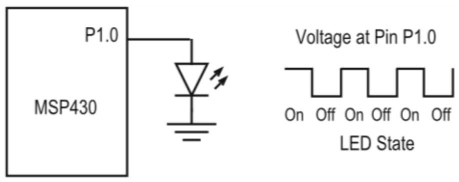
MSP430 Assembly Programming

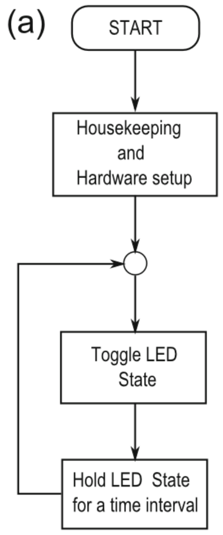
Programming Levels

- 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 syntaxis and notation.

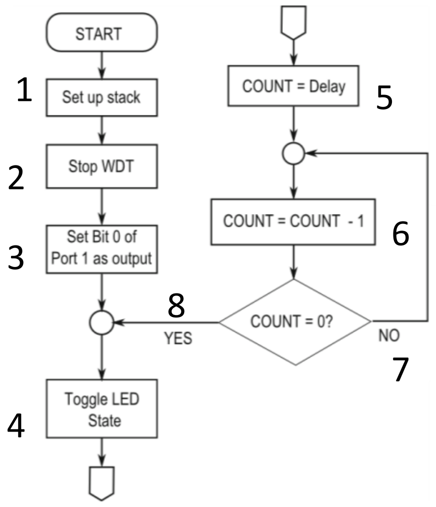
Example: Blinking LED Program

Objective: blink led by toggling of voltage levels




(a)
START
Housekeeping and Hardware setup
Toggle LED State
Hold LED State for a time interval

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

```
mov.w    #0x300,SP          ⟷  4031  0300
mov.w    #0x5A80,&0x0120    ⟷  40B2  5A80  0120
bis.b    #001,&0x0022       ⟷  D3D2  0022
xor.b    #001,&0x0021       ⟷  E3D2  0021
mov.w    #0xC350,R15        ⟷  403F  C350
dec.w    R15                ⟷  831F
jnz      0x3FC              ⟷  23FE
jmp      0x3F2              ⟷  3FF9
```

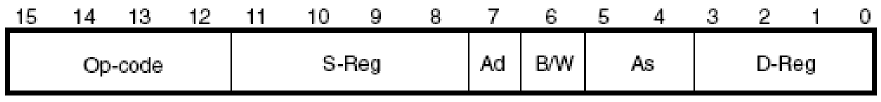
There is a one-to-one correspondence between machine language and assembly language instructions

MSP430 Machine language structure (1/3)

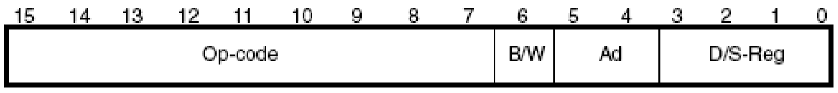
- Length of One, two or three words:
  - **Instruction word**
  - **Instruction word - Source Info – Destination Info**
  - **Instruction word – (Source or Dest) Info**
- 2 operand instructions with source and destination
- 1 operand instructions with dest or source
- 0 operand instruction: Only one: RETI <u>return from interrupt</u>
- JUmps

MSP430 Machine language structure (2/3)

Two operand instructions (most significant nibble 4 to F)

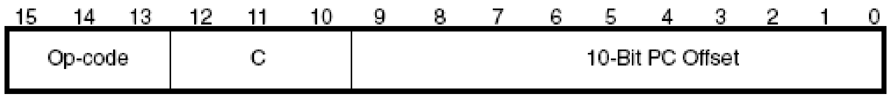


Single operand instruction (most sign. Nibble 1; reti = 1300)



MSP430 Machine language structure (3/3)

Jumps: Most significant nibble is 2 o3 ; eight jumps



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

Non MEMORY LOCATIONS

Mode	Syntax	DATA location	Source	Destination
Immediate	#X	Data is X	OK	X
Register	Rn	Is in Rn	OK	OK

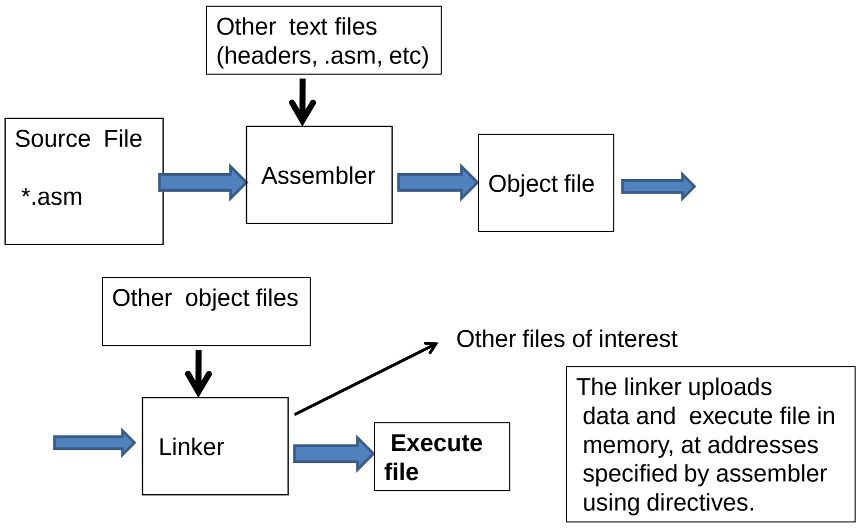
MEMORY LOCATIONS

Direct	X	at address X	OK	OK
Absolute	&X	at address X	OK	OK
Indexed	X(Rn)	At address Rn + X	OK	OK
Indirect	@Rn	At address Rn	OK	X
Indirect with autoincrement	@Rn+	At address Rn. In addition Rn incremented by data size after execution	OK	X

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

ASSEMBLY PROCESS

Assembly process (Two pass)


Other text files (headers, .asm, etc)
Source File
*.asm
Assembler
Object file
Other object files
Other files of interest
Linker
Execute file
The linker uploads data and execute file in memory, at addresses specified by assembler using directives.

Assembly process (1/2): Source and assembly

- To **assembly** is to convert assembly language to machine language.
- To **dissasembly** 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: assemblies one line at a time
  - Line interpreter: only one line
- Assembler: Usually reads the source fi
- Two-pass assemblers: allows use of defined constants, labels, macros, comments, etc.
  - Read the source once to decode user defined vocabulary
- Format of source statetment in two-pass assembler:
  - [Label] [mnemonics] [operands] [;comment]

Important Facts

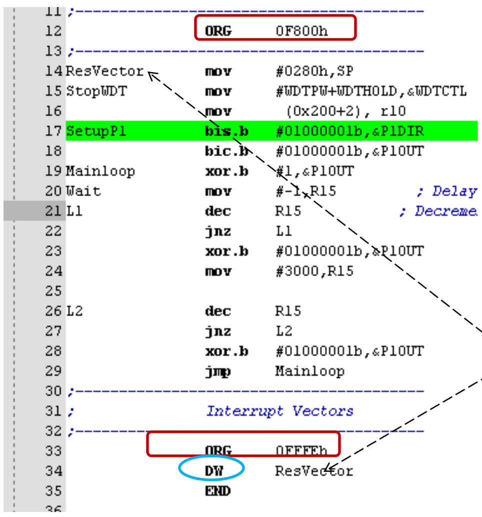
- Source statement: A line in the source file
- Source fields (may consist of only one)

  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



```
;-----------------------------------------
                 ORG      0F800h
;-----------------------------------------
ResVector        mov      #0280h,SP
StopWDT          mov      #WDTPW+WDTHOLD,&WDTCTL
                 mov       (0x200+2), r10
SetupP1          bis.b    #01000001b,&P1DIR
                 bic.b    #01000001b,&P1OUT
Mainloop         xor.b    #1,&P1OUT
Wait             mov      #-1,R15            ; Delay
L1               dec      R15              ; Decreme
                 jnz      L1
                 xor.b    #01000001b,&P1OUT
                 mov      #3000,R15

L2               dec      R15
                 jnz      L2
                 xor.b    #01000001b,&P1OUT
                 jmp      Mainloop
;-----------------------------------------
;                Interrupt Vectors
;-----------------------------------------
                 ORG      0FFFEh
                 DW       ResVector
                 END
```

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 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 significan 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] = **3A2F**

   This was a word size operand
2. mov.b &0x204,&0x0306 yields [0306] = AB**2F**
3. mov.b &0x205, 0x0306 yields [0306] = AB**3A**
4. add.b 0X0307,&0205h yields [0204]= **AB**CD

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] = AB**50**
  - mov.b R5, R6 yields R6 = **0050**
  - add.b &0x0205, R5 yields R5 = **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* (*dest* ← *src*)
  - mov src,dest = mov.w src, dest; mov.b src, dest
- **push** *src* (1. SP← SP-2; 2. (SP) ← src)
  - **push.b** only pushes the byte, but moves SP two places.
- **pop** *dest* **= mov** @SP+, *dest*
  - This is an emulated instruction
  - **pop.b** *dest* moves only byte, but SP← 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* *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
  - <u>Addition overflow</u> if two signed integers of same sign yields a result of different sign
  - <u>Subtraction overflow</u> 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** <u>Decimal addition</u>
  - 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: R5 = 1238 R6 = 7684, C=0
  - dadd.w R5, R6 $\rightarrow$ R6 = 8922, C=0
  - dadd.b R5,R6 $\rightarrow$ R6 = 0022, C=1

Emulated arithmetic operations

- **adc dest** = addc #0,dest (add carry to dest)
- **dadc dest** = dadd #0, dest (decimal addition of carry)
- **dec dest** = sub #1,dest (decrement destination)
- **decd dest** = sub #2,dest (decrement dest twice)
- **inc dest** = add #1,dest (increment destination)
- **incd dest** = add # 2, dest (increment dest twice)
- **sbc dest** = subc #0, dest (subtract borrow)
- **tst dest** = 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* ← *src* .and. *Dest*
2. **xor** ***src, dest*** realizes *dest* ← *src* .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* .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 *dest* ← *src* .or. *dest,* :
    4. Sets bits selectd with mask
5. **bic** ***src, dest*** realizes *dest* ← *src'* .and. *Dest :*
    5. *clear bits selected with masks.*

- *Examples*

R12= 35ABh = 0011 0101 1010 1011, [0204] = 28 (28h)

R15= AB96h = 1010 1011 1001 0110

bis R12,R15 **1011 1111 1011 1111→R15= BFBFh** Flags: unchanged

bic R12,R15 **1000 1010 0001 0100→R15= 8A14h** Flags: unchange

bis.b R12,R15 **1011 1111→R15= 00BFh** Flags: unchanged

bic.b R12,R15 **0001 0100→R15=0014h** Flags: unchanged

bis.b #77, &0x204 (01001101.**or** 00101000 =01101101) **[0204] = 6D**

Emulated Logic Instructions

•<u>Manipulating flags:</u>

clc = bic #1,SR (C← 0)

clz = bic #2,SR (Z← 0)

cln = bic #4,SR (N← 0)

setc = bis #1,SR (C← 1)

setz = bis #2,SR (Z← 1)

setn = bis #4,SR (N← 1)

- <u>Inverting a destination</u>

inv dest = xor #0FFFFh, dest

inv.b dest = xor.b # 0FFh, dest

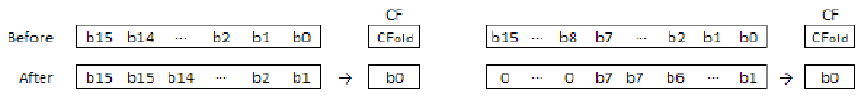
Toggles (inverts) all bits in dest

Rolls and Rotates in MSP430

Rolling (Shifting) and Rotating Data bits

- Two core instructions:
  - Right rolling arithmetic: **rra** ***dest***
  - Right rotation through Carry: **rrc** ***dest***
- Two emulated instructions
  - Left rolling arithmetic: **rla** ***dest*** = **add** ***dest,dest***
  - Rotate left through carry: **rlc** ***dest*** = **addc** ***dest,dest***
- Roll = Shift

Right arithmetic shift or roll rra:



Word rra.w dest = rra dest Byte rra.b dest

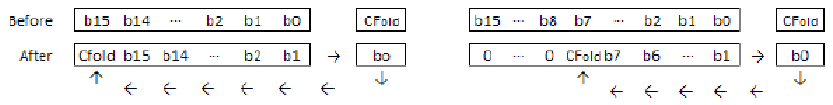
<u>Arithmetic Interpretation</u>: sign Divide by two (dividend = Q*2 + r, 0=< 1

R7 = FB0Fh = <u>1</u>**11110110000111**1 = (-1265)

rra R7 ⟶ <u>1</u>**111110110000111** $\boxed{1}$ C=1 Z=0 N=1 V=0

$\boxed{\text{New R7 = FD87h = > -633} \quad\quad \text{-1265 = (-633)*2 +1}}$

Right Rotation Through Carry rrc



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

rrc r5 → R5 = 0100 0011 1001 1101 = 439D C=1

Some Uses:

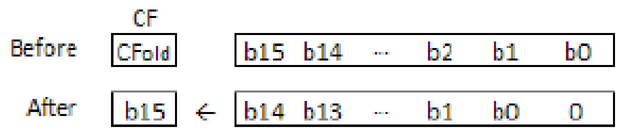
- To divide by 2 unsigned numbers by first clearing C
  - 873Bh = 34619; 439Dh = 17309 34619 = 17309x2 + 1
- To extract bits from lsb to msb
- Think of other possibilities

Left rollings

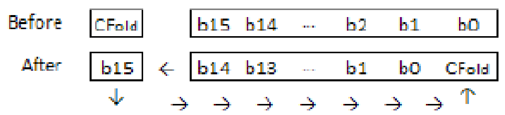
Left rolling:

rla dest = add dest,dest ➔ 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

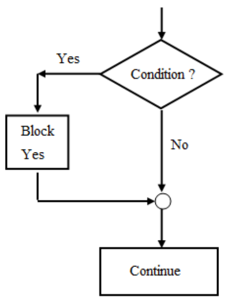
Emulated

- **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

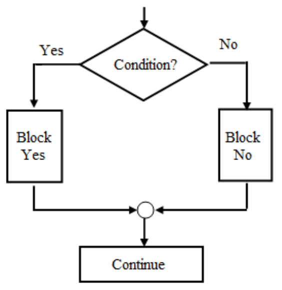
```
Test_Ints:    ------
              jump_if_No  Continue

Block_Yes:    -----
              ------

End_Block:    -------
Continue:     -------
```

(b) Negative_jump code structure

```
Test_Ints:    ------
              jump_if_Yes   Block_Yes
              jmp       Continue

Block_Yes:    -----
              ------

End_Block:    -------
Continue:     -------
```

(c) Positive_Jump code structure

IF_ELSE



(a) Flow Diagram IF_ELSE Structure

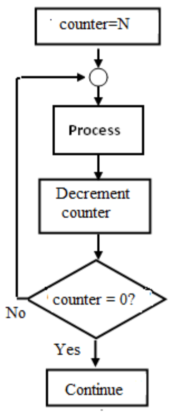
```
Test_Inst:    -----------
                  jump_if_NO  Block_NO

Block_Yes:    ---------
              ---------

End_Yes:      jmp Continue
Block_NO      ----------
              ----------


End_NO        ----------
Continue:     -----------
```

(b) Negative_jump code structure

```
Test_Inst:    -----------
              jump_if_YES  Block_YES
              jump_if_NO   Block_NO
Block_Yes:    ---------
              ---------

End_Yes:      jmp Continue
Block_NO      ----------
              ----------


End_NO        ----------
Continue:     -----------
```

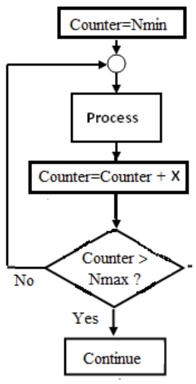
(c) Positive_Jump code structure

FOR-LOOPS



```
              mov   #N,counter
FOR_Loop:     ---------------
              -------------
              dec   counter
End_FOR       jnz   FOR_Loop

Continue:     -------
```

(a) For counter= N to 1, step -1



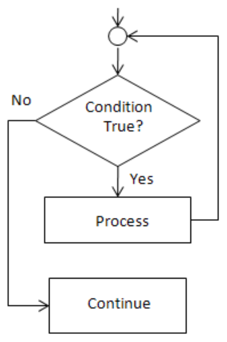
```
              mov  #Nmin, Counter
FOR_Loop:     ------------
              ------------

              add  #X, Counter
              cmp  #Nmax,Counter
 IF_Lower:    jlo   FOR_Loop
IF  Equal:     jeq  FOR_Loop
IF_Greater:    -----------
               ------------
```

(b) For Counter = Nmin to Nmax, step X

WHILE LOOP

```
WHILE <condition is TRUE> DO
        Loop Process
END_WHILE
```

(a)



(b)

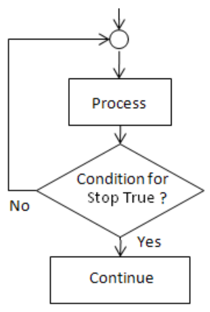
```
                --------------
                --------------
WhileTest:      <Instruction for testing parameter>
                <Jump_if_False  to Continue>
While_Loop:     -------------
                -------------
                <Process, changing parameter>
EndWhile:       jmp  WhileTest

Continue:       -----------
                ------------
                ------------
```

(c)

REPEAT-UNTIL LOOP

```
REPEAT    Loop Process
UNTIL <Stop Condition True>
END UNTIL
```

(a)

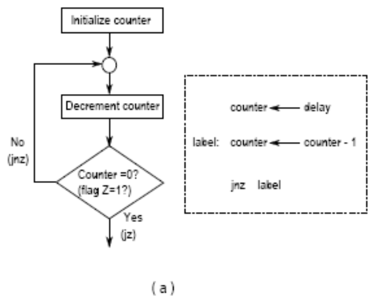


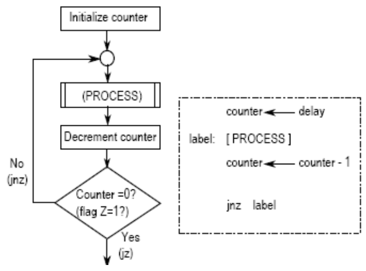
(b)

```
RepeatLoop:     -------------
                -------------
                <Process, changing parameter>
                ---------------
RepeatTest:     <Instruction for testing parameter>
EndRepeat:      <Jump_if_False  to  RepeatLoop>

Continue:       -----------
                -----------
                ------------
```

(c)

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
```