

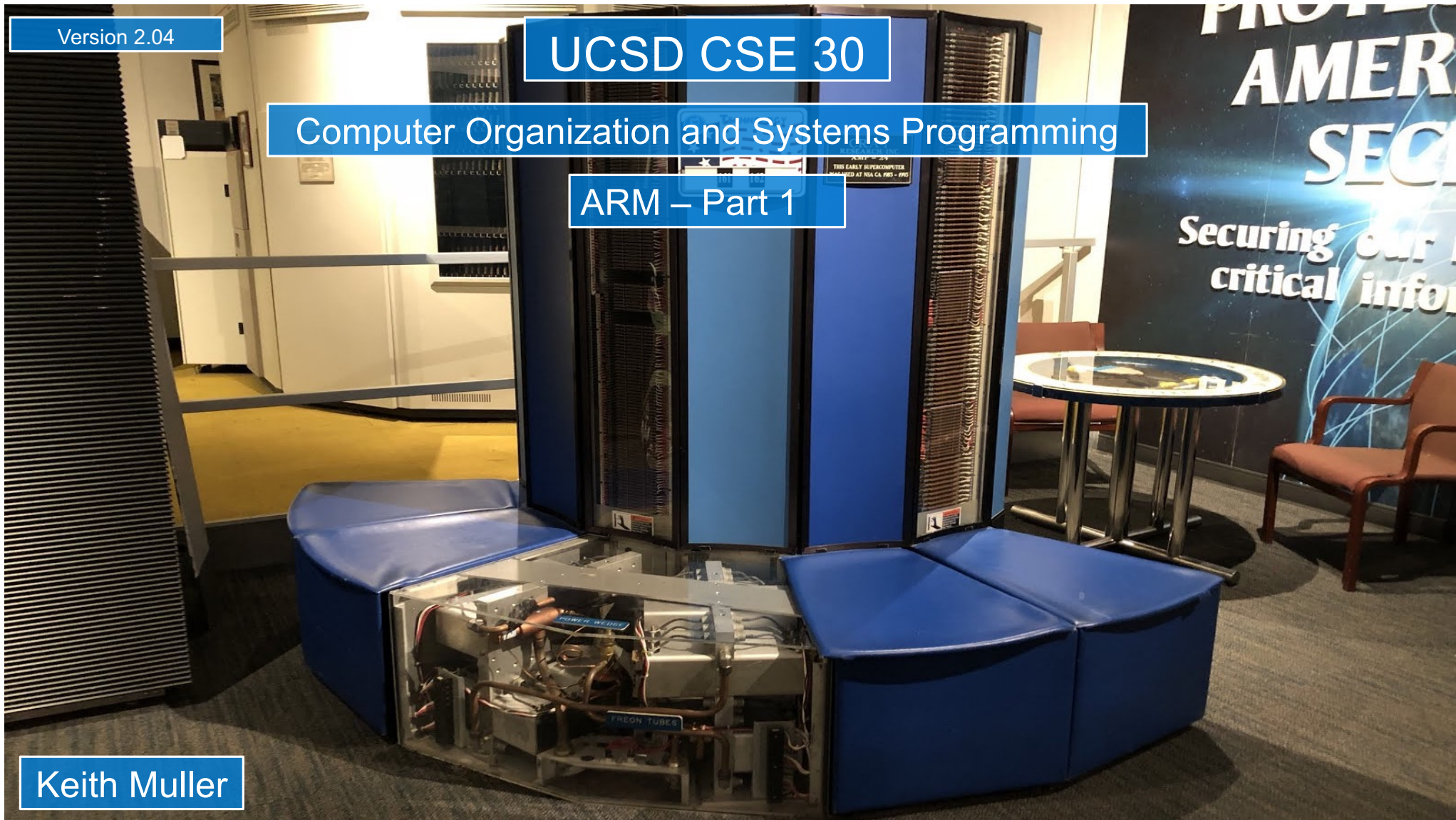
Version 2.04

UCSD CSE 30

Computer Organization and Systems Programming

ARM – Part 1

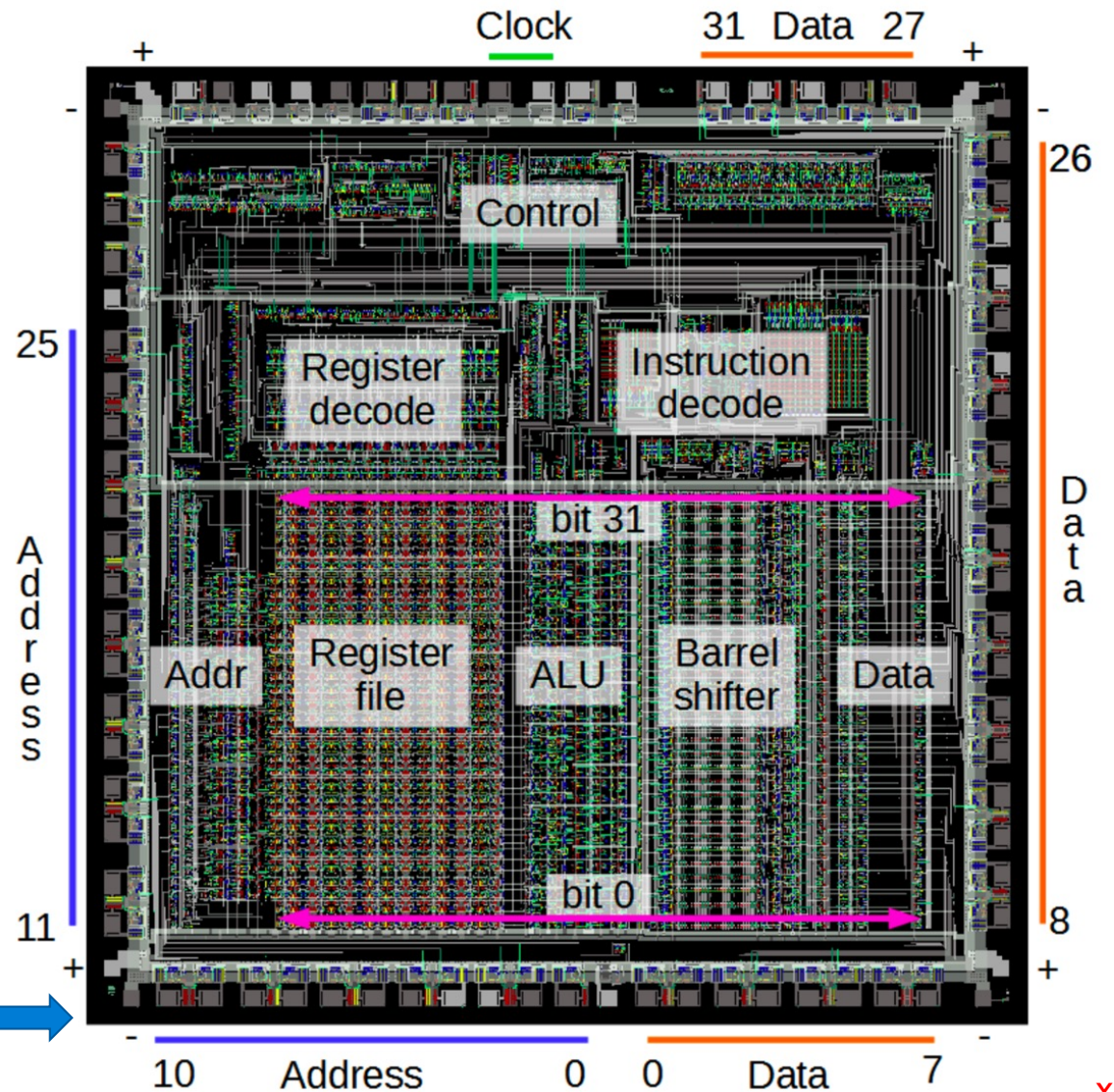
Keith Muller



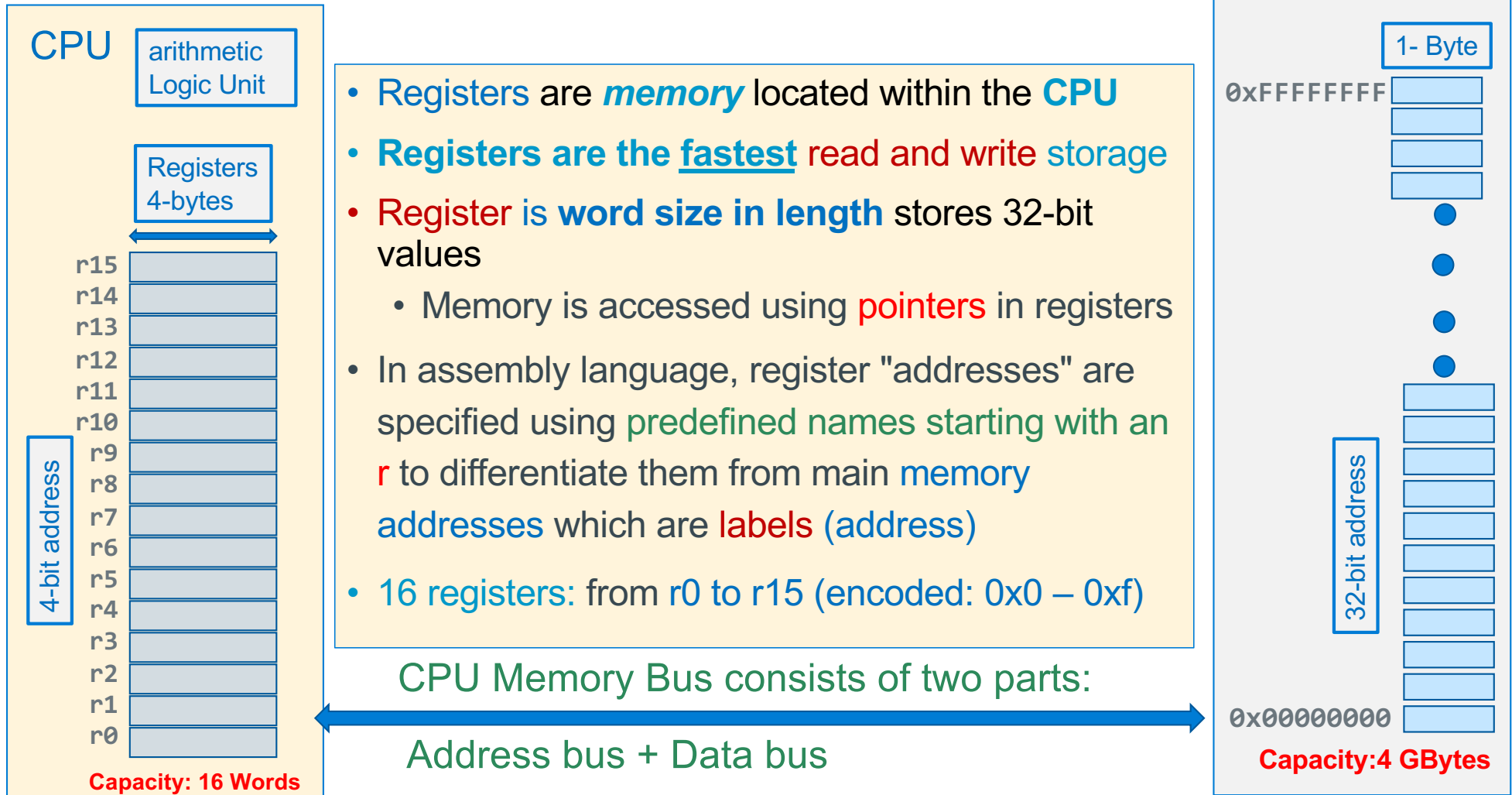
Arm Core Floorplan

- **Control:** Specifies the operation of the CPU
- **Register File:** Memory inside the CPU
 - Instructions reference these directly
- **ALU:** Arithmetic Logic Unit: Arithmetic and bitwise hardware (on the bits)
- **Barrel shifter:** (shifts bits in a register during instruction execution - Later)
- **Instruction Decode:** Interprets the the bits in an instruction to determine what the instruction means
- **Register Decode:** controls the registers in during instruction execution
- **Address and Data:** Interface to external RAM (like memory dimms)

Single core *arm* die *Floorplan*



32-Bit Arm - Registers



Using Arm-32 Registers

- There are two basic groups of registers, **general purpose** and **special use**
- **General purpose registers** can be used to contain up to 32-bits of data, but you must follow the **rules** for their use
 - Rules specify how registers are to be used so software can **communicate** and share the use of registers (later slides)
- Special purpose registers: have a dedicated hardware use (r15 the pc) or special use when used with certain instructions (r13 & r14)
- r15/pc is the program counter that contains the address of an instruction being executed (not exactly ... later)

Special Use Registers
program counter

r15/pc

Special Use Registers
function call implementation
& long branching

r14/lr

r13/sp

r12/ip

r11/fp

Preserved registers
Called functions **can't change**

r10

r9

r8

r7

r6

r5

r4

Scratch Registers
First 4 Function Parameters
Function return value
Called functions **can change**

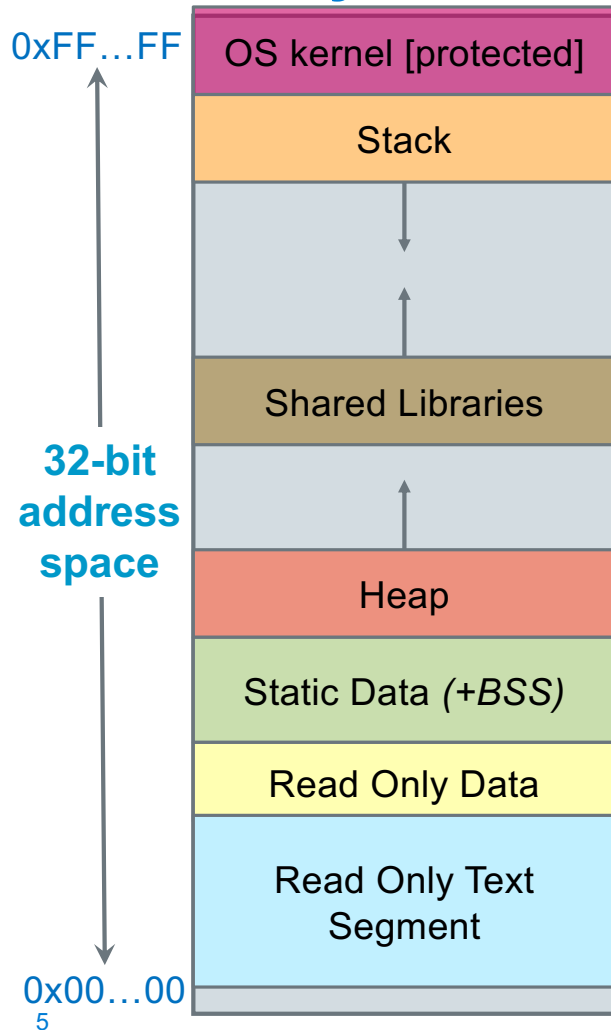
r3

r2

r1

r0

Assembly and Machine Code



- Machine Language (or code): Set of instructions the CPU executes are encoded in memory using patterns of ones and zeros (like binary numbers)
- Assembly language is a **symbolic version** of the machine language
- Each assembly statement (called an **Instruction**)
 - Executes **exactly one** from a list of simple commands
 - Instructions describe operations (e.g., =, +, -, *)
 - Execution proceeds from low to high memory one instruction at a time unless there is a branch
- One line of **arm32 machine code** contains one instruction in one word (32 bits)

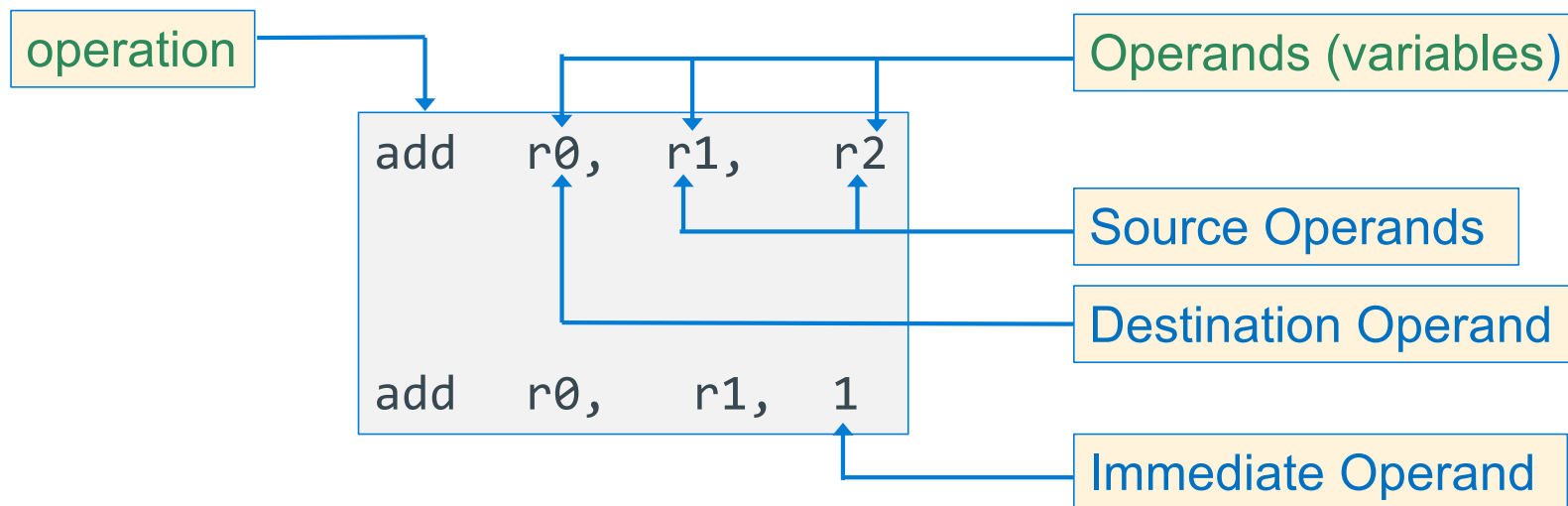
Instruction Address	Instruction (4-bytes) contents	Assembly Equivalent
1040c:	e28db004	add fp, sp, 4
10410:	e59f0010	ldr r0, [pc, 16]
10414:	ebffffb3	bl 102e8 <printf>
10418:	e3a00000	mov r0, 0
1041c:	e24bd004	sub sp, fp, 4

high <- low bytes

Machine Code

Anatomy of an Assembly instruction (3 address instruction)

- Assembly language instructions specify an **operation** and the **operands** to the instruction (arguments of the operation)
- Three basic types of **operands**
 - **Destination**: where the result will be stored
 - **Source**: where data is read from
 - **Immediate**: an actual value like the **1** in $y = x + 1$



Meaning of an Instruction - ARM

- Operations are abbreviated into **opcodes** (1– 5 letters)
- **Assembly Instructions** are specified with a rigid syntax
 - **Opcodes** are followed by **arguments**
 - Usually the **destination argument is next**, then **one or more source arguments** (this is not strictly the case, but it is generally true)
- Why this order?
- Analogy to C or Java

```
int r0, r1, r2;  
r0 = r1 + r2;           // C
```

```
add    r0, r1, r2      // assembly
```

32-Bit Arm - Registers

All computations (add, subtract, etc.) are performed in the ALU

Arithmetic & Logic Unit (ALU)

registers

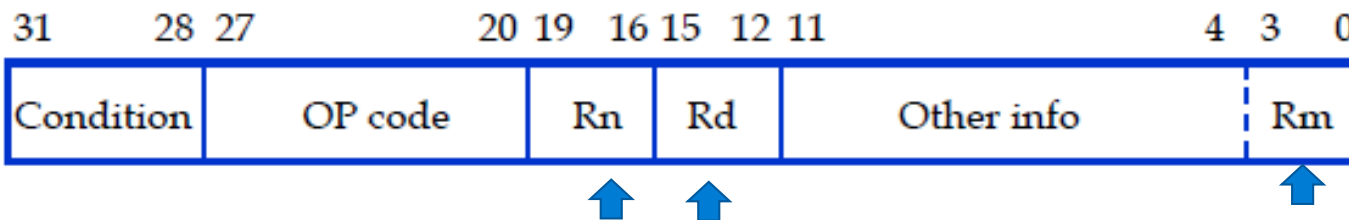
4-bytes

r15
r14
r13
r12
r11
r10
r9
r8
r7
r6
r5
r4
r3
r2
r1
r0

4-bit address

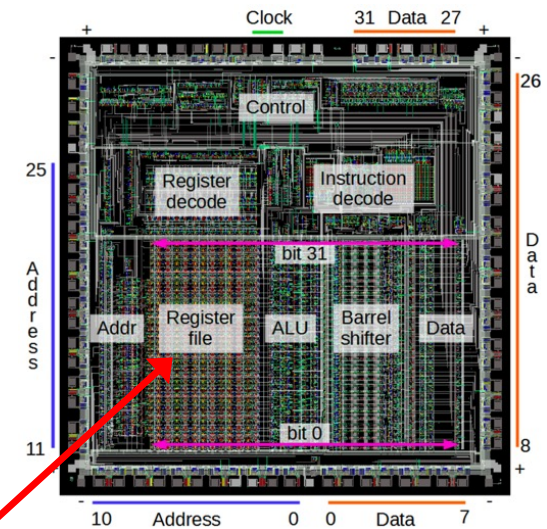
Capacity: 16 Words

- Almost all arithmetic, logic operations and data movement operations involve at least one register
- As a result, Register addresses are **directly** encoded into 4-bit fields in machine instructions (see below)



Program Execution: A Series of Instructions

- Instructions are **retrieved sequentially** from memory
- Each instruction **executes to completion before** the next instruction is completed
- Conceptually the pc (program counter) points at executing instruction
- exceptions: loops, function calls, traps,...



Memory Content in Text segment

add r0, r1, r1 Low memory

add r0, r0, r0

add r0, r0, r0

sub r1, r0, r1 High memory

Register contents inside the CPU

r0 = 1 r1 = 2 initial values

r0 = 4 r1 = 2

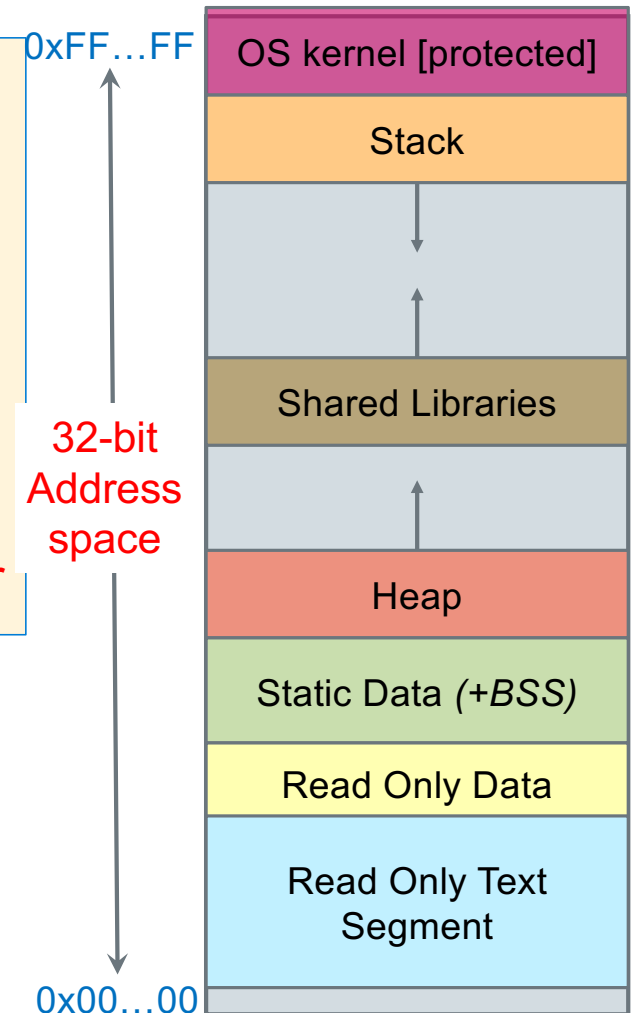
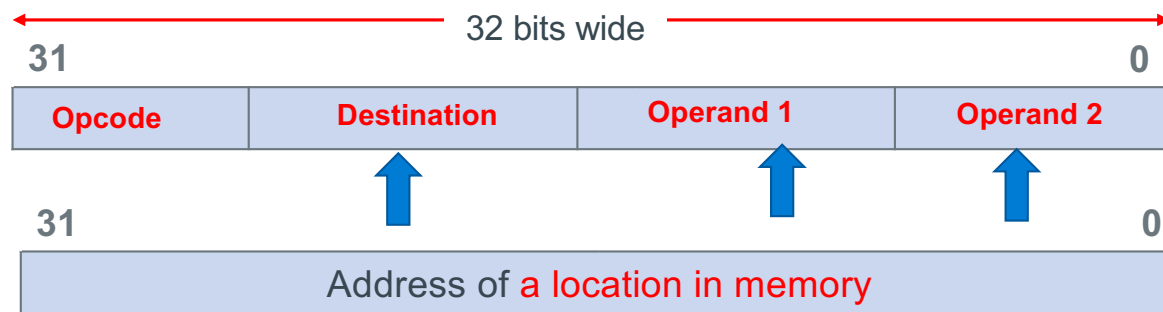
r0 = 8 r1 = 2

r0 = 16 r1 = 2

r0 = 16 r1 = 14

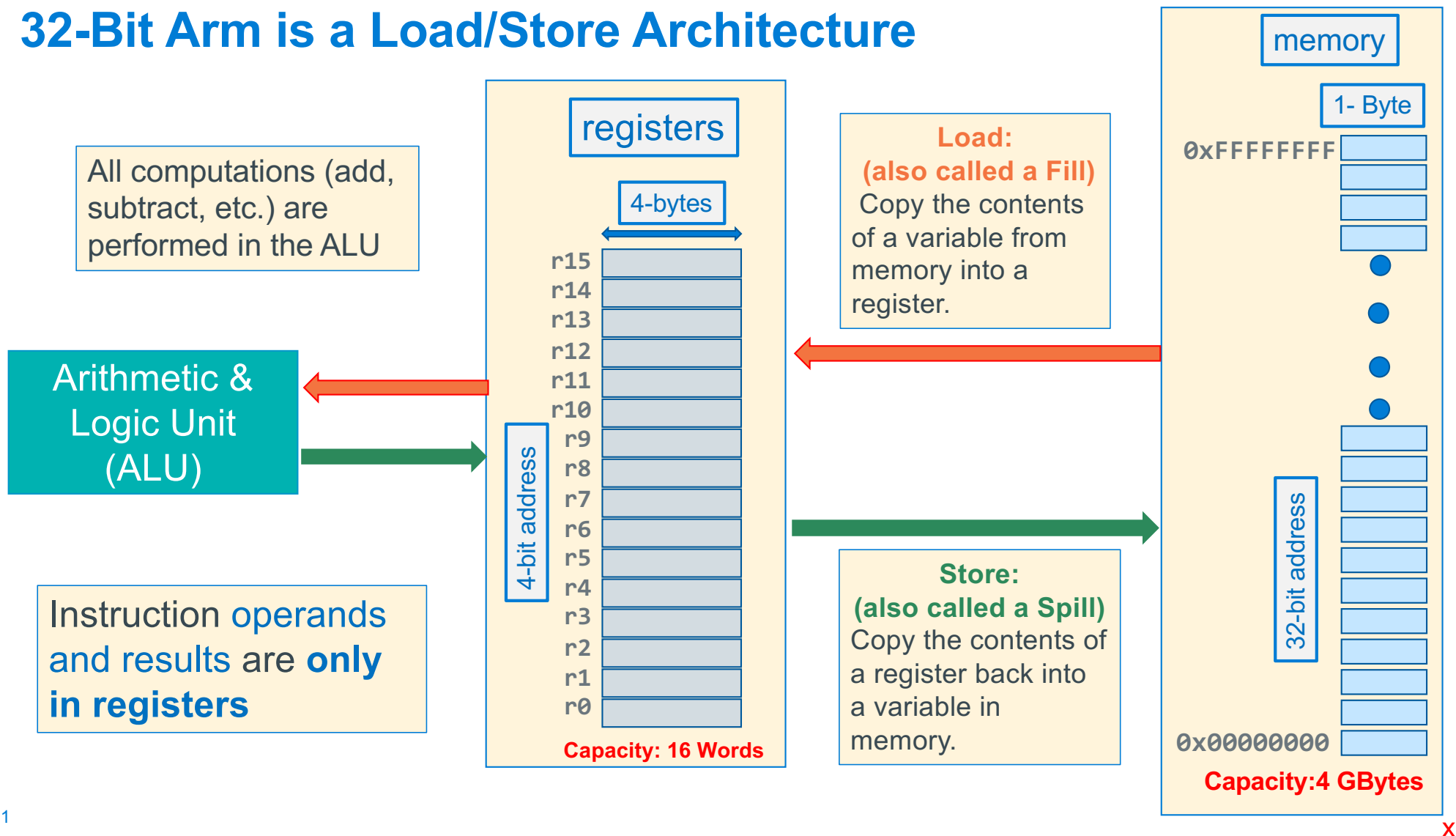
How to Access Memory?

- Consider $a = b + c$ are operands are in memory
 - Operation code: add Destination: a
 - Operand 1: b Operand 2: c
- Aarch32 Instructions are always word size: 32 bits wide
 - Some bits must be used to specify the operation code
 - Some bits must be used to specify the destination
 - Some bits must be used to specify the operands
- Address space is 32 bits wide so put a **POINTER** in a register



NOT ENOUGH BITS for FULL Addresses to be stored in the instruction

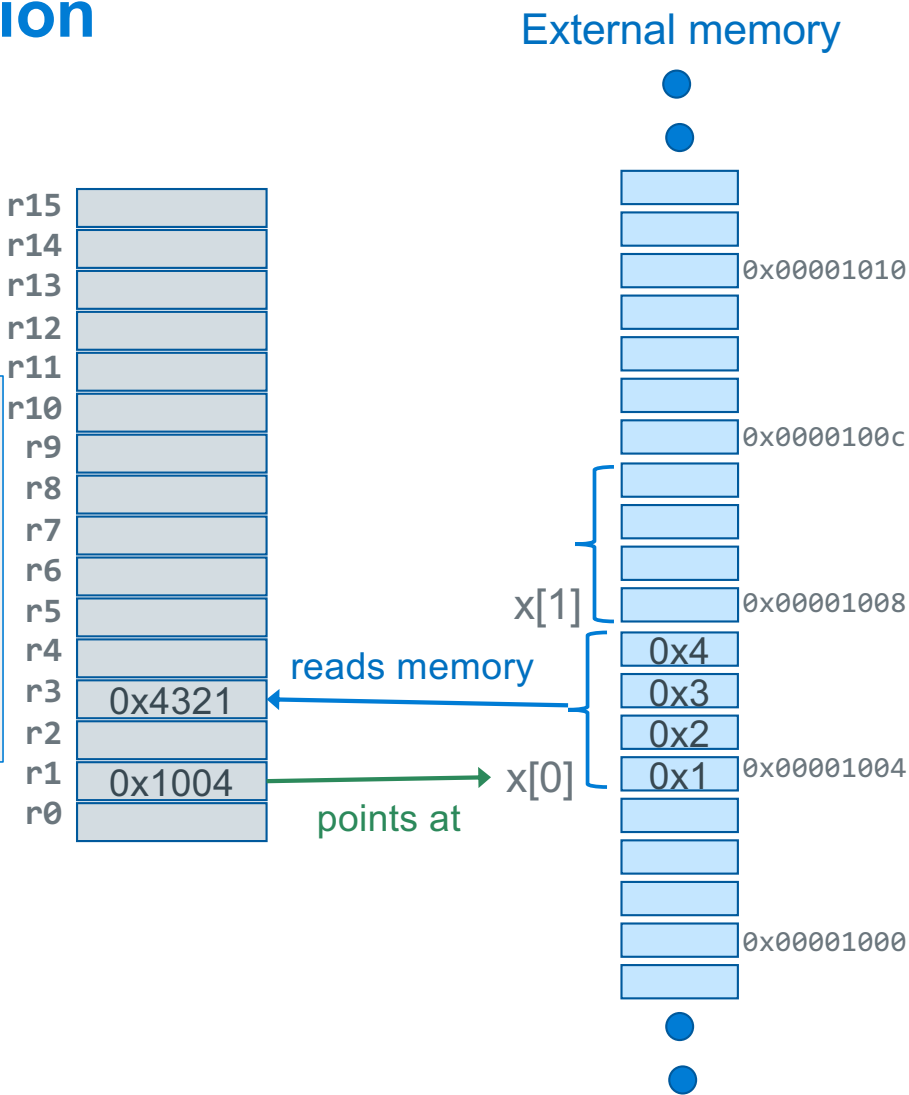
32-Bit Arm is a Load/Store Architecture



Load/Store Concept: Load Operation

```
int x[2] = {0x4321, 0x0};  
x[1] = x[0];
```

```
// load store concept  
  
int r3;  
int x[2];  
int *r1;  
  
r1 = &x;           // r1 contains x's address  
r3 = *(r1);         // read memory , load register 3
```



Load/Store Concept: Store Operation

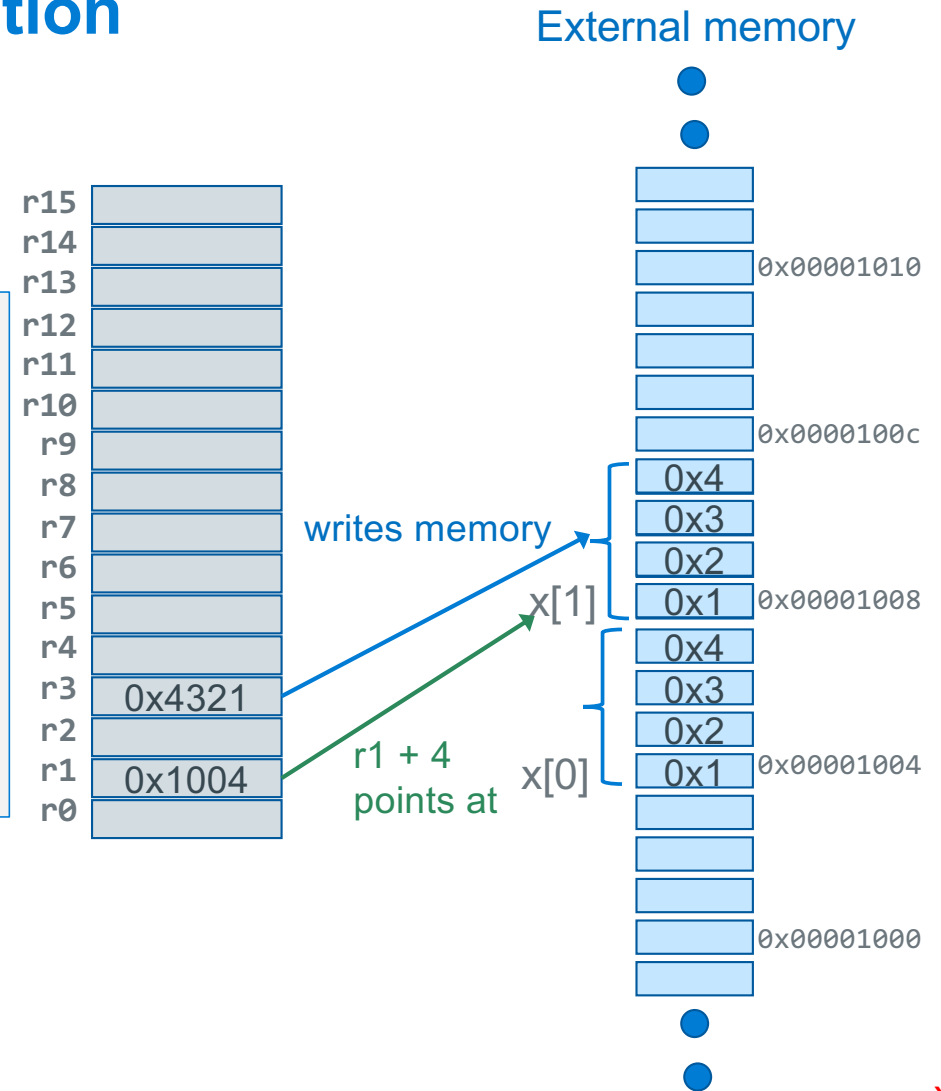
```
int x[2] = {0x4321, 0x0};  
x[1] = x[0];
```

```
// load store concept
```

```
int r3;  
int x[2];  
int *r1;
```

```
r1 = &x;           // r1 contains x's address  
r3 = *(r1);        // read memory , load register 3
```

```
r1 = r1 + 1;  
*(r1) = r3;        // store register 3 to memory
```



Arm Register Summary

- 16 Named registers r0 – r15
- The operands of almost all instructions are registers
- To **operate on a variable in memory** do the following:
 1. Load the value(s) from memory into a register
 2. Execute the instruction
 3. Store the result back into memory (**only if needed!**)
- Going to/from memory is expensive
 - 4X to 20X+ **slower** than accessing a register
- **Strategy:** Keep variables in registers as much as possible

AArch32 Instruction Categories

- **Data movement to/from memory**
 - **Data Transfer Instructions** between memory and registers
 - Load, Store
- **Arithmetic and logic**
 - **Data processing Instructions** (registers only)
 - Add, Subtract, Multiply, Shift, Rotate, ...
- **Control Flow**
 - Compare, Test, If-then, Branch, function calls
- **Miscellaneous**
 - **Traps (OS system calls)**, Breakpoints, wait for events, interrupt enable/disable, data memory barrier, data synchronization barrier
 - Many others that we will not cover in the class

Arithmetic and
logic

Data Movement

Control Flow

Miscellaneous

First Look: Copying Values Between Registers - MOV

```
mov  r0, r1
```

```
// Copies all 32 bits of the  
// value in register r1 into  
// register r0
```

register direct "addressing"

register r1



register r0

```
mov  r0, 100
```

```
// Expands an 8-bit (imm8)  
// value 100  
// stored in the instruction  
// into the register r0
```

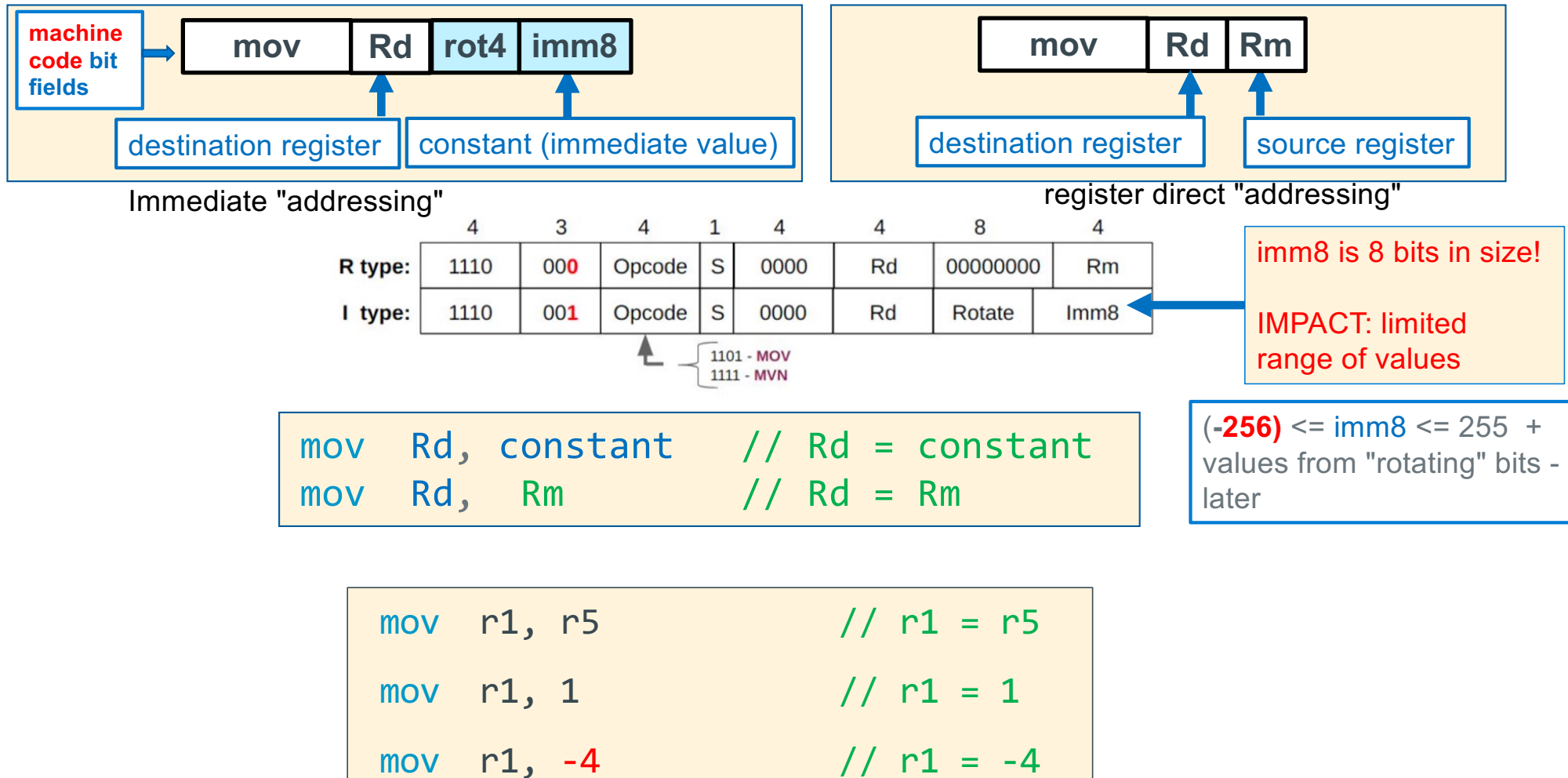
Immediate "addressing"

100

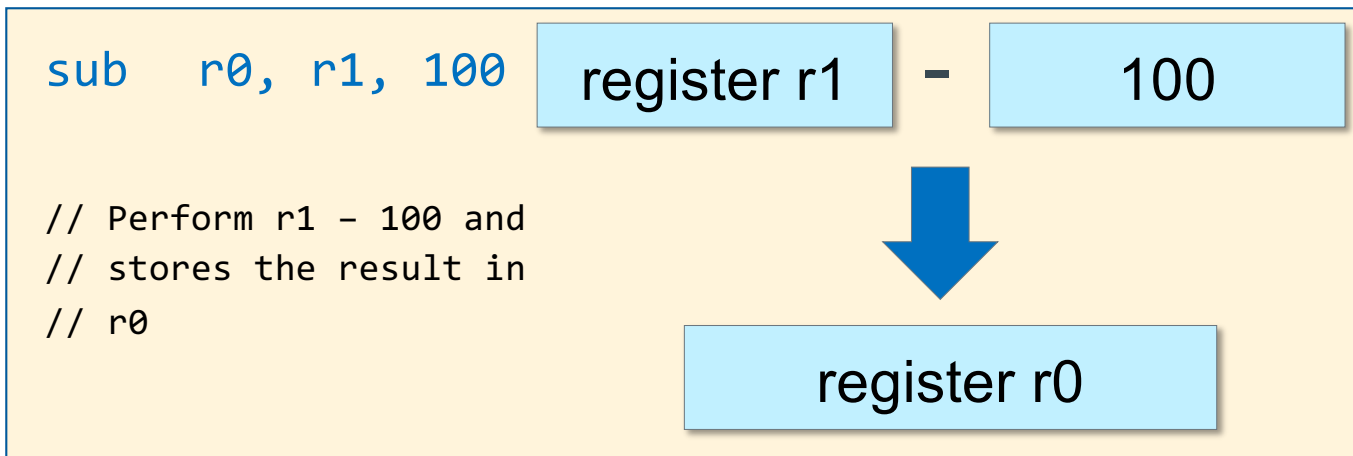
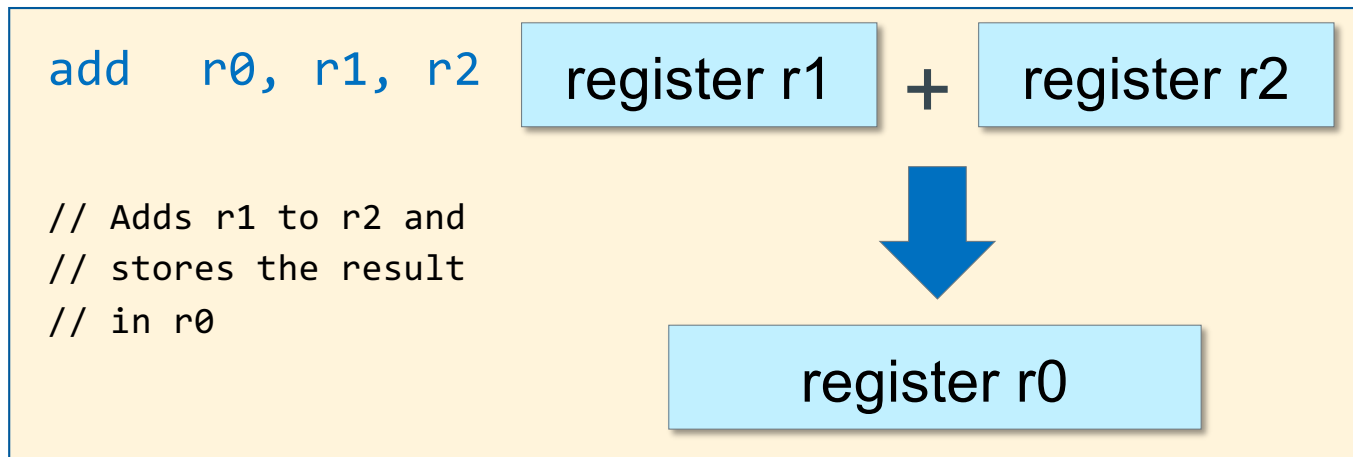


register r0

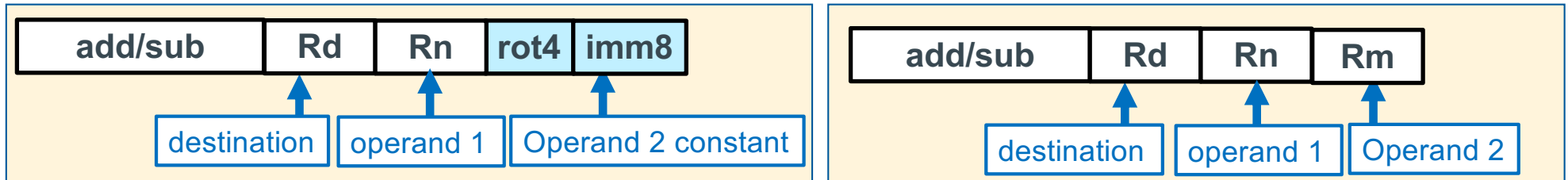
mov – Copies Register Content between registers



First Look: Add/Sub Registers



add/sub – Add or Subtract two integers



```
add  Rd, Rn, constant    // Rd = Rn + constant
sub  Rd, Rn, constant    // Rd = Rn - constant
add  Rd,  Rn, Rm         // Rd = Rn + Rm
sub  Rd,  Rn, Rm         // Rd = Rn - Rm
```

```
add   r1, r2, r3    // r1 = r2 + r3
sub   r1, r1, 1     // r1 = r1 - 1; or r1--
add   r1, r2, 234   // r1 = r2 + 234
```

Writing a Sequence of Add & Subtract Instructions

- You need to perform the following sequence of integer adds/subtracts

$$a = b + c + d - e;$$

- Since ARM uses a **three-operand instruction** set, you can only operate on **two operands** at a time
- So, you need to use **one register** as an **accumulator** and create **a sequence of add instructions** to build up the solution

```
r0 ← a
r1 ← b
r2 ← c
r3 ← d
r4 ← e
```

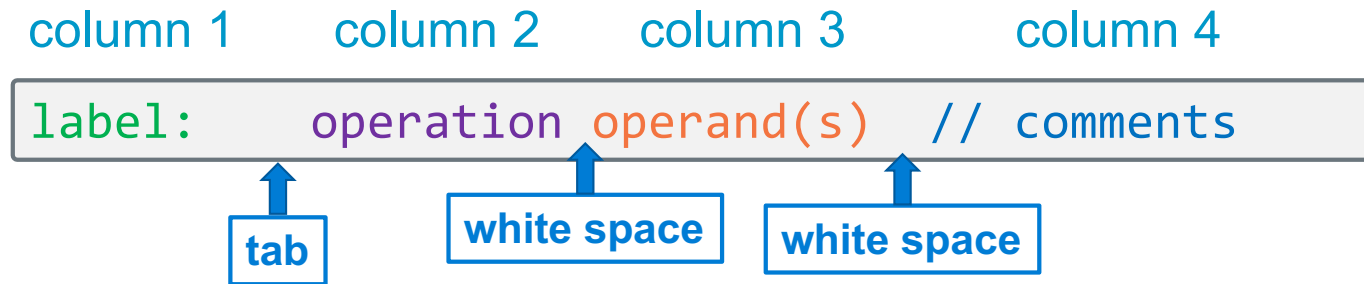
```
a = b + c + d - e;
r0 = r1 + r2 + r3 - r4;
r0 = ((r1 + r2) + r3) - r4;
r0 = r1 + r2;
r0 = r0 + r3
r0 = r0 - r4
```

```
add    r0, r1, r2
add    r0, r0, r3
sub     r0, r0, r4
```

```
a = (b + c) - 5;
r0 = (r1 + r2) - 5;
```

```
add    r0, r1, r2
sub     r0, r0, 5
```

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

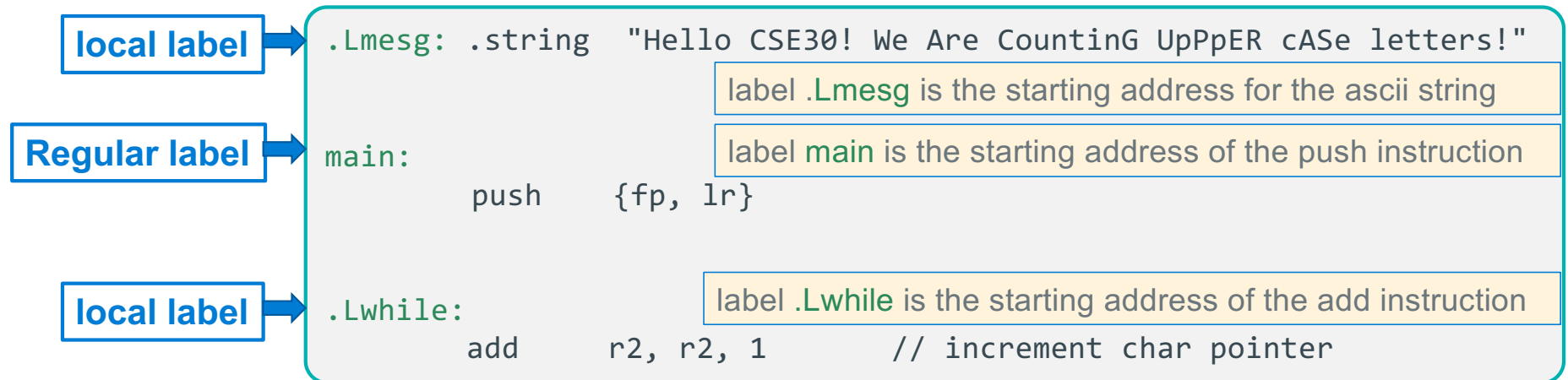
```
label:  operation operand(s)  // comment

        // 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

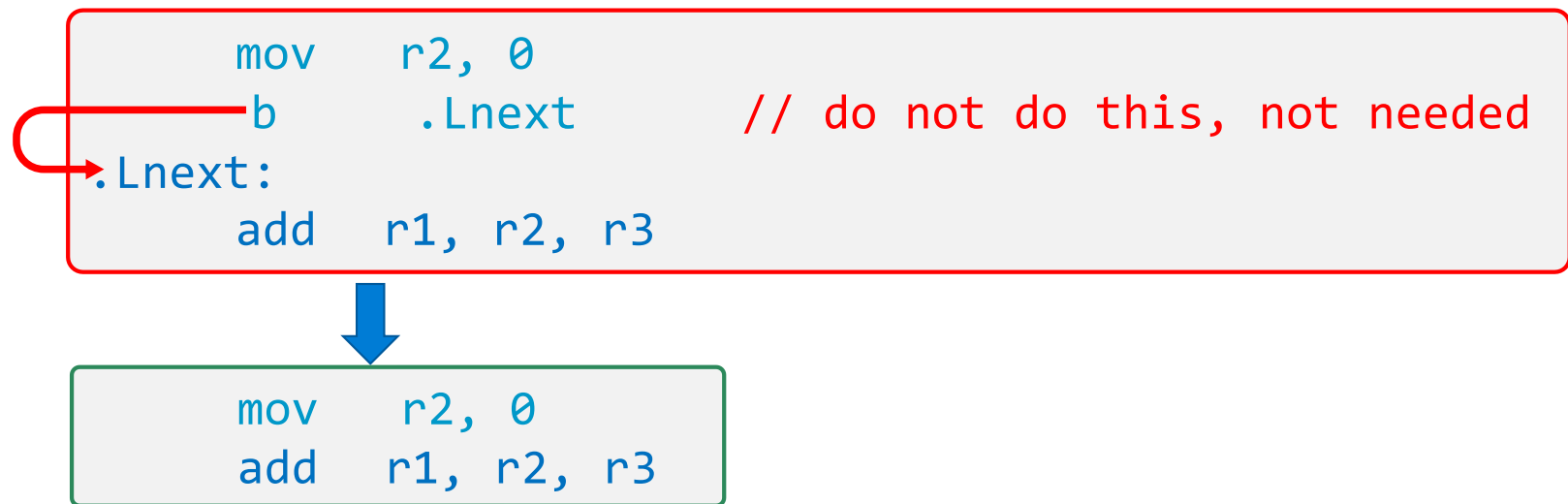
Branch target (local label)

Backward Branch (Infinite loop)

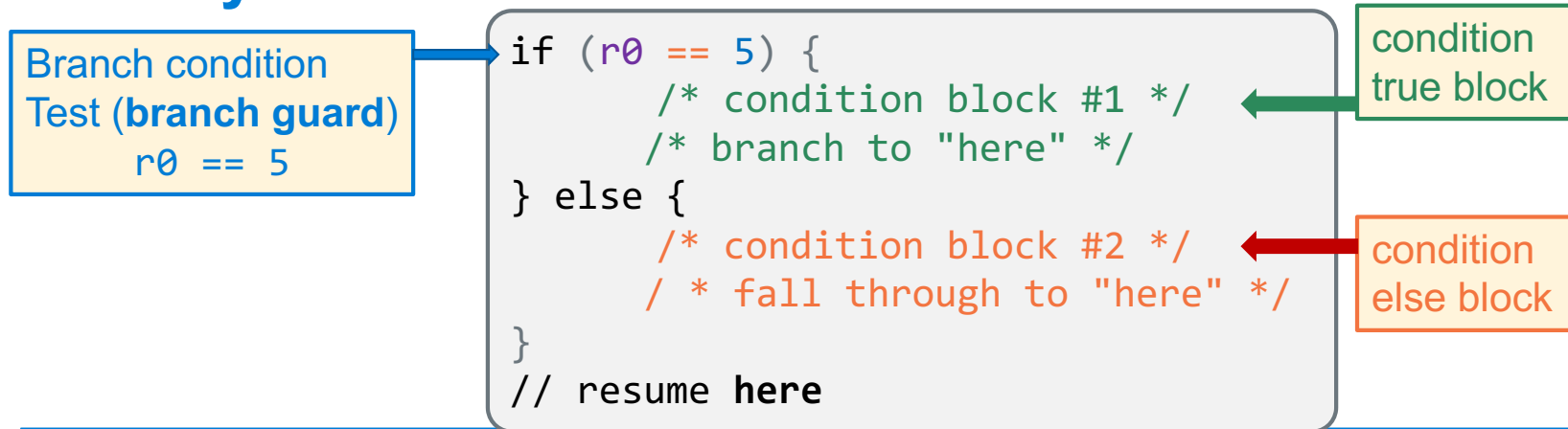
```
.Lbackward:
    add r1, r2, 4
    sub r1, r2, 4
    add r4, r6, r7
    b .Lbackward
// not reachable unless
// there is a label after the .b above
```

- 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!

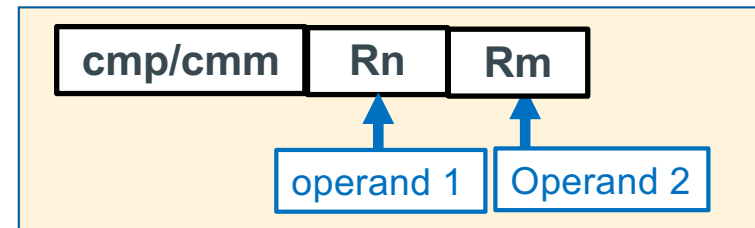
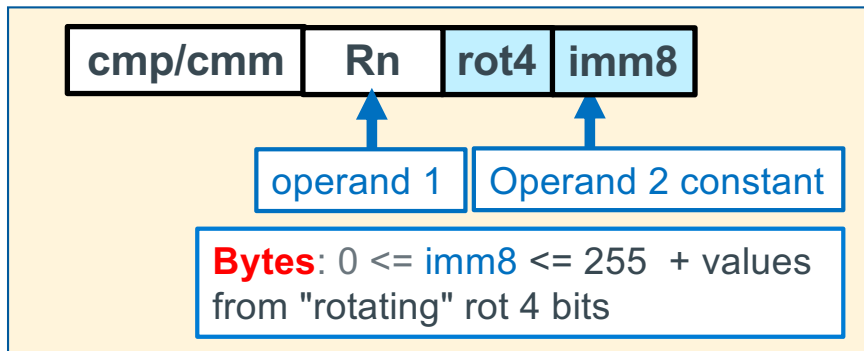


Anatomy of a Conditional Branch: If statement



- **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;
```



```
    cmp    r0, 5  
    bne    .Lendif  
  
    add    r1, r2, r3  
    add    r2, r2, 1  
  
    .Lendif:  
    mov    r3, r2
```

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

Fall through

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



```
    cmp    r0, 5  
    bgt    .Lendif  
  
    mov    r1, r2  
    add    r2, r2, 1  
  
    .Lendif:  
    mov    r3, r2
```

Fall through

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:

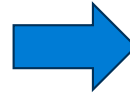
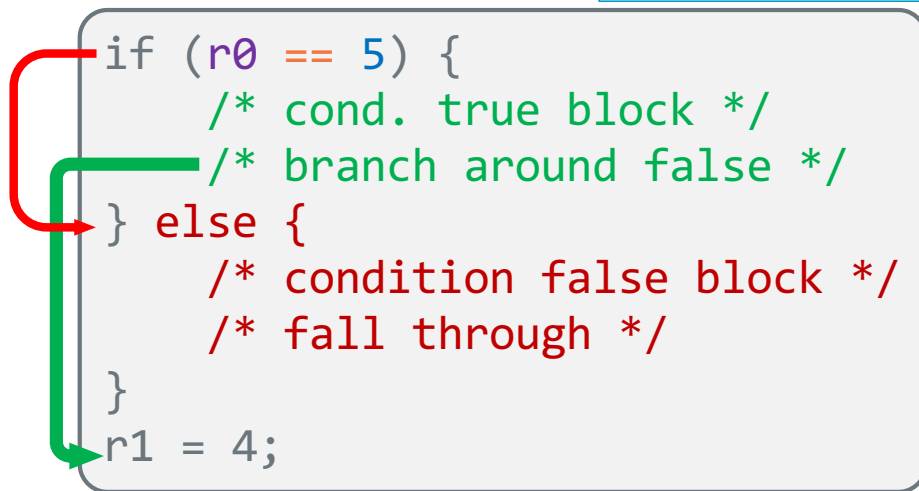
1. **adjust** the conditional test to branch to the **False Block**
2. **Fall through** to the **True Block**
3. Bottom of the **True Block** **unconditionally branches** around the **False block**

<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

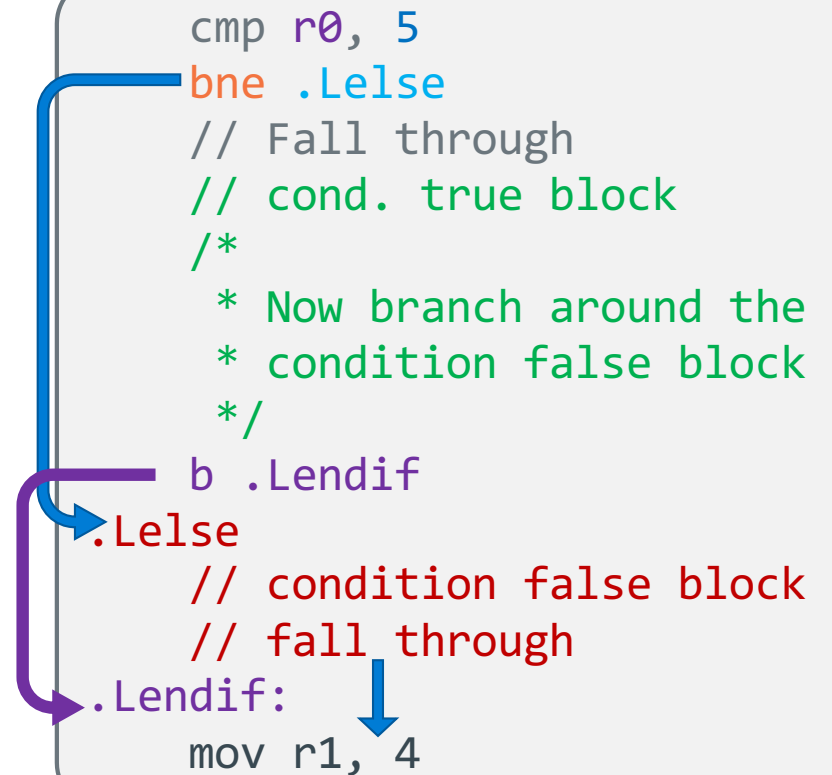
If $r0 == 5$ false
then branch **to**
false block

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

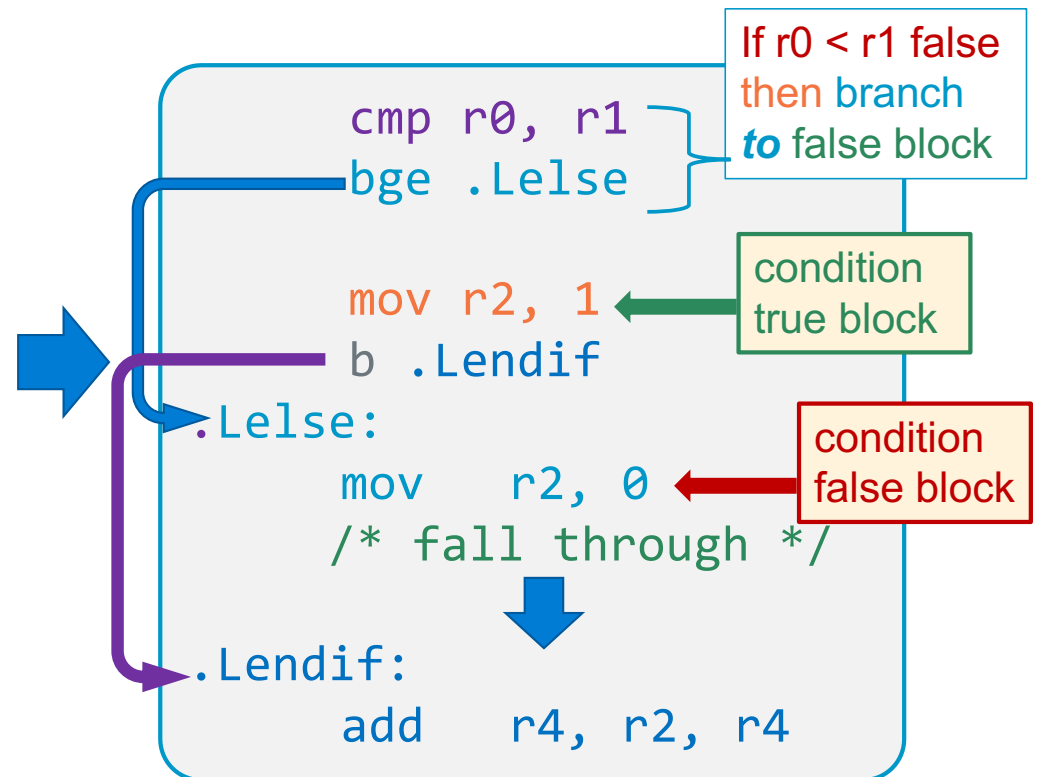
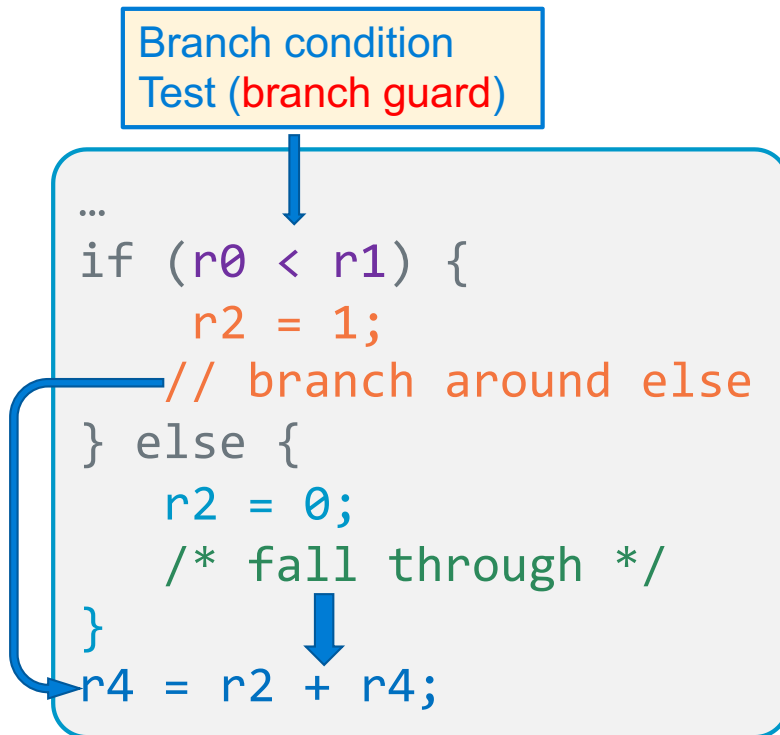


If $r0 == 5$ false
then branch
to false block

```
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

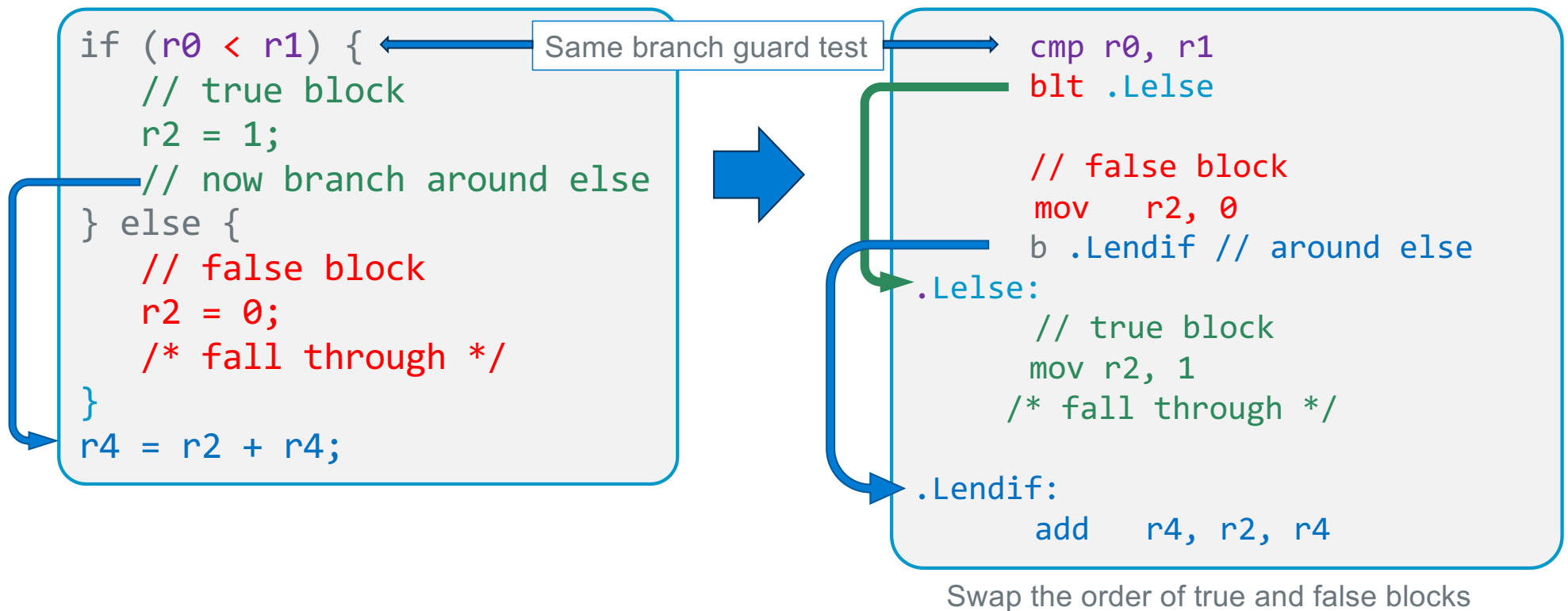
Branch condition
Test (branch guard)

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

condition
true block

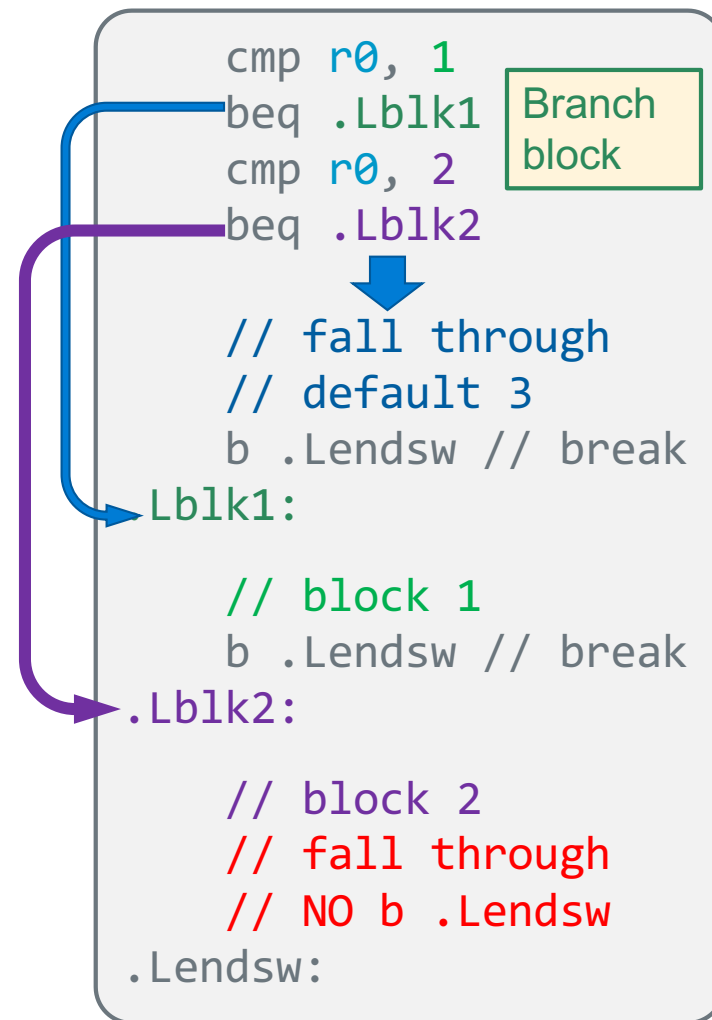
condition
false block

Preserving the same branch guard test



Switch Statement

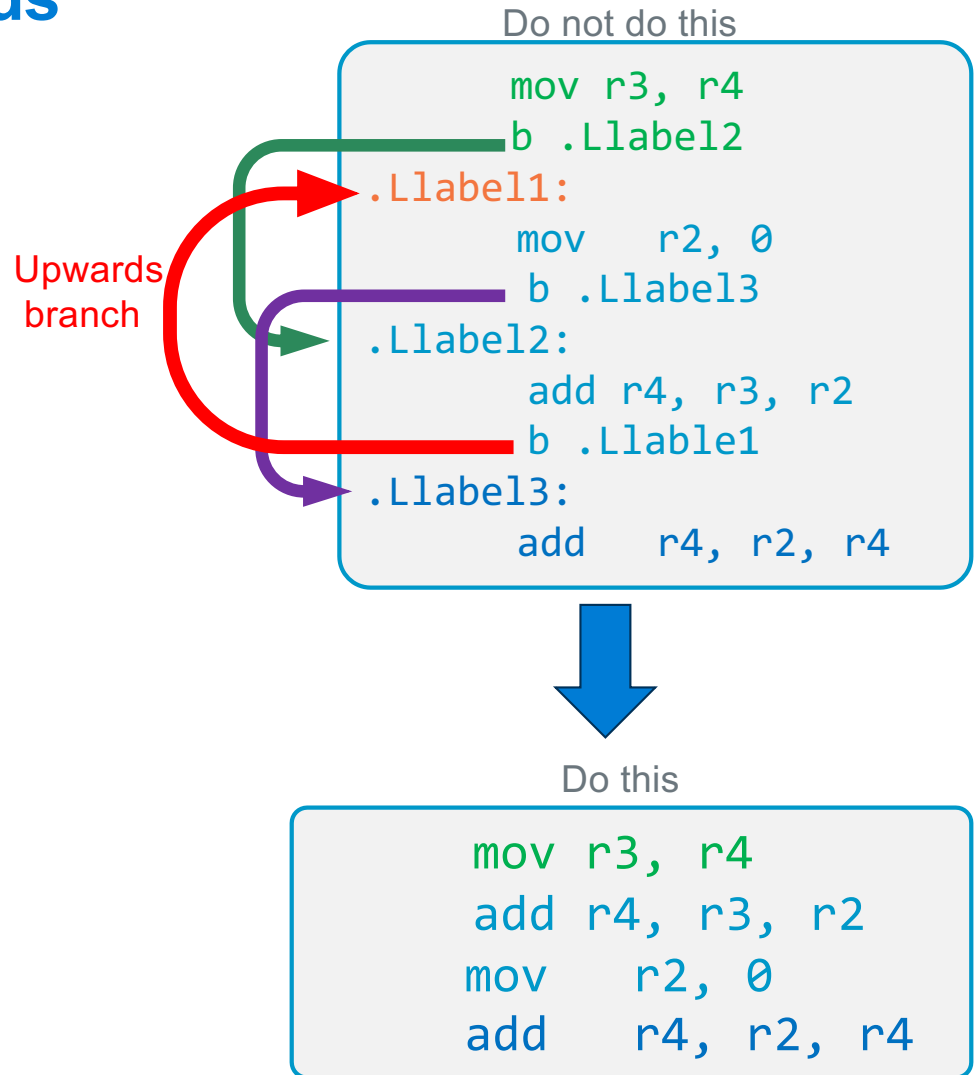
```
switch (r0) {  
  case 1:  
    // block 1  
    break;  
  case 2:  
    // block 2  
    break;  
  default:  
    // default 3  
    break;  
}
```



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
```

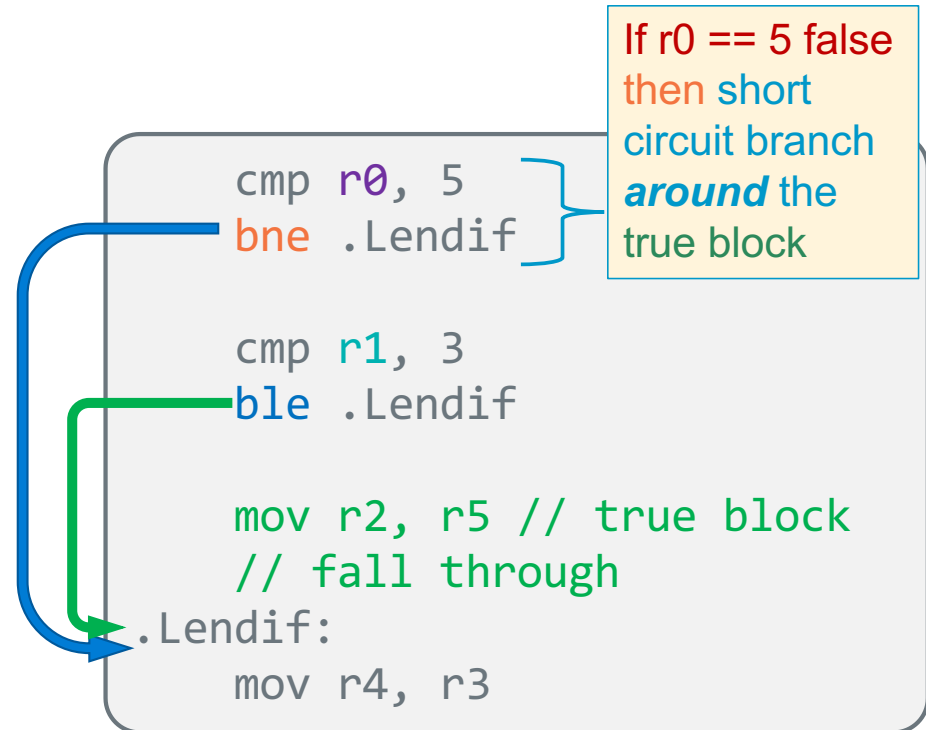


- Each expression argument is evaluated **in sequence** from left to right including any **side effects** (modified using parenthesis), **before** (optionally) evaluating the next expression argument
- If after evaluating an argument, the **value of the entire expression can be determined**, then the **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

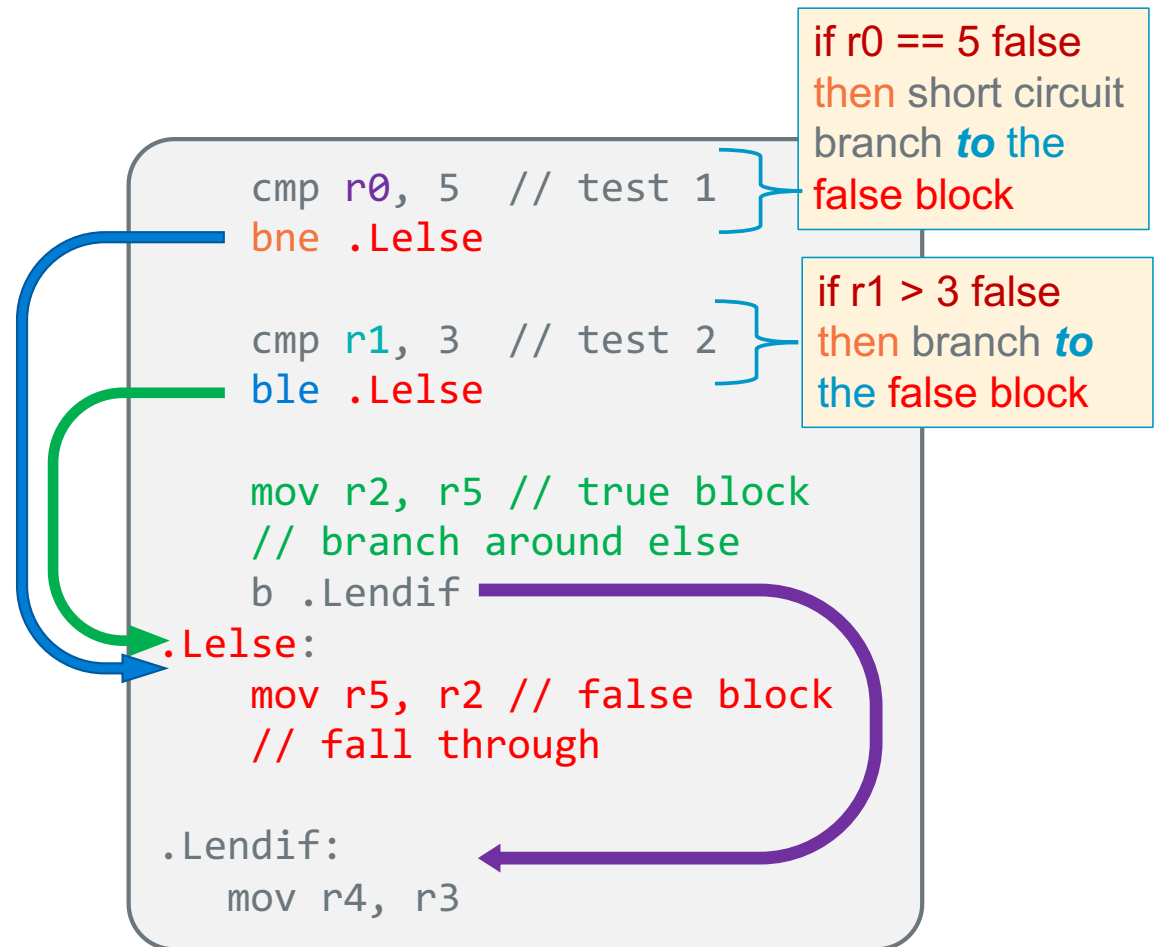
```
if ((r0 == 5) && (r1 > 3)) {  
    r2 = r5; // true block  
    /* fall through */  
}  
r4 = r3;
```



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 - 3

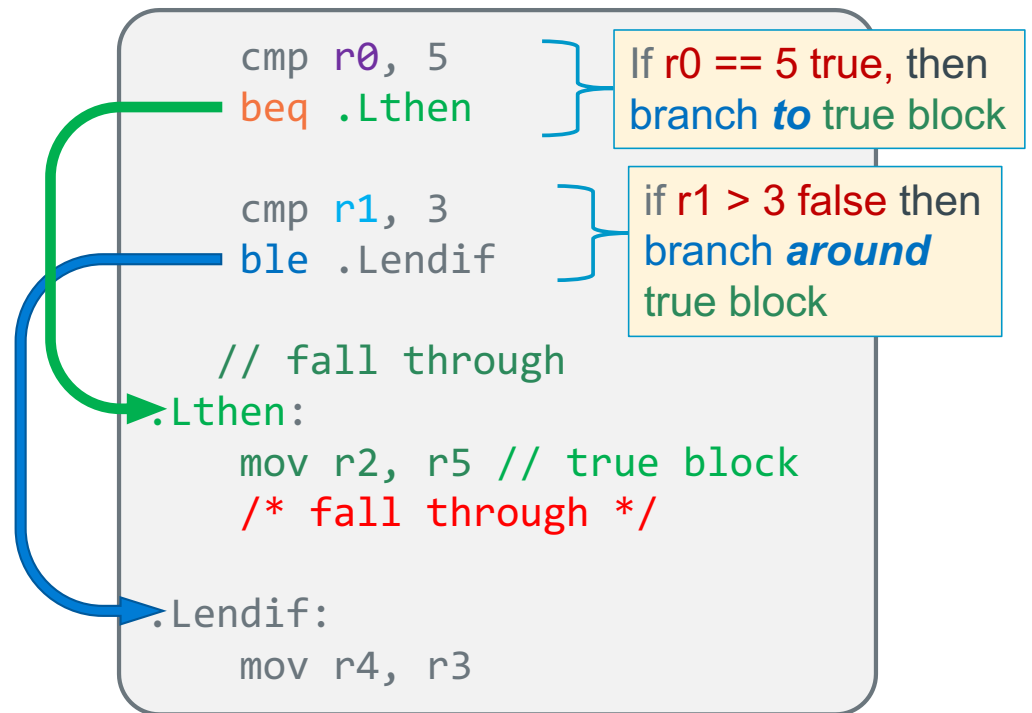
```
if (test1 && test2 & ... & testn) {  
    // true block  
    // branch around else  
} else {  
    // False block  
    /* fall through */  
}
```

All compound && tests use an **inverse test** and if they evaluate to **true** branch **to the false block**

```
cmp  xx,xx // test 1  
inverse test branch .Lelse  
  
cmp  xx, xx // test 2  
inverse test branch .Lelse  
  
    // more tests  
  
cmp  xx, xx // test n  
inverse test branch .Lelse  
  
mov  r2, r5 // true block  
// branch around else  
b .Lendif  
.Lelse:  
    mov r5, r2 // false block  
    // fall through  
  
.Lendif:  
    mov r4, r3
```

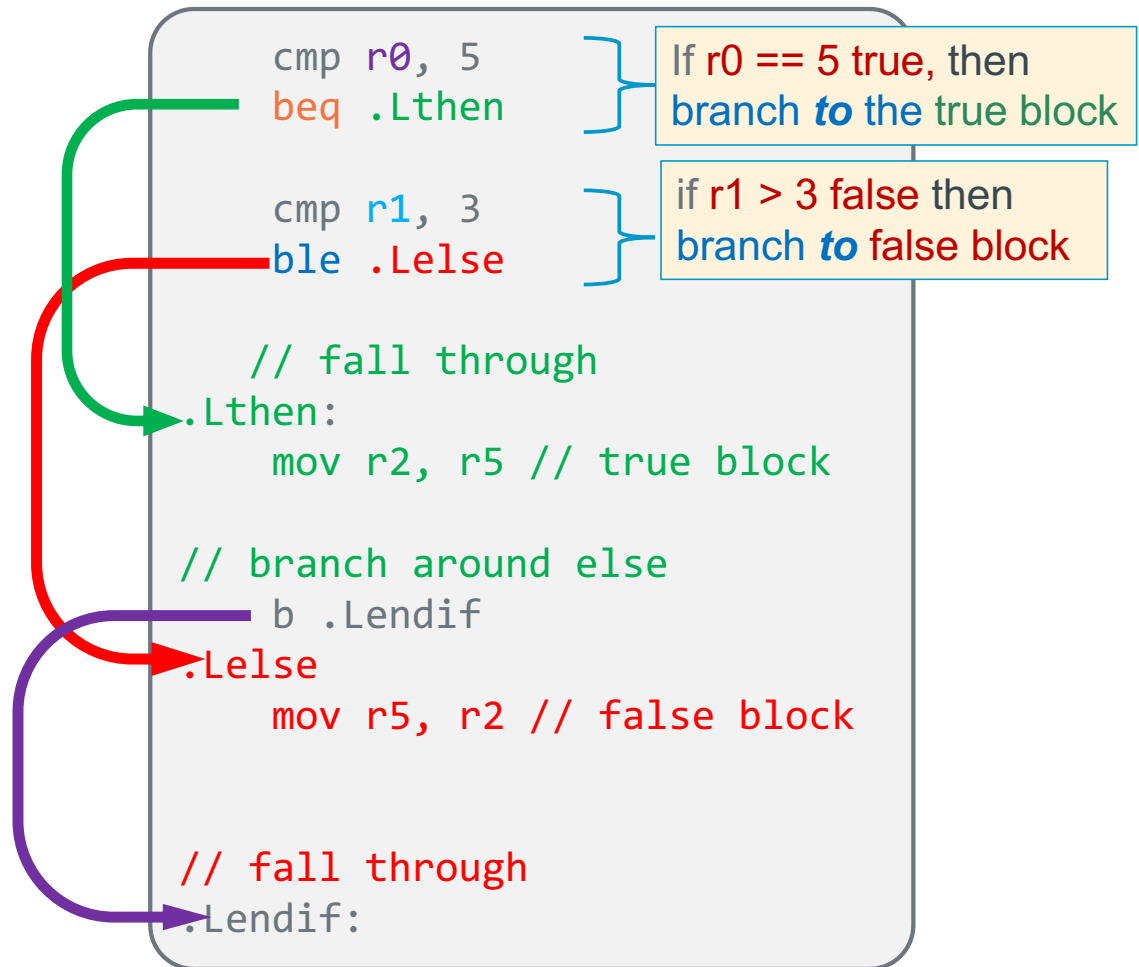
Program Flow – If statements || compound tests - 1

```
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 – If statements || compound tests - 3

```
if (test1 || test2 || .. || testn) {  
    // true block  
    /* branch around else */  
} else {  
    // false block  
    /* fall through */  
}
```

All compound || tests (except the last one) use the **same test** as C and if they evaluate to **true branches to the true block**

The last (leftmost) compound || test uses an **inverse test** and if it evaluates to **true branches to false block**

```
cmp XX, XX      // test 1  
same test branch .Lthen  
  
cmp XX, XX      // test 2  
same test branch .Lthen  
  
// other tests  
  
cmp XX, XX      // test n  
inverse test branch .Lelse  
  
// fall through  
.Lthen:  
    // true block  
    // branch around else  
b .Lendif  
.Lelse  
    // false block  
    // fall through  
.Lendif:
```

Program Flow – multiple branches, one cmp

```
if ((r0 > 5) {  
    /* condition block 1 */  
    // branch to endif  
} else if (r0 < 5){  
    /* condition block 2 */  
    // branch to endif  
} else {  
    /* condition block 3 */  
    // fall through to endif  
}  
// endif  
r1 = 11;
```

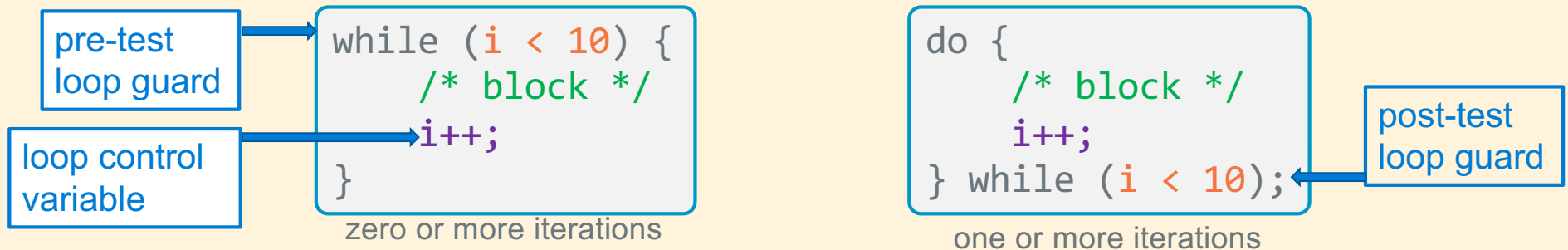
- There are many other ways to do this

```
cmp r0, 5  
bgt .Lblk1  
blt .Lblk2  
// fall through  
// condition block 3  
b .Lendif  
.Lblk1:  
    // condition block 1  
    b .Lendif  
.Lblk2:  
    // condition block 2  
    b .Lendif  
.Lendif:  
    mov r1, 5
```

special case: multiple
branches from one cmp

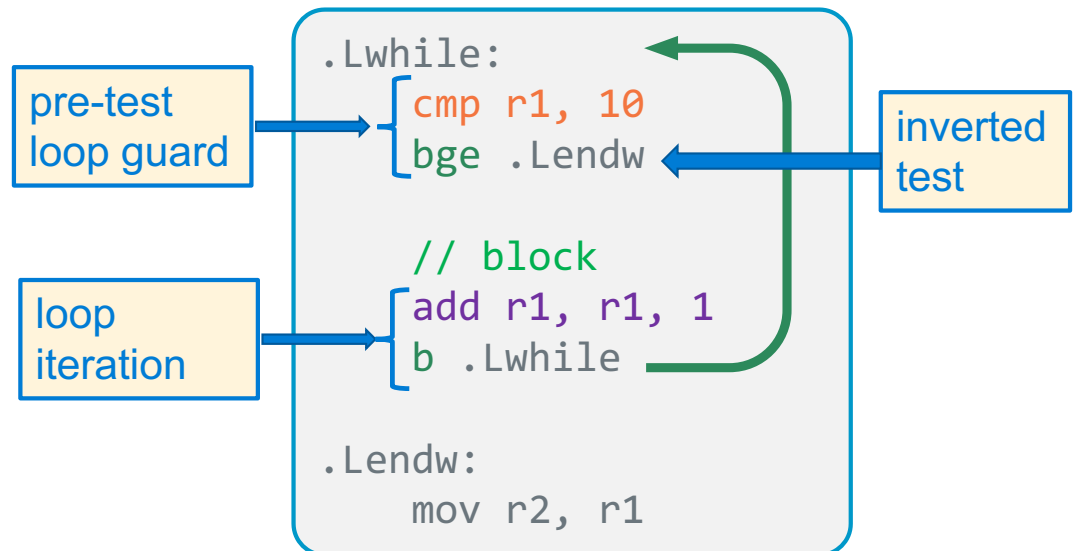
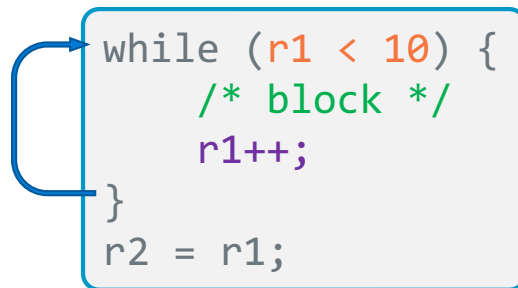
Program Flow – Pre-test and Post-test Loop Guards

- 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

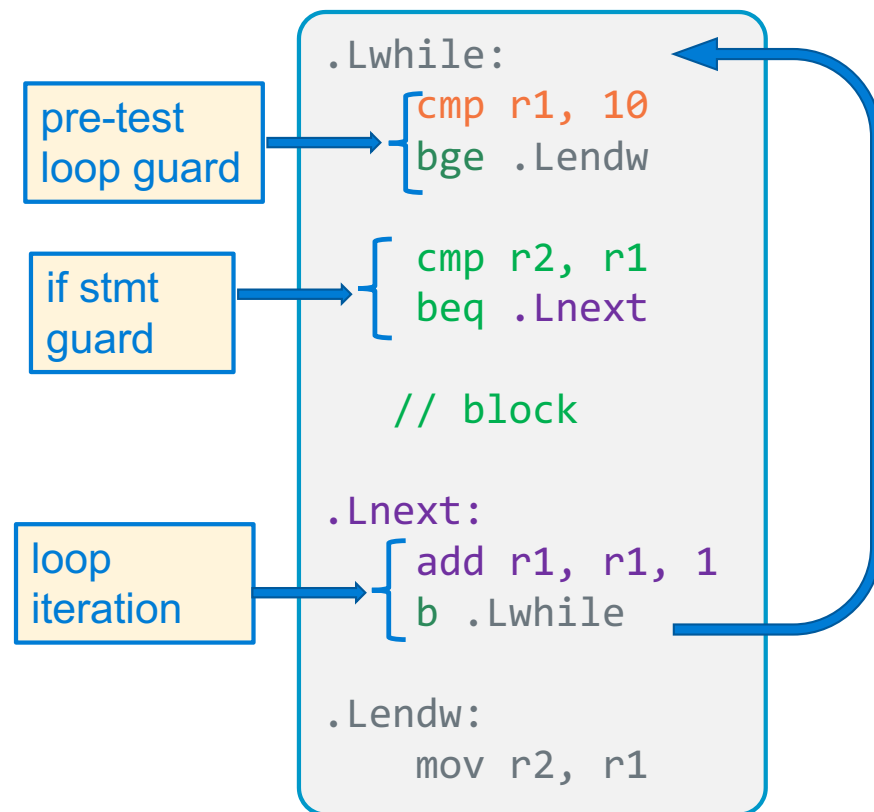
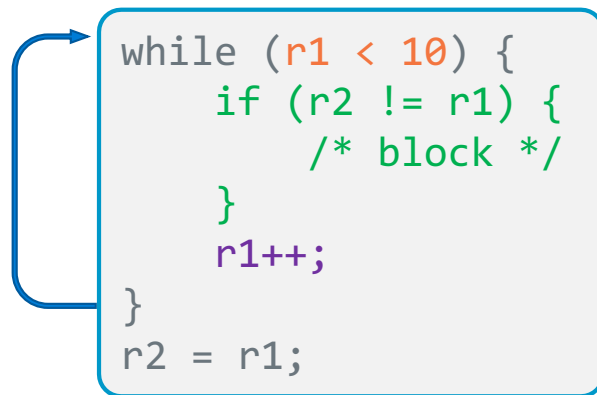


- post-test loop guard is at the bottom of the loop
 - If the test evaluates to true, execution branches to the top of the loop
 - If the test evaluates to false, execution falls through the instruction following the loop

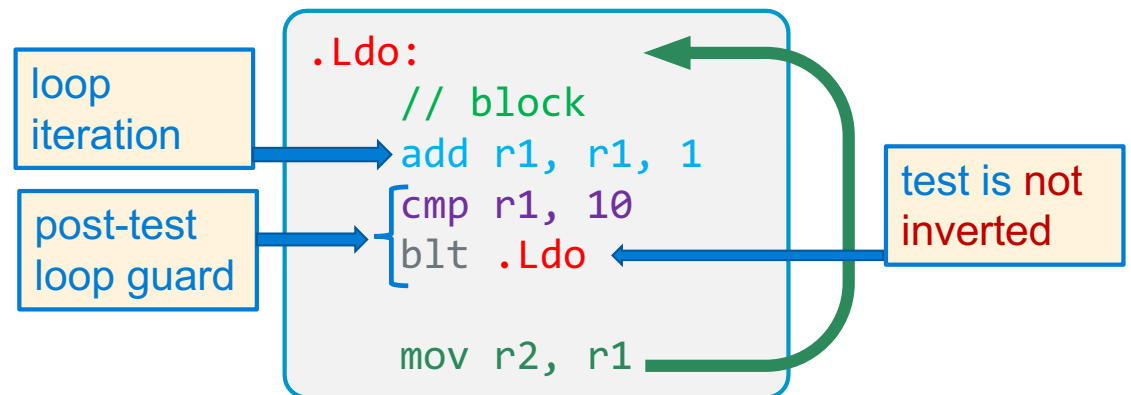
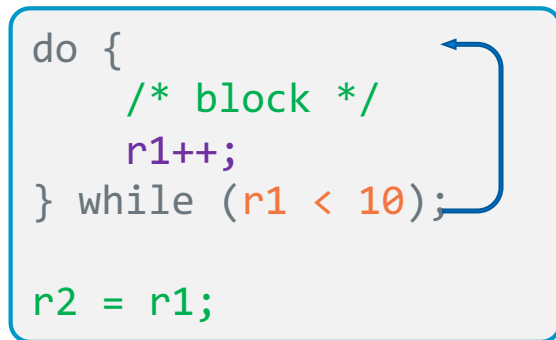
Pre-Test Guards - While Loop



Pre-Test Guards - While Loop

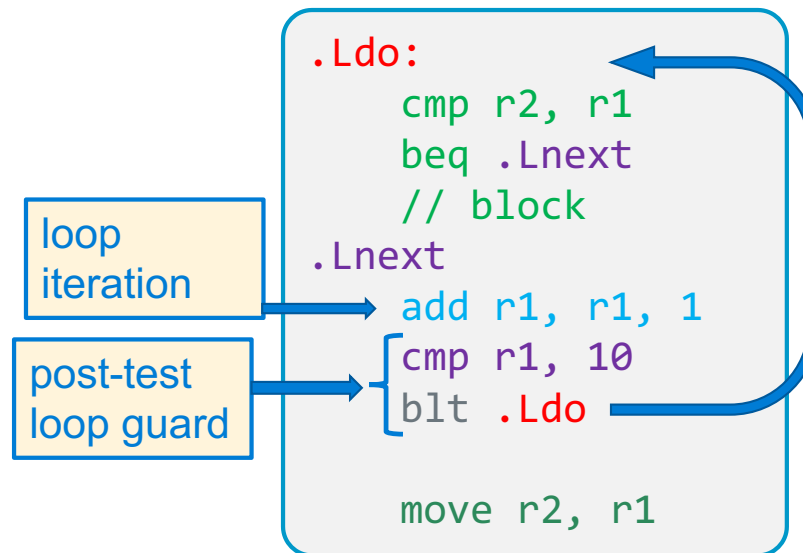


Post-Test Guards – Do While Loop

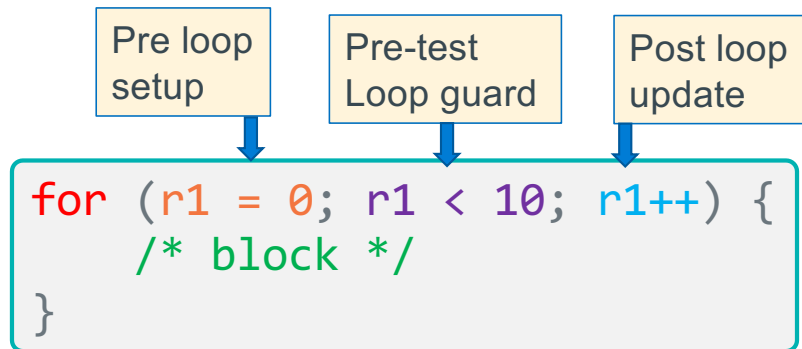


Post-Test Guards – Do While Loop

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

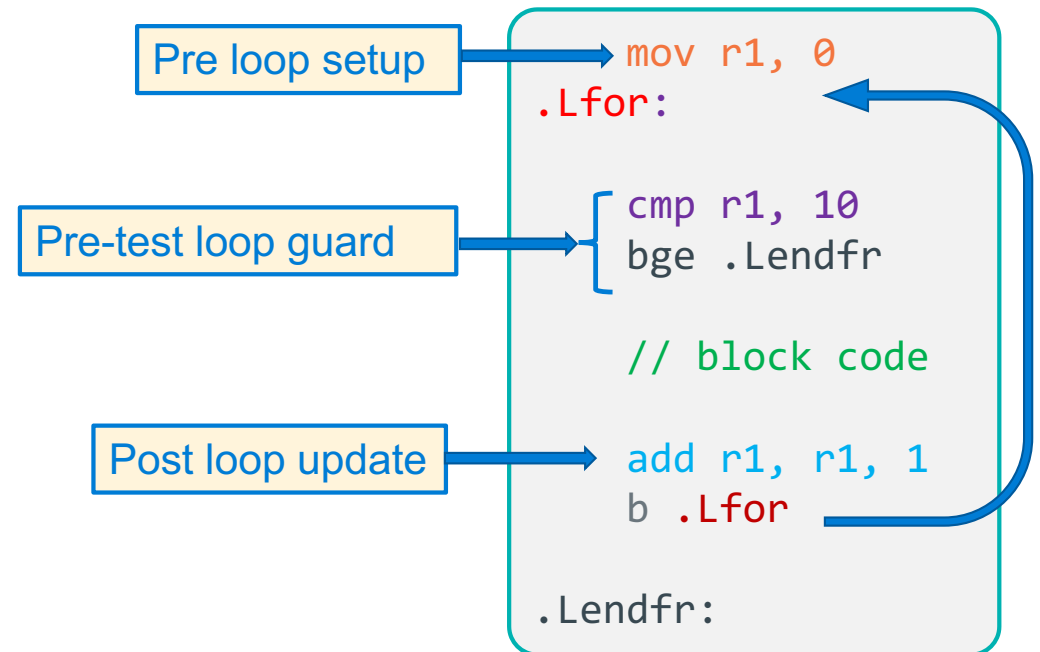


Program Flow – Counting (For) Loop Version 1

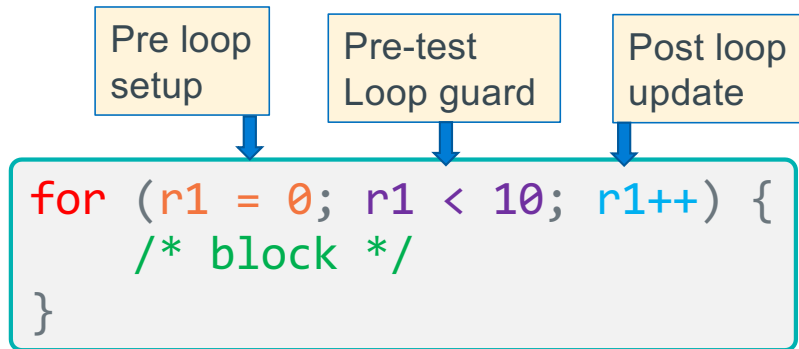


A **counting loop** has three parts:

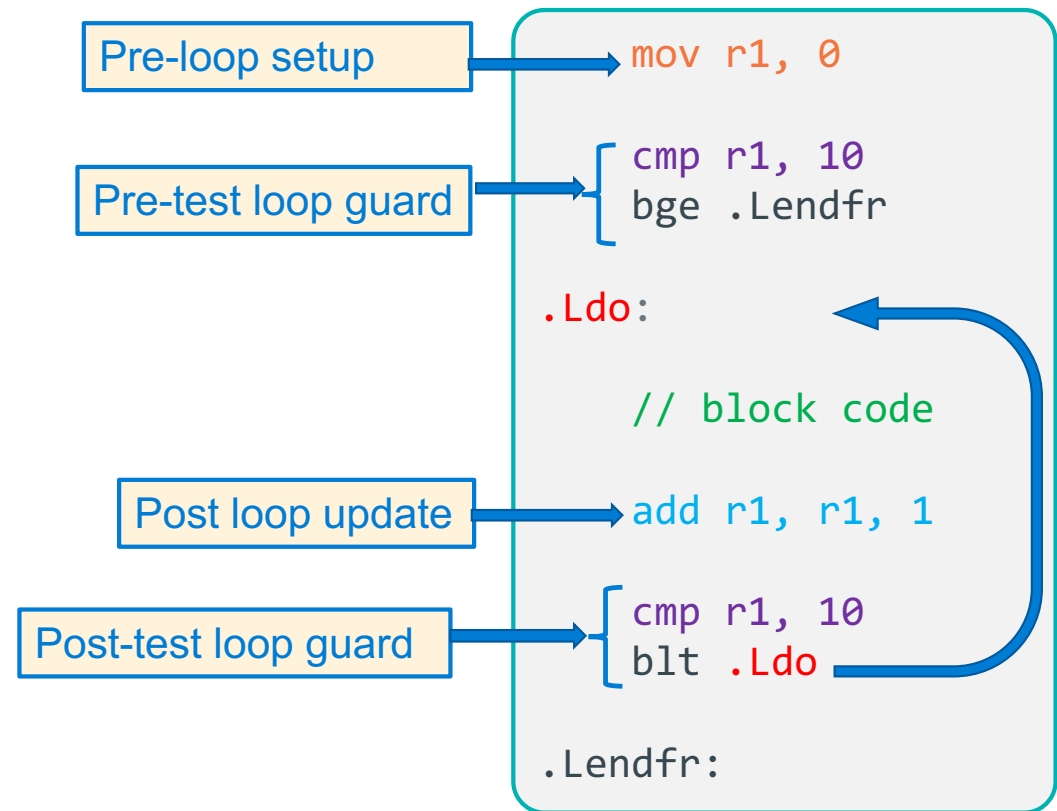
1. Pre-loop setup
2. Pre-test loop guard conditions
3. Post-loop update



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

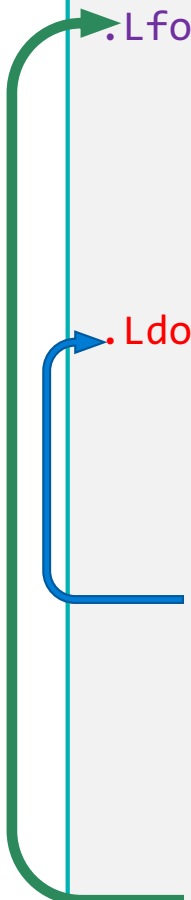
```
for (r3 = 0; r3 < 10; r3++) {  
    r0 = 0;  
  
    do {  
        r0 = r0 + r1++;  
    } while (r1 < 10);  
  
    // fall through  
    r2 = r2 + r1;  
}
```



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
```



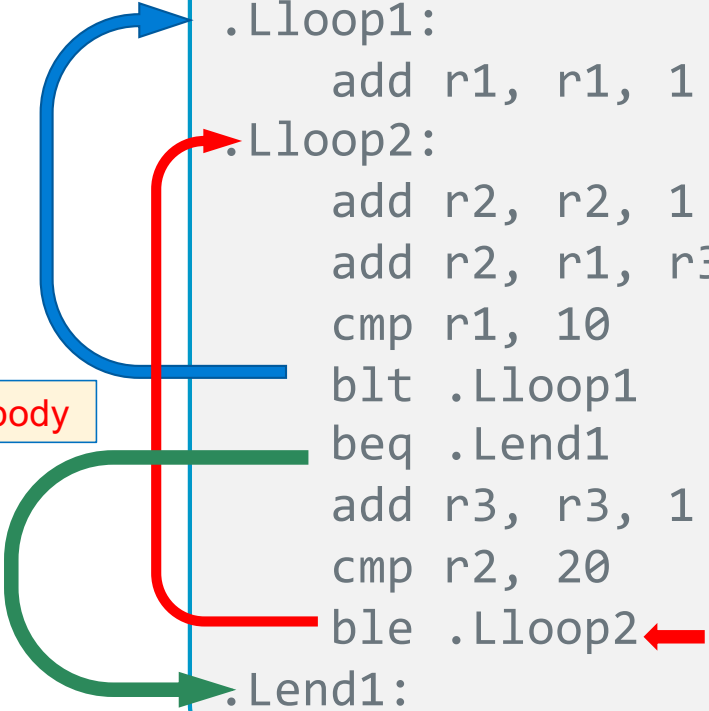
Keep loops Properly Nested: Do not branch into the middle of a loop

- 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:
```



Extra Slides

CPU Operational Overview: Executing Machine Code

Everything has a **memory address**: instructions & data

- Machine code uses addresses for loops, branches, function calls, variables, etc.

1 Fetch

- read the instruction into memory (**fetch**)
 - program counter is **automatically incremented (+4)** to **contain the address of the next instruction in memory**
- Instructions are 32 bits

2 Decode

- Decodes the instruction** and sets up execution

3 Execute

- CPU completes the **execution** of the instruction
- Execution may alter the pc to take branches, etc.
- Go to **fetch**

r15/pc
r15/pc
r15/pc
r15/pc

r15/pc
r15/pc

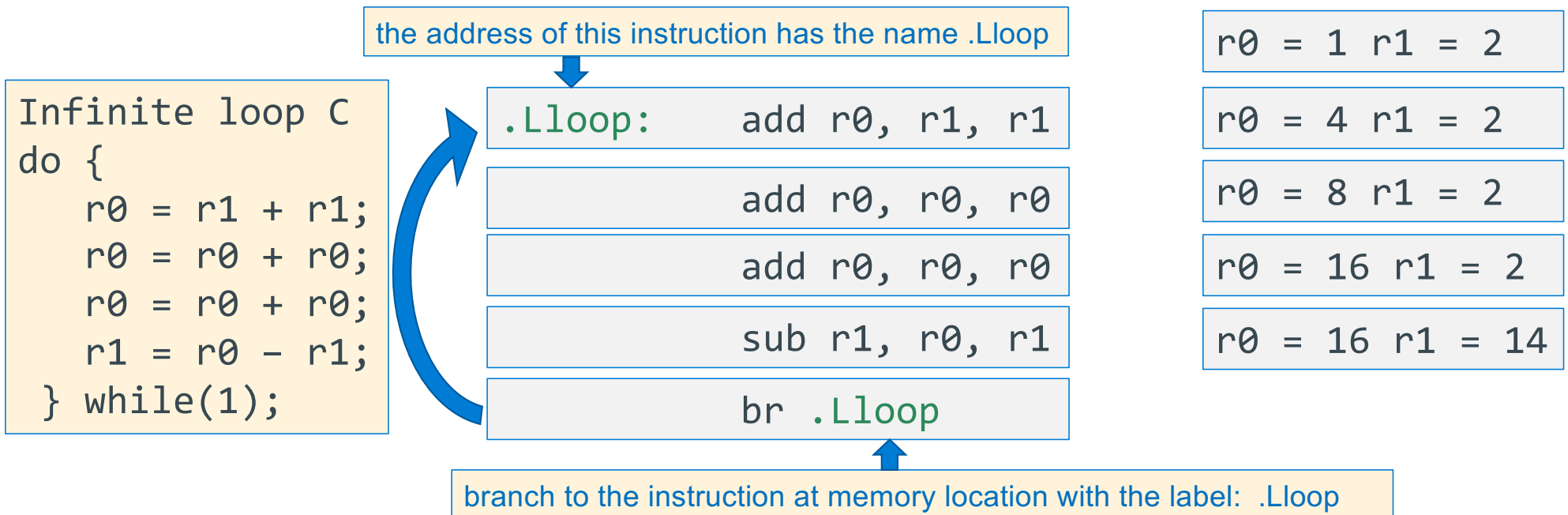
text segment in memory

address	contents	assembly version
0001042c	<inloop>:	
1042c	e3530061	cmp r3, 0x61
10430	ba000002	blt 10440 <store>
10434	e353007a	cmp r3, 0x7a
10438	ca000000	bgt 10440 <store>
1043c	e2433020	sub r3, r3, #32
00010440	<store>:	
10440	e7c13002	strb r3, [r1, r2]
10444	e2822001	add r2, r2, 0x1
10448	e7d03002	ldrb r3, [r0, r2]
1044c	e3530000	cmp r3, 0x0
10450	1afffff5	bne 1042c <inloop>

Edited output For output Created by the command
`%objdump -d a.out`
`%objdump -d -S a.out adds source code`

Program Execution: Looping in the Execution Flow

- Repeat the series of instructions in a loop means **altering the flow of execution**
- This is used with if statements and loops
- Below is an **infinite** loop (br instruction: unconditional branch: "goto")



Branch Target Address (BTA): What Is imm24?

- Previous slide: **phases of execution:**
(1) fetch, (2) decode, (3) execute
- The pc (r15) contains the address of the **instruction being fetched**, which is two instructions ahead or **executing instruction + 8 bytes**
- **Branch target address** (or imm24) is the **distance measured** in the **# of instructions** (signed, 2's complement) from the **fetch address** contained in **r15** when executing the branch

executing instruction

decode instruction

fetch instruction

```

0001042c <inloop>:
1042c: e3530061      cmp r3, 0x61
10430: ba000002      blt 10440 <store>
10434: e353007a      cmp r3, 0x7a
10438: ca000000      bgt 10440 <store>
1043c: e2433020      sub r3, r3, #32

00010440 <store>:
10440: e7c13002      strb r3, [r1, r2]
10444: e2822001      add r2, r2, 0x1
10448: e7d03002      ldrb r3, [r0, r2]
1044c: e3530000      cmp r3, 0x0
10450: 1affffff5     bne 1042c <inloop>
    
```

BTA: + 2 instructions

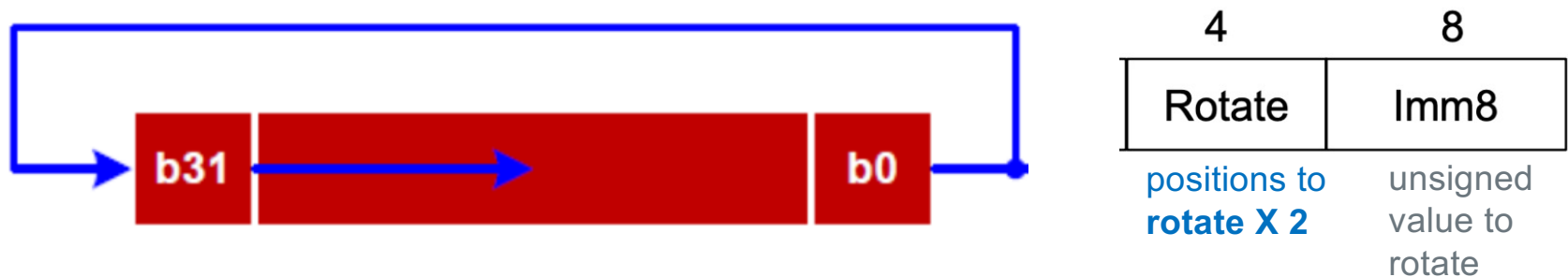
```

target address    = 0x10440
fetch address     = 0x10438
distance(bytes)   = 0x00008
distance(instructions) = 0x8/(4 bytes/instruction)= 0x2
    
```

imm24	0x 00 00 02
-------	-------------

How are I – Type Constants Encoded in the instruction?

- Aarch32 provides only 8-bits for specifying an immediate constant value
- Without "rotation" immediate values are limited to the range of positive 0-255
- Imm8 expands to 32 bits and does a rotate right to achieve additional constant values (YUCK)



<i>rot4 value</i>	32-bit constant result
0000	0000 0000 0000 0000 0000 0000 0000 1111 1111
0001 (2 bits)	1100 0000 0000 0000 0000 0000 0000 0011 1111

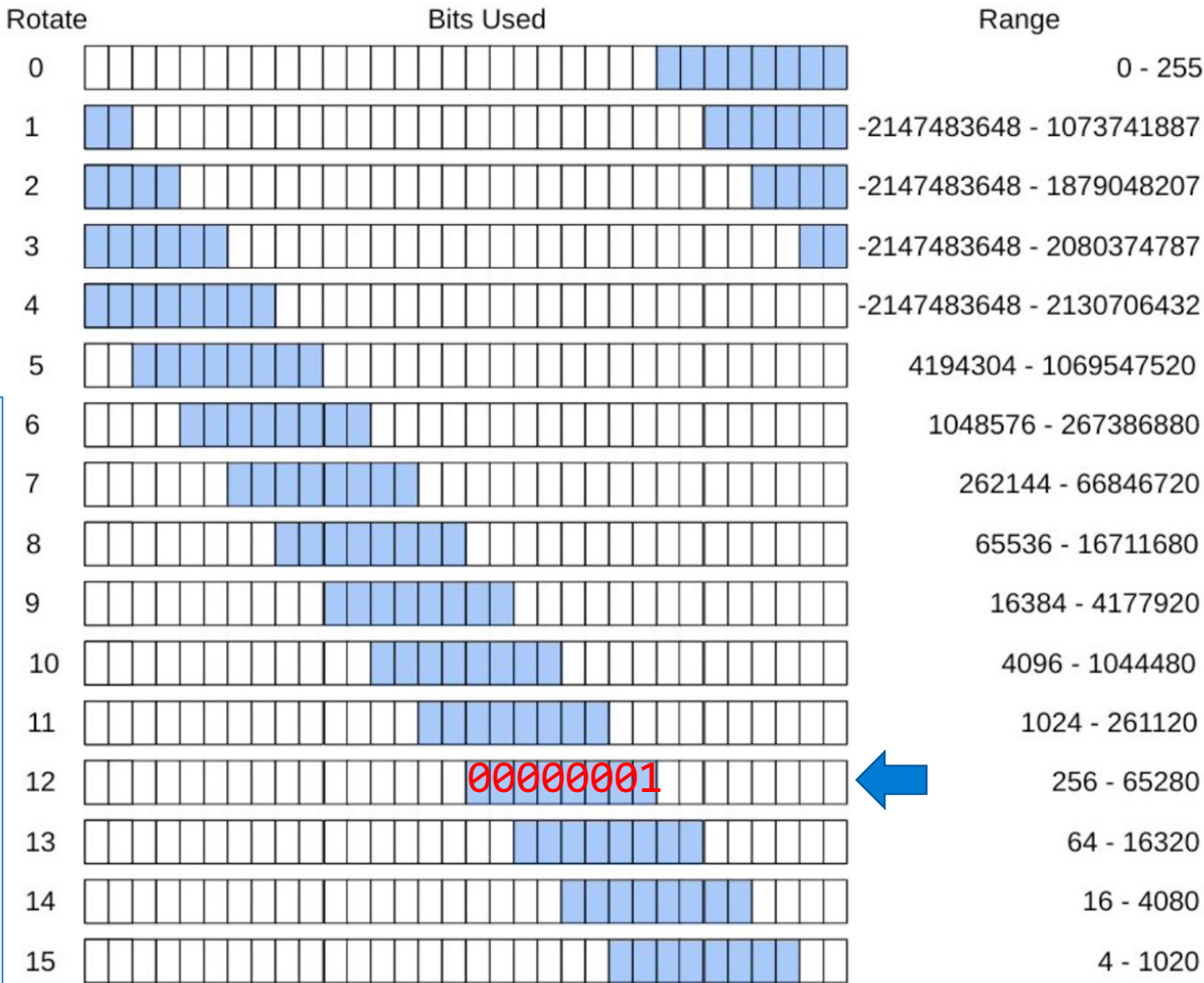
results are interpreted as a 2's complement number

two bits rotated right

Rot4 - Imm8 Values

4	8
Rotate	Imm8
positions to rotate X 2	unsigned value to rotate

- How would 256 be encoded?
 - rotate = 12, imm8 = 1
- Bottom line:** the assembler will do this for you
- If you try and use an immediate value that it cannot generate it will give an error
- There is a workaround - later



results are interpreted as a 2's complement number