

Version 1.02

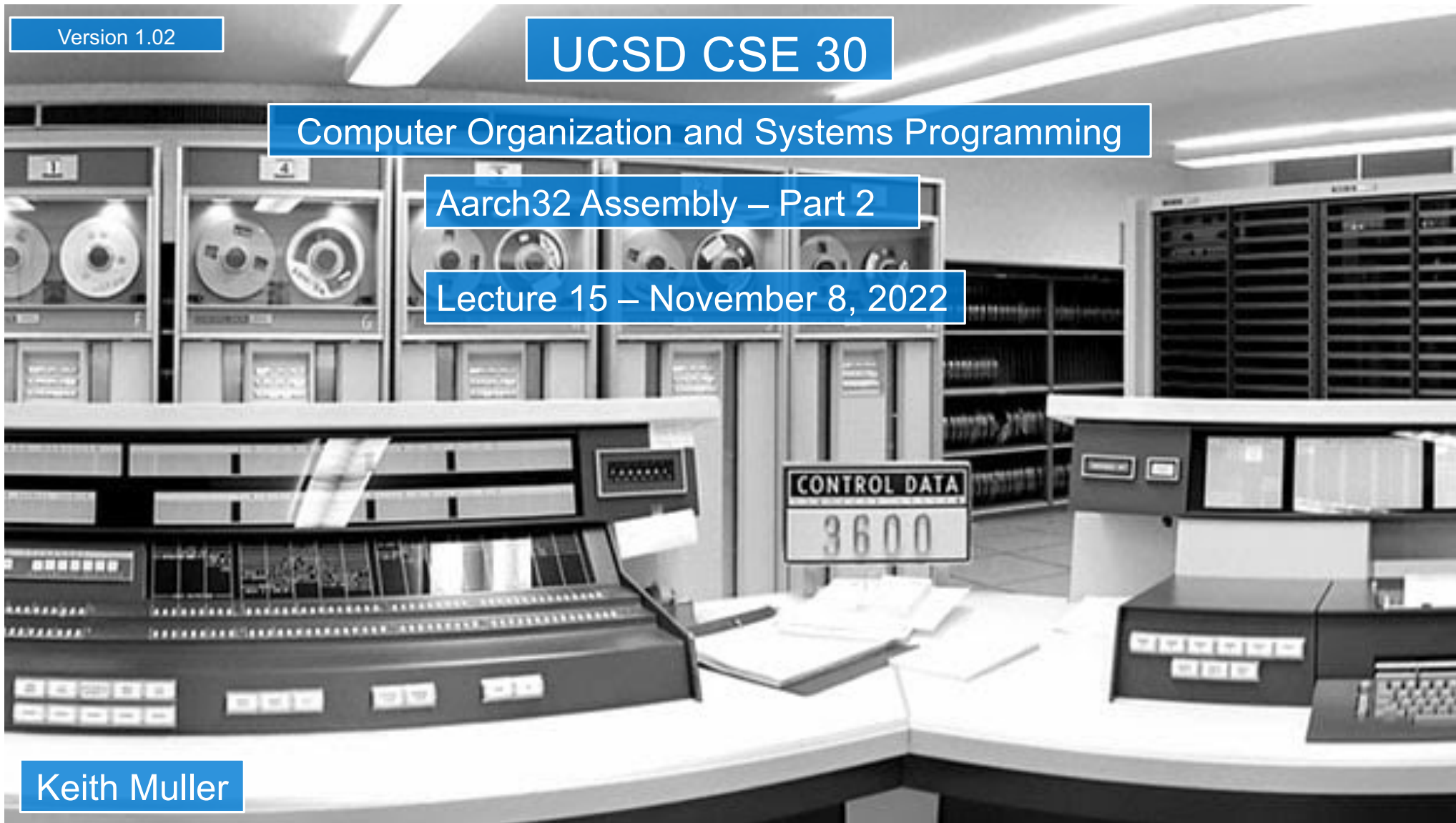
# UCSD CSE 30

## Computer Organization and Systems Programming

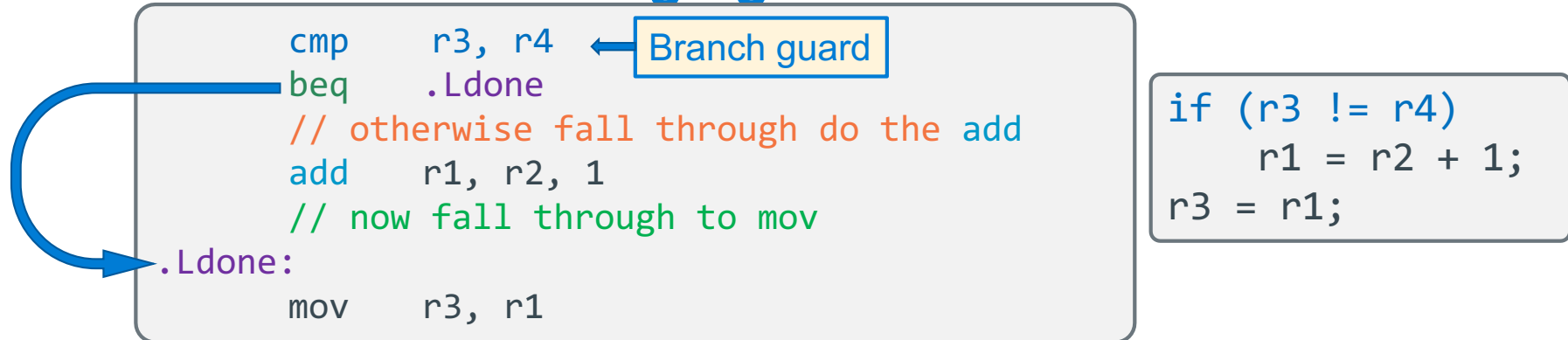
### Aarch32 Assembly – Part 2

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



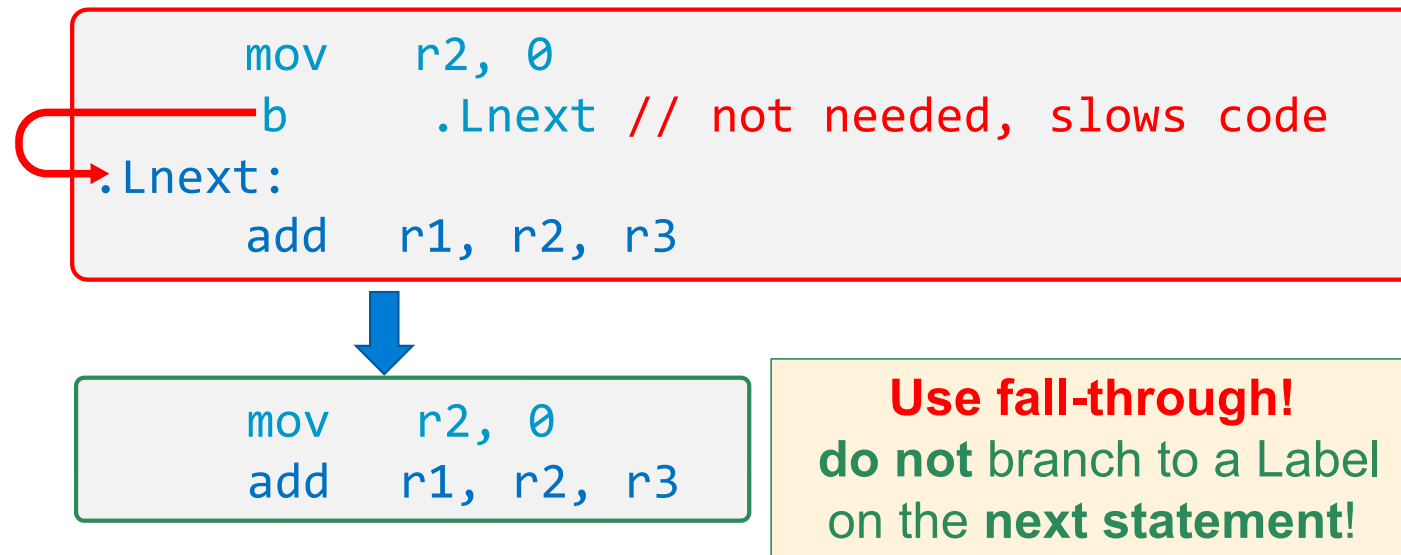
Branch instruction

**b**suffix .**L**label

- 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!

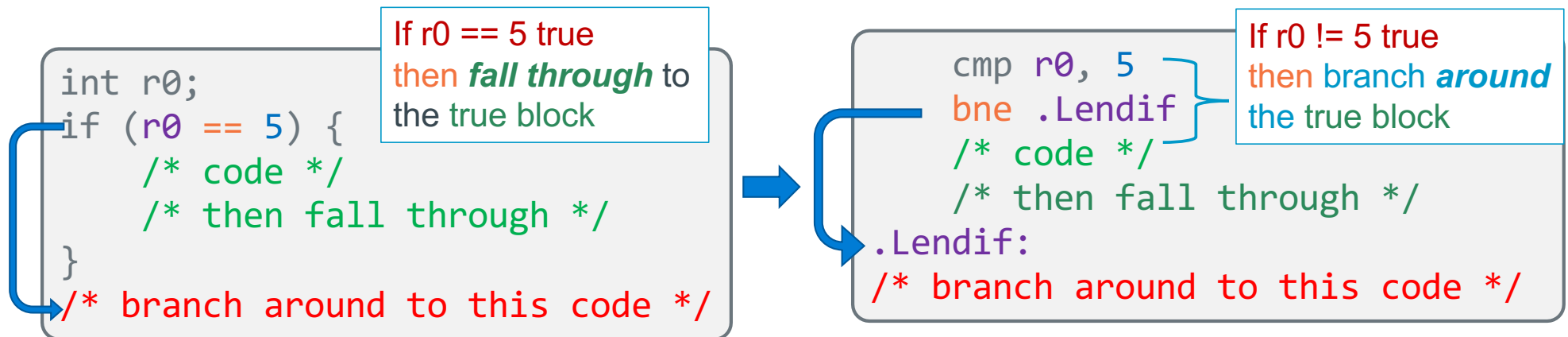
```
mov    r1, 1
mov    r2, 2
mov    r3, 3
mov    r4, 4
mov    r7, 7
```

## 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 <b>beq</b> .Lendif //code .Lendif:</pre>	<pre>cmp r0, 5 <b>bne</b> .Lendif // code .Lendif:</pre>



## Branch Guard "*Adjustment*" Table

### Preserving Block Order In Code

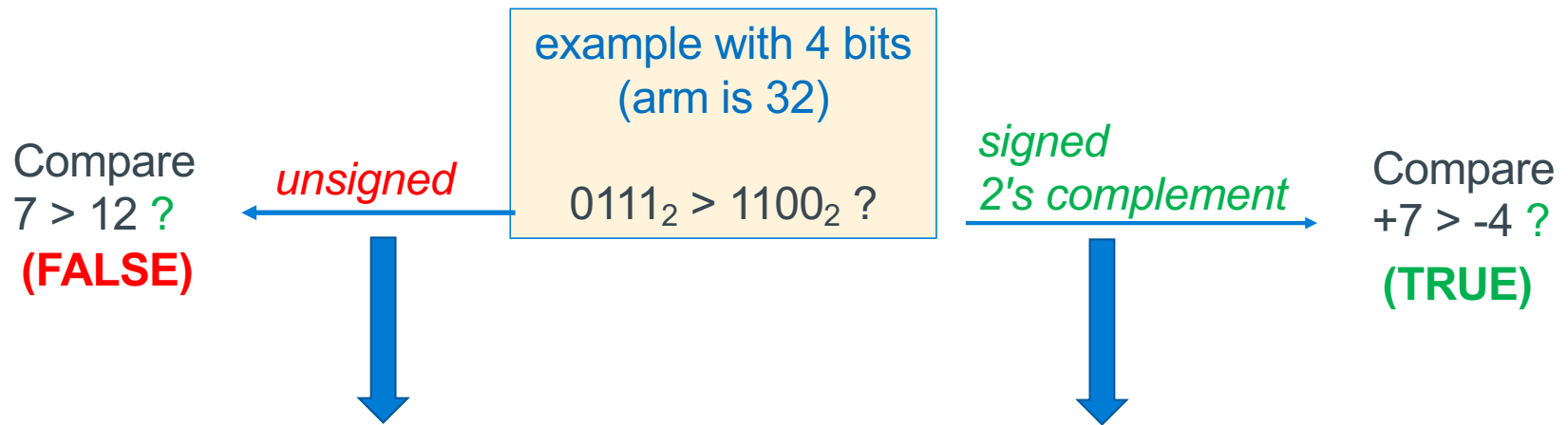
Compare in C	"Inverse" Compare in C	"Inverse" Signed Assembly	"Inverse" Unsigned Assembly
==	!=	bne	bne
!=	==	beq	beq
>	<=	ble	bls
>=	<	blt	blo
<	>=	bge	bhs
<=	>	bgt	bhi

```
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:
>	<b>BHI</b> ( <i>Higher Than</i> )	<b>BGT</b> ( <i>Greater Than</i> )
>=	<b>BHS</b> ( <i>Higher Than or Same</i> ) ( <i>BCS</i> )	<b>BGE</b> ( <i>Greater Than or Equal</i> )
<	<b>BLO</b> ( <i>Lower Than</i> ) ( <i>BCC</i> )	<b>BLT</b> ( <i>Less Than</i> )
<=	<b>BLS</b> ( <i>Lower Than or Same</i> )	<b>BLE</b> ( <i>Less Than or Equal</i> )
==	<b>BEQ</b> ( <i>Equal</i> )	
!=	<b>BNE</b> ( <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;
```

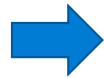


```
cmp    r0, 5  
bne    .Lendif  
add     r1, r2, r3  
add     r2, r2, 1  
.Lendif:  
mov     r3, r2
```

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



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



```
cmp    r0, r1  
bls    .Lendif  
mov     r1, r0  
.Lendif:  
mov     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

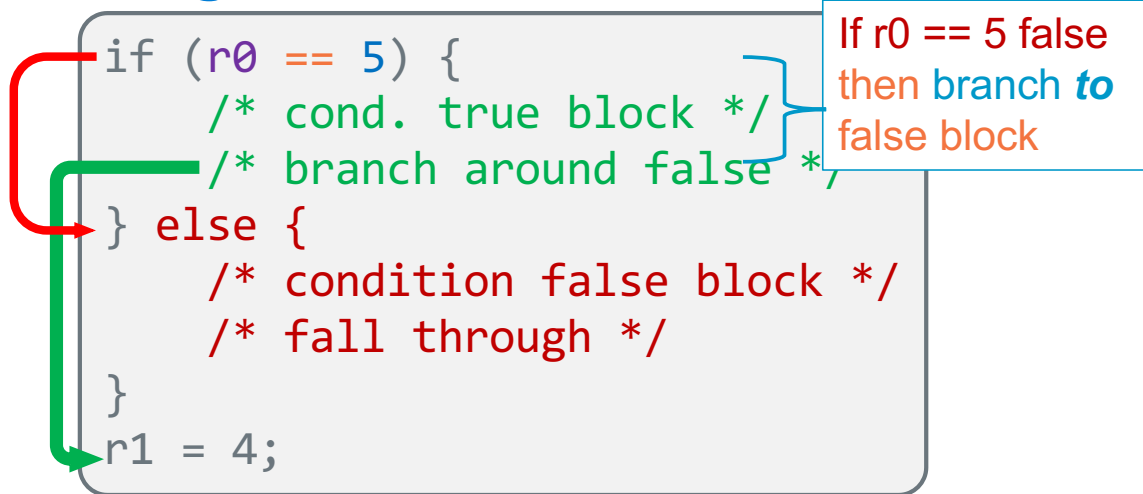
Branch condition  
Test (branch guard)

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

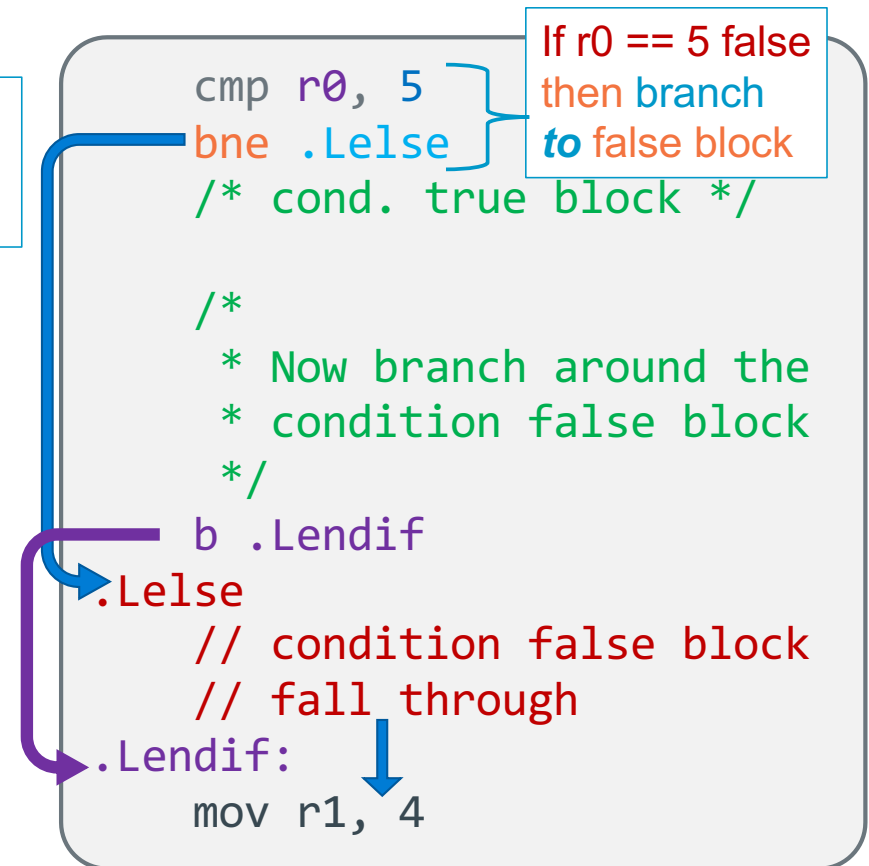
condition  
true block

condition  
false block

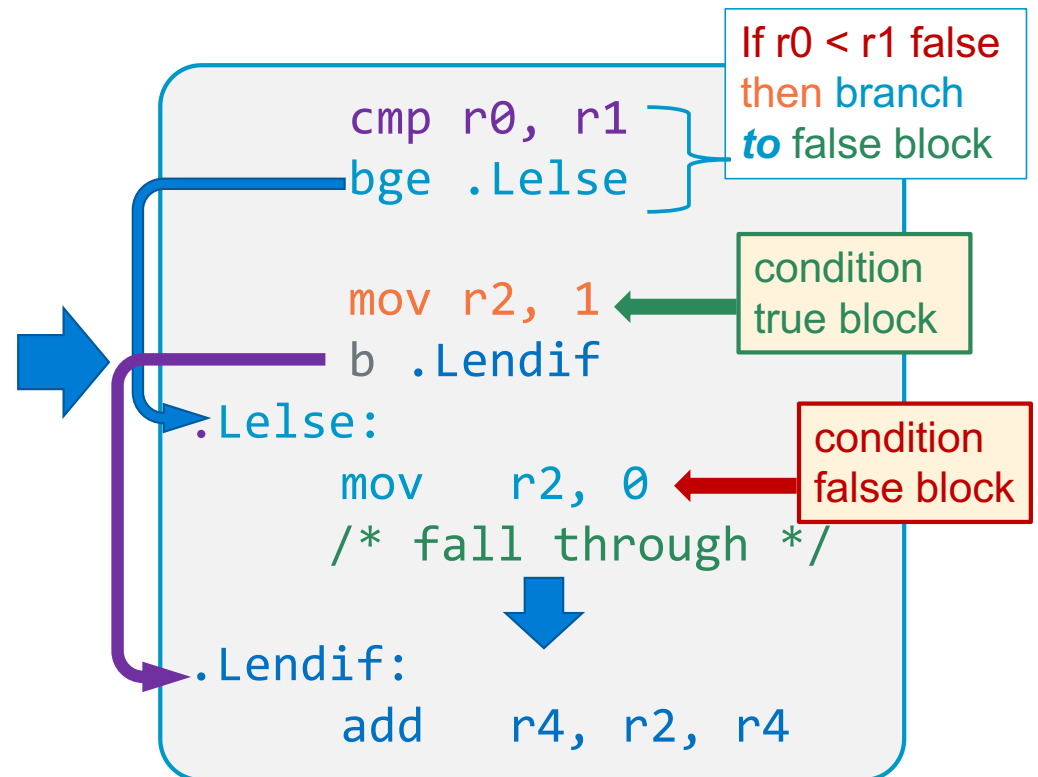
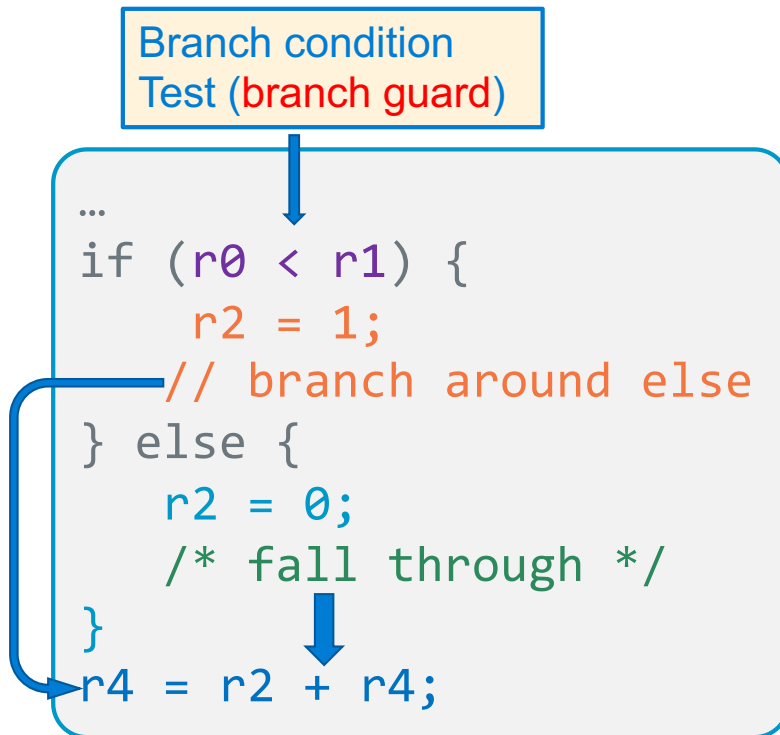
## Program Flow: If with an Else



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;
```

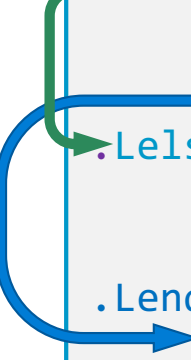


```
cmp r0, r1  
bge .Lelse  
mov r2, 1  
b .Lendif // around else  
.Lelse:  
    mov r2, 0  
    /* fall through */  
.Lendif:  
    add r4, r2, r4
```



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;  
}
```

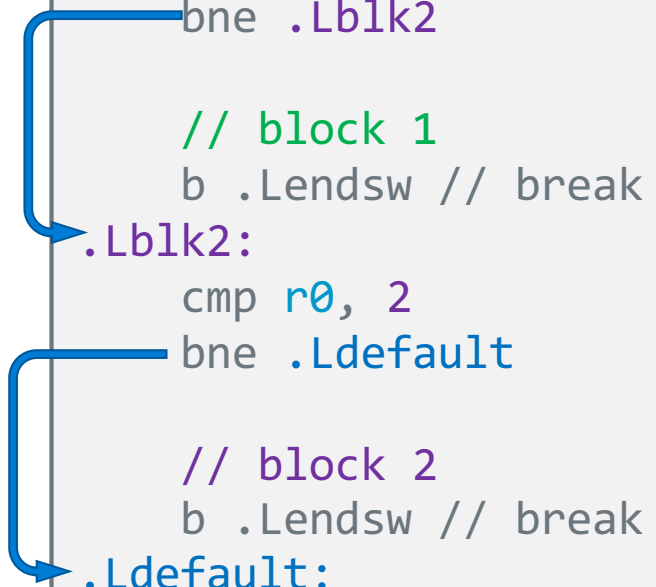
```
    cmp r0, 1  
    beq .Lblk1  
    cmp r0, 2  
    beq .Lblk2  
    // fall through  
    // default 3  
    b .Lendsw // break  
.Lblk1:  
    // block 1  
    b .Lendsw // break  
.Lblk2:  
    // block 2  
    // fall through  
    // NO b .Lendsw  
.Lendsw:
```

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:
```

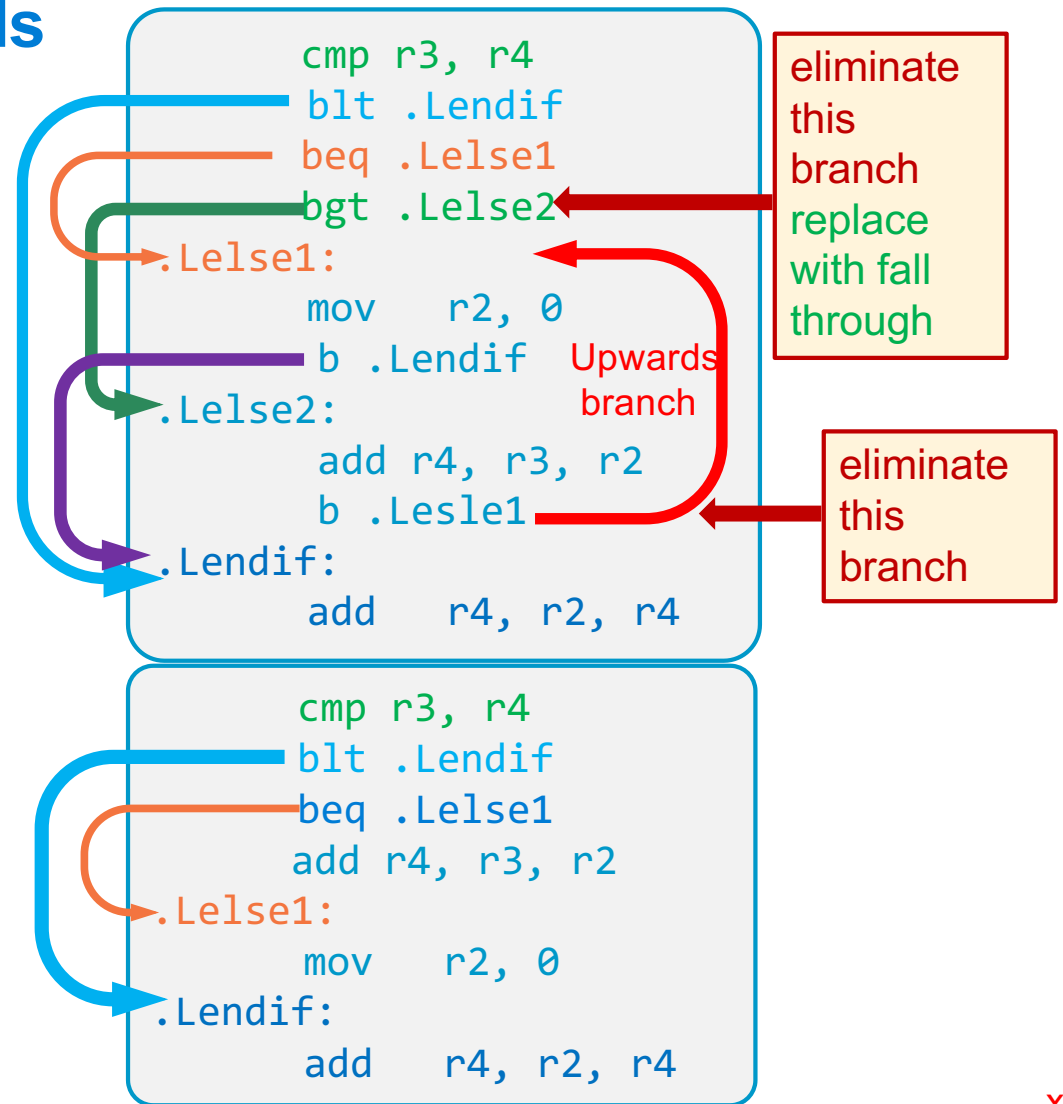




## Bad Style: Branching Upwards (When **Not** a loop)

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
```

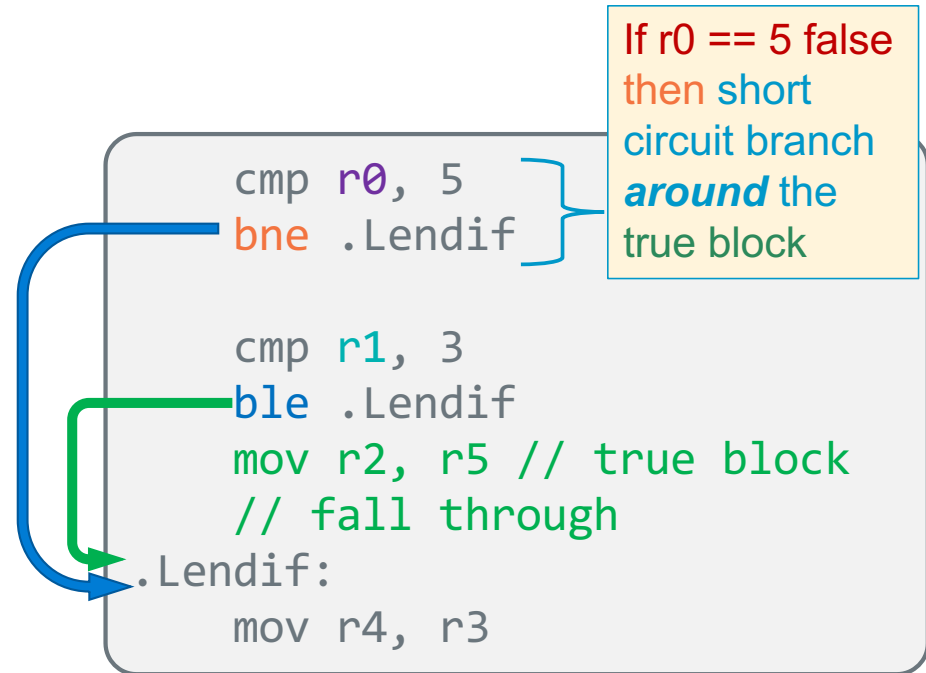


- Each expression argument is evaluated in sequence from left to right including any side effects (modified using parenthesis), before (optionally) evaluating the next expression argument
- If after evaluating an argument, the value of the entire expression can be determined, then the remaining arguments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

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

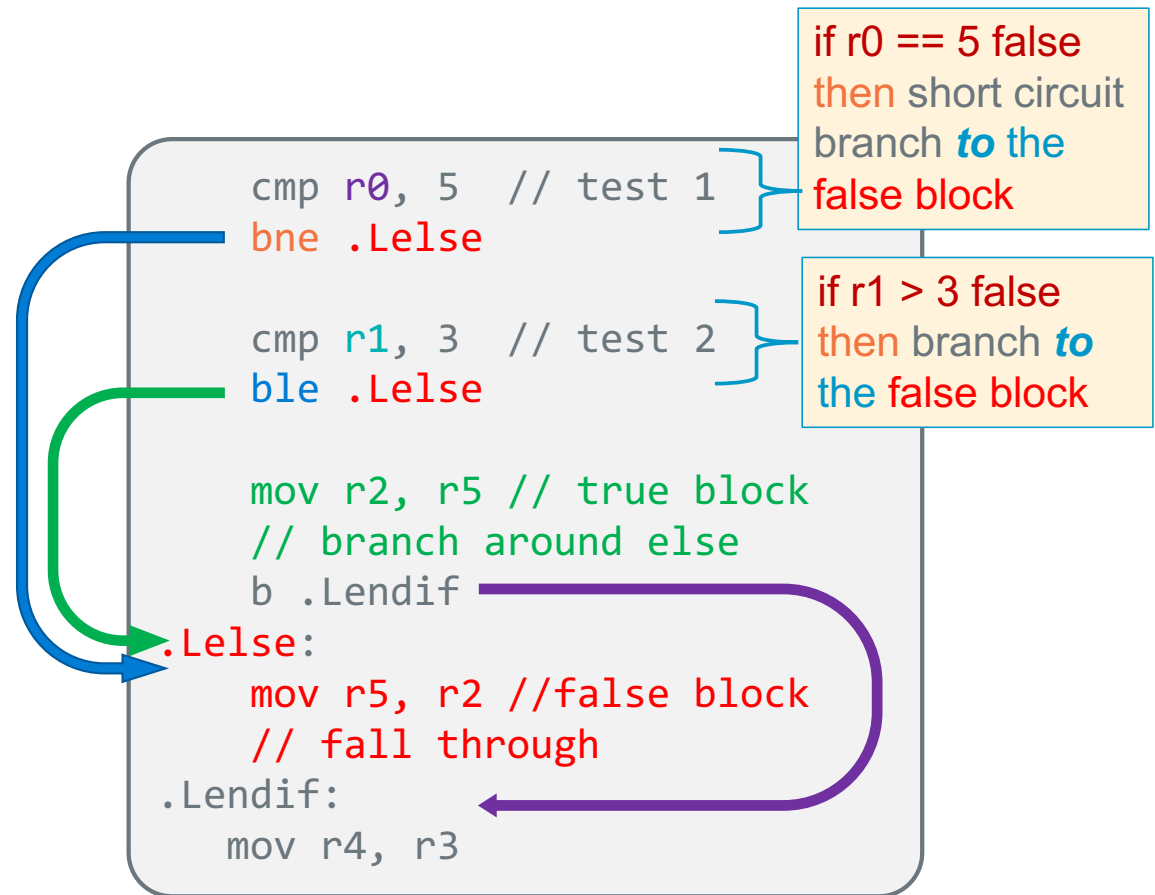
## Program Flow – If statements && compound tests - 1

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



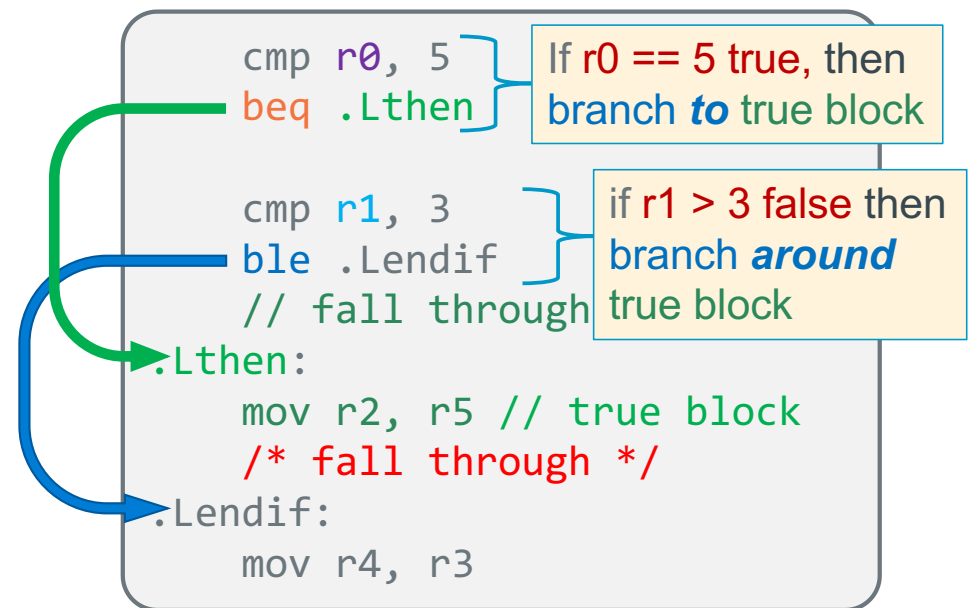
## Program Flow – If statements && compound tests - 2

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



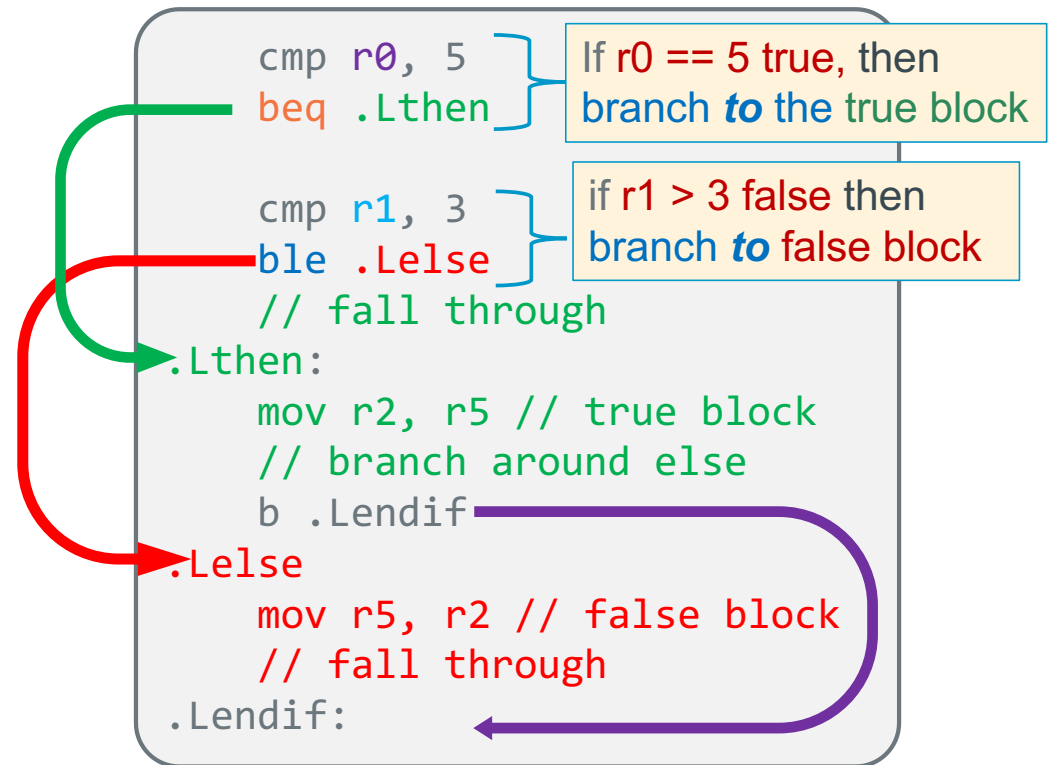
## Program Flow – If statements || compound tests - 1

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



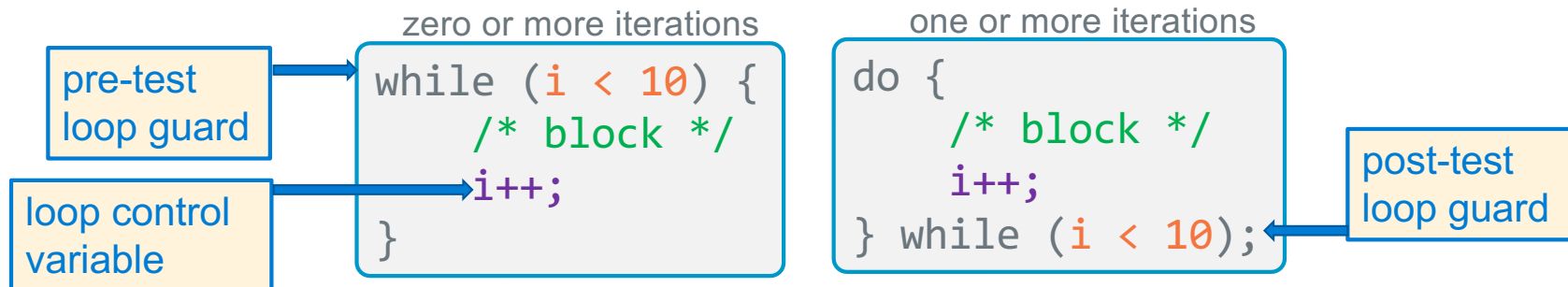
## Program Flow – If statements || compound tests - 2

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

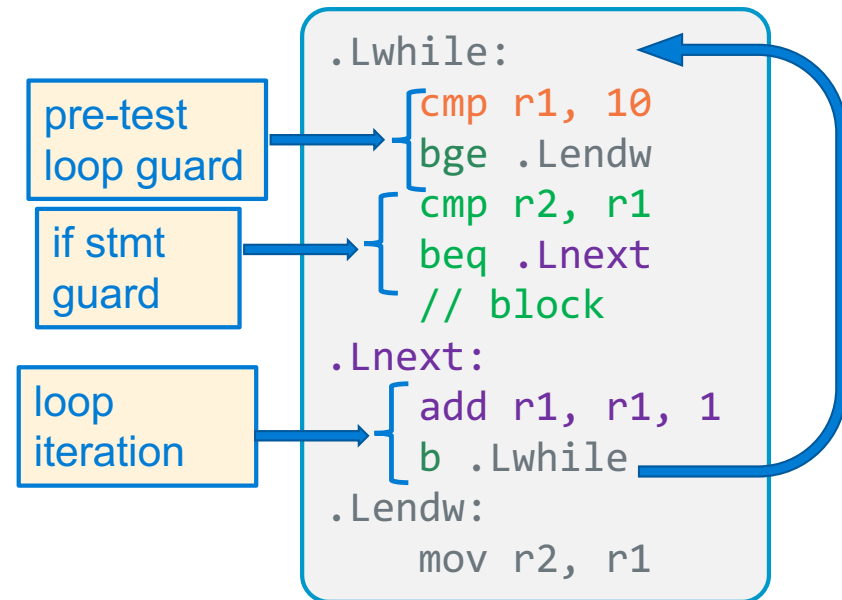
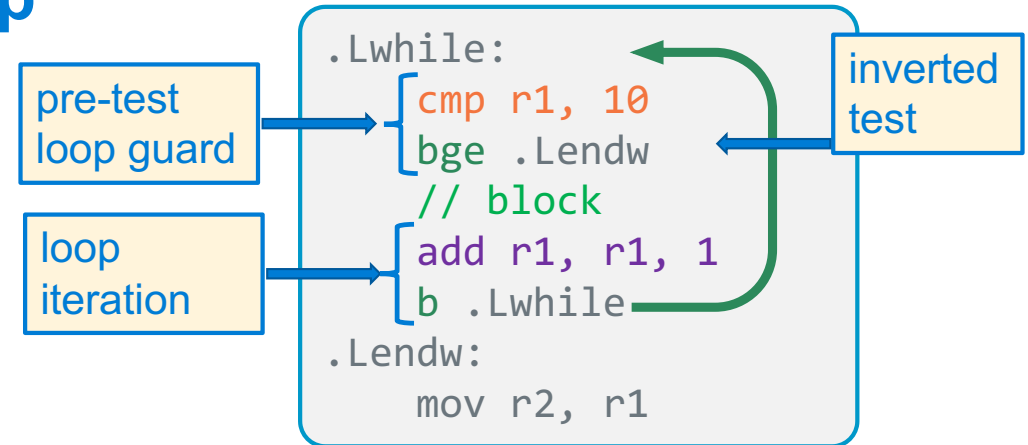
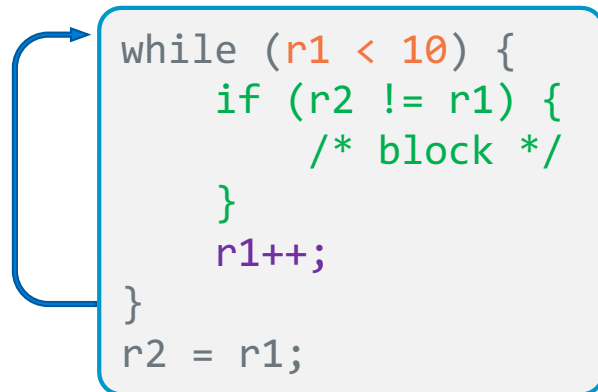
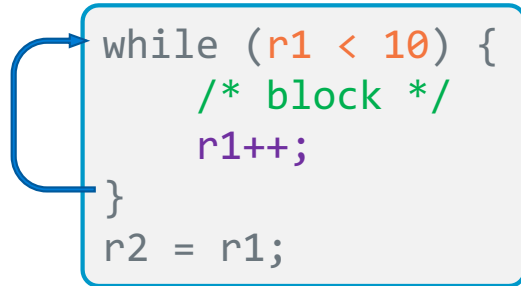


## Program Flow – Pre-test and Post-test Loop Guards

- 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



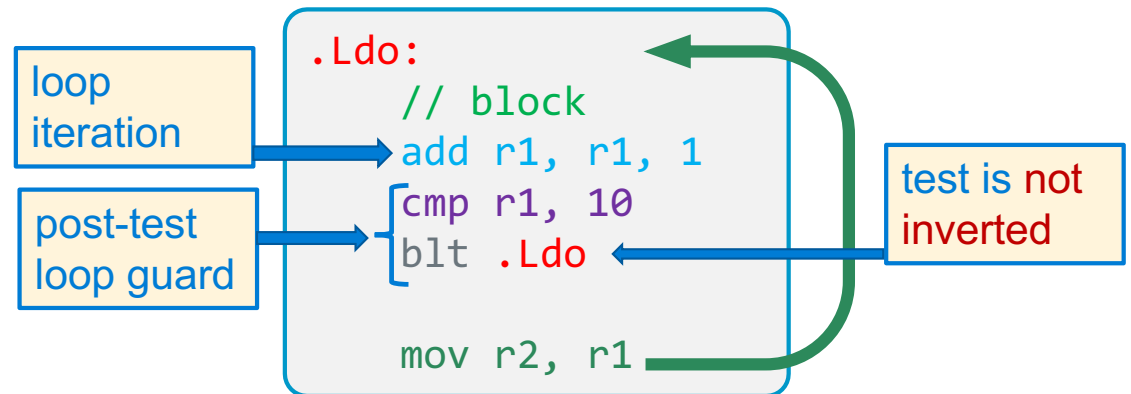
## Pre-Test Guards - While Loop



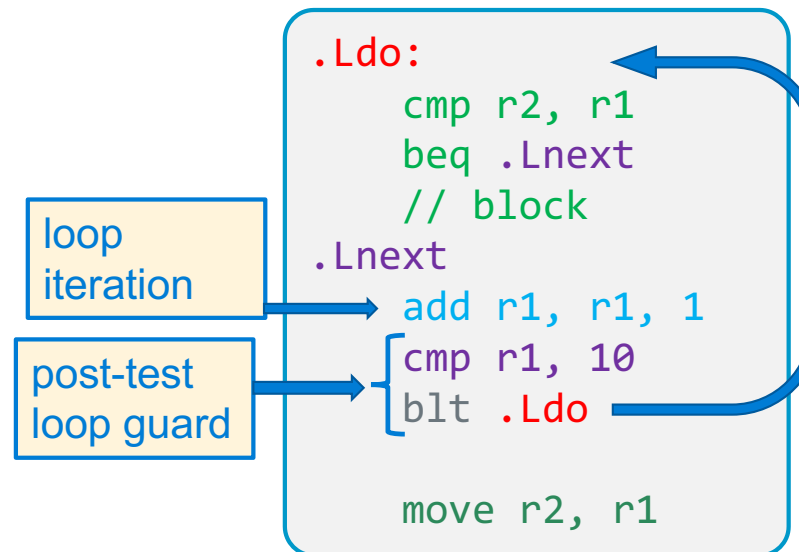


## Post-Test Guards – Do While Loop

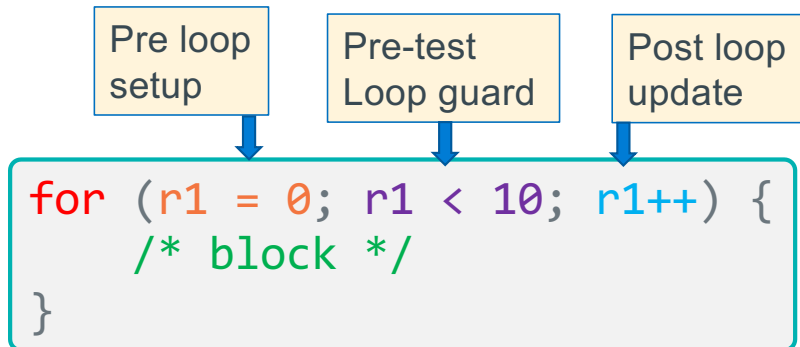
```
do {  
    /* block */  
    r1++;  
} while (r1 < 10);  
  
r2 = r1;
```



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

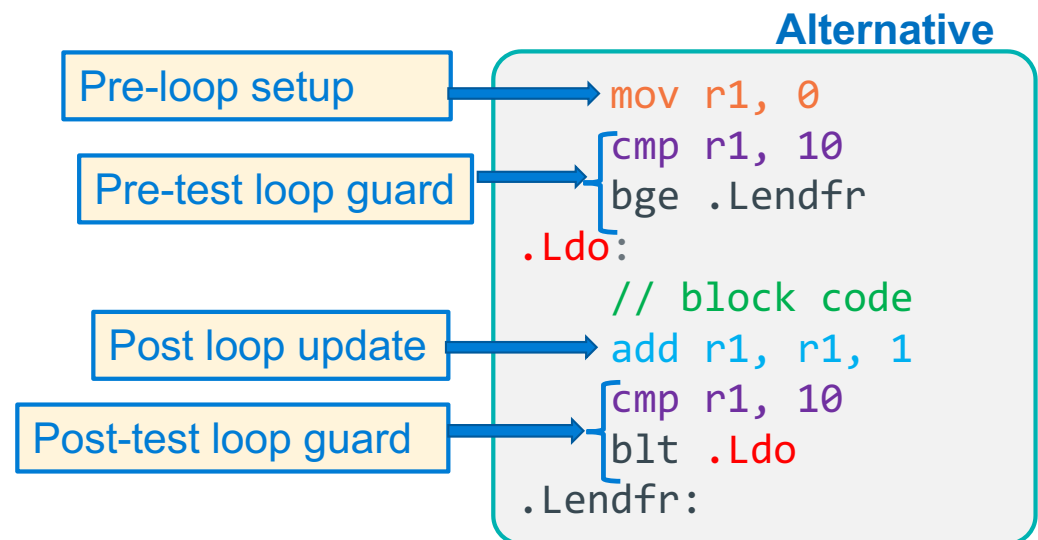
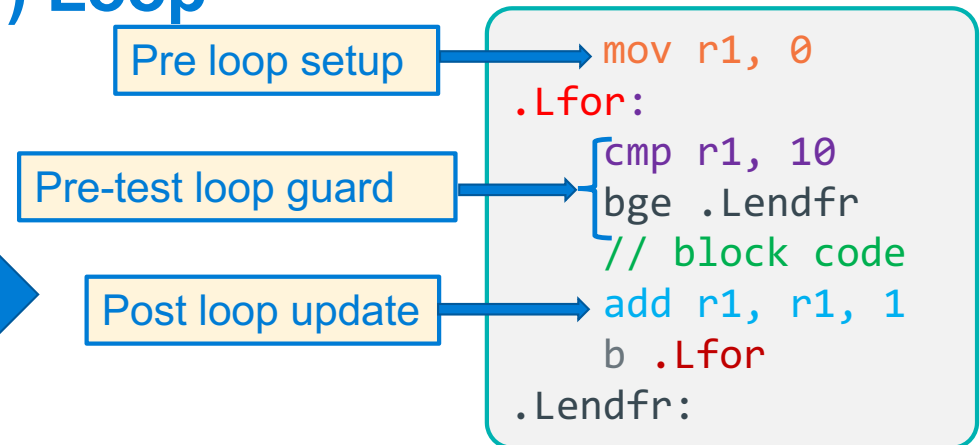


# Program Flow – Counting (For) Loop




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**



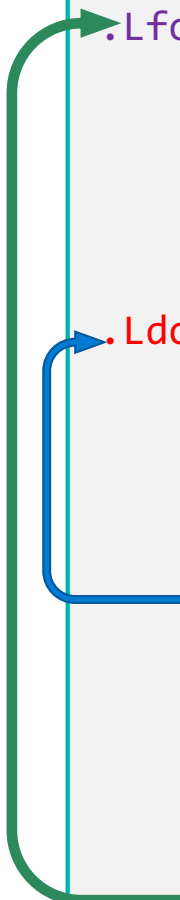
## Nested loops

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



- 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 1.02

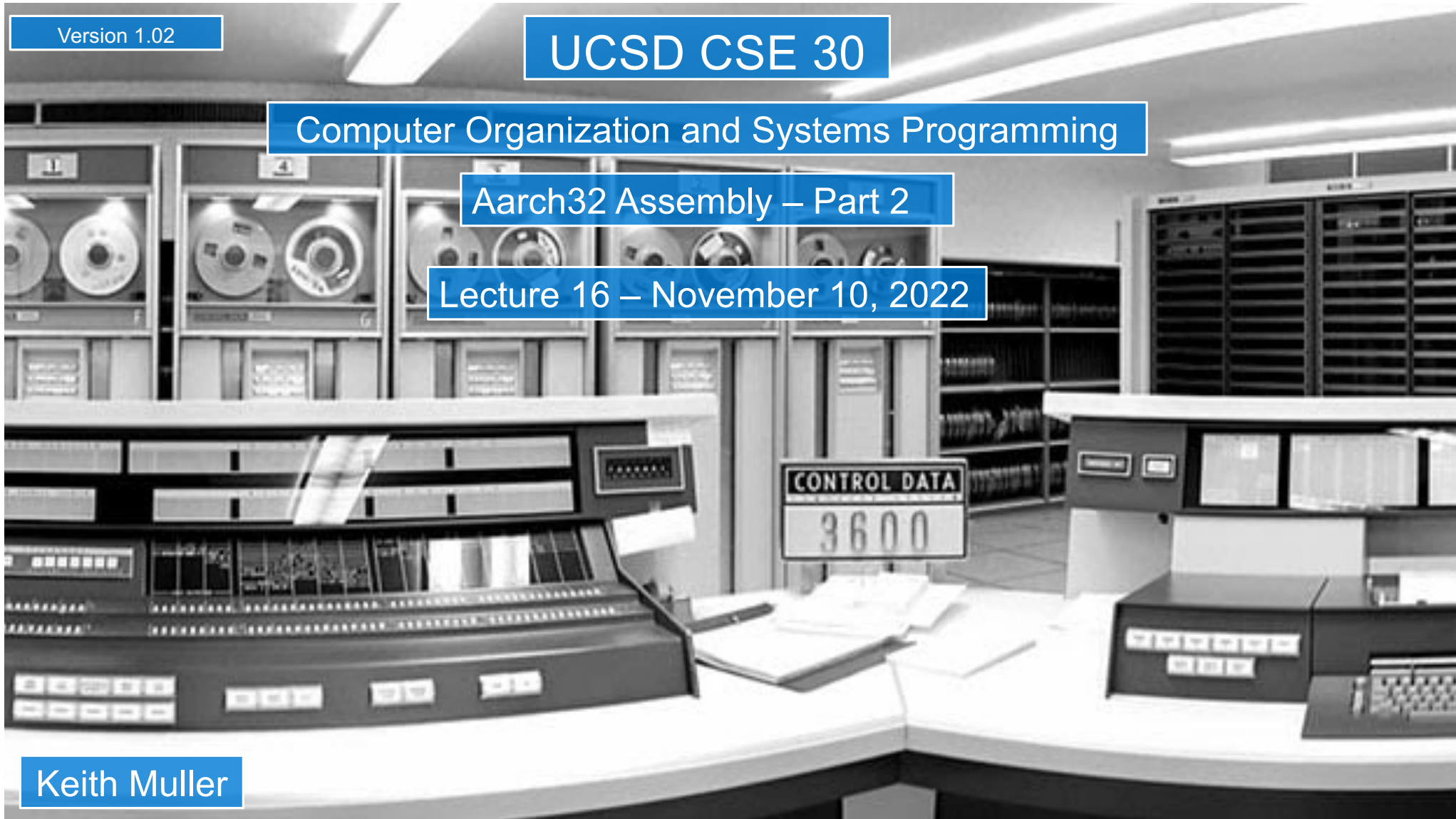
# UCSD CSE 30

## Computer Organization and Systems Programming

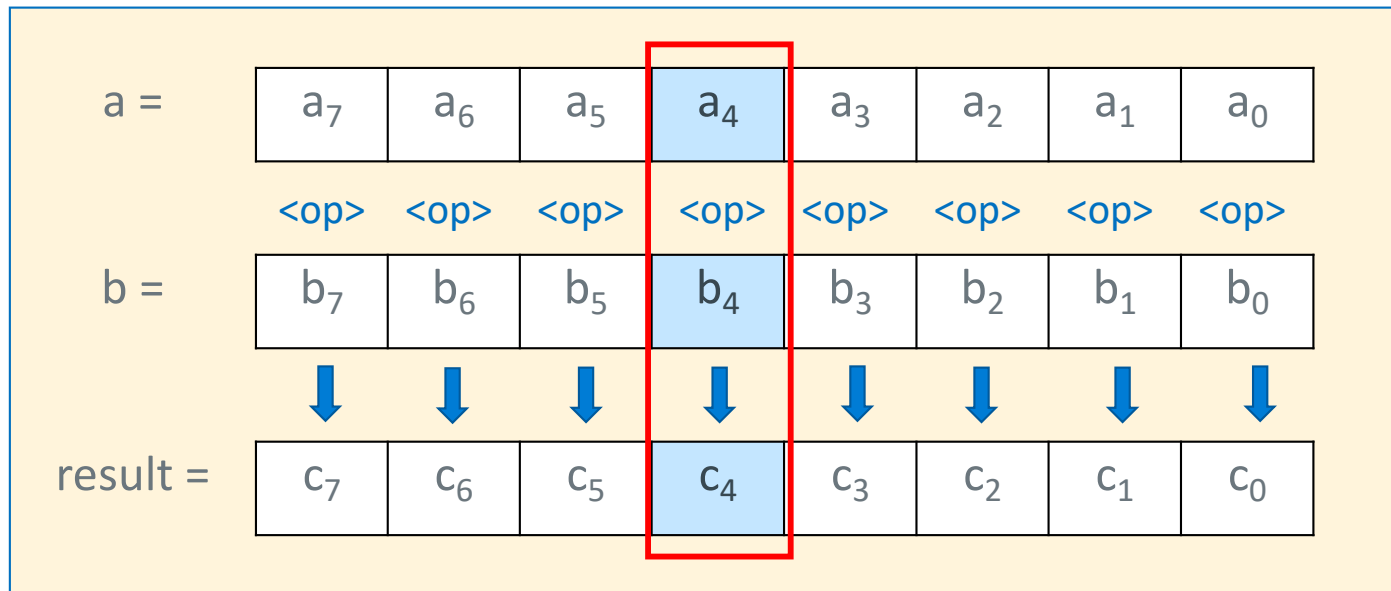
### Aarch32 Assembly – Part 2

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

~	1	1	0	0
	--	--	--	--
	0	0	1	1

	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
```

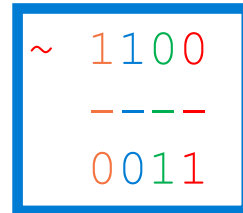
register r0

0x0c



0xffff fff3

Bitwise NOT



- A **bitwise NOT** operation

0x 0c

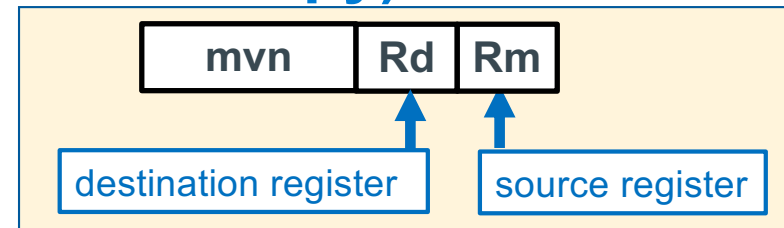
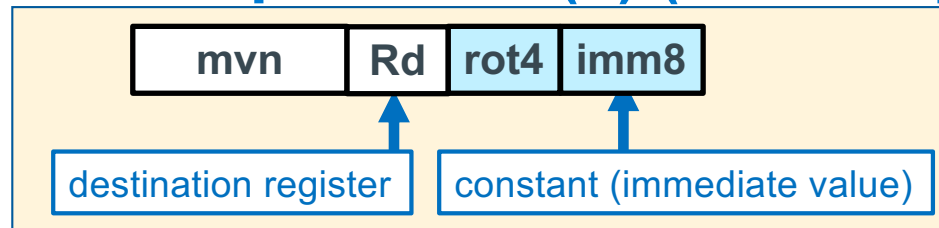
↓ imm8 expansion

0x0000000c

↓ bitwise not

0xfffffffff3

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

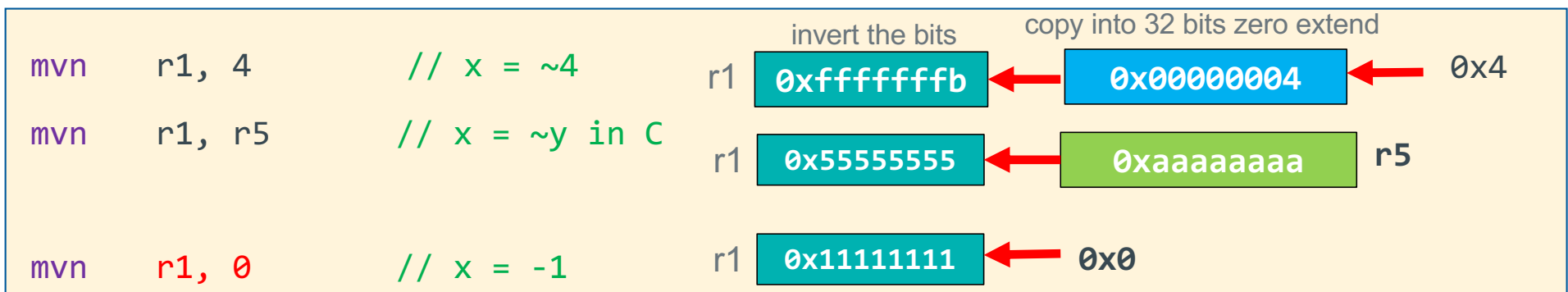


```
mvn Rd, constant // Rd = constant
mvn Rd, Rm       // Rd = Rm
```



**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 &amp;&amp; b</code>	<code>a &amp; 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)`

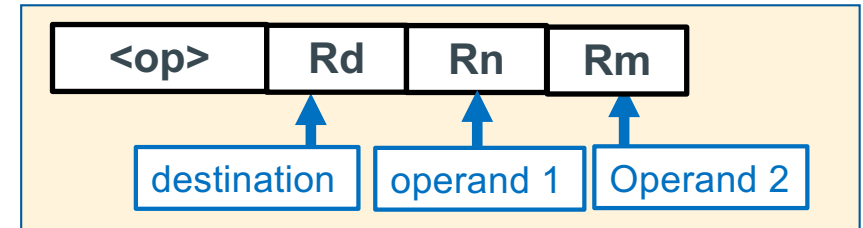
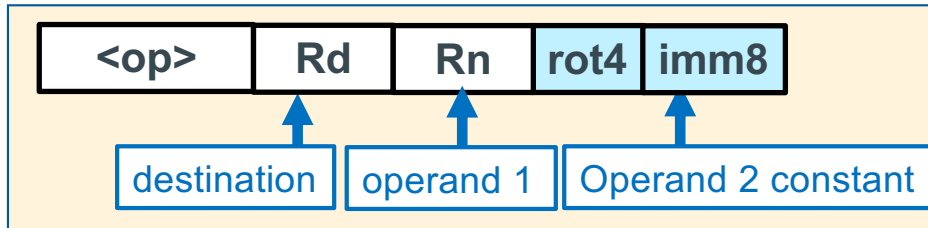
`0x10 && 0x01 = 0x01 (Boolean)`

**! 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>&lt;op&gt;</code> description	C Syntax	Arm <code>&lt;op&gt;</code> Syntax	Operation
Bitwise <b>AND</b>	<code>~x</code>	<code>and Rd, Rn, Op2</code>	$R_d \leftarrow R_n \& Op2$
<b>Bit Clear</b> each bit in Op2 that is a 1, the same bit in $R_d$ , is cleared	<code>&lt;none&gt;</code>	<code>bic Rd, Rn, Op2</code>	$R_d \leftarrow R_n \& \sim Op2$
Bitwise <b>OR</b>	<code>a &amp; b</code>	<code>orr Rd, Rn, Op2</code>	$R_d \leftarrow R_n   Op2$
Exclusive <b>OR</b>	<code>a ^ b</code>	<code>eor Rd, Rn, Op2</code>	$R_d \leftarrow R_n \wedge Op2$

## The act (operation) of *Masking*

a =	a <sub>7</sub>	a <sub>6</sub>	a <sub>5</sub>	a <sub>4</sub>	a <sub>3</sub>	a <sub>2</sub>	a <sub>1</sub>	a <sub>0</sub>
	<op>	<op>	<op>	<op>	<op>	<op>	<op>	<op>
b =	b <sub>7</sub>	b <sub>6</sub>	b <sub>5</sub>	b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>	b <sub>0</sub>
result =	c <sub>7</sub>	c <sub>6</sub>	c <sub>5</sub>	c <sub>4</sub>	c <sub>3</sub>	c <sub>2</sub>	c <sub>1</sub>	c <sub>0</sub>

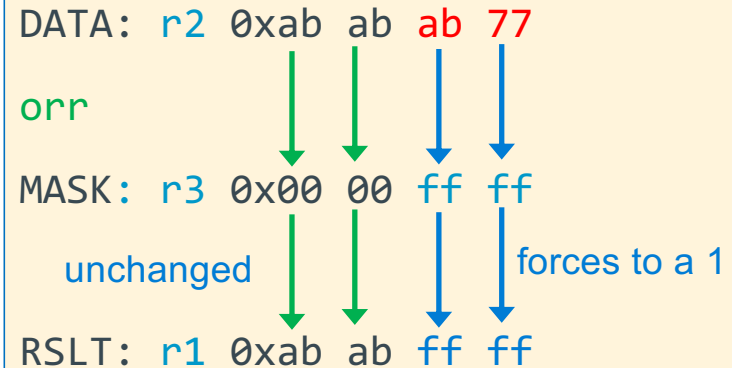
- Bit masks access/modify specific bits in memory
- Masking act of applying a mask to a value with a specific op:
  - **orr**: 0 passes bit unchanged, 1 sets bit to 1
  - **eor**: 0 passes bit unchanged, 1 inverts the bit
  - **bic**: 0 passes bit unchanged, 1 clears it
  - **and**: 0 clears the bit, 1 passes bit unchanged

## Mask on and Mask off

force lower 16 bits to 1 "**mask on**" operation

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

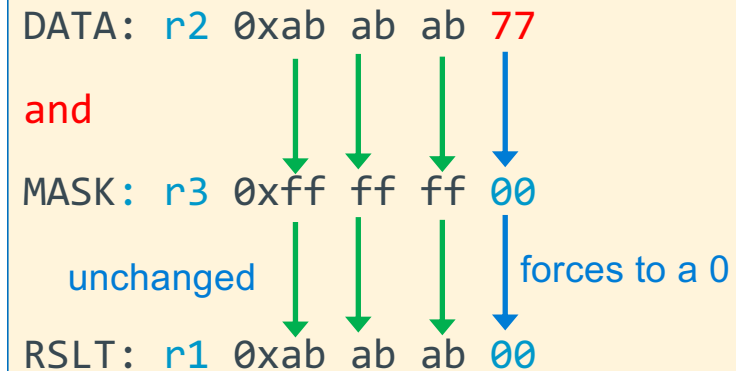
**orr** r1, r2, r3



force lower 8 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



## Mask off versus Bit Clear

force lower 8 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

DATA: r2 0xab ab ab 77

**and**

MASK: r3 0xff ff ff df

unchanged

forces to a 0

RSLT: r1 0xab ab ab 57

df:	1101	1111
77:	0111	0111
57:	0101	0111

clear bit 5 to a 0 without changing the other bits

r1 = r2 & ~r3

**bic** r1, r2, r3

r3:	0010	0000
~r3:	1101	1111

DATA: r2 0xab ab ab 77

**bic**

MASK: r3 0x00 00 00 20

unchanged

clears bit 5

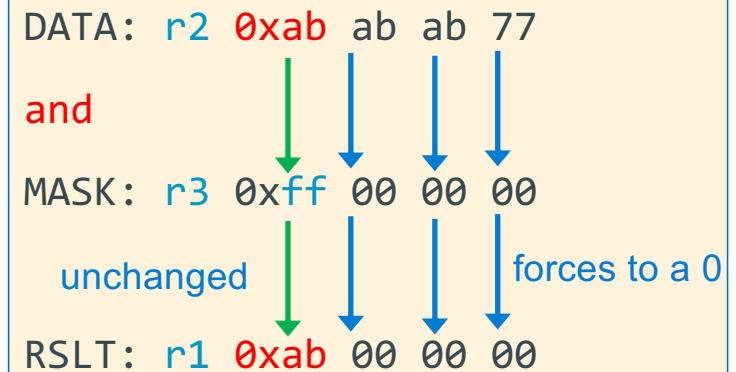
RSLT: r1 0xab ab ab 57

## 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





## 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, r3

cmp r1, 0

bne .Lis\_set

// code

.Lis\_set

DATA: r2 0xab ab ab 77

**and**

MASK: r3 0x00 00 00 02 is bit 1 set?

forces to a 0

unchanged

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

77: 0111 0111  
01: 0000 00**1**0  
and  
**r1:** 0000 00**1**0

75: 0111 0101  
01: 0000 00**1**0  
and  
**r1:** 0000 00**0**0

## Even/Odd : MOD %<power of 2>

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

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

```
and r1, r2, r3
```

```
cmp r1, 1
```

```
beq .Lodd
```

```
// code
```

```
.Lodd:
```

**remainder (mod):**  $\text{num \% d}$  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
```

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)

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*) the lower 8-bits "**bit toggle**" operation

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

**eor** r1, r2, r3

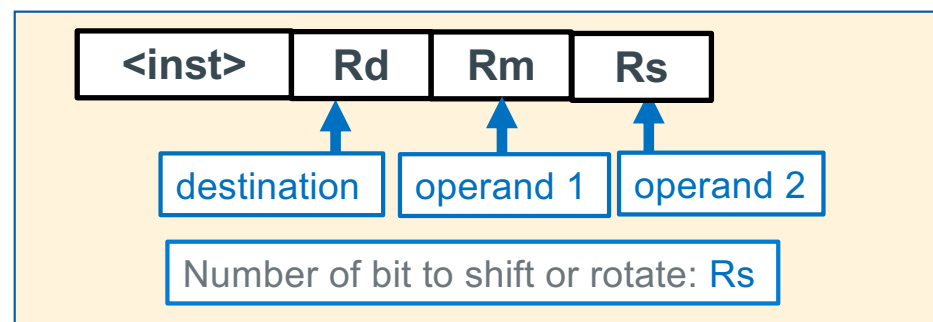
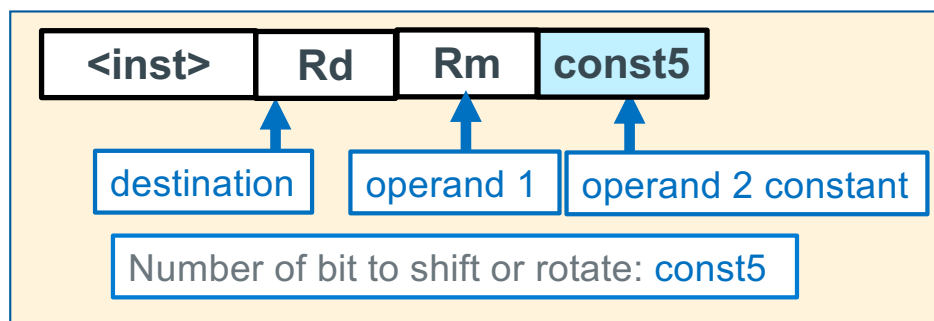
- Observation: When applied twice, it returns the original value (symmetric encoding)
- With a mask of all 1's is a 1's compliment

DATA: r2 0xab ab ab 77  
**eor**  
 MASK: r3 0x00 00 00 ff  
 unchanged ↓ ↓ ↓ ↓ inverts (flips)  
 RSLT: r1 0xab ab ab 88

77: 0111 0111  
 88: 1000 1000

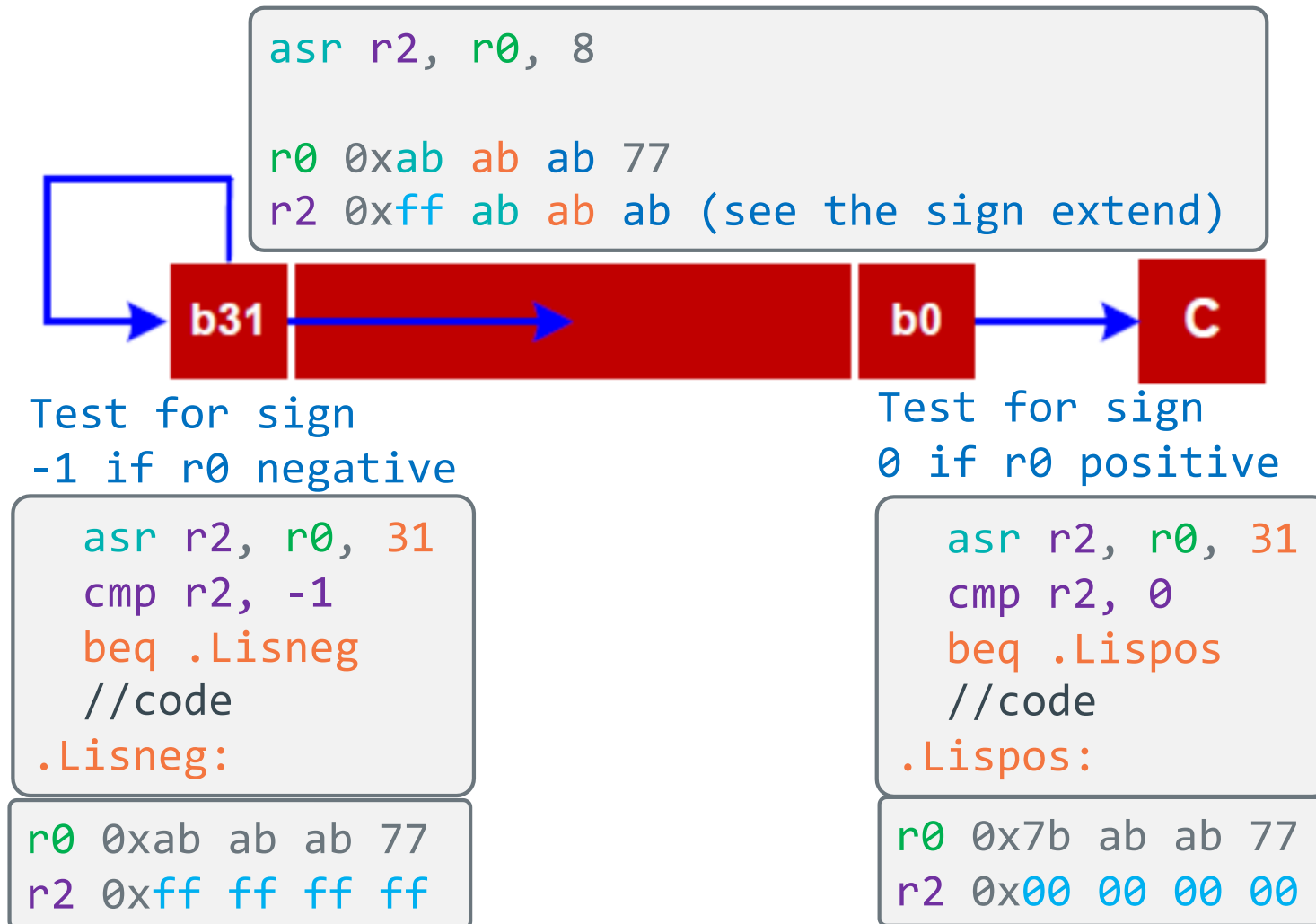
DATA: r1 0xab ab ab 88  
**eor**  
 MASK: r3 0x00 00 00 ff apply a 2<sup>nd</sup> time  
 ↓ ↓ ↓ ↓ inverts (flips)  
 RSLT: r1 0xab ab ab 77 original value!

# Shift and Rotate Instructions



Instruction	Syntax	Operation	Notes	Diagram
Logical Shift Left <i>int x;</i> <i>x &lt;&lt; 1;</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 &gt;&gt; 1;</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 &gt;&gt; 1;</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 &gt;&gt; 1)   (x &lt;&lt; 31);</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	

## Arithmetic Shift Right (there is no arithmetic shift left)



# 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
```



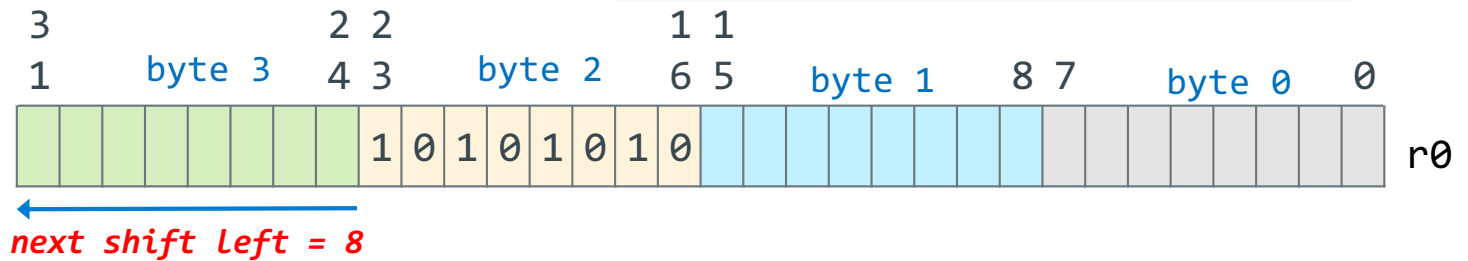
```
ror r2, r0, 8
```

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

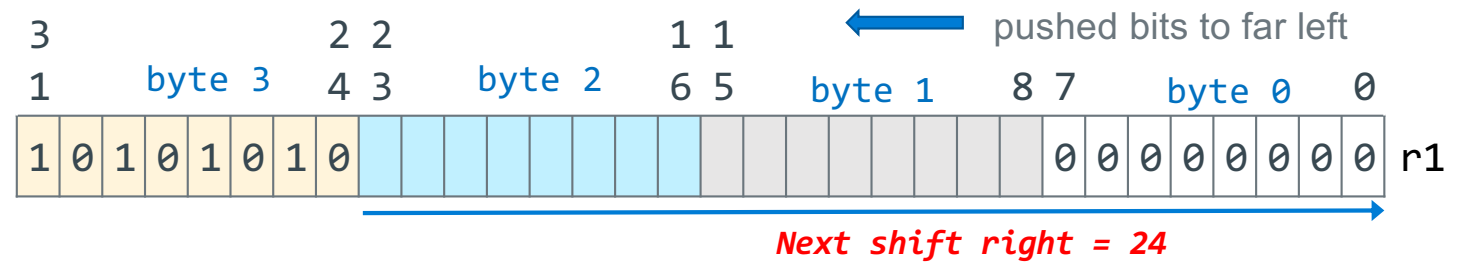
# Extracting/Isolating Unsigned Bitfields

Hint: Useful for PA8

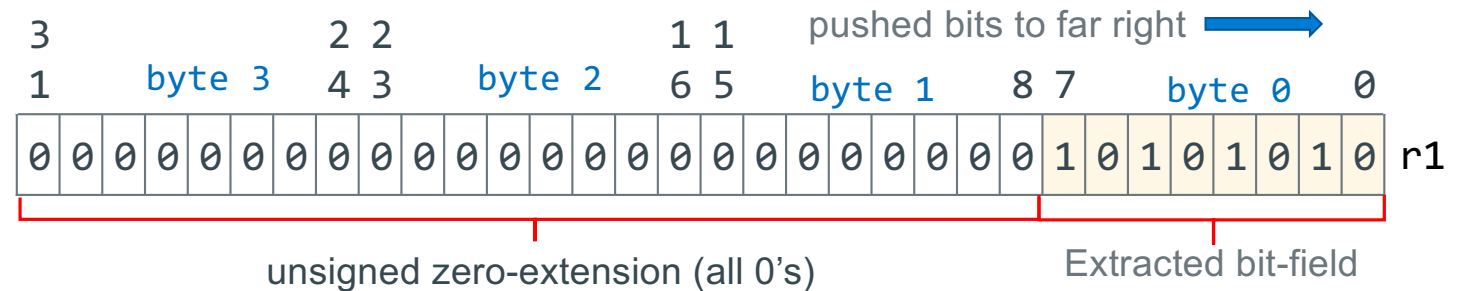
- Move byte 2 in r0 to byte 0 in r1



lsl r1, r0, 8

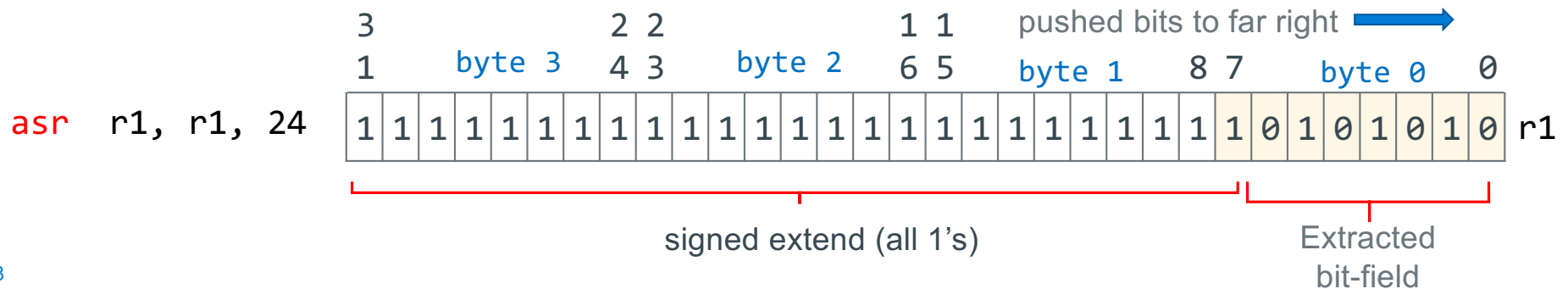
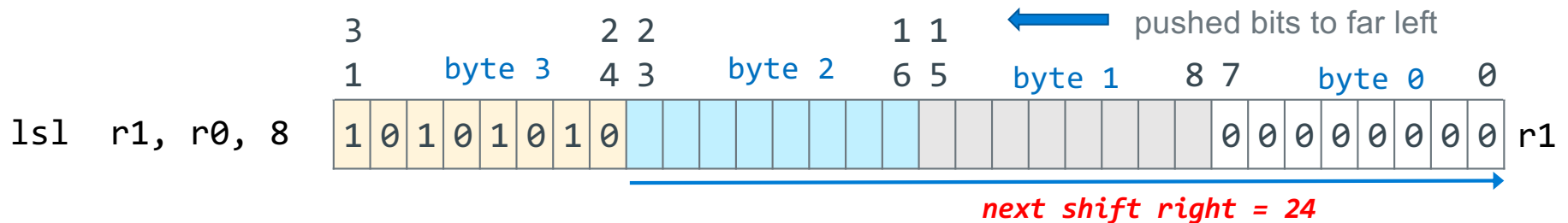
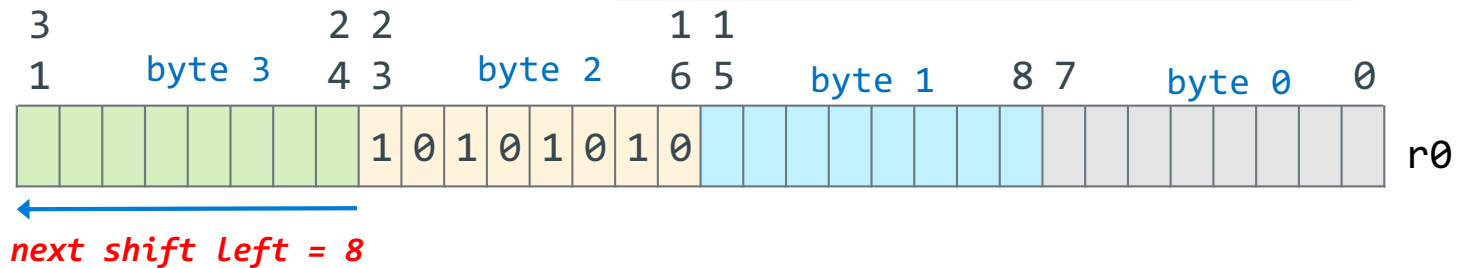


lsr r1, r1, 24



# Extracting Signed Bitfields

- Move byte 2 in r0 to byte 0 in r1





# Inserting Bitfields – Inserting Source Field into Destination Field

Task: Insert source into destination

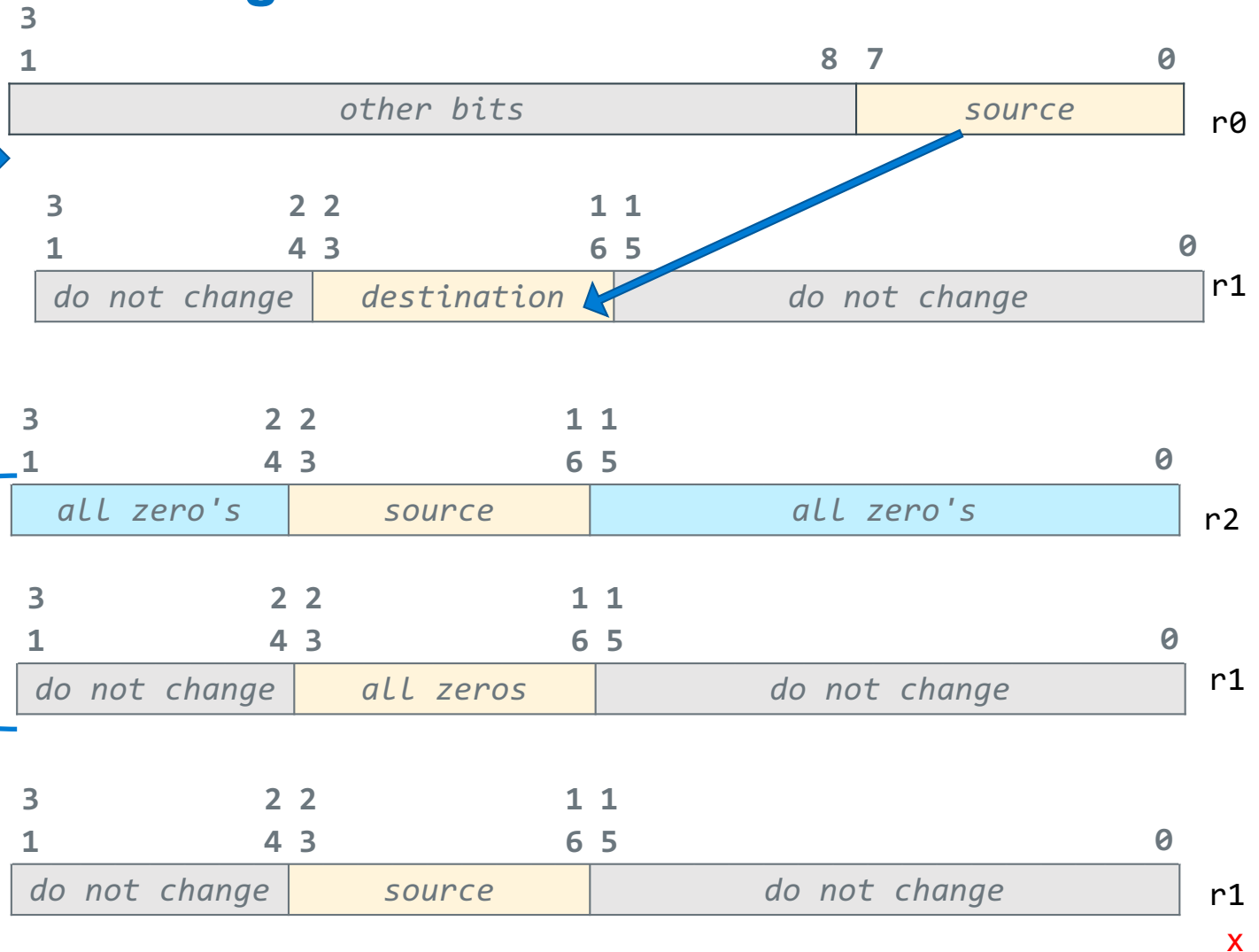
a	b	a   b
0	0	0
0	1	1
1	0	1
1	1	1

## Approach

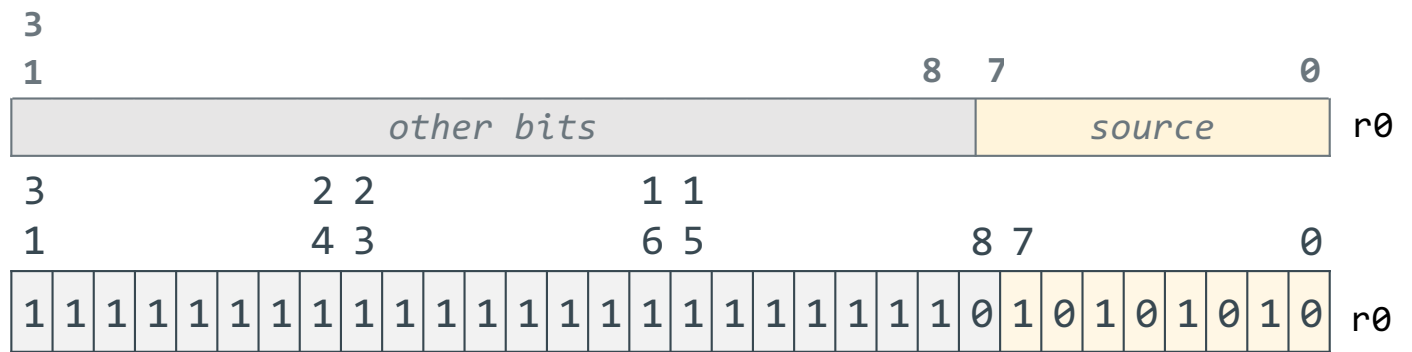
- (1) isolate source field
- (2) clear destination field
- (3) Bitwise **or** together

orr r1, r1, r2

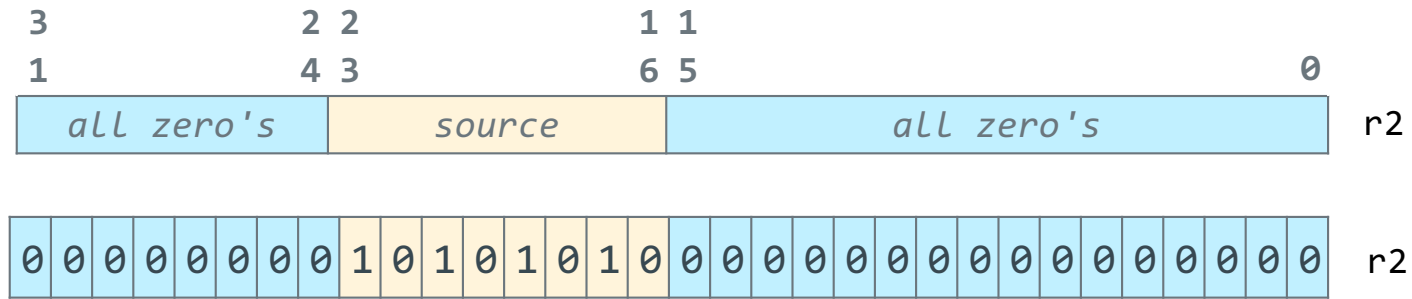
results in



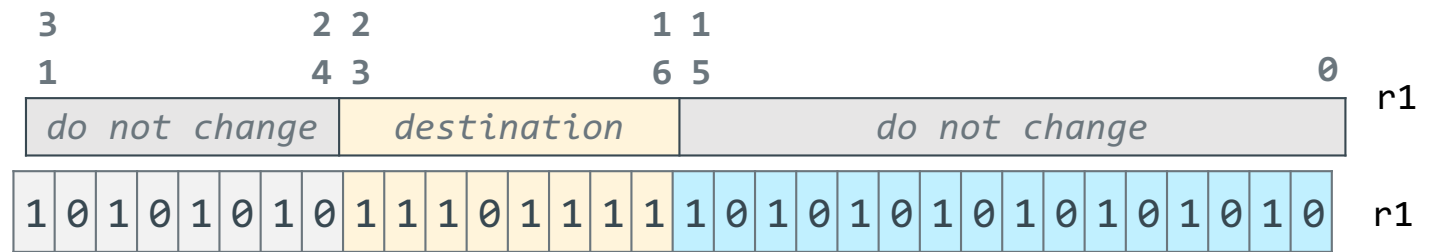
# Inserting Bitfields – Isolating the Source Field



```
isolate source field  
  
lsl    r2, r0, 24  
lsr    r2, r2, 8
```



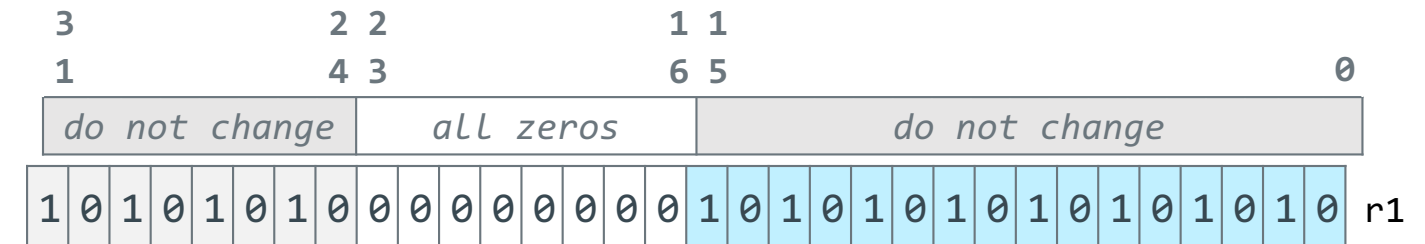
## Inserting Bitfields – Clearing the Destination Field



```
clear the  
destination field  
ror    r1, r1, 24
```



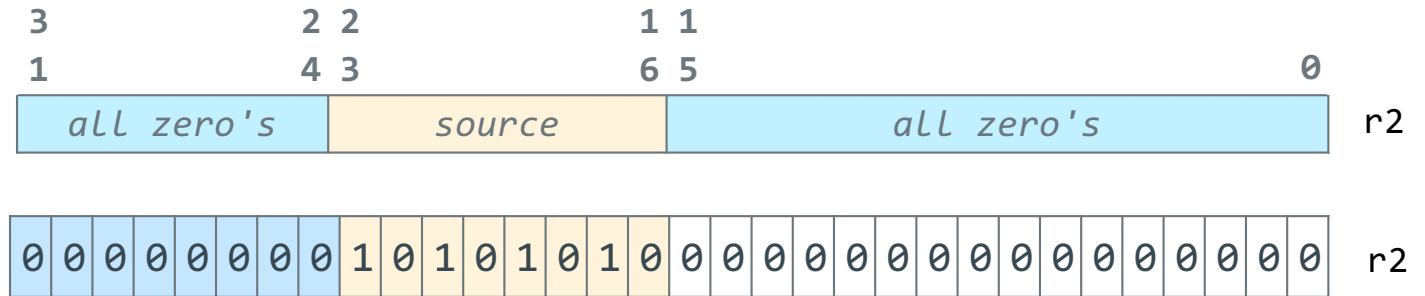
```
ls1    r1, r1, 8
```



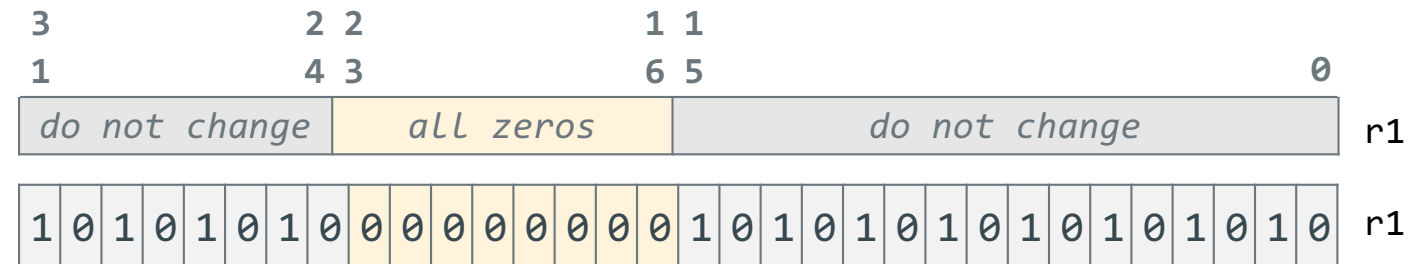
```
ror    r1, r1, 16
```

# Inserting Bitfields – Combining Isolated Source and Cleared Destination

isolated source



field cleared in  
destination



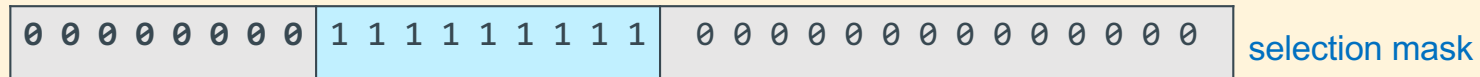
inserted field  
orr r1, r1, r0



# Masking Summary

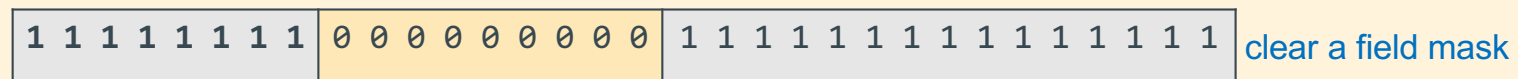
**Select a field:** Use **and** with a **mask** of one's surrounded by zero's to select the bits that have a 1 in the mask, all other bits will be set to zero

selects this field when used with and

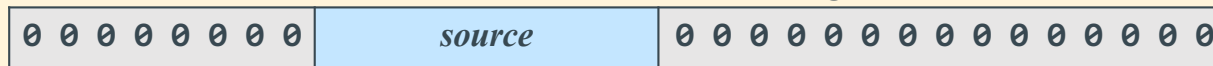


**Clear a field:** Use **and** with a mask of zero's surrounded by one's to select the bits that have a 1 in the mask, all other bits will be set to zero

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



Keep these bits

0 0 0 0 0 0 0 0

Keep these bits

# Creating Segments, Definitions In Assembly Source

- The following assembler directives indicate the **start** of a **memory segment specification**
  - **Remains in effect** until the next segment directive is seen

```
.bss
    // start uninitialized static segment variables definitions
    // does not consume any space in the executable file
.data
    // start initialized static segment variables definitions
.section .rodata
    // start read-only data segment variables definitions
.text
    // start read-only text segment (code)
```

- Define a **literal**, **static variable** or **global** variable in a segment

```
Label: .size_directive expression, ... expression
```

- **Label**: this is the **variables name**
- **Size\_Directive** tells the **assembler** *how much space to **allocate*** for that **variable**
- Each **optional expression** specifies the contents of one memory location of **.size\_directive**
  - **expression** can be in **decimal**, **hex** (0x...), **octal** (0...), **binary** (0b...), **ASCII** (' '), **string** " "

# Assembly Source File Template

```
// File Header
.arch armv6                // armv6 architecture instructions
.arm                      // arm 32-bit instruction set
.fpu vfp                  // floating point co-processor
.syntax unified           // modern syntax

// BSS Segment (only when you have initialized globals)
.bss

// Data Segment (only when you have uninitialized globals)
.data

// Read-Only Data (only when you have literals)
.section .rodata

// Text Segment - your code
.text

// Function Header
.type main, %function      // define main to be a function
.global main              // export function name
main:
// function prologue        // stack frame setup
    // your code for this function here
// function epilogue        //stack frame teardown

// function footer
.size main, (. - main)

// File Footer
.section .note.GNU-stack,"",%progbits // stack/data non-exec
.end
```

- assembly programs end in **.S**
  - That is a **capital .S**
  - **example:** test.S
- Always use gcc to assemble
  - **\_start()** and C runtime
- File has a complete program  
**gcc file.S**
- File has a partial program  
**gcc -c file.S**
- Link files together  
**gcc file.o cprog.o**

# Memory Segment Data Alignment

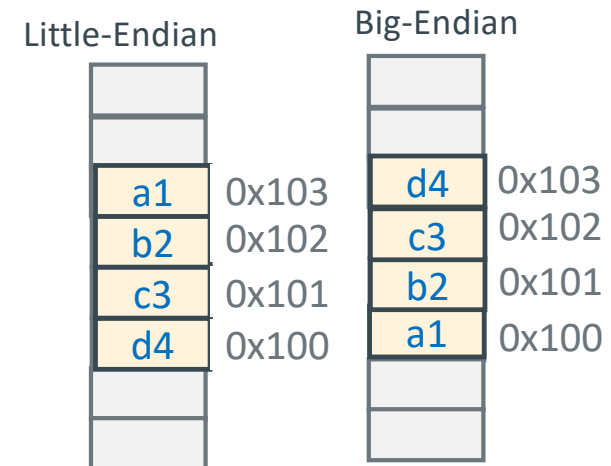
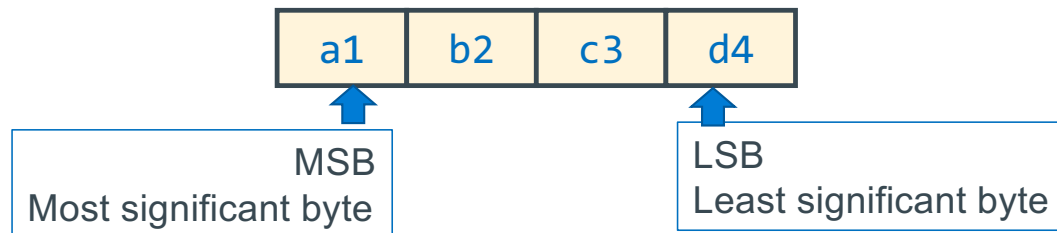
- **Word** is the **number of bytes** necessary to store an **address (32-bits on Pi-cluster)** – **hardware defined**
- The **address** of **any sized** unit of memory is always the **address** of the **first byte**
- Hardware often requires Variables to be *"aligned"* to specific starting addresses based on type
- char (1 byte)
  - can start at any address
- short (2 bytes) can start only at addresses ending
  - b..00 or b..10 (.align 1) // **last bit must be 0**
- int (4 bytes) can start only at address ending in
  - 0b..00 (.align 2) // **last two bits must be 0**

32-bit units (4 bytes)	16-bit units (2 Bytes)	8-bit units (1 Byte)	Addr. (binary)
	Start at b..10		b..10011
Start At b..00	Start at b..00		b..10010
	Start at b..10		b..10001
	Start at b..00		b..10000
Start at b..00	Start at b..10		b..01111
	Start at b..00		b..01110
	Start at b..10		b..01101
	Start at b..00		b..01100
Start at b..00	Start at b..10		b..01011
	Start at b..00		b..01010
	Start at b..10		b..01001
	Start at b..00		b..01000
	Start at b..10		b..00111
Start at b..00	Start at b..00		b..00110
	Start at b..10		b..00101
	Start at b..00		b..00100



# Byte Ordering of Numbers In Memory: Endianness

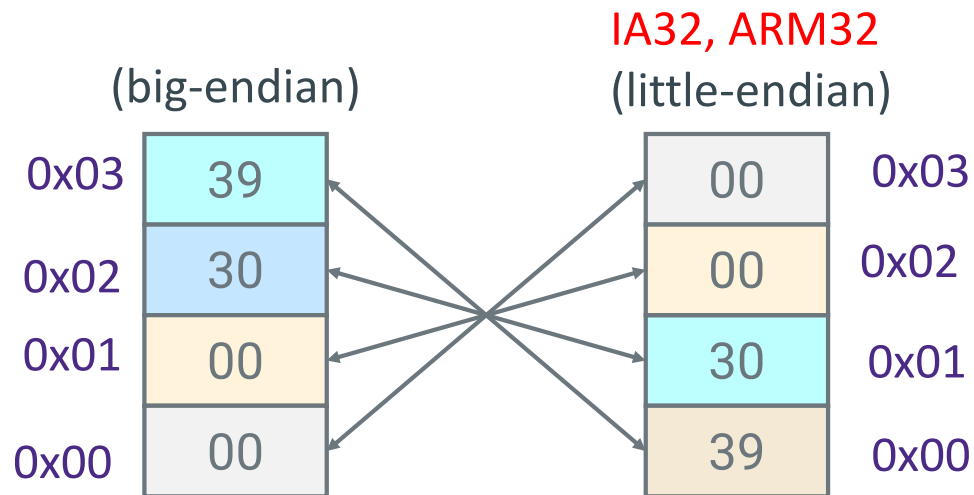
- Two different ways to place multi-byte integers in a **byte addressable** memory
- **Big-endian**: **Most** Significant Byte (“**big end**”) starts at the **lowest (starting)** address
- **Little-endian**: **Least** Significant Byte (“**little end**”) starts at the **lowest (starting)** address
- Example: 32-bit integer with 4-byte data



## Byte Ordering Example

Decimal:	12345
Binary:	0011 0000 0011 1001
Hex:	3 0 3 9

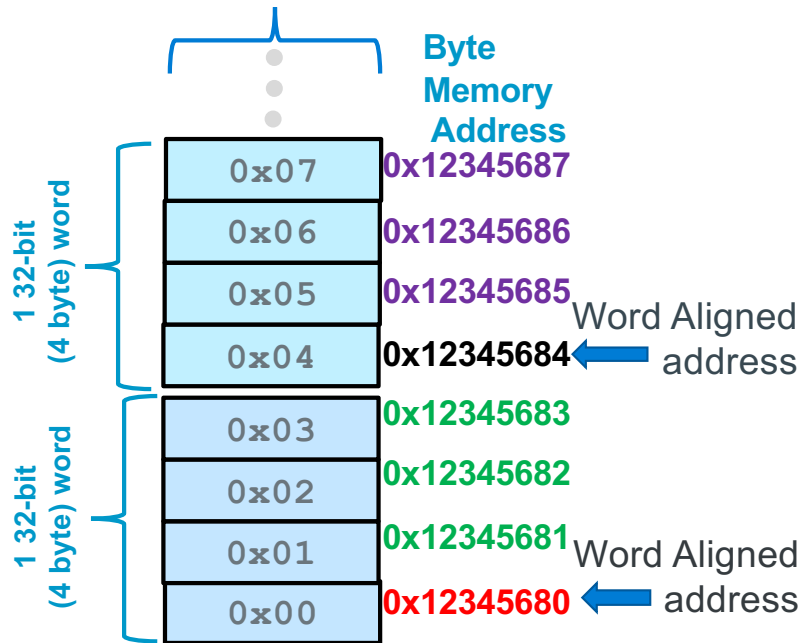
```
int x = 12345;  
// or x = 0x00003039; // show all 32 bits
```



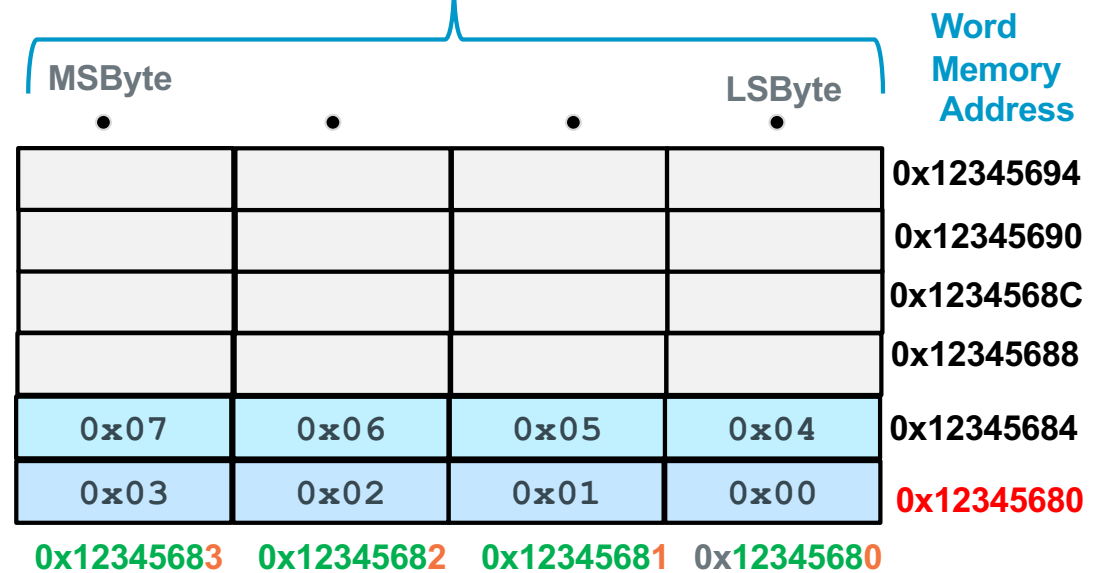
# Byte Addressable Memory Shown as 32-bit words

1 byte Memory Content

One byte per row



Contents of Memory  
One 32-bit (4 byte) word per row



Byte address

**Observation**  
32-bit aligned addresses  
rightmost 2 bits of the address are always 0

# Defining Static Variables: Allocation and Initialization

Variable SIZE	Directive	.align	C static variable Definition	Assembler static variable Definition
8-bit char (1 byte)	.byte		char chx = 'A' char string[] = { 'A', 'B', 'C', 0 };	chx: .byte 'A' string: .byte 'A', 'B', 0x42, 0
16-bit int (2 bytes)	.hword .short	1	short length = 0x55aa;	length: .hword 0x55aa
32-bit int (4 bytes)	.word .long	2	int dist = 5; int *distptr = &dist;  int array[] = { 12, ~0x1, 0xCD, -1 };	dist: .word 5 distptr: .word dist  array: .word 12, ~0x1, 0xCD, -3
strings '\0' term	.string		char class[] = "cse30";	class: .string "cse30"

```
int num;           // 4 bytes
int *ptr = &num;   // 4 bytes
char *lit = "456"; // 4bytes, "456" string literal
char msg[] = "123"; // 4 bytes - array
```

```
.bss
num: .word 0
.data
ptr: .word num
lit: .word .Lmsg
msg: .string "123"
.section .rodata
.Lmsg: .string "456"
```

initializes  
a pointer

## Defining Static Array Variables

```
Label:    .size_directive expression, ... expression
```

```
In C:      int int_buf[100];
           int array[] = {1, 2, 3, 4, 5};
           char buffer[100];

.bss
int_buf:    .space 400    // convert 100 to 400 bytes
char_buf:   .space 100

.data
array:      .word 1, 2, 3, 4, 5
one_buf:    .space 100, 1 // 100 bytes each byte filled with 1
```

**.space size, fill**

- Allocates **size** bytes, each of which contain the value **fill**
- Both **size** and **fill** are absolute expressions
- If the comma and **fill** are **omitted**, **fill** is assumed to be **zero**
- **.bss section**: Must be used **without a specified fill**

# Static Variable Alignment: Using .align

Accessing **address aligned** memory based on data type has the best performance



SIZE	Directive	Address ends in	Align Directive
8-bit char -1 byte	.byte	0b..0 or 0b..1	
16-bit int -2 bytes	.hword .short	0b..0	.align 1
32-bit int -4 bytes	.word .long	0b..00	.align 2

4 bytes	2 Bytes	1 Byte	Addr. (hex)
	Addr = 0x0E		0x0F
			0x0E
Addr = 0x0C	Addr = 0x0C		0x0D
			0x0C
	Addr = 0x0A		0x0B
Addr = 0x08	Addr = 0x08		0x0A
			0x09
			0x08
	Addr = 0x06		0x07
Addr = 0x04	Addr = 0x04		0x06
			0x05
			0x04
	Addr = 0x02		0x03
Addr = 0x00	Addr = 0x00		0x02
			0x01
			0x00

- .align n** before variable definition to specify memory alignment requirements
- Tells the assembler the **next line that allocates memory** must **start** at the next higher memory address **where** the lower **n** address bits are zero
  - At the **first use of any Segment directive**, alignment **starts at an 8-byte aligned address** (for doubles)
  - Easy approach: Allocate from largest size variables to smallest size variables

# Data Segment Variable Alignment

```
.data
ch:    .byte 'A','B','C','D','E'
str:    .string "HIT"
ary:    .hword 0, 1
a:      .byte 'A'
b:      .byte 'B'
xx:     .word 2
```

```
% gcc -c -Wa,-ahlns all.S
1          .data
2 0000 41424344 ch:    .byte 'A','B','C','D','E'
2         45
3 0005 48495400 str:    .string "HIT"
4 0009 00000100 ary:    .hword 0, 1
5 000d 41      a:      .byte 'A'
6 000e 42      b:      .byte 'B'
8 000f 02000000 xx:     .word 2
```

address      contents

- Output on the right side is generated by:
- `%gcc -c -Wa,-ahlns all.S`

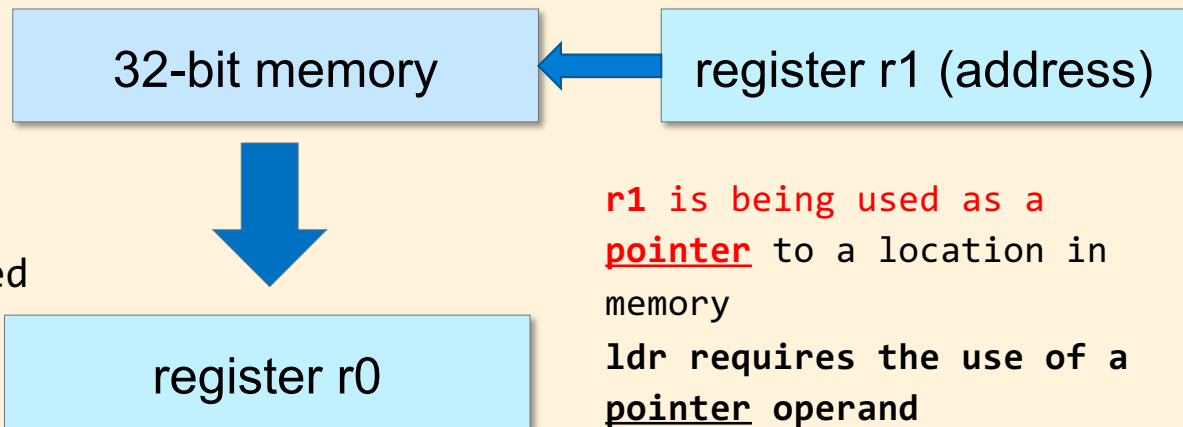
```
.data
xx:     .word 2
ch:     .byte 'A','B','C','D','E'
        .align 2
str:    .string "HI"
        .align 1
ary:    .hword 0, 1
a:      .byte 'A'
b:      .byte 'B'
```

```
gcc -c -Wa,-ahlns all.S
1          .data
2 0000 02000000 xx:     .word 2
3 0004 41424344 ch:     .byte 'A','B','C','D','E'
3         45
4 0009 00000000        .align 2
5 000c 484900    str:    .string "HI"
6 000f 00        .align 1
7 0010 00000100 ary:    .hword 0, 1
8 0014 41      a:      .byte 'A'
9 0015 42      b:      .byte 'B'
```

## Load/Store: Register Base Addressing

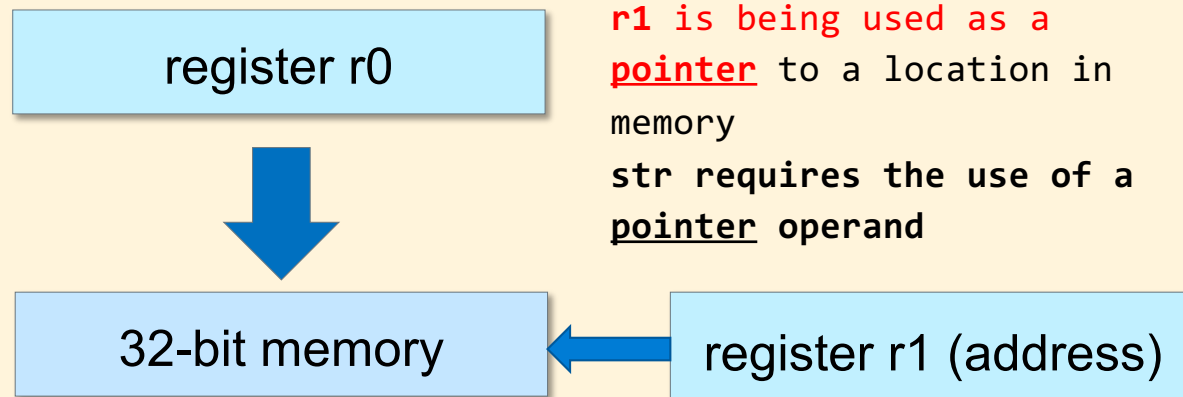
**ldr r0, [r1]**

Copies a 32-bit word from the memory location whose address is contained in r1 (r1 is a pointer) into register r0



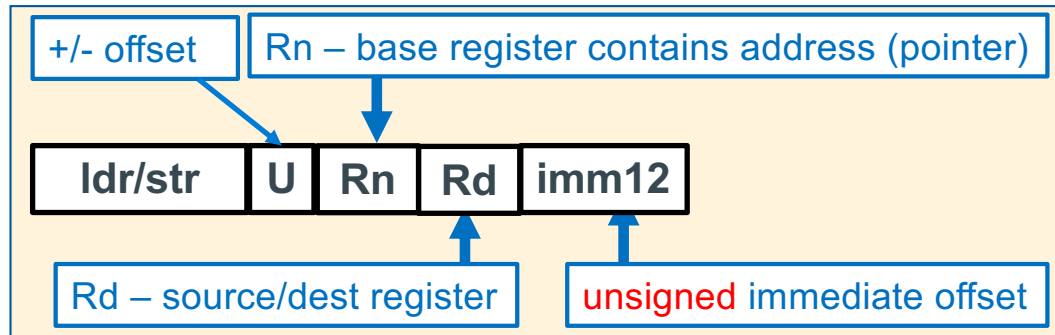
**str r0, [r1]**

Copies all 32 bits of the value held in register r0 to the 32-bit memory location contained in register r1 (r1 pointer)





# LDR/STR – Base Register + Immediate Offset Addressing



- **Register Base Addressing:**

- **Pointer Address:** Rn; **source/destination data:** Rd
- **Unsigned pointer address** is stored in the **base register**

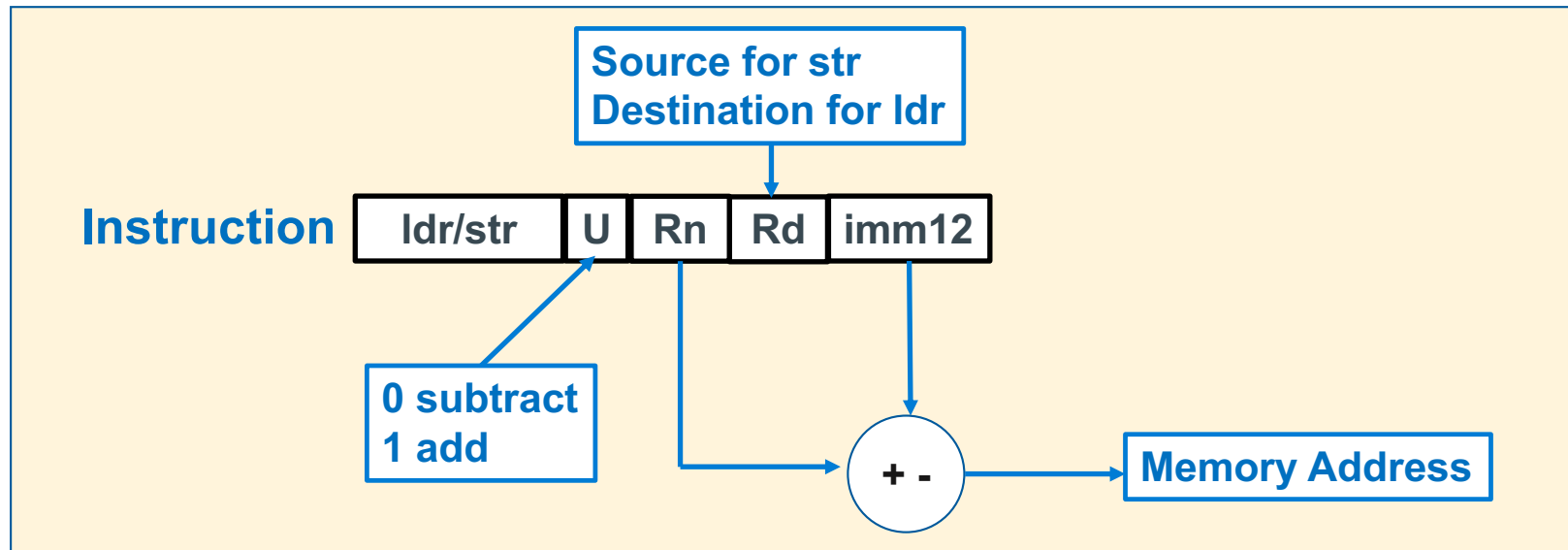
- **Register Base + immediate offset Addressing:**

- **Pointer Address** = register content + immediate offset
- **Unsigned offset integer immediate value (bytes)** is added or subtracted (**U bit above says to add or subtract**) from the **pointer address** in the **base register**

```
ldr/str  Rd,  [Rn, +/- imm12]  // base register pointer + offset  imm12 in bytes
                                     -4095 <= imm12 <= 4095 (bytes)

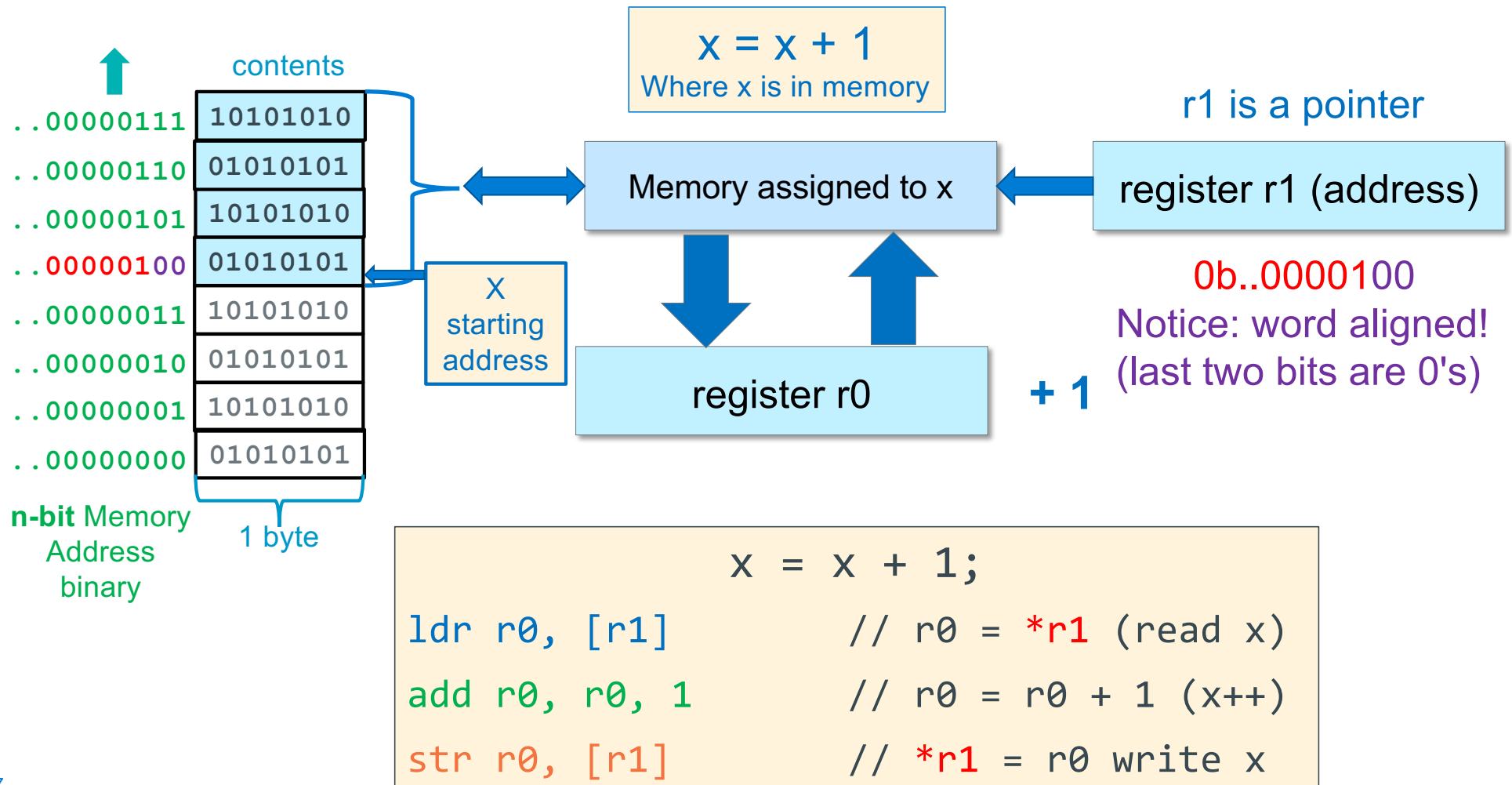
ldr/str  Rd,  [Rn]              // base register pointer + 0 offset (imm12 is 0)
```

## ldr/str Register Base and Register + Immediate Offset Addressing



Syntax	Address	Examples
<code>ldr/str Rd, [Rn +/- constant]</code> constant is in bytes	<code>Rn + or - constant</code> same $\longrightarrow$	<code>ldr r0, [r5,100]</code> <code>str r1, [r5, 0]</code> <code>str r1, [r5]</code>

# Example Base Register Addressing Load – Modify – Store



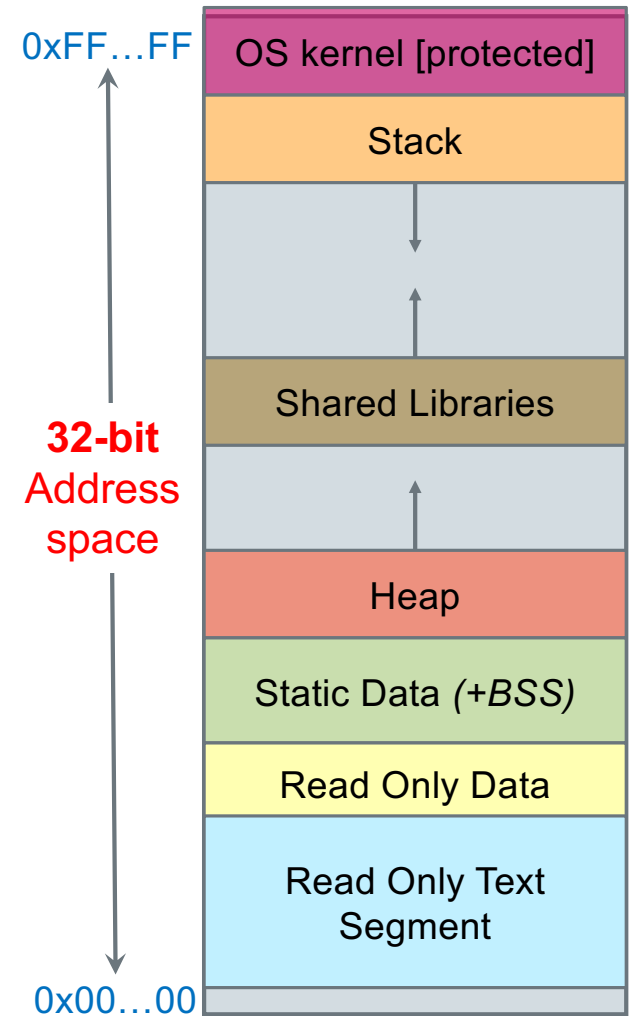
## How to get a memory pointer into a register?

- Assembler **creates a table of pointers** in the **text segment** called the **literal table**
- For each variable in one of the data segments you reference in a special form of the ldr instruction (next slide), the assembler makes an entry for that variable whose contents is the 32-bit Label address

```
.bss
y: .space 4c

.data
x: .word 200

.text
// your code
// last line of your code
// below is created by the assembler
.word y      // contents: 32-bit address of y
.word x      // contents: 32-bit address of x
```



# Loading and using pointers in registers

- Tell the assembler to create and USE a literal table to obtain the address (Lvalue) of a label into a register:

`ldr/str Rd, =Label // Rd = address`

- Example to the right: `y = x;`

two step to **load** a **memory** variable

- load the pointer to the memory
- read (load) from \*pointer

two steps **store** to a **memory** variable

- load the pointer to the memory
- write (store) to \*pointer

```
.bss
y: .space 4
```

```
.data
x: .word 200
```

```
.text
// function header
main:

// load the address, then contents
// using r2
ldr r2, =x      // int *r2 = &x
ldr r2, [r2]    // r2 = *r2;
// &x was only needed once above
// Note: r2 was a pointer then an int
// no "type" checking in assembly!

// store the contents of r2
ldr r1, =y      // int *r1 = &y
str r2, [r1]    // *r1 = r2
...
```

## How to use the literal table to get a big constant into a register

- In data processing instructions, the field **imm8 + rotate 4 bits** is too small to store many numbers outside of the range of -256 to 255, how do you get larger immediate values into a register?



fails



```
mov    r0, 1023
```

xxx.s:24: Error: invalid constant (3ff) after fixup

replacement



```
ldr    r0, =1023
```

- Answer: use **ldr** instruction with the constant as an operand: **=constant**
- Assembler creates a **literal table entry** with the **constant**

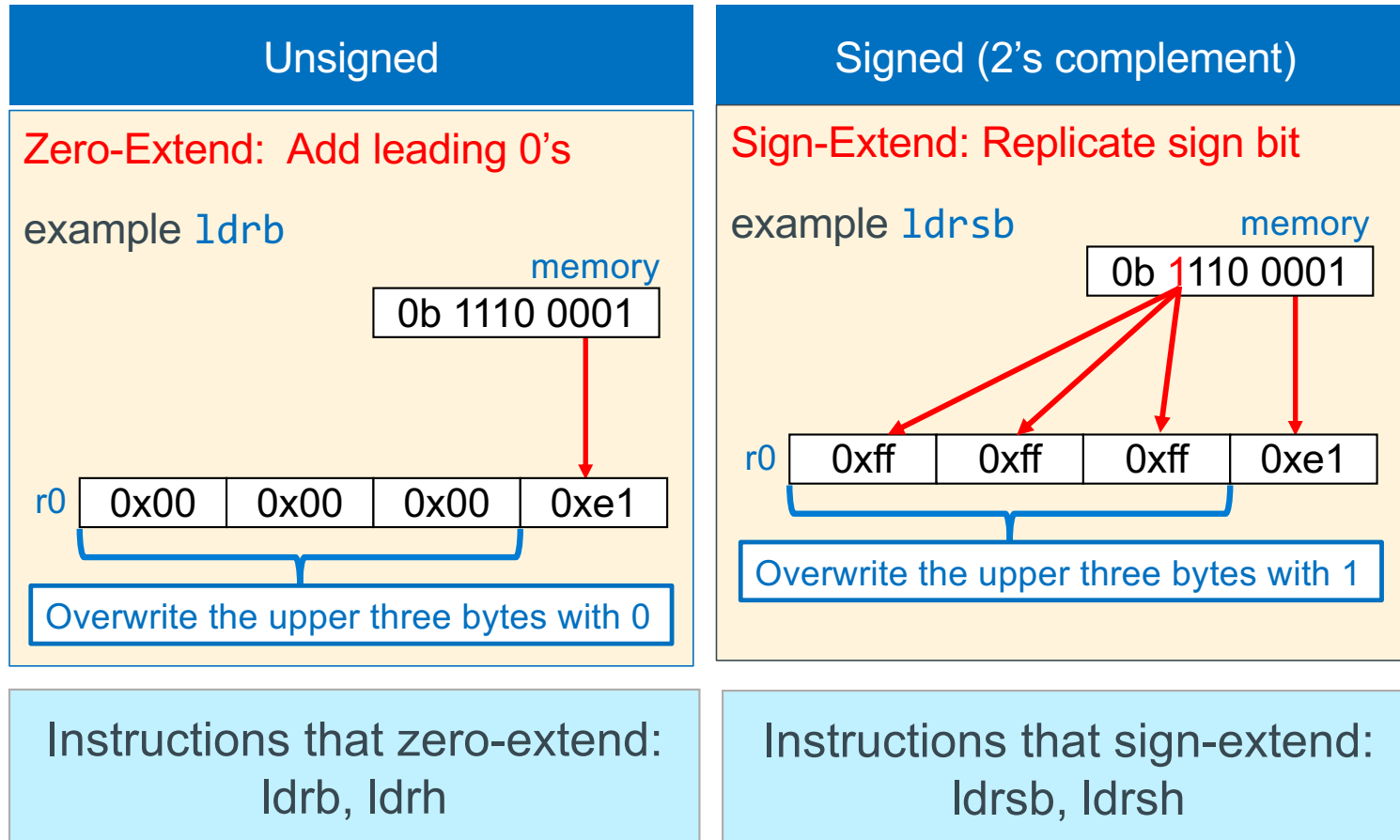
```
ldr    Rd, =constant    // =constant
ldr    r1, =0x2468abcd   // loads the constant 0x246abcd into r1
```

## Loading and Storing: Variations List

- Load and store have **variations** that move 8-bits, 16-bits and 32-bits
- Load into a register with less than 32-bits will **set the upper bits not filled from memory differently depending** on which **variation of the load instruction** is used
- Store will only select the lower 8-bit, lower 16-bits or all 32-bits of the register to copy to memory

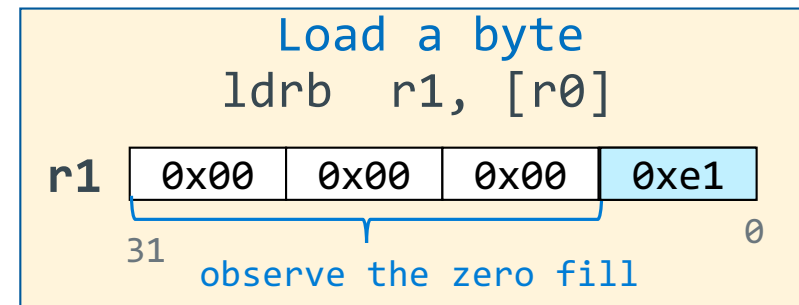
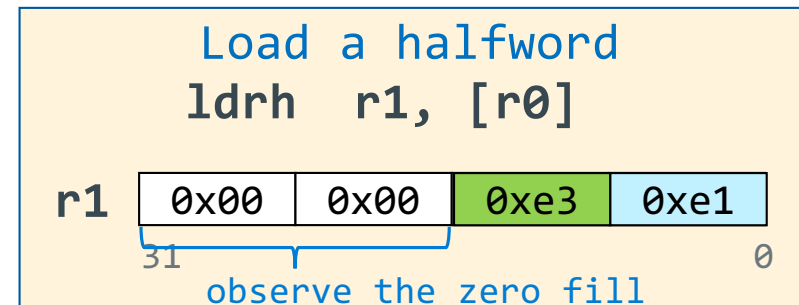
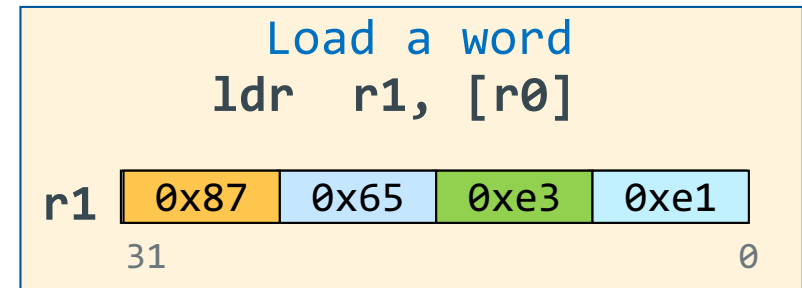
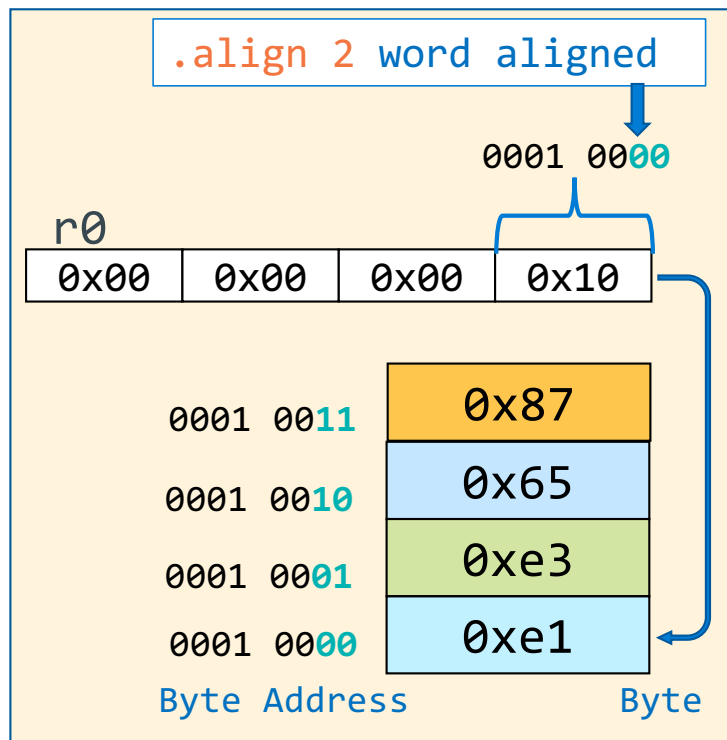
Instruction	Meaning	Sign Extension	Memory Address Requirement
<b>ldr<b>sb</b></b>	load signed byte	sign extension	none (any byte)
<b>ldrb</b>	load unsigned byte	zero fill (extension)	none (any byte)
<b>ldr<b>sh</b></b>	load signed halfword	sign extension	halfword (2-byte aligned)
<b>ldrh</b>	load unsigned halfword	zero fill (extension)	halfword (2-byte aligned)
<b>ldr</b>	load word	---	word (4-byte aligned)
<b>str<b>b</b></b>	store low byte (bits 0-7)	---	none (any byte)
<b>str<b>h</b></b>	store halfword (bits 0-15)	---	halfword (2-byte aligned)
<b>str</b>	store word (bits 0-31)	---	word (4-byte aligned)

## Loading 32-bit Registers From Memory Variables < 32-Bits Wide

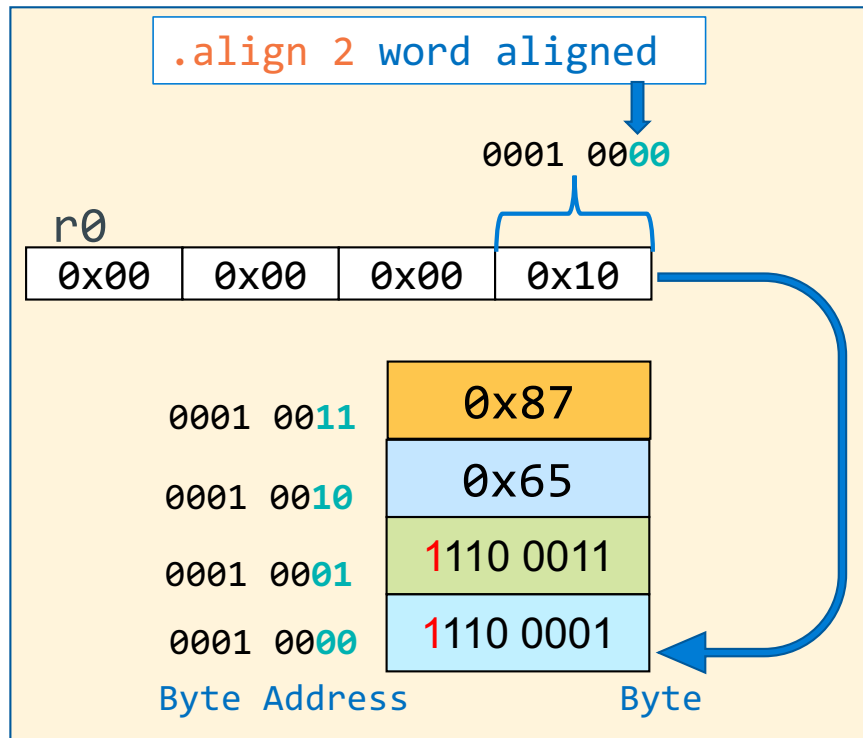




# Load a Byte, Half-word, Word

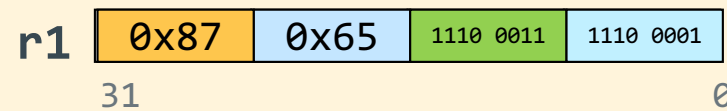


## Signed Load a Byte, Half-word, Word



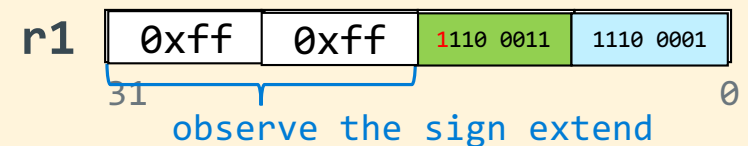
Load a word (no change)

ldr r1, [r0]



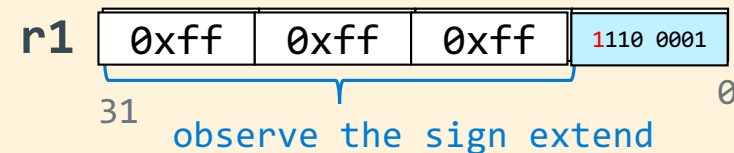
Load a halfword

ldrsh r1, [r0]

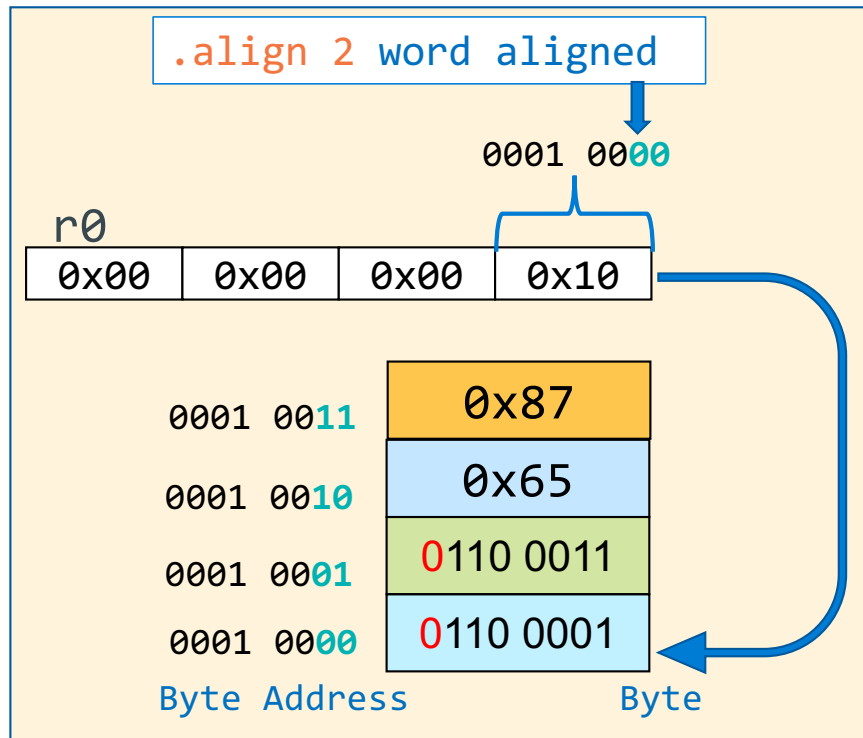


Load a byte

ldrsb r1, [r0]

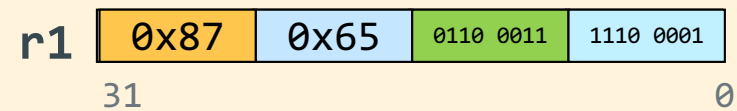


## Signed Load a Byte, Half-word, Word



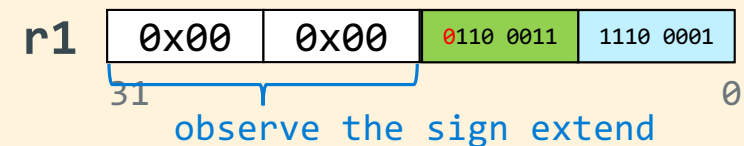
Load a word (no change)

ldr r1, [r0]



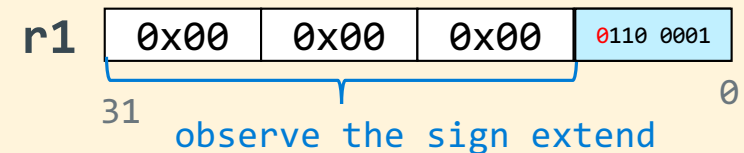
Load a halfword

ldrsh r1, [r0]

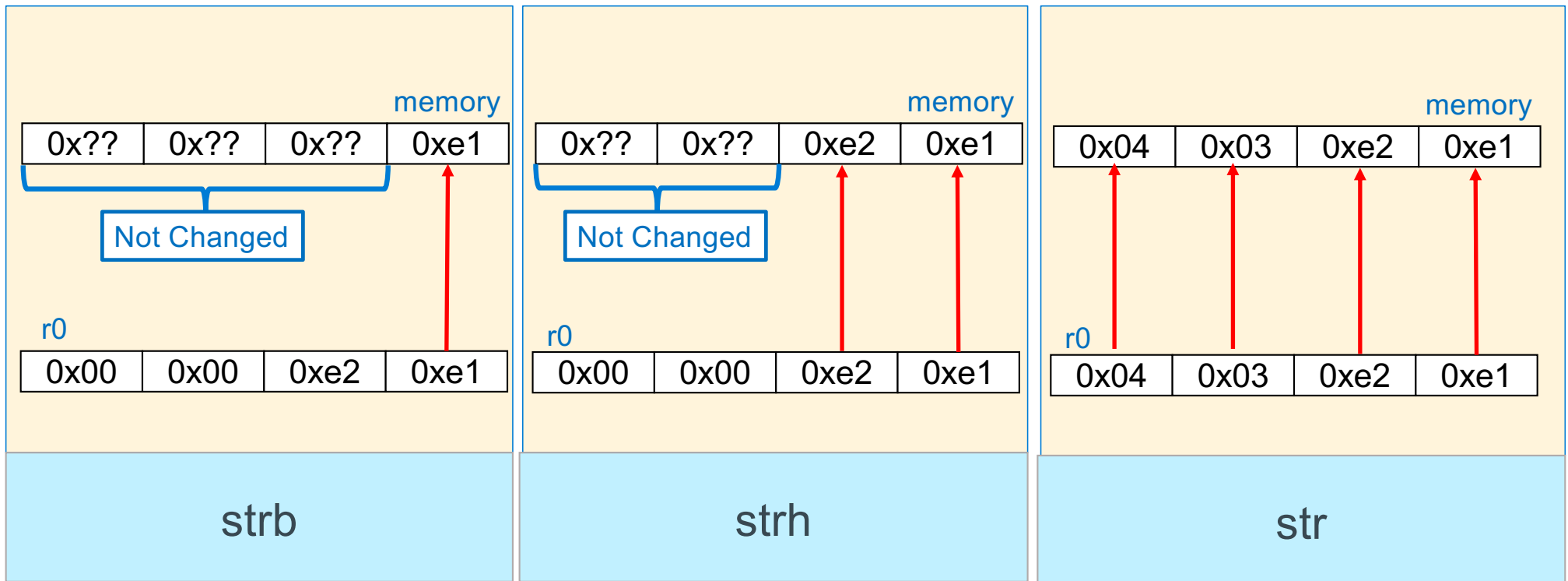


Load a byte

ldrshb r1, [r0]



## Storing 32-bit Registers To Memory 8-bit, 16-bit, 32-bit



# Store a Byte, Half-word, Word

initial value in r0

0x20	0x00	0x00	0x00
------	------	------	------

**Store a byte**  
`strb r1, [r0]`

r1: 

0x87	0x65	0xe3	0xe1
------	------	------	------

  
 31 0

Byte Address      Byte

0x20000003	0x33
0x20000002	0x22
0x20000001	0x11
0x20000000	0xe1

observe other bytes NOT altered

**Store a halfword**  
`strh r1, [r0]`

r1: 

0x87	0x65	0xe3	0xe1
------	------	------	------

  
 31 0

Byte Address      Byte

0x20000003	0x33
0x20000002	0x22
0x20000001	0xe3
0x20000000	0xe1

**Store a word**  
`str r1, [r0]`

r1: 

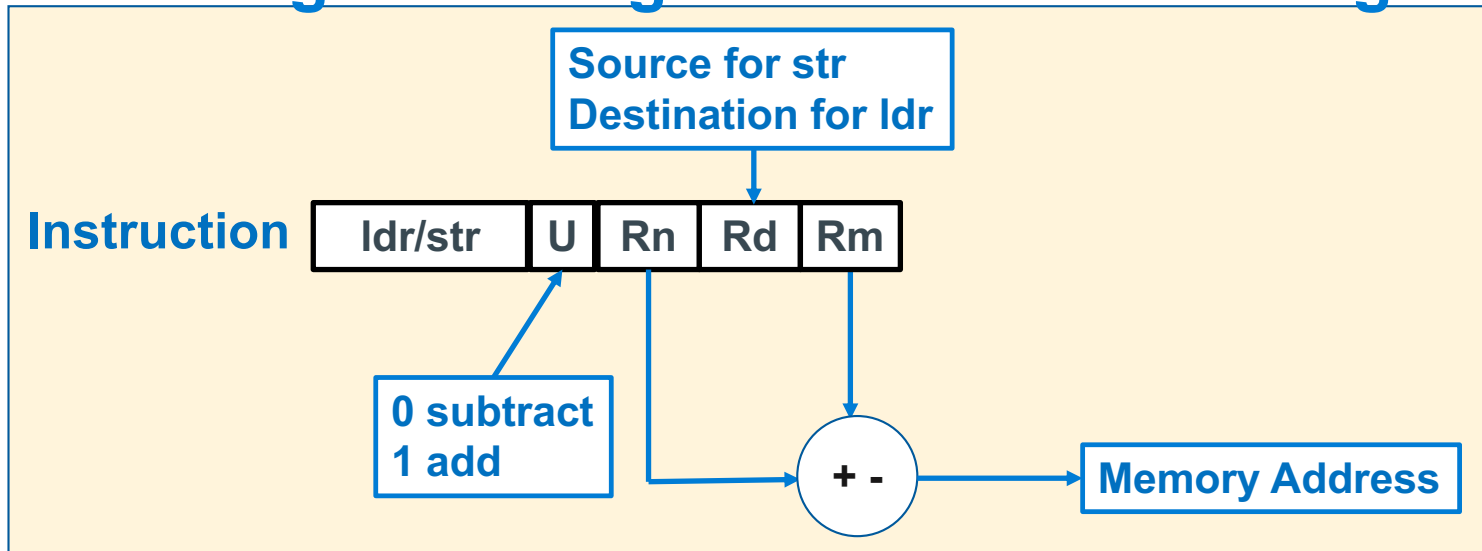
0x87	0x65	0xe3	0xe1
------	------	------	------

  
 31 0

Byte Address      Byte

0x20000003	0x87
0x20000002	0x65
0x20000001	0xe3
0x20000000	0xe1

## ldr/str Base Register + Register Offset Addressing



**Pointer Address = Base Register + Register Offset**

- **Unsigned** offset integer **in a register (bytes)** is either added/subtracted from the **pointer address** in the **base register**

Syntax	Address	Examples
<code>ldr/str Rd, [Rn +/- Rm ]</code>	$Rn + \text{ or } - Rm$	<code>ldr r0, [r5, r4]</code> <code>str r1, [r5, r4]</code>

## Reference: Addressing Mode Summary for use in CSE30

index Type	Example	Description
Pre-index immediate	<code>ldr r1, [r0]</code>	$r1 \leftarrow \text{memory}[r0]$ $r0$ is unchanged
Pre-index immediate	<code>ldr r1, [r0, 4]</code>	$r1 \leftarrow \text{memory}[r0 + 4]$ $r0$ is unchanged
Pre-index immediate	<code>str r1, [r0]</code>	$\text{memory}[r0] \leftarrow r1$ $r0$ is unchanged
Pre-index immediate	<code>str r1, [r0, 4]</code>	$\text{memory}[r0 + 4] \leftarrow r1$ $r0$ is unchanged
Pre-index register	<code>ldr r1, [r0, +-r2]</code>	$r1 \leftarrow \text{memory}[r0 \pm r2]$ $r0$ is unchanged
Pre-index register	<code>str r1, [r0, +-r2]</code>	$\text{memory}[r0 \pm r2] \leftarrow r1$ $r0$ is unchanged

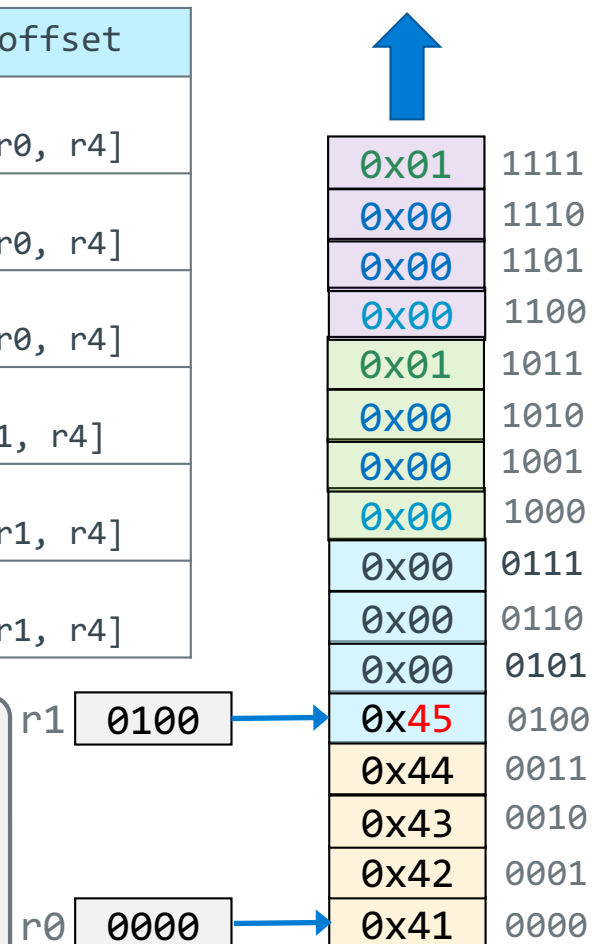
## Array addressing with ldr/str

Array element	Base addressing	Immediate offset	register offset
ch[0]	ldrb r2, [r0]	ldrb r2, [r0, 0]	mov r4, 0 ldrb r2, [r0, r4]
ch[1]	add r0, r0, 1 ldrb r2, [r0]	ldrb r2, [r0, 1]	mov r4, 1 ldrb r2, [r0, r4]
ch[2]	add r0, r0, 2 ldrb r2, [r0]	ldrb r2, [r0, 2]	mov r4, 2 ldrb r2, [r0, r4]
x[0]	ldr r2, [r1]	ldr r2, [r1, 0]	mov r4, 0 ldr r2, [r1, r4]
x[1]	add r1, r1, 4 ldrb r2, [r1]	ldrb r2, [r1, 4]	mov r4, 4 ldrb r2, [r1, r4]
x[2]	add r1, r1, 8 ldrb r2, [r0]	ldrb r2, [r1, 8]	mov r4, 8 ldrb r2, [r1, r4]

table rows are  
independent instructions

```

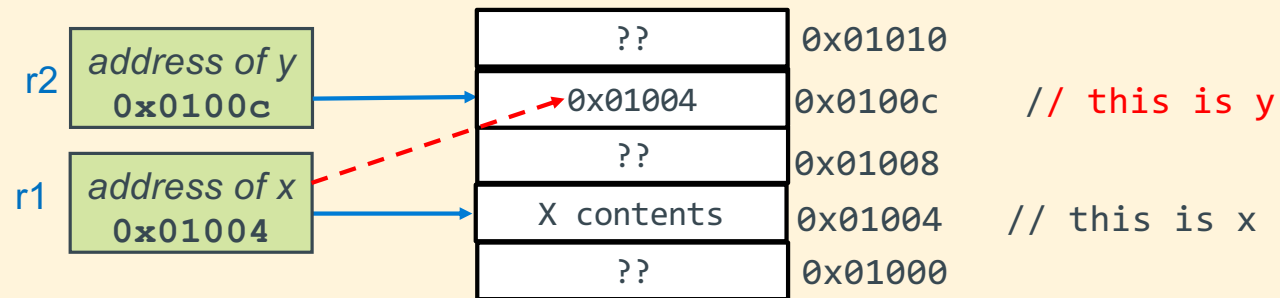
.data
ch: .byte 0x41, 0x42, 0x43, 0x44
x:  .word 0x00000045
   .word 0x01000000
   .word 0x01020304
.text
ldr r0, =ch
ldr r1, =x
    
```





## ldr/str practice - 1

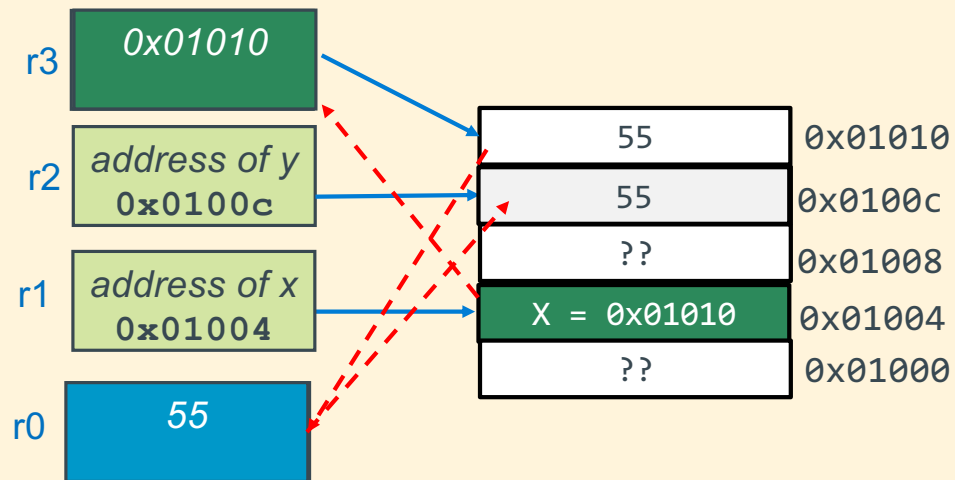
r1 contains the Address of X (defined as int X) in memory; r1 points at X  
r2 contains the Address of Y (defined as int \*Y) in memory; r2 points at Y  
write Y = &X;



str r1, [r2] // y ← &x

## ldr/str practice - 2

r1 contains the Address of X (defined as `int *X`) in memory r1 points at X  
r2 contains the Address of Y (defined as `int Y`) in memory; r2 points at Y  
write `Y = *X;`



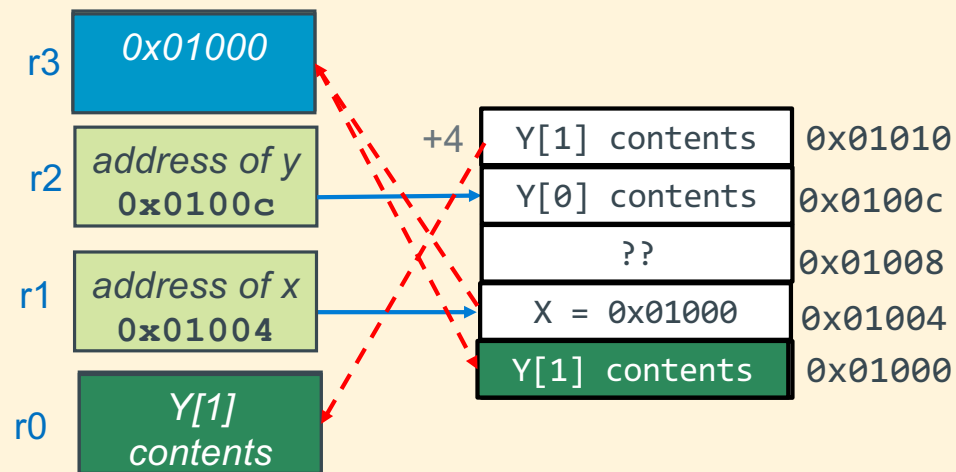
```
ldr    r3, [r1]    // r3 ← x (read 1)
ldr    r0, [r3]    // r0 ← *x (read 2)
str    r0, [r2]    // y ← *x
```

## ldr/str practice - 3

r1 contains Address of X (defined as `int *X`) in memory; r1 points at X

r2 contains Address of Y (defined as `int Y[2]`) in memory; r2 points at `&(Y[0])`

`write *X = Y[1];`



```
ldr    r0, [r2, 4]    // r0 ← y[1]
ldr    r3, [r1]        // r3 ← x
str     r0, [r3]       // *x ← y[1]
```

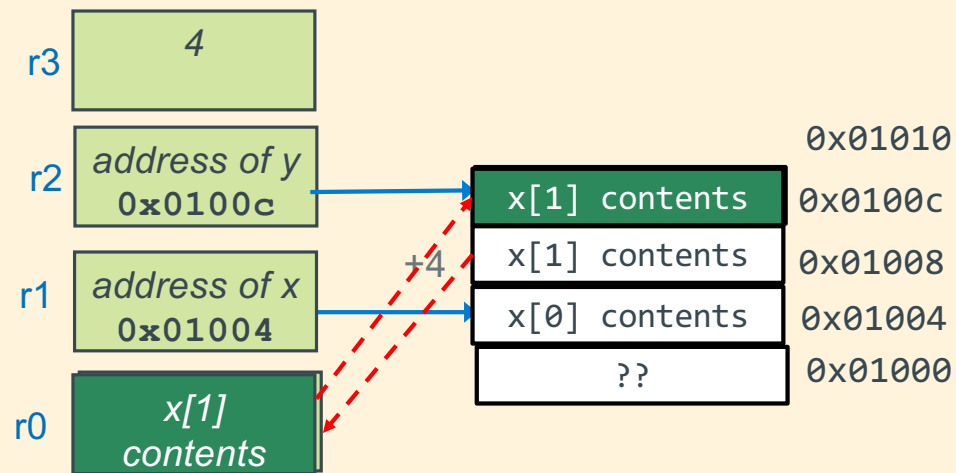
## ldr/str practice - 4

r1 contains Address of X (defined as `int X[2]`) in memory; r1 points at `&(x[0])`

r2 contains Address of Y (defined as `int Y`) in memory; r2 points at Y

r3 contains a 4

write `Y = X[1];`



```
ldr    r0, [r1, r3] // r0 ← x[1]
```

```
str    r0, [r2]     // y ← x[1]
```

## Label (Address) Math

- You can have the assembler calculate some useful values for you
- One common use is calculating the distance in bytes between two labels
- The dot (.) refers to the address on the current line (the next byte after a previous space allocation)

```
.section .rodata
.Lst: .string "The value of x is %d\n"
.equ STSZ, (. - .Lst)    // number of bytes in .Lst includes \0
.equ STLEN, STSZ - 1     // string length of .Lst
```

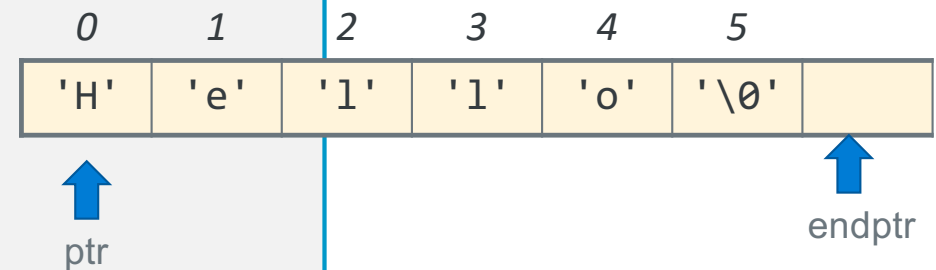
## Example: Base Register Addressing with Arrays

```
#include <stdio.h>
#include <stdlib.h>

char msg[] = "Hello CSE30! We Are CountinG UpPER cASe letters!";

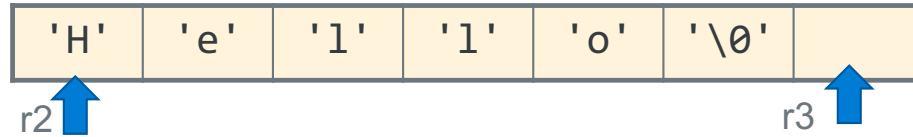
int
main(void)
{
    int cnt = 0;
    char *endpt = msg + sizeof(msg)/sizeof(*msg);
    char *ptr = msg;

    while(ptr < endpt) {
        if ((*ptr >= 'A') && (*ptr <= 'Z'))
            cnt++;
        ptr++;
    }
    return EXIT_SUCCESS;
}
```



## Example: Base Register Addressing with Arrays

- Iterates a pointer (r2) through the array
- r3 contains the address +1 past the end of the string
- MSGSZ is the size of the array (including the '\0') if you wanted to excluded the '\0', then subtract 1 from MSGSZ
- Use **ldrb** as **msg** is an array of chars



```

.data          // segment
msg:.string    "Hello CSE30! We Are CountinG UpPER cASe letters!"
.equ          MSGSZ, (. - msg) // number of bytes in msg
.section .rodata

mov     r1, 0           // initialize cnt
ldrb    r2, =msg        // ptr point to &msg
add     r3, r2, MSGSZ   // endpt points after end

.Lwhile:
cmp     r2, r3          // at end of buffer yet?
bge     .Lexit          loop guard

ldrb    r0, [r2]        // get next char (base addressing)
cmp     r0, 'A'         // is it less than an 'A' ?
blt     .Lendif         // if so, not CAP (short circuit)
cmp     r0, 'Z'         // is it greater than a 'Z'?
bgt     .Lendif         // if so, not CAP
add     r1, r1, 1       // it is a CAP, so increment cnt
.Lendif:
add     r2, r2, 1       // move to next char
b       .Lwhile        //go to loop guard at top of while
.Lexit:

```

## Example: Base Register + Offset Register

```
mov    r1, 0           // initialize cnt
ldr    r2, =msg        // ptr point to &msg
add    r3, r2, MSGSZ   // endpt points after end
.Lwhile:
  cmp   r2, r3          // at end of buffer yet?
  bge   .Lexit

  ldrb  r0, [r2]         // get next char
  cmp   r0, 'A'          // is it less than an 'A' ?
  blt   .Lendif          // if so, not CAP
  cmp   r0, 'Z'          // is it greater than a 'Z'?
  bgt   .Lendif          // if so, not CAP
  add   r1, r1, 1        // is a CAP increment

.Lendif:
  add   r2, r2, 1        // move to next char
  b     .Lwhile          //go to loop guard while top
.Lexit:
```

Using Base register pointer with an end pointer

```
mov    r1, 0
ldr    r2, =msg
mov    r3, 0           // index reg
.Lwhile:
  cmp   r3, MSGSZ      // are we done?
  bge   .Lexit

  ldrb  r0, [r2, r3]
  cmp   r0, 'A'
  blt   .Lendif
  cmp   r0, 'Z'
  bgt   .Lendif
  add   r1, r1, 1

.Lendif:
  add   r3, r3, 1      // index++
  b     .Lwhile
.Lexit:
```

Using Base register pointer + Offset register



## Example: Base Register + Register Offset Two Buffers

```
#include <stdio.h>
#include <stdlib.h>
#define SZ 6

int src[SZ] = {1, 3, 5, 7, 9, 11};

int dest[SZ];
int
main(void)
{
    for (int i = 0; i < SZ; i++)
        dest[i] = src[i];

    return EXIT_SUCCESS;
}
```

- Make sure to index by bytes and increment the index register by `sizeof(int) = 4`

```
.data          // segment
src:.word      1, 3, 5, 7, 9, 11
               .equ      SZ, (. - src) // bytes msg
dest:.space    SZ
               .equ      INT_STEP, 4
...

ldr    r0, =src           // ptr to src
ldr    r1, =dest          // ptr to dest
mov    r2, 0

.Lfor:
    cmp    r2, SZ          // in bytes!
    bge    .Lexit

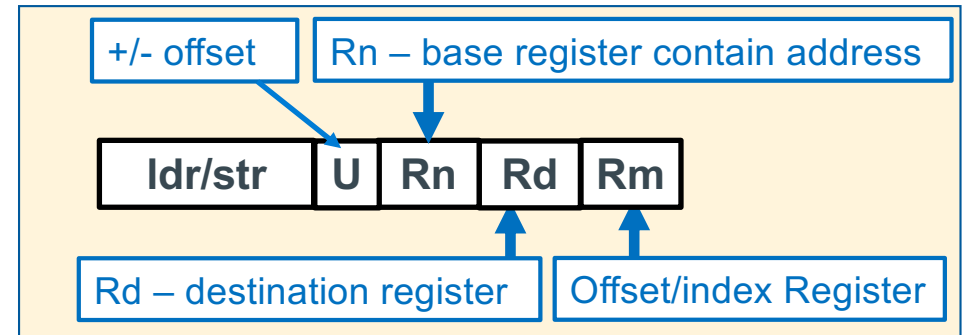
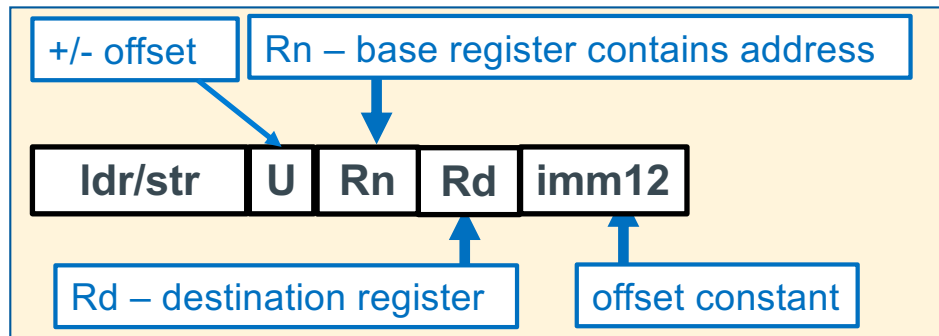
    ldr    r3, [r0, r2]
    str    r3, [r1, r2]
    add    r2, r2, INT_STEP
    b      .Lfor

.Lexit:
```

one increment  
covers both arrays

## Extra Slides

## Reference: LDR/STR – Register To/From Memory Copy

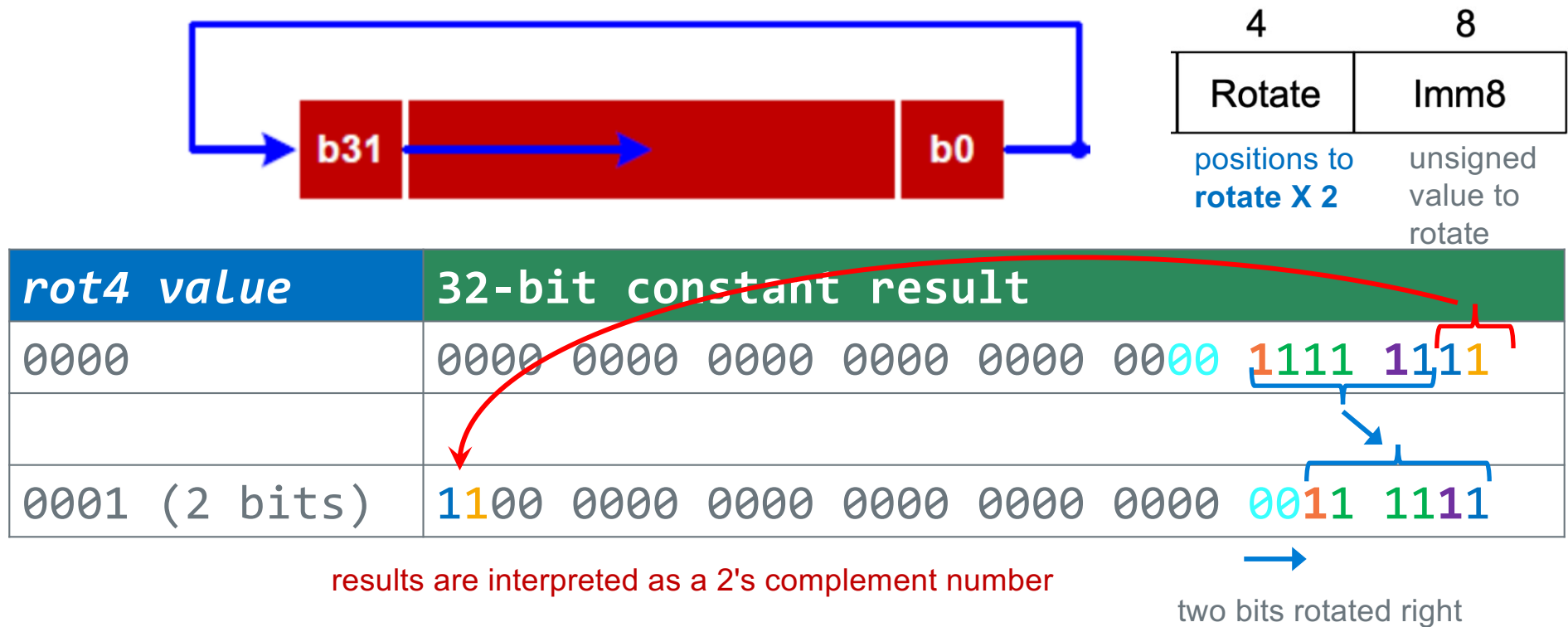


```
ldr/str Rd, [Rn, +/- imm12] // base register pointer + offset  imm12 in bytes
                             -4095 <= imm12 <= 4095 (bytes)
ldr/str Rd, [Rn]             // base register pointer + 0 (imm12 is 0)
ldr/str Rd, [Rn, +/- Rm]     // base register pointer +/- offset register
```

```
ldr      r1, =var_x           // r1 = &var_x
str      r1, =mylabel+4       // *(mylabel+4) = r1
ldr      r1, =0x246abcd       // load an immediate into r1
ldr      r1, [r3]             // y = *r3 (4 bytes)
str      r1, [r0]             // *r0 = r1
ldr      r1, [r3, -4]         // y = *(r3 - 4) (4 bytes)
str      r1, [r0, r2]         // *(r0 + r2) = r1
```

## How are I – Type Constants Encoded in the instruction?

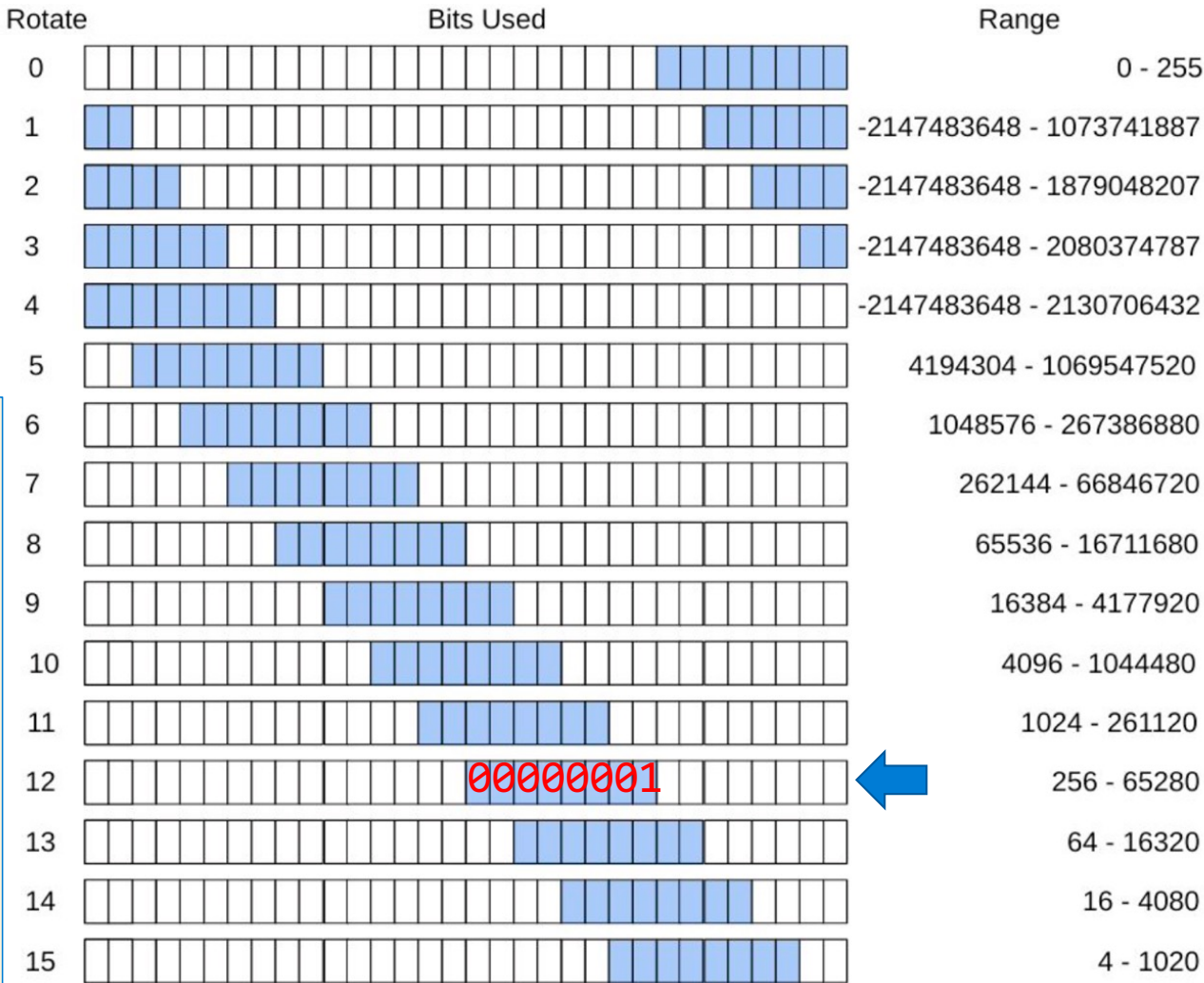
- Aarch32 provides only 8-bits for specifying an immediate constant value
- Without "rotation" immediate values are limited to the range of positive 0-255
- Imm8 expands to 32 bits and does a rotate right to achieve additional constant values (YUCK)



# 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

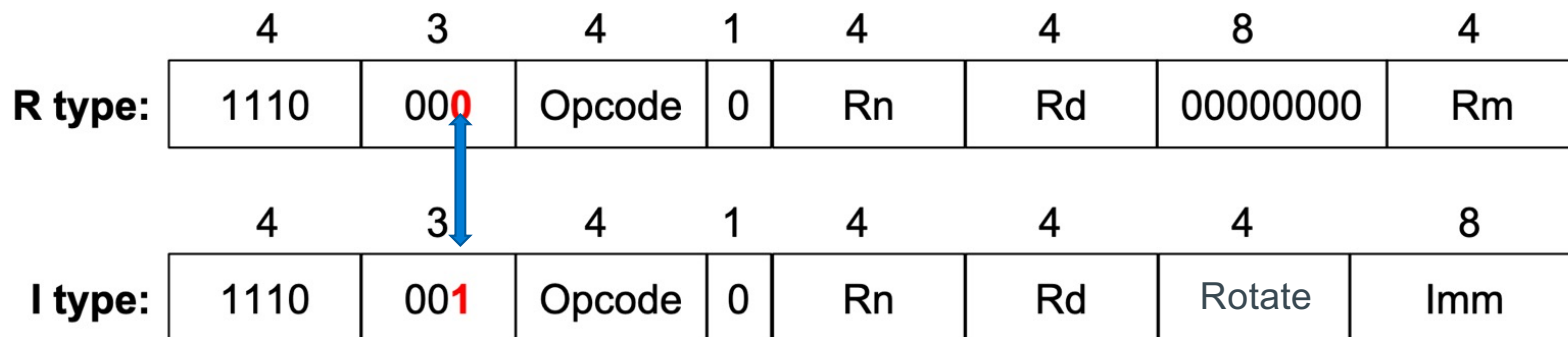
```

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

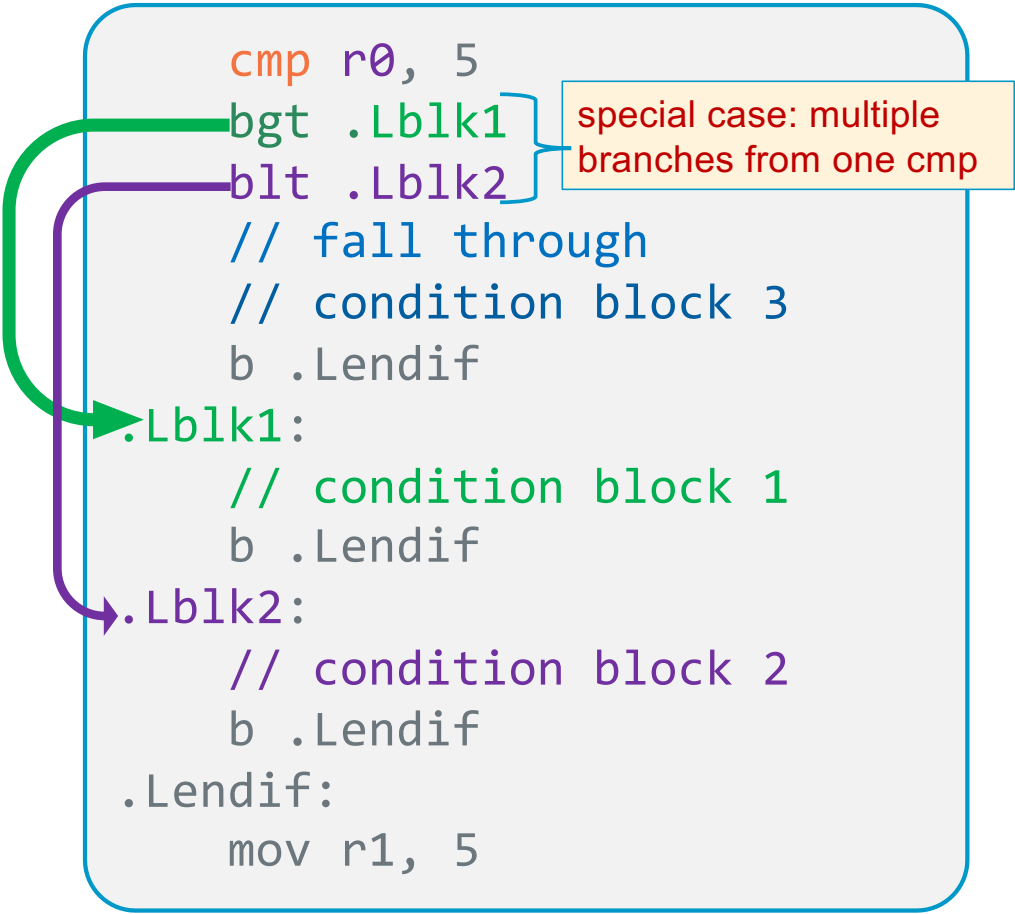


## Program Flow – multiple branches, one cmp

```
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
```





## Literal Table (Array) each entry is a pointer to a different Label

- Assembler automatically inserts into the text segment an array (table) of pointers
- Each entry contains a 32-bit address of one of the labels
- Uses r15 (PC) as base register to load the entry into a reg  

$\text{displacement (bytes)} - 8$

The assembler creates this table before generating the .o file

```
.bss
y: .space 4

.data
x: .word 200

.section .rodata
.Lmsg: .string "Hello World"

.text
main:
(address)ldr r0, [PC, displacement] // replaces: ldr r0, =y
    <last line of your assembly, typically a function return>

.word y      // entry #1 32-bit address for y
.word x      // entry #2 32-bit address for x
.word .Lmsg  // entry #3 32-bit address for .Lmsg
```

## Literal Table (Array) each entry is a pointer to a different Label

The displacement is different for each use. As the PC is different at each instruction

```
.bss
y: .space 4
.data
x: .word 200
.section .rodata
.Lmsg: .string "Hello World"
.text
main:
(address)ldr r0, [PC, displacement1] // replaces: ldr r0, =y
(address)ldr r0, [PC, displacement2] // replaces: ldr r0, =y
<last line of your assembly, typically a function return>
.word y // entry #1 32-bit address for y
.word x // entry #2 32-bit address for x
.word .Lmsg // entry #3 32-bit address for .Lmsg
```

displacement1 - 8

displacement2 - 8

# ARM Assembly Source File: Header

## File Header

At the top of every  
ARM source file

```
.arch    armv6           // armv6 architecture
.arm     // arm 32-bit instruction set
.fpu     vfp             // floating point co-processor
.syntax  unified         // modern syntax
```

```
// Contents of the other memory segment include .text (your code)
```

### **.arch** <architecture>

- Specifies the target architecture to generate machine code
- Typically specify oldest ARM arch you want the code to run on – most arm CPUs are backwards compatible

### **.arm**

- Use the 32-bit ARM instructions, There is an alternative 16-bit instruction set called thumb that we will not be using

### **.fpu** <version>

- Specify which floating point co-processor instructions to use (OPTIONAL we will not be using floating point)

# ARM Assembly Source File: Header and Footer

## File Header

At the top of every ARM source file

```
.arch    armv6           // armv6 architecture
.arm     // arm 32-bit instruction set
.fpu     vfp             // floating point co-processor
.syntax  unified         // modern syntax
```

```
// Contents of the other memory segment include .text (your code)
```

## File Footer

At the bottom of every ARM source file

```
.section .note.GNU-stack,"",%progbits // set stack/data non-exec
.end

// everything past the .end is ignored!
// Debugging notes etc
```

## `.syntax unified`

- use the standard ARM assembly language syntax called *Unified Assembler Language (UAL)*

## `.section .note.GNU-stack,"",%progbits`

- tells the linker to **make the stack and all data segments not-executable** (no instructions in those sections) – security measure

## `.end`

- at the end of the source file, everything written after the `.end` is ignored

# Function Header and Footer Assembler Directives

**function entry point**  
address of the first  
instruction in the function  
**Must not be a local label**  
**(does not start with .L)**

```
        .text
Function Header {
    .global myfunc           // make myfunc global for linking
    .type    myfunc, %function // define myfunc to be a function
    .equ     FP_OFF, 4       // fp offset in main stack frame
myfunc:
    // function prologue, stack frame setup
    // your code
    // function epilogue, stack frame teardown
Function Footer {
    .size myfunc, (. - myfunc)
```

**.global function\_name**

- Exports the function name to other files. Required for main function, optional for others

**.type name, %function**

- The **.type** directive sets the **type of a symbol/label name**
- %function** specifies that **name** is a function (name is the address of the first instruction)

**equ FP\_OFF, 4**

- Used for basic stack frame setup; the number 4 will change – later slides

**.size name, bytes**

- The **.size** directive is used to **set the size associated with a symbol**
- Used by the linker to exclude unneeded code and/or data when creating an executable file
- It is also used by the **debugger** gdb
- bytes is best calculated as an expression: (period is the current address in a memory segment)**

**In CSE30 required use: .size name, (. - name)**