

Version 2.03

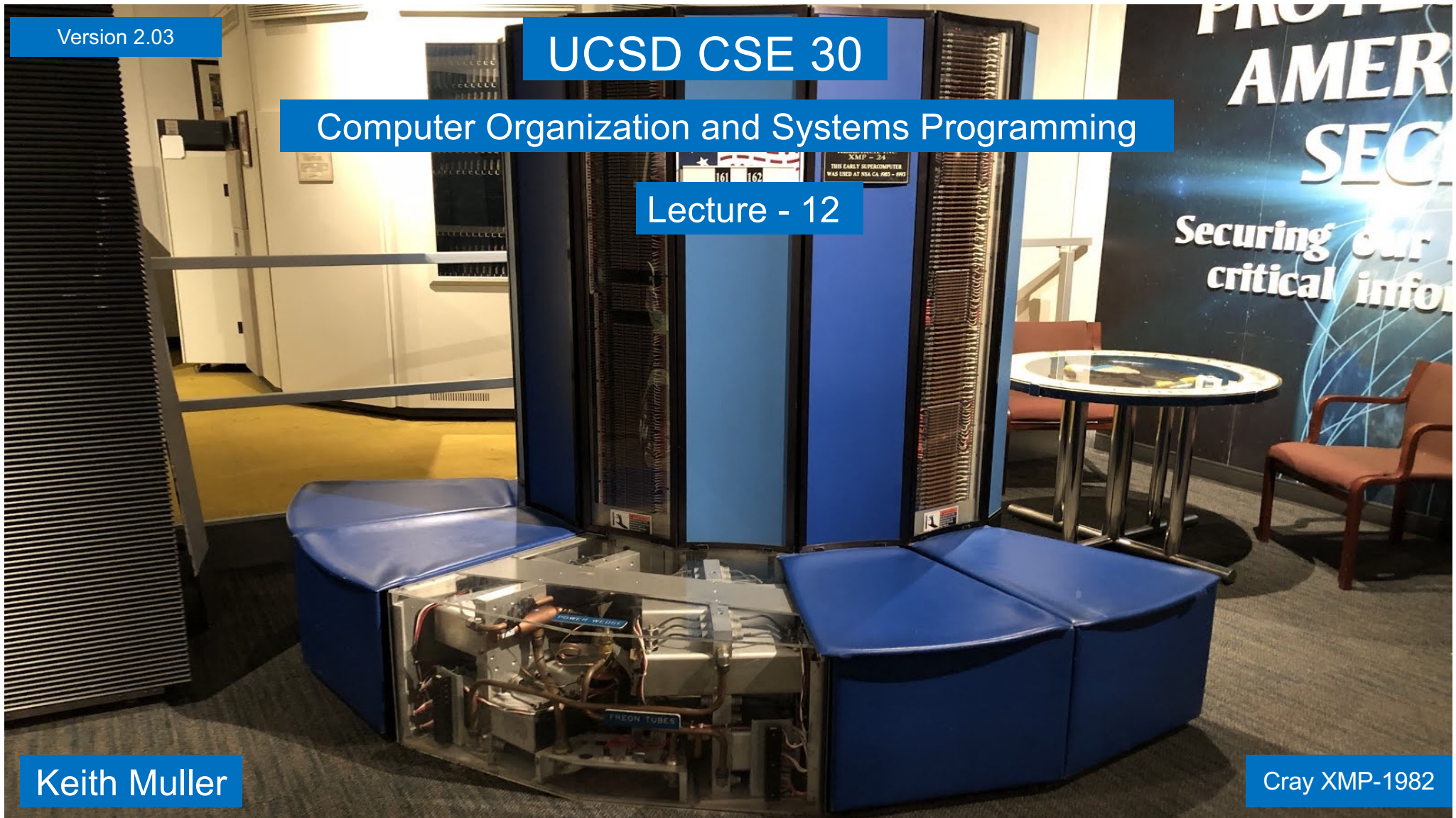
UCSD CSE 30

Computer Organization and Systems Programming

Lecture - 12

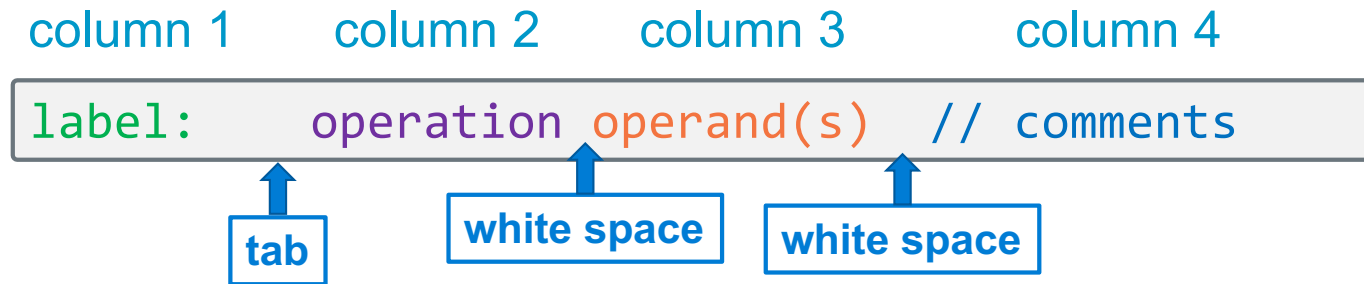
Keith Muller

Cray XMP-1982





Line Layout in an Arm Assembly Source



- Assembly language source text files are **line oriented** (each ending in a '\n')
- **Each line represents** a **starting address in memory** and does **one of**:
 1. Specifies the **contents of memory** for a **variable** (segments containing data)
 2. Specifies the **contents of memory** for an **instruction** (text segment)
 3. **Assembler directives** **tell the assembler to do something** (for example, change label scope, define a macro, etc.) that **does not allocate memory**
- **Each line** is **organized into up to four columns**
 - Not every column is used on each line
 - Not every line will result in **memory being allocated**

Labels in Arm Assembly - 1

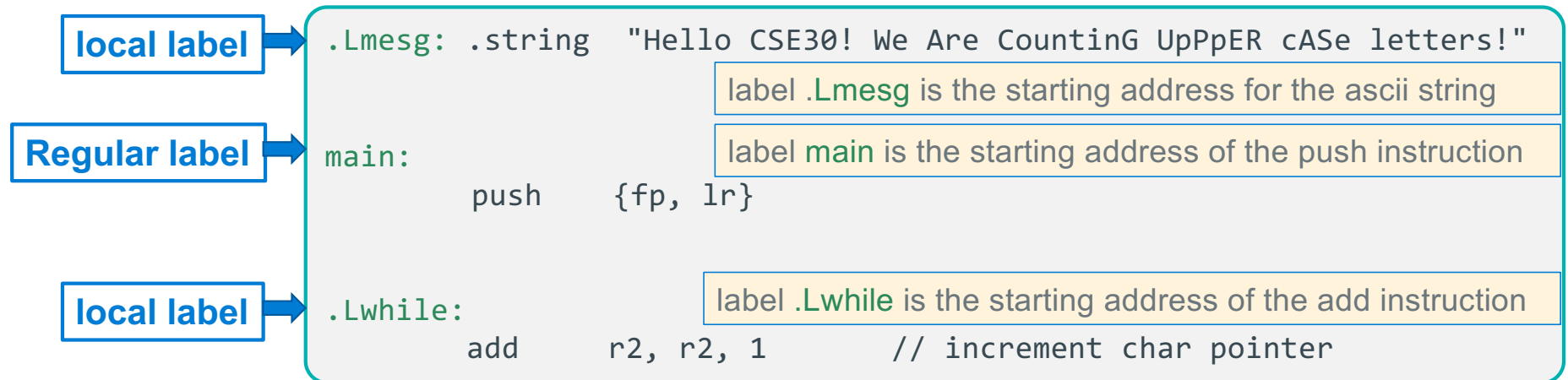
```
label:  operation operand(s)  // comment

        // assembler directive below
cnt:    .word 5                /* define a global int cnt = 5; */

        /* instruction example below */
add     r1    r2, r3          // add the values
```

1. **Labels** (optional); starts in column 1 (often on a line by itself ABOVE the "operation")
 - **Only put a label on a line** when you need to **associate** a **name** (a global variable, a function name, a loop/ branch target, etc.) to that **line's location in memory**
 - You then refer to the address **by name** in an **instruction**
2. **Operation type 1: assembler directives** (all start with a period e.g. **.word**)
3. **Operation Type 2: assembly language instructions**
4. **Zero or more operands** as required by the instruction or assembler directive
5. **Comments**: C and C++ style; also @ in the place of a C++ comment //

Labels in Arm Assembly - 2



- Remember, a **Label** associates a **name** with **memory location**
- **Regular Label:**
 - Used with a **Function name** (label) or all **static variables** in any of the data segments
- **Local Label:** Name starts with **.L** (local label prefix) only usable in the same file
 1. **Targets for**
 - a) branches: if switch, goto, break, continue,
 - b) loops: for, while, do-while
 2. **Anonymous variables** (the address of **literal** not the address of **foo** in the following)
`char *foo = "literal";`

Unconditional Branching – Forces Execution to Continue at a Specified Label (goto)



imm24 is Relative direction
from the branch instruction (in +/- instructions)

Unconditional Branch instruction (*branch to only local labels in CSE30*)

b **.Llabel**

- Causes an unconditional branch (aka goto) to the instruction with the address **.Llabel**
- **.Llabel** is called a **branch target label** (the "*target*" of a branch instruction)
- **Be careful! do not to branch to a function label!**
- **.Llabel**: translates into an number offset being **imm24 shifted left two bits** (+/- 32 MB)

```
        b        .Ldone                // goto .Ldone
        :
.Ldone:
        add      r0, EXIT_SUCCESS      // set return value
```

Examples of of Unconditional Branching

Unconditional Branch Forward

```
b .Lforward
add r1, r2, 4
add r0, r6, 2
add r3, r7, 4
.Lforward:
sub r1, r2, 4
```

Not a
practical
example as
this code is
unreachable

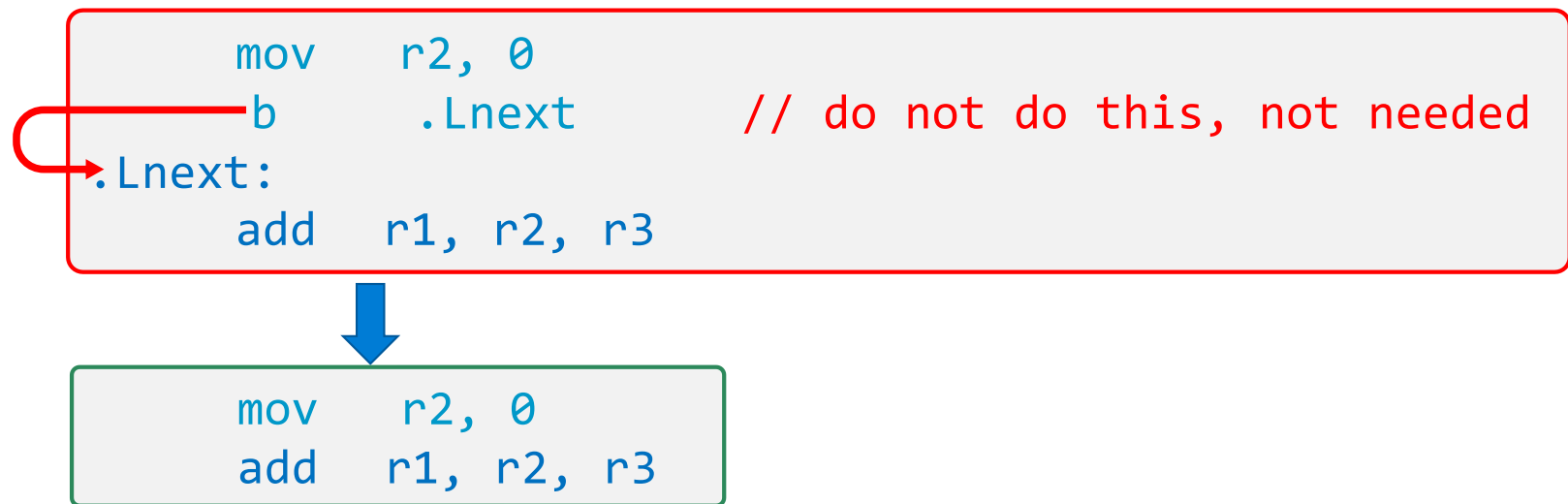
Branch target (local label)

Backward Branch (Infinite loop)

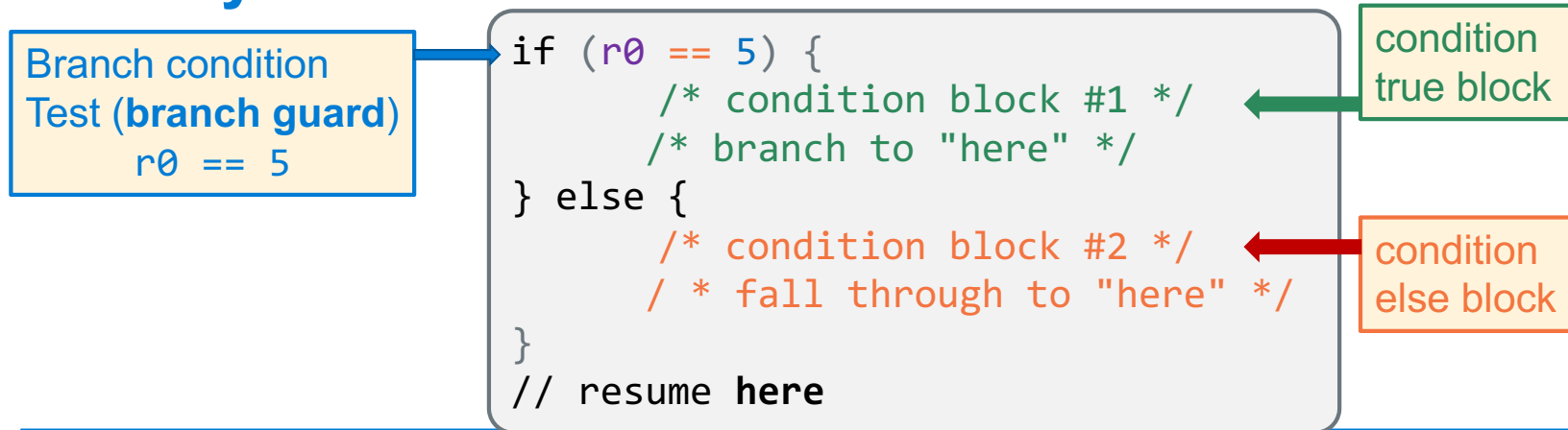
```
.Lbackward:
    add r1, r2, 4
    sub r1, r2, 4
    add r4, r6, r7
    b .Lbackward
// not reachable unless
// there is a label after the .b above
```

- Branches are used to change execution flow using labels as the branch target
- In these example, **.Lforward** and **.Lbackward** are the branch target labels
- Branch target labels are placed at the beginning of the line (or above it)
- Caution: Backward branches should only used with loops!

Never Branch to the following instruction: It is not needed!

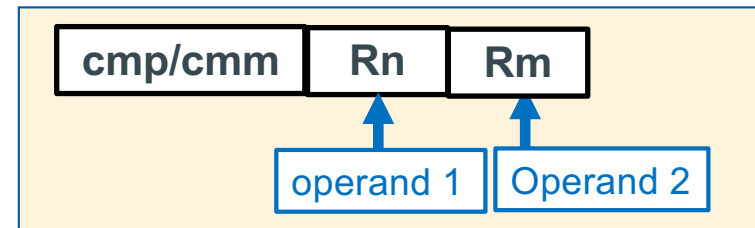
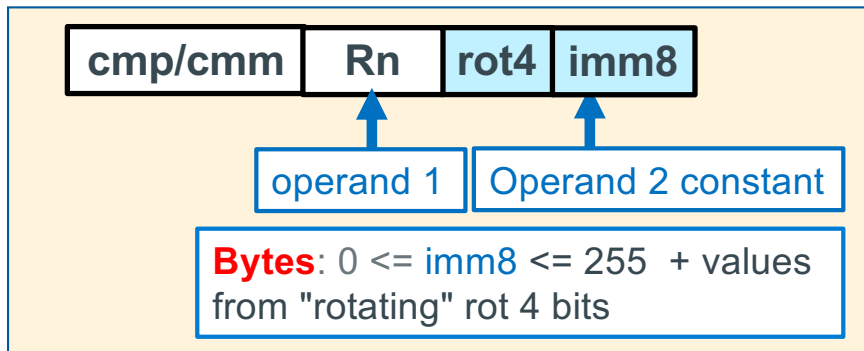


Anatomy of a Conditional Branch: If statement



- **Branch guard:** determines when to execute the "condition true block" or the "condition false block"
- **In C**, when the branch guard (condition test) evaluates non-zero you *fall through* to the **condition true** block, otherwise you branch to the **condition false (else)** block
- Step 1: evaluate the branch guard(s) (involves one or more compares/tests)
- Step 2: If branch guard evaluates to be **false**
 - **branch around** the **true block** and execute the **else block**
 - **otherwise "fall through"** and execute the **true block**
- **Block order** in C is where the **True Block** appears above the **False block**

cmp/cmm – Making Conditional Tests



```
cmp  Rn, constant    // Rn - constant; then sets condition flags
cmm  Rn, constant    // Rn + constant; then sets condition flags
cmp  Rn, Rm           // Rn - Rm; then sets condition flags
cmm  Rn, Rm           // Rn + Rm; then sets condition flags
```

The values stored in the registers `Rn` and `Rm` are not changed
The assembler will automatically substitute `cmm` for negative immediate values

```
cmp    r1, 0          // r1 - 0 and sets flags on the result
cmp    r1, r2          // r1 - r2 and sets flags on the result
```

Quick Overview of the Condition Bits/Flags



- The CPSR is a special register (like the other registers) in the CPU
- The four bits at the left are called the Condition Code flags
 - Summarize the result of a previous instruction
 - Not all instruction will change the CC bits
- Specifically, Condition Code flags are set by cmm/cmp (and others)

Example: `cmp r4, r3`

- **N** (Negative) flag: Set to 1 when the result of $r4 - r3$ is negative, set to 0 otherwise
- **Z** (Zero) flag: Set to 1 when the results of $r4 - r3$ is 0, set to 0 otherwise
- **C** (Carry bit) flag: Set to 1 when $r4 - r3$ does not have a borrow, set to 0 otherwise
- **V** flag (oVerflow): Set to 1 when $r4 - r3$ causes an overflow, set to 0 otherwise

Conditional Tests: Implementing ARM Branch guards

imm24 is Relative direction from the branch instruction (in +/- instructions)

cond	b _{suffix}	imm24
------	---------------------	-------

Branch instruction

b_{suffix} .Llabel



Use a local label with branch instructions

Condition	Meaning	Flag Checked
BEQ	Equal	Z = 1
BNE	Not equal	Z = 0
BGE	Signed \geq ("Greater than or Equal")	N = V
BLT	Signed $<$ ("Less Than")	N \neq V
BGT	Signed $>$ ("Greater Than")	Z = 0 && N = V
BLE	Signed \leq ("Less than or Equal")	Z = 1 N \neq V
BMI	Minus/negative	N = 1
BPL	Plus - positive or zero (non-negative)	N = 0
B	Branch Always (unconditional)	

- Bits in the condition field specify the **conditions** when the branch happens
- If the condition evaluates to be **true**, next instruction executed is at **.Llabel**:
- If the condition evaluates to be **false**, next instruction executed is immediately after the branch
- Unconditional branch is when the condition is **"always"**

Branch and Loop Guard Strategy

```
cmp r0, 10  
ble .Lendif  
// True Block  
.Lendif:
```

Branch guard

How to implement a **branch/loop guard** in CSE30

1. Use a **cmp/cmm** instruction to set the condition bits
2. Follow the **cmp/cmm** with **one or more variants of the conditional branch instruction**
 - **Conditional branch instructions** if evaluate to true (based on the flags set by the cmp) the next instruction will be the one at the branch label
 - **Otherwise**, execution **falls through** to **the instruction that immediately follows the branch**
 - You may have **one or more conditional branches** after a single cmp/cmm

Program Flow: Simple If statement, No Else

<i>C source Code</i>	<i>Incorrect Assembly</i>	<i>Correct Assembly</i>
<pre>int r0; if (r0 > 10) { // True Block }</pre>	<pre>cmp r0, 10 bgt .Lendif // True Block .Lendif:</pre>	<pre>cmp r0, 10 ble .Lendif // True Block .Lendif:</pre>

- Realize that in ARM assembly you can only either "fall through" to the next instruction or branch to a specific instruction
- Approach: **adjust** the conditional test then **branch around** the **true block**
- Use a conditional test that specifies the inverse of the condition used in C
 - This preserves **C block order**

Branch Guard "*Adjustment*" Table

Preserving C Block Order In Assembly

Compare in C	"Inverse" Compare in C	Assembly using Inverse Compare
==	!=	bne
!=	==	beq
>	<=	ble
>=	<	blt
<	>=	bge
<=	>	bgt


```
if (r0 compare 5)
    /* condition true block */
    /* then fall through */
}
```

```
cmp r0, 5
inverse compare .Lelse
// condition true block
// then fall through
.Lendif:
```

Arm Conditional Branching *Simple IF no else*

C If statement


```
int r0;  
if (r0 == 5) {  
    /* condition true block */  
    /* then fall through */  
}  
/* branch around to this code */
```



If r0 == 5 true
then **fall through** to
the true block

ARM If statement

```
cmp r0, 5  
bne .Lendif  
/* condition true block */  
/* then fall through */  
.Lendif:  
/* branch around to this code */
```

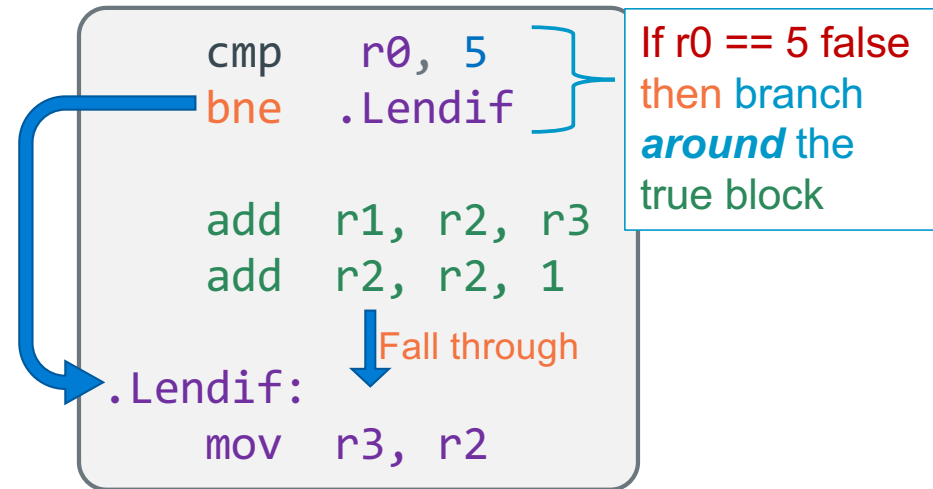


If r0 == 5 false
then branch **around**
the true block

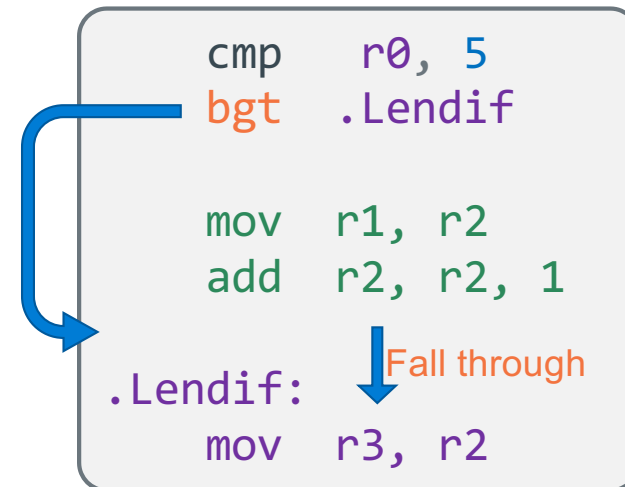
- If statements in ARM
- **Step 1:** make a conditional test using a **cmp** instruction
- **Step 2:** if test evaluates to **FALSE**, **branch around** the **condition true** block with a one of the conditional branch instruction

If statement examples – Branch Around the True block!

```
int r0;  
if (r0 == 5) {  
    r1 = r2++ + r3;  
}  
r3 = r2;
```



```
int r0;  
if (r0 <= 5) {  
    r1 = r2++;  
}  
r3 = r2;
```



Branching Avoid: Spaghetti Code ("goto structure")

- Do not use **unnecessary branches**
- Optimize your use of "fall throughs"
- For example: **Do not** make a **conditional branch** around an **unconditional branch** that immediately follows it

Do not do the following:

```
cmp r0, 0
```

```
beq .Lthen
```

```
b .Lendif
```

```
.Lthen:
```

```
add r1, r1, 1
```

```
.Lendif:
```

```
add r1, r1, 2
```

Caution!
Two adjacent
branches

Do the following:

```
cmp r0, 0
```

```
bne .Lendif
```

```
// fall through
```

```
add r1, r1, 1
```

```
.Lendif:
```

```
add r1, r1, 2
```

Program Flow: If with an Else

Approach:

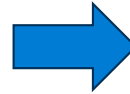
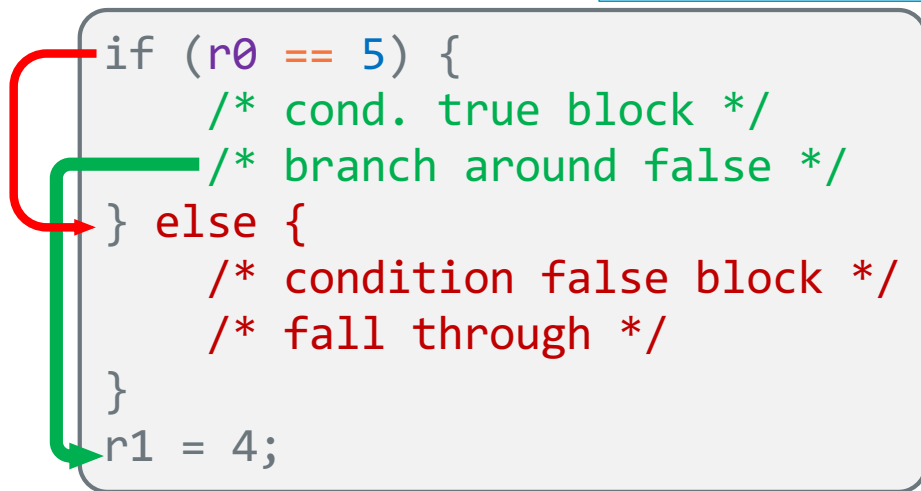
1. **adjust** the conditional test to branch to the **False Block**
2. **Fall through** to the **True Block**
3. Bottom of the **True Block** **unconditionally branches** around the **False block**

<i>C source Code</i>	<i>Assembly</i>
<pre>int r0; if (r0 > 10) { // true block // branch always around the false block } else { // false block }</pre>	<pre>cmp r0, 10 ble .Lelse // fall through // true block b .Lendif .Lelse: // false block // fall through .Lendif:</pre>

If with an Else Examples

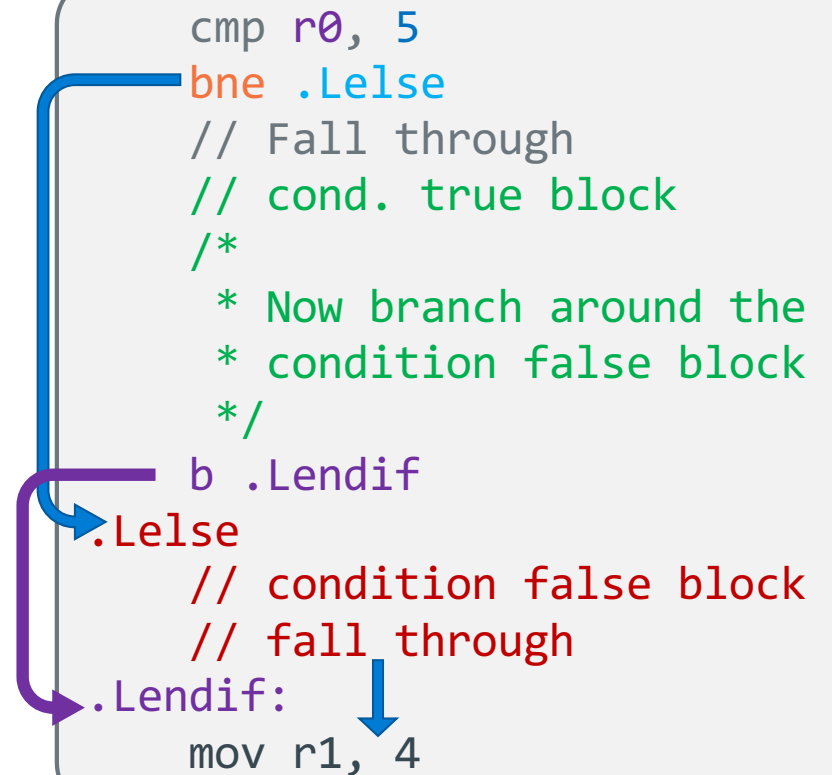
If $r0 == 5$ false
then branch **to**
false block

```
if (r0 == 5) {  
    /* cond. true block */  
    /* branch around false */  
} else {  
    /* condition false block */  
    /* fall through */  
}  
r1 = 4;
```

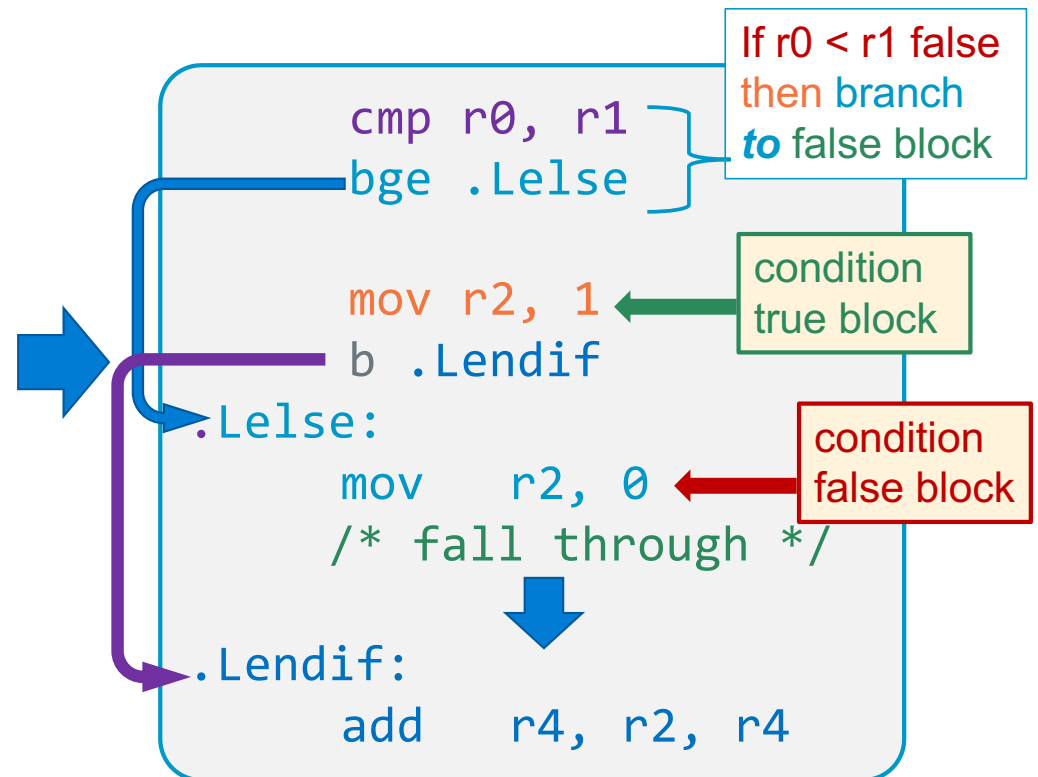
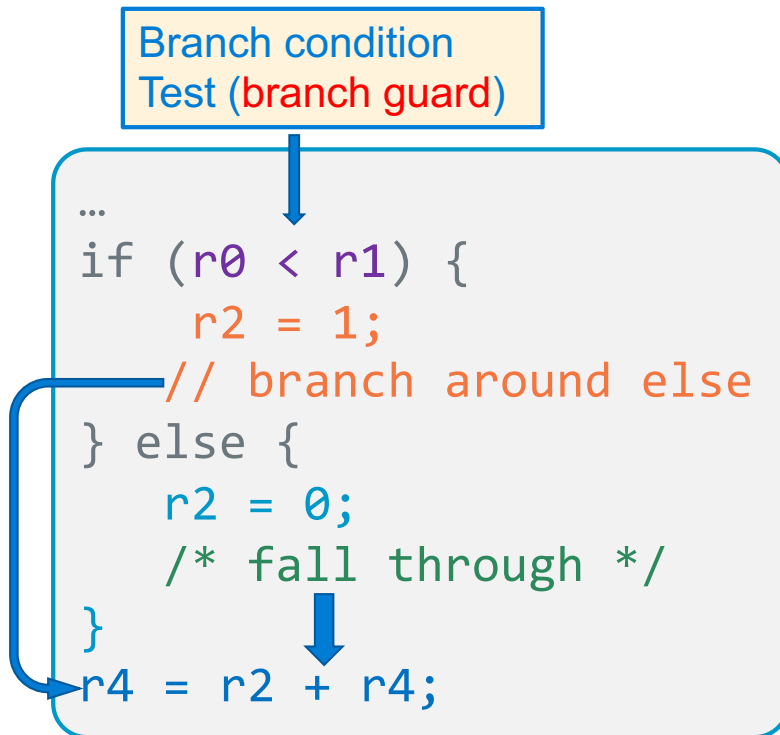


If $r0 == 5$ false
then branch
to false block

```
cmp r0, 5  
bne .Lelse  
// Fall through  
// cond. true block  
/*  
 * Now branch around the  
 * condition false block  
 */  
b .Lendif  
.Lelse  
// condition false block  
// fall through  
.Lendif:  
mov r1, 4
```



If with an Else Examples



If statement – C Block Reordering

Branch condition
Test (branch guard)

```
if (r0 == 5) {  
    /* condition block #1 */  
} else {  
    /* condition block #2 */  
    /* * fall through */  
}
```

condition
true block

condition
false block

- Block order: (the **order** the **blocks appear** in C code) can be changed by **inverting** the conditional test, **swapping** the order of the **true** and **false** blocks

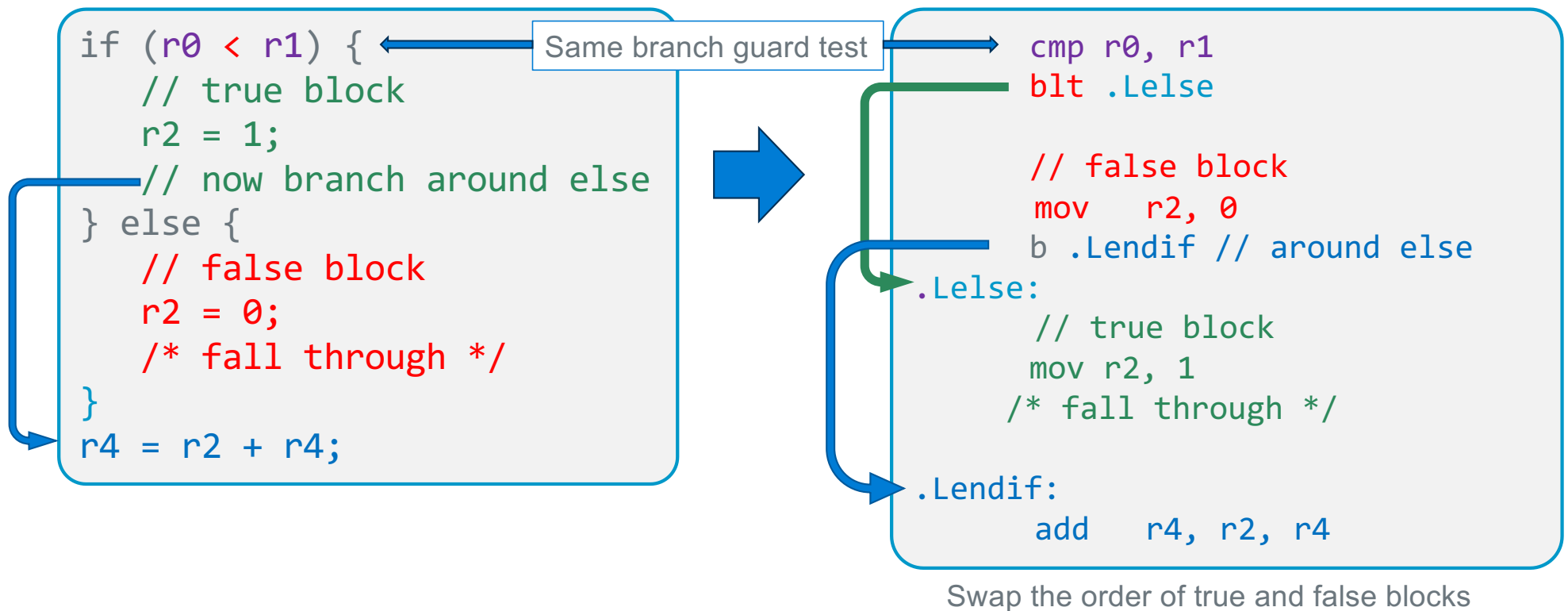
Branch condition
Test (branch guard)

```
if (r0 != 5) {  
    /* condition block #2 */  
} else {  
    /* condition block #1 */  
    /* * fall through */  
}
```

condition
true block

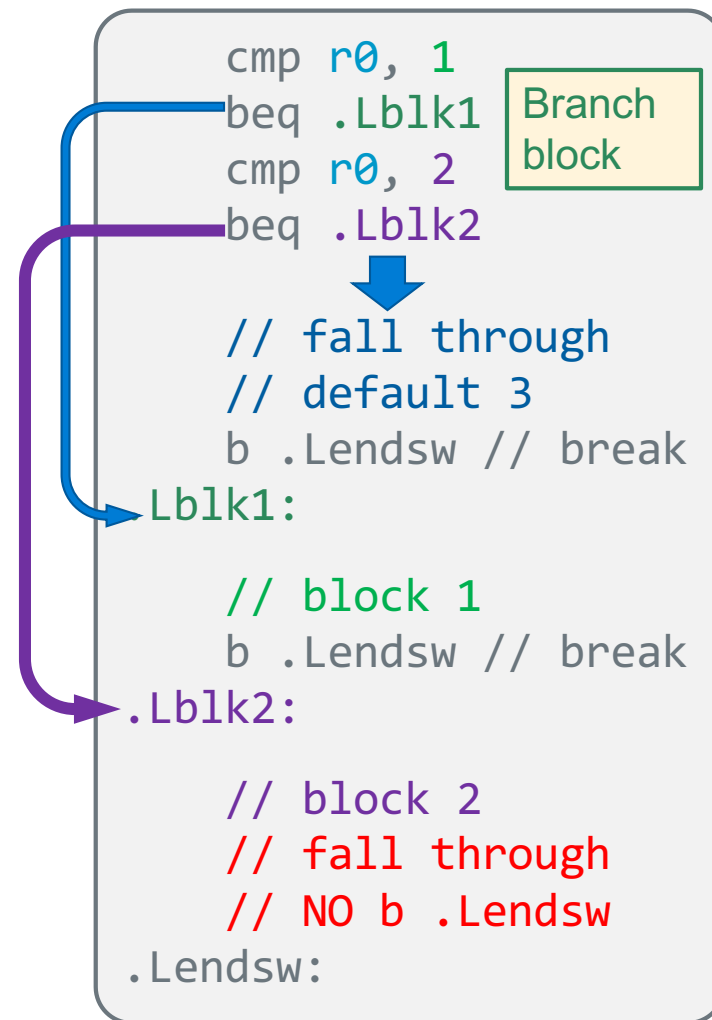
condition
false block

Preserving the same branch guard test



Switch Statement

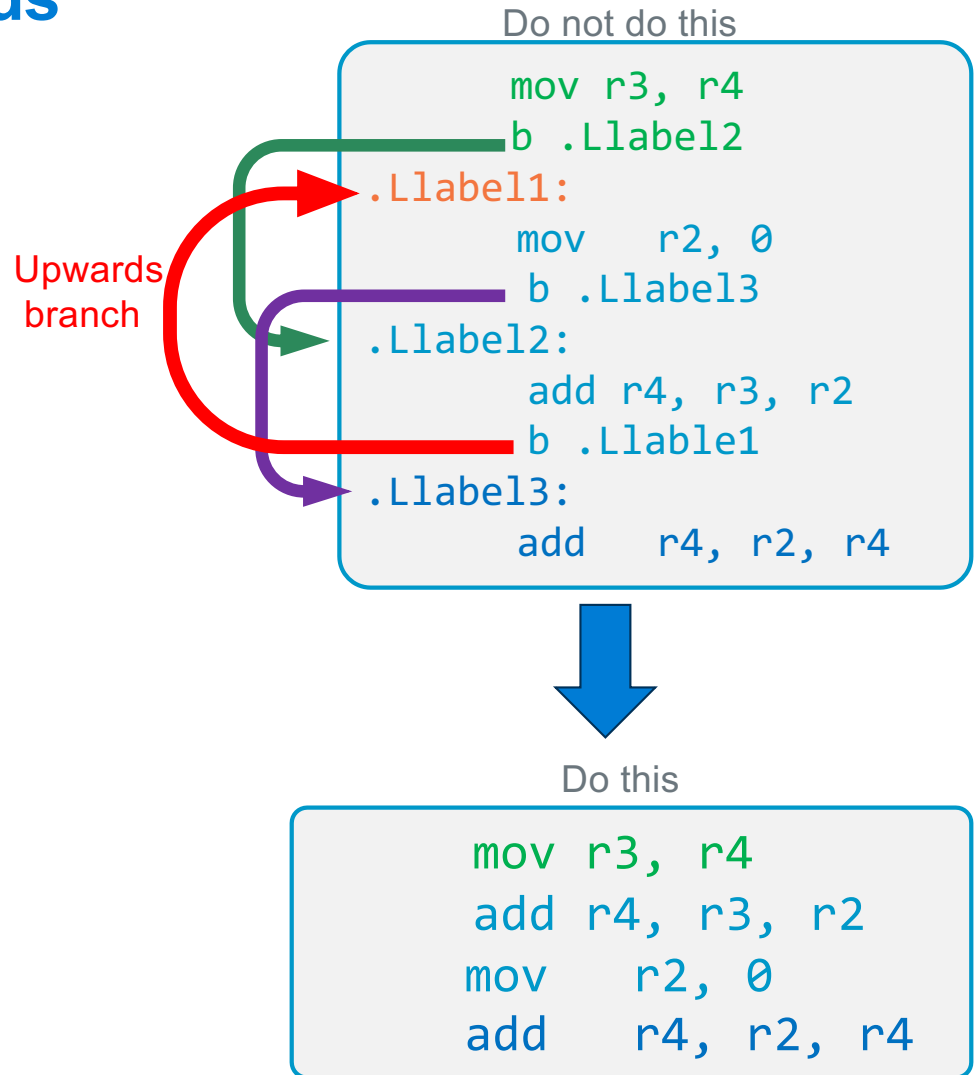
```
switch (r0) {  
  case 1:  
    // block 1  
    break;  
  case 2:  
    // block 2  
    break;  
  default:  
    // default 3  
    break;  
}
```



Bad Style: Branching Upwards (When **Not a loop**)

Do not Branch "Upwards" unless it is part of a loop (later slides)

- If you cannot easily write the equivalent C code for your assembly code, you may have code that is harder to read than it should be
- **Action:** adjust your assembly code to have a similar structure as an equivalent version written in C



Review – Short Circuit or Minimal Evaluation

- In evaluation of conditional guard expressions, C uses what is called **short circuit** or **minimal evaluation**

```
if ((x == 5) || (y > 3)) // if x == 5 then y > 3 is not evaluated
```

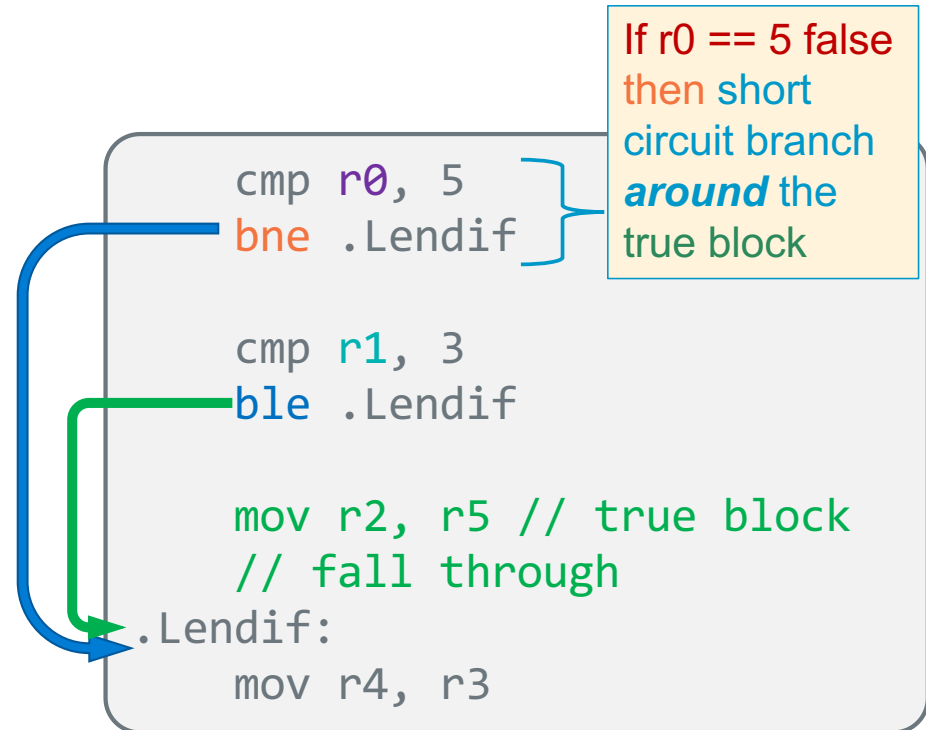


- Each expression argument is evaluated **in sequence** from left to right including any **side effects** (modified using parenthesis), **before** (optionally) evaluating the next expression argument
- If after evaluating an argument, the **value of the entire expression can be determined**, then the **remaining arguments are NOT evaluated (for performance)**

```
if ((a != 0) && func(b)) // if a is 0, func(b) is not called
    // do something
```

Program Flow – If statements && compound tests - 1

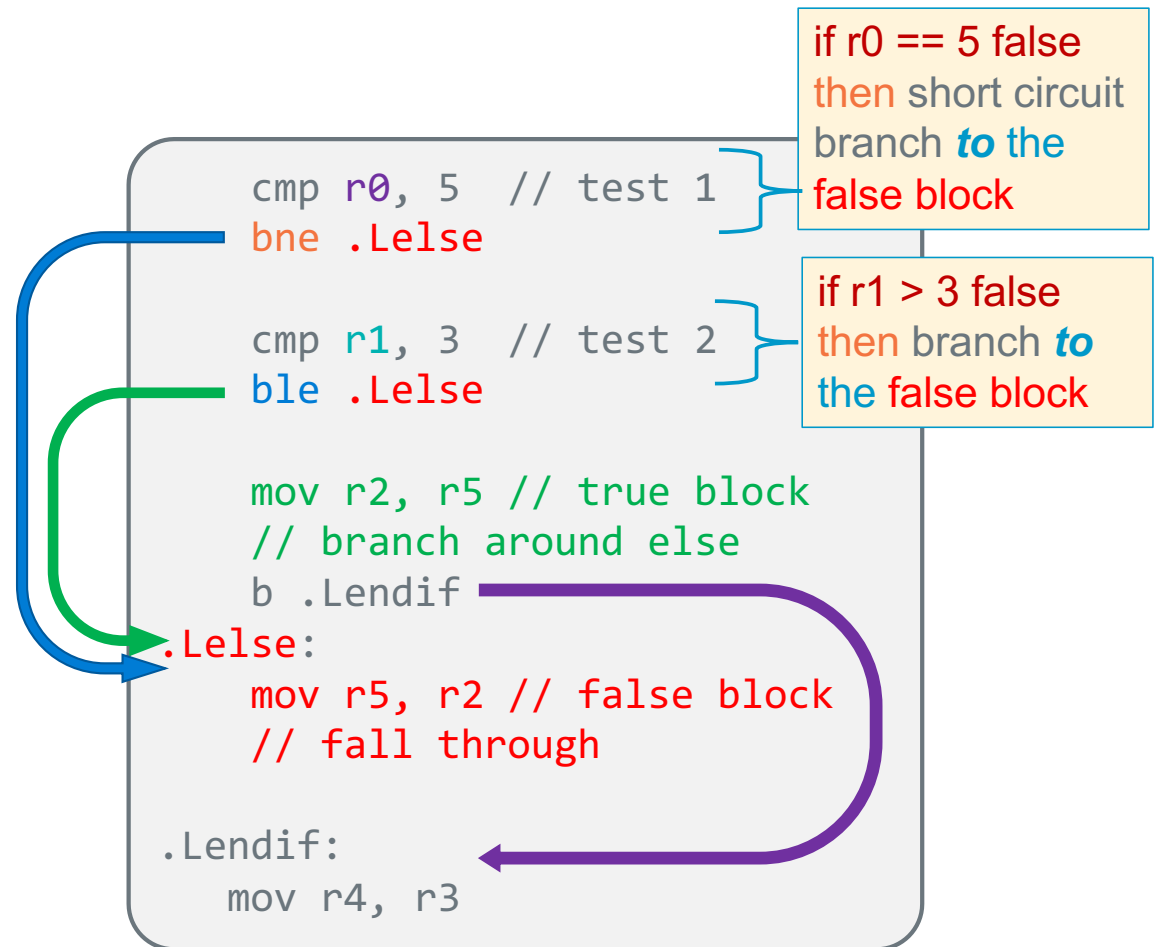
```
if ((r0 == 5) && (r1 > 3)) {  
    r2 = r5; // true block  
    /* fall through */  
}  
r4 = r3;
```



Program Flow – If statements && compound tests - 2

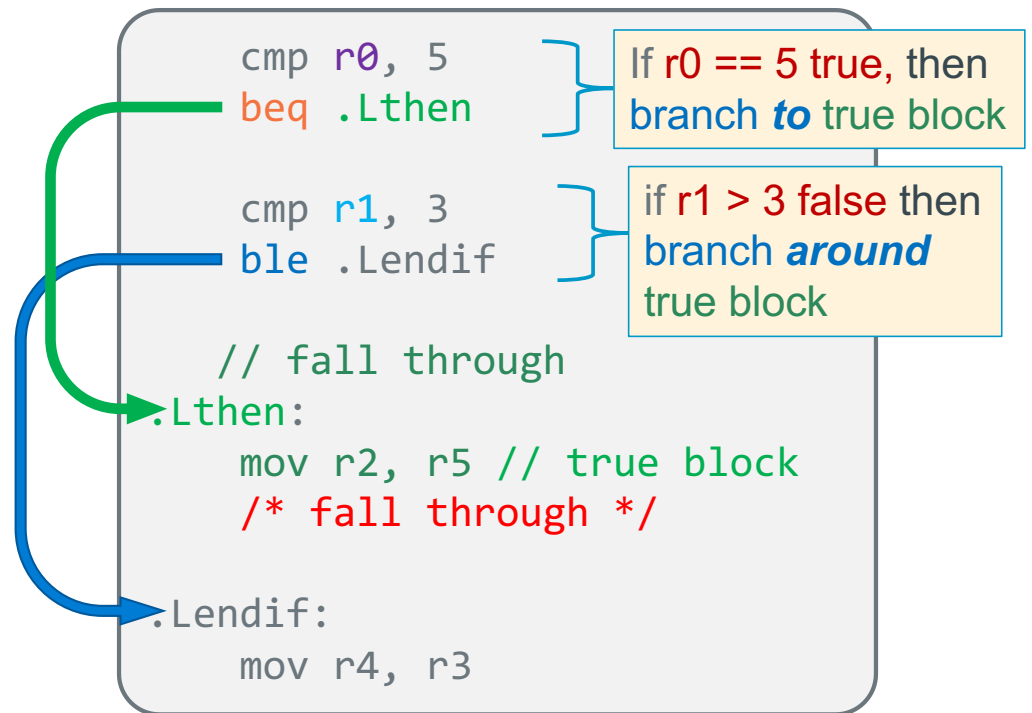
test1 test2

```
if ((r0 == 5) && (r1 > 3))
{
    r2 = r5; // true block
    // branch around else
} else {
    r5 = r2; False block */
    /* fall through */
}
r4 = r3;
```



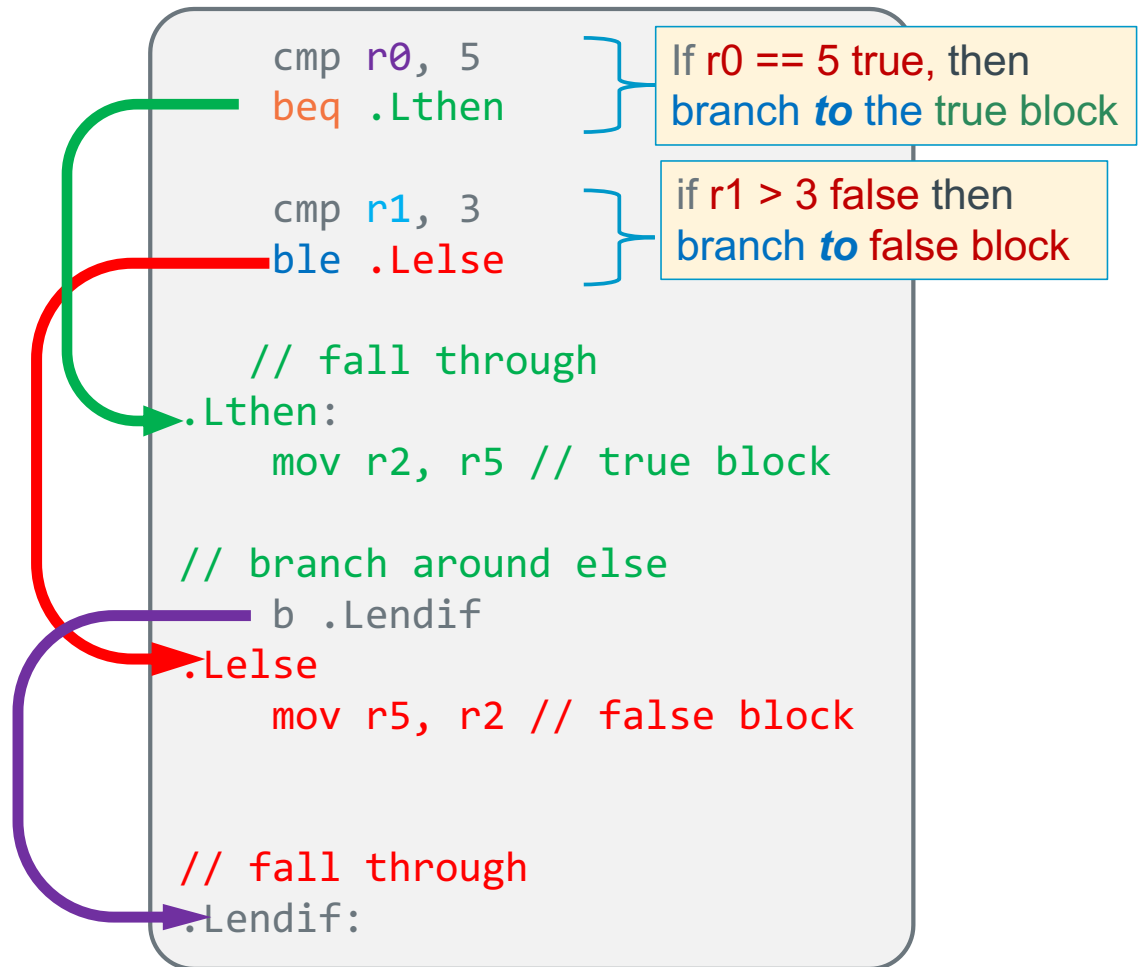
Program Flow – If statements || compound tests - 1

```
if ((r0 == 5) || (r1 > 3)) {  
    r2 = r5; // true block  
    /* fall through */  
}  
r4 = r3;
```



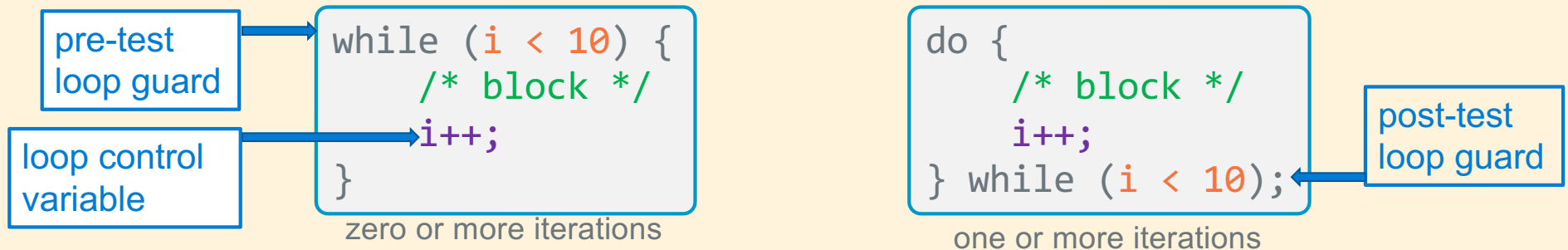
Program Flow – If statements || compound tests - 2

```
if ((r0 == 5) || (r1 > 3)) {  
    r2 = r5; // true block  
    /* branch around else */  
} else {  
    r5 = r2; // false block  
    /* fall through */  
}
```



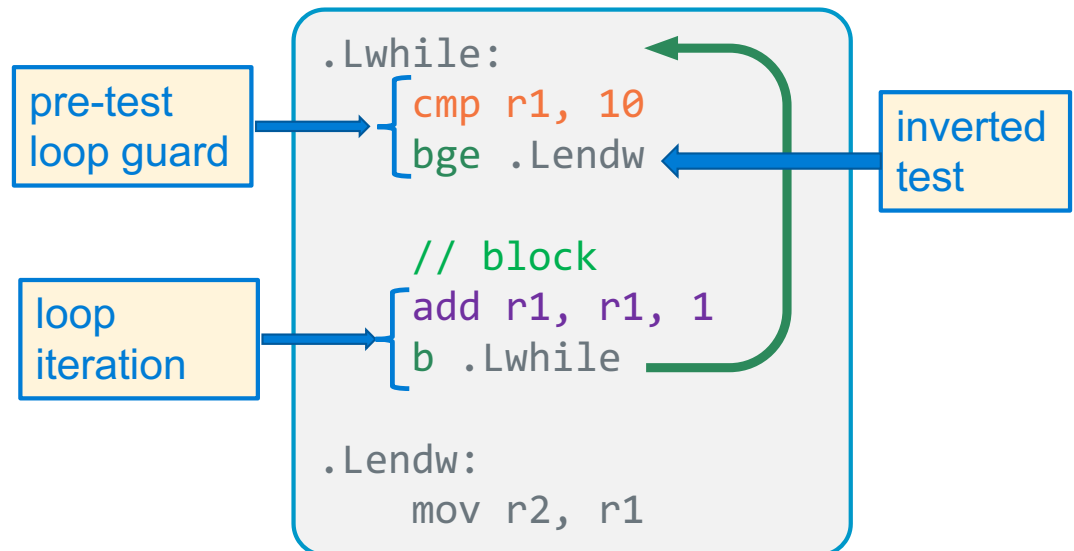
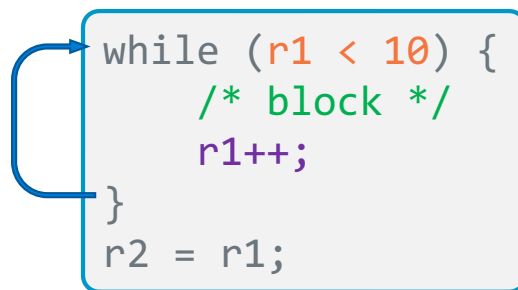
Program Flow – Pre-test and Post-test Loop Guards

- loop guard: code that must evaluate to true before the next iteration of the loop
- If the loop guard test(s) evaluate to true, the *body of the loop* is executed again
- pre-test loop guard is at the top of the loop
 - If the test evaluates to true, execution falls through to the loop body
 - if the test evaluates to false, execution branches around the loop body

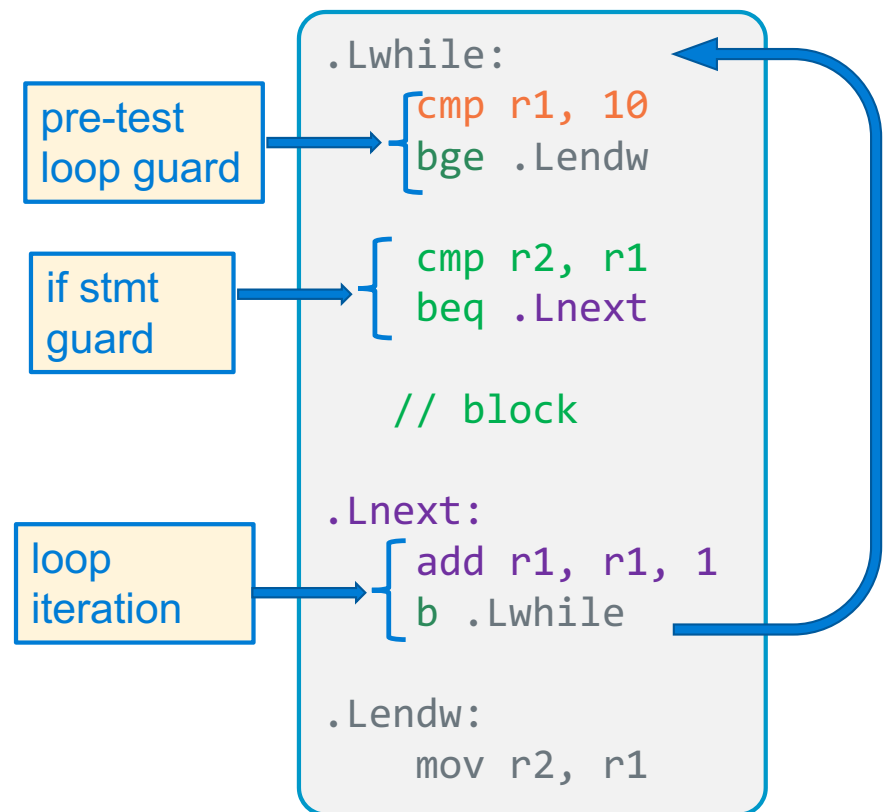
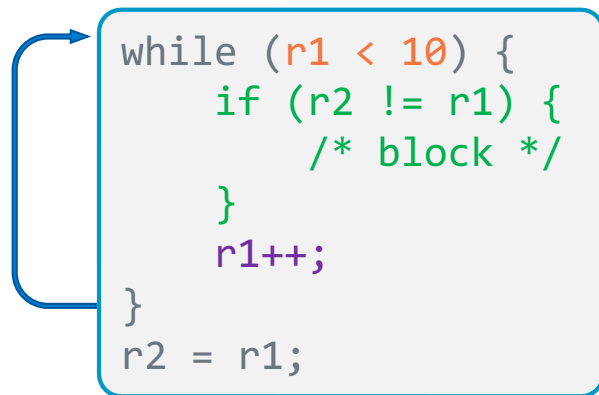


- post-test loop guard is at the bottom of the loop
 - If the test evaluates to true, execution branches to the top of the loop
 - If the test evaluates to false, execution falls through the instruction following the loop

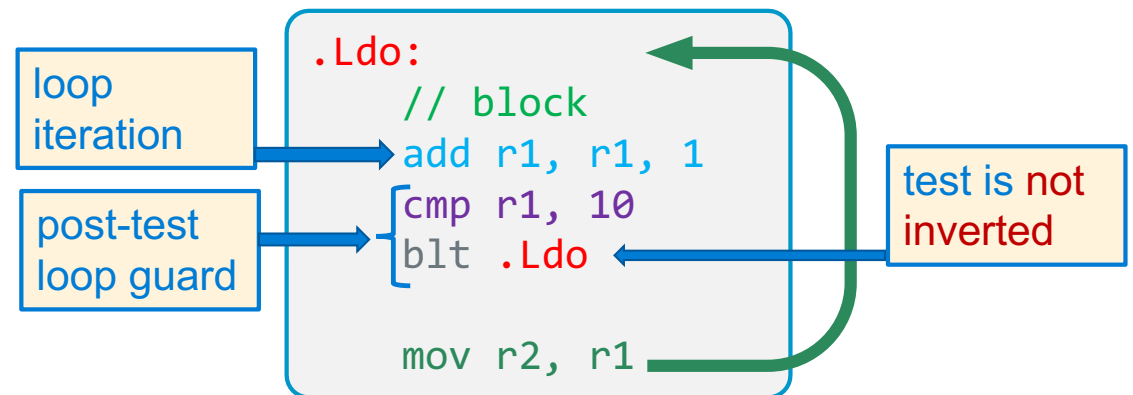
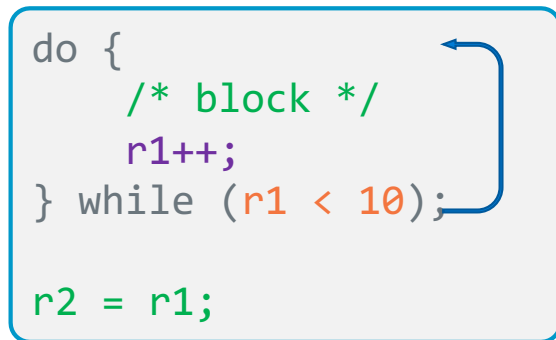
Pre-Test Guards - While Loop



Pre-Test Guards - While Loop

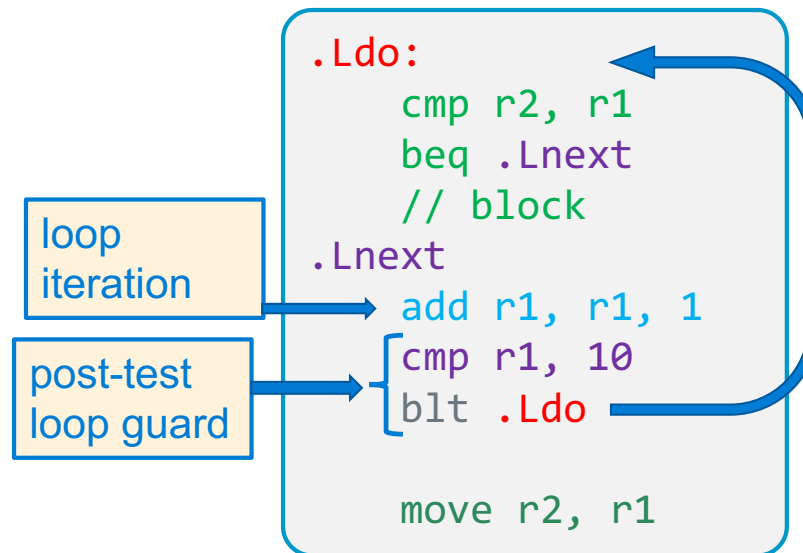


Post-Test Guards – Do While Loop

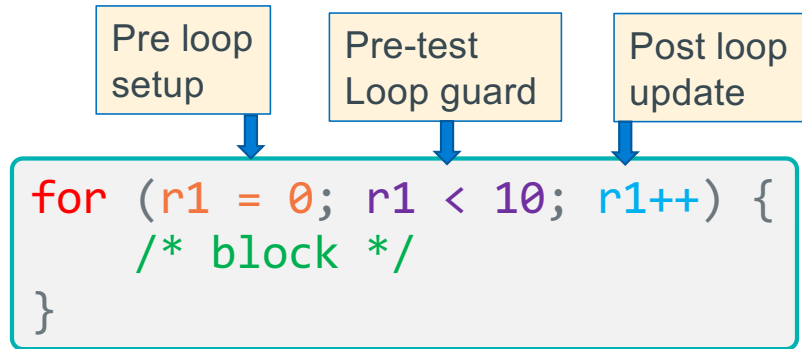


Post-Test Guards – Do While Loop

```
do {  
    if (r2 != r1) {  
        /* block */  
    }  
    r1++;  
} while (r1 < 10);  
r2 = r1;
```

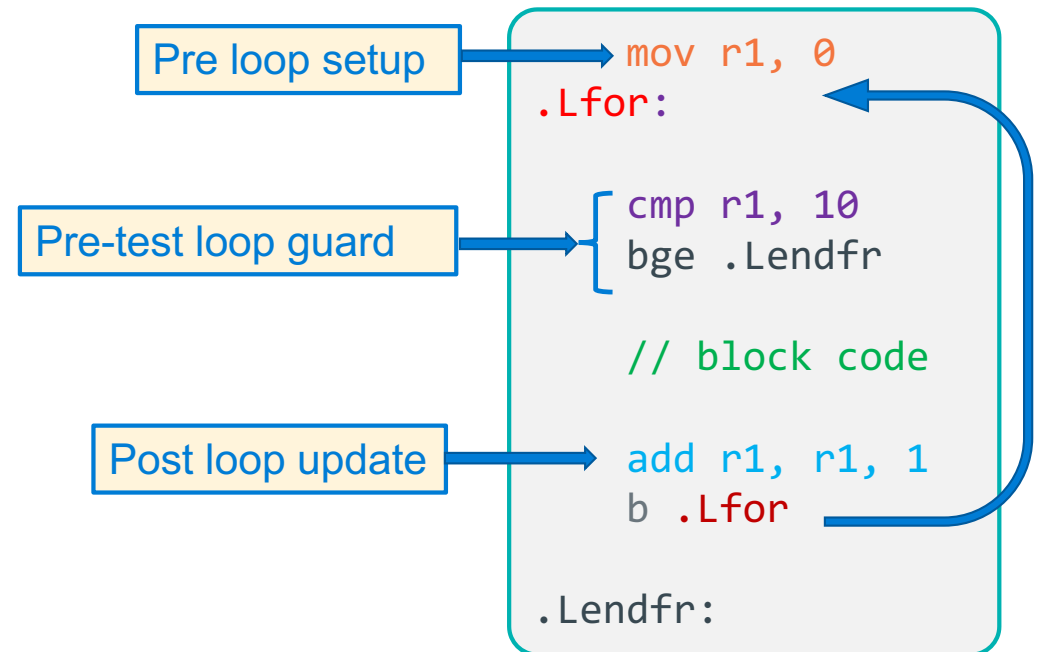


Program Flow – Counting (For) Loop Version 1

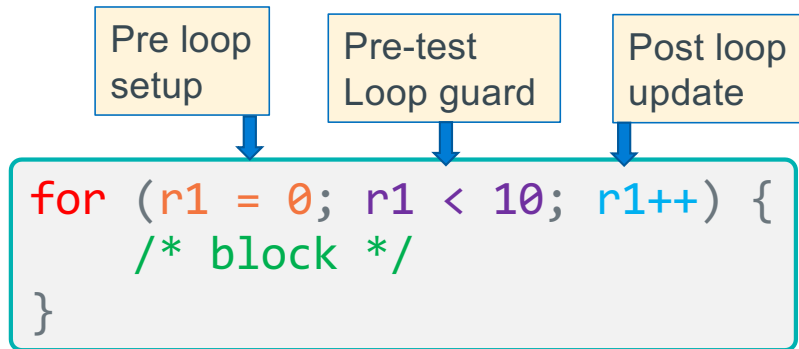


A **counting loop** has three parts:

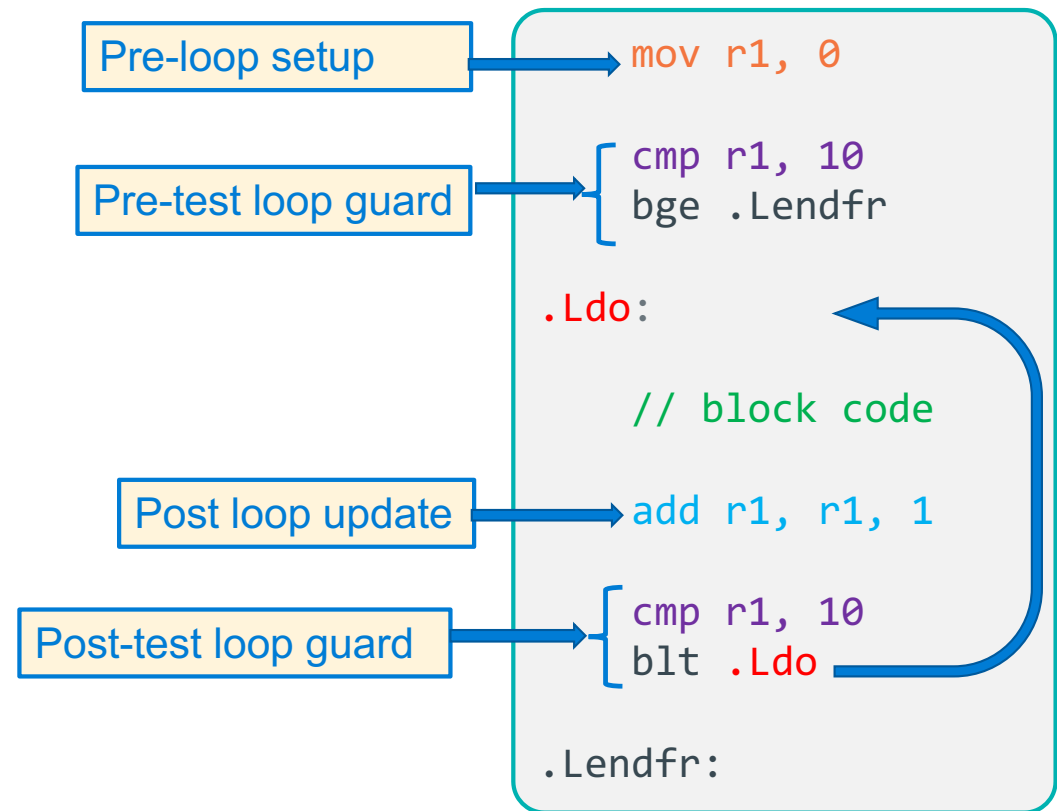
1. Pre-loop setup
2. Pre-test loop guard conditions
3. Post-loop update



Program Flow – Counting (For) Loop – Version 2




- Alternative:
- **move** Pre-test loop guard before the loop
- **Add** post-test loop guard
 - *converts* to *do while*
 - **removes** an **unconditional branch**



Nested loops

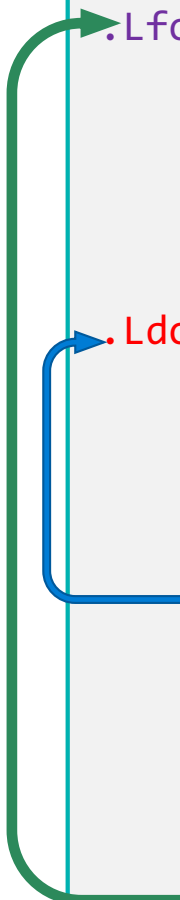
```
for (r3 = 0; r3 < 10; r3++) {  
    r0 = 0;  
  
    do {  
        r0 = r0 + r1++;  
    } while (r1 < 10);  
  
    // fall through  
    r2 = r2 + r1;  
}
```



r5 = r0;

- Nest loop blocks as you would in C or Java

```
mov r3, 0  
.Lfor:  
    cmp r3, 10      // loop guard  
    bge .Lendfor  
  
    mov r0, 0  
  
    .Ldo:  
        add r0, r0, r1  
        add r1, r1, 1  
  
        cmp r1, 10  // loop guard  
        blt .Ldo  
  
        // fall through  
        add r2, r2, r1  
  
        add r3, r3, 1 // loop iteration  
        b .Lfor  
    .Lendfor:  
        mov r5, r0
```



Keep loops Properly Nested: Do not branch into the middle of a loop

- It is hard to understand and debug loops when you **branch into the middle of a loop**
- **Keep loops proper nested**

Bad practice: branch into loop body

Do not do the following:

```
.Lloop1:
    add r1, r1, 1
.Lloop2:
    add r2, r2, 1
    add r2, r1, r3
    cmp r1, 10
    blt .Lloop1
    beq .Lend1
    add r3, r3, 1
    cmp r2, 20
    ble .Lloop2 ←
.Lend1:
```

