logical operation between two operands is performed by the instruction:

```
AND source,destination or AND.W source,destination
AND.B source,destination
```

Arithmetic and logic operations (19/31)

- **Logic operations - Logic instructions (continued):**
  - In the following example, the value 0xFF is moved to the register R4, and the value 0x0C is moved to the register R5;
  - The `AND` logic operation is performed between these two registers, putting the result in register R5:

```
MOV.B #0xFF,R4      ; load operand 1 in R4
MOV.B #0x0C,R5      ; load operand 2 in R5
AND.B R4,R5         ; AND result located in R5
```

  - The code begins by loading the operands in to registers R4 and R5. The result of the logical operation between the two registers is placed in R5;

Arithmetic and logic operations (20/31)

- **Logic operations - Logic instructions (continued):**
  - The `XOR` logic operation between two operands is performed by the instruction:

```
XOR source,destination or XOR.W source,destination
XOR.B source,destination
```

  - In the following example, the value 0xFF is moved to the register R4, and the value 0x0C is moved to the register R5;
  - The logical `XOR` operation is performed between the two registers, putting the result in register R5:

```
MOV.B #0x00FF,R4   ; load operand 1 into R4
MOV.B #0x000C,R5   ; load operand 1 into R5
XOR.B R4,R5        ; XOR result located in R5
```

Arithmetic and logic operations (21/31)

- **Logic operations - Logic instructions (continued):**

  - The `NOT` logical operation on one operand is performed by the `INV` (invert) instruction:

    ```
    INV   destination or INV.W destination
    INV.B destination
    ```

  - The `XOR` logic operation between two operands was performed in the previous example;

  - The following example demonstrates a way to implement an `OR`;

  - The code begins by loading the operands into registers R4 and R5. The contents of the registers is inverted, then the logical `&` (`AND`) operation is performed between them.

Arithmetic and logic operations (22/31)

- **Logic operations - Logic instructions (continued):**

```
MOV #0x1100,R4   ; load operand 1 into R4
MOV #0x1010,R5   ; load operand 2 into R5

INV R4      ; invert R4 bits. R4 = 0xEEFF
INV R5      ; invert R5 bits. R5 = 0xEFEF

AND R4,R5 ; AND logic operation between R4 and R5
INV R5      ; invert R5 bits. R5 = 0x1110
```

- The operation `OR` can also be performed with the BIS instruction.

Arithmetic and logic operations (23/31)

- **Logic operations - Displacement and rotation with carry:**

  - Multiplication and division operations on a value by multiples of 2 are achieved using the arithmetic shift left (multiplication) or the shift right (division) operations;

  - The arithmetic shift left is performed by the instruction:

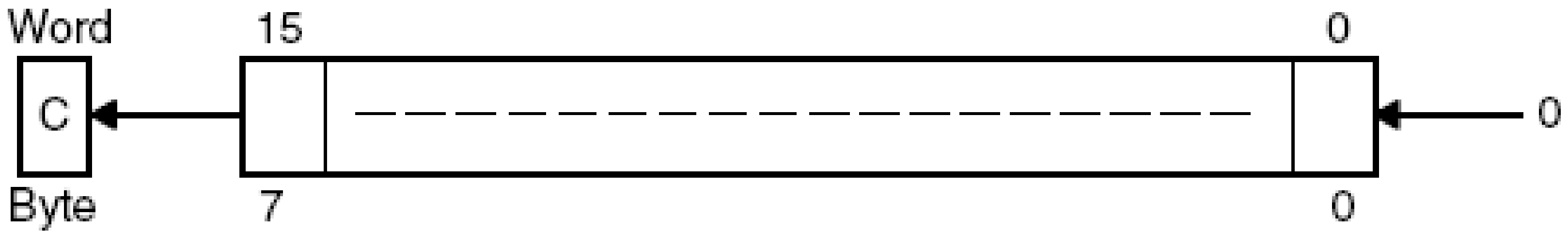
```
RLA     destination or RLA.W destination
RLA.B   destination
```

UBI

Arithmetic and logic operations (24/31)

- **Logic operations - Displacement and rotation with carry (continued):**

  - The `RLA` instruction produces an arithmetic shift left of the destination by inserting a zero in the least significant bit, while the most significant bit is moved out to the carry flag (C).



Arithmetic and logic operations (25/31)

- **Logic operations - Displacement and rotation with carry (continued):**
  - As an example, the registers R5 and R4 are loaded with 0x00A5 and 0xA5A5, respectively, forming a 32-bit value in the structure R5:R4;
  - A shift left performs the multiplication by 2:

```
MOV    #0x00A5,R5   ; load the value 0x00A5 into R5
MOV    #0xA5A5,R4   ; load the value 0xA5A5 into R4

RLA    R5       ; shift most significant word left R5
RLA    R4       ; shift least significant word left R4
ADC    R5       ; add the carry bit of R4 in R5
```

Arithmetic and logic operations (26/31)

- **Logic operations - Displacement and rotation with carry (continued):**

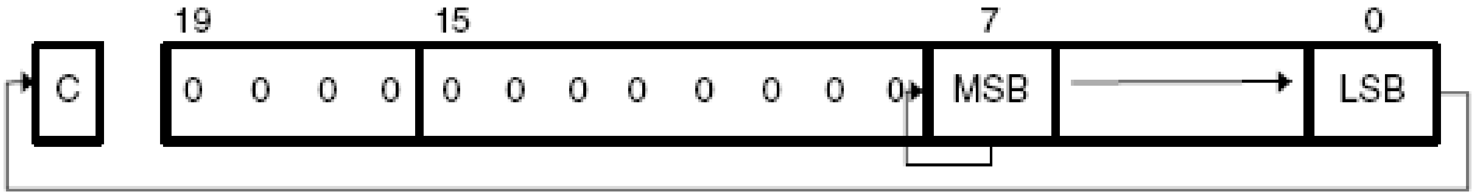
  - The arithmetic shift right is made by the instruction:

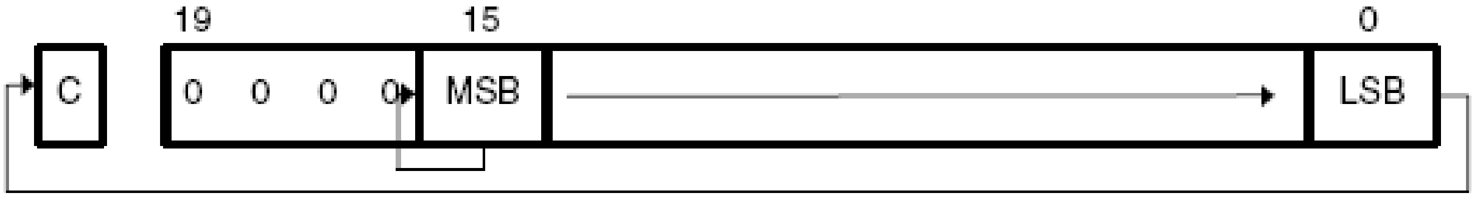
```
RRA    destination or RRA.W destination
RRA.B destination
```

  - The RRA operation produces the arithmetic shift right of the destination, preserving the MSB state, while the least significant bit is copied into the carry flag (C).

Arithmetic and logic operations (27/31)

- **Logic operations - Displacement and rotation with carry (continued):**
  - The arithmetic shift right is made by the instruction:



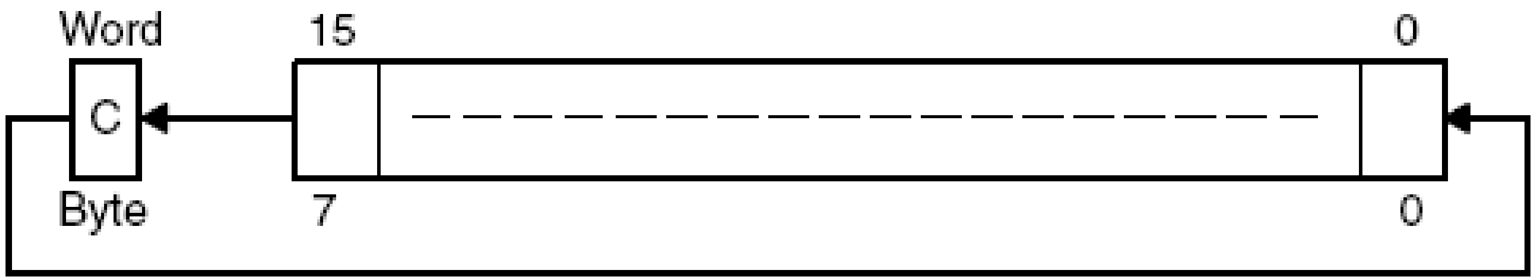


  - The CPU status flags are updated to reflect the operation result.

Arithmetic and logic operations (28/31)

- **Logic operations - Displacement and rotation with carry (continued):**
  - The rotation of a value can be performed using the carry flag;
  - This allows selection of the bit to be rotated into the value;
  - The left or right shift with the carry flag can be performed by the instruction:

```
RLC     destination or RLC.W destination
RLC.B destination
```

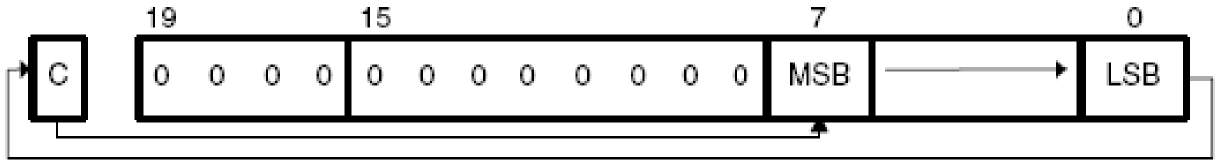


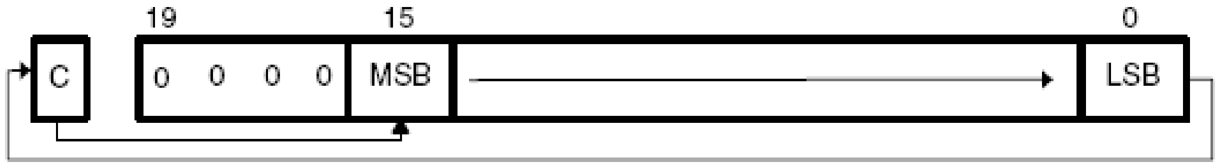
Arithmetic and logic operations (29/31)

- **Logic operations - Displacement and rotation with carry (continued):**
  - The `RLC` operation shifts the destination left, moving the carry flag (C) into the LSB, while the MSB is moved into the carry flag (C).

```
RRC    destination or RRC.W destination
RRC.B destination
```

The `RRC` operation shifts the destination right, moving the carry flag (C) into the MSB, and the LSB is moved to the carry flag (C):





UBI

Arithmetic and logic operations (30/31)

- **Logic operations - Displacement and rotation with carry (continued):**
  - In the following example, the register pair R4:R5 are loaded with 0x0000 and 0xA5A5, respectively, forming a 32-bit value;
  - A shift left of R4:R5 multiplies the value by 2. This example is similar to the previous one, but requires one less instruction:

```
MOV   #0x0000,R4  ; load R4 with #0x0000
MOV   #0xA4A5,R5  ; load R5 with #0xA5A5

RLA   R5   ; Rotate least significant word left
RLC   R4   ; Rotate most significant word left
```

Arithmetic and logic operations (31/31)

- **Logic operations - Bytes exchange:**
  - To swap the destination bytes contents of a 16-bit register, the following instruction is used:

```
SWPB   destination
```

  - The operation has no effect on the state of the CPU flags;
  - In the following example, the bytes of register R5 are exchanged:

```
MOV # 0x1234, R5; move the value 0x1234 to R5
SWPB R5; exchange the LSB with the MSB. R5 = 0x3412
```

  - The above instruction sequence starts by loading the value 0x1234 into the register R5, followed by exchanging the contents of the LSB and the MSB of register R5.

Program flow control (1/16)

- **MSP430 CPU Testing - Bit testing:**

  - Bit testing can be used to control program flow;
  - Hence, the CPU provides an instruction to test the state of individual or multiple bits;
  - The operation performs a logical `&` (`AND`) operation between the source and destination operands;
  - The result is ignored and none of the operands are changed.

Program flow control (2/16)

❑ MSP430 CPU Testing - Bit testing (continued):

- This task is performed by the instruction:

```
BIT source,destination or BIT.W source,destination
BIT.B source,destination
```

- As a result of the operation, the CPU state flags are updated:
  - V: reset;
  - N: set if the MSB of the result is 1, otherwise reset;
  - Z: set if the result is zero, otherwise reset;
  - C: set if the result is not zero, otherwise reset.

Program flow control (3/16)

- **MSP430 CPU Testing - Bit testing (continued):**

  - For example, to test if bit R5.7 is set:

```
MOV    #0x00CC,R5   ; load the value #0x00CC to R5

BIT    #0x0080,R5   ; test R5.7
```

  - The result of the logical `AND` operation is 0x0080, the bit R5.7 is set;

  - In this case, the result modifies the flags (V = 0, N = 0, Z = 0, C = 1).

Program flow control (4/16)

- **MSP430 CPU Testing - Comparison with zero:**

  - Another operation typically used in the monitoring of program loops is the comparison with zero, to determine if a value has decremented to zero;
  - This operation is performed by the emulated instruction:

```
TST source or TST.W source
TST.B source emulated by CMP(.B or .W) #0,dst
```

  - As a result of the operation, the CPU state flags are updated:
    - V: reset;
    - N: set if the MSB of the result is 1, otherwise reset;
    - Z: set if the result is zero, otherwise reset;
    - C: set.

Program flow control (5/16)

❑ **MSP430 CPU Testing - Comparison with zero (continued):**

- For example, to test if register R5 is zero:

```
MOV    #0x00CC,R5   ; move the value #0x00CC to R5
TST    R5           ; test R5
```

- In this case, the comparison of the register R5 with #0x0000 modifies the flags (V = 0, N = 0, Z = 0, C = 1).

Program flow control (6/16)

❑ MSP430 CPU Testing - Values comparison:

- Two operands can be compared using the instruction:

```
CMP source,destination or CMP.W source,destination
CMP.B source,destination ( dst-src )
```

- The comparison result updates the CPU status flags:
  - V: set if an overflow occurs;
  - N: set if the result is negative, otherwise reset (src>dst);
  - Z: set if the result is zero, otherwise reset (src=dst);
  - C: set if there is a carry, otherwise reset (src<= dst).

Program flow control (7/16)

❑ MSP430 CPU Testing - Values comparison (continued):

- In the following example, the contents of register R5 are compared with the contents of register R4:

```
MOV    #0x0012,R5   ; move the value 0x0012 to R5
MOV    #0x0014,R4   ; move the value 0x0014 to R4
CMP    R4,R5
```

- The registers comparison modifies the CPU status flags (V = 0, N = 1, Z = 0, C = 0).

Program flow control (8/16)

- **Program flow branch:**
  - The program flow branch without any constraint is performed by the instruction:

```
 BR     destination
```

  - This instruction is only able to reach addresses in the address space below 64 kB;
  - Several addressing modes can be used e.g. `BR R5`, `BR @R5`, `BR @R5+` etc.

UBI

Program flow control (9/16)

- In addition to the previous instructions it is possible to jump to a destination in the range PC +512 to -511 words using the instruction:

```
    JMP destination
```

- **Conditional jump:**
  - Action can be taken depending on the values of the CPU status flags;
  - Using the result of a previous operation, it is possible to perform jumps in the program flow execution;
  - The new memory address must be in the range +512 to -511 words.

Program flow control (10/16)

- **Conditional jump - Jump if equal (Z = 1):**

```
JEQ destination or JZ destination


label1:
        MOV    0x0100,R5
        MOV    0x0100,R4
        CMP    R4,R5
        JEQ    label1
```

  - The above example compares the contents of registers R4 and R5;

  - As they are equal, the flag Z is set, and therefore, the jump to address `label1` is executed.

Program flow control (11/16)

- **Conditional jump - Jump if different (Z = 0):**

```
JNE destination or JNZ destination


label2:
        MOV     #0x0100,R5
        MOV     #0x0100,R4
        CMP     R4,R5
        JNZ     label2
```

  - The above example compares the contents of registers R4 and R5;

  - As they are equal, the flag Z is set, and therefore, the jump to address `label2` is not executed.

Program flow control (12/16)

- **Conditional jump - Jump if higher or equal (C = 1) – without sign:**

```
JC destination or JHS destination


label3:
          MOV    #0x0100,R5
          BIT    #0x0100,R5
          JC     label3
```

  - The above example tests the state of bit R5.8;

  - As bit 8 is set, the flag C is set, and therefore, the jump to address `label3` is executed.

Program flow control (13/16)

- **Conditional jump - Jump if lower (C = 0) – without sign:**

```
JNC destination or JLO destination


label4:
      MOV   #0x0100,R5
      BIT   #0x0100,R5
      JNC   label4
```

  - The above example tests the state of R5.8;

  - As it is set, the flag C is set, and therefore, the jump to address `label4` is not executed.

Program flow control (14/16)

- **Conditional jump - Jump if higher or equal (N=0 and V=0) or (N = 1 and V = 1) – with sign:**

```
JGE destination


label5:
      MOV    #0x0100,R5
      CMP    #0x0100,R5
      JGE    label5
```

  - The above example compares the contents of register R5 with the constant #0x0100;
  - As they are equal, both flags N and V are reset, and therefore, the jump to address `label5` is executed.

UBI

Program flow control (15/16)

- **Conditional jump - Jump if lower (N = 1 and V = 0) or (N = 0 and V = 1) – with sign**

```
JL destination


label6:
        MOV    #0x0100,R5
        CMP    #0x0100,R5
        JL     label6
```

  - The above example compares the contents of register R5 with the constant #0x0100;

  - As they are equal, both flags N and V are reset, and therefore, the jump to address `label6` is not executed.

Program flow control (16/16)

- **Conditional jump:**
  - To perform a jump if the flag (N = 1) is set use the instruction (Jump if negative):

```
JN destination

label7:
      MOV    #0x0100,R5
      SUB    #0x0100,R5
      JN     label7
```

  - The above example subtracts #0x0100 from the contents of register R5;
  - As they are equal, the flag N is reset, and therefore, the jump to address `label7` is not executed.

Stack pointer management (1/10)

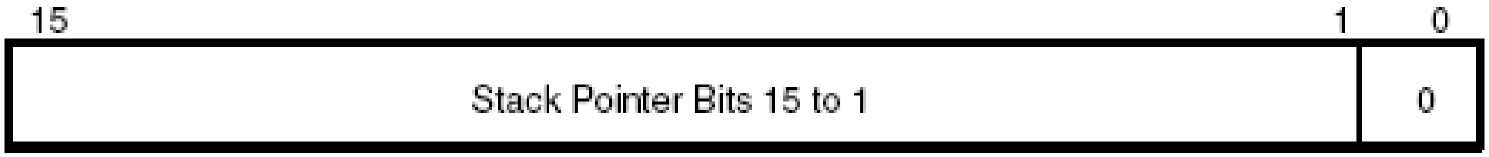
- The SP is used by the CPU to store the return address of routines and interrupts;

- For interrupts, the system status register SR is also saved;

- An automatic method of increment and decrement of the SP is used for each stack access;

- The stack pointer should be initialized by the user to point to an even memory address in the RAM space. This gives the correct word alignment.

- MSP430 CPU stack pointer:



Stack pointer management (2/10)

- **Stack access functions:**

  - The data values are placed on the stack using the instruction:

    ```
    PUSH   source  or  PUSH   source
    PUSH.B source
    ```

  - The contents of the register SP are decremented by two and the contents of the source operand are then placed at the address pointed to by the register SP;

  - In the case of the instruction `PUSH.B`, only the LSB address on the stack is used to place the source operand, while the MSB address pointed to by the register SP+1 remains unchanged.

Stack pointer management (3/10)

❑ Stack access functions (continued):

- In the following example, a byte from register R5 is placed on the stack, followed by a word from register R4;

- The code that performs this task is:

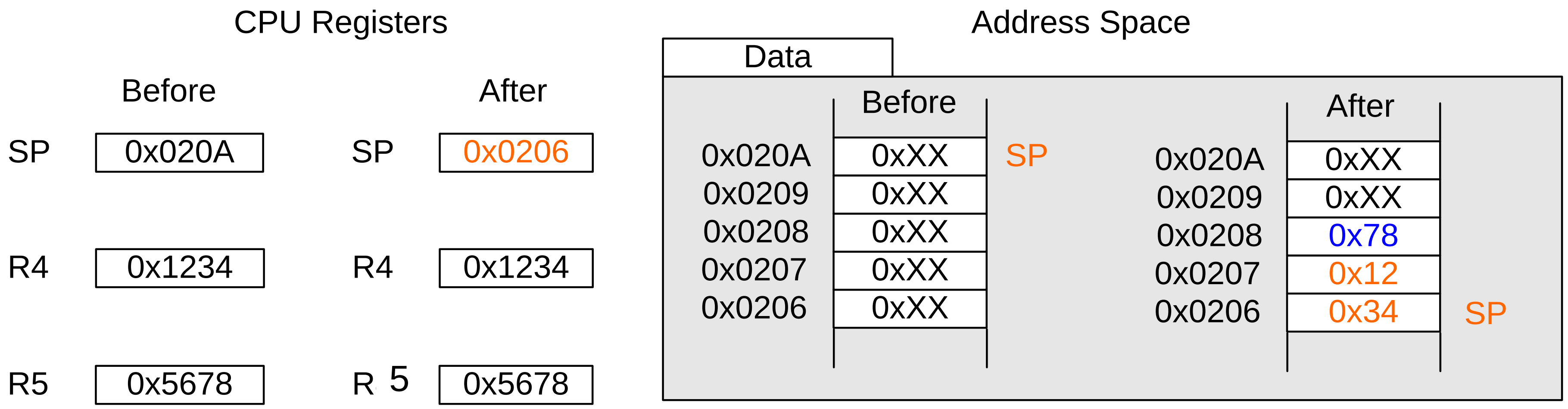
```
PUSH.B   R5 ; move the register R5 LSB to the stack
PUSH     R4 ; move R4 to the stack
```

- The instruction `PUSH.B R5` decrements the register SP by two, pointing to 0x0208;
- The LSB of register R5 is then placed in memory;
- The instruction `PUSH R4` decrements the register SP again by two and places the contents of register R4 on the stack;
- The register SP points to the last element that was placed on the stack.

UBI

Stack pointer management (4/10)

- **Stack access functions (continued):**
  - In the above example, a byte from register R5 is placed on the stack, followed by a word from register R4.



Stack pointer management (5/10)

- **Stack access functions (continued):**
  - A data value is taken off the stack using the instruction:
    ```
    POP   destination  or  POP   destination
    POP.B destination
    ```
  - The contents of the memory address pointed to by the register SP is moved to the destination operand;
  - Then, the contents of the register SP is incremented by two;
  - In the case of an instruction `POP.B`, only the LSB of the address is moved;
  - If the destination is a register, then the other bits are zeroed.

Stack pointer management (6/10)

- **Stack access functions (continued):**

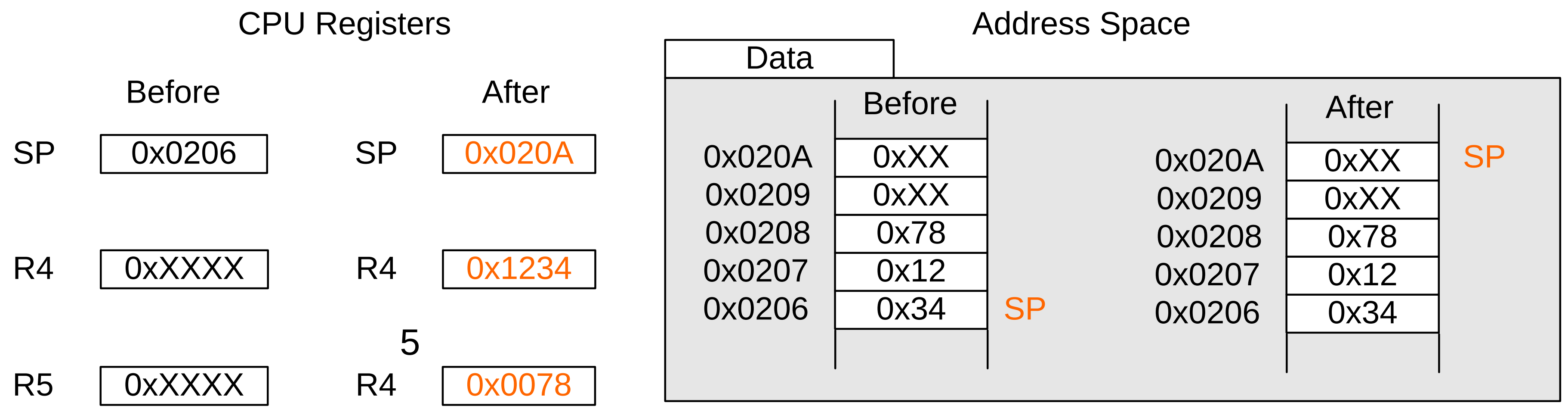
  - The code that performs the task is:

```
POP    R4      ; remove a word from the stack
POP.B  R5      ; remove a byte from the stack
```

  - The instruction `POP R4` extracts the word pointed to by register SP from the stack and puts it in register R4;
  - Then, the stack pointer is incremented by two, to point to memory address 0x0208;
  - The instruction `POP.B R5` moves the byte pointed to by register SP to register R5;
  - The stack pointer is then incremented by two, to point to memory address 0x020A;
  - At the end of the operation, the register SP points to the last element that was placed on the stack, but not yet retrieved.

Stack pointer management (7/10)

- **Stack access functions (continued):**



Stack pointer management (8/10)

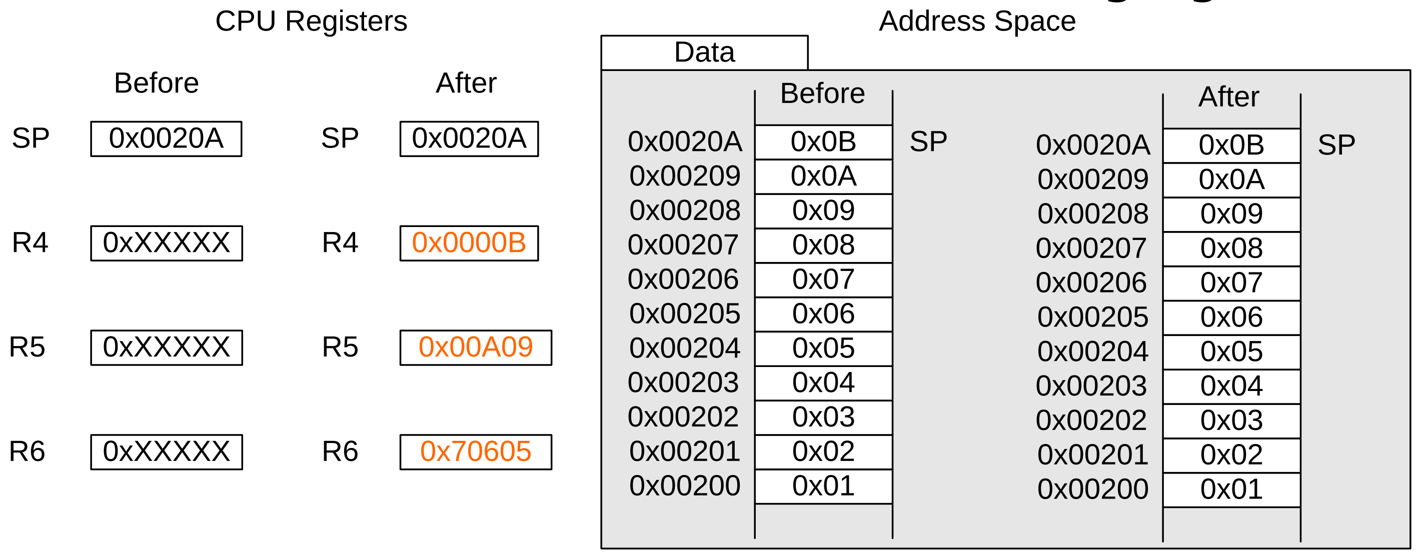
- **Data access on stack with the SP in indexed mode:**
  - The stack contents can be accessed using the register SP in indexed mode;
  - Using this method, it is possible to place and remove data from the stack, without changing the register SP;
  - The MSP430 places the data in the memory space of stack in the Little Endian format;
  - Therefore, the most significant byte is always the highest memory address;

Stack pointer management (9/10)

- **Data access on stack with the SP in indexed mode (cont.):**
  - The code below moves information from the stack to registers, without modifying the register SP:

```
MOV.B  0(SP),R4    ; byte stack access
MOV    -2(SP),R5   ; word stack access
MOVX.A -6(SP),R6   ; address stack access
```

  - The stack structure is shown in the following figure:



CPU Registers

	Before		After
SP	0x0020A	SP	0x0020A
R4	0xXXXXX	R4	0x0000B
R5	0xXXXXX	R5	0x00A09
R6	0xXXXXX	R6	0x70605

Address Space — Data

	Before			After	
0x0020A	0x0B	SP	0x0020A	0x0B	SP
0x00209	0x0A		0x00209	0x0A	
0x00208	0x09		0x00208	0x09	
0x00207	0x08		0x00207	0x08	
0x00206	0x07		0x00206	0x07	
0x00205	0x06		0x00205	0x06	
0x00204	0x05		0x00204	0x05	
0x00203	0x04		0x00203	0x04	
0x00202	0x03		0x00202	0x03	
0x00201	0x02		0x00201	0x02	
0x00200	0x01		0x00200	0x01	

Stack pointer management (10/10)

- **Data access on stack with the SP in indexed mode (cont.):**
  - The first line of code places the value pointed to by register SP in register R4, i.e., the contents of the memory address 0x0020A are moved to register R4;
  - The second line of code moves the word located at SP - 2 = 0x00208 to register R5;
  - And finally, the third line of code moves the contents of the address SP - 6 = 0x00204 to register R6;
  - The entire procedure is performed without modifying the register SP value.

Routines (1/8)

- During the development of an application, repeated tasks are identified and can be separated out into routines;
- This piece of code can then be executed whenever necessary;
- It can substantially reduce the overall code size;
- The use of routines allows structuring of the application;
- It helps code debugging and facilitate understanding of the operation.

Routines (2/8)

- **Invoking a routine:**
  - A routine is identified by a `label` in assembly language;
  - A routine call is made at the point in the program where execution must be changed to perform a task. When the task is complete, it is necessary to return to just after the point where the routine call was made;
  - Two different instructions are available to perform the routine call, namely `CALL`.
  - The following instruction can be used if the routine is located in the address below 64 kB:

```
CALL   destination
```

Routines (3/8)

- **Invoking a routine (continued):**
  - This instruction decrements the register SP by two, to store the return address;
  - The register PC is then loaded with the routine address and the routine is executed;
  - The instruction can be used with any of the addressing modes;
  - The return is performed by the instruction `RET`.

Routines (4/8)

❑ **Routine return:**

- The routine execution return depends on the call type that is used;
- If the routine is called using the instruction `CALL`, the following instruction must be used to return:

```
RET
```

- This instruction extracts the value pointed to by the register SP and places it in the PC;

Routines (5/8)

- **Passing parameters to the routine:**
  - There are two different ways to move data to a routine;
  - The first way makes use of registers:
    - The data values needed for execution of the routine are placed in pre-defined registers before the routine call;
    - The return from execution of the routine can use a similar method.

UBI

Routines (6/8)

- **Passing parameters to the routine (continued):**
  - The second method uses the stack:
    - The parameters necessary to execute the routine are placed on the stack using the `PUSH` instruction;
    - The routine can use any of the methods already discussed to access the information;
    - To return from the routine, the stack can again be used to pass the parameters using a `POP` instruction;
    - Care is needed in the use of this method to avoid stack overflow problems;
    - Generally, the stack pointer must set back to the value just before pushing parameters after execution of the routine.

Routines (7/8)

- **Passing parameters to the routine (continued):**
  - **Routine examples:**
    - The following examples bring together the concepts mentioned in this section;
    - In the first example, the routine is in the address space below 64 kB;
    - Therefore, the instruction `CALL` is used to access the routine;
    - The parameters are passed to the routine through registers.

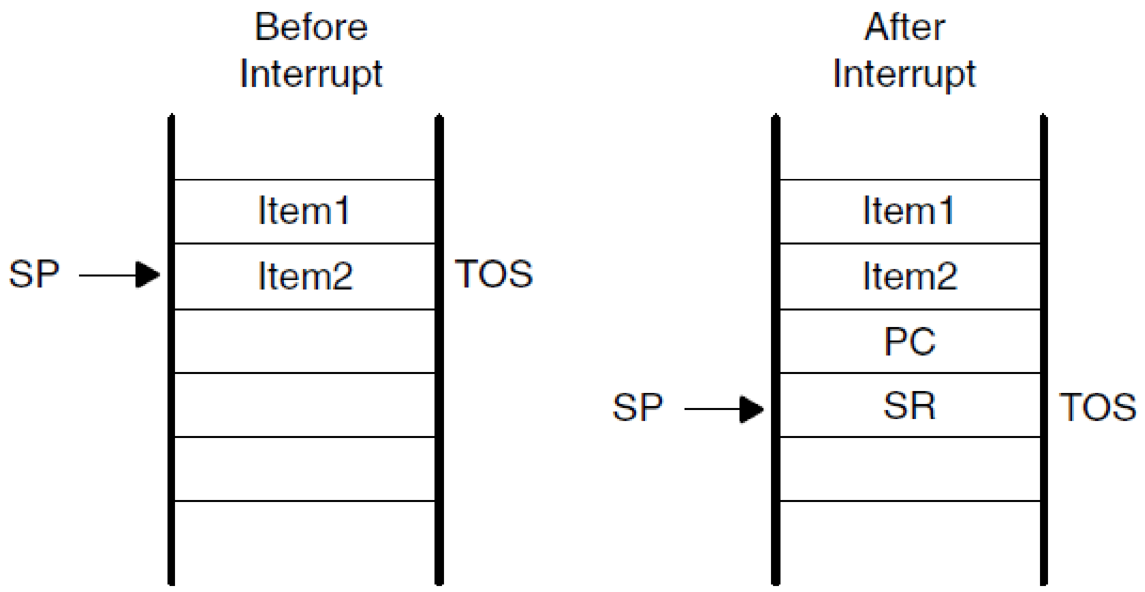
Routines (8/8)

- **Passing parameters to the routine (continued):**
  - **Routine examples – Example 1:**

```
;--------------------
; Routine
;--------------------
adder:
  ADD    R4,R5
  RET    ; return from routine
;--------------------
; Main
;--------------------
  MOV    &var1,R4      ; parameter var1 in R4
  MOV    &var2,R5      ; parameter var2 in R5
  CALL   #adder        ; call routine adder
  MOV    R5,&var3      ; result R5 in var3
```

Interrupts (1/2)

- **Stack management during an interrupt:**
  - During an interrupt, the PC and SR registers are automatically placed in the stack;
  - When the instruction RETI is executed, the PC and SR registers are restored, enabling the return to the program execution point before the interrupt occurred.
  - Interrupt processing for the MSP430 CPU:



UBI

Interrupts (2/2)

❑ **Stack management during an interrupt (continued):**

- An important aspect consists in modifying the low power mode in which the device was before the interrupt;
- As the register SR is restored in the output of the interrupt, its contents stored in the stack can be modified prior to execution of the RETI instruction;
- Thus, an new operation mode will be used. For example, executing the instruction:

```
BIC    #00F0,0(SP) ; clear bits CPUOFF, OSCOFF,
                   ; SCG0 and SCG1
```

- The register SR is loaded in order to remain the device active after ending the interrupt.