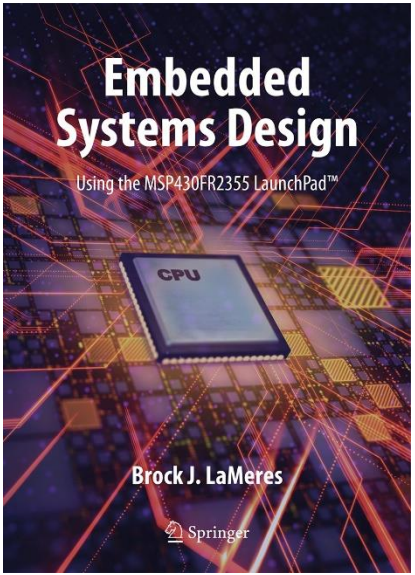
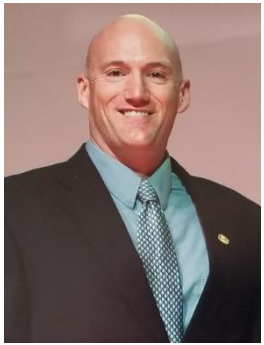
Embedded Systems Design

Chapter 8: Program Flow Instructions

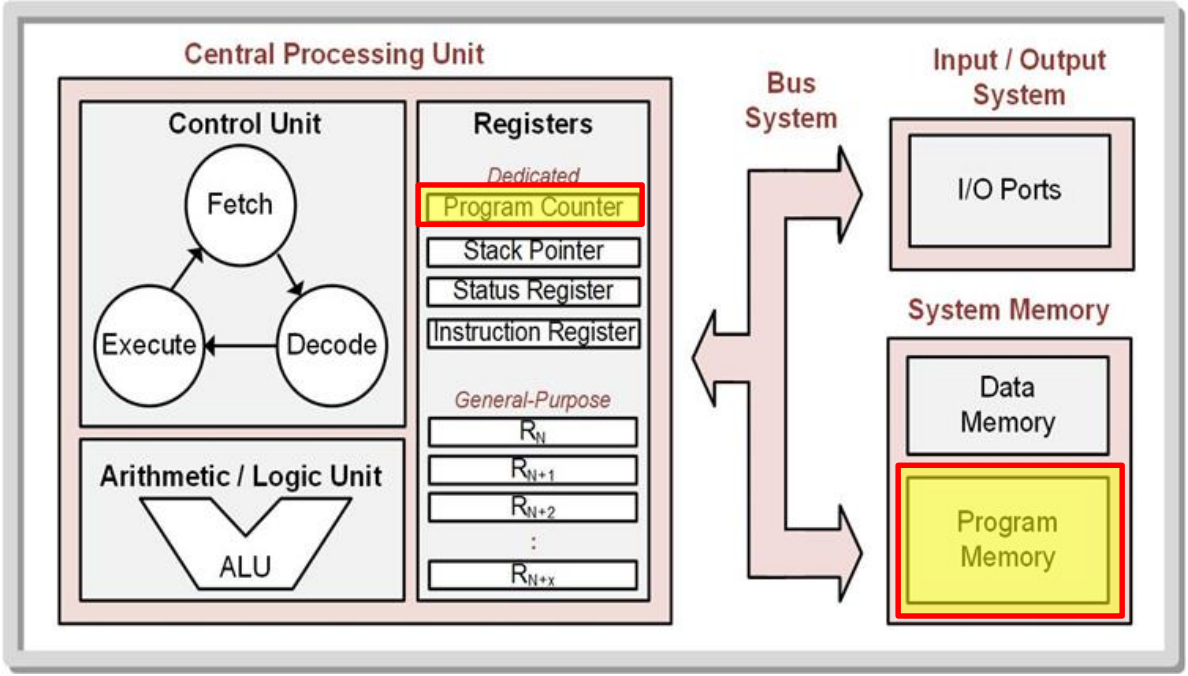
8.1 Unconditional Jumps & Branches - Overview





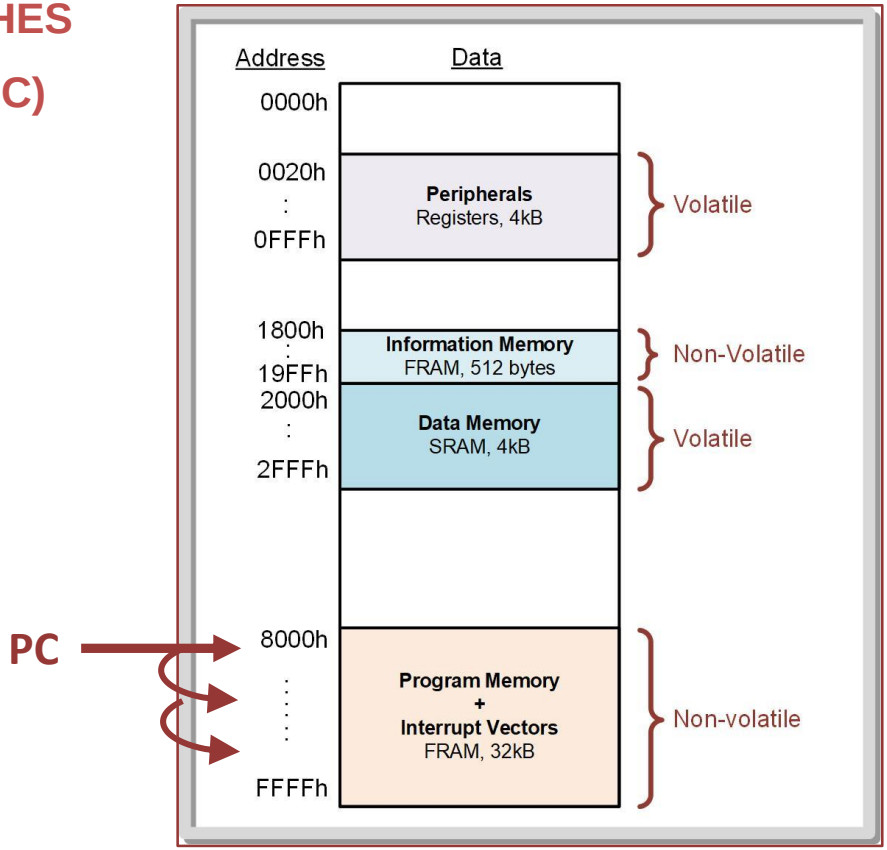
8.1 JUMPS & BRANCHES

- **Program Counter (PC)** – holds address of the next instruction to fetch while executing the current instruction.



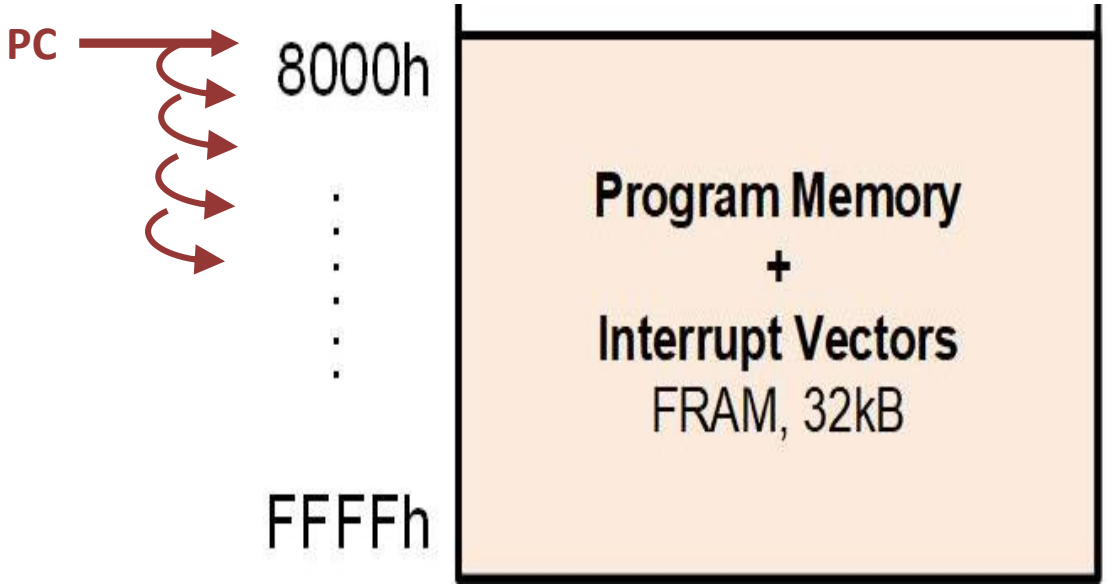
8.1 JUMPS & BRANCHES

- **Program Counter (PC)**



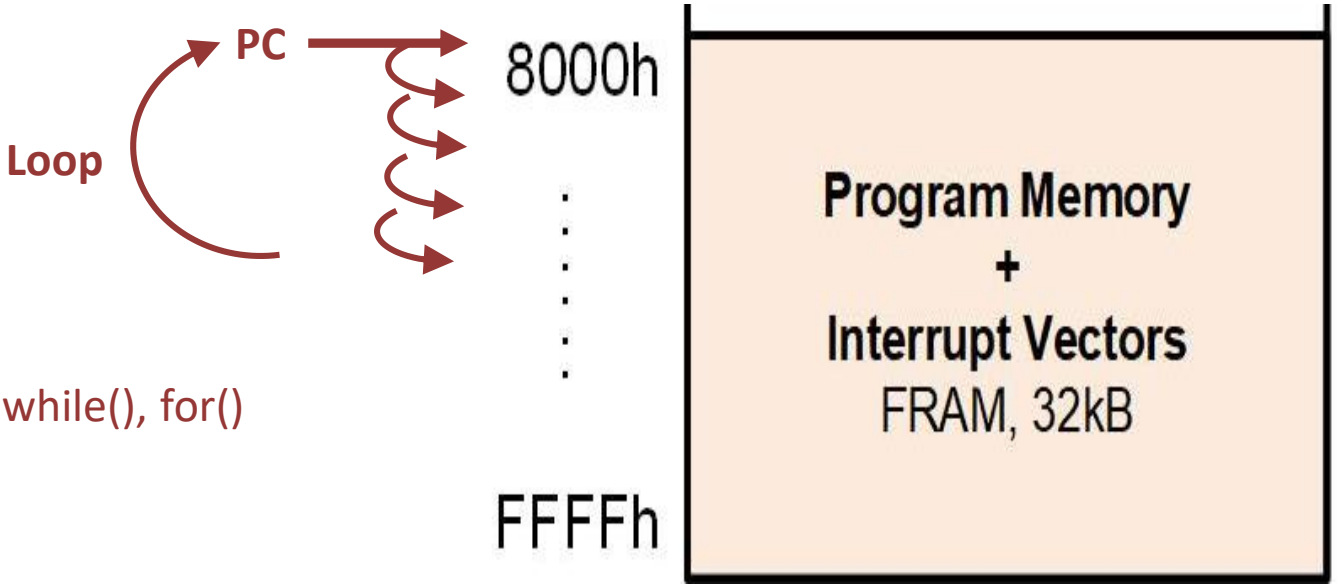
8.1 JUMPS & BRANCHES

- Program Counter (PC)



8.1 JUMPS & BRANCHES

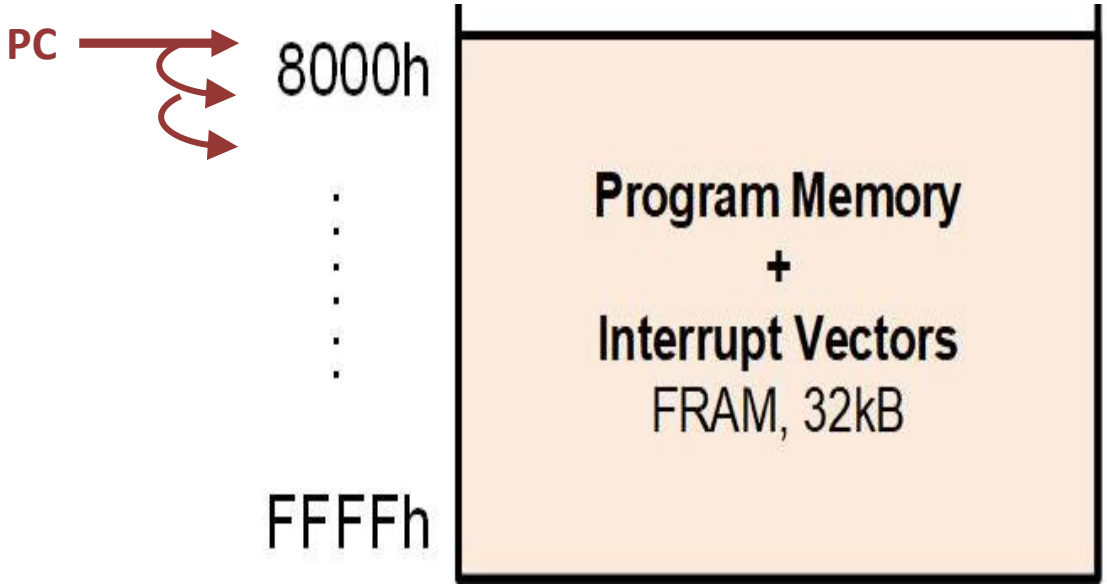
- **Program Counter (PC)**



while(), for()

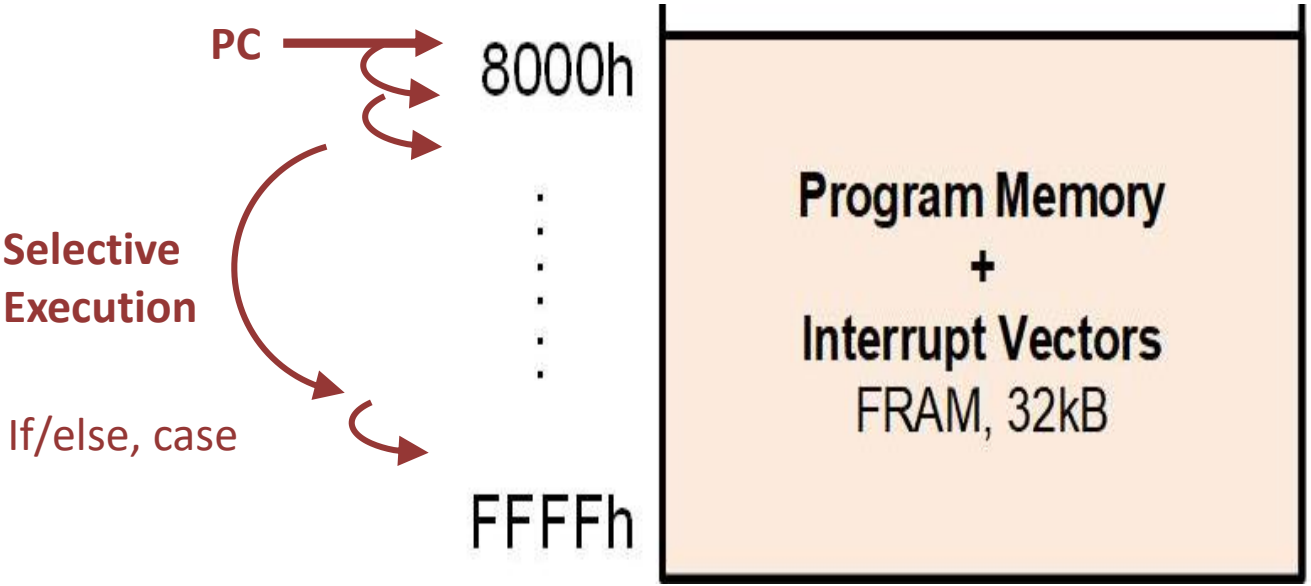
8.1 JUMPS & BRANCHES

- Program Counter (PC)



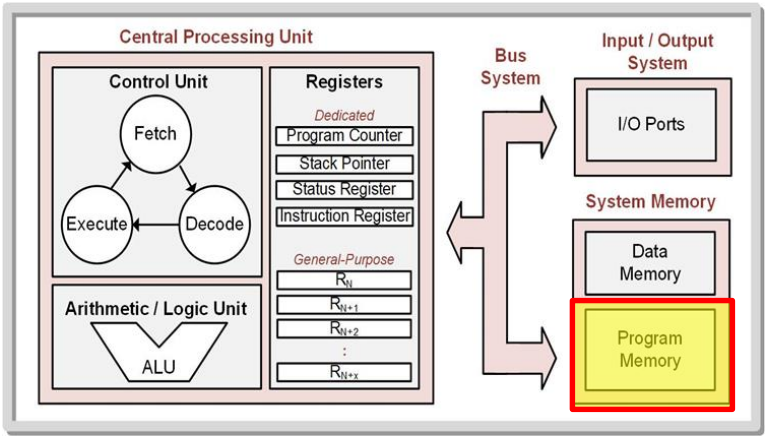
8.1 JUMPS & BRANCHES

- Program Counter (PC)



8.1 JUMPS & BRANCHES

- **Jump/Branch** – instructions that set the PC to a different value other than the next subsequent instruction in memory.
- This allows blocks of instructions to be repeated (a loop) or blocks of code to be selectively executed (if/else).



8.1 JUMPS & BRANCHES

- **Unconditional instructions** – alter the PC when they are executed.

8.1 JUMPS & BRANCHES

- **Unconditional instructions** – alter the PC when they are executed.

- **Conditional instructions** – only alter the PC when certain conditions exist on the VNZC flags in the SR.

8.1 JUMPS & BRANCHES

- **Unconditional instructions** – alter the PC when they are executed.

- **Conditional instructions** – only alter the PC when certain conditions exist on the VNZC flags in the SR.

- **Address labels** are essential to program flow instructions.

8.1 JUMPS & BRANCHES

Unconditional

Program Flow Instructions

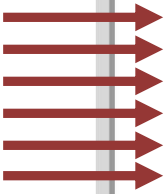
Mnemonic	Operand Format	Description	Behavior	Status Bits V N Z C
jmp	Label	Jump Always	PC updated w/ Label	- - - -
jeq, jz	Label	Jump to label if Z=1	PC updated w/ Label	- - - -
jne, jnz	Label	Jump to label if Z=0	PC updated w/ Label	- - - -
jc	Label	Jump to label if C=1	PC updated w/ Label	- - - -
jnc	Label	Jump to label if C=0	PC updated w/ Label	- - - -
jn	Label	Jump to label if N=0	PC updated w/ Label	- - - -
jge	Label	Jump to label if ≥	PC updated w/ Label	- - - -
jl	Label	Jump to label if <	PC updated w/ Label	- - - -
call	dst	Call subroutine	PC updated w/ dst, PC put on stack	- - - -
ret+		Return from subroutine	return address pulled from stack, put in PC	- - - -
reti		Return from ISR	TOS → SR, SP+2→SP, TOS → PC, SP+2→SP	* * * *
br+	dst	Branch Indirectly to dst	PC updated w/ dst	- - - -
nop		No operation	Copy R3 into R3	- - - -

+ = Emulated instruction.

* = Status bit is affected.
\- = Status bit is not affected.
0 = Status bit is cleared.
1 = Status bit is set.

8.1 JUMPS & BRANCHES

Conditional (VNZC)



Program Flow Instructions

Mnemonic	Operand Format	Description	Behavior	Status Bits V N Z C
jmp	Label	Jump Always	PC updated w/ Label	- - - -
jeq, jz	Label	Jump to label if Z=1	PC updated w/ Label	- - - -
jne, jnz	Label	Jump to label if Z=0	PC updated w/ Label	- - - -
jc	Label	Jump to label if C=1	PC updated w/ Label	- - - -
jnc	Label	Jump to label if C=0	PC updated w/ Label	- - - -
jn	Label	Jump to label if N=0	PC updated w/ Label	- - - -
jge	Label	Jump to label if ≥	PC updated w/ Label	- - - -
jl	Label	Jump to label if <	PC updated w/ Label	- - - -
call	dst	Call subroutine	PC updated w/ dst, PC put on stack	- - - -
ret+		Return from subroutine	return address pulled from stack, put in PC	- - - -
reti		Return from ISR	TOS → SR, SP+2→SP, TOS → PC, SP+2→SP	* * * *
br+	dst	Branch Indirectly to dst	PC updated w/ dst	- - - -
nop		No operation	Copy R3 into R3	- - - -

+ = Emulated instruction.

* = Status bit is affected.
\- = Status bit is not affected.
0 = Status bit is cleared.
1 = Status bit is set.

8.1 JUMPS & BRANCHES

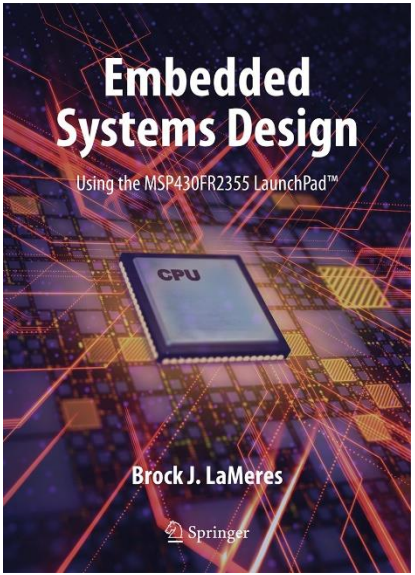
- **Branch** – an unconditional program flow instruction that simply moves the value of the *src* operand into PC.
- One instruction (br)
- Almost always uses immediate mode addressing to specify the address of the label.
- The branch instruction takes three words of program memory.

8.1 JUMPS & BRANCHES

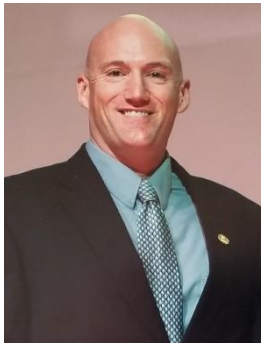
- **Jump** - program flow instruction that either unconditionally or conditionall alters the PC, but does not use the *src* operand as an address directly.
- The **jmp** instruction executes faster because it only takes one word of program memory compared to the three words that **br** takes.
- If the jump offset calculated during assembly happens to be outside of the -511 to +512 range, the assembler will give a “jump out of range” error. A branch will fix this error.
- A jump almost always uses symbolic mode addressing with an addressing with an address label of where to jump.

CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.1 UNCONDITIONAL JUMPS & BRANCHES - OVERVIEW

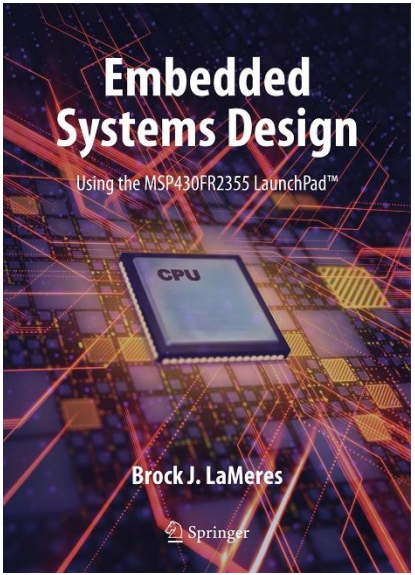


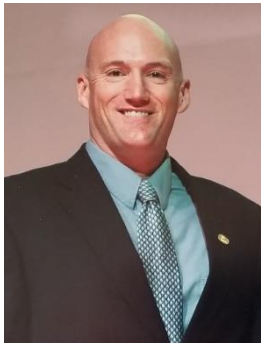
www.youtube.com/c/DigitalLogicProgramming_LaMeres



CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.1 UNCONDITIONAL JUMPS & BRANCHES - EXAMPLE





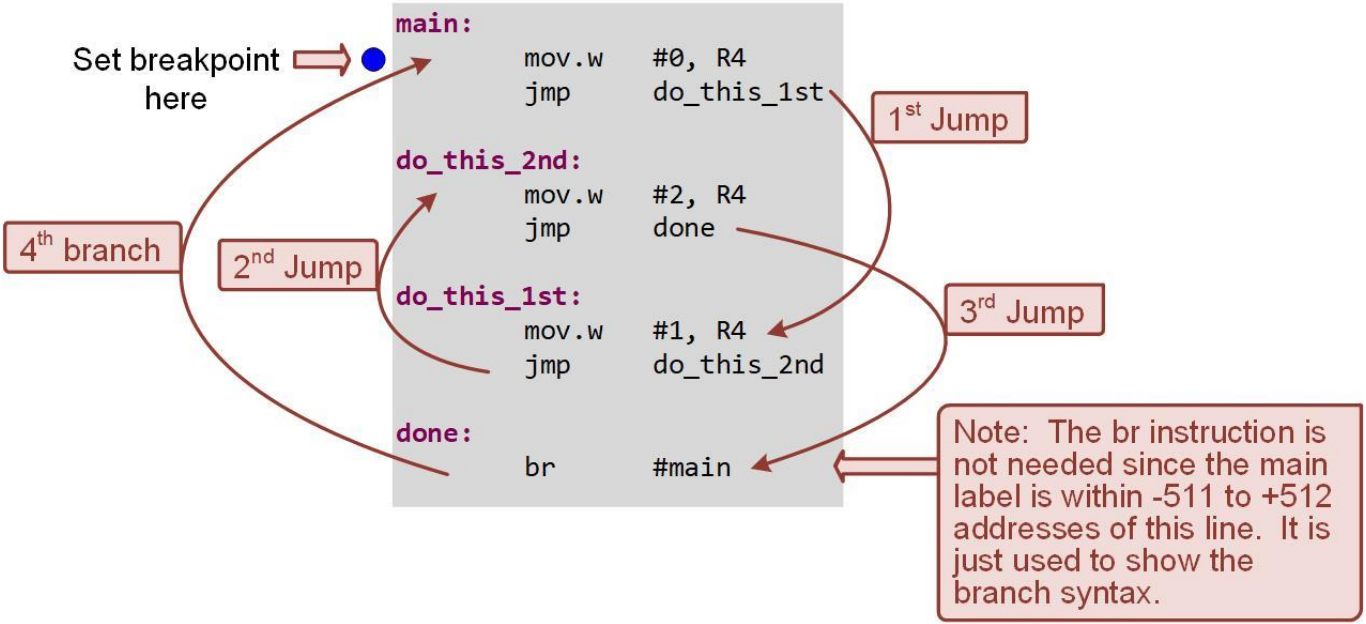
8.1 UNCONDITIONAL BRANCH

- **Branch** – an unconditional program flow instruction that simply moves the value of the *src* operand into PC.
- One instruction (**br**)
- Almost always uses immediate mode addressing to specify the address of the label.
- The branch instruction takes three words of program memory.

8.1 UNCONDITIONAL JUMP

- **Jump** - program flow instruction that always alters the PC, but does not use the *src* operand as an address directly.
- The **jmp** instruction executes faster because it only takes one word of program memory compared to the three words that **br** takes.
- If the jump offset calculated during assembly happens to be outside of the -511 to +512 range, the assembler will give a "jump out of range" error. A branch will fix this error.
- A jump almost always uses symbolic mode addressing with an addressing with an address label of where to jump.

EXAMPLE: USING UNCONDITIONAL JUMPS & BRANCHES



EXAMPLE: USING UNCONDITIONAL JUMPS & BRANCHES

Step 1: Create a new Empty Assembly-only project titled:

Asm_Flow_Jump_n_Branch

Step 2: Type in the following code into the main.asm file where the comments say “Main loop here.”

Image Courtesy of https://neodem.wp.horizon.ac.uk/

EXAMPLE: USING UNCONDITIONAL JUMPS & BRANCHES

Step 3: Debug your program and correct any errors.

Step 4: Set a breakpoint before the first instruction

mov.w #0, R4

Step 5: Open the register viewer and expand the Core Registers item to see the Program Counter.

Step 6: Run your program to the breakpoint.

Step 7: Step your program to observe its operations.

https://neodem.wp.horizon.ac.uk/

EXAMPLE: USING UNCONDITIONAL JUMPS & BRANCHES

Step 6: Run your program to the breakpoint.

Step 7: Step your program to observe its operation.

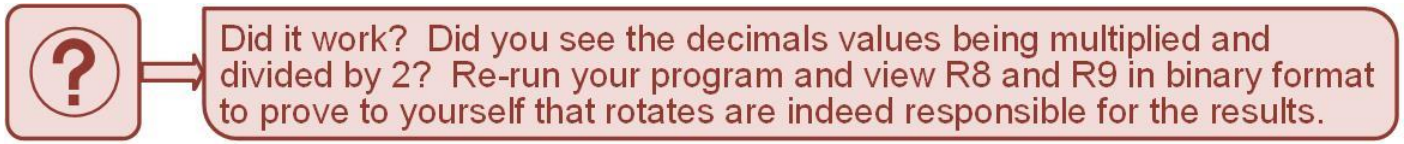
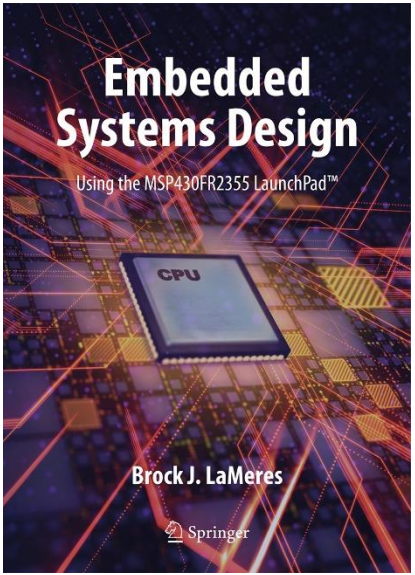


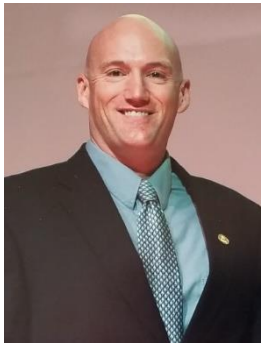
Image Courtesy of https://neodem.wp.horizon.ac.uk/

CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.1 UNCONDITIONAL JUMPS & BRANCHES - EXAMPLE

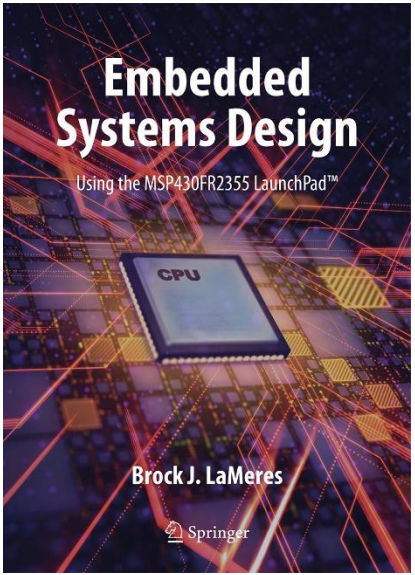


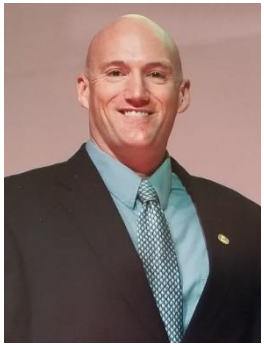
www.youtube.com/c/DigitalLogicProgramming_LaMeres



CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.2 CONDITIONAL JUMPS – CARRY-BASED JUMPS





8.2 CONDITIONAL JUMPS

- **Conditional jumps** – alter the program counter when certain conditions exist in the status flags within the status register.
- **Conditional jump instruction** – alter the program counter if the condition is true.
- If the condition is true, the program counter jumps to a new location in the program.
- If the condition is false, the program counter will simply move onto the next instruction residing in memory.

Image Courtesy of https://neodem.wp.horizon.ac.uk/

8.2.1 CARRY-BASED JUMPS

- **Jump if carry (jc)** – alter the program counter if C = 1, otherwise it will simply move on to the next instruction in memory.
- **Jump if no carry (jnc)** – alter the program counter if C = 0, otherwise it will simply move on to the next instruction in memory.

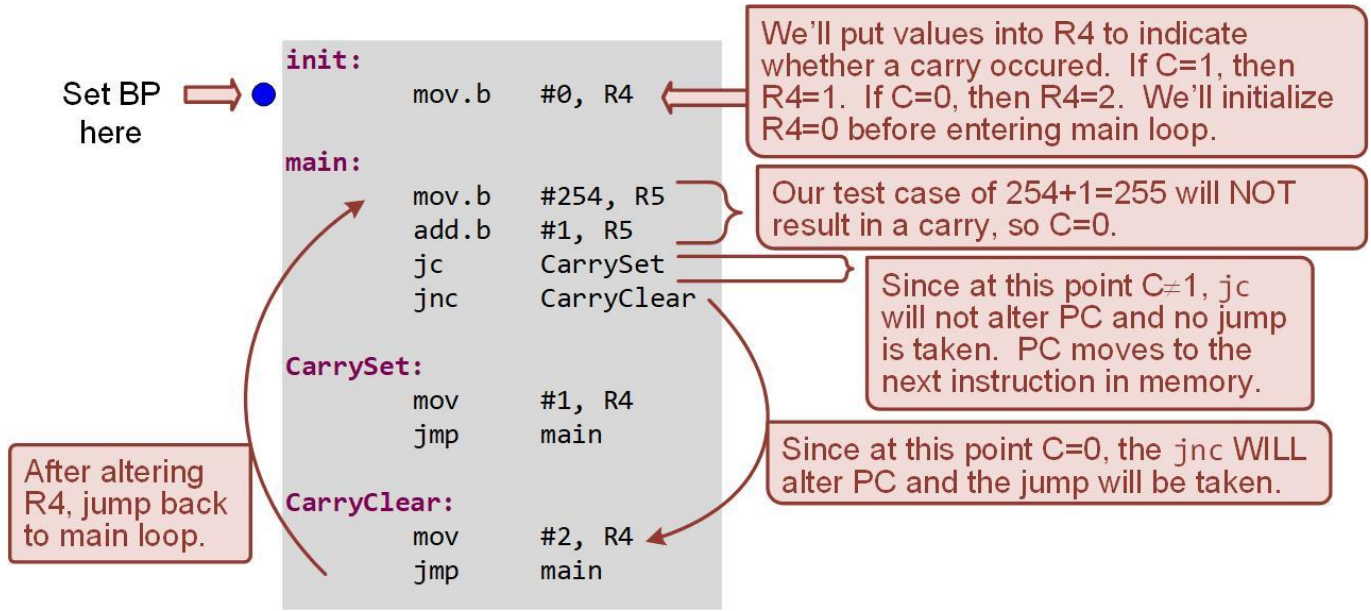
Program Flow Instructions

Mnemonic	Operand Format	Description	Behavior	Status Bits V N Z C
jmp	Label	Jump Always	PC updated w/ Label	- - - -
jeq, jz	Label	Jump to label if Z=1	PC updated w/ Label	- - - -
jne, jnz	Label	Jump to label if Z=0	PC updated w/ Label	- - - -
jc	Label	Jump to label if C=1	PC updated w/ Label	- - - -
jnc	Label	Jump to label if C=0	PC updated w/ Label	- - - -
jn	Label	Jump to label if N=0	PC updated w/ Label	- - - -
jge	Label	Jump to label if ≥	PC updated w/ Label	- - - -
jl	Label	Jump to label if <	PC updated w/ Label	- - - -
call	dst	Call subroutine	PC updated w/ dst, PC put on stack	- - - -
ret+		Return from subroutine	return address pulled from stack, put in PC	- - - -
reti		Return from ISR	TOS → SR, SP+2→SP, TOS → PC, SP+2→SP	* * * *
br+	dst	Branch Indirectly to dst	PC updated w/ dst	- - - -
nop		No operation	Copy R3 into R3	- - - -

+ = Emulated instruction.

* = Status bit is affected.
- = Status bit is not affected.
0 = Status bit is cleared.
1 = Status bit is set.

EXAMPLE: USING JUMPS BASED ON THE CARRY FLAG (JC, JNC)



EXAMPLE: USING JUMPS BASED ON THE CARRY FLAG (JC, JNC)

Step 1: Create a new Empty Assembly-only project titled:

Asm_Flow_Carry_Jumps

Step 2: Type in the following code into the main.asm file where the comments say “Main loop here.”

EXAMPLE: USING JUMPS BASED ON THE CARRY FLAG (JC, JNC)

Step 3: Debug your program and correct any errors.

Step 4: Set a breakpoint before the first instruction

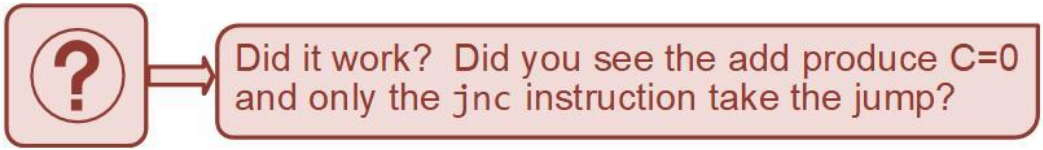
mov.b #0, R4

Step 5: Open the register viewer and expand the Core Registers and the Status Register so you can see the Carry flag, PC, R4, and R5. Change the format of R4 and R5 to decimal.

EXAMPLE: USING JUMPS BASED ON THE CARRY FLAG (JC, JNC)

Step 6: Run your program to the breakpoint.

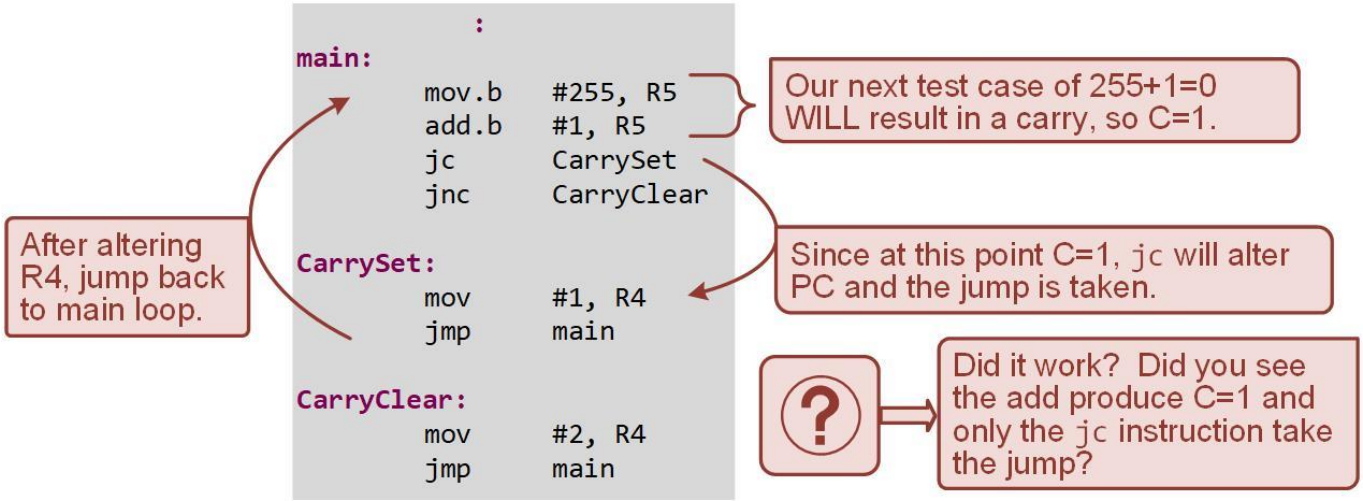
Step 7: Step your program to observe its operation.



Step 8: Now change the src of the second mov instruction from 254 to 255 (**mov.b #255, R5**).

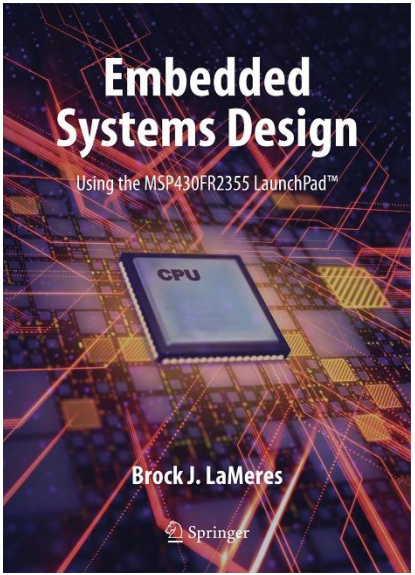
Step 9: Debug the program and observe the behavior of the program.

EXAMPLE: USING JUMPS BASED ON THE CARRY FLAG (JC, JNC)

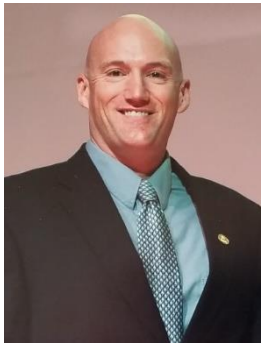


CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.2 CONDITIONAL JUMPS – CARRY-BASED JUMPS

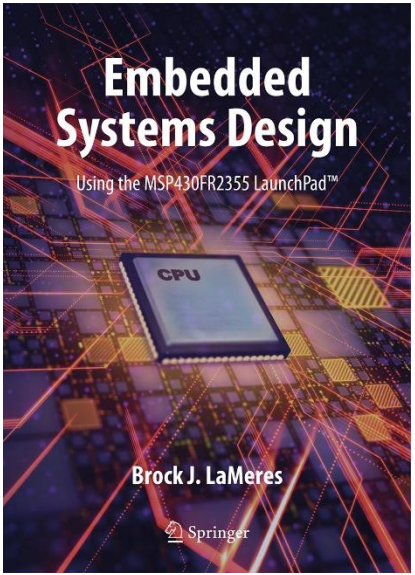


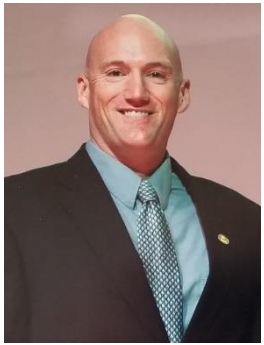
www.youtube.com/c/DigitalLogicProgramming_LaMeres



CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.2 CONDITIONAL JUMPS – ZERO-BASED JUMPS





8.2.2 ZERO-BASED JUMPS

- **Jump if zero (jz)** – alter the program counter if Z = 1, otherwise it will simply move on to the next instruction in memory.
- **Jump if no zero (jnz)** – alter the program counter if Z = 0, otherwise it will simply move on to the next instruction in memory.

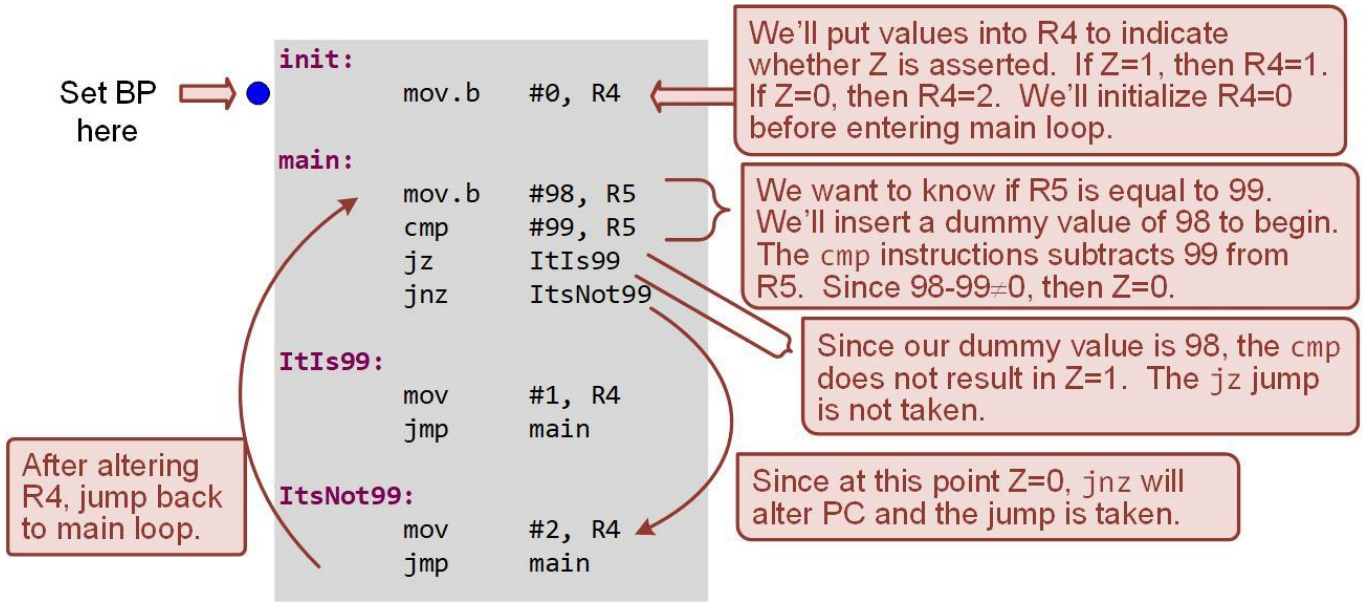
Program Flow Instructions

Mnemonic	Operand Format	Description	Behavior	Status Bits V N Z C
jmp	Label	Jump Always	PC updated w/ Label	- - - -
jeq, jz	Label	Jump to label if Z=1	PC updated w/ Label	- - - -
jne, jnz	Label	Jump to label if Z=0	PC updated w/ Label	- - - -
jc	Label	Jump to label if C=1	PC updated w/ Label	- - - -
jnc	Label	Jump to label if C=0	PC updated w/ Label	- - - -
jn	Label	Jump to label if N=0	PC updated w/ Label	- - - -
jge	Label	Jump to label if ≥	PC updated w/ Label	- - - -
jl	Label	Jump to label if <	PC updated w/ Label	- - - -
call	dst	Call subroutine	PC updated w/ dst, PC put on stack	- - - -
ret+		Return from subroutine	return address pulled from stack, put in PC	- - - -
reti		Return from ISR	TOS → SR, SP+2→SP, TOS → PC, SP+2→SP	* * * *
br+	dst	Branch Indirectly to dst	PC updated w/ dst	- - - -
nop		No operation	Copy R3 into R3	- - - -

+ = Emulated instruction.

* = Status bit is affected.
- = Status bit is not affected.
0 = Status bit is cleared.
1 = Status bit is set.

EXAMPLE: USING JUMPS BASED ON THE ZERO FLAG (JZ, JNZ)



EXAMPLE: USING JUMPS BASED ON THE ZERO FLAG (JZ, JNZ)

Step 1: Create a new Empty Assembly-only project titled:

Asm_Flow_Zero_Jumps

Step 2: Type in the following code into the main.asm file where the comments say “Main loop here.”

EXAMPLE: USING JUMPS BASED ON THE ZERO FLAG (JZ, JNZ)

Step 3: Debug your program and correct any errors.

Step 4: Set a breakpoint before the first instruction

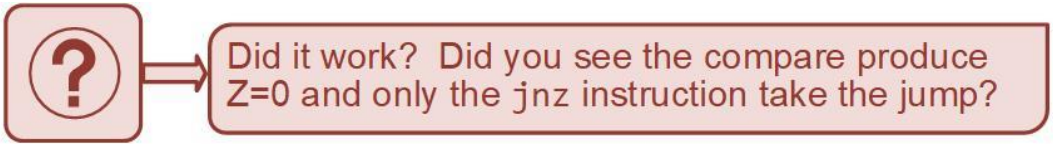
mov.b #0, R4

Step 5: Open the register viewer and expand the Core Registers and the Status Register so you can see the Zero flag, PC, R4, and R5. Change the format of R4 and R5 to decimal.

EXAMPLE: USING JUMPS BASED ON THE ZERO FLAG (JZ, JNZ)

Step 6: Run your program to the breakpoint.

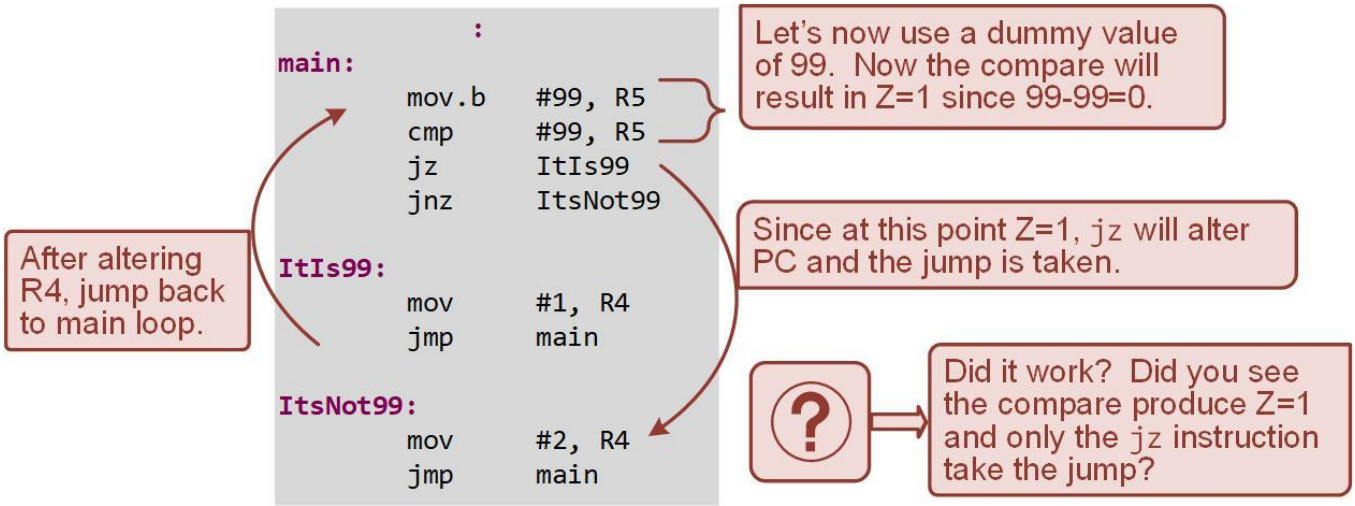
Step 7: Step your program to observe its operation.



Step 8: Now change the src of the second mov instruction from 98 to 99 (**mov.b #99, R5**).

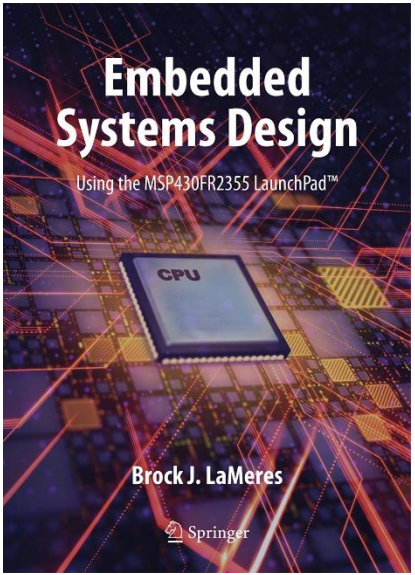
Step 9: Debug the program and observe the behavior of the program.

EXAMPLE: USING JUMPS BASED ON THE ZERO FLAG (JZ, JNZ)

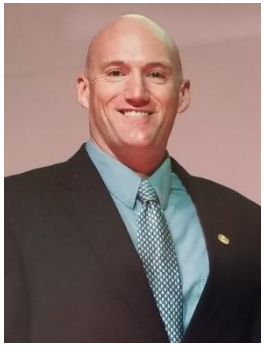


CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.2 CONDITIONAL JUMPS – ZERO-BASED JUMPS

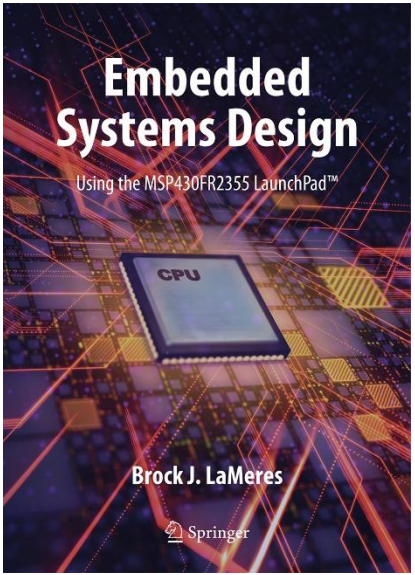


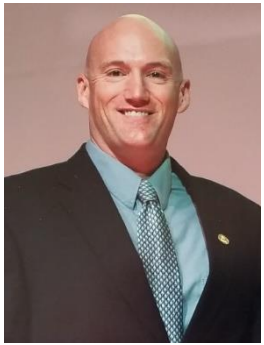
www.youtube.com/c/DigitalLogicProgramming_LaMeres



CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.2 CONDITIONAL JUMPS – NEGATIVE-BASED JUMPS





8.2.3 NEGATIVE-BASED JUMPS

- **Jump if negative (jn)** – alter the program counter if N = 1, otherwise it will simply move on to the next instruction in memory.

- There is no j**ump if not negative** instruction in the MSP430 instruction set; however, this condition can be created using the logic that if the result is not negative, it must be positive.

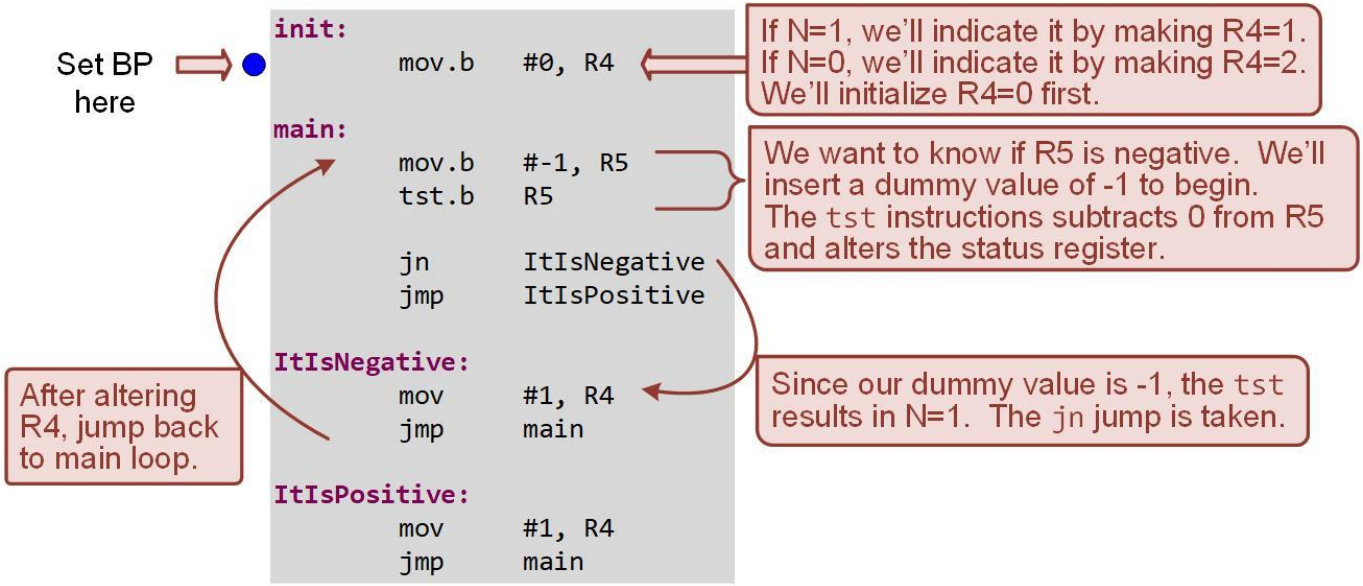
Program Flow Instructions

Mnemonic	Operand Format	Description	Behavior	Status Bits V N Z C
jmp	Label	Jump Always	PC updated w/ Label	- - - -
jeq, jz	Label	Jump to label if Z=1	PC updated w/ Label	- - - -
jne, jnz	Label	Jump to label if Z=0	PC updated w/ Label	- - - -
jc	Label	Jump to label if C=1	PC updated w/ Label	- - - -
jnc	Label	Jump to label if C=0	PC updated w/ Label	- - - -
jn	Label	Jump to label if N=0	PC updated w/ Label	- - - -
jge	Label	Jump to label if ≥	PC updated w/ Label	- - - -
jl	Label	Jump to label if <	PC updated w/ Label	- - - -
call	dst	Call subroutine	PC updated w/ dst, PC put on stack	- - - -
ret+		Return from subroutine	return address pulled from stack, put in PC	- - - -
reti		Return from ISR	TOS → SR, SP+2→SP, TOS → PC, SP+2→SP	* * * *
br+	dst	Branch Indirectly to dst	PC updated w/ dst	- - - -
nop		No operation	Copy R3 into R3	- - - -

+ = Emulated instruction.

* = Status bit is affected.
- = Status bit is not affected.
0 = Status bit is cleared.
1 = Status bit is set.

EXAMPLE: USING JUMPS BASED ON THE NEGATIVE FLAG (JN)



EXAMPLE: USING JUMPS BASED ON THE NEGATIVE FLAG (JN)

Step 1: Create a new Empty Assembly-only project titled:

Asm_Flow_Negative_Jumps

Step 2: Type in the following code into the main.asm file where the comments say “Main loop here.”

EXAMPLE: USING JUMPS BASED ON THE NEGATIVE FLAG (JN)

Step 3: Debug your program and correct any errors.

Step 4: Set a breakpoint before the first instruction

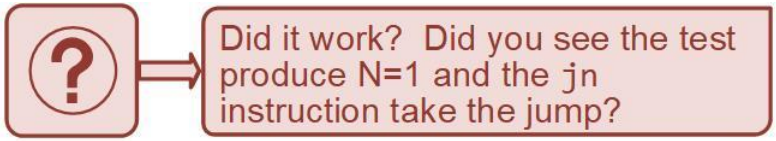
mov.b #0, R4

Step 5: Open the register viewer and expand the Core Registers and the Status Register so you can see the Negative flag, PC, R4, and R5. Change the format of R4 and R5 to decimal.

EXAMPLE: USING JUMPS BASED ON THE NEGATIVE FLAG (JN)

Step 6: Run your program to the breakpoint.

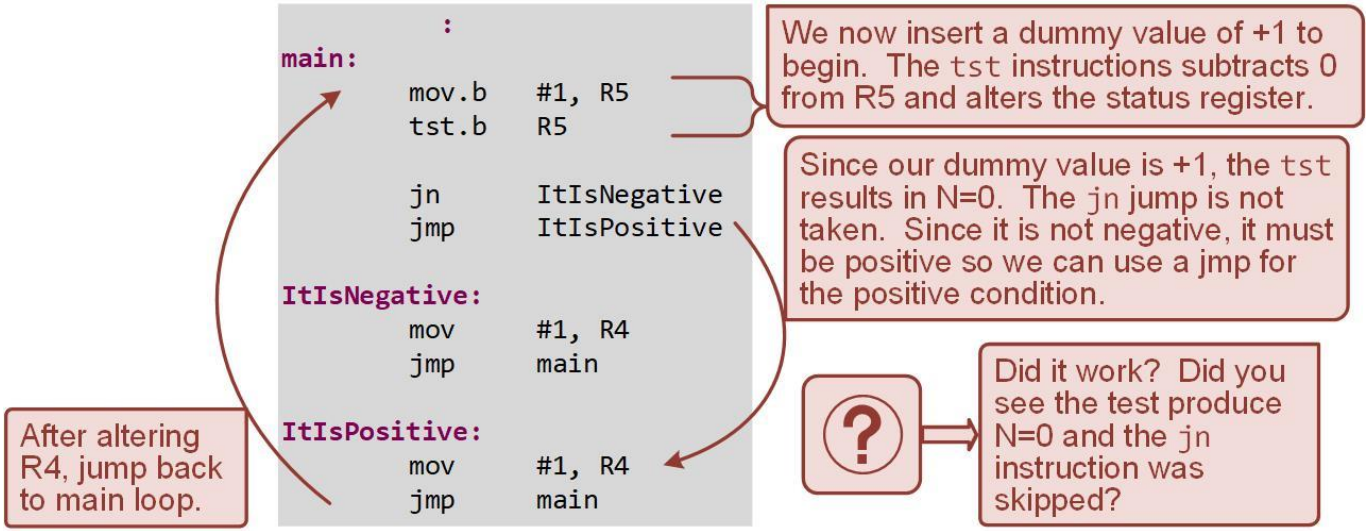
Step 7: Step your program to observe its operation.



Step 8: Now change the src of the second mov instruction from -1 to 1 (**mov.b #1, R5**).

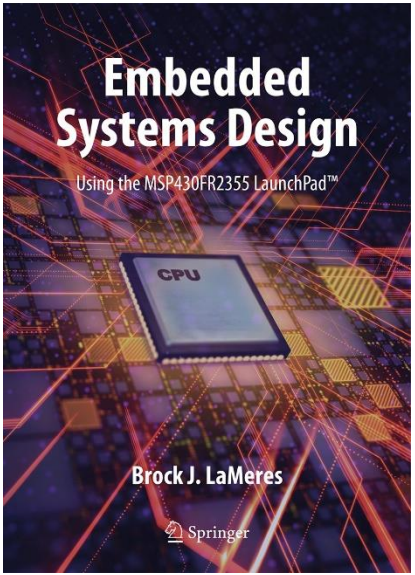
Step 9: Debug the program and observe the behavior of the program.

EXAMPLE: USING JUMPS BASED ON THE NEGATIVE FLAG (JN)

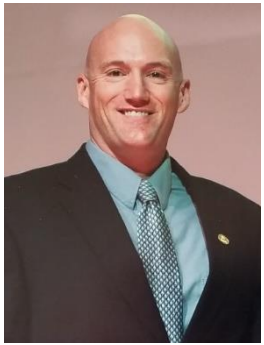


CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.2 CONDITIONAL JUMPS – NEGATIVE-BASED JUMPS

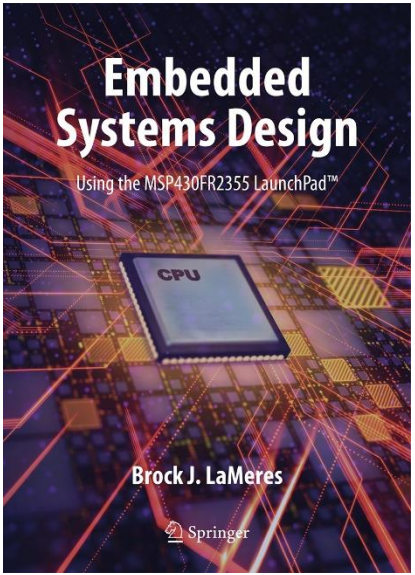


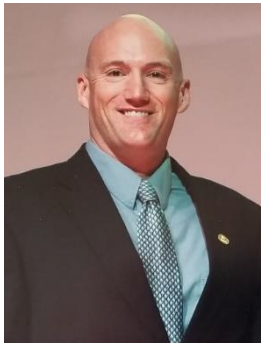
www.youtube.com/c/DigitalLogicProgramming_LaMeres



CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.2 CONDITIONAL JUMPS – OVERFLOW-BASED JUMPS





8.2.4 OVERFLOW-BASED JUMPS

Program Flow Instructions

Mnemonic	Operand Format	Description	Behavior	Status Bits V N Z C
jmp	Label	Jump Always	PC updated w/ Label	- - - -
jeq, jz	Label	Jump to label if Z=1	PC updated w/ Label	- - - -
jne, jnz	Label	Jump to label if Z=0	PC updated w/ Label	- - - -
jc	Label	Jump to label if C=1	PC updated w/ Label	- - - -
jnc	Label	Jump to label if C=0	PC updated w/ Label	- - - -
jn	Label	Jump to label if N=0	PC updated w/ Label	- - - -
jge	Label	Jump to label if ≥	PC updated w/ Label	- - - -
jl	Label	Jump to label if <	PC updated w/ Label	- - - -
call	dst	Call subroutine	PC updated w/ dst, PC put on stack	- - - -
ret+		Return from subroutine	return address pulled from stack, put in PC	- - - -
reti		Return from ISR	TOS → SR, SP+2→SP, TOS → PC, SP+2→SP	* * * *
br+	dst	Branch Indirectly to dst	PC updated w/ dst	- - - -
nop		No operation	Copy R3 into R3	- - - -

+ = Emulated instruction.

* = Status bit is affected.
\- = Status bit is not affected.
0 = Status bit is cleared.
1 = Status bit is set.

8.2.4 OVERFLOW-BASED JUMPS

- **Jump if greater than or equal (jge)** – provide the ability to jump based on inequalities and also to consider two's compliment overflow. Jumps when

  N xor V = 0

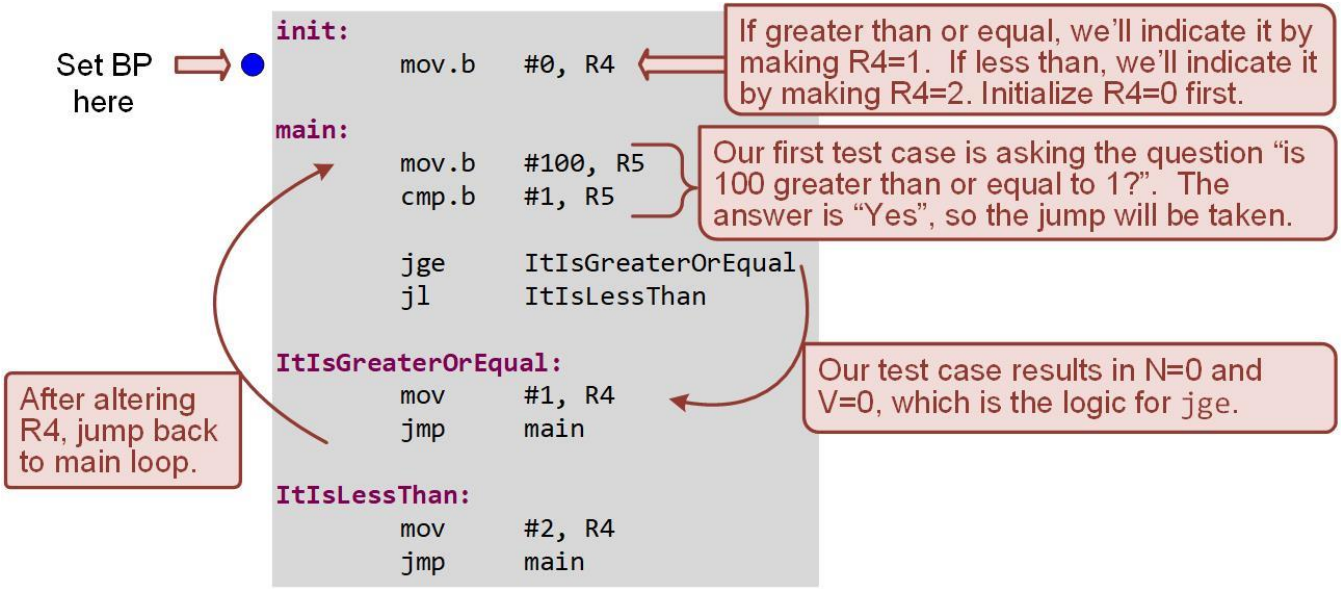
- These jumps use both the N-flag and V-flag and assume the results are signed numbers.

- **jump if less than (jl)** - provide the ability to jump based on inequalities and also to consider two's compliment overflow. Jumps when

  N xor V = 1

- Both instructions are easier to understand by simply using their mnemonic description (> and <).

EXAMPLE: USING JUMPS BASED ON INEQUALITIES (JGE, JL)



EXAMPLE: USING JUMPS BASED ON INEQUALITIES (JGE, JL)

Step 1: Create a new Empty Assembly-only project titled:

Asm_Flow_Inequality_Jumps

Step 2: Type in the following code into the main.asm file where the comments say “Main loop here.”

EXAMPLE: USING JUMPS BASED ON INEQUALITIES (JGE, JL)

Step 3: Debug your program and correct any errors.

Step 4: Set a breakpoint before the first instruction

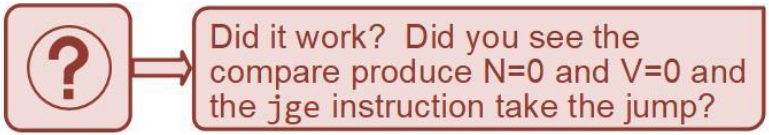
mov.b #0, R4

Step 5: Open the register viewer and expand the Core Registers and the Status Register so you can see the N and V flags, PC, R4, and R5. Change the format of R4 and R5 to decimal.

EXAMPLE: USING JUMPS BASED ON INEQUALITIES (JGE, JL)

Step 6: Run your program to the breakpoint.

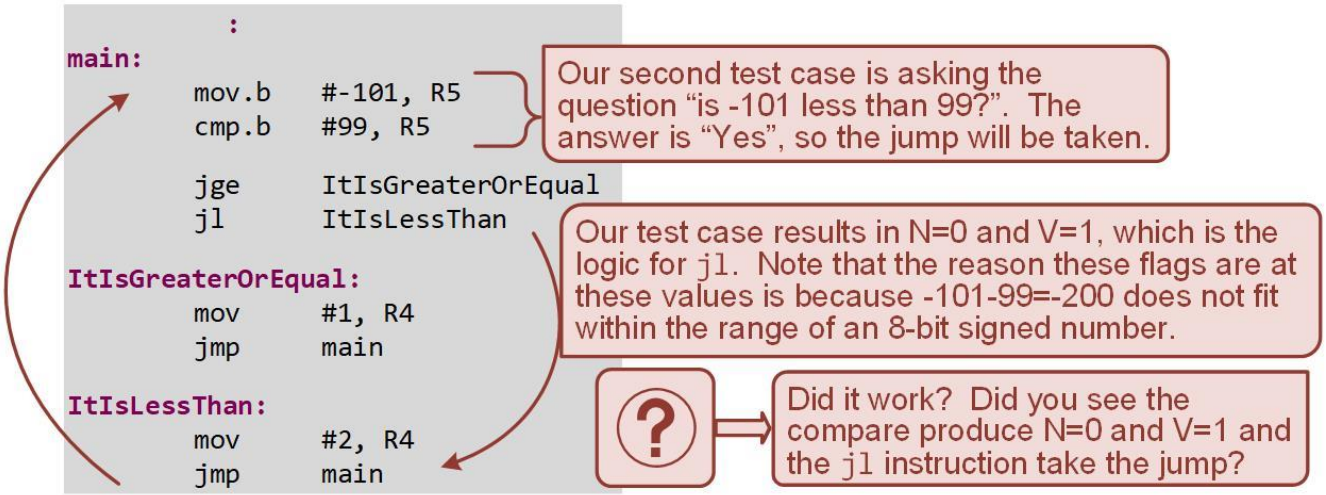
Step 7: Step your program to observe its operation.



Step 8: Now change the src values of the test case to the code shown: (**mov.b #101, R5**, **cmp #99, R5**).

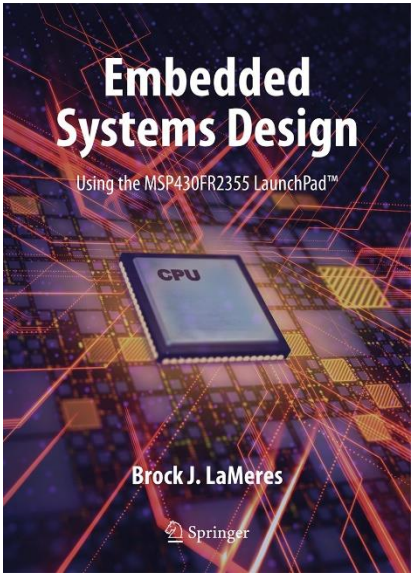
Step 9: Debug the program and observe the behavior of the program.

EXAMPLE: USING JUMPS BASED ON INEQUALITIES (JGE, JL)

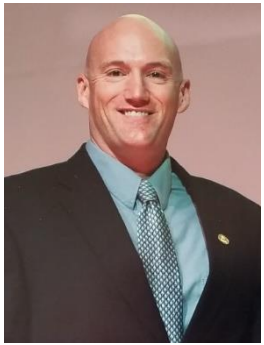


CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.2 CONDITIONAL JUMPS – OVERFLOW-BASED JUMPS

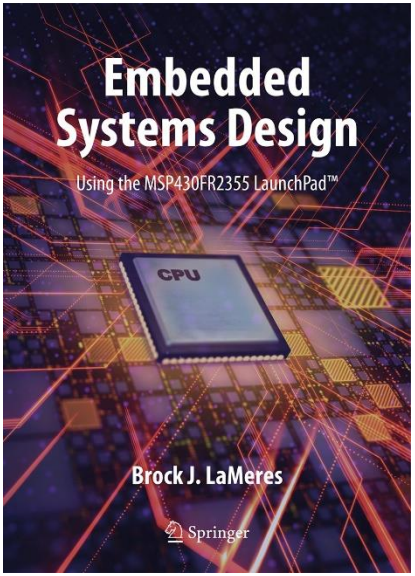


www.youtube.com/c/DigitalLogicProgramming_LaMeres



CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.3 IMPLEMENTING COMMON PROGRAMMING CONSTRUCTS IN ASSEMBLY - WHILE() LOOPS



8.3.1 IMPLEMENTING WHILE() LOOP FUNCTIONALITY

- **While() Loop** – sequence of statements that will continually execute as long as a Boolean condition at the beginning of the loop is satisfied.
- In **assembly**, this behavior is implemented with a combination of test and compare, and conditional jump instructions.
- In the **C programming**, the Boolean condition is inserted within the parenthesis of the while() keyword and the statements to be executed are listed within curly brackets ({}).

EXAMPLE: IMPLEMENTING WHILE() LOOPS IN ASSEMBLY

C Code Equivalent

```
while (Var1 == 3)
{
   Var2 = 1;
}
```

Execute loop as long as Var1=3. Exit otherwise.

```
while (1)
{
   Var2 = 2;
}
```

Execute forever.

Assembly Code Implementation

```
while1:
        cmp.w   #3, Var1
        jnz     end_while1

        mov.w   #1, Var2
        jmp     while1
end_while1:

while2:
        mov.w   #2, Var2
        jmp     while2
end_while2:

;---------------------
; Memory Allocation
;---------------------
        .data
        .retain

Var1:   .short  3
Var2:   .space  2
```

The first thing we do is check the Boolean condition that enables the loop. If this condition is NOT true, we exit the loop.

At the end of the while() loop we always jump to the top and re-check the condition.

This 2nd while() loop example shows a common way to create an infinite loop using an unconditional jump.

Here is how these directives allocate memory:

Label	Addr	Data
Var1	2000h	*0003h*
Var2	2002h	-

EXAMPLE: IMPLEMENTING WHILE() LOOPS IN ASSEMBLY

Step 1: Create a new Empty Assembly-only project titled:

Asm_Flow_While_Loops

Step 2: Type in the following code into the main.asm file where the comments say “Main loop here.”

EXAMPLE: IMPLEMENTING WHILE() LOOPS IN ASSEMBLY

Step 3: Debug your program and correct any errors.

Step 4: Set a breakpoint before the first instruction

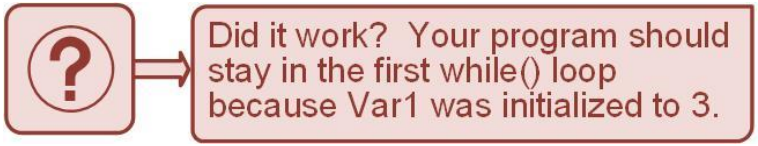
cmp.w #3, Var1

Step 5: Open the memory browser and go to address 0x2000. You should see a word at 0x2000 initialized to 3 and a word at 0x2002 reserved with no values.

EXAMPLE: IMPLEMENTING WHILE() LOOPS IN ASSEMBLY

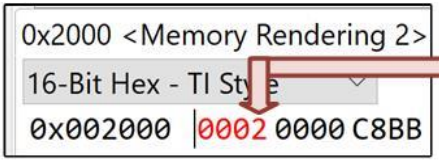
Step 6: Run your program to the breakpoint.

Step 7: Step your program to observe the first while() loop's behavior.



EXAMPLE: IMPLEMENTING WHILE() LOOPS IN ASSEMBLY

Step 8: Now you are going to manually change the value of Var1 from 3 to 2 in the memory browser.



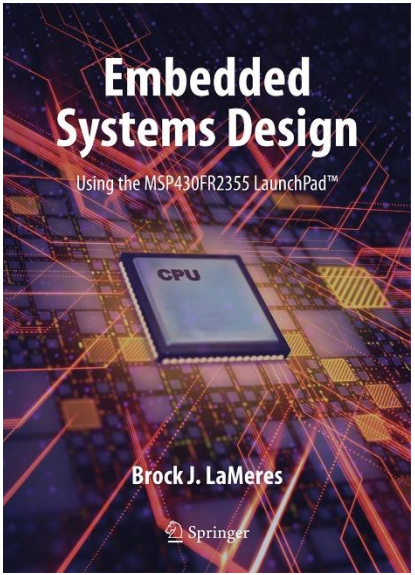
Click in the value at address 0x2000, delete the 3, type in 2, press enter. The new value will take effect once you step the program again.

Step 9: Continue stepping your program with the new value of Var1 = 2.

Did it work? Your program should have exited the first while() loop and entered the second while() loop where it stayed forever.

CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

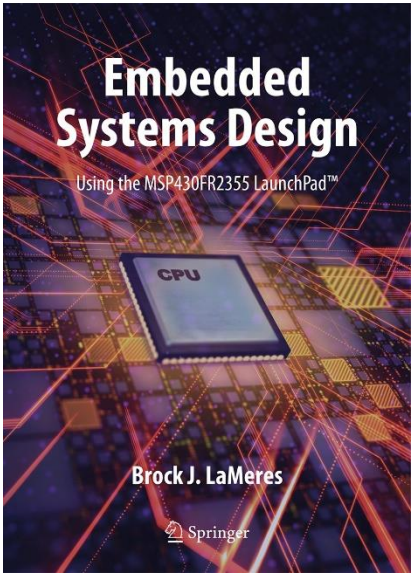
8.3 IMPLEMENTING COMMON PROGRAMMING CONSTRUCTS IN ASSEMBLY - WHILE() LOOPS



www.youtube.com/c/DigitalLogicProgramming_LaMeres

CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

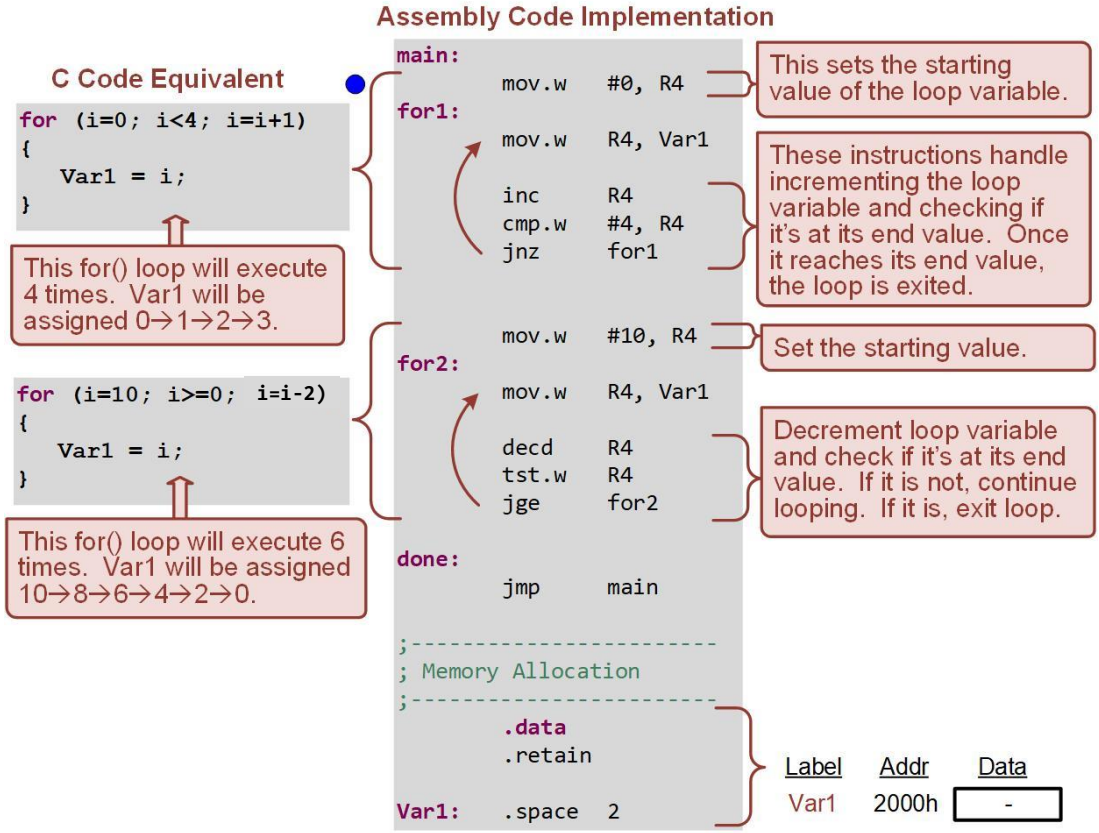
8.3 IMPLEMENTING COMMON PROGRAMMING CONSTRUCTS IN ASSEMBLY - FOR() LOOPS



8.3.2 IMPLEMENTING FOR() LOOP FUNCTIONALITY

- **For() Loop** – a sequence of statements that will execute a fixed number of times.
- The number of times to iterate is specified by stating a **loop variable**, the **starting value of the variable**, the **final value of the variable**, and the **method that the variable will be incremented/decremented**.
- The loop variable can be used as an offset when accessing blocks of storage.
- In assembly, for() loop functionality is accomplished using increments/decrements, test, compare, and conditional jump instructions.

EXAMPLE: IMPLEMENTING FOR() LOOPS IN ASSEMBLY



EXAMPLE: IMPLEMENTING FOR() LOOPS IN ASSEMBLY

Step 1: Create a new Empty Assembly-only project titled:

Asm_Flow_For_Loops

Step 2: Type in the following code into the main.asm file where the comments say “Main loop here.”

EXAMPLE: IMPLEMENTING FOR() LOOPS IN ASSEMBLY

Step 3: Debug your program and correct any errors.

Step 4: Set a breakpoint before the first instruction

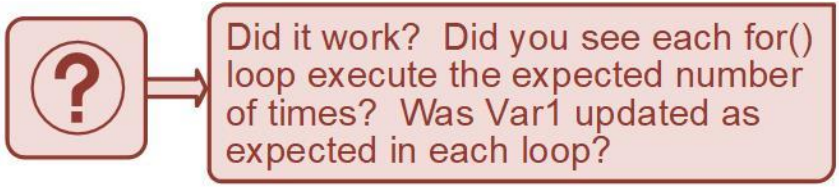
mov.w #0, R4

Step 5: Open the memory browser and go to address 0x2000. You should see a word at 0x2000 reserved with no values.

EXAMPLE: IMPLEMENTING FOR() LOOPS IN ASSEMBLY

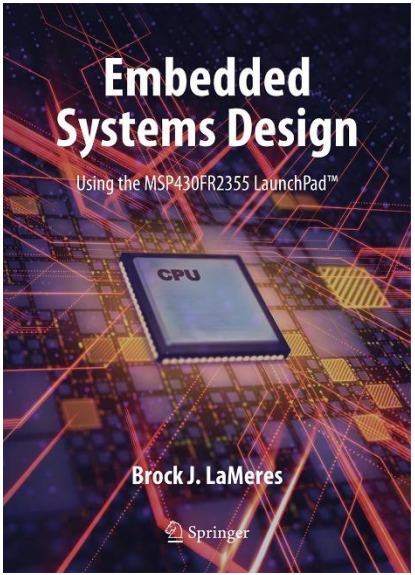
Step 6: Run your program to the breakpoint.

Step 7: Step your program to observe the for() loop's behavior.



CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

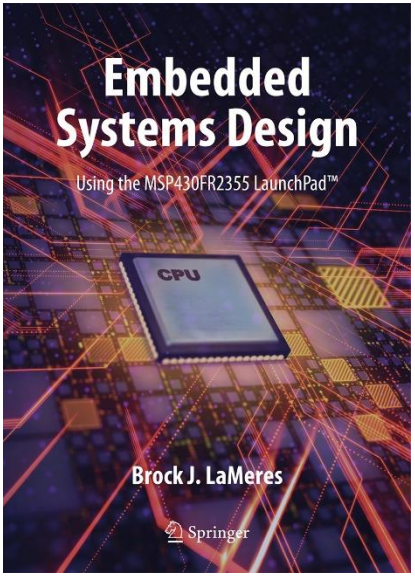
8.3 IMPLEMENTING COMMON PROGRAMMING CONSTRUCTS IN ASSEMBLY - FOR() LOOPS



www.youtube.com/c/DigitalLogicProgramming_LaMeres

CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.3 IMPLEMENTING COMMON PROGRAMMING CONSTRUCTS IN ASSEMBLY - IF/ELSE STATEMENTS



8.3.3 IMPLEMENTING IF/ELSE FUNCTIONALITY

- **If/else Statement** – allows statements to be selectively executed based on the result of a Boolean condition.
- A Boolean condition is entered after the *if* portion of the construct.
- If the **condition is true**, the statements listed within the subsequent curly brackets will be executed.
- If the **condition is not true**, then the first set of statements are skipped, and the statements listed within the curly brackets after the *else* portion of the construct are executed.

8.3.3 IMPLEMENTING IF/ELSE FUNCTIONALITY

- If/else statements can be nested to provide the ability to check multiple Boolean conditions.

- In assembly, if/else functionality is accomplished using compare, unconditional jump, and conditional jump instructions.

EXAMPLE: IMPLEMENTING IF/ELSE STATEMENTS IN ASSEMBLY

C Code Equivalent

```
int Cnt1;

Cnt1 = 0;

while (1)
{
    if (Cnt1 == 10)
    {
        Cnt1 = 0;
    }
    else
    {
        Cnt1 = Cnt1 + 1;
    }
}
```

Assembly Code Implementation

```
        mov.w   #0, R15
while:

if:
        cmp.w   #5, R15
        jnz     else
        mov.w   #0, R15
        jmp
        end_if
else:
        inc.w   R15
end_if:

end_while:
        jmp     while
```

First, check whether the Boolean condition is true using a compare. If it isn't, jump to else label.

If the condition is true, these statements will be executed and then a jump to the end of the if/else is taken.

EXAMPLE: IMPLEMENTING IF/ELSE STATEMENTS IN ASSEMBLY

Step 1: Create a new Empty Assembly-only project titled:

Asm_Flow_If_Else

Step 2: Type in the following code into the main.asm file where the comments say “Main loop here.”

EXAMPLE: IMPLEMENTING IF/ELSE STATEMENTS IN ASSEMBLY

Step 3: Debug your program and correct any errors.

Step 4: Set a breakpoint before the first instruction

mov.w #0, R15

Step 5: Open the register browser and observe R15.

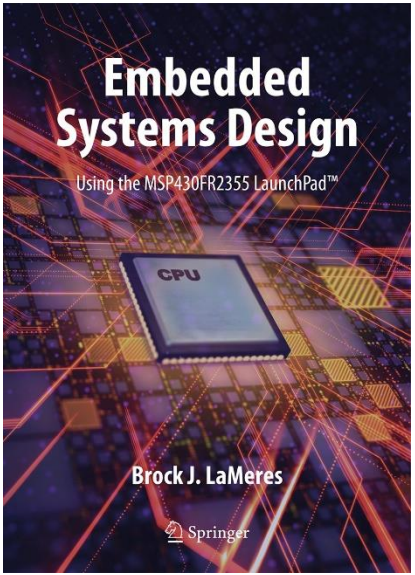
Step 6: Run your program to the breakpoint.

Step 7: Step your program to observe the if/else construct's behavior.

Did it work? You should have seen R15 increment from 0→1→2→3→4→5→0→1→ … and repeat forever.

CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.3 IMPLEMENTING COMMON PROGRAMMING CONSTRUCTS IN ASSEMBLY - IF/ELSE STATEMENTS

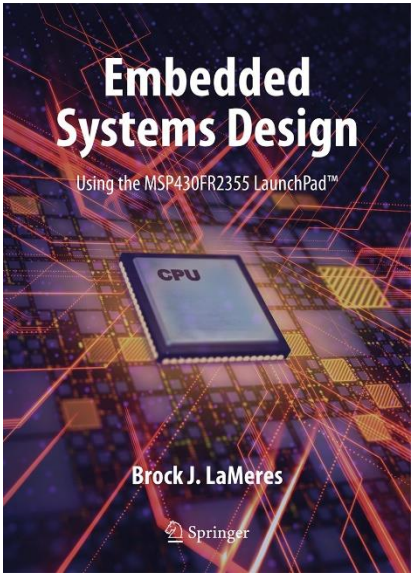


www.youtube.com/c/DigitalLogicProgramming_LaMeres

Embedded Systems Design

Chapter 8: Program Flow Instructions

8.3 Implementing Common Programming Constructs in Assembly - Switch/Case Statements



Brock J. LaMeres, Ph.D.

8.3.4 IMPLEMENTING SWITCH/CASE FUNCTIONALITY

- **Switch/Case statement** – allows a variable to be tested against a list of values.
- This first value in the list that matches the variable will result int eh execution of statements associated with the value.
- Similar to nested if/else statements, but the syntax is more amenable to large lists of comparisons.
- This is implemented with a sequence of compare instructions, each with an associated conditional jump to a corresponding series of instructions to be executed.

EXAMPLE: IMPLEMENTING SWITCH/CASE STATEMENTS IN ASSEMBLY

C Code Equivalent

```
int VarIn = 0;
int VarOut = 0;

while (1)
{
  switch(VarIn) {
    case 0  : VarOut = 0x01;
              break;
    case 1  : VarOut = 0x02;
              break;
    case 2  : VarOut = 0x04;
              break;
    case 3  : VarOut = 0x08;
              break;
    default : VarOut = 0x00;
   }
}
```

Assembly Code Implementation

```
        mov.w   #0, R14  ; VarIn
        mov.w   #0, R15  ; VarOut
while:

switch:
        cmp.w   #0, R14
        jz      case0
        cmp.w   #1, R14
        jz      case1
        cmp.w   #2, R14
        jz      case2
        cmp.w   #3, R14
        jz      case3
        jmp     default
case0:
        mov.w   #0001h, R15
        jmp     end_switch
case1:
        mov.w   #0002h, R15
        jmp     end_switch
case2:
        mov.w   #0004h, R15
        jmp     end_switch
case3:
        mov.w   #0008h, R15
        jmp     end_switch
default:
        mov.w   #0000h, R15

end_switch:

end_while:
        jmp     while
```

These instructions compare the input to a list of values and then jump to the respective case label.

Each case executes the desired instruction(s) and then jumps to the end of the switch statement.

EXAMPLE: IMPLEMENTING SWITCH/CASE STATEMENTS IN ASSEMBLY

Step 1: Create a new Empty Assembly-only project titled:

Asm_Flow_Switch_Case

Step 2: Type in the following code into the main.asm file where the comments say “Main loop here.”

EXAMPLE: IMPLEMENTING SWITCH/CASE STATEMENTS IN ASSEMBLY

Step 3: Debug your program and correct any errors.

Step 4: Set a breakpoint before the first instruction

mov.w #0, R14

Step 5: Open the register browser and observe R14 and R15.

EXAMPLE: IMPLEMENTING SWITCH/CASE STATEMENTS IN ASSEMBLY

Step 6: Run your program to the breakpoint.

Step 7: Step your program to observe the switch/case statemenet's behavior.

Step 8: Manually alter the value of R14 by clicking in the register browser and entering values between 0 and 3. Remember to press enter and step again for these values to take place.

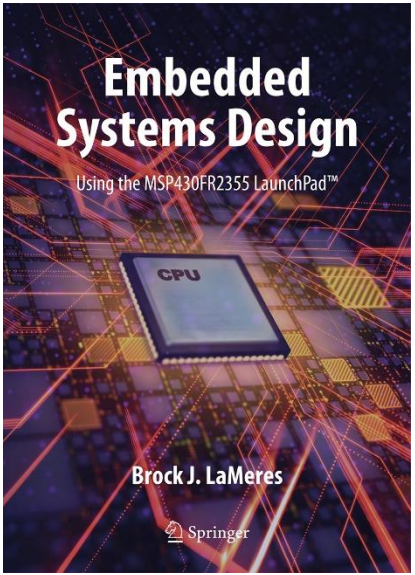
Did it work? As you enter different values for R14, you should see corresponding jumps to the appropriate case label and R15 should be updated accordingly.

Image Courtesy of https://neodem.wp.horizon.ac.uk/

Embedded Systems Design

Chapter 8: Program Flow Instructions

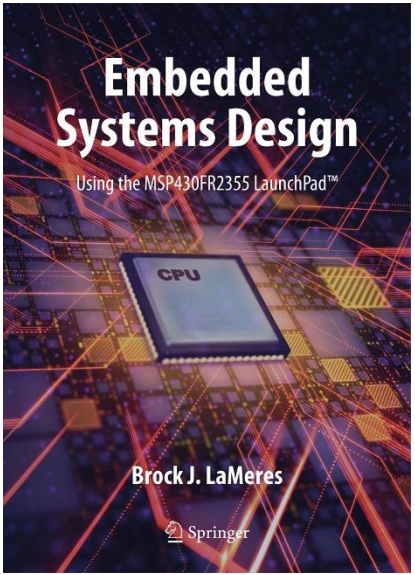
8.3 Implementing Common Programming Constructs in Assembly - Switch/Case Statements

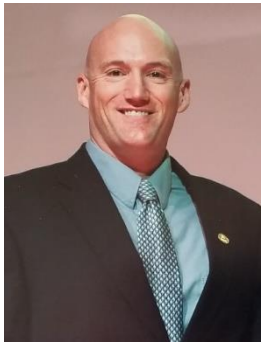


www.youtube.com/c/DigitalLogicProgramming_LaMeres

CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

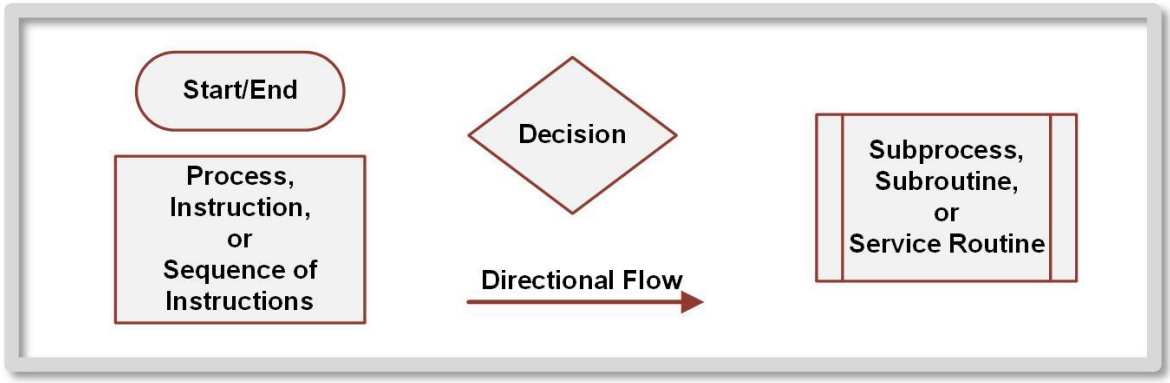
8.4 FLOW CHARTS





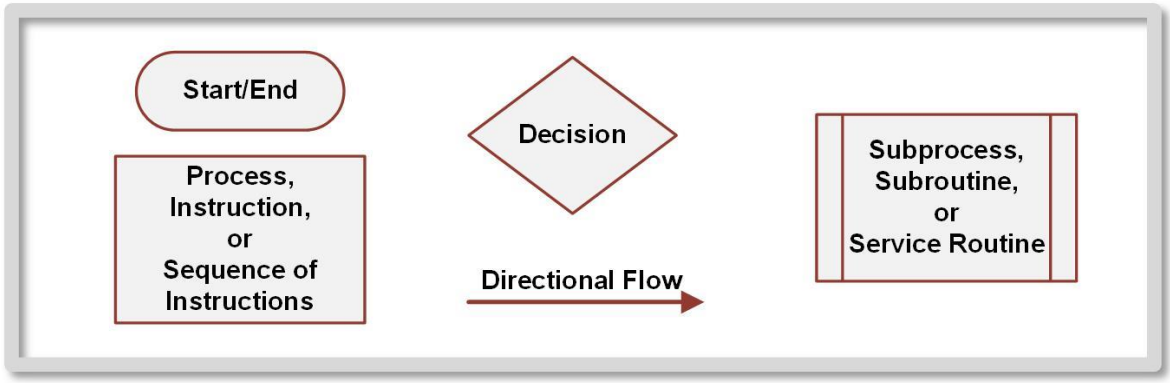
8.4 FLOW CHARTS

- **Flow chart** – graphical depiction of the behavior of a program.
- Flow charts allow the algorithm to be thought through prior to implementation.
- **Oval** – represents the start and end to a program.
- **Rectangle** – represents a process, which can be a single instruction or a sequence of instructions that accomplishes a specific task.



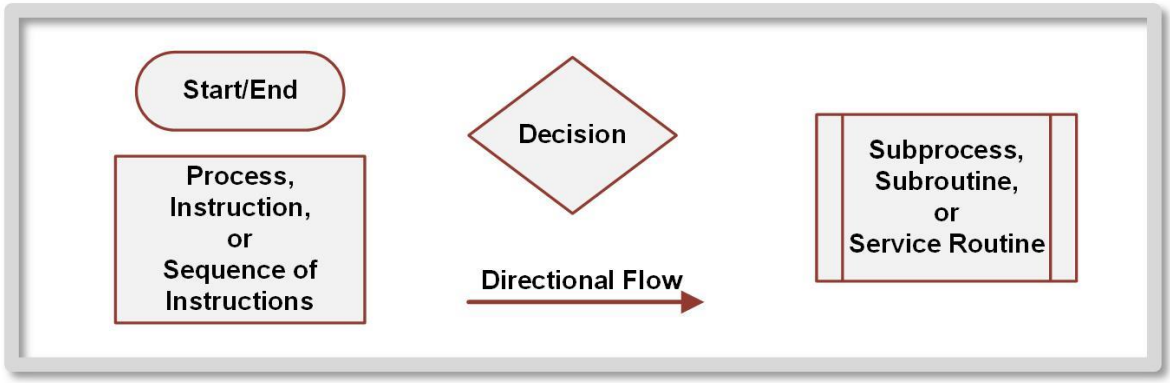
8.4 FLOW CHARTS

- **Diamond** – represents a decision where the corners of the shape represent different paths the program can take based on the decision.
- **Rectangle with double sides** – represents a sequence of instructions that occurs separate from the main program flow; subroutine or service routine.



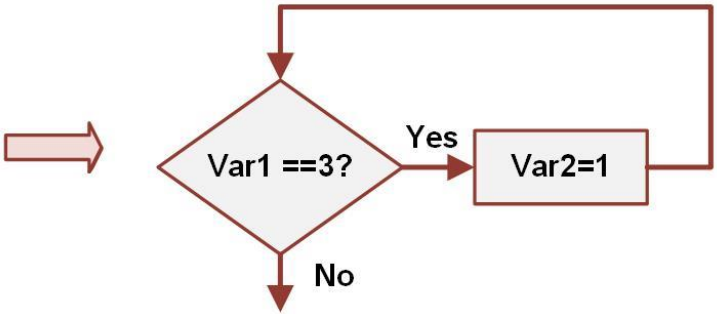
8.4 FLOW CHARTS

- **Diamond** – represents a decision where the corners of the shape represent different paths the program can take based on the decision.
- **Rectangle with double sides** – represents a sequence of instructions that occurs separate from the main program flow; subroutine or service routine.



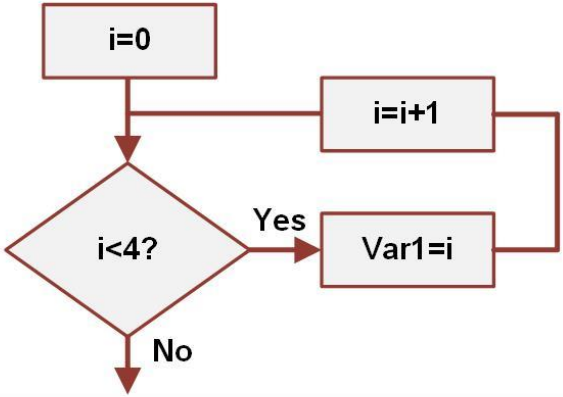
8.4 FLOW CHARTS – WHILE() LOOP

```
while (Var1 == 3)
{
  Var2 = 1;
}
```



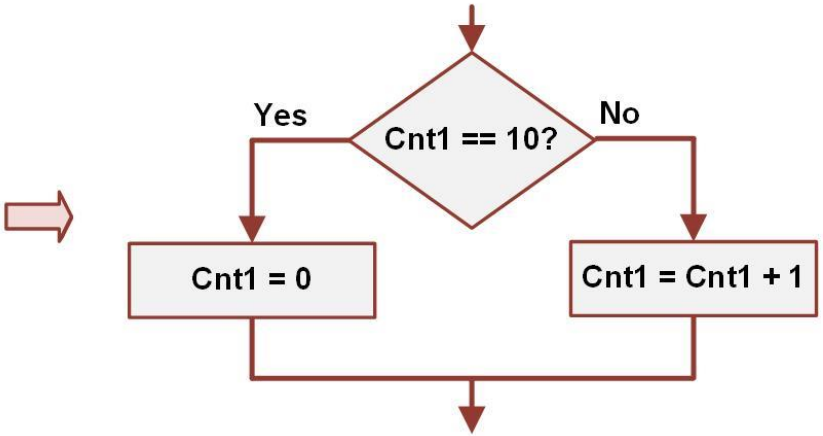
8.4 FLOW CHARTS – FOR() LOOP

```
for (i=0; i<4; i=i+1)
{
   Var1 = i;
}
```

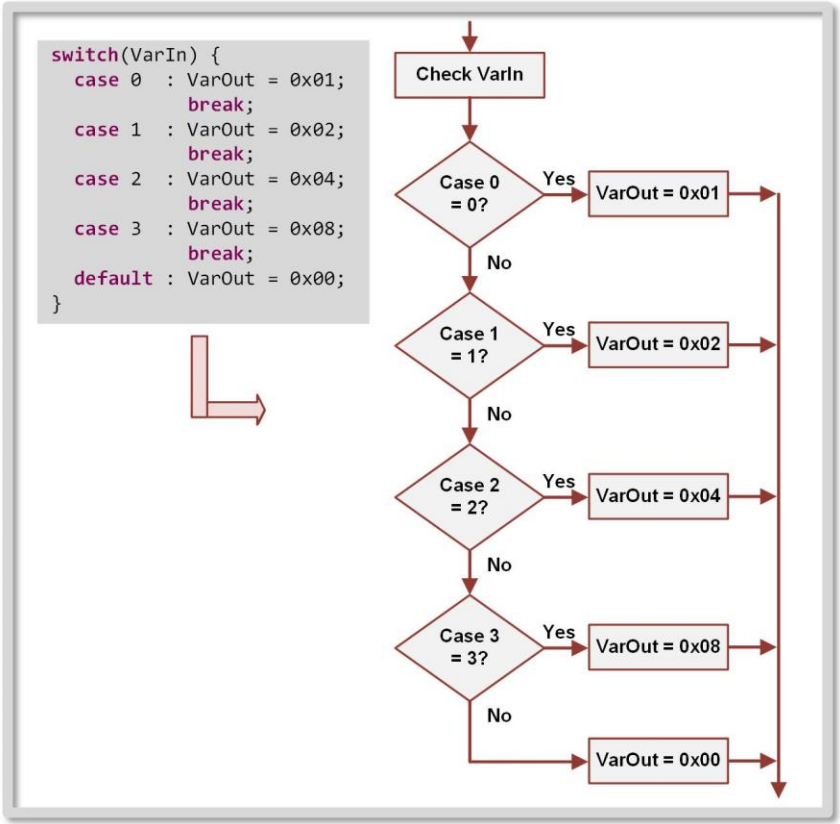


8.4 FLOW CHARTS – IF/ELSE STATEMENT

```
if (Cnt1 == 10)
{
  Cnt1 = 0;
}
else
{
  Cnt1 = Cnt1 + 1;
}
```

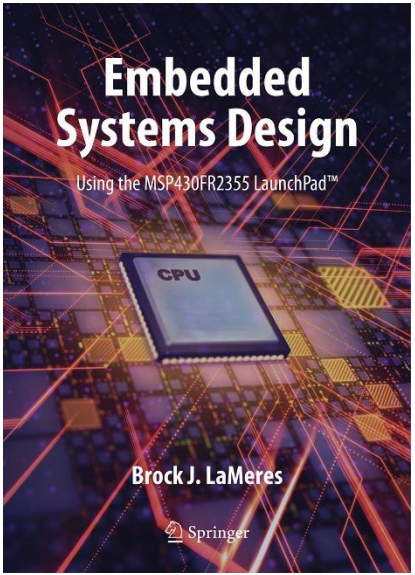


8.4 FLOW CHARTS – SWITCH/CASE STATEMENT



CHAPTER 8: PROGRAM FLOW INSTRUCTIONS

8.4 FLOW CHARTS

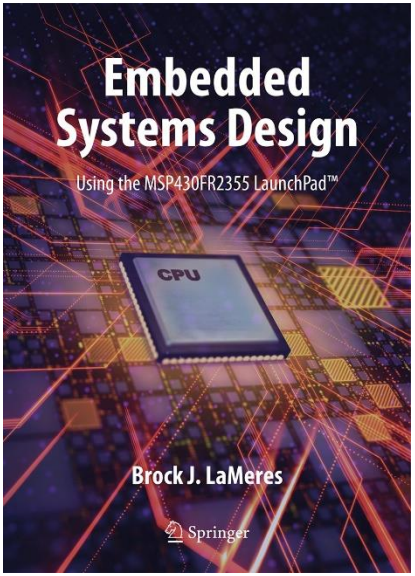


www.youtube.com/c/DigitalLogicProgramming_LaMeres

Embedded Systems Design

Chapter 8: Program Flow Instructions

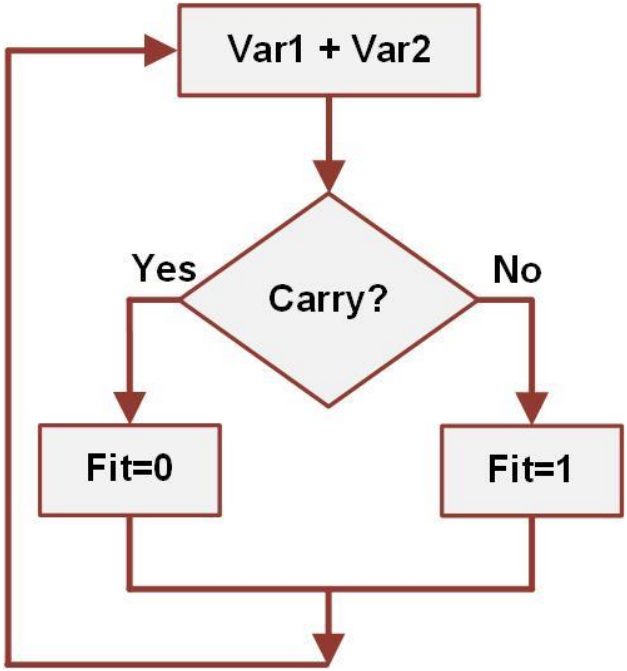
8.4 Flow Charts – Implementing Programs from Flow Charts



Brock J. LaMeres, Ph.D.

EXAMPLE: IMPLEMENTING ASSEMBLY CODE FROM A FLOW CHART

Create an assembly program that will implement the functionality of the following flow chart. Var1, Var2, and Fit will be 16-bit variables in data memory that we need to reserve. Var1 and Var2 are updated by another process, but our program will need to update Fit per flow chart.



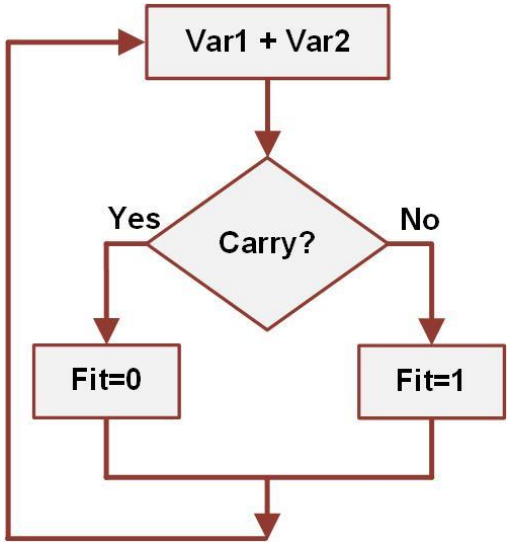
Step 1: Create a new Empty Assembly-only project titled:

Asm_Flow_DesignFromFlowChart

EXAMPLE: IMPLEMENTING ASSEMBLY CODE FROM A FLOW CHART

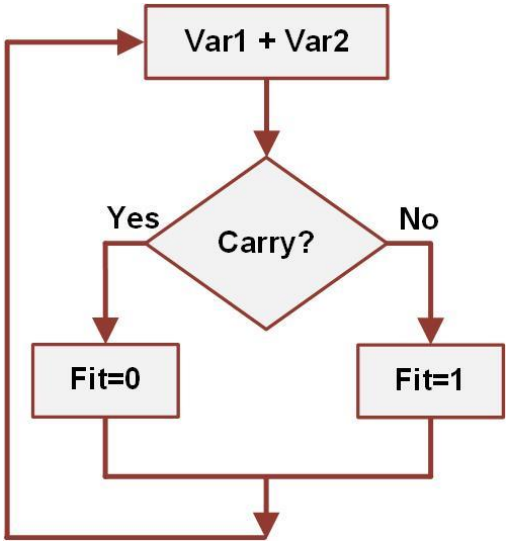
Step 2: Let's think about the functionality of this flow chart.

- First, we'll need to reserve 3x words of data memory called Var1, Var2, and Fit.
- Second, we'll add Var1 and Var2 to the main program.
- Third, we will need to check if the C-flag was asserted. We can use the **jc** and **jnc** conditional jumps for that.
- Those jumps will go to labels that will set the value of Fit.
- Finally, this loop needs to be repeated forever, which can be done by doing an unconditional jump to the start of the program ("main").



EXAMPLE: IMPLEMENTING ASSEMBLY CODE FROM A FLOW CHART

Step 3: Type in the following code into the main.asm file where the comments say “Main loop here.”




```
main:
        mov.w   Var1, R4
        add.w   Var2, R4
        jc      Carry
        jnc     NoCarry
Carry:
        mov.w   #0, Fit
        jmp     done
NoCarry:
        mov.w   #1, Fit
        jmp     done
done:
        jmp     main
;-----------------------
; Memory Allocation
;-----------------------
        .data
        .retain
Var1:   .space  2
Var2:   .space  2
Fit:    .space  2
```

EXAMPLE: IMPLEMENTING ASSEMBLY CODE FROM A FLOW CHART

Step 4: Debug your program and correct any errors.

Step 5: Set a breakpoint before the first instruction

Step 6: Open the memory browser and observe Var1, Var2, and Fit starting at address 0x2000h.

Step 7: Run your program to the breakpoint.

Step 8: Step your program to observe its behavior.

Image Courtesy of https://neodem.wp.horizon.ac.uk/

EXAMPLE: IMPLEMENTING ASSEMBLY CODE FROM A FLOW CHART

Step 9: Manually change the values of Var1 and Var2 to generate a carry. Try setting Var1 = FFFFh and Var2 = 1 to generate a carry.

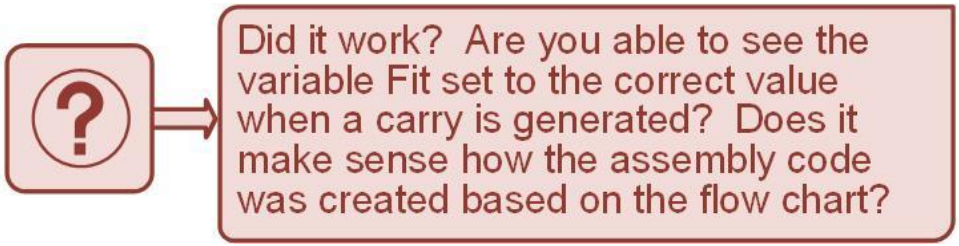
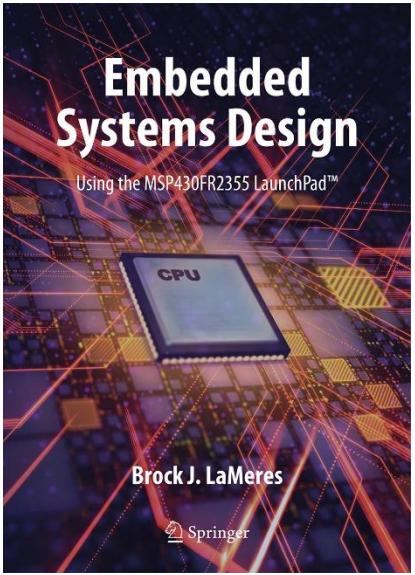


Image Courtesy of https://neodem.wp.horizon.ac.uk/

Embedded Systems Design

Chapter 8: Program Flow Instructions

8.4 Flow Charts – Implementing Programs from Flow Charts



www.youtube.com/c/DigitalLogicProgramming_LaMeres

Brock J. LaMeres, Ph.D.