MSP430 Assembly Programming

Programming Levels

• Machine language
  – Description of instructions in terms of 0’s and 1’s. This is the model for the actual contents in program memory
  – Normally presented in Hex Notation.
• Assembly language
  – Machine language in human-readable form. Allows better control of program
• High level language
  – Preferred by most programmers because of its English like syntax and notation.
Example: Blinking LED Program

Objective: blink led by toggling of voltage levels

Programming Machine Language

1: 4031 0300
2: 40B2 5A80 0120
3: D3D2 0022
4: E3D2 0021
5: 403F C350
6: 831F
7: 23FE
8: 3FF9
Programming Assembly Language

There is a one-to-one correspondence between machine language and assembly language instructions

MSP430 Machine language structure (1/3)

- Length of One, two or three words:
  - Instruction word
  - Instruction word - Source Info – Destination Info
  - Instruction word – (Source or Dest) Info
- 2 operand instructions with source and destination
- 1 operand instructions with dest or source
- 0 operand instruction: Only one: RETI return from interrupt
- Jumps
MSP430 Machine language structure (2/3)

Two operand instructions (most significant nibble 4 to F)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op-code</td>
<td>S-Reg</td>
<td>Ad</td>
<td>B/W</td>
<td>As</td>
<td>D-Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Single operand instruction (most sign. Nibble 1; reti = 1300)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op-code</td>
<td>B/W</td>
<td>Ad</td>
<td>D/S-Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MSP430 Machine language structure (3/3)

Jumps: Most significant nibble is 2 o3; eight jumps

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op-code</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10-Bit PC Offset</td>
<td></td>
</tr>
</tbody>
</table>

OPCODE = 001

Eight conditions for C: 000 ... 111

Largest jump: +- 2^10 bytes.
Basic Structure of MSP430 Assembly instruction (1/3)

• **Mnemonics** *source, destination*
  – (For 2 operand instructions)

• **Mnemonics** *Operand*
  – (For 1 operand instructions or jumps)

• **Mnemonics**
  – For instructions **reti** or 0 operand emulated instructions

• **.b and .w** appended to mnemonics differentiate operand size. Default .w when none
  – **mov.b** r4, r5;  **mov.w** r4, r5 = **mov** r4, r5

Basic Structure of MSP430 Assembly instruction: jump operands (2/3)

• In interpreters or one-pass assemblers, the jump operand is the number of bytes that must be added to current PC (pointing to next address).
  – Signed 11-bit integer.

• In two-pass assembler, it is an even address or the symbolic label name given to the address.
Basic Structure of MSP430 Assembly instruction: general cases (3/3)

- Operands are written with a syntax associated to the addressing mode
- Operands can be
  - Register names
  - Integers
- Integers can be:
  - Base ten integers (no suffix or prefix)
  - Base 2 (suffix b or B)
  - Base 16 (suffix h or H, or prefix 0x; it cannot begin with a letter)
  - Base 8 (suffix q or Q)

MSP430 Addressing modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Syntax</th>
<th>DATA location</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>#X</td>
<td>Data is X</td>
<td>OK</td>
<td>X</td>
</tr>
<tr>
<td>Register</td>
<td>Rn</td>
<td>Is in Rn</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Direct</td>
<td>X</td>
<td>at address X</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Absolute</td>
<td>&amp;X</td>
<td>at address X</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Indexed</td>
<td>X(Rn)</td>
<td>At address Rn + X</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Indirect</td>
<td>@Rn</td>
<td>At address Rn</td>
<td>OK</td>
<td>X</td>
</tr>
<tr>
<td>Indirect with autoincrement</td>
<td>@Rn+</td>
<td>At address Rn. In addition Rn incremented by data size after execution</td>
<td>OK</td>
<td>X</td>
</tr>
</tbody>
</table>

Notes: Rn is a register name; X is an integer number (valid constant)
ASSEMBLY PROCESS

Assembly process (Two pass)

Source File

*.asm

Assembler

Object file

Other text files
(headers, .asm, etc)

Linker

Execute file

Other object files

Other files of interest

The linker uploads data and execute file in memory, at addresses specified by assembler using directives.
Assembly process (1/2): Source and assembly

- To **assemble** is to convert assembly language to machine language.
- To **disassemble** goes from machine to assembly
- **Source file**: text file containing assembly instructions and directives
  - **Directive**: ‘instruction’ for assembler and linker, not loaded into program memory
  - **Source statement**: each line in the source file
- **Object file**: file with machine language instructions that results from **assembler**
- **Execute file**: machine language file loaded into memory. It results from **linker**

Assembly process (2/2) : Assembler

- Interpreter: assembles one line at a time
  - Line interpreter: only one line
- Assembler: Usually reads the source file
- Two-pass assemblers: allows use of defined constants, labels, macros, comments, etc.
  - Read the source once to decode user defined vocabulary
- Format of source statement in two-pass assembler:
  - [Label] [mnemonics] [operands] [;comment]
Important Facts

- Source statement: A line in the source file
- Source fields (may consist of only one)
  Label  Mnemonics Operand(s) ;Comment
- Label always goes on first column
- Comment always follows a semicolon (;)
- On the first column, always a label, blank, or a comment, nothing else

Directives and instructions

- **Instructions** are for CPU, loaded and executed
  – Always the same, irrespectively of the assembler
- **Directives**: commands for Assembler
  – Serve to manage memory, create macros, define data, etc.
- The last source statement is the directive **END**
- In IAR, traditionally directives are in capital case (END, DB, DW, etc), mnemonic instructions in small case
Important Directives and points

```
14:  ____________
15:  ORG OF800h
16:  /-----------------------------
17:  RESVector: mov #0208h,SP
18:  MOVDPIC: mov #077h,MOVDPIC,R0
19:  mov #0200421, R10
20:  RET: bpl #0100002h, #101A
21:  bge #0100008h, #1011
22:  Mainloop: mov #1, #1011
23:  mov #0100002h, #101OUT
24:  mov #0900, R15
25:  /-----------------------------
26:  dec R15             ; Decrease
27:  jne L1
28:  mov #0100002h, #101OUT
29:  mov #0000, R15
30:  /-----------------------------
31:  Interrupt Vectors
32:  /-----------------------------
33:  ORG OF800h
34:  ResVector
35:  EBI
```

**ORG <address>:**
Tells the linker the address where to start placing the data/instruction that follow.

**DW:** Tells the assembler that the data following is 16-bit size.

**Reset vector:** Address of first instruction to be fetched. (In this ex. Label ResVector)

OTHER GENERAL CONCEPTS
Constant generators registers R3 and R2 in MSP430

• For certain immediate mode source values, behavior is similar to register mode by using a constant generator register.
• Values #0, #1, #2, #1-1 are generated by R3.
• Value #4 is automatically generated by R2
• Absolute value 0 in 0(Rn) is generated by R2
• Change is transparent to user; must be taken into account for timing and instruction size (see ex. 4.4)

Constant generator examples

• Language machine for mov #3, R5 is 4035 0003, two cycles
  – Instruction word 4035:
    • most significant nibble 4 for mov, least significant 5 for R5
    • 03 indicates immediate mode source, data being the number that follows (0003),
• Language machine for mov #2, R5 is 4325
  – 3-2: immediate value with R3 generating #2 in word instruction (0-0-1-0) in nibble 2
• Language machine for mov.b #2,R5 is 4365
  – 3-6: immediate value with R3 generating 2 in byte instruction (0-1-1-0) in nibble 6
Byte size Operand (1/2)

• For memory data, instruction works as required
  – Examples with [0204] = 3A2F, [0306] = ABCD
    • That is, [0204]=2F, [0205]=3A, [0306]=CD, [0307]=AB
  1. mov &0x204, 0x0306 yields [0306] = \textbf{3A2F}
    This was a word size operand
  2. mov.b &0x204,&0x0306 yields [0306] = \textbf{AB2F}
  3. mov.b &0x205, 0x0306 yields [0306] = \textbf{AB3A}
  4. add.b 0X0307,&0205h yields [0204]= \textbf{ABCD}

Byte size operands (2/2)

• Working in register mode:
  – When in source, take LSB only.
  – When in Destination: result goes to LSB, MSB = 00

• Examples with initial R5 = CA50, R6= 2345, [0204]=ABCD
  – mov.b R5, 0x0204 yields [0204] = \textbf{AB50}
  – mov.b R5, R6 yields R6 = \textbf{0050}
  – add.b &0x0205, R5 yields R5 = \textbf{00FB}
INSTRUCTIONS

MSP430 Instruction Set

• 27 **Core** Instructions and 24 **emulated** instructions
• Core instructions are the machine native language instructions, supported by hardware.
• Emulated instructions are “macros” translated automatically by the assembler to an equivalent core instruction
  – Defined to make programming easier to read and write
  – Already standard.
Data transfer instructions

- **mov** *src, dest*  \( (\text{dest} \leftarrow \text{src}) \)
  - \( \text{mov src,dest} = \text{mov.w src, dest}; \text{mov.b src, dest} \)
- **push** *src*  \( (1. \text{SP} \leftarrow \text{SP}-2; 2. \text{(SP)} \leftarrow \text{src}) \)
  - **push.b** only pushes the byte, but moves SP two places.
- **pop** *dest*  \( = \text{mov @} \text{SP}+, \text{dest} \)
  - This is an emulated instruction
  - **pop.b dest** moves only byte, but SP \( \leftarrow \text{SP}+2 \) anyway
- Verify the your understanding by checking the following instructions in your assembler.
  - push #0xABCD, push.b #0xCD, pop r10, pop.b r11
- **swpb** *src*  \( \text{Swaps low and most significant bytes of word src} \)
  - No byte operation allowed

Remarks on Push and Pop (1/3)

- To recover an original item, the number of push operations must equal the number of pop operations
  - In MSP430 and orthogonal mcu’s, we can “simulate” a pop or push adjustment by manipulating SP instead.
    - Example: add #2,SP to get the update of a pop without actually doing a pop.
- When just storing temporarily, for later retrieve, be careful with number and order of stack operations
  - Pop in reverse order
  - **Never start with a pop** because you get garbage.
Some Remarks on push and pop (2/3)

- Correct
  ```
  push R6
  push R5
  ` other instructions
  pop R5
  pop R6
  ```
- ‘Pop’ the last item you ‘pushed’

- Incorrect (be careful)
  ```
  push R6
  push R5
  ` other instructions
  pop R6
  pop R5
  ```
- But you can define macro `swp R5,R6`
  this way! Ha ha

Remarks on stack management

- Push and Pop operations are managed with the stack pointer SP
- Once a data is popped out, it cannot be retrieved with a stack operation
  – Use `mov` for this purpose
- You can work with items in stack the same way you work with memory.
Arithmetic Instructions

*** Affect flags ***

Normal Flag effects for addition and subtraction

- **C** = 0 if not carry; C = 1 if carry
  - For addition: C=1 means a carry is generated
  - For subtraction: C=0 means a borrow is needed;
- **Z** = 0 if destination is not zero, Z=1 if result is zero (cleared)
- **N** = most significant bit of destination
- **V** = 1 if overflow in addition or subtraction, V=0 if not overflow
  - Addition overflow if two signed integers of same sign yields a result of different sign
  - Subtraction overflow if difference between numbers of different sign has the sign of subtrahend
Core Addition and Subtraction Instructions - 1/4 - (.w and .b )

- **add** src, dest -- Add source to destination
  - dest ← dest + src

- **addc** src, dest -- Add source and carry to destination
  - dest ← dest + src + Carry Flag

- **sub** src, dest -- Subtract source from destination
  - dest ← dest + not(src) + 1

Core Addition and Subtraction Instructions - 2/4- (.w and .b )

- **subc** src, dest or **sbb** src, dest-- Subtract source and borrow from destination
  - dest ← dest + not(src) + Carry

- **cmp** src, dest -- Compare dest to src
  - dest + not(src )+ 1, no change of operands.

- **sxt** dest-- Sign-extend LSByte to word
  - Bit(15)=Bit(14)= ... = Bit(8) ← Bit(7)
    - Example R5 = A587, **sxt** R5 → R5 = FF87
    - Example R6 = A577, **sxt** R6 → R6 = 0077
Core Addition and Subtraction (3/4):
Using Compare (cmp)

- **comp** src, dest is usually encoded to compare two numbers A=dest, B=src, in order to take a decision based on their relationship.
- Decision is made with a conditional jump.
- Conditions directly tested in MSP430 are
  - (1) If A=B then…..; (2) If A \neq B then ….  
  - (3) If A \geq B then …. [for signed and unsigned]  
  - (5) If A< B then ..... [for signed and unsigned]  
  - (7) If A <0  then ....

---

Core Addition and Subtraction Instructions - 3/4- (.w and .b )

- **dadd** src, dest  **Decimal addition**
  - Used for BCD formats
  - dest \leftarrow BCD addition ( dest + src + Carry)
  - C =1 if result is greater than 9999 for words or 99 for bytes.
  - V is undefined
- Example:  \text{R5} = 1238   \text{R6} = 7684,   \text{C}=0
  - dadd.w \text{R5}, \text{R6} \rightarrow \text{R6} = 8922,  \text{C}=0
  - dadd.b \text{R5}, \text{R6} \rightarrow \text{R6} = 0022,  \text{C}=1
Emulated arithmetic operations

- \texttt{adc dest} = \texttt{addc #0,dest} (add carry to dest)
- \texttt{dadc dest} = \texttt{dadd #0, dest} (decimal addition of carry)
- \texttt{dec dest} = \texttt{sub #1,dest} (decrement destination)
- \texttt{decd dest} = \texttt{sub #2,dest} (decrement dest twice)
- \texttt{inc dest} = \texttt{add #1,dest} (increment destination)
- \texttt{incd dest} = \texttt{add #2, dest} (increment dest twice)
- \texttt{sbc dest} = \texttt{subc #0, dest} (subtract borrow)
- \texttt{tst dest} = \texttt{cmp #0, dest} (Ojo C=1, V=0 always)

Logic Bitwise Instructions

And MSP430 logic bitwise instructions
Effects on Flags

• Flags are affected by logic operations as follows, under otherwise indicated:
  • Z=1 if destination is cleared
  • Z=0 if at least one bit in destination is set
  • C= Z’ (Flag C is always equal to the inverted value of Z in these operations)
  • N = most significant bit of destination
  • V = 0

Core bitwise Instructions affecting flags (1/2)

1. **and src, dest** realizes  \( dest \leftarrow src \text{ .and. Dest} \)

2. **xor src, dest** realizes  \( dest \leftarrow src \text{ .xor. Dest} \)
   - a) Most common, but not exclusive, use is for inverting (toggle) selected bits, as indicated by the mask (source)
   - b) Problem: show that the sequence xor r5,r6 xor r6,r5 xor r5,r6 has the effect of swapping the contents of registers r5 and r6.

3. **bit src, dest** realizes  \( src \text{ .and. dest} \) but only affects flags
Core bitwise Instructions affecting flags (2/2)

- Examples starting with
  R12= 35AB = 0011 0101 1010 1011
  R15= AB96 = 1010 1011 1001 0110

  and R12,R15  0010 0001 1000 0010→R15=2182   R12= 35AB
  bit R12,R15  → R15 = AB96, R12 = 35AB
  Flags for both cases: C=1, Z=0, N=0, V=0

  and.b R12,R15  1000 0010→R15=0082   R12= 35AB
  bit.b R12, R15  → R15 = AB96, R1=35AB
  Flags for both cases: C=1, Z=0, N=1, V=0

  xor R12,R15  1001 1110 0011 1101→R15=9E3Dh C=1, Z=0, N=1, V=0
  xor.b R12,R15  0011 1101→R15=003Dh C=1, Z=0, N=0, V=0

Remarks on bit instruction in MSP430

- Since C=Z’, either the carry flag C or the zero flag C can be used as information about condition.
- In bit src, dest C=1 (Z=0) means that at least one bit among those tested is not 0.
- In bit #BITn, dest, where BITn is the word where all but the n-th bit are 0, C=tested bit
  - Example R15= 0110 1100 1101 1001 then
    bit #BIT14,R15 yields C=1 = bit 14;
    bit #BIT5,R15 yields C = 0 = bit 5.
Core bitwise Instructions (2)
– not affecting flags –

4. **bis src, dest** realizes \( \text{dest} \leftarrow \text{src} \lor \text{dest} \):
   - Sets bits selected with mask

5. **bic src, dest** realizes \( \text{dest} \leftarrow \text{src}' \land \text{Dest} \):
   - Clear bits selected with masks.

*Examples*

\[
\begin{align*}
\text{R12} &= \text{35ABh} = 0011\ 0101\ 1010\ 1011, \quad [0204] = 28 \quad (28h) \\
\text{R15} &= \text{AB96h} = 1010\ 1011\ 1001\ 0110 \\
\text{bis R12,R15} &\quad 1011\ 1111\ 1011\ 1111 \rightarrow \text{R15= BFBFh} \quad \text{Flags: unchanged} \\
\text{bic R12,R15} &\quad 1000\ 1010\ 0001\ 0100 \rightarrow \text{R15= 8A14h} \quad \text{Flags: unchanged} \\
\text{bis.b R12,R15} &\quad 1011\ 1111 \rightarrow \text{R15= 00Bfh} \quad \text{Flags: unchanged} \\
\text{bic.b R12,R15} &\quad 0001\ 0100 \rightarrow \text{R15= 0014h} \quad \text{Flags: unchanged} \\
\text{bis.b } #77, \&0x204 &\quad (01001101.or 00101000 =01101101) \quad [0204] = 6D
\end{align*}
\]

Emulated Logic Instructions

• Manipulating flags:

\[
\begin{align*}
\text{clc} &= \text{bic } #1, \text{SR} \quad (C \leftarrow 0) \\
\text{clz} &= \text{bic } #2, \text{SR} \quad (Z \leftarrow 0) \\
\text{cln} &= \text{bic } #4, \text{SR} \quad (N \leftarrow 0) \\
\text{setc} &= \text{bis } #1, \text{SR} \quad (C \leftarrow 1) \\
\text{setz} &= \text{bis } #2, \text{SR} \quad (Z \leftarrow 1) \\
\text{setn} &= \text{bis } #4, \text{SR} \quad (N \leftarrow 1)
\end{align*}
\]

• Inverting a destination

\[
\begin{align*}
\text{inv dest} &= \text{xor } #0FFFFh, \text{dest} \\
\text{inv.b dest} &= \text{xor.b } #0FFh, \text{dest} \\
\text{Toggles (inverts) all bits in dest}
\end{align*}
\]
Rolls and Rotates in MSP430

Rolling (Shifting) and Rotating Data bits

• Two core instructions:
  – Right rolling arithmetic:  \texttt{rra dest}
  – Right rotation through Carry:  \texttt{rrc dest}

• Two emulated instructions
  – Left rolling arithmetic:  \texttt{rla dest = add dest,dest}
  – Rotate left through carry:  \texttt{rlc dest = addc dest,dest}

• Roll = Shift
Right arithmetic shift or roll  rra:

Word  \texttt{rra.w\ dest = rra\ dest}

Byte  \texttt{rra.b\ dest}

Arithmetic Interpretation:  \text{sign\ Divide\ by\ two} \quad (\text{dividend} = Q^*2 + r, \ 0 =< 1)

\begin{align*}
R7 &= \texttt{FB0Fh} = 1111101100001111 = (-1265) \\
\text{rra } R7 &= 1111101100001111 \\
\text{New } R7 &= \texttt{FD87h} = > -633 = (-633)*2 +1
\end{align*}

Right Rotation Through Carry  rrc

Example:  \quad C= 0 \quad R5 = 873B = 1000\ 0111\ 0011\ 1011

\begin{align*}
\text{rrc } R5 &= 0100\ 0011\ 1001\ 1101 = 439D \quad C=1
\end{align*}

Some Uses:

\begin{itemize}
  \item To divide by 2  unsigned numbers by first clearing C
    \begin{itemize}
      \item 873Bh = 34619;  \quad 439Dh = 17309 \quad 34619 = 17309x2 + 1
    \end{itemize}
  \item To extract bits from lsb to msb
  \item Think of other possibilities
\end{itemize}
Left rollings

Left rolling:

\[ \text{rla dest} = \text{add dest,dest} \rightarrow \text{multiply by 2 (may need carry)} \]

Other uses: extract bits from msb to lsb

---

Left Rotation Through Carry

- Arithmetic Interpretation: \( 2x + C \)
- Think of other uses
Program Flow Instructions

Do not affect flags

Jumps 1/2 (Core)

- jz/jeq  label or address  (jump if zero/equal)
- jnz/jneq  label or address  (jump if not zero/equal)
- jc/jhe  label or address  (jump if carry/ higher or equal -- for unsigned numbers)
- jnc/jlo  label or address  (jump if not carry/ lower-- for unsigned numbers)
Jumps and subroutine 2/2 (Core)

- `jge` label or address (jump if greater or equal -- for signed numbers)
- `jl` label or address (jump if less -- for signed numbers)
- `jn` label or address (jump if flag N=1)
- `jmp` label or address (unconditional jump)
- `call` dest
  - calls subroutine;
  - dest follows addressing mode conventions
- `reti` : return from interrupt

Emulated

- **Emulated program flow**
  - `br` dest = `mov` dest, PC (Unconditional branch)
  - `ret` = `mov` @SP+, PC (return from interrupt)

- **Miscellaneous:**
  - `dint` = `bic` #8, SR (Disable interrupts)
  - `eint` = `bis` #8, SR (Enable interrupts)
  - `nop` = `mov` R3, R3 (Do nothing; one cycle delay)
Conditional and loop structures

Code structure

IF-THEN

(a) Flow Diagram If_Then Structure

(b) Negative_jump code structure

(c) Positive_jump code structure
IF_ELSE

(a) Flow Diagram IF_ELSE Structure
(b) Negative_jump code structure
(c) Positive_jump code structure

FOR-LOOPS

(a) For counter N to 1, step -1
(b) For Counter = Nmin to Nmax, step X
WHILE LOOP

(a) WHILE <condition is true> DO
(b) Loop Process
(c) END WHILE

Condition True?

Process

Continue

--------
WhileTest: <Instruction for testing parameter>
<Jump_if_False_to Continue>
While_Loop: --------
<Process, changing parameter>
EndWhile: jmp WhileTest
Continue: --------
--------

REPEAT-UNTIL LOOP

(a) REPEAT Loop Process
(b) UNTIL <Stop Condition True>
End UNTIL

Condition for Stop True?

Process

Continue

--------
RepeatLoop: --------
--------
<Process, changing parameter>
RepeatTest: <Instruction for testing parameter>
EndRepeat: jmp If_False to RepeatLoop
Continue: --------
--------
--------
Examples: Delay loop and iteration loop

Example:
```
mov  #delay,R15
DelLoop: dec R15
        jnz DelLoop
```

Example: Extended delay loop
```
mov  #delay,R15
DelLoop: nop
        dec R15
        jnz DelLoop
```