

Version 1.05

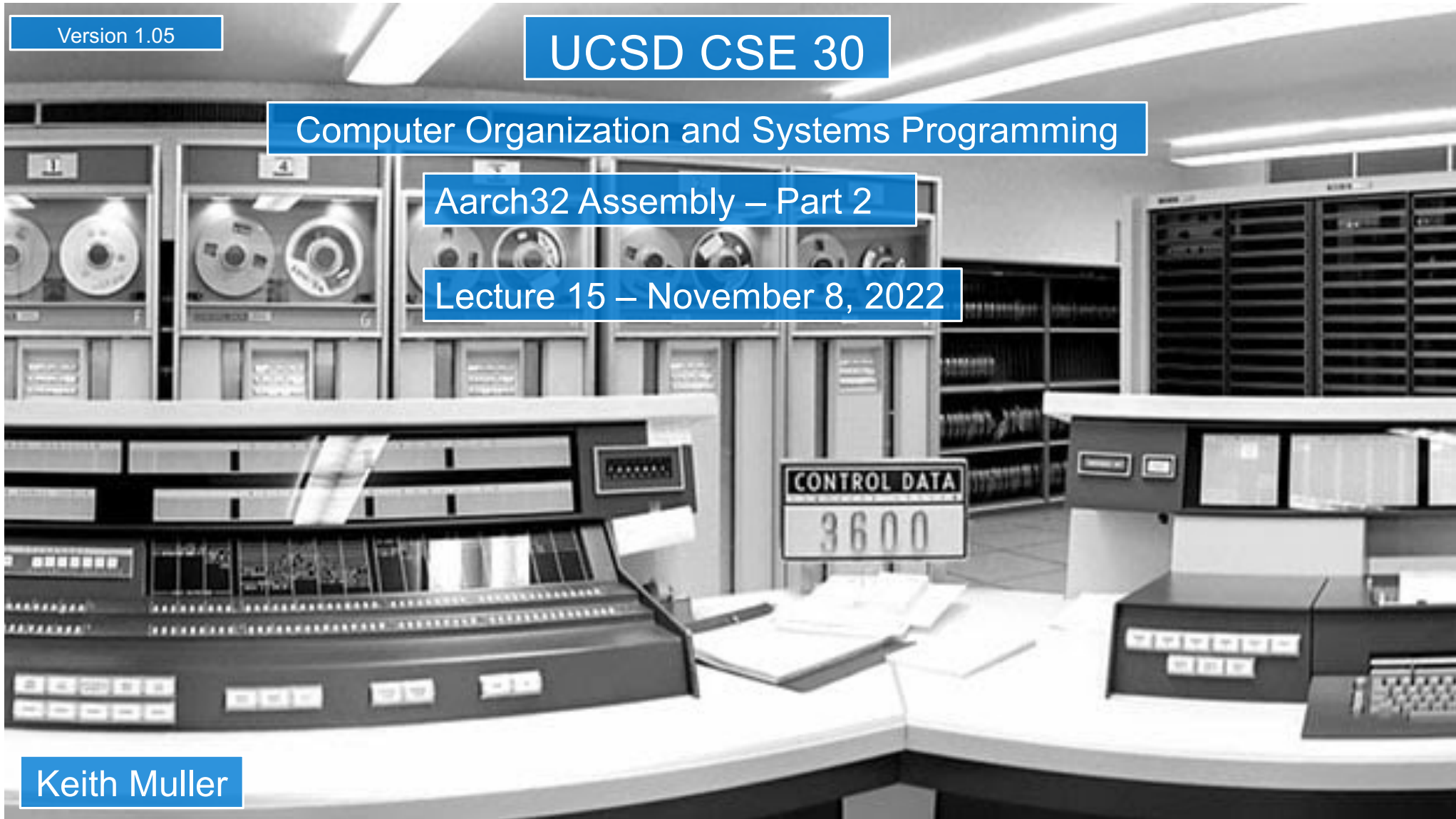
UCSD CSE 30

Computer Organization and Systems Programming

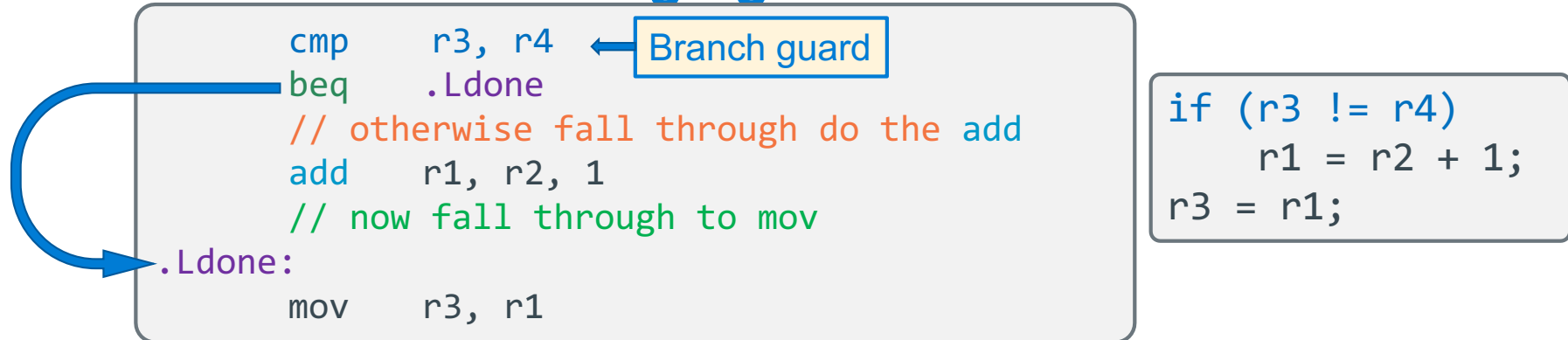
Aarch32 Assembly – Part 2

Lecture 15 – November 8, 2022

Keith Muller



Conditional Branch: Changing the Next Instruction to Execute



Condition	Meaning	Flag Checked
BEQ	Equal	Z = 1
B	Always (unconditional)	

```
cmp    r3, r4    // r3 - r4
// if r3 != r4 sets Z = 0
```

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

Examples: Guards (Conditional Tests) and their Inverse

Compare in C	<i>"Inverse"</i> Compare in C
==	!=
!=	==
>	<=
>=	<
<	>=
<=	>

Conditional Branch: Changing the Next Instruction to Execute



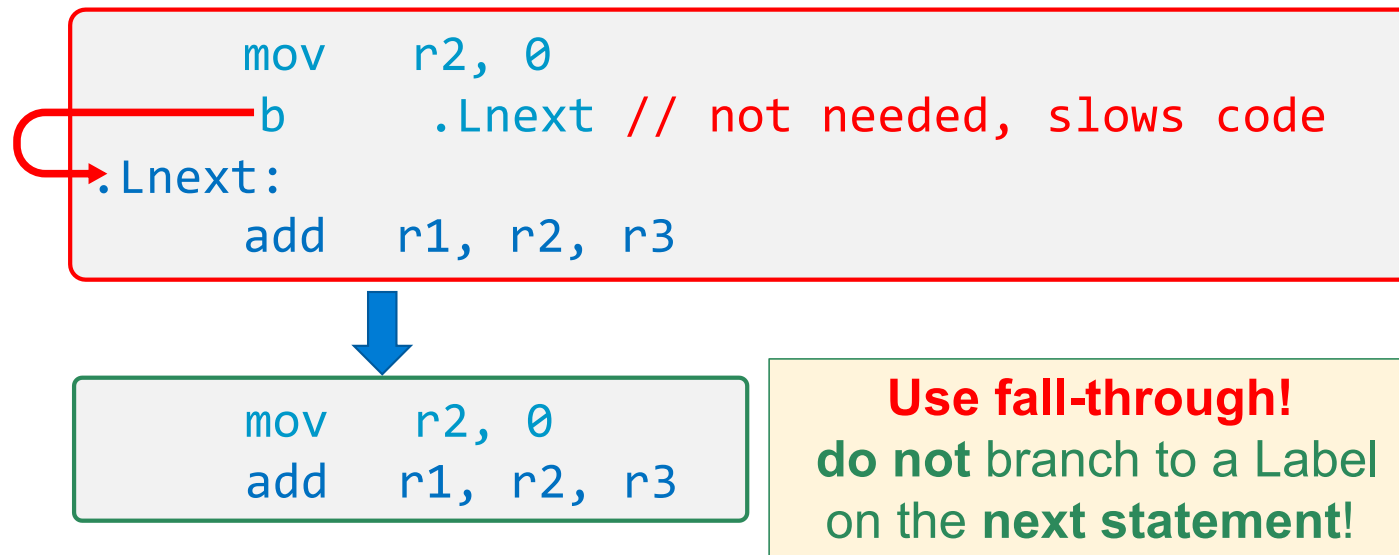
Branch instruction

bsuffix .Llabel

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

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
BHS	Unsigned \geq ("Higher or Same") or Carry Set	C = 1
BLO	Unsigned $<$ ("Lower") or Carry Clear	C = 0
BHI	Unsigned $>$ ("Higher")	C = 1 && Z = 0
BLS	Unsigned \leq ("Lower or Same")	C = 0 Z = 1
BMI	Minus/negative	N = 1
BPL	Plus - positive or zero (non-negative)	N = 0
BVS	Overflow	V = 1
BVC	No overflow	V = 0
B (BAL)	Always (unconditional)	

Eliminate unnecessary branches and labels: use **Fall Throughs**



Branching, What not to do: Spaghetti Code

```
    mov    r1, 1
    mov    r2, 2
    b      .Lthree
    mov    r5, 5
    b      .Lsix
.Lthree:
    mov    r3, 3
    mov    r4, 4
    b      .Lseven
.Lsix:
    mov    r6, 6
.Lseven:
    mov    r7, 7
```


Observation
Using **many branch** commands (conditional or unconditional) is an indication you should look to reorganize your code

To the left are many unreachable sections of code

Much faster and easier to read!

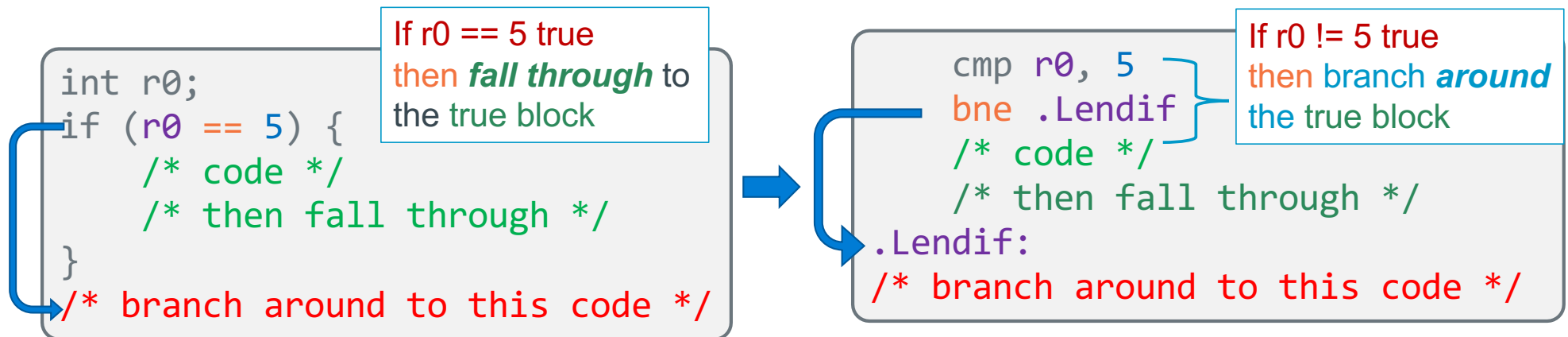
```
    mov    r1, 1
    mov    r2, 2
    mov    r3, 3
    mov    r4, 4
    mov    r7, 7
```

Program Flow: Simple If statement, No Else

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

<i>C source Code</i>	<i>Incorrect Assembly</i>	<i>Correct Assembly</i>
<pre>int r0; if (r0 == 5) { //code }</pre>	<pre>cmp r0, 5 beq .Lendif //code .Lendif:</pre>	<pre>cmp r0, 5 bne .Lendif // code .Lendif:</pre>



Branch Guard "*Adjustment*" Table

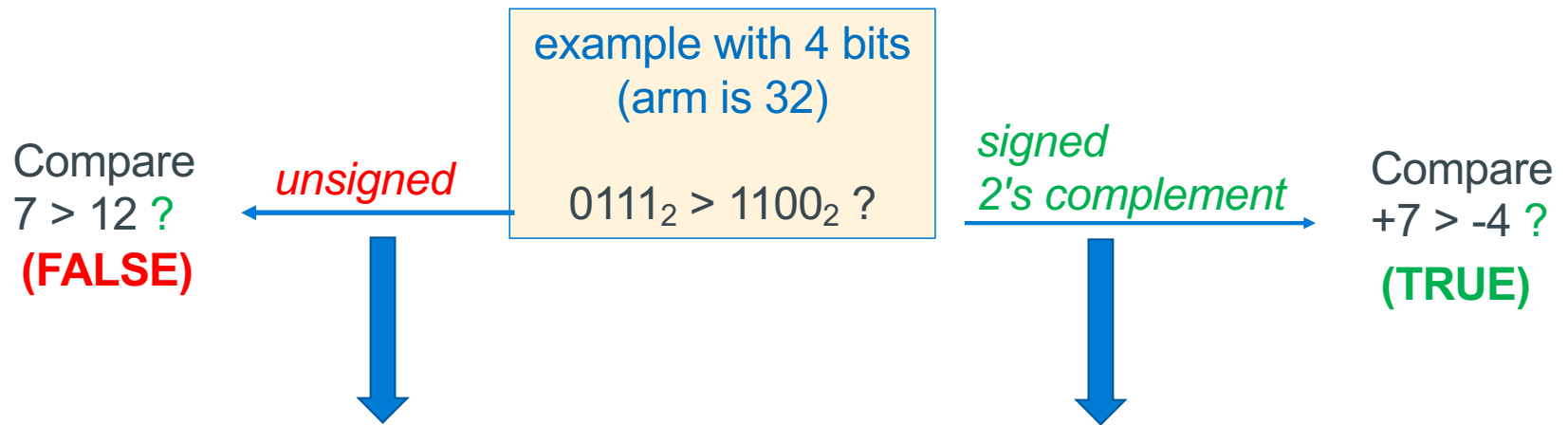
Preserving Block Order In Code

Compare in C	"Inverse" Compare in C	"Inverse" Signed Assembly	"Inverse" Unsigned Assembly
==	!=	bne	bne
!=	==	beq	beq
>	<=	ble	bls
>=	<	blt	blo
<	>=	bge	bhs
<=	>	bgt	bhi

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

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


When do you use a Signed or Unsigned Conditional Branch?



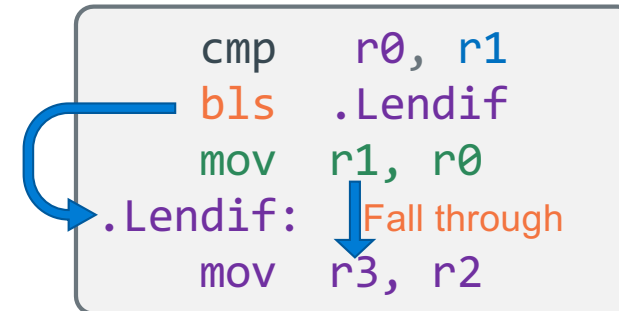
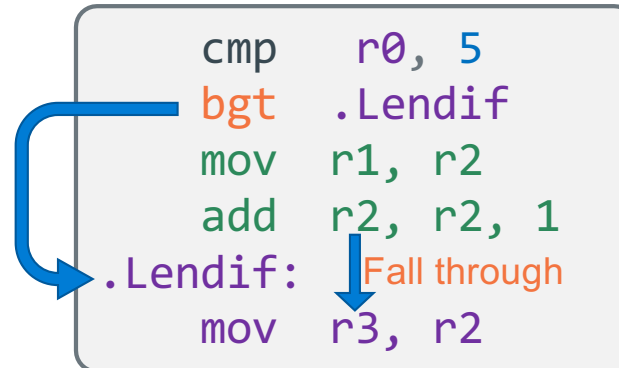
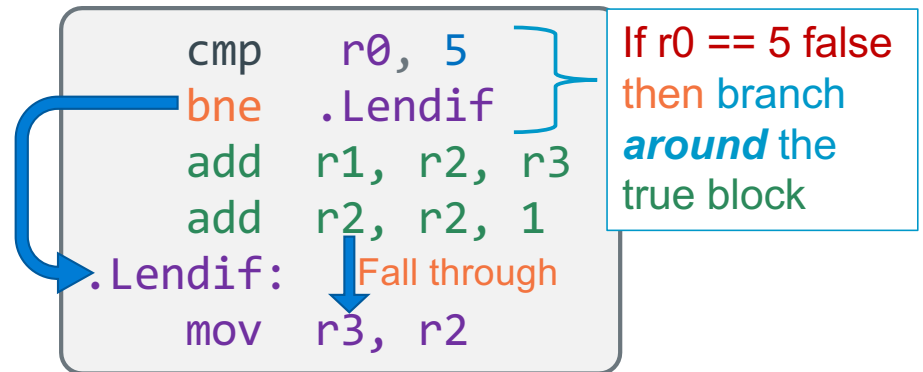
Condition	Suffix For Unsigned Operands:	Suffix For Signed Operands:
>	BHI (<i>Higher Than</i>)	BGT (<i>Greater Than</i>)
>=	BHS (<i>Higher Than or Same</i>) (<i>BCS</i>)	BGE (<i>Greater Than or Equal</i>)
<	BLO (<i>Lower Than</i>) (<i>BCC</i>)	BLT (<i>Less Than</i>)
<=	BLS (<i>Lower Than or Same</i>)	BLE (<i>Less Than or Equal</i>)
==	BEQ (<i>Equal</i>)	
!=	BNE (<i>Not Equal</i>)	

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;
```

```
unsigned int r0, r1;  
if (r0 > r1) {  
    r1 = r0;  
}  
r3 = r2;
```



Branching: Using Fall through!

Some call this "goto like" structure

- Do not use unnecessary branches when a “fall through” works
- You can see this by structures that have 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
```

Anatomy of a Conditional Branch: If - Else statement

Branch condition
Test (branch guard)

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

condition
true block

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** 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

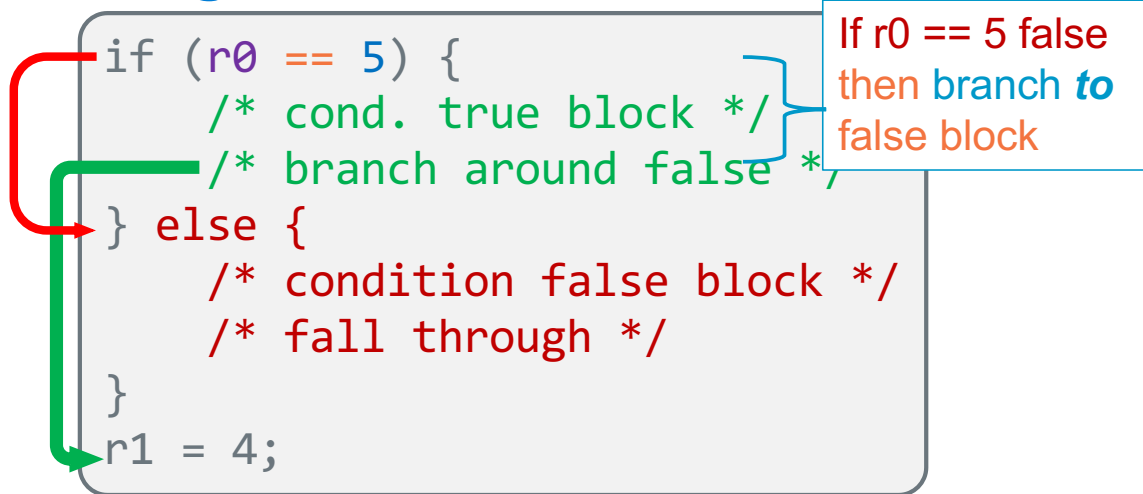
Branch condition
Test (branch guard)

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

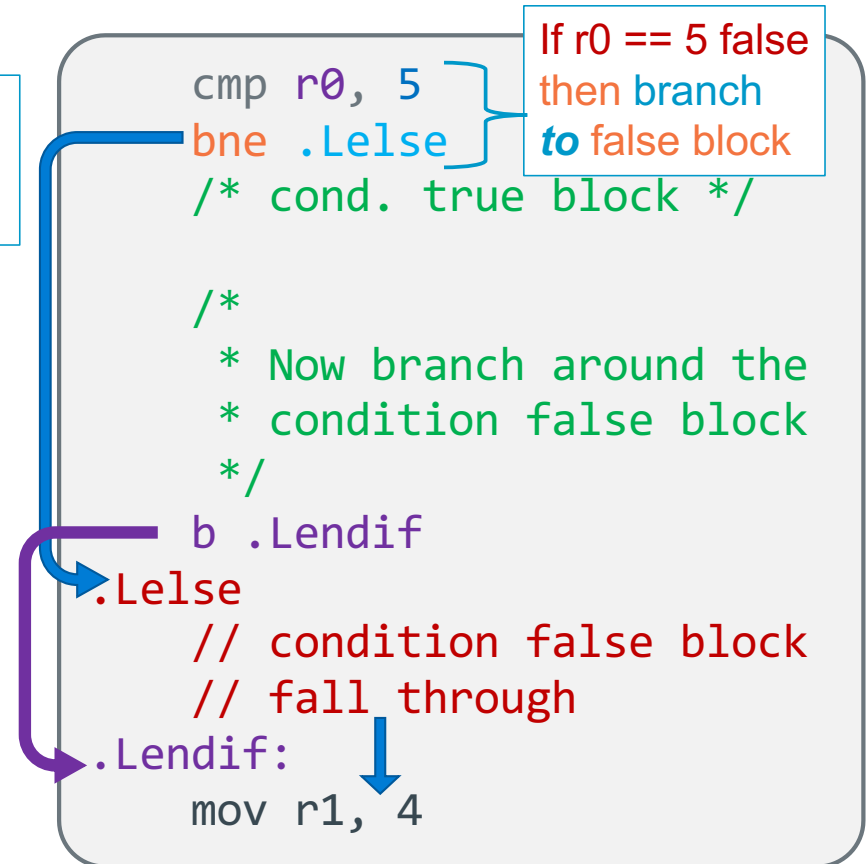
condition
true block

condition
false block

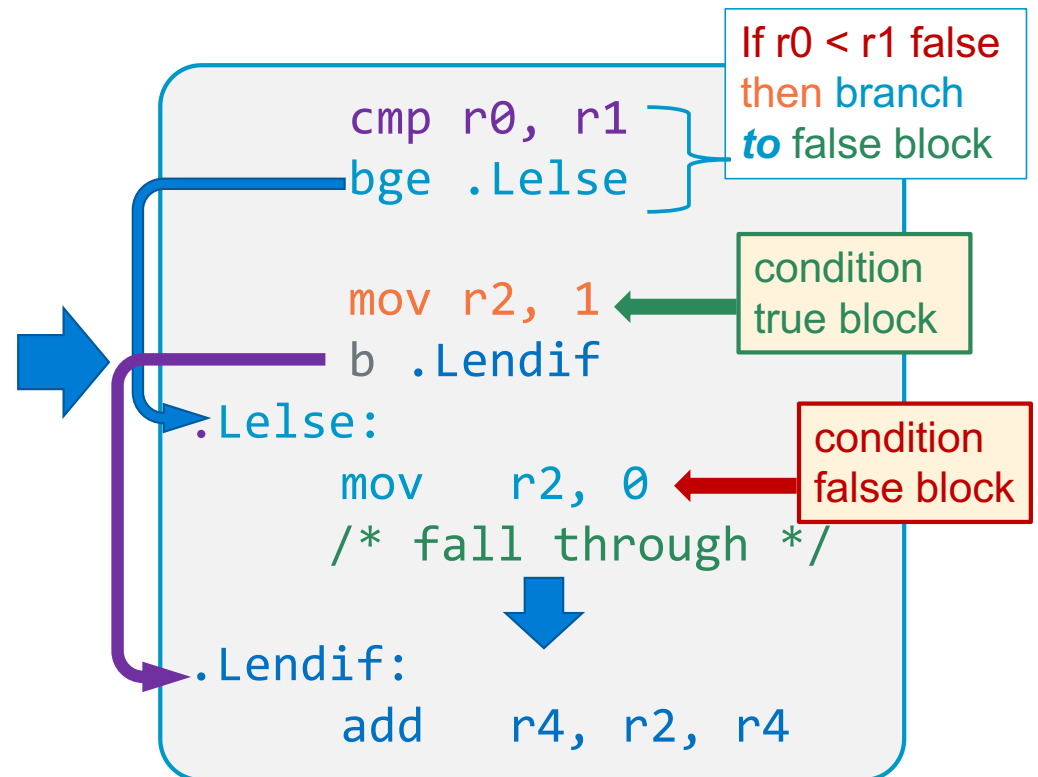
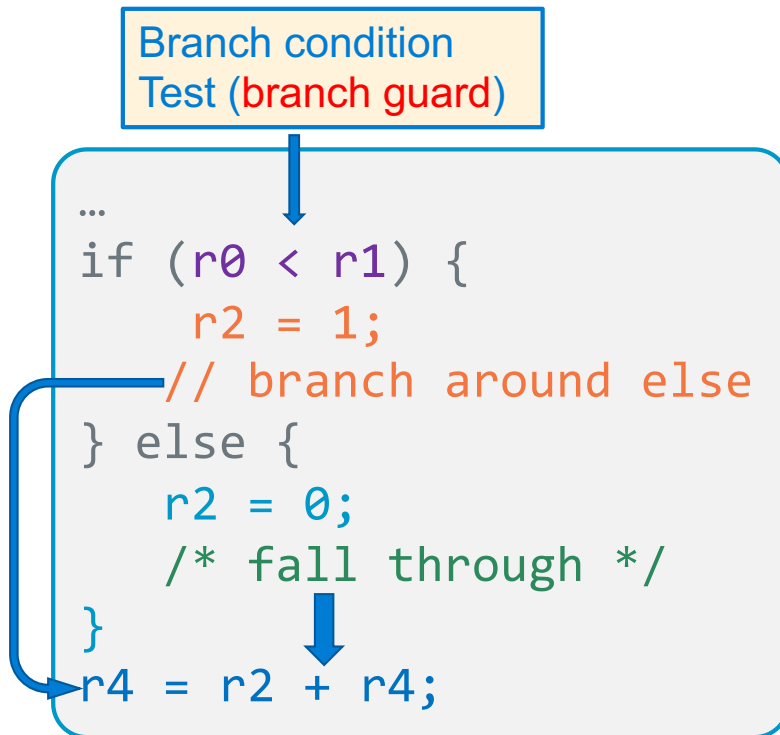
Program Flow: If with an Else



1. Make the adjustment to the conditional test to **branch to** the false block
2. When you finish the true block, you do an **unconditional branch around** the false block
3. The **false block falls through** to the following instructions




If with an Else Examples




If with an Else Block order: All These Are Equivalent


```
if (r0 < r1) {  
    r2 = 1;  
    // now branch around else  
} else {  
    r2 = 0;  
    /* fall through */  
}  
r4 = r2 + r4;
```



```
if (r0 >= r1) {  
    r2 = 0;  
    // now branch around else  
} else {  
    r2 = 1;  
    /* fall through */  
}  
r4 = r2 + r4;
```

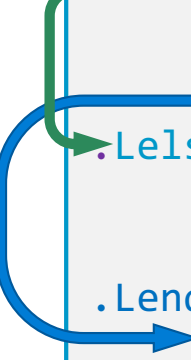


```
cmp r0, r1  
bge .Lelse  
mov r2, 1  
b .Lendif // around else  
.Lelse:  
    mov    r2, 0  
    /* fall through */  
.Lendif:  
    add    r4, r2, r4
```



Same test
swapped blocks

```
cmp r0, r1  
blt .Lelse  
mov    r2, 0  
b .Lendif // around else  
.Lelse:  
    mov r2, 1  
    /* fall through */  
.Lendif:  
    add    r4, r2, r4
```



Switch Statement

Approach 1 – Branch Block

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

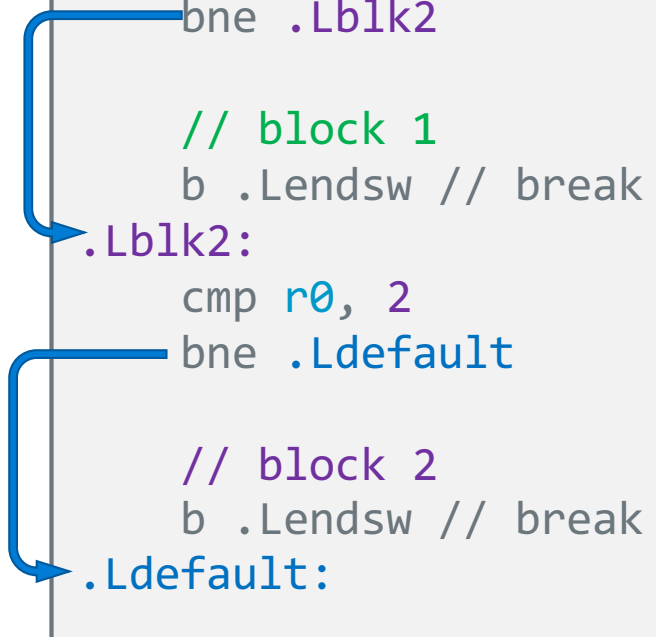
```
    cmp r0, 1  
    beq .Lblk1  
    cmp r0, 2  
    beq .Lblk2  
    // fall through  
    // default 3  
    b .Lendsw // break  
.Lblk1:  
    // block 1  
    b .Lendsw // break  
.Lblk2:  
    // block 2  
    // fall through  
    // NO b .Lendsw  
.Lendsw:
```

Branch block



Approach 2 – if else equiv.

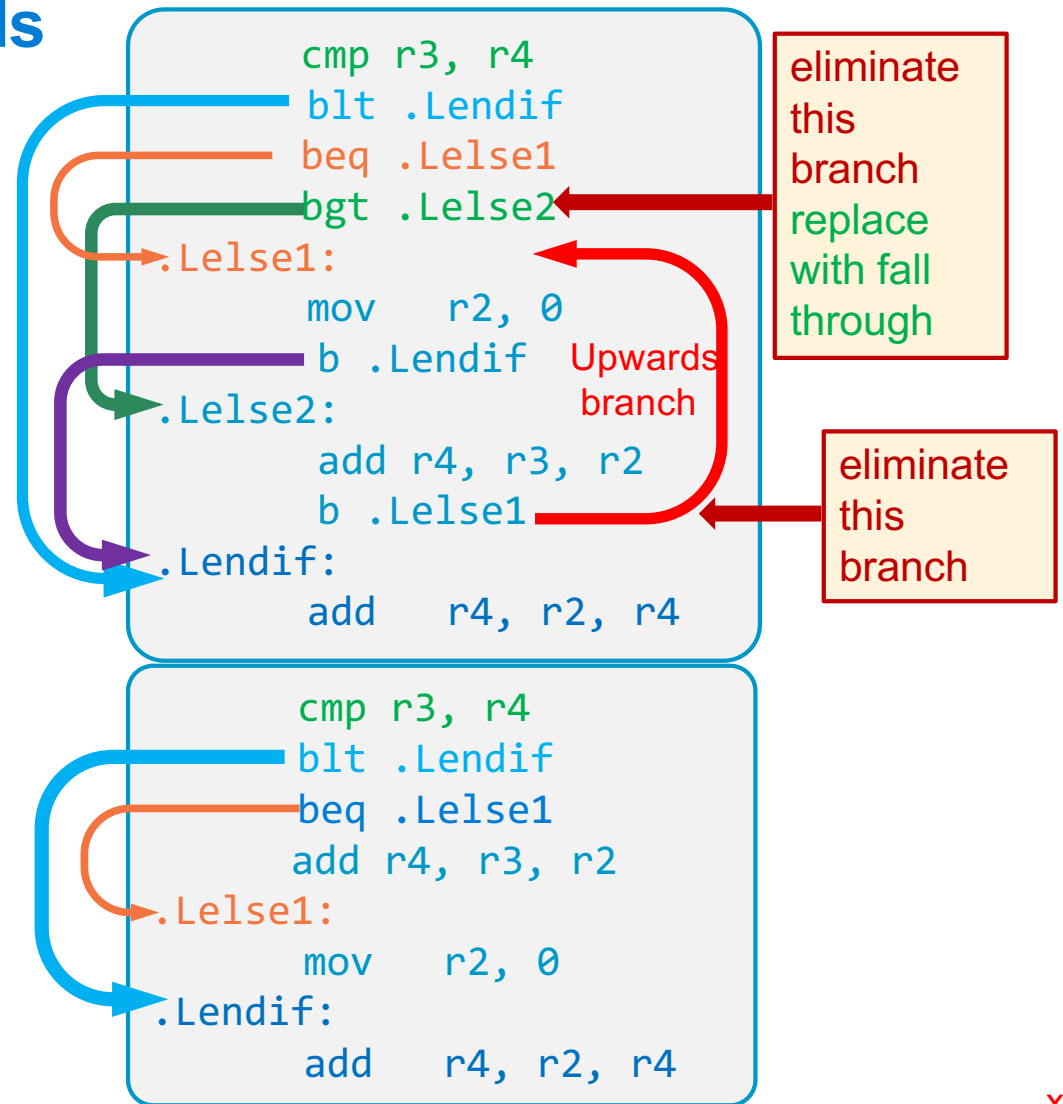
```
    cmp r0, 1  
    bne .Lblk2  
    // block 1  
    b .Lendsw // break  
.Lblk2:  
    cmp r0, 2  
    bne .Ldefault  
    // block 2  
    b .Lendsw // break  
.Ldefault:  
    // default 3  
    // fall through  
    // NO b .Lendsw  
.Lendsw:
```



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



Program Flow – 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
```

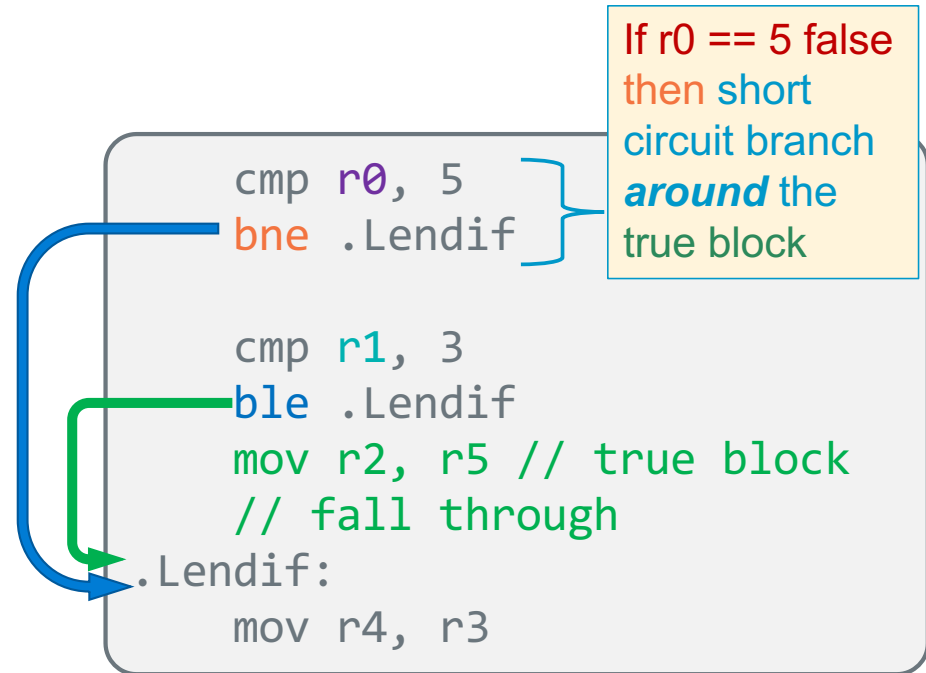


- Each expression argument is evaluated **in sequence** from left to right including any **side effects** (modified using parenthesis), **before** (optionally) evaluating the next expression argument
- If after evaluating an argument, the **value of the entire expression can be determined**, then the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

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

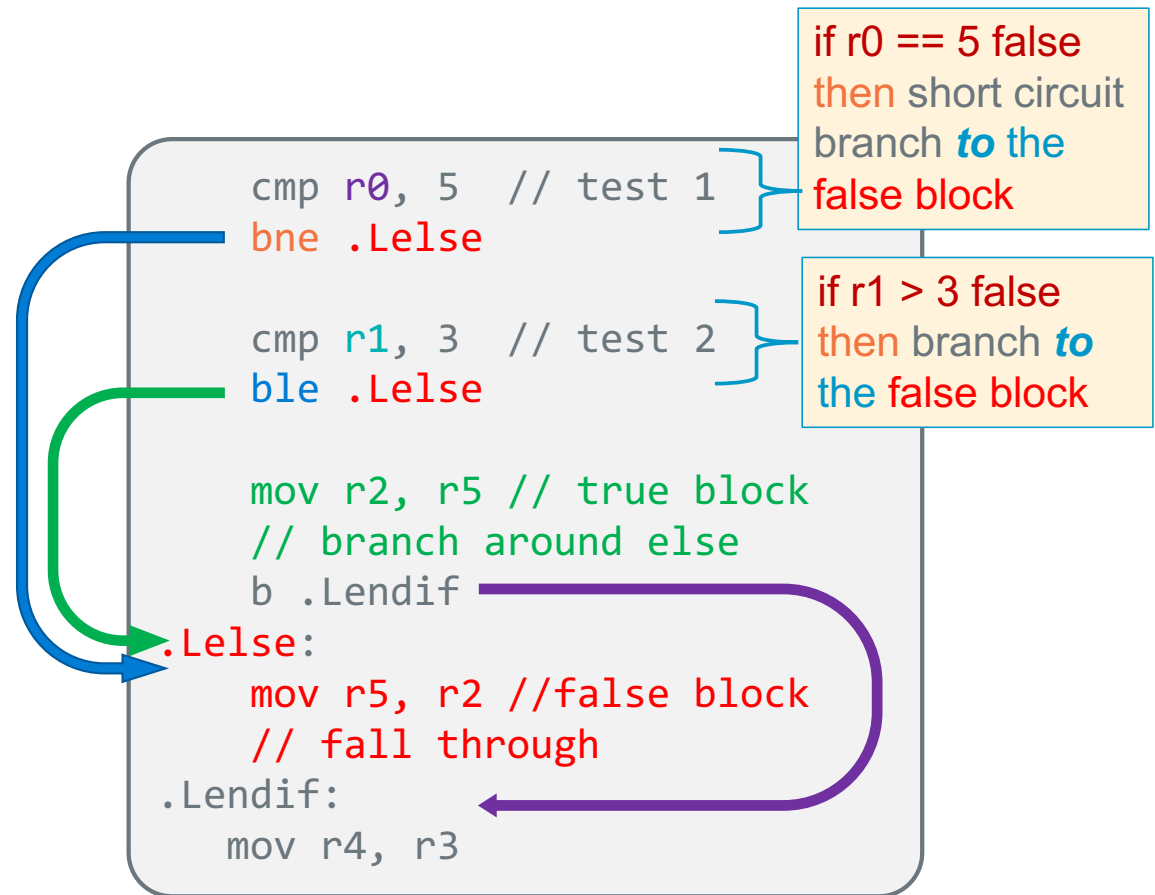
Program Flow – If statements && compound tests - 1

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



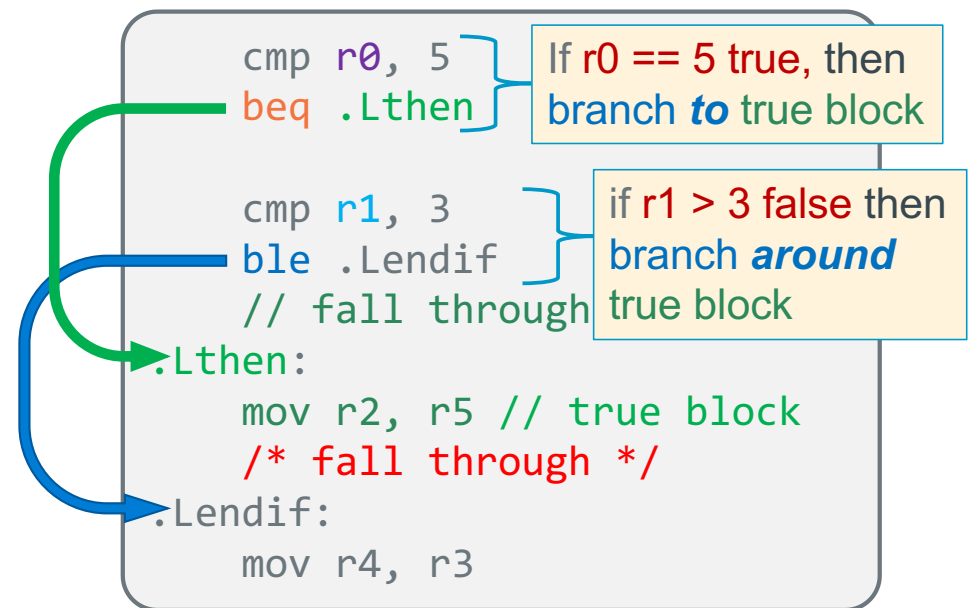
Program Flow – If statements && compound tests - 2

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



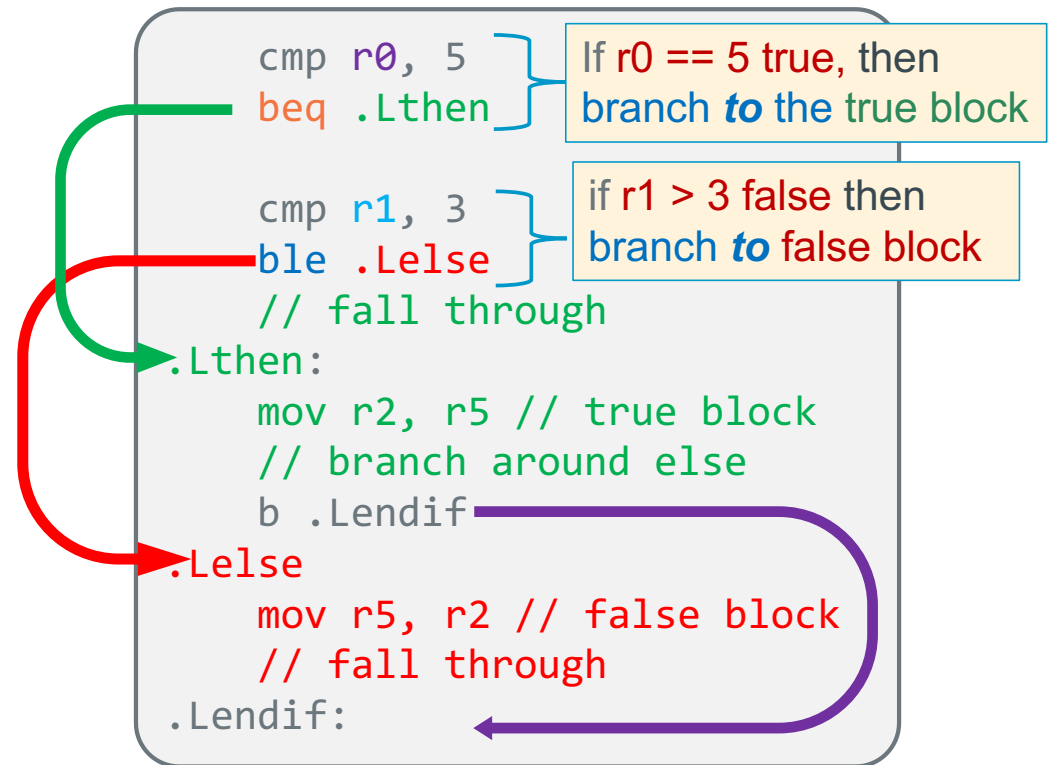
Program Flow – If statements || compound tests - 1

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



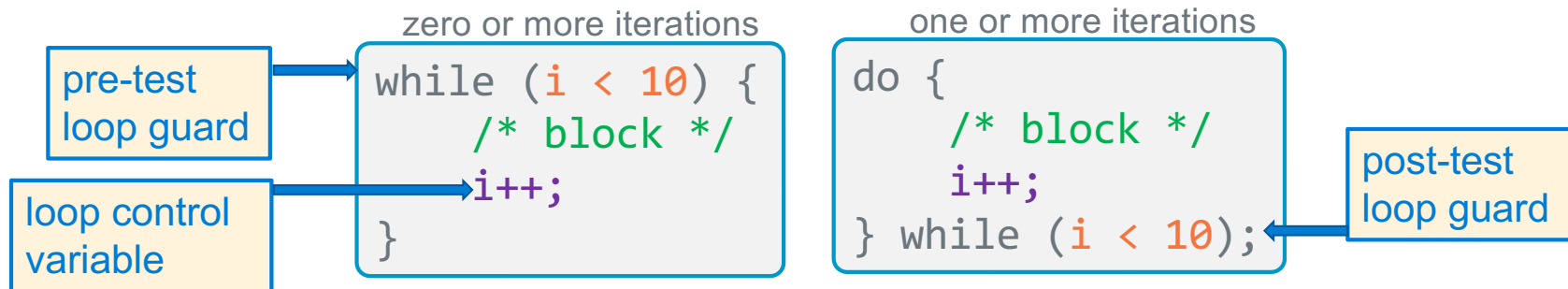
Program Flow – If statements || compound tests - 2

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

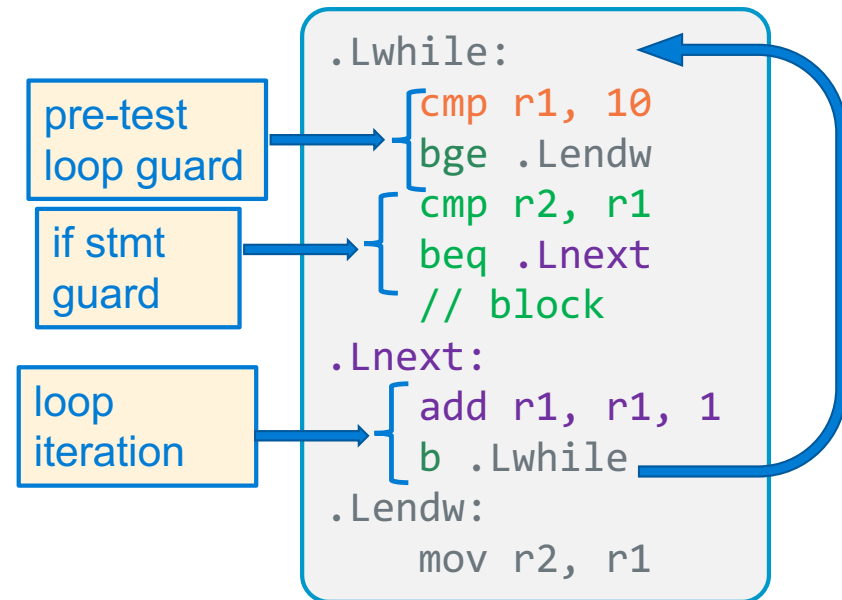
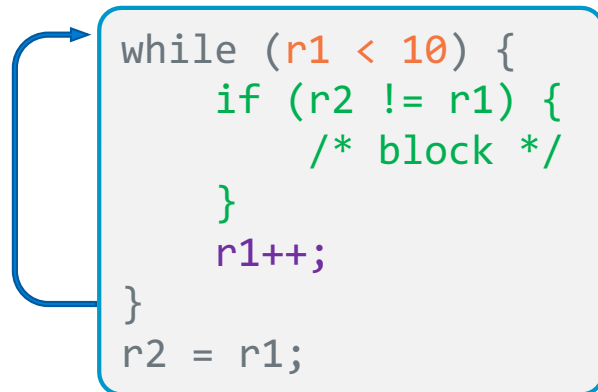
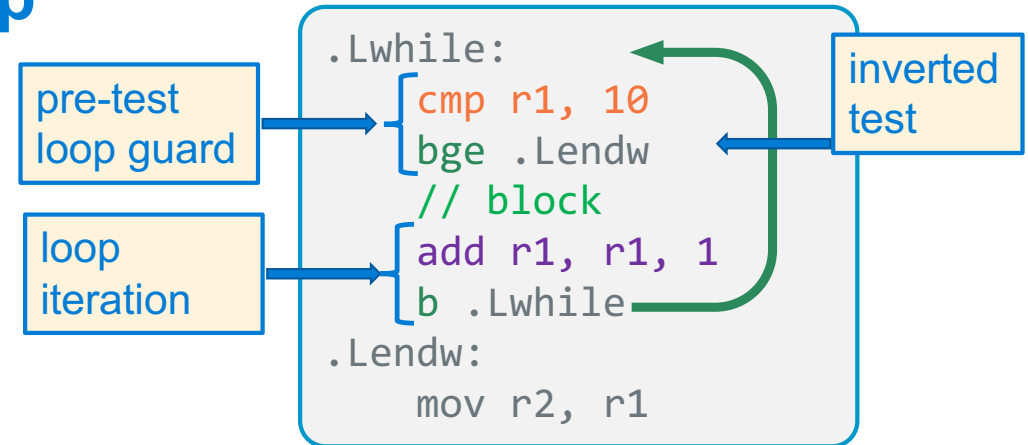
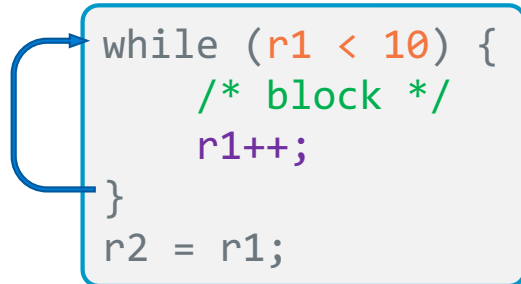


Program Flow – Pre-test and Post-test Loop Guards

- 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



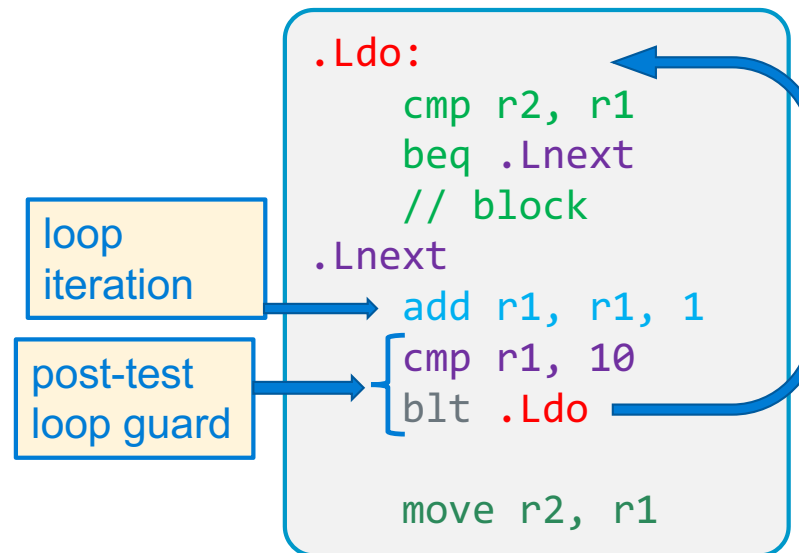
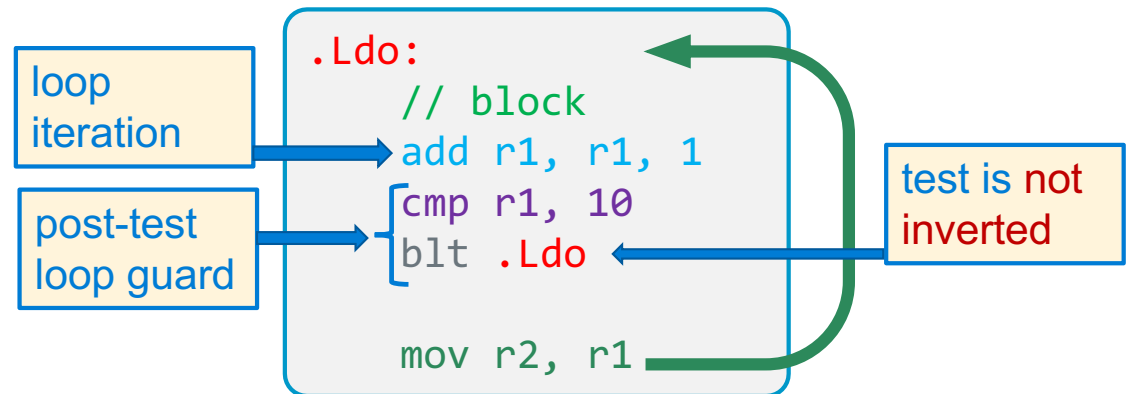
Pre-Test Guards - While Loop



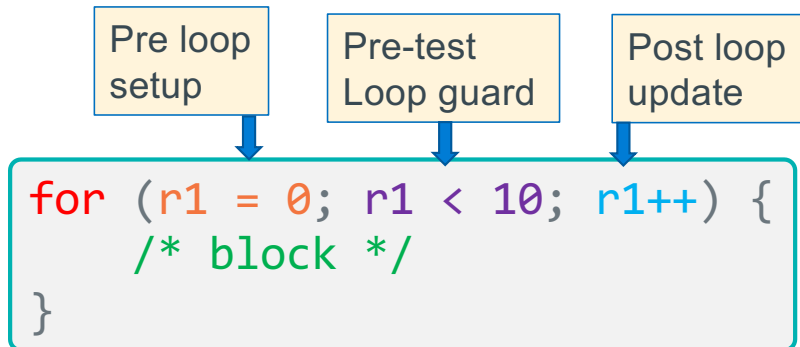
Post-Test Guards – Do While Loop

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

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

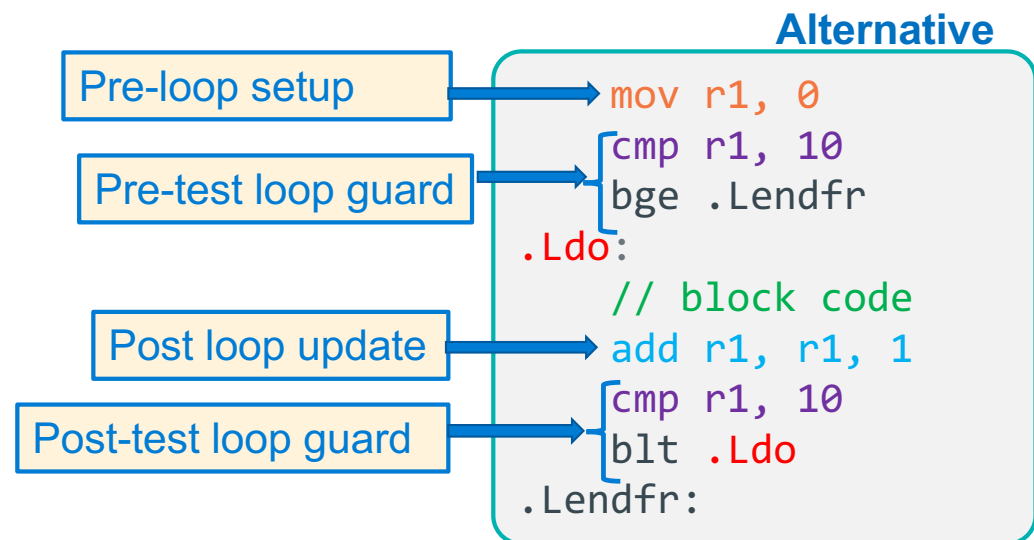
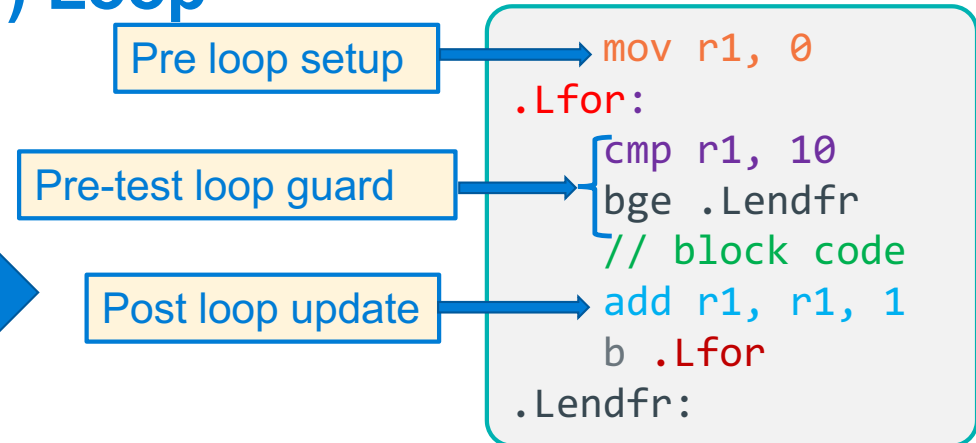


Program Flow – Counting (For) Loop




A **counting loop** has three parts:

1. Pre-loop setup
 2. Pre-test loop guard conditions
 3. Post-loop update
- Alternative:
 - move Pre-test loop guard before the loop
 - Add post-test loop guard
 - *converts* to *do while*
 - **removes** an **unconditional branch**



Nested loops

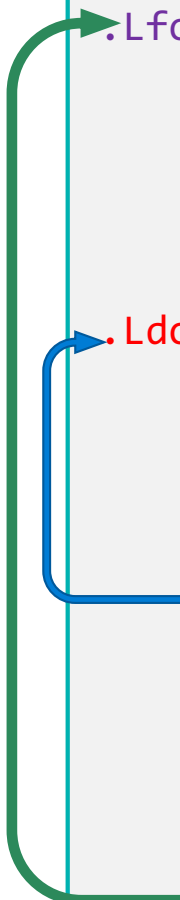
```
for (r3 = 0; r3 < 10; r3++) {  
    r0 = 0;  
  
    do {  
        r0 = r0 + r1++;  
    } while (r1 < 10);  
  
    // fall through  
    r2 = r2 + r1;  
}
```



r5 = r0;

- Nest loop blocks as you would in C or Java
- **Do not branch into the middle of a loop**, this is hard to read and is prone to errors

```
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:
```

Version 1.05

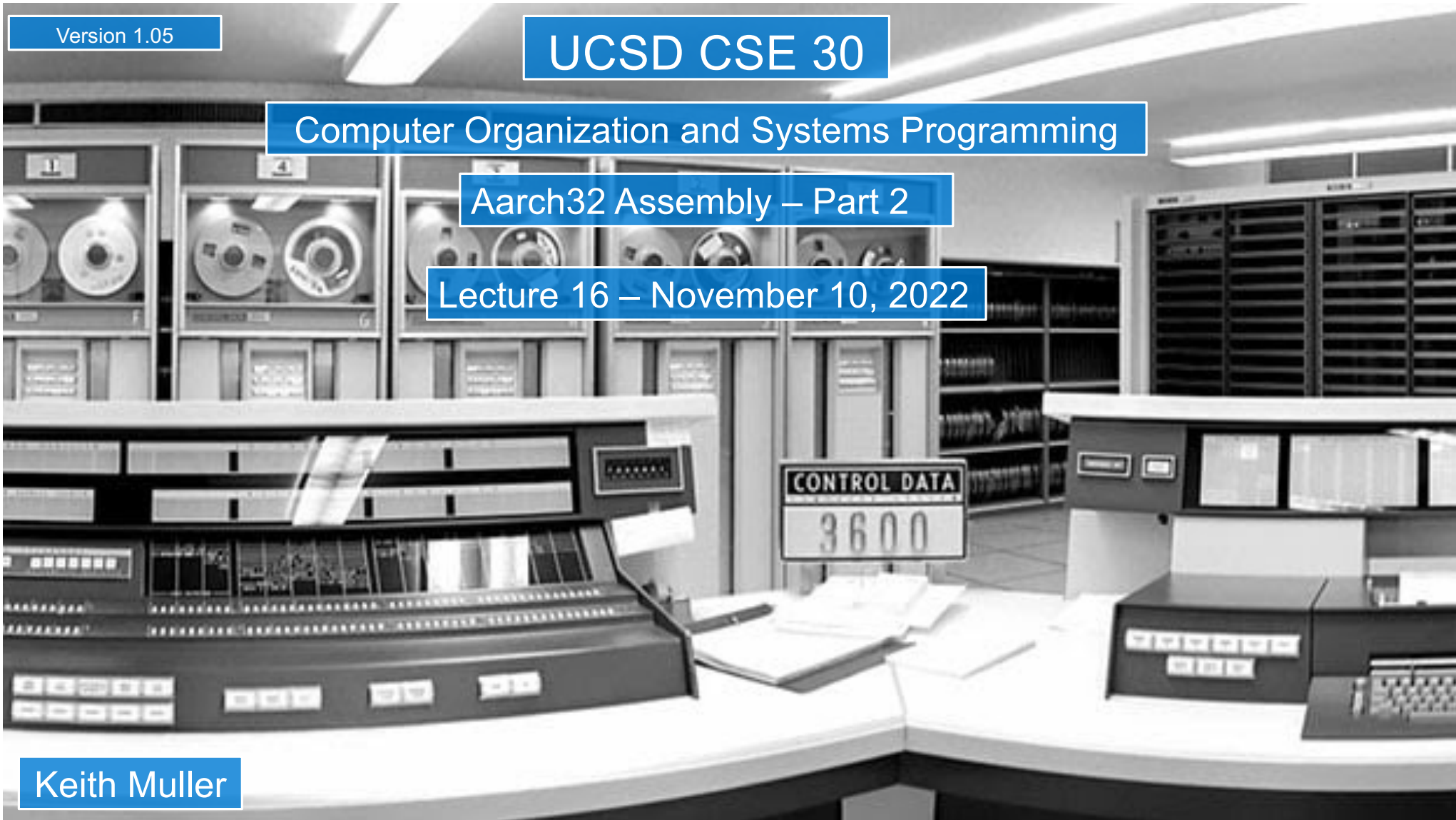
UCSD CSE 30

Computer Organization and Systems Programming

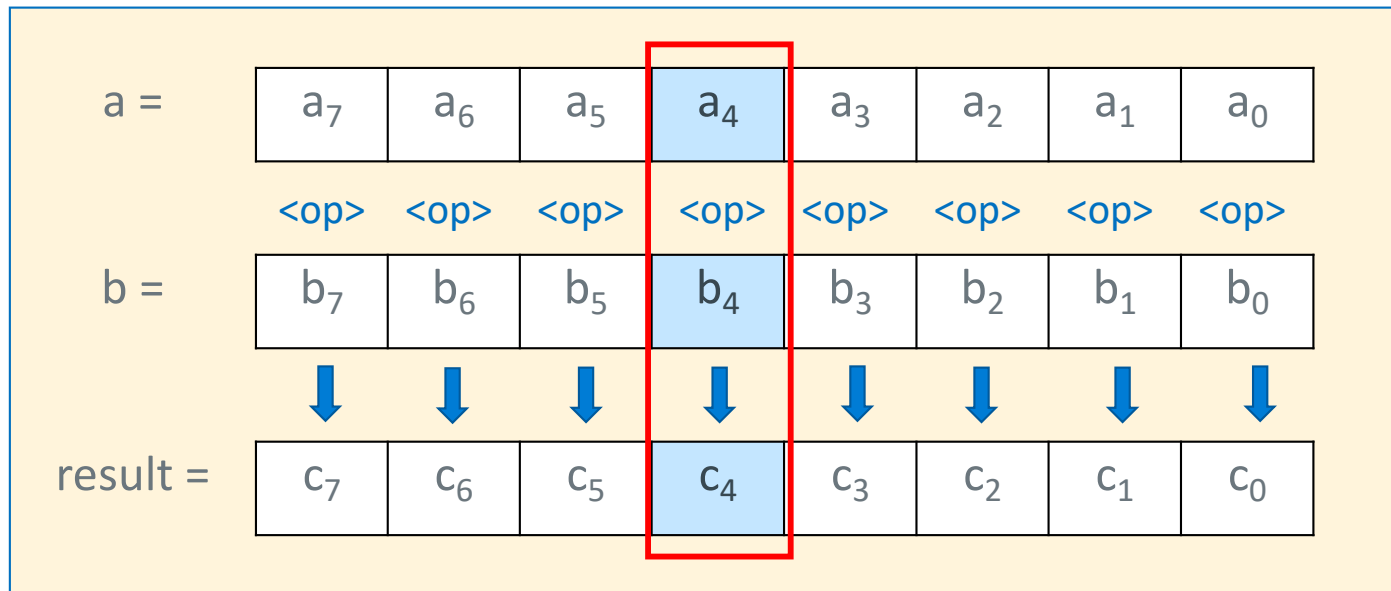
Aarch32 Assembly – Part 2

Lecture 16 – November 10, 2022

Keith Muller



What is a Bitwise Operation?



- Bitwise operators are applied independently to each of the corresponding bit positions in each variable
- Each bit position of the result depends only on bits in the **same bit position** within the operands

Bitwise (Bit to Bit) Operators in C

output = \sim a;

a	\sim a
0	1
1	0

output = a & b;

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

& with 1 to let a bit through
& with 0 to set a bit to 0

output = a | b;

a	b	a b
0	0	0
0	1	1
1	0	1
1	1	1

| with 1 to set a bit to 1
| with 0 to let a bit through

output = a ^ b; //EOR

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

^ with 1 will flip the bit
^ with 0 to let a bit through

Bitwise
NOT

\sim 1100

0011

Bitwise
AND

0110
& 1100

0100

Bitwise
OR

0110
1100

1110

Bitwise
EOR

0110
^ 1100

1010

Bitwise Not (vs Boolean Not)

in C
int output = ~a;

a	~a
0	1
1	0

Bitwise NOT

~ 1100
-- --
0011

	Bitwise Not
number	0101 1010 0101 1010 1111 0000 1001 0110
~number	1010 0101 1010 0101 0000 1111 0110 1001

Meaning	Operator	Operator	Meaning
Boolean NOT	!b	~b	Bitwise NOT

Boolean operators act on the entire value not the individual bits

Type	Operation	result
bitwise	~0x01	1111 1111 1111 1111 1111 1111 1111 1110
Boolean	!0x01	0000 0000 0000 0000 0000 0000 0000 0000

First Look: Copying Values To Registers – MVN (not)

mvn r0, r1

```
// Copies all 32 bits  
// of the value held  
// in register r1 into  
// the register r0  
// then does a bitwise NOT
```

register r1



register r0

mvn r0, 12

```
// Expands an imm8 value 0x0c  
// stored in the instruction  
// into a register then does  
// a bitwise NOT
```

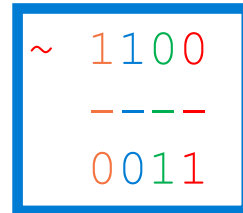
register r0

0x0c



0xffff fff3

Bitwise NOT



- A **bitwise NOT** operation

0x 0c

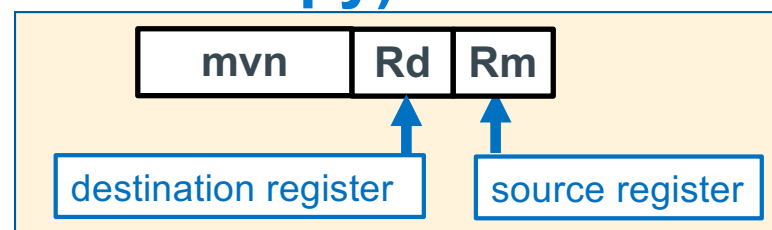
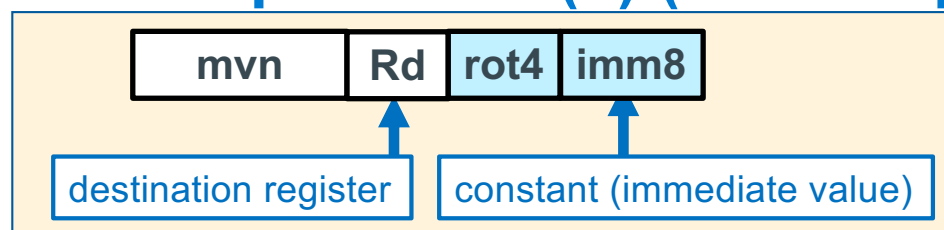
↓ imm8 expansion

0x0000000c

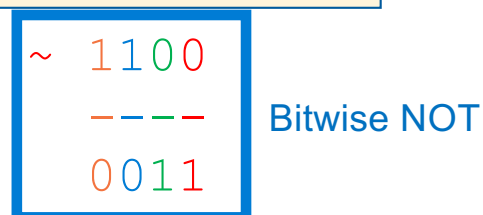
↓ bitwise not

0xfffffffff3

mvn – Copies NOT (~) (1's Complement Copy)

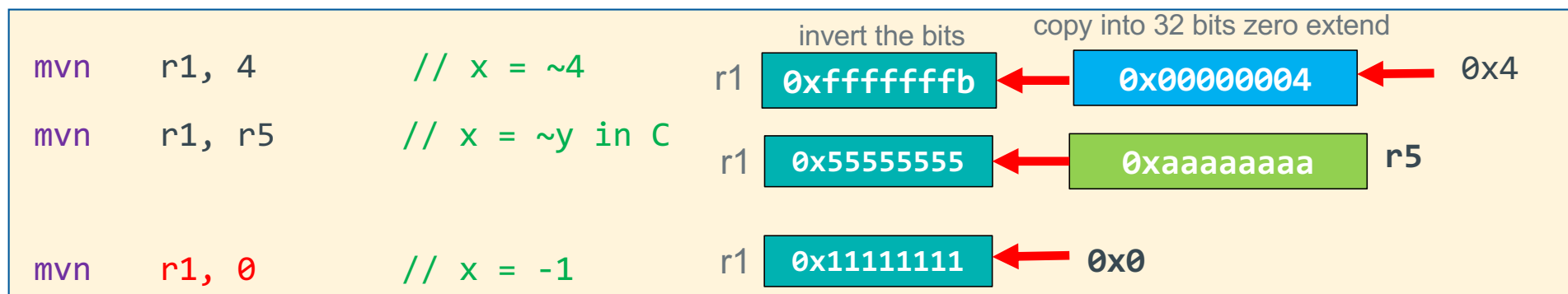


```
mvn Rd, constant // Rd = constant
mvn Rd, Rm       // Rd = Rm
```



bitwise NOT operation. Immediate (constant) version copies to 32-bit register, then does a bitwise NOT

imm8	extended imm8	inverted imm8	signed base 10
0x00	0x00 00 00 00	0xff ff ff ff	-1
0xff	0x00 00 00 ff	0xff ff ff 00	-256



Bitwise versus C Boolean Operators

Meaning	Operator	Operator	Meaning
Boolean AND	<code>a && b</code>	<code>a & b</code>	Bitwise AND
Boolean OR	<code>a b</code>	<code>a b</code>	Bitwise OR
Boolean NOT	<code>!b</code>	<code>~b</code>	Bitwise NOT

Boolean operators **act on the entire value not the individual bits**

& versus &&

`0x10 & 0x01 = 0x00 (bitwise)`

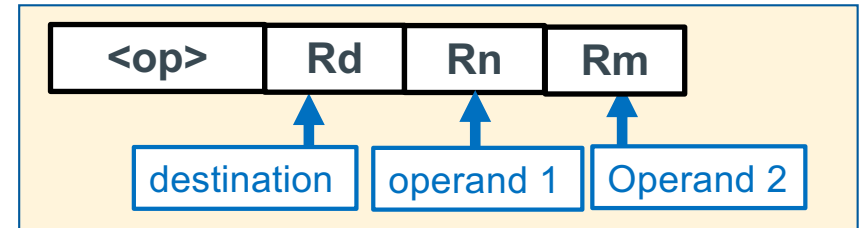
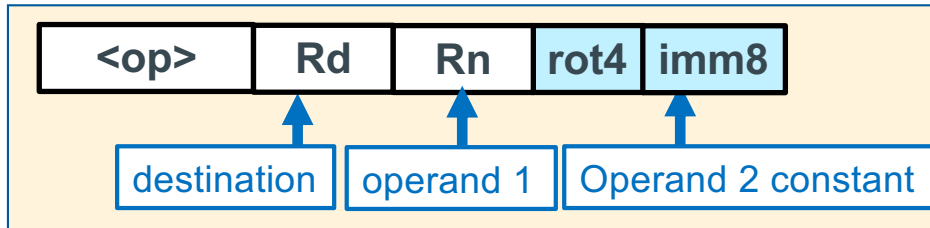
`0x10 && 0x01 = 0x01 (Boolean)`

! versus ~

`~0x01 = 0xfffffffffe (bitwise)`

`!0x01 = 0x0 (Boolean)`

Bitwise Instructions



`<op> Rd, Rn, constant // Rd = Rn <op> constant`
`<op> Rd, constant // Rd = Rd <op> constant`
`<op> Rd, Rn, Rm // Rd = Rn <op> Rm`

Bytes: $0 \leq \text{imm8} \leq 255$ + values from "rotating" rot 4 bits

Bitwise <code><op></code> description	C Syntax	Arm <code><op></code> Syntax <i>Op2: either register or constant value</i>	Operation
Bitwise AND	<code>a & b</code>	<code>and Rd, Rn, Op2</code>	$R_d = R_n \& Op2$
Bit Clear each bit in Op2 that is a 1, the same bit in R_d , is cleared	<code>a & ~b</code>	<code>bic Rd, Rn, Op2</code>	$R_d = R_n \& \sim Op2$
Bitwise OR	<code>a b</code>	<code>orr Rd, Rn, Op2</code>	$R_d = R_n Op2$
Exclusive OR	<code>a ^ b</code>	<code>eor Rd, Rn, Op2</code>	$R_d = R_n ^ Op2$

The act (operation) of *Masking*



- Bit masks access/modify specific bits in memory
- Masking act of applying a mask to a value with a specific op:
 - **orr**: 0 passes bit unchanged, 1 sets bit to 1 `(a = b | c; // in C)`
 - **eor**: 0 passes bit unchanged, 1 inverts the bit `(a = b ^ c; // in C)`
 - **bic**: 0 passes bit unchanged, 1 clears it `(a = b & ~c; // in C)`
 - **and**: 0 clears the bit, 1 passes bit unchanged `(a = b & c; // in C)`

Mask on and Mask off

force bits to 1 "mask on" operation

- 1 to **set a bit to 1**
- 0 to let a **bit through unchanged**

```
orr r1, r2, r3
```

```
r1 = r2 | r3; // in C
```

Example: force lower 16 bits to 1

DATA: r2 0xab ab ab 77

orr

MASK: r3 0x00 00 ff ff

unchanged

forces to a 1

RSLT: r1 0xab ab ff ff

Example: force lower 8 bits to 1

DATA: r2 0xab ab ab 77

```
orr r1 r2, 0xff
```

```
r1 = r2 | 0xff; // in C
```

RSLT: r1 0xab ab ff ff

Mask on and Mask off

force bits to 0 "mask off" operation

- 0 to **set a bit to 0** ("clears the bit")
- 1 to let a **bit through unchanged**

and r1, r2, r3

r1 = r2 & r3; // in C

Example: force lower 8 bits to 0

DATA: r2 0xab ab ab 77

and

MASK: r3 0xff ff ff 00

unchanged

forces to a 0

RSLT: r1 0xab ab ab 00

Example: force lower 8 bits to 0

DATA: r2 0xab ab ab 77

and r1 r2, 0xffffffff00

r1 = r2 & 0xffffffff00; // in C

RSLT: r1 0xab ab ab 00

Mask off versus Bit Clear

force bits to 0 "**mask off**" operation

- 0 to **set a bit to 0** ("clears the bit")
- 1 to let a **bit through unchanged**

and r1, r2, r3

r1 = r2 & r3; // In C

clear specific bits which are 1 in a mask, 0 bits in the mask are unchanged

r1 = r2 & ~r3; // in C

bic r1, r2, r3

Example: force only bit 5 to 0

DATA: r2 0xab ab ab 77 77: 0111 0111

and

MASK: r3 0xff ff ff df df: 1101 1111

unchanged

forces to a 0

RSLT: r1 0xab ab ab 57 57: 0101 0111

r1 = r2 & 0xffffffffdf; // In C

clear bit 5 to a 0 without changing the other bits

DATA: r2 0xab ab ab 77

bic

r3: 0010 0000
~r3: 1101 1111

MASK: r3 0x00 00 00 20

unchanged

clears bit 5

RSLT: r1 0xab ab ab 57

r1 = r2 & ~0x00000020; // in C

Extracting (Isolate) a Field of Bits with a mask

extract top 8 bits of r2 into r1

- 0 to **set a bit to 0** ("clears the bit")
- 1 to let a **bit through unchanged**

and r1, r2, r3

DATA: r2 0xab ab ab 77

and

MASK: r3 0xff 00 00 00

unchanged

RSLT: r1 0xab 00 00 00

extract top 8 bits of r2 into r1

DATA: r2 0xab ab ab 77

and r1, r2, 0xff000000

RSLT: r1 0xab 00 00 00

r1 = r2 & 0xff000000; // in C

Finding if a bit is set

query the status of a bit "**bit status**" operation

- 0 to **set a bit to 0** ("clears the bit")
- 1 to let a **bit through unchanged**

```
and r1, r2, 0x02
```

```
cmp r1, 0
```

```
beq .Lendif
```

```
// code for is set
```

```
.Lendif:
```

```
unsigned int r1, r2;  
// code  
r1 = r2 & 0x02  
if (r1 != 0) {  
    // code for is set  
}
```

Example is bit 1 set

DATA: r2 0xab ab ab 77

and

MASK: 0x00 00 00 02 is bit 1 set?

forces to a 0 unchanged

RSLT: r1 0x00 00 00 02 != 0 if set

```
unsigned int r2;  
// code  
if ((r2 & 0x02) != 0) {  
    // code for is set  
}
```

Even/Odd

Even or odd, check LSB (same as mod %2)

check LSB (bit 0) if set then odd, else even

```
and r1, r2, 0x01
```

```
cmp r1, 0x01
```

```
bne .Lendif
```

```
// code for handling odd numbers
```

```
.Lendif:
```

```
unsigned int r2;
```

```
// code
```

```
if ((r2 & 0x01) != 0) {
```

```
    // code for handling odd numbers
```

```
}
```

DATA: r2 0xab ab ab 77

and

MASK: r3 0x00 00 00 01 (mod 2 even or odd)

forces to a 0 unchanged

RSLT: r1 0x00 00 00 01 (odd)

MOD %<power of 2>

remainder (mod): num \% d where $\text{num} \geq 0$ and $d = 2^k$
mask = $2^k - 1$ so for mod 16, mask = $16 - 1 = 15$
and r1, r2, r3

Example: %2

DATA: r2 0xab ab ab 77

and

MASK: r3 0x00 00 00 01 (mod 2 even or odd)

forces to a 0 unchanged

RSLT: r1 0x00 00 00 01 (odd)

Example: Mod 16

DATA: r2 0xab ab ab 77

and

MASK: r3 0x00 00 00 0f (mod 16)

forces to a 0 unchanged

RSLT: r1 0xab 00 00 07

Flipping bits: bit toggle Used in PA8

invert (*flip*) bits "bit toggle" operation

- 1 **will flip the bit**
- 0 to **let a bit through**

eor r1, r2, r3

- Observation: When applied twice, it returns the original value (symmetric encoding)
- With a mask of all 1's is a 1's compliment

Example: *flip* the lower 8-bits

eor r1, r2, 0xff

```
unsigned int r1, r2;
r1 = r2 ^ 0xff;
```

Example: invert (*flip*) the lower 8-bits

DATA: r2 0xab ab ab **77** **77: 0111 0111**

eor

MASK: r3 0x00 00 00 **ff**

unchanged

inverts (flips)

RSLT: r1 0xab ab ab **88** **88: 1000 1000**

DATA: r1 0xab ab ab **88**

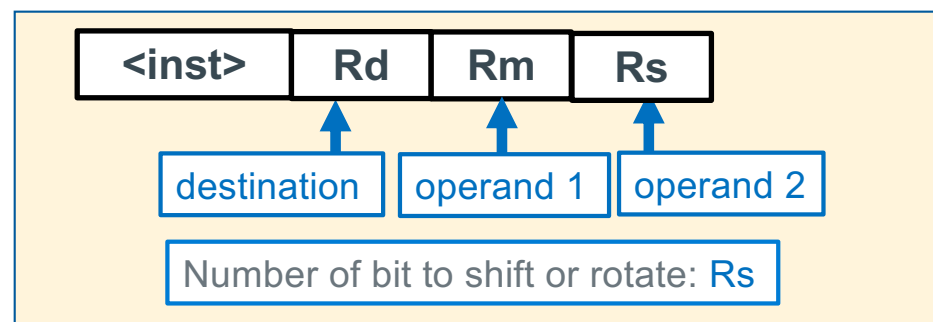
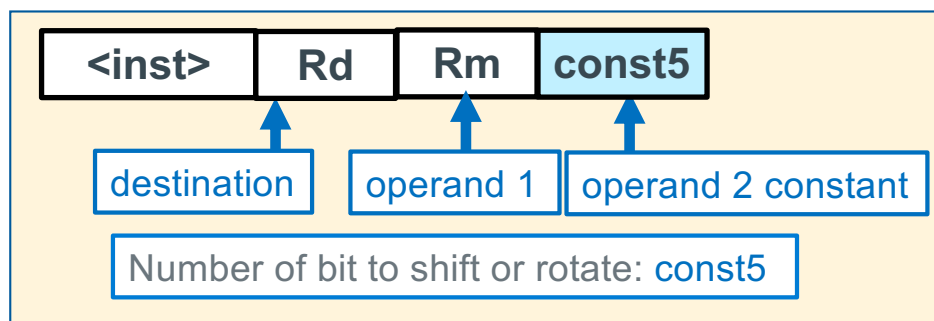
eor

MASK: r3 0x00 00 00 **ff** **apply a 2nd time**

inverts (flips)

RSLT: r1 0xab ab ab **77** **original value!**

Shift and Rotate Instructions



Instruction	Syntax	Operation	Notes	Diagram
Logical Shift Left <i>int x;</i> <i>unsigned int x</i> <i>x << n;</i>	LSL $R_d, R_m, const5$ LSL R_d, R_m, R_s	$R_d \leftarrow R_m \ll const5$ $R_d \leftarrow R_m \ll R_s$	Zero fills shift: 0 - 31	
Logical Shift Right <i>unsigned int x;</i> <i>x >> n;</i>	LSR $R_d, R_m, const5$ LSR R_d, R_m, R_s	$R_d \leftarrow R_m \gg const5$ $R_d \leftarrow R_m \gg R_s$	Zero fills shift: 1 - 32	
Arithmetic Shift Right <i>int x;</i> <i>x >> n;</i>	ASR $R_d, R_m, const5$ ASR R_d, R_m, R_s	$R_d \leftarrow R_m \gg const5$ $R_d \leftarrow R_m \gg R_s$	Sign extends shift: 1 - 32	
Rotate Right <i>unsigned int x;</i> <i>x = (x >> n) (x << (32-n));</i>	ROR $R_d, R_m, const5$ ROR R_d, R_m, R_s	$R_d \leftarrow R_m \text{ ror } const5$ $R_d \leftarrow R_m \text{ ror } R_s$	right rotate rot: 0 - 31	

Shift Operations in C

- n is number of bits to shift a variable x of width w bits
- Shifts by $n < 0$ or $n \geq w$ are *undefined*
- Left shift ($x \ll N$) – **Multiplies by 2^N**
 - Shift N bits left, Fill with 0s on right
- In C: behavior of \gg is determined by compiler
 - gcc: it depends on data type of x (signed/unsigned)
- Right shift ($x \gg N$) - **Divides by 2^N**
 - Logical shift (for unsigned variables)
 - Shift N bits right, Fill with 0s on left
 - Arithmetic shift (for signed variables) – Sign Extension
 - Shift N bits right while **Replicating** the most significant bit on left
 - Maintains sign of x
- In Java: logical shift is \ggg and arithmetic shift is \gg

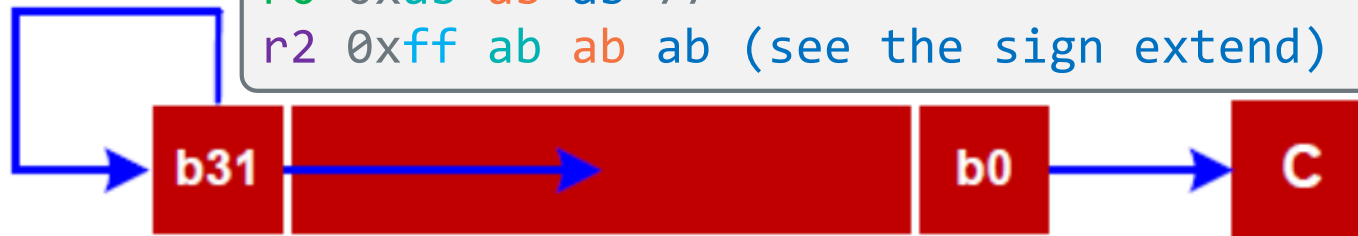


Arithmetic Shift Right (there is no arithmetic shift left)

```
asr r2, r0, 8
```

```
r0 0xab ab ab 77
```

```
r2 0xff ab ab ab (see the sign extend)
```



Test for sign
-1 if r0 negative

```
asr r2, r0, 31  
cmp r2, -1  
bne .Lendif  
//code neg #  
.Lendif:
```

```
r0 0xab ab ab 77  
r2 0xff ff ff ff
```

```
int i;  
//code  
if ((i>>31) == -1) {  
    // code neg #  
}
```

```
int i;  
//code  
if ((i>>31) == 0) {  
    // code pos #  
}
```

Test for sign
0 if r0 positive

```
asr r2, r0, 31  
cmp r2, 0  
bne .Lendif  
//code positive #  
.Lendif:
```

```
r0 0x7b ab ab 77  
r2 0x00 00 00 00
```


Logical Shift & Rotate Operations



```
lsr r2, r0, 8
```

```
r0 0xab ab ab 77
r2 0x00 ab ab ab
```



```
lsl r2, r0, 8
```

```
r0 0xab ab ab 77
r2 0xab ab 77 00
```



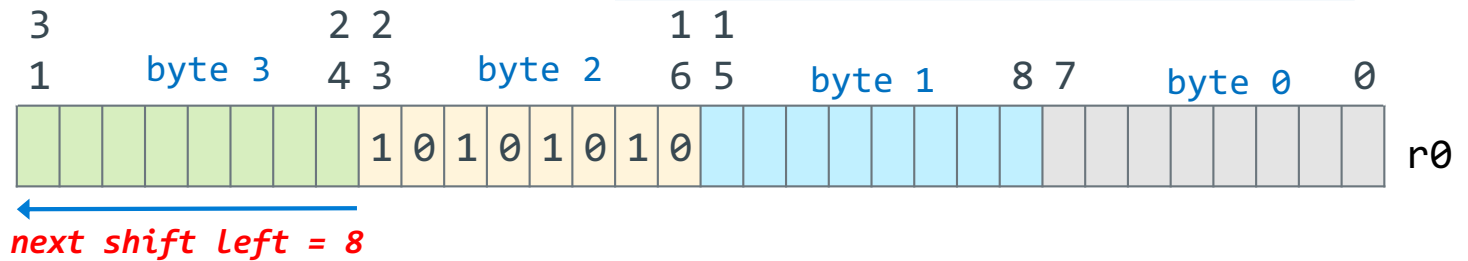
```
ror r2, r0, 8
```

```
r0 0xab ab ab 77
r2 0x77 ab ab ab
```

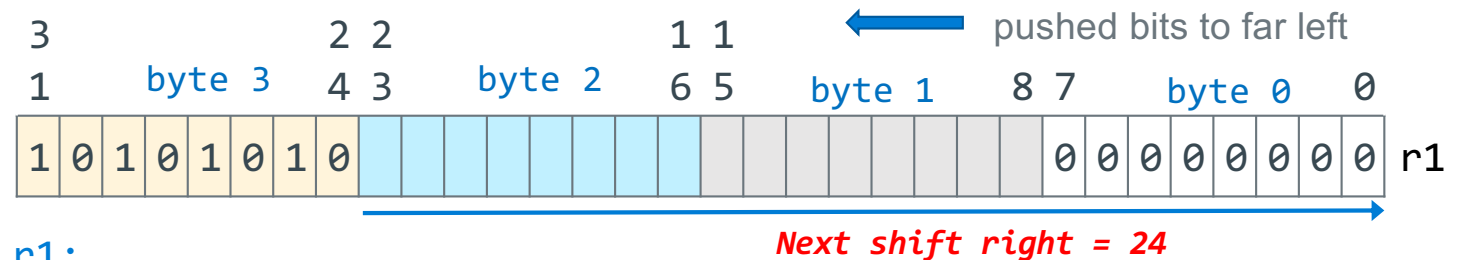
Extracting/Isolating Unsigned Bitfields

Hint: Useful for PA8

- Move byte 2 in r0 to byte 0 in r1



```
lsl r1, r0, 8
```

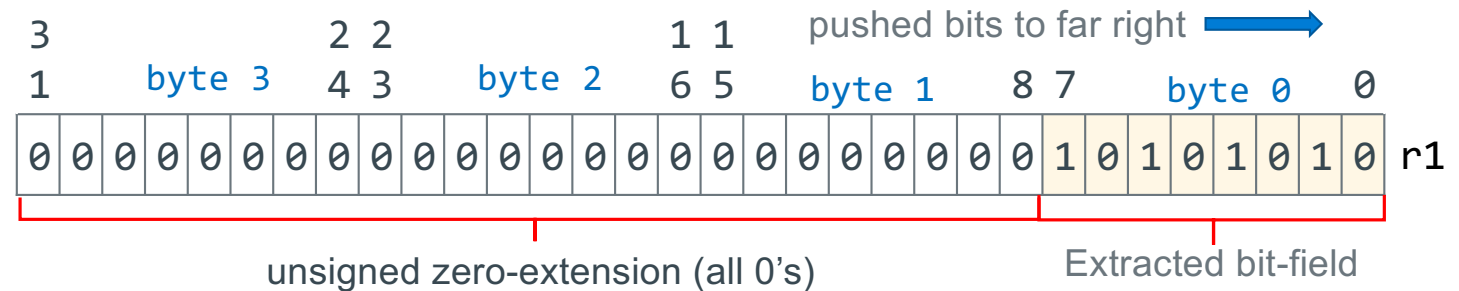


```
unsigned int r0,r1;
```

```
r1 = r0 << 8;
```

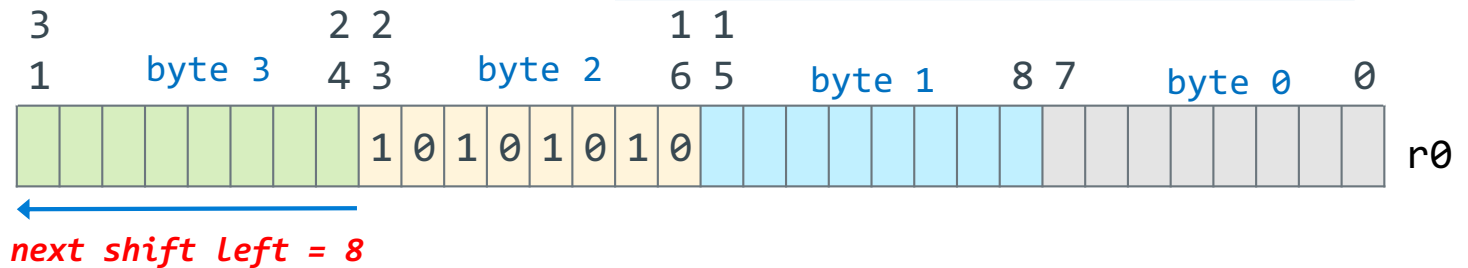
```
lsl r1, r1, 24
```

```
r1 = r1 >> 24;
```

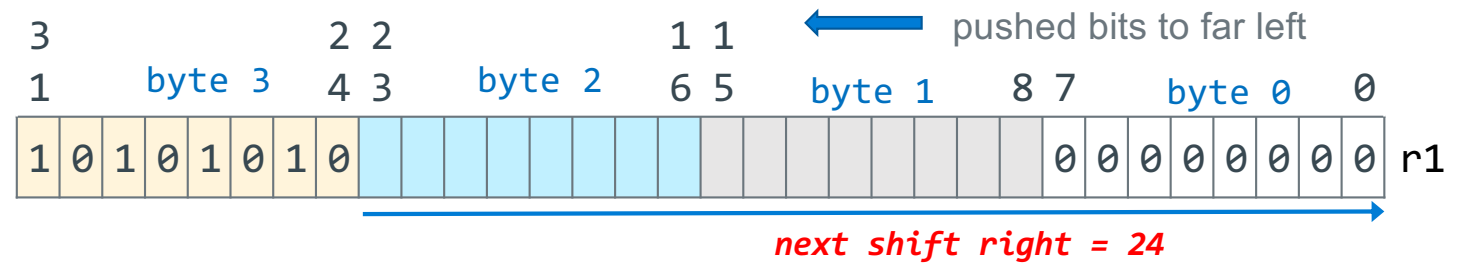


Extracting Signed Bitfields

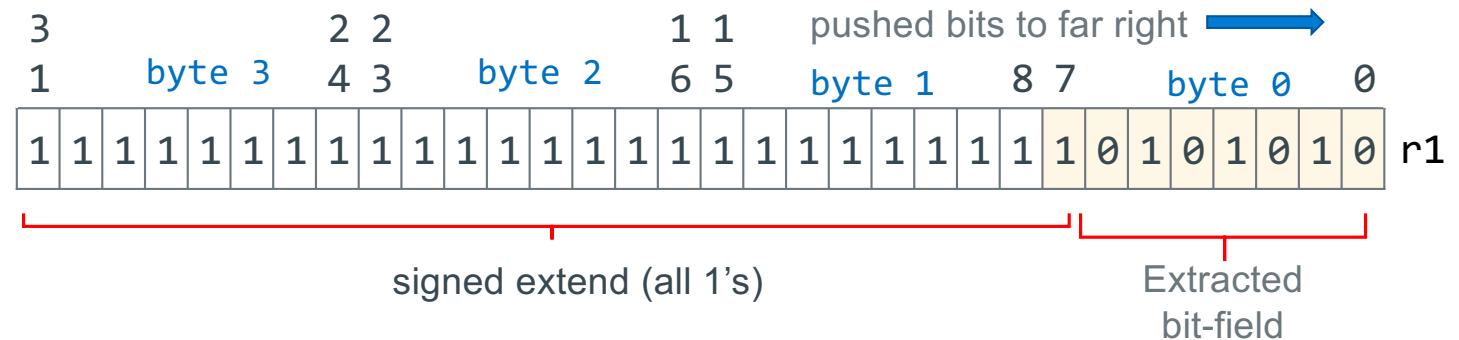
- Move byte 2 in r0 to byte 0 in r1



```
lsl r1, r0, 8
int r0,r1;
r1 = r0 << 8;
```



```
asr r1, r1, 24
r1 = r1 >> 24;
```



Inserting Bitfields – Inserting Source Field into Destination Field

Task: Insert source into destination

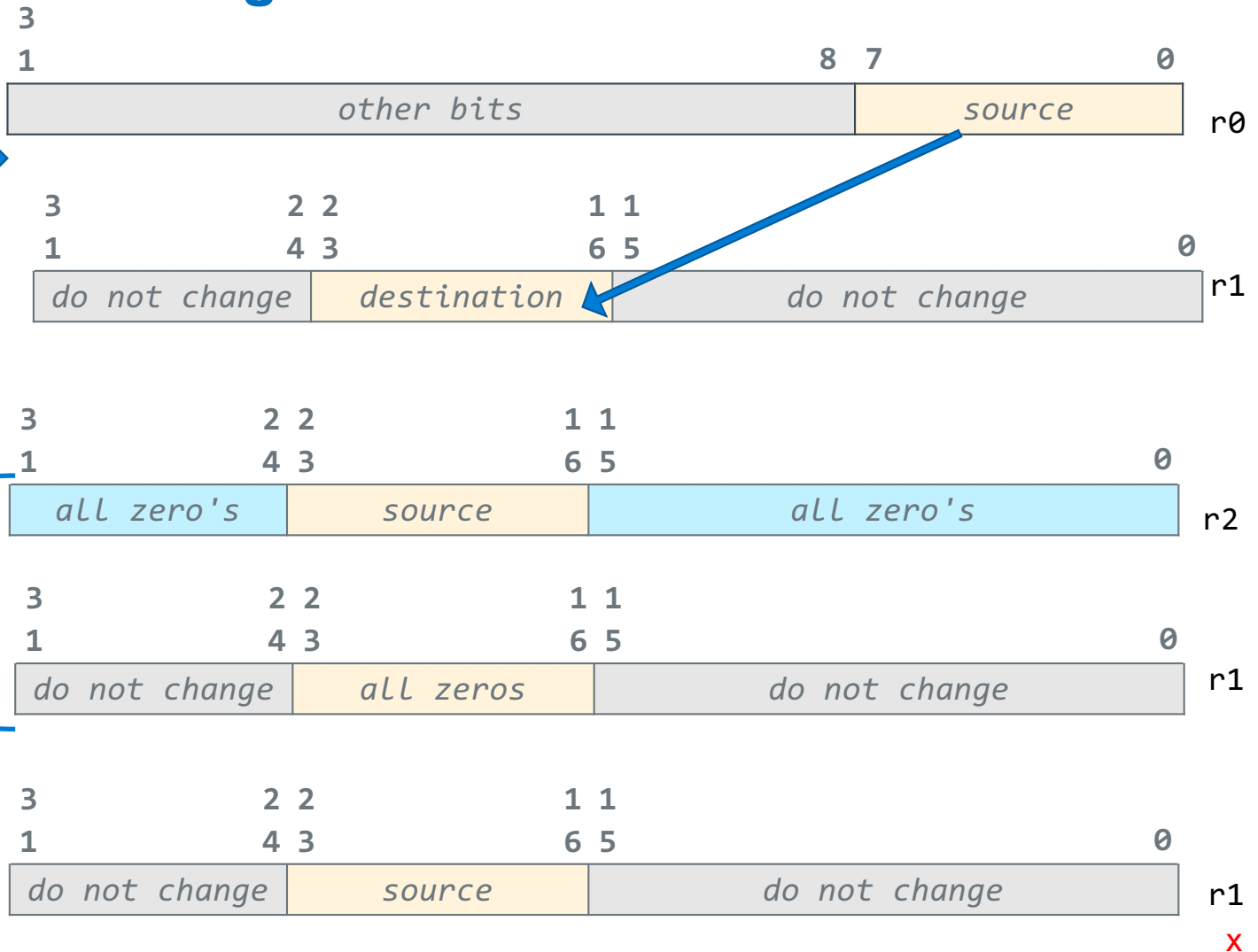
a	b	a b
0	0	0
0	1	1
1	0	1
1	1	1

Approach

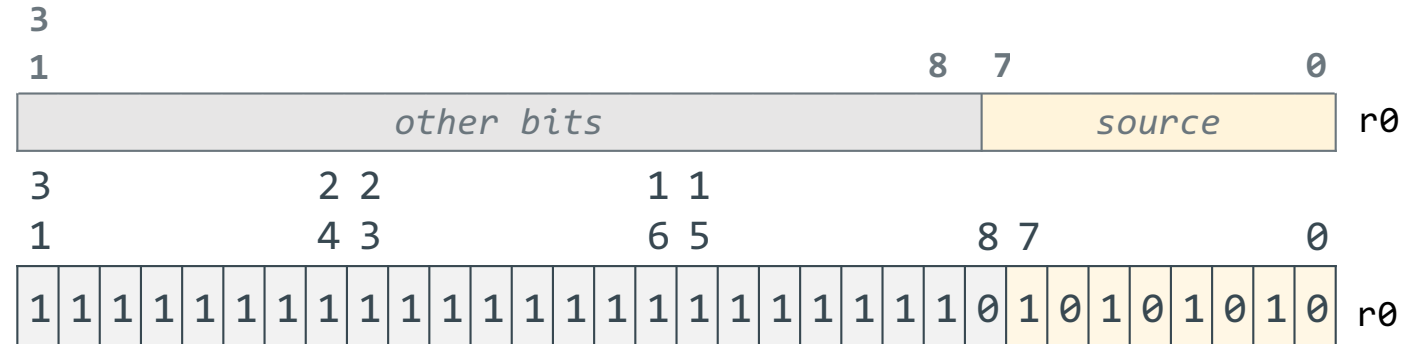
- (1) isolate source field
- (2) clear destination field
- (3) Bitwise **or** together

```
orr    r1, r1, r2
r1 =   r1 | r2;
```

results in



Inserting Bitfields – Isolating the Source Field



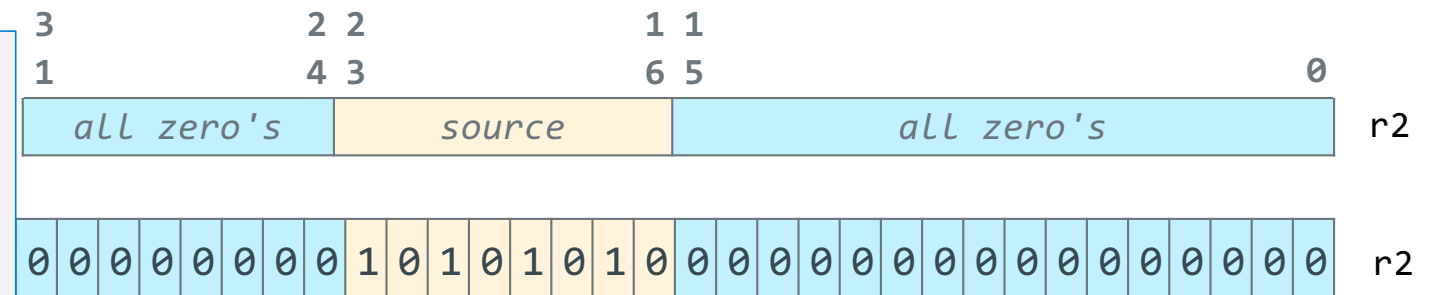
isolate source field

```
lsl    r2, r0, 24
```

```
lsr    r2, r2, 8
```

```
r2 = r0 << 24;
```

```
r2 = r2 >> 8;
```

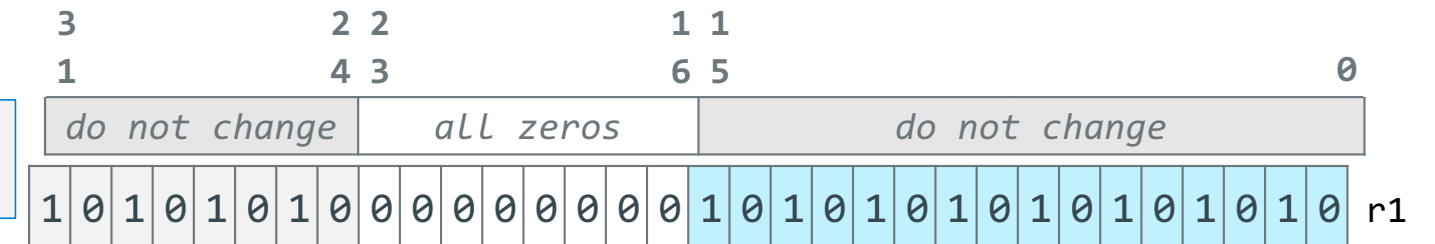
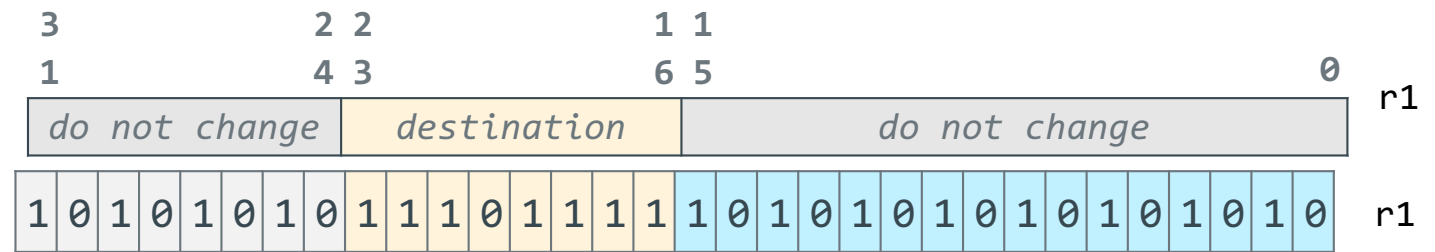


Inserting Bitfields – Clearing the Destination Field

```
clear the
destination field
ror    r1, r1, 24
r1=(r1>>24)|(r1<<8);
```

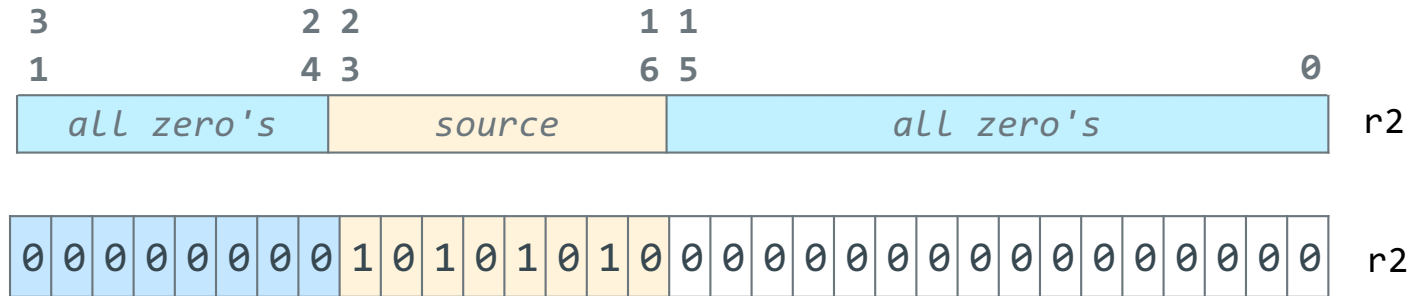
```
lsl    r1, r1, 8
r1 = r1 << 8;
```

```
ror    r1, r1, 16
r1= (r1>>16)|(r1<<16);
```

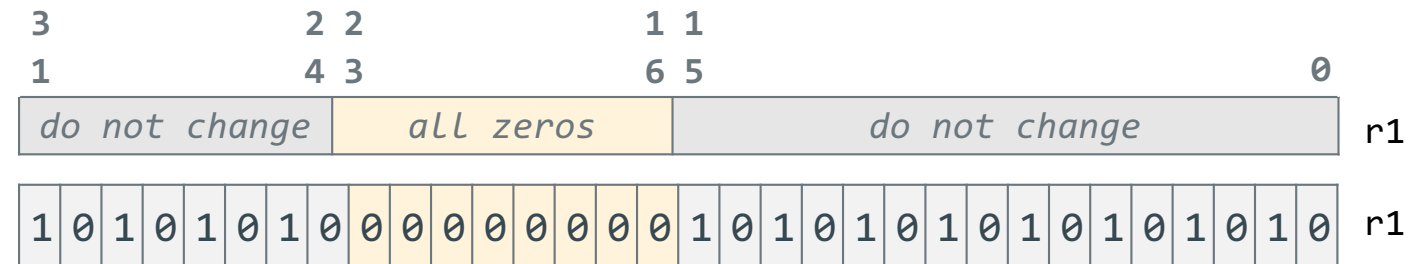


Inserting Bitfields – Combining Isolated Source and Cleared Destination

isolated source



field cleared in
destination



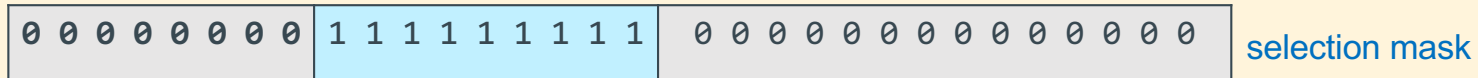
inserted field
orr r1, r1, r0
r1 = r1 | r0;



Masking Summary

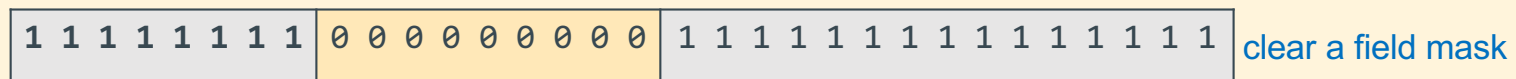
Select a field: Use **and** with a **mask** of one's surrounded by zero's to select the bits that have a 1 in the mask, all other bits will be set to zero

selects this field when used with and

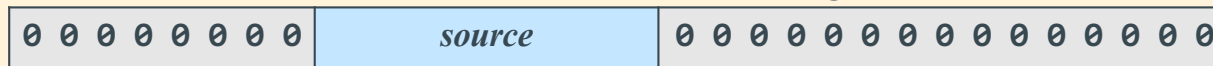


Clear a field: Use **and** with a mask of zero's surrounded by one's to select the bits that have a 1 in the mask, all other bits will be set to zero

clears this field when used with and



Isolate a field: Use **lsl**, **lsl**, **rot** to get a field surrounded by zeros



lsl to get this edge into msb

lsl to get this edge into lsb

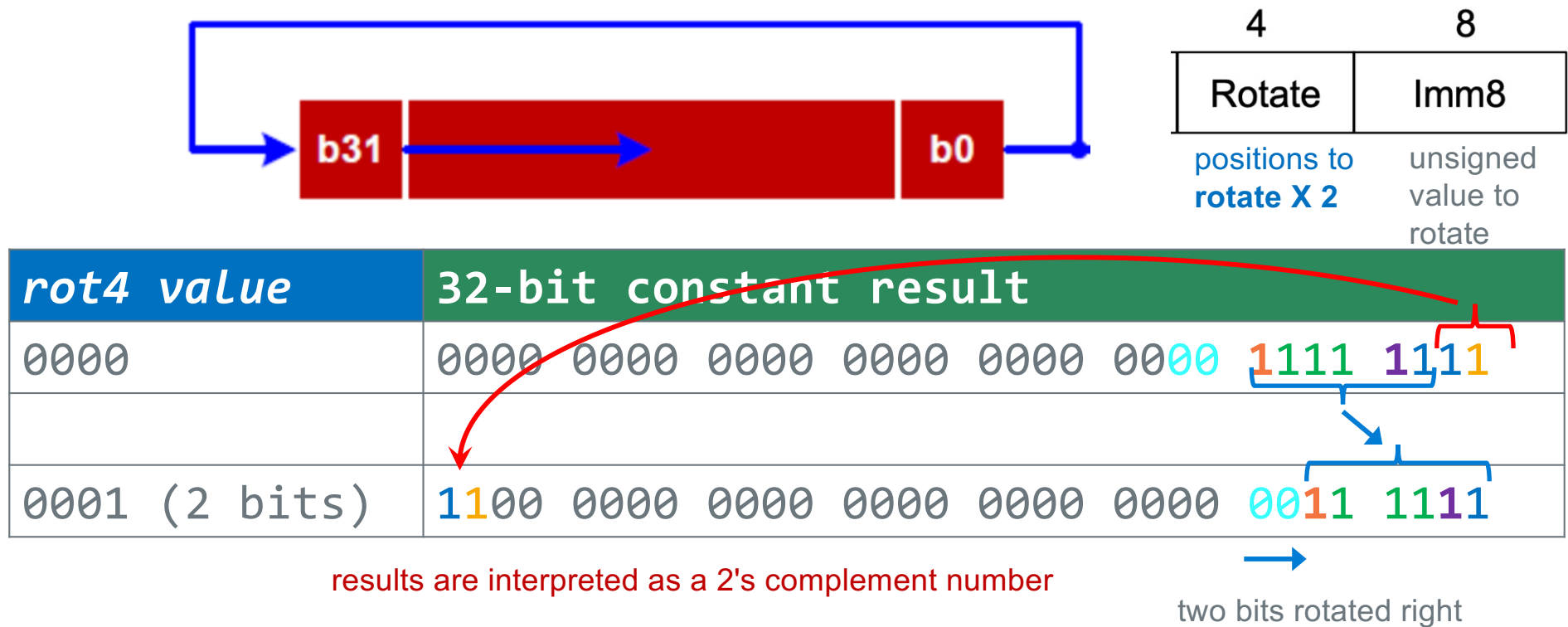
Insert a field: Use **orr** with fields surrounded by zeros



Extra Slides

How are I – Type Constants Encoded in the instruction?

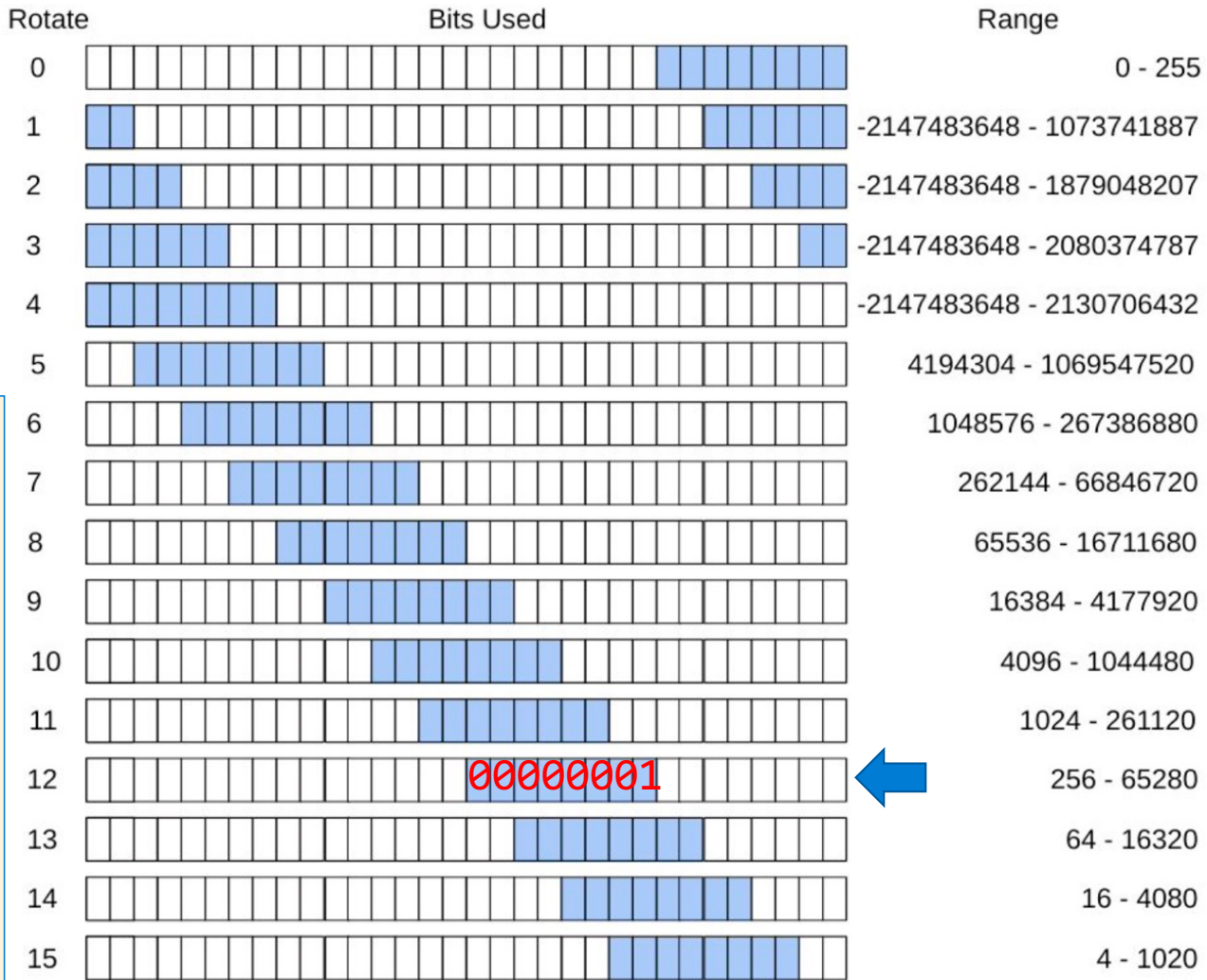
- Aarch32 provides only 8-bits for specifying an immediate constant value
- Without "rotation" immediate values are limited to the range of positive 0-255
- Imm8 expands to 32 bits and does a rotate right to achieve additional constant values (YUCK)



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

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

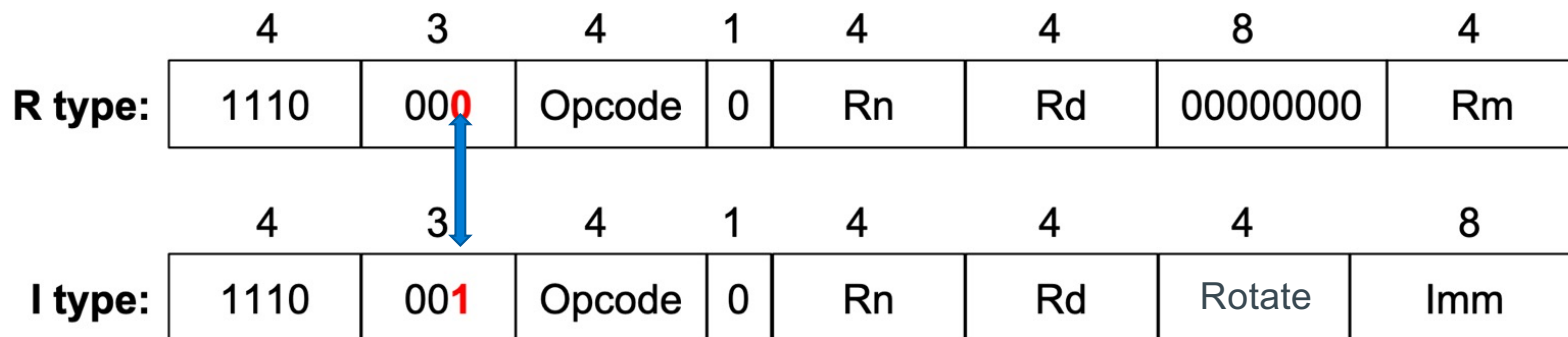
```

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

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

Basic Arm Machine Code Instructions

- Instructions consist of several fields that **encode** the **opcode** and arguments to the opcode
- Special fields enable extended functionality - later
- Several 4-bit **operand** fields for specifying the **source and destination** of the operation, usually one of the 16 registers
- **Embedded constants** ("*immediate values*") of various size and "configuration"
- Basic Data processing instruction formats (below)
- R type instruction: `add r0, r1, r2` // third operand is a register
- I type instruction: `add r0, r0, 1` // third operand is an immediate value

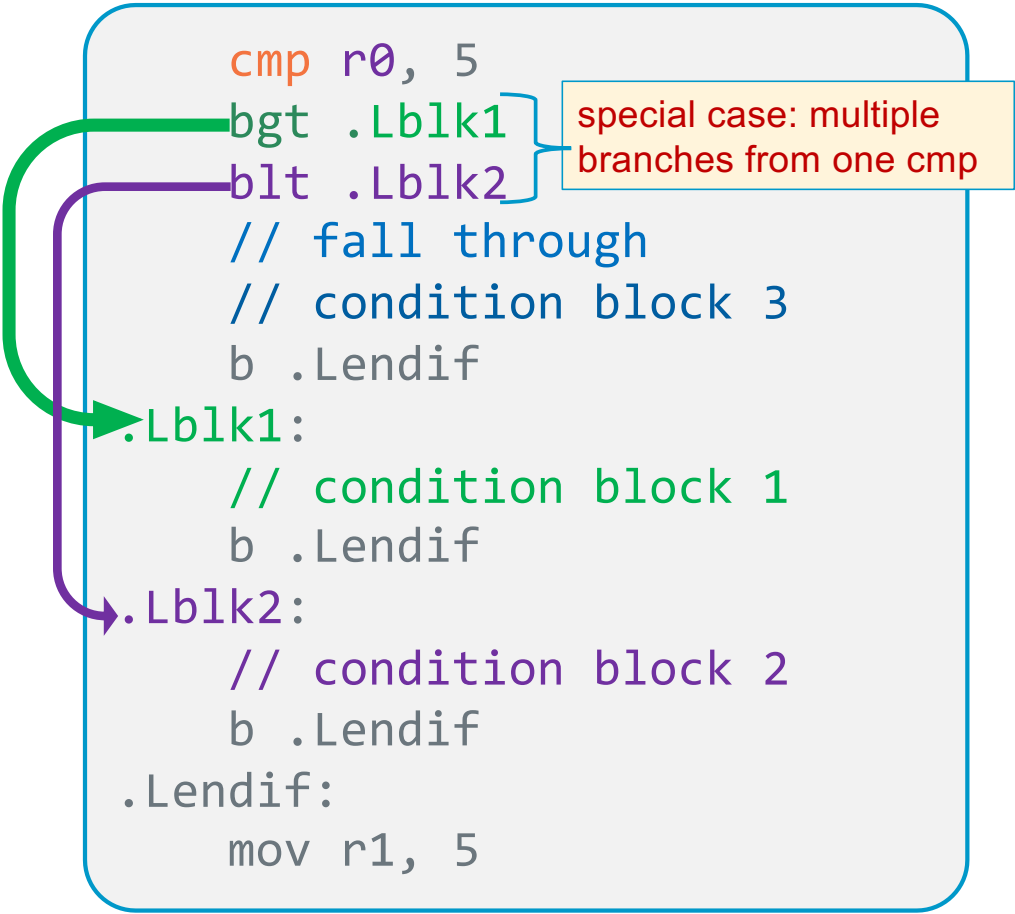


Program Flow – multiple branches, one cmp

```
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