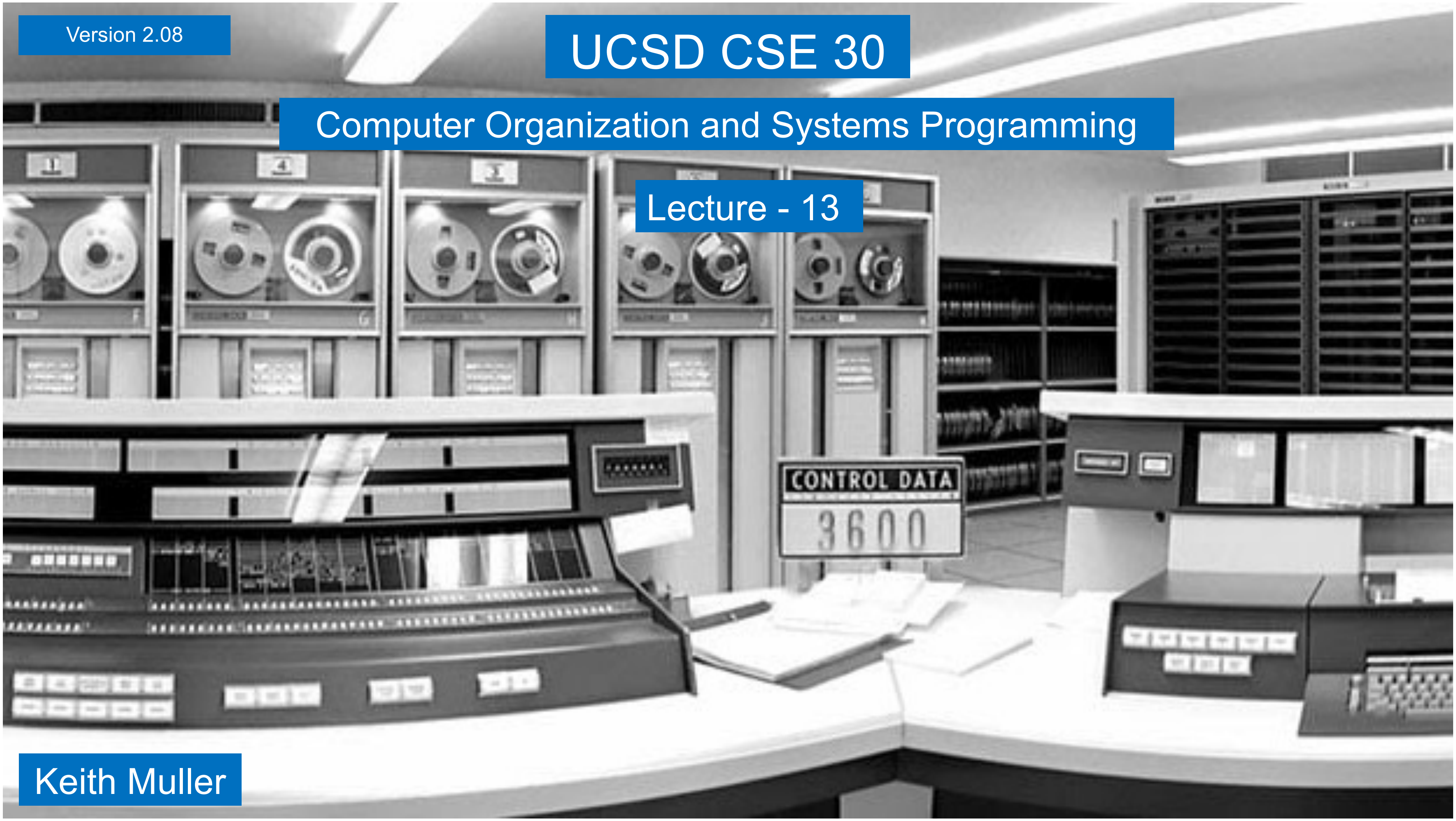


Version 2.08

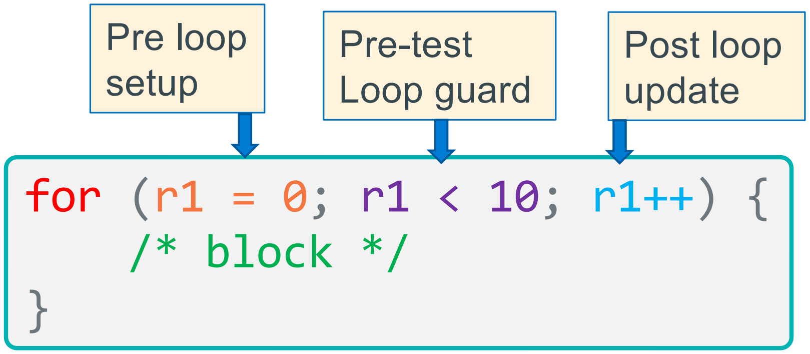
UCSD CSE 30

Computer Organization and Systems Programming

Lecture - 13

Keith Muller

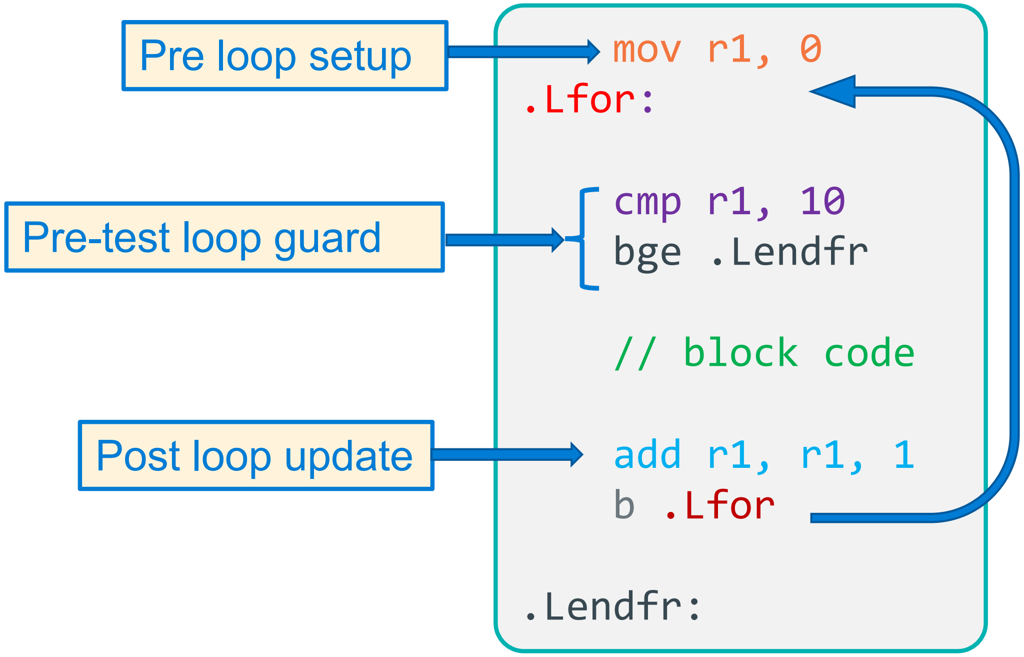
Program Flow – Counting (For) Loop Version 1



```
for (r1 = 0; r1 < 10; r1++) {
    /* block */
}
```

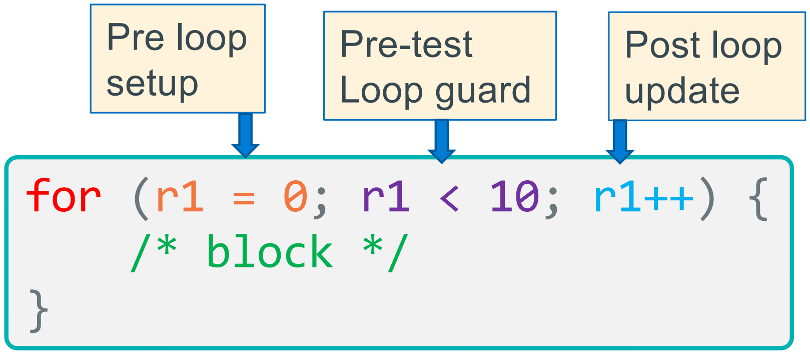
A **counting loop** has three parts:

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



```
     mov r1, 0
.Lfor:

     cmp r1, 10
     bge .Lendfr

     // block code

     add r1, r1, 1
     b  .Lfor

.Lendfr:
```

Program Flow – Counting (For) Loop – Version 2



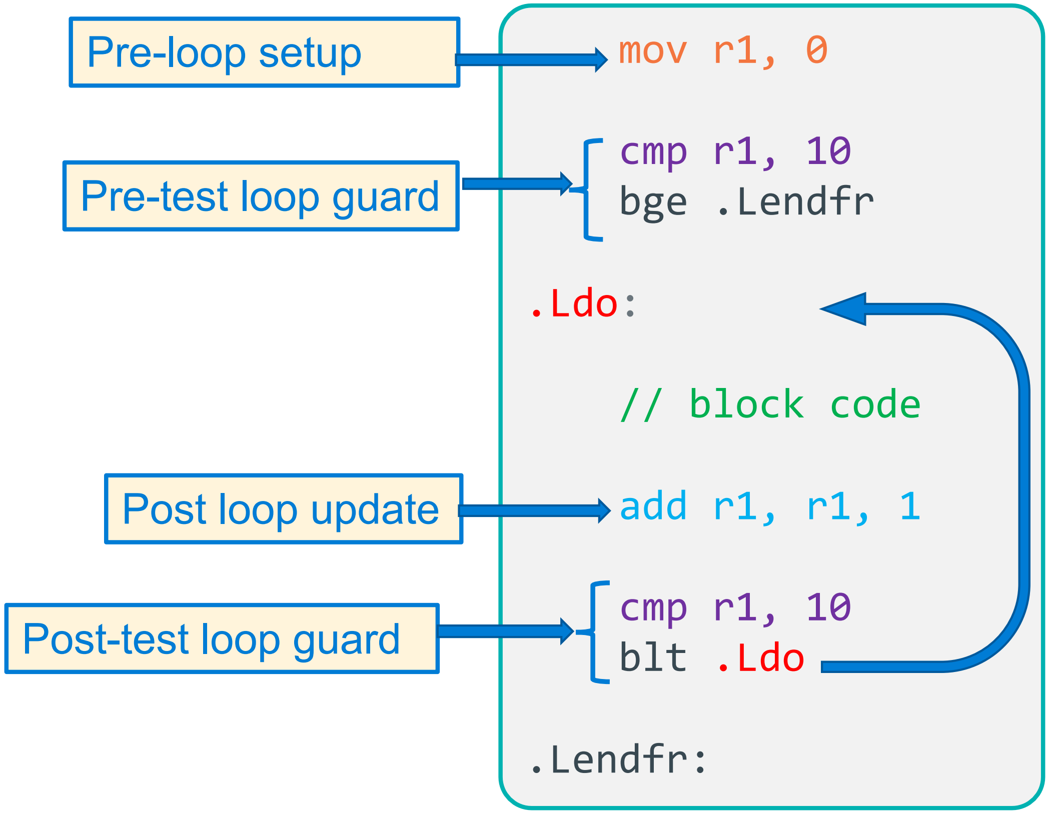
Pre loop setup

Pre-test Loop guard

Post loop update

```
for (r1 = 0; r1 < 10; r1++) {
    /* block */
}
```

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



Pre-loop setup

Pre-test loop guard

Post loop update

Post-test loop guard

```
    mov r1, 0

    cmp r1, 10
    bge .Lendfr

.Ldo:

    // block code

    add r1, r1, 1

    cmp r1, 10
    blt .Ldo

.Lendfr:
```

Nested loops

```
for (r3 = 0; r3 < 10; r3++) {
    r0 = 0;

    do {
        r0 = r0 + r1++;
    } while (r1 < 10);

    // fall through
    r2 = r2 + r1;
}
r5 = r0;
```

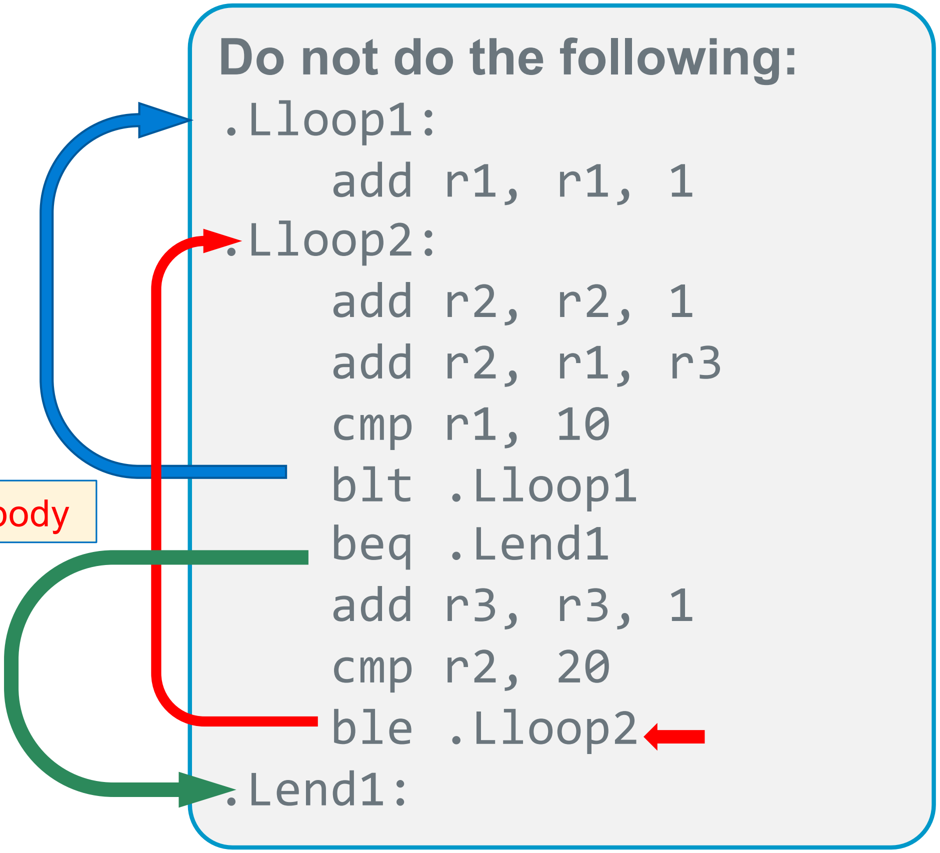
- Nest loop blocks as you would in C or Java

```
    mov r3, 0
.Lfor:
    cmp r3, 10      // loop guard
    bge .Lendfor

    mov r0, 0

.Ldo:
    add r0, r0, r1
    add r1, r1, 1

    cmp r1, 10   // loop guard
    blt .Ldo

    // fall through
    add r2, r2, r1

    add r3, r3, 1 // loop iteration
    b .Lfor
.Lendfor:
    mov r5, r0
```

x

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**

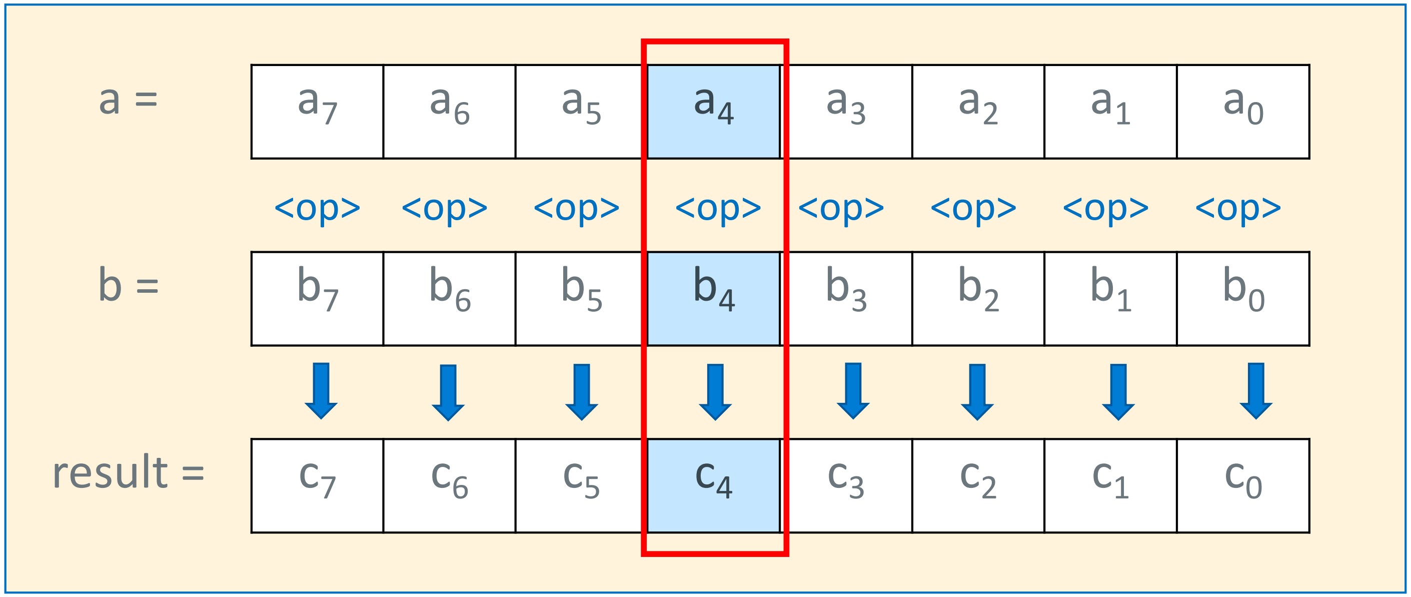


Do not do the following:

```
.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:
```

Bad practice: branch into loop body

What is a Bitwise Operation?



a =	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
	<op>	<op>	<op>	<op>	<op>	<op>	<op>	<op>
b =	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
result =	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

- 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

x

Bitwise (Bit to Bit) Operators in C

```
output = ~a;
```

a	~a
0	1
1	0

Bitwise
NOT

```
~ 1100
  ----
  0011
```

```
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**

Bitwise
AND

```
  0110
& 1100
  ----
  0100
```

```
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**

Bitwise
OR

```
  0110
| 1100
  ----
  1110
```

```
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
EOR

```
  0110
^ 1100
  ----
  1010
```

x

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

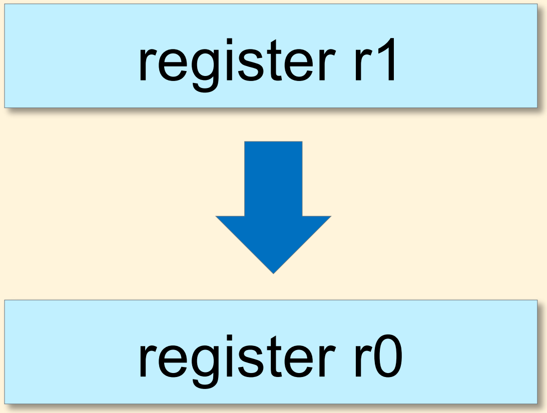
Meaning	*Operator*	*Operator*	*Meaning*
Boolean NOT	!b	~b	Bitwise NOT

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

MVN (not)

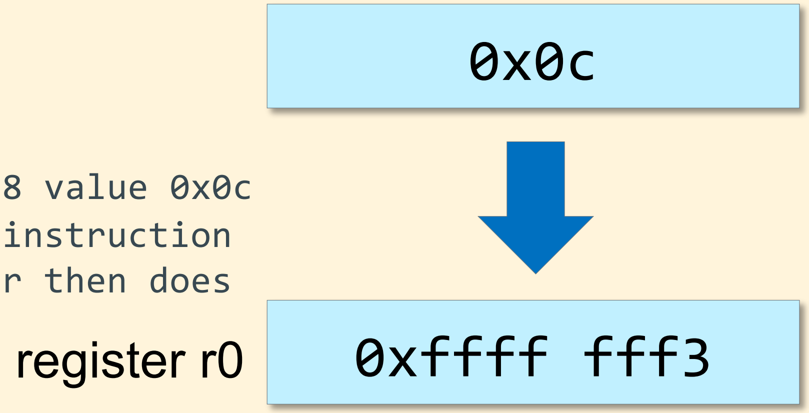
```
mvn   r0, r1


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



register r1

register r0

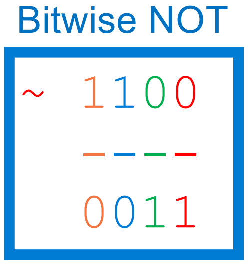
```
mvn   r0, 12


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



0x0c

register r0

0xffff fff3

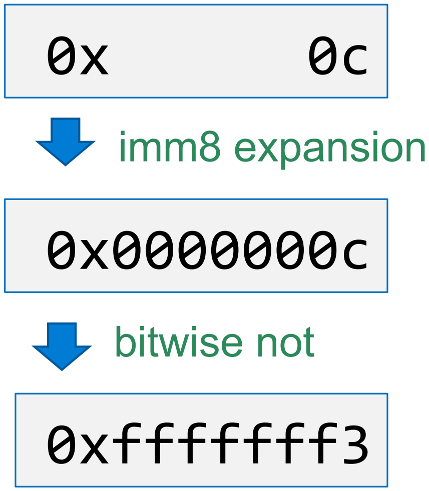


Bitwise NOT

~ 1100

0011

- A **bitwise NOT** operation



0x 0c

imm8 expansion

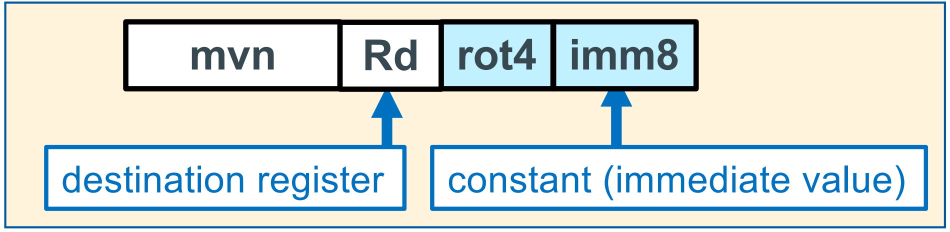
0x0000000c

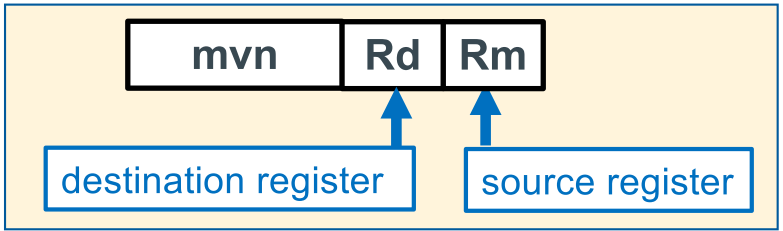
bitwise not

0xfffffff3

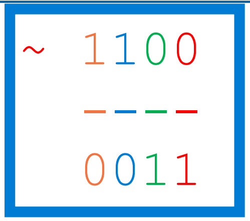
x

mvn – Copies NOT (~)





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

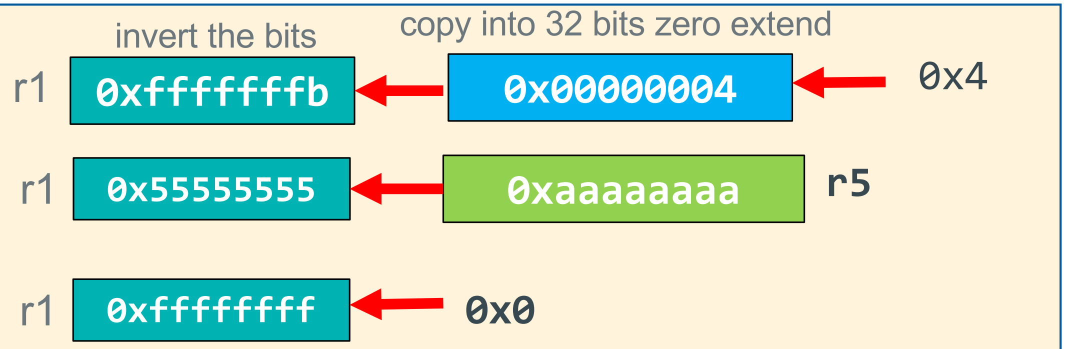


Bitwise NOT

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

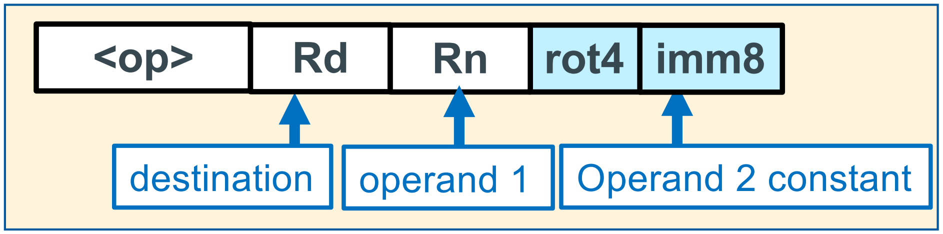
imm8	extended imm8	inverted imm8	signed base 10
0x00	0x00 00 00 00	0xff ff ff ff	-1
0xff	0x00 00 00 ff	0xff ff ff 00	-256

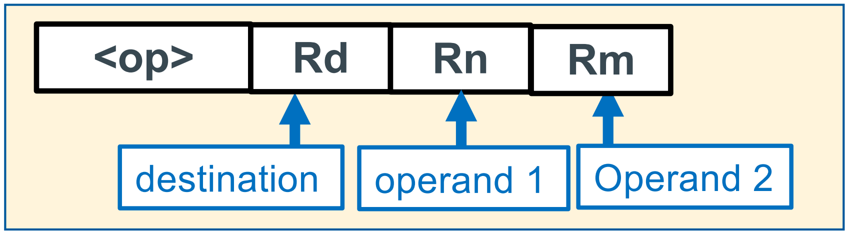
```
mvn    r1, 4          // x = ~4

mvn    r1, r5        // x = ~y in C


mvn    r1, 0         // x = -1
```



x

Bitwise Instructions





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

Bytes: 0 <= imm8 <= 255 + values from "rotating" rot 4 bits

Bitwise <op> description	C Syntax	*Arm <op> Syntax Op2: either register or constant value*	*Operation*
Bitwise **AND**	a & b	and R_d, R_n, Op2	R_d = R_n & Op2
Bitwise **OR**	a | b	orr R_d, R_n, Op2	R_d = R_n | Op2
Exclusive **OR**	a ^ b	eor R_d, R_n, Op2	R_d = R_n ^ Op2
Bitwise **NOT**	a = ~b	mvn R_d, R_n	R_d = ~R_n

x

Bitwise versus C Boolean Operators

Boolean Operators

Meaning	*Operator*
`Boolean AND`	`a && b`
`Boolean OR`	`a || b`
`Boolean NOT`	`!b`

Bitwise Operators

Operator	*Meaning*
`a & b`	`Bitwise AND`
`a | b`	`Bitwise OR`
`~b`	`Bitwise NOT`

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

bitwise & versus Boolean &&

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

bitwise ~ versus `Boolean !`

```
~0x01 = 0xfffffffe (bitwise)
!0x01 = 0x0 (Boolean)
!0xff = 0x0 (Boolean)
```

The act (operation) of *Masking*

a =	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
	<op>	<op>	<op>	<op>	<op>	<op>	<op>	<op>
b =	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
result =	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

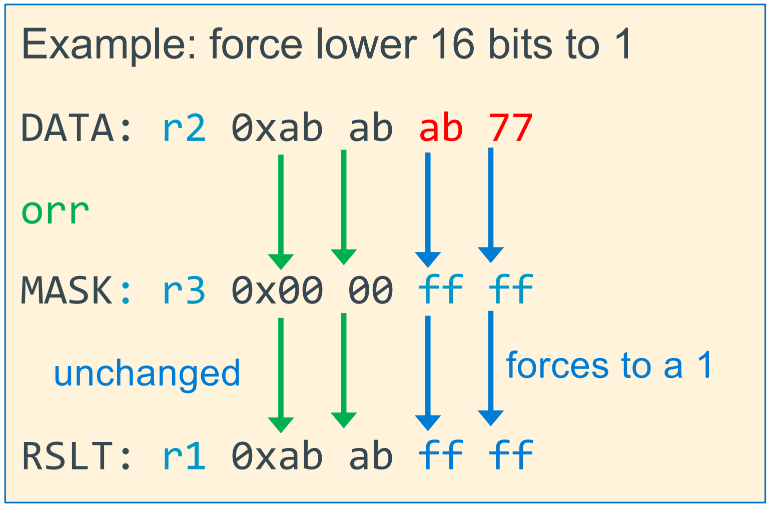
- Bit masks access/modify specific bits in memory
- Masking act of applying a mask to a value with a specific op:
- `orr:` 0 passes bit unchanged, 1 sets bit to 1 `(a = b | c; // in C)`
- `eor:` 0 passes bit unchanged, 1 inverts the bit `(a = b ^ c; // in C)`
- `and:` 0 clears the bit, 1 passes bit unchanged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

Mask on

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

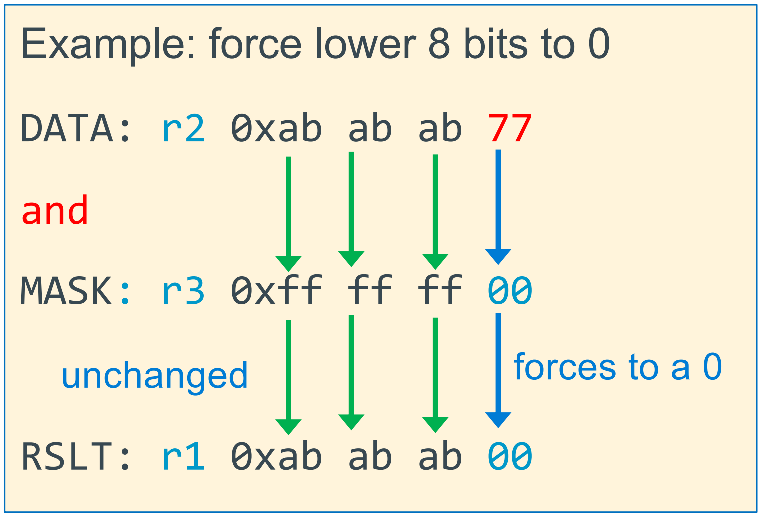
```
DATA: r2 0xab ab ab 77

orr   r1   r2, 0xff

r1 = r2 | 0xff; // in C

RSLT: r1 0xab ab ab ff
```

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
RSLT: r1 0xab ab ab 00
```

Example: force lower 8 bits to 0

```
DATA: r2 0xab ab ab 77

and r1  r2, 0xffffff00

r1 = r2 & 0xffffff00; // in C

RSLT: r1 0xab ab ab 00
```

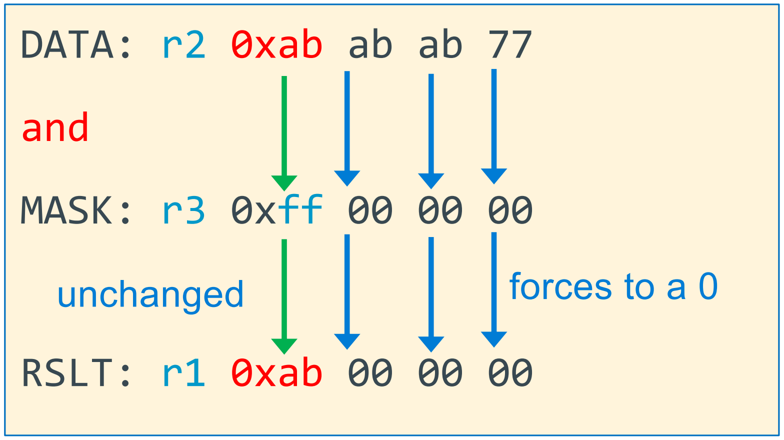
x

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
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
```

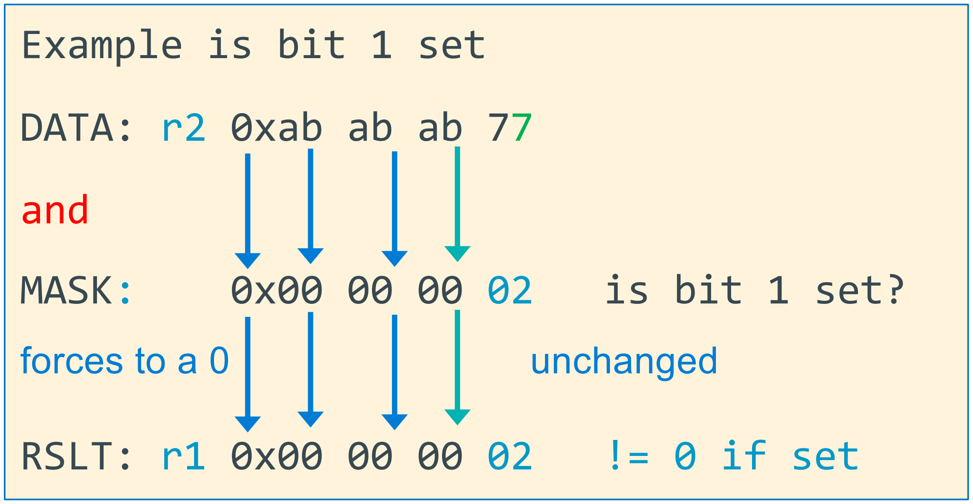
x

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:
```



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

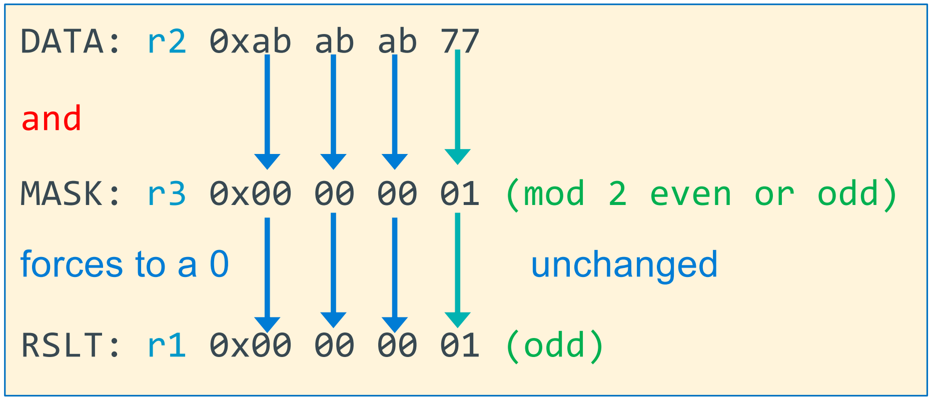
```
unsigned int r2;
// code
if ((r2 & 0x02) != 0) {
    // code for is set
}
```

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
}
```



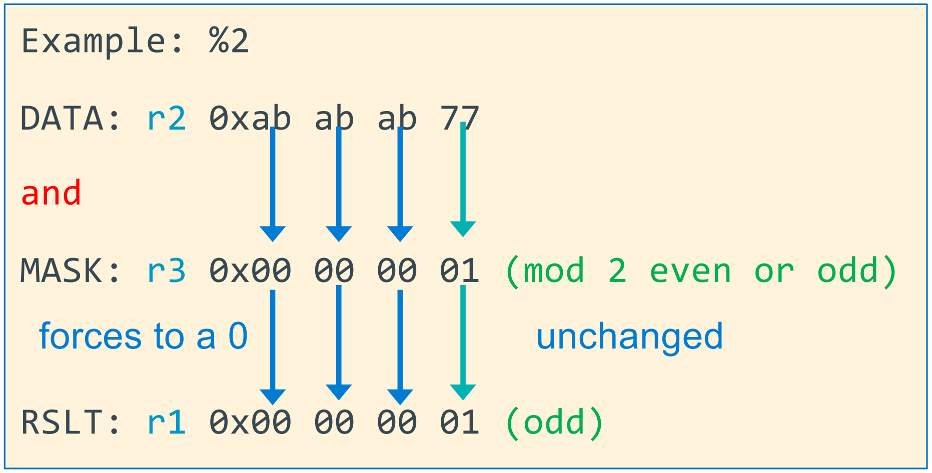
```
DATA: r2 0xab ab ab 77
and
MASK: r3 0x00 00 00 01 (mod 2 even or odd)
forces to a 0            unchanged
RSLT: r1 0x00 00 00 01 (odd)
```

x

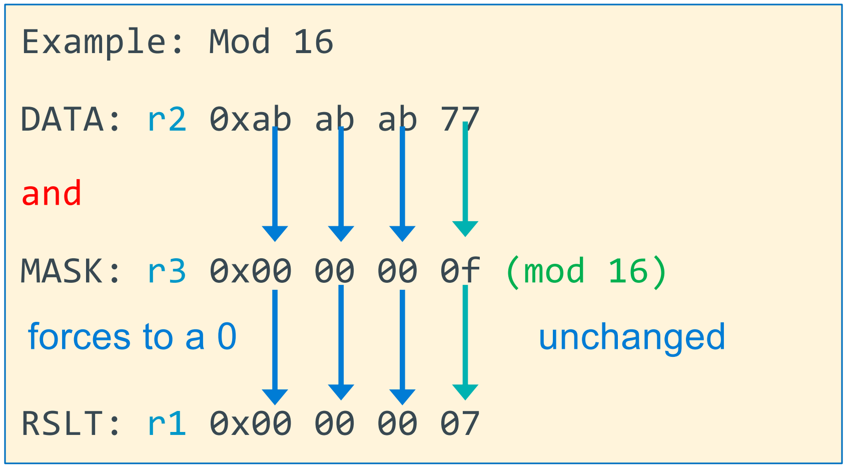
MOD %<power of 2>

remainder (mod): num % d where num ≥ 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 0x00 00 00 07
```

x

Flipping bits: bit toggle
Used in PA7/PA8

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

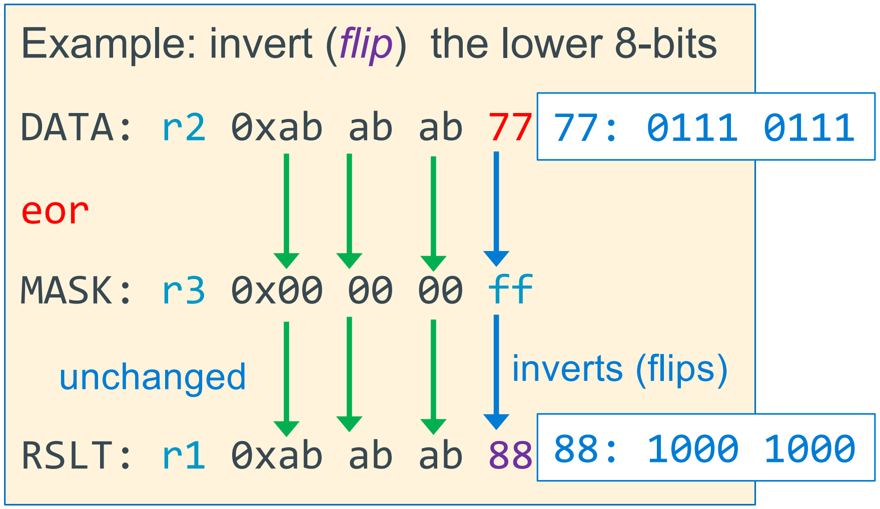
- 1 **will flip the bit**
- 0 to let a **bit through**

```
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

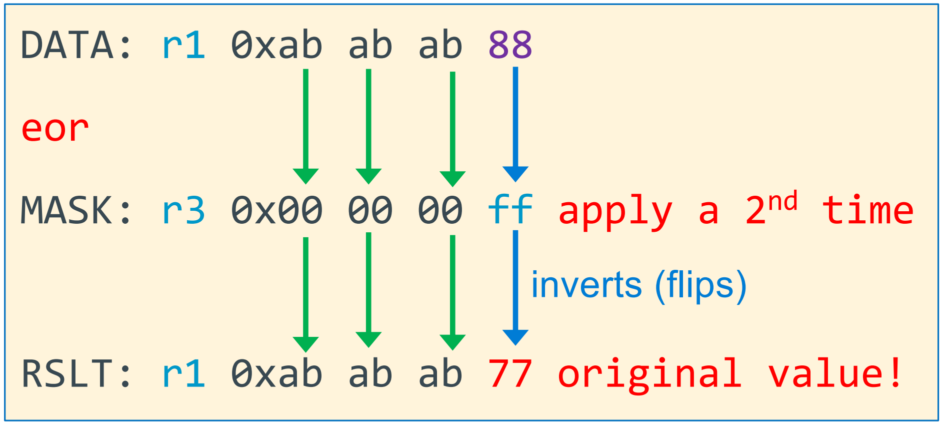
```
Example: flip the lower 8-bits

    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!
```

x

Unsigned Integers (positive numbers) with Fixed # of Bits

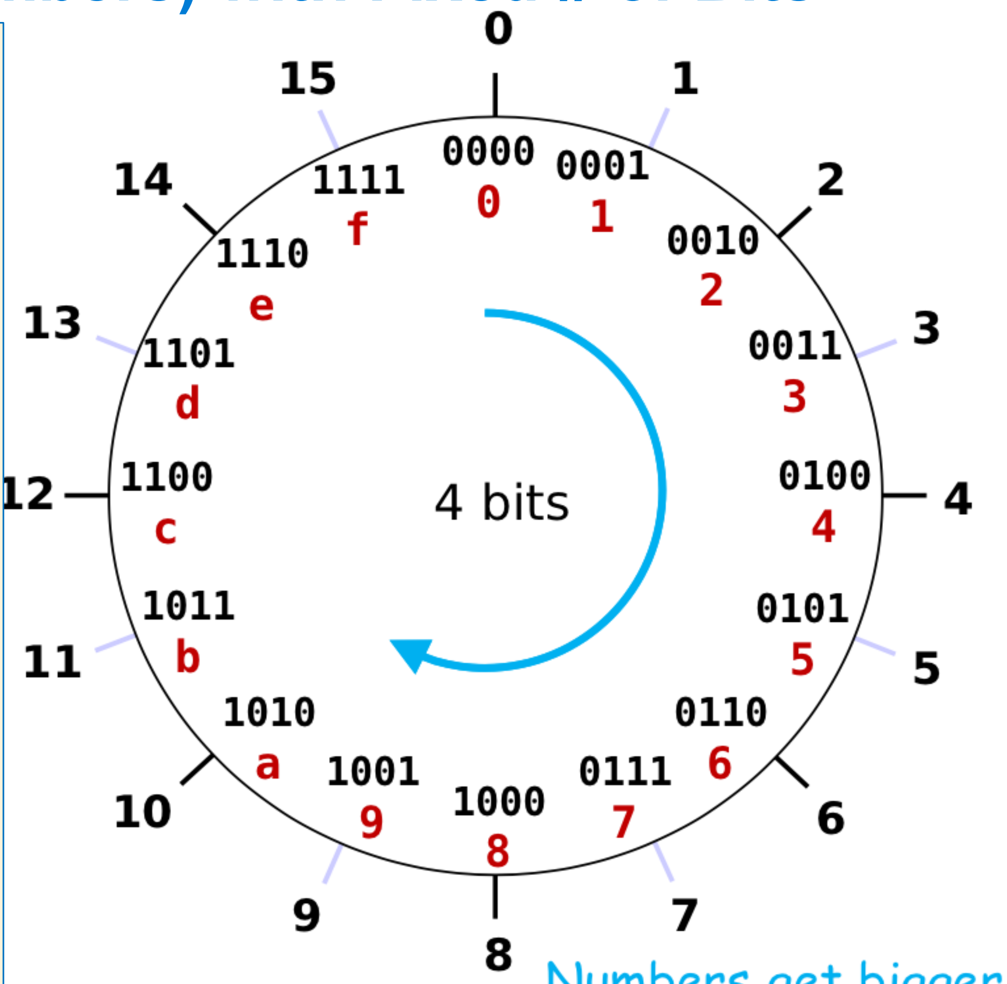
- 4 bits is 2^4 = ONLY 16 distinct values
- **Modular** (C operator: %) or clock math
  - Numbers start at 0 and “wrap around” after 15 and go back to 0
- Keep adding 1

  wraps (clockwise)

  0000 -> 0001 … -> 1111 -> 0000
- Keep subtracting 1

  wraps (counter-clockwise)

  1111 -> 1110 … -> 0000 -> 1111
- Addition and subtraction use normal "carry" and "borrow" rules, just operate in binary

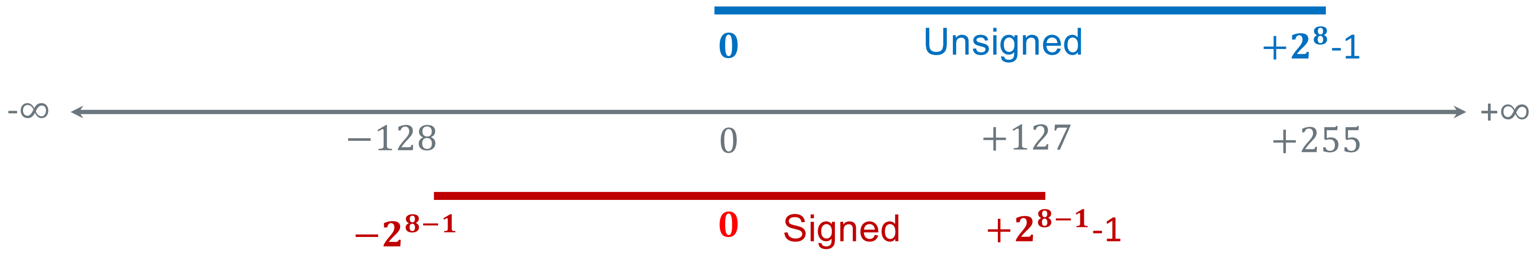


Numbers get bigger in this direction

x

Problem: How to Encode Both Positive and Negative Integers

- How do we represent the negative numbers within a fixed number of bits?
  - Allocate some bit patterns to negative and others to positive numbers (and zero)
- $\mathbf{2^n}$ distinct bit patterns to encode positive and negative values
- **Unsigned values:** $0 \ldots 2^n{-}1$ ← -1 comes from counting 0 as a "positive" number
- **Signed values:** $-2^{n-1} \ldots 2^{n-1}{-}1$ (dividing the range in ~ half including 0)
- **On a number line (below):** 8-bit integers – signed and unsigned (*e.g.*, **`char in C`**)

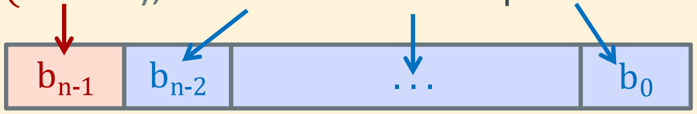


Same "width" (same number of encodings), just shifted in value

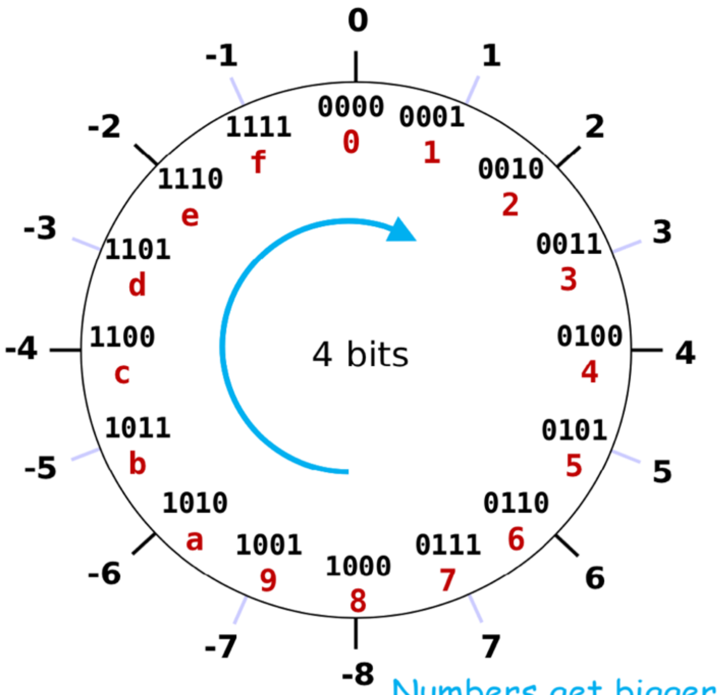
Two’s Complement: The MSB Has a *Negative Weight*

$$2's\ Comp = -b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_1 2^1 + b_0 2^0$$

b_{n-1} weight is (-2^{n-1}), all other bits: have positive weights $(+2^i)$



- 4-bit (w = 4) weight = -2^{4-1} = -2^3 = -8
  - 1010_2 **unsigned**:
    $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$ = **10**
  - 1010_2 **two’s complement**:
    $-1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$ = -8 + 2 = **–6**
  - -8 in **two’s complement:**
    1000_2 = -2^3 + 0 = -8
  - -1 in **two’s complement:**
    1111_2 = -2^3 + (2^3 – 1) = -8 + 7 = **-1**



Numbers get bigger in this direction

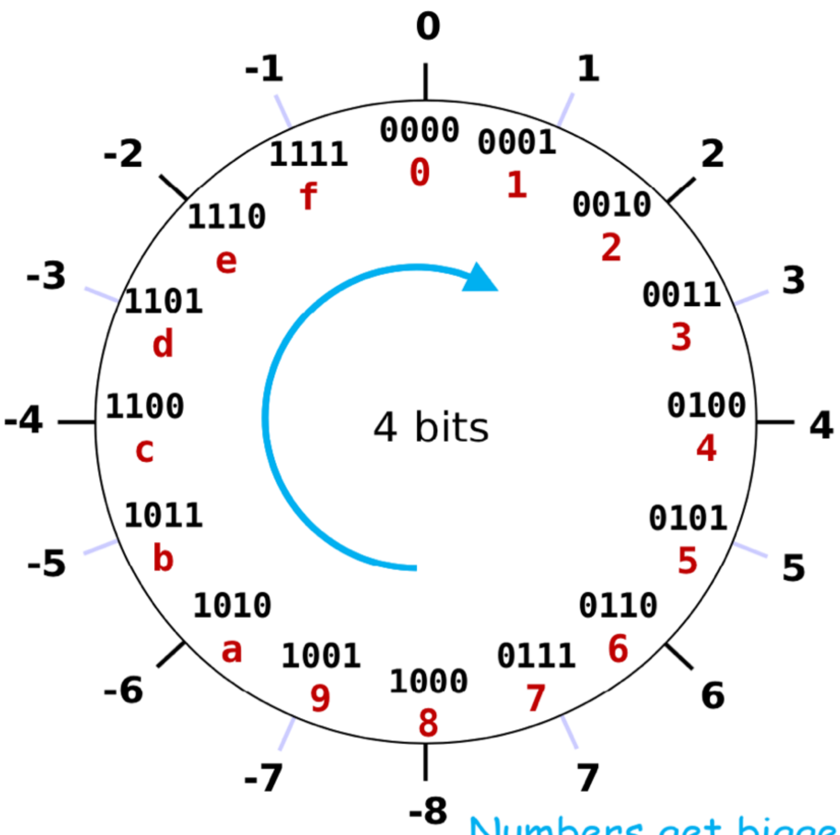
x

2's Complement Signed Integer Method

- Positive numbers encoded same as unsigned numbers
- All negative values have a one in the leftmost bit
- All positive values have a zero in the leftmost bit
  - This implies that 0 is a positive value
- Only one zero
- **For n bits, Number range is $-(2^{n-1})$ to $+(2^{n-1}$ - 1)**
  - Negative values “go 1 further” than the positive values
- Example: the range for 8 bits:

  -128, -127, .. 0, .. 126, **+127**
- Example the range for 32 bits:

  -2147483648 .. 0, .. **+2147483647**
- *Arithmetic is the same as with unsigned binary!*



Numbers get bigger
in this direction

x

Sign Extension (how type promotion works)

- Sometimes you need to work with integers encoded with different number of bits

  8 bits (char) -> (16 bits) `short` -> (32 bits) `int`

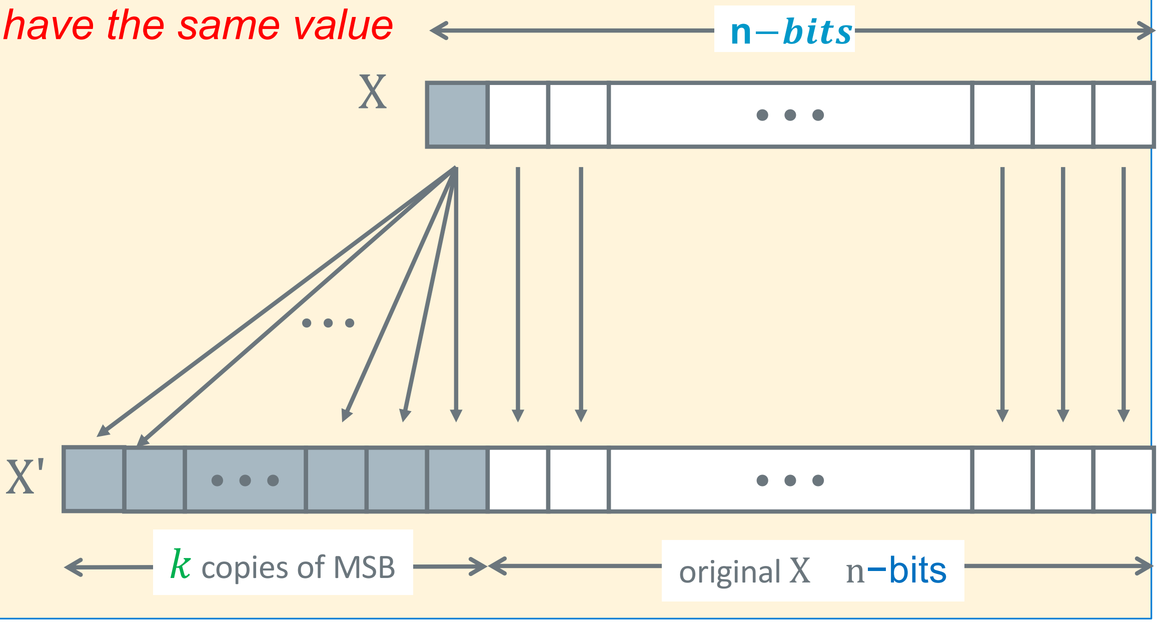
- **Sign extension increases the number of bits: n-bit** wide signed integer X, ***EXPANDS*** to a ***wider*** n−bit + k-bit signed integer X' where *both have the same value*

Unsigned

- Just add leading zeroes to the left side

Two's Complement Signed:

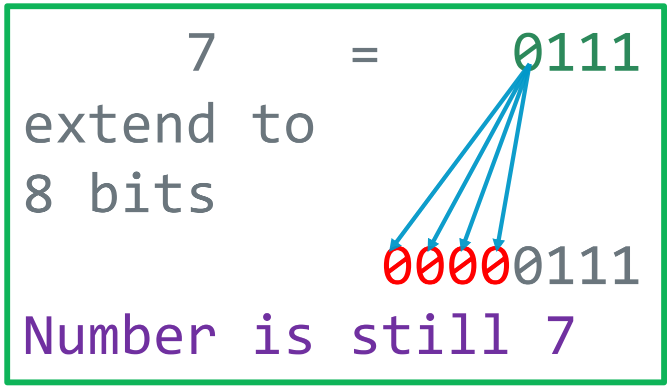
- If positive, add leading zeroes on the left
  - Observe: Positive stay positive
- If negative, add leading ones on the left
  - Observe: Negative stays negative



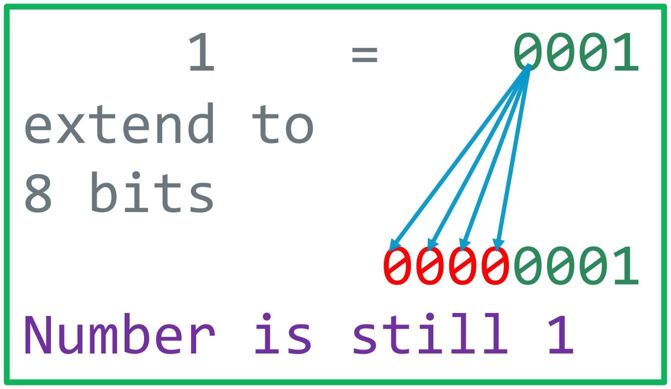
x

Example: Two’s Complement Sign or bit Extension - 1

- Adding 0's in front of a positive numbers does not change its value



```
    7     =     0111
extend to
8 bits
            00000111
Number is still 7
```

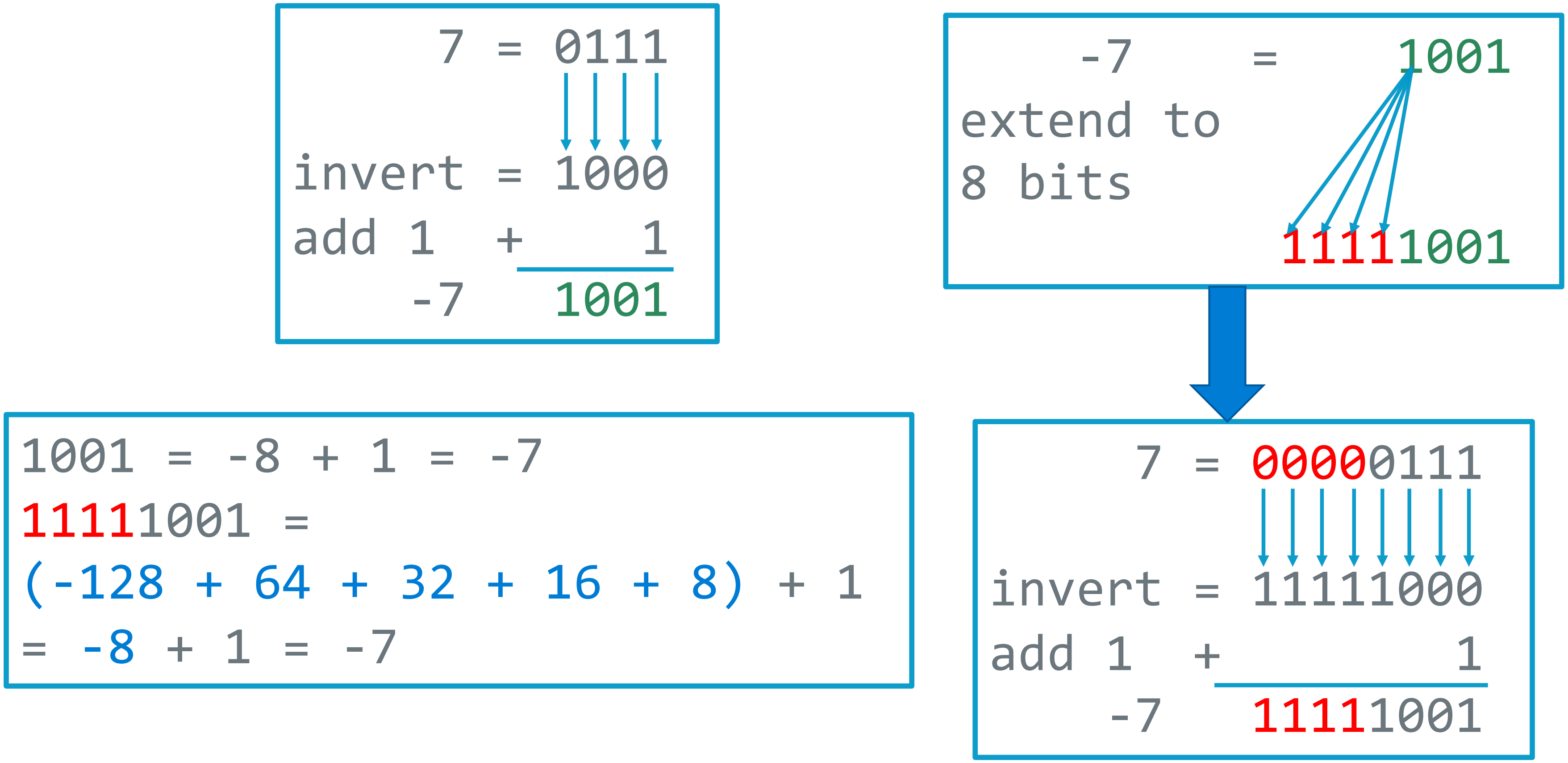


```
    1     =     0001
extend to
8 bits
            00000001
Number is still 1
```

x

Example: Two’s Complement Sign or bit Extension -2

- Adding 1's if front of a negative number does not change its value



```
     7 = 0111
invert = 1000
add 1  +    1
    -7   1001
```

```
   -7    =    1001
extend to
8 bits
            11111001
```

```
     7 = 00000111
invert = 11111000
add 1  +        1
    -7   11111001
```

```
1001 = -8 + 1 = -7
11111001 =
(-128 + 64 + 32 + 16 + 8) + 1
= -8 + 1 = -7
```

Sign Extension in C: Type casts

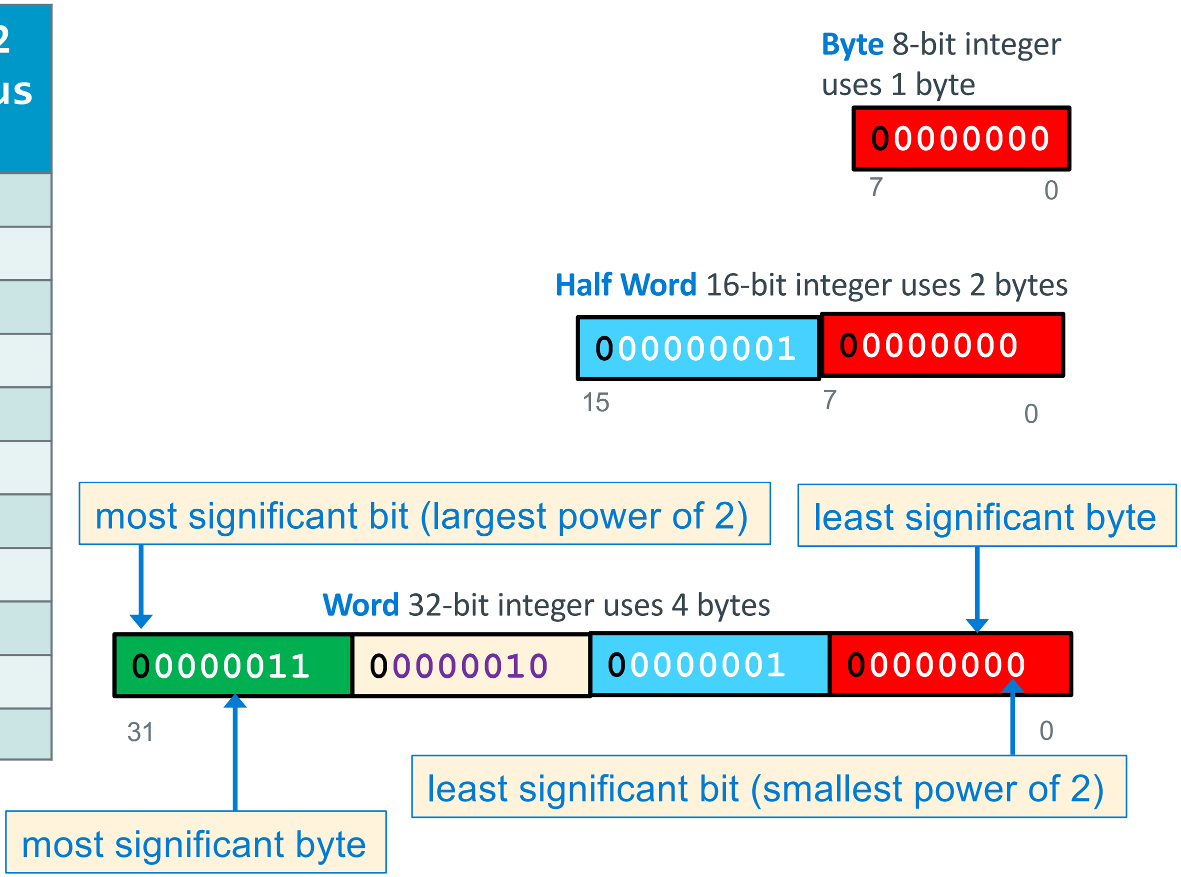
- Convert from smaller to larger integral data types
- C and Java automatically performs sign extension
- Example (on pi-cluster with 32-bit int)

```c
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    signed char c = -1;
    signed int i = c;
    unsigned char d = 1;
    unsigned int j = d;
    printf("c decimal = %hd\n", c);
    printf("c = 0x%hhx\n", c);
    printf("i decimal = %d\n", i);
    printf("i = 0x%x\n", i);
    printf("\nd decimal = %hd\n", d);
    printf("d = 0x%hhx\n", d);
    printf("j decimal = %d\n", j);
    printf("j = 0x%x\n", j);
    return EXIT_SUCCESS;
}
```

```
%./a.out
c decimal = -1
c = 0xff
i decimal = -1
i = 0xffffffff

d decimal = 1
d = 0x1
j decimal = 1
j = 0x1
```

Different Type of Numbers each have a Fixed # of Bits
Spanning one or more contiguous bytes of memory

C Data Type	AArch-32 contiguous Bytes
char (arm unsigned)	1
short int	2
unsigned short int	2
int	4
unsigned int	4
long int	4
long long int	8
float	4
double	8
long double	8
pointer *	4



Byte 8-bit integer uses 1 byte

00000000

7 0

Half Word 16-bit integer uses 2 bytes

00000001 00000000

15 7 0

most significant bit (largest power of 2)

least significant byte

Word 32-bit integer uses 4 bytes

00000011 00000010 00000001 00000000

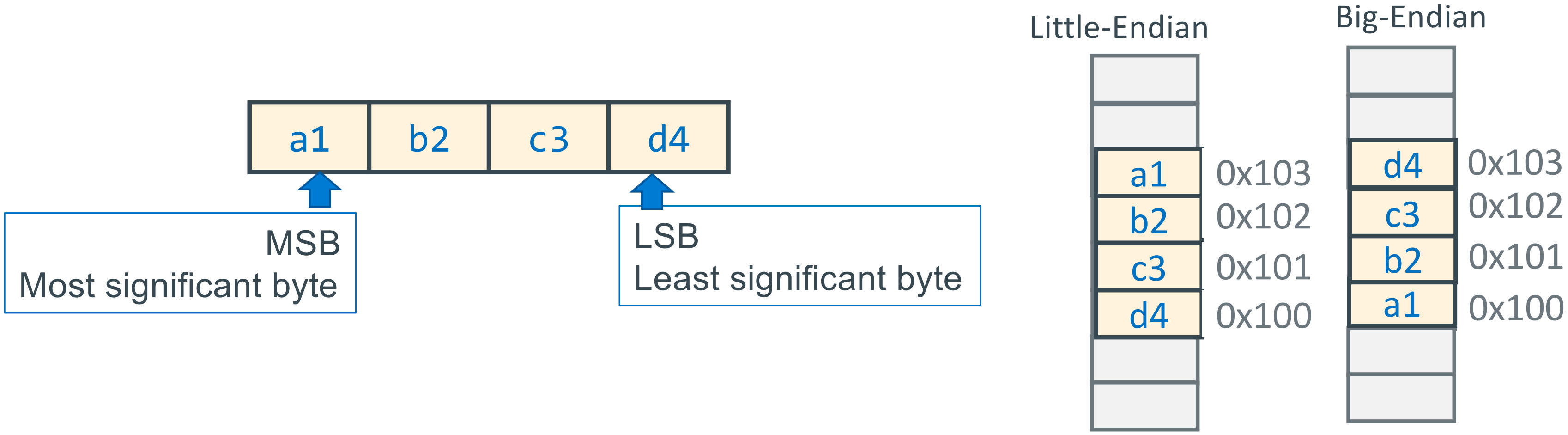
31 0

least significant bit (smallest power of 2)

most significant byte

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



a1	b2	c3	d4

MSB
Most significant byte

LSB
Least significant byte

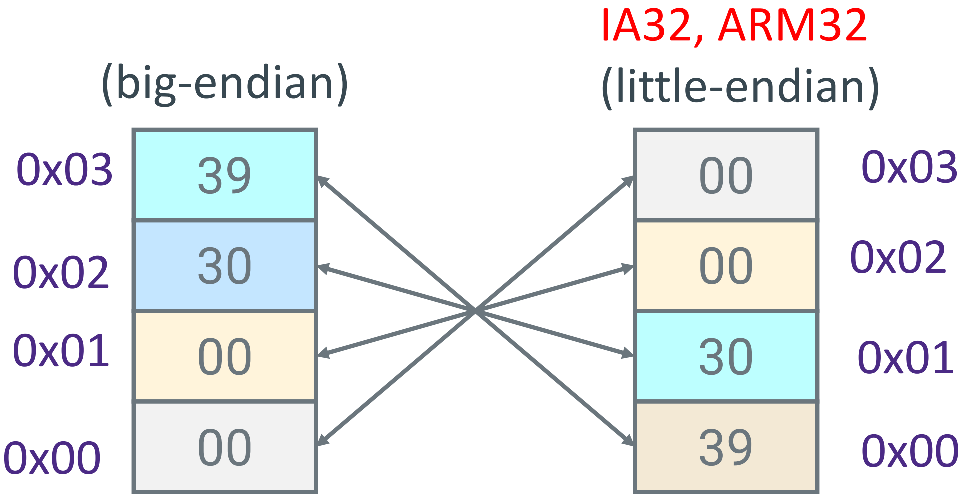
Little-Endian		Big-Endian	
a1	0x103	d4	0x103
b2	0x102	c3	0x102
c3	0x101	b2	0x101
d4	0x100	a1	0x100

x

Byte Ordering Example

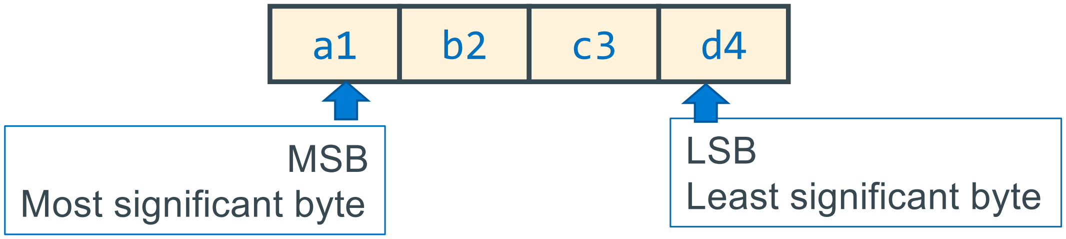
```
Decimal:   12345
Binary:     0011  0000  0011  1001
Hex:          3     0     3     9
```

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



x

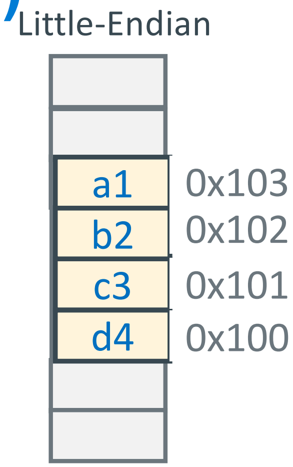
Using pointers to examine byte order (on pi-cluster)



a1 b2 c3 d4

MSB
Most significant byte

LSB
Least significant byte

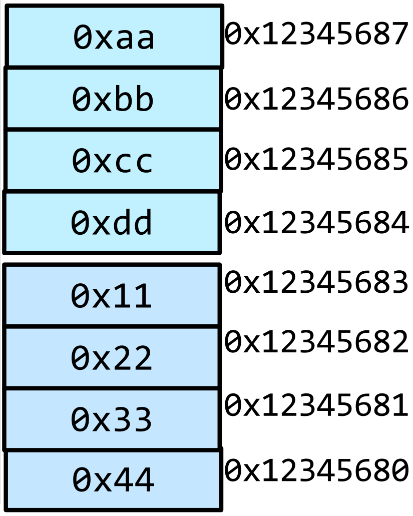


Little-Endian

a1	0x103
b2	0x102
c3	0x101
d4	0x100

```
#include <stdio.h>
#include <stdlib.h>
#define SZ 2
int main()
{
    unsigned int foo[SZ] = {0x11223344, 0xaabbccdd};
    unsigned char *chptr = (unsigned char *)foo;

    // print from MSB to LSB – high to low memory)
    for (int i = sizeof(foo)-1; i >= 0; i--)
        printf("byte %d: %x\n", i, *(chptr + i));

    return EXIT_SUCCESS;
}
```

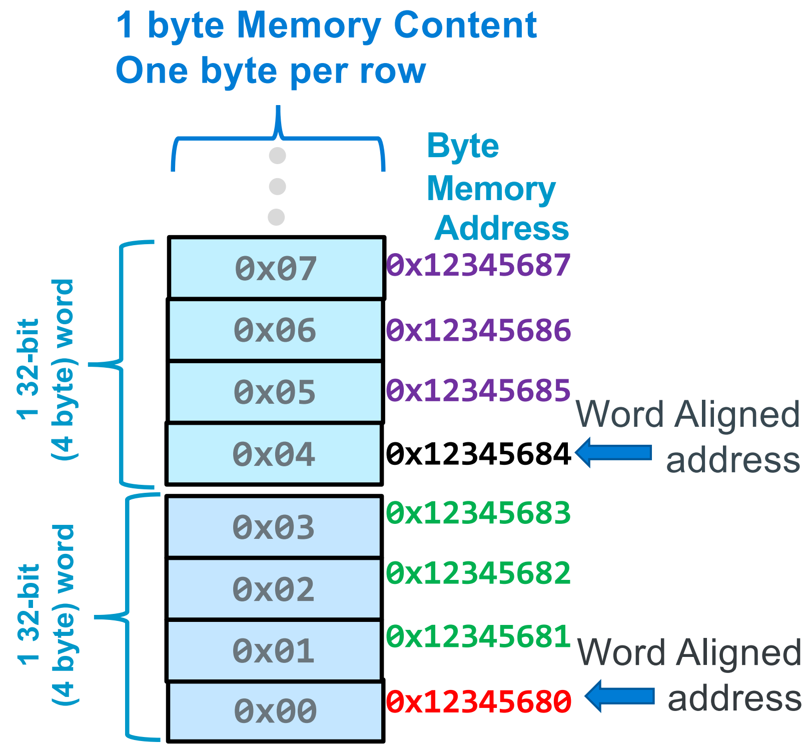
```
$ ./a.out
byte 7: aa
byte 6: bb
byte 5: cc
byte 4: dd
byte 3: 11
byte 2: 22
byte 1: 33
byte 0: 44
```



0xaa	0x12345687
0xbb	0x12345686
0xcc	0x12345685
0xdd	0x12345684
0x11	0x12345683
0x22	0x12345682
0x33	0x12345681
0x44	0x12345680

x

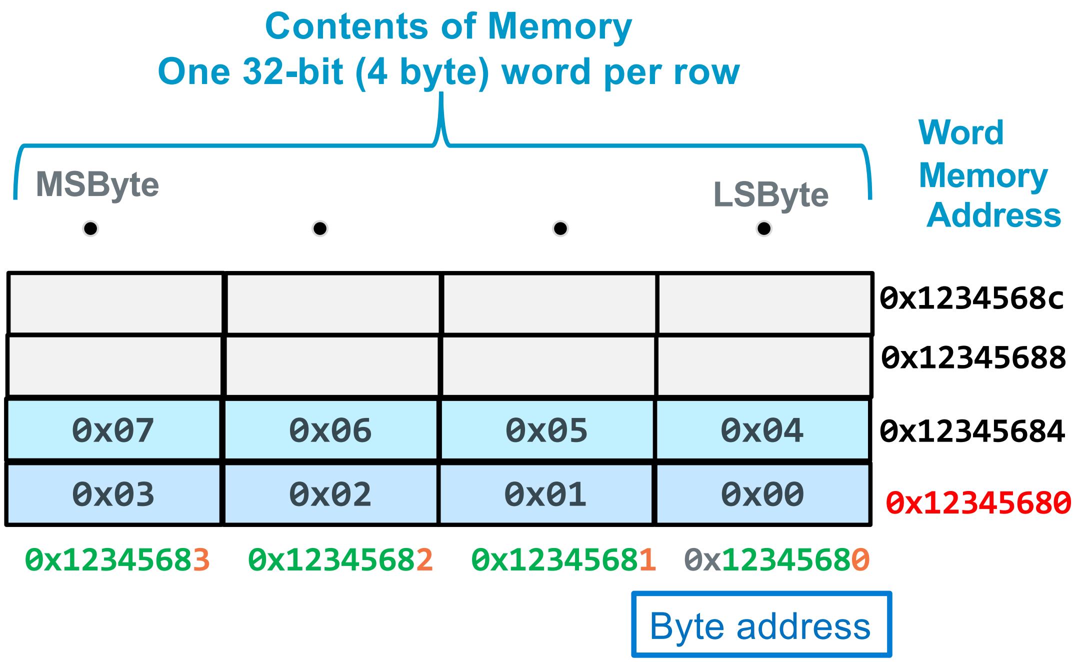
Byte Addressable Memory Shown as 32-bit words



1 byte Memory Content
One byte per row

Byte
Memory
Address

0x07	0x12345687
0x06	0x12345686
0x05	0x12345685
0x04	0x12345684
0x03	0x12345683
0x02	0x12345682
0x01	0x12345681
0x00	0x12345680



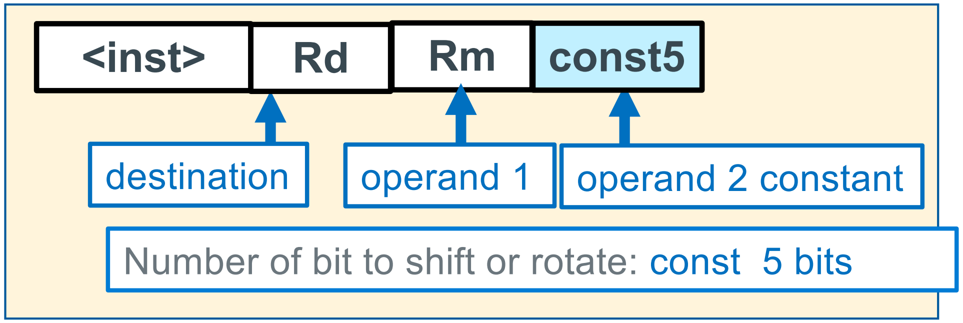
Contents of Memory
One 32-bit (4 byte) word per row

MSByte			LSByte	Word Memory Address
				0x1234568c
				0x12345688
0x07	0x06	0x05	0x04	0x12345684
0x03	0x02	0x01	0x00	0x12345680
0x12345683	0x12345682	0x12345681	0x12345680	

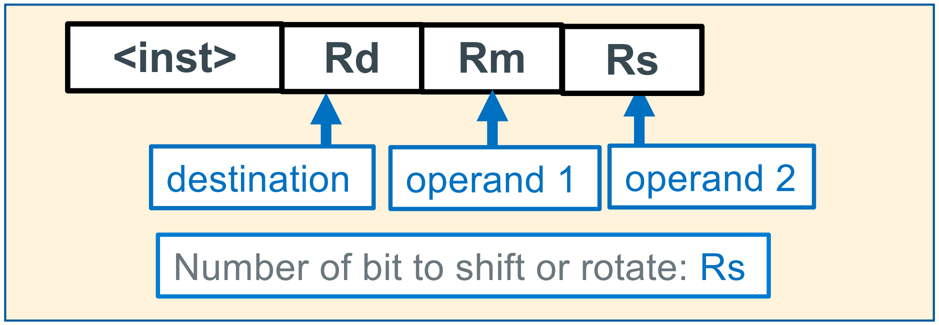
Observation
32-bit aligned addresses
rightmost 2 bits of the address are always 0

x

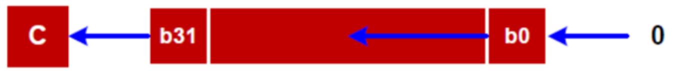
Shift and Rotate Instructions

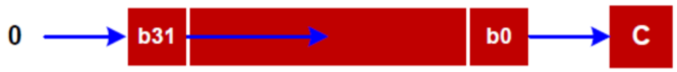


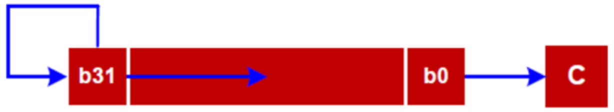
Number of bit to shift or rotate: const 5 bits

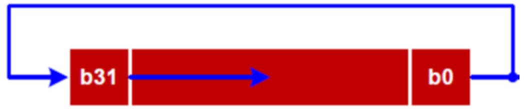


Number of bit to shift or rotate: Rs

Instruction	Syntax	Operation	Notes	Diagram
Logical Shift Left int x; or unsigned int x x << n;	lsl R_d, R_m, *const5* lsl R_d, R_m, R_s	$R_d \leftarrow R_m$ << *const5* $R_d \leftarrow R_m$ << R_s	Zero fills shift: 0 - 31	

Logical Shift Right
unsigned int x;
x >> n;
	lsr R_d, R_m, *const5*
	lsr R_d, R_m, R_s
		$R_d \leftarrow R_m$ >> *const5*
		$R_d \leftarrow R_m$ >> R_s
			Zero fills
			shift: 1 – 32
				

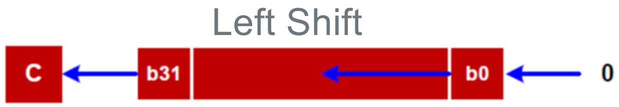
Arithmetic Shift Right
int x;
x >> n;
	asr R_d, R_m, *const5*
	asr R_d, R_m, R_s
		$R_d \leftarrow R_m$ >> *const5*
		$R_d \leftarrow R_m$ >> R_s
			Sign extends
			shift: 1 - 32
				

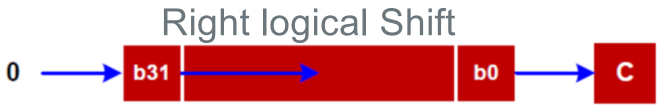
Rotate Right
unsigned int x;
x = (x>>n)|(x<<(32-n));
	ror R_d, R_m, *const5*
	ror R_d, R_m, R_s
		$R_d \leftarrow R_m$ ror *const5*
		$R_d \leftarrow R_m$ ror R_s
			right rotate
			rot: 0 - 31
				

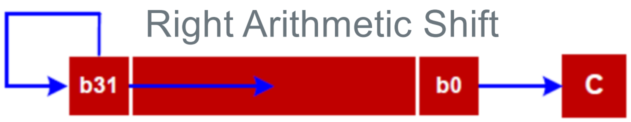
x

Shift Operations in C

- n is number of bits to shift a variable x of width w bits
- Shifts by `n < 0` or `n ≥ w` are *undefined*
- Left shift (`x << N`) – **Multiplies by 2^N**
  - Shift N bits left, Fill with `0`s on right
- **In C:** behavior of **>>** is determined by compiler
  - gcc: it depends on data type of `x` (signed/unsigned)
- Right shift (`x >> 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 **>>>** and arithmetic shift is >>



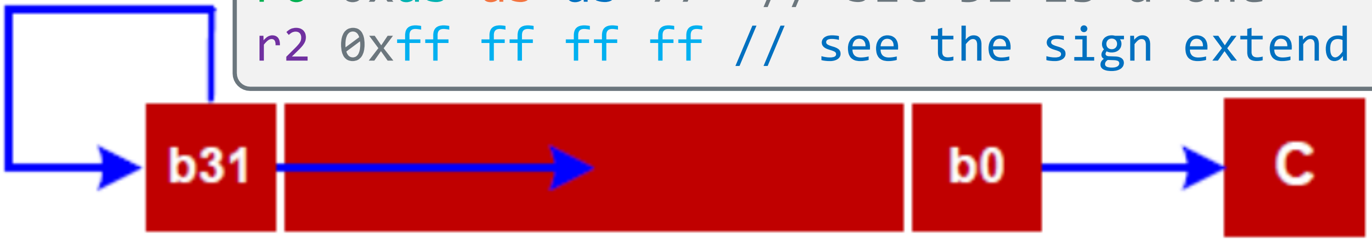




x

Arithmetic Shift Right Example: Testing Sign

```
asr r2, r0, 31

r0 0xab ab ab 77  // bit 31 is a one
r2 0xff ff ff ff // see the sign extend
```



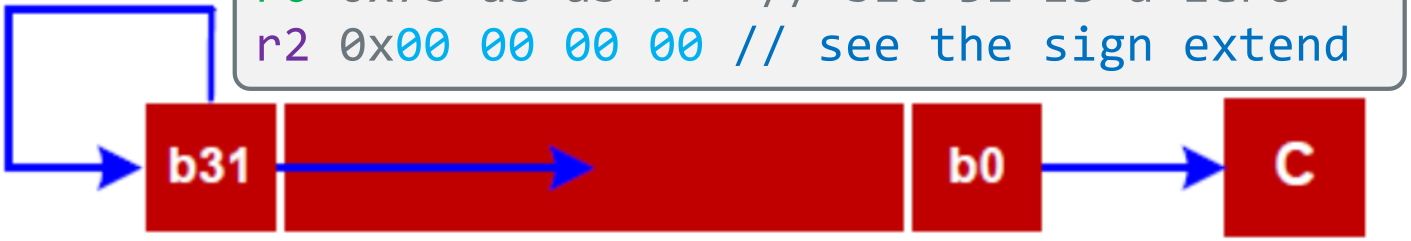
```
int i;
//code
if ((i>>31) == -1) {
  // code neg #
}
```

Test for sign
-1 if r0 negative

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

Arithmetic Shift Right Example: Testing SIgn

```
asr r2, r0, 31

r0 0x7b ab ab 77  // bit 31 is a zero
r2 0x00 00 00 00 // see the sign extend
```

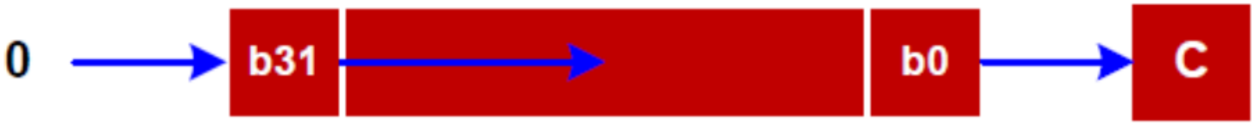


Test for sign
0 if r0 positive

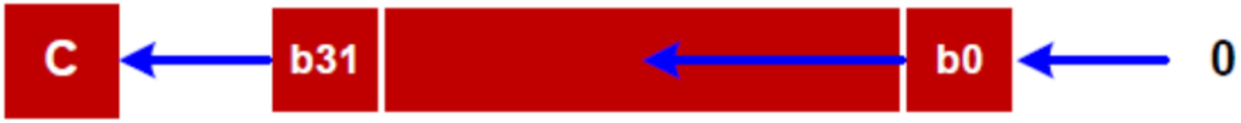
```
int i;
//code
if ((i>>31) == 0) {
  // code pos #
}
```

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

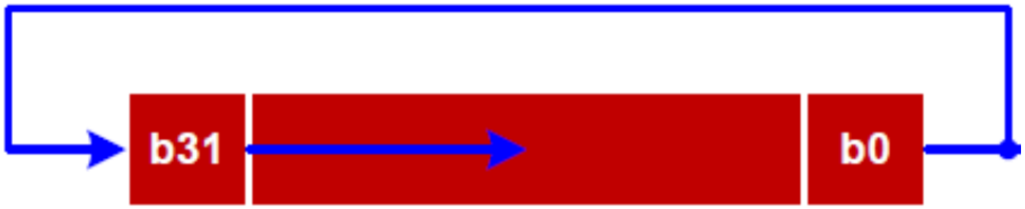
Logical Shift & Rotate Operations



```
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
```



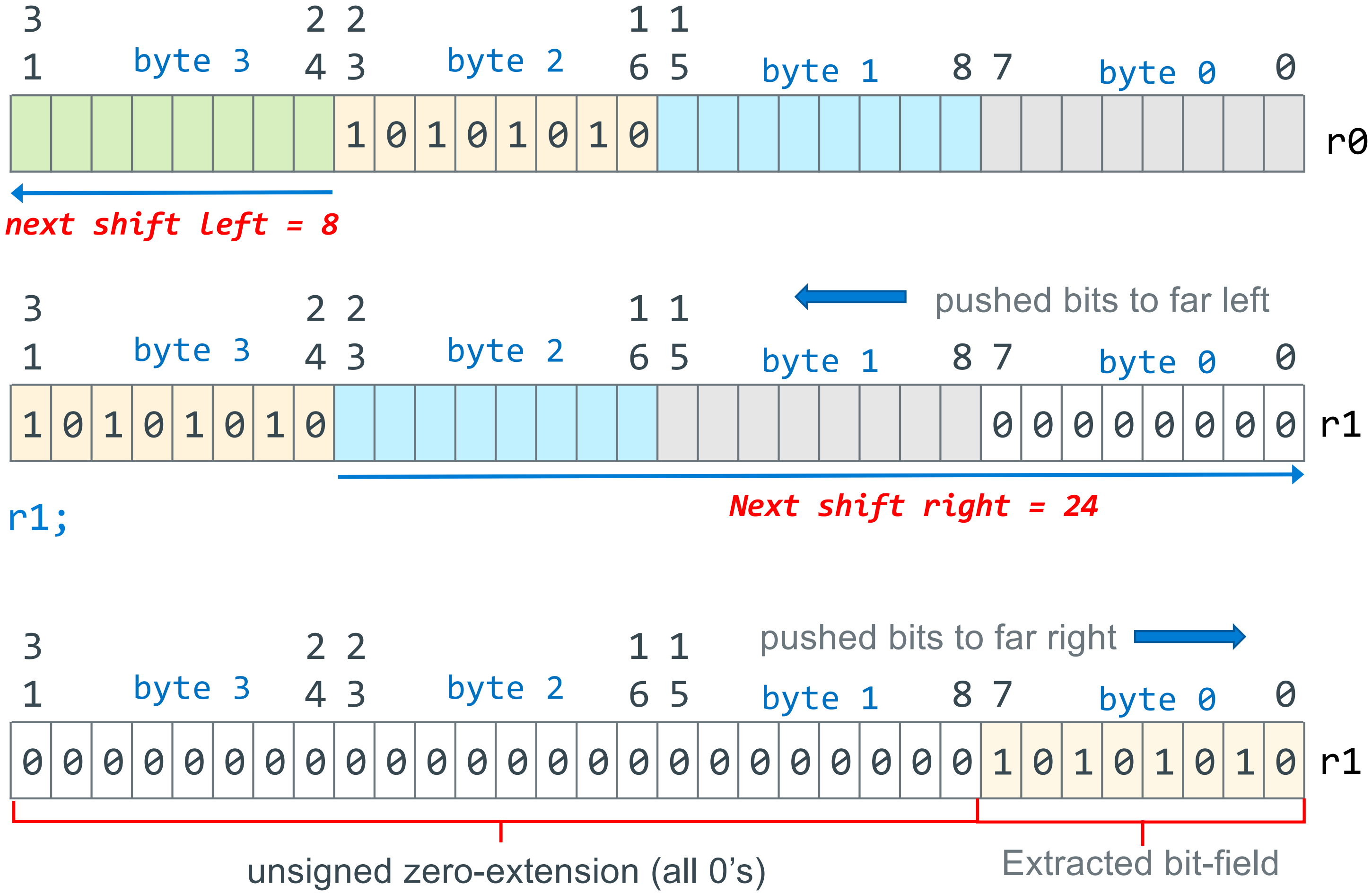
```
ror r2, r0, 8

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

x

Extracting/Isolating Unsigned Bitfields

Hint: Useful for PA7

- Move byte 2 in r0 to byte 0 in r1



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

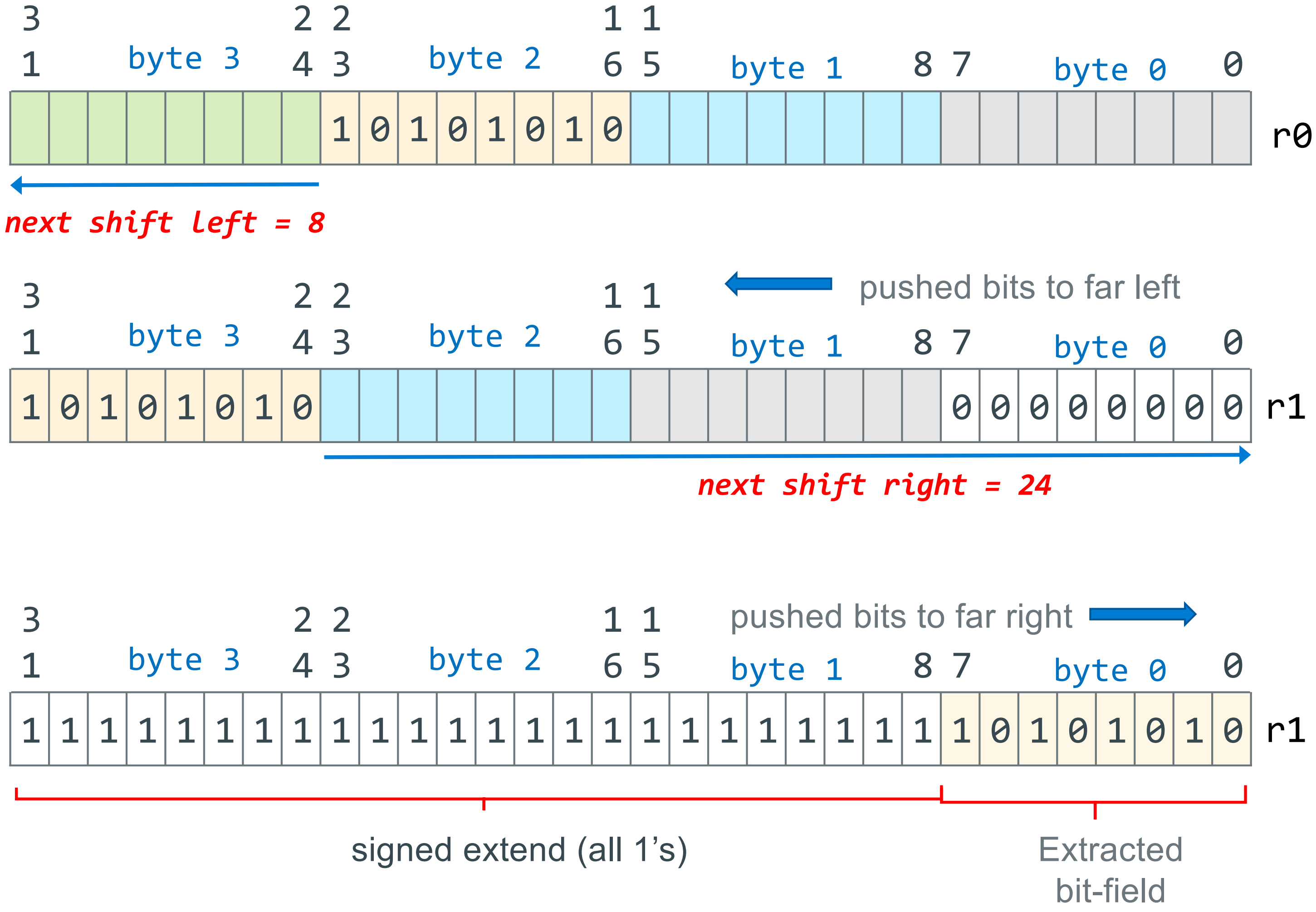
```
unsigned int r0,r1;
r1 = r0 << 8;
```

```
lsr   r1, r1, 24
r1 = r1 >> 24;
```

x

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

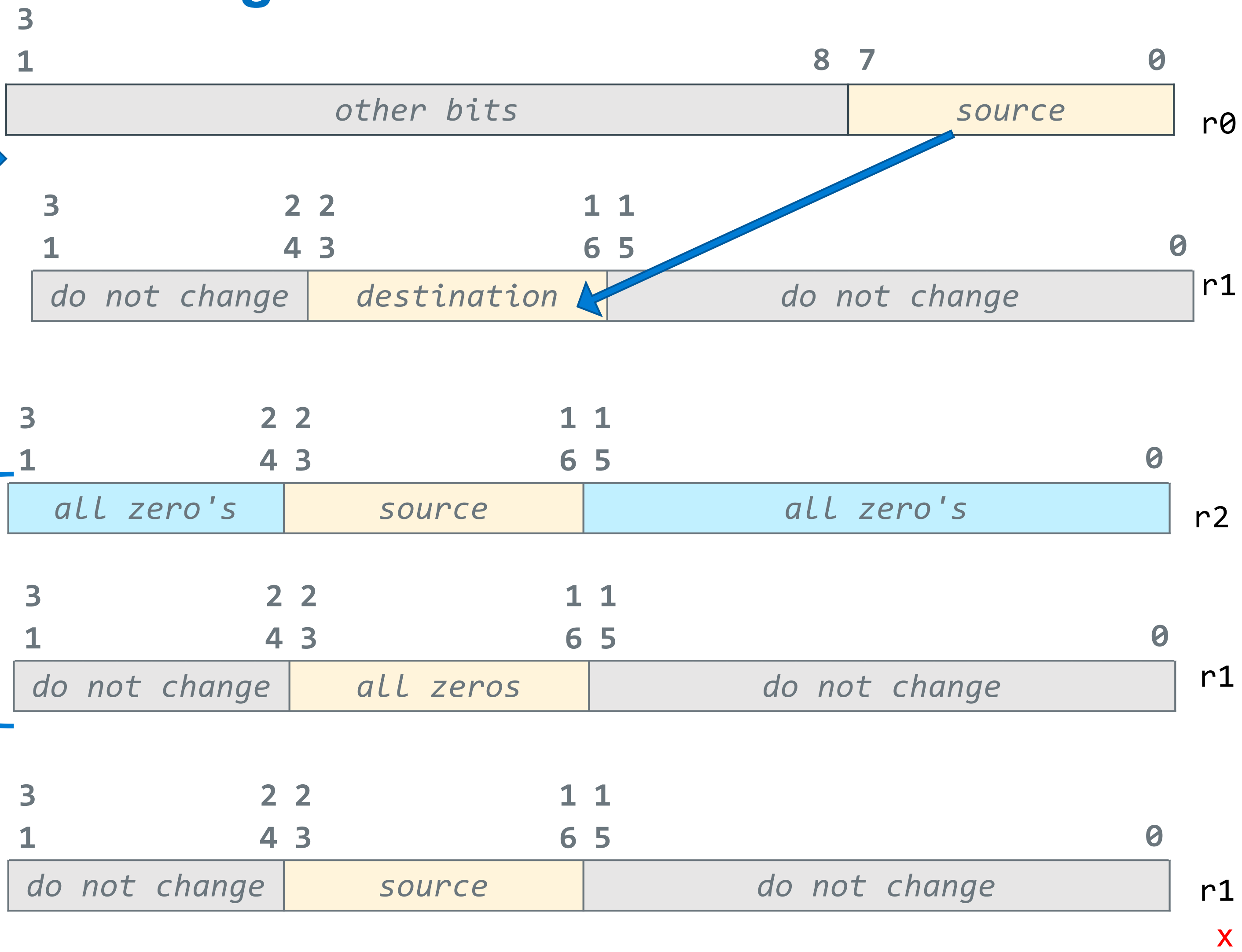
Task: Insert source into destination

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

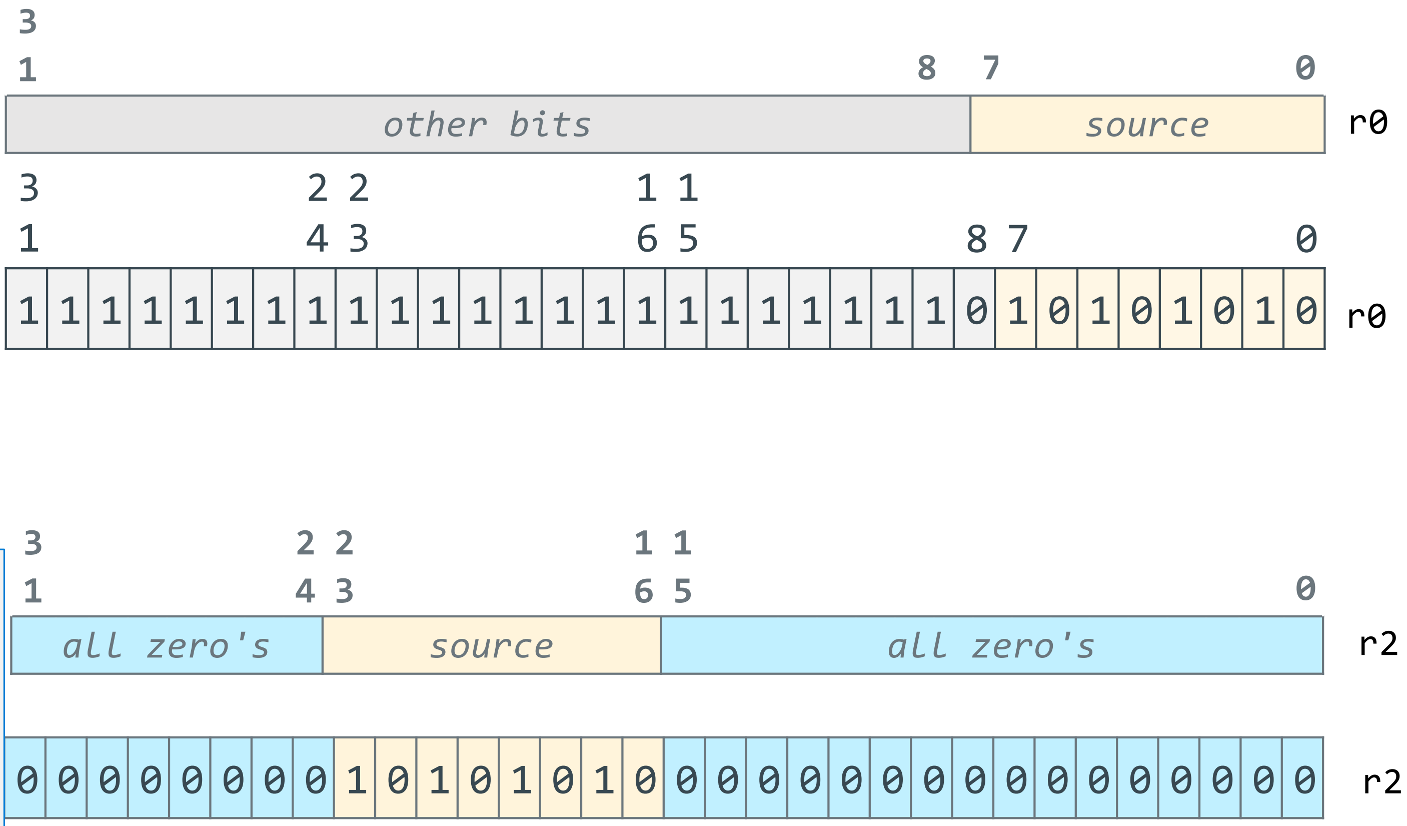
Approach
(1) isolate source field
(2) clear destination field
(3) Bitwise or together

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

results in



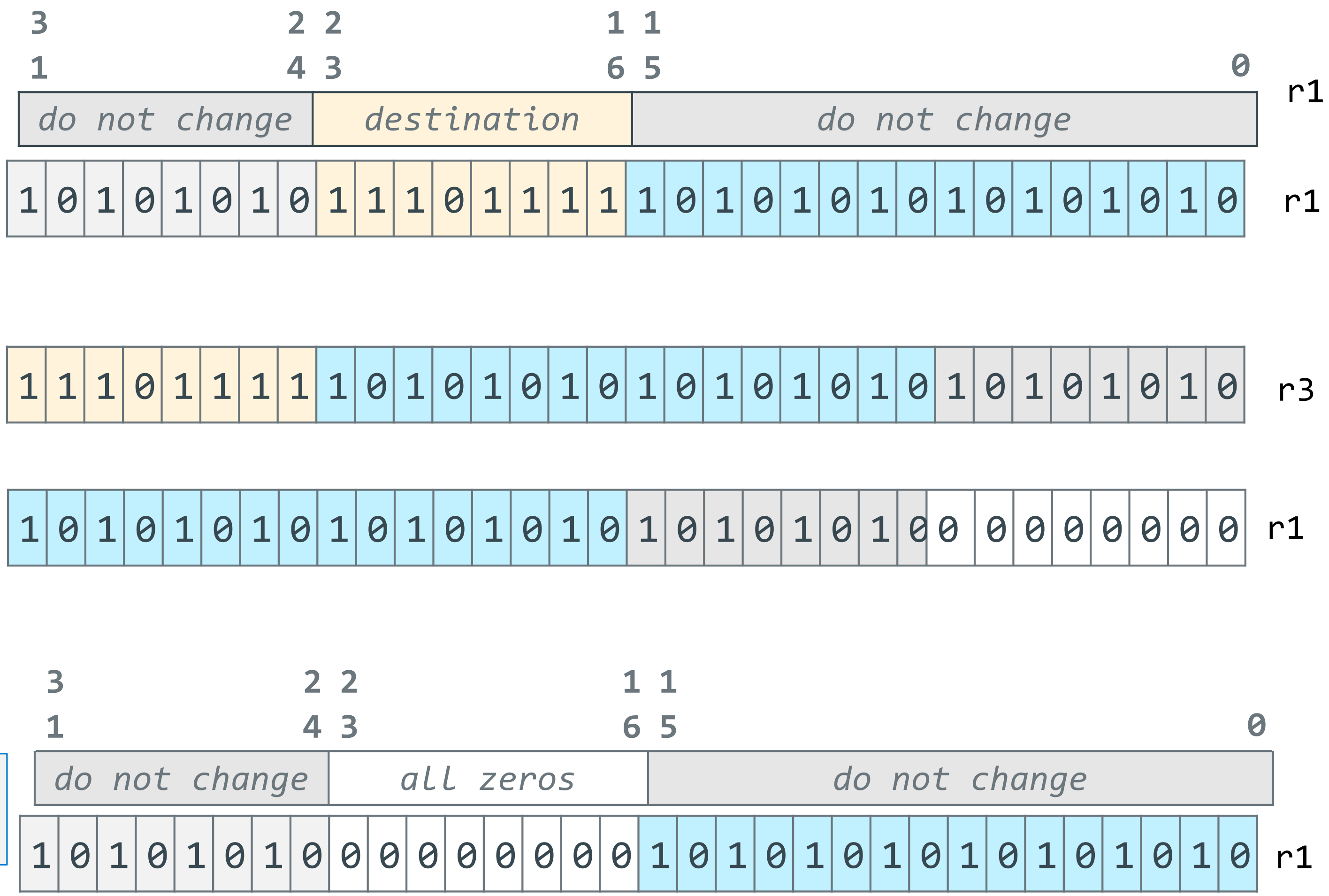
Inserting Bitfields – Isolating the Source Field



```
isolate source field

lsl    r2, r0, 24
lsr    r2, r2, 8

r2 = r0 << 24;
r2 = r2 >> 8;
```

x

Inserting Bitfields – Clearing the Destination Field



```
1 0 1 0 1 0 1 0 1 1 1 0 1 1 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0   r1
```

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

```
1 1 1 0 1 1 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0   r3
```

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

```
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 00 0 0 0 0 0 0 0   r1
```

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

```
1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0   r1
```

x

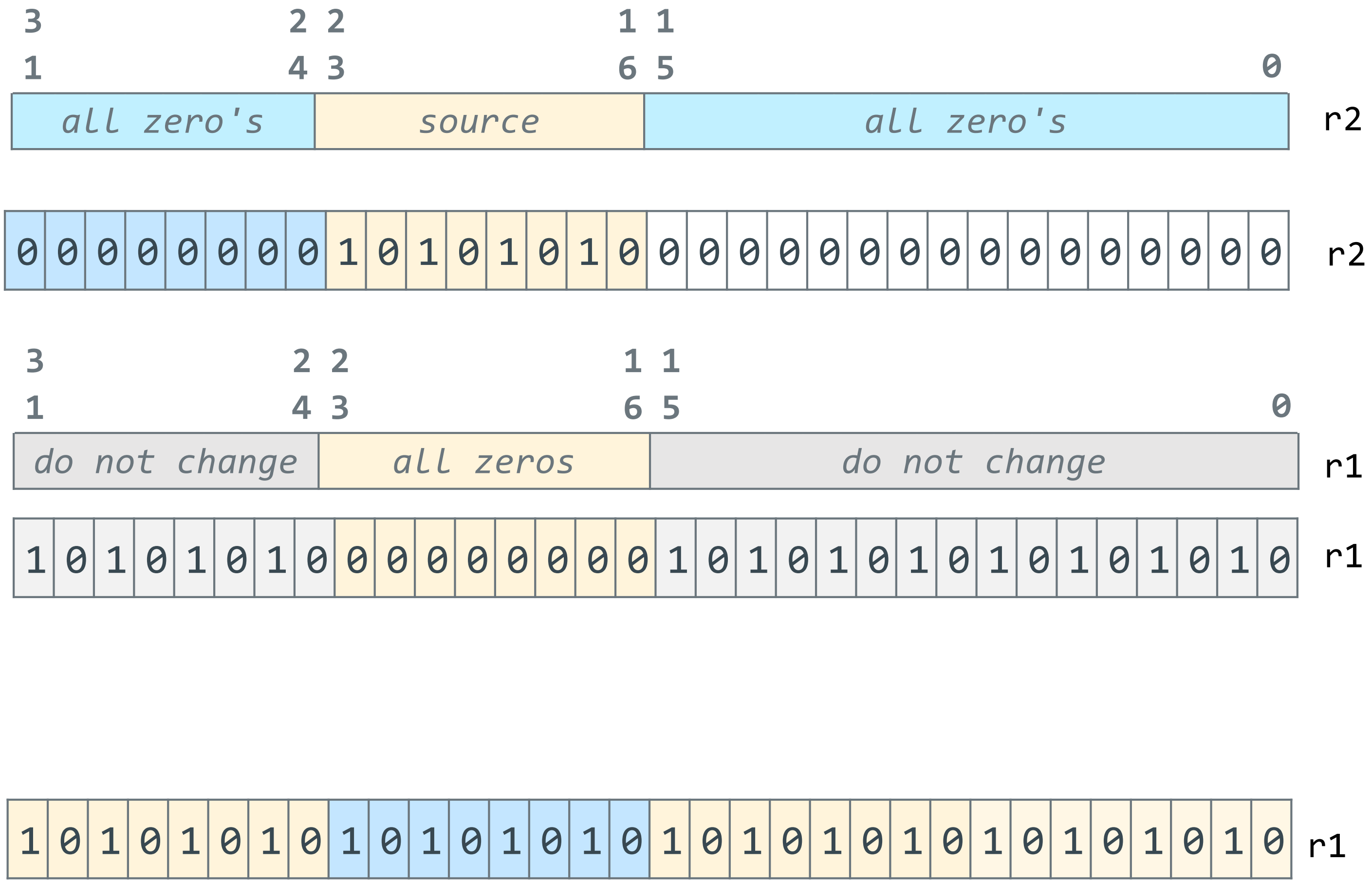
Inserting Bitfields – Combining Isolated Source and Cleared Destination

isolated source

field cleared in destination

inserted field

```
orr    r1, r1, r0
r1 = r1 | r0;
```



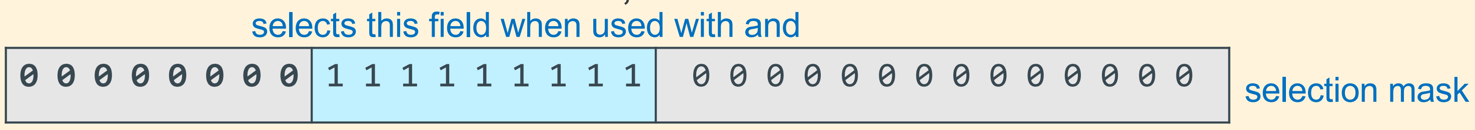
Example: Swapping bits7,6 with bits 1,0

																									7	6	5	4	3	2	1	0	
`source`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	r0
`and   r1, r0, 0x3` `(optional bits 31-8 == 0)`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	r1
`lsl   r1, r1, 6`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	r1
`lsr   r1, r0, 6`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	r2
`and   r0, r0, 0xc3`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	r0
`orr   r0, r0, r2`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	r0
`orr   r0, r0, r1`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	1	r0

x

Masking Summary

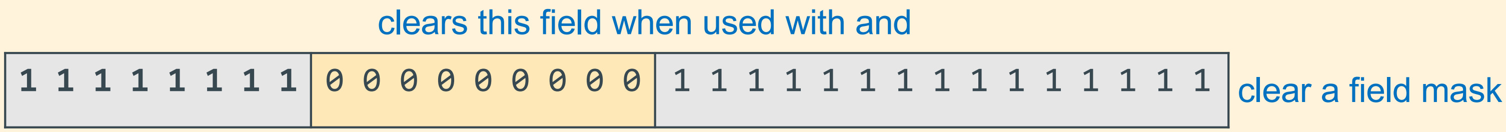
Select a field: Use **`and`** with a mask of one's surrounded by zero's to select the bits that have a 1 in the mask, all other bits will be set to zero


selects this field when used with and

`0 0 0 0 0 0 0 0` | `1 1 1 1 1 1 1 1 1` | `0 0 0 0 0 0 0 0 0 0 0 0 0 0`

selection mask

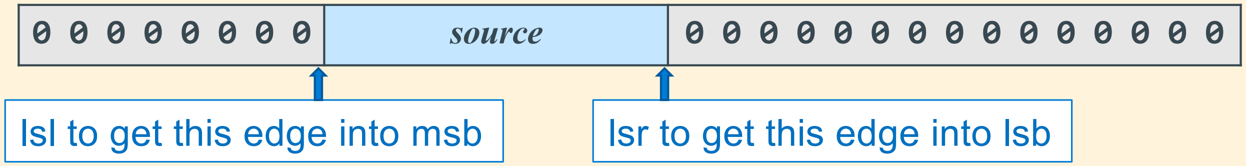
Clear a field: Use **`and`** with a mask of zero's surrounded by one's to select the bits that have a 1 in the mask, all other bits will be set to zero


clears this field when used with and

`1 1 1 1 1 1 1 1` | `0 0 0 0 0 0 0 0 0` | `1 1 1 1 1 1 1 1 1 1 1 1 1 1 1`

clear a field mask

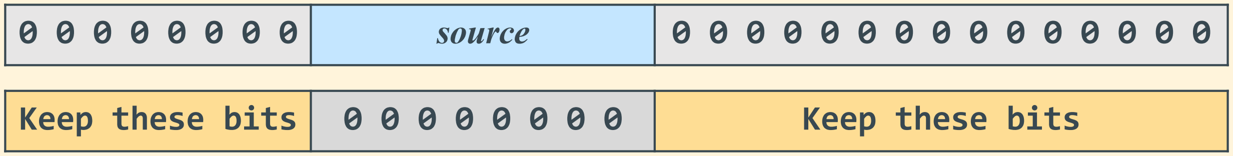
Isolate a field: Use **`lsr, lsl, rot`** to get a field surrounded by zeros


`0 0 0 0 0 0 0 0` | *source* | `0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`

lsl to get this edge into msb

lsr to get this edge into lsb

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


`0 0 0 0 0 0 0 0` | *source* | `0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`

`Keep these bits` | `0 0 0 0 0 0 0 0` | `Keep these bits`

x