

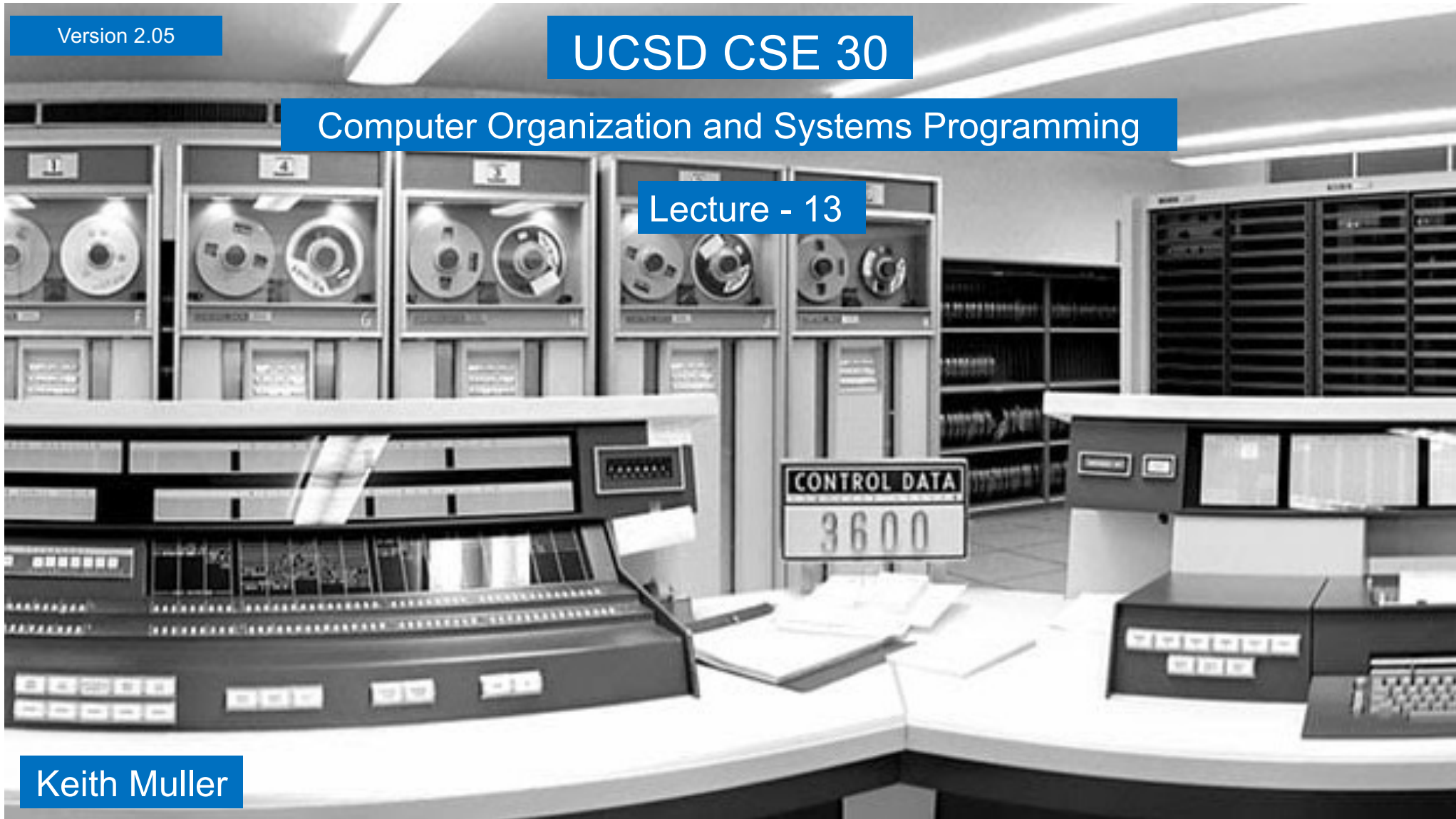
Version 2.05

UCSD CSE 30

Computer Organization and Systems Programming

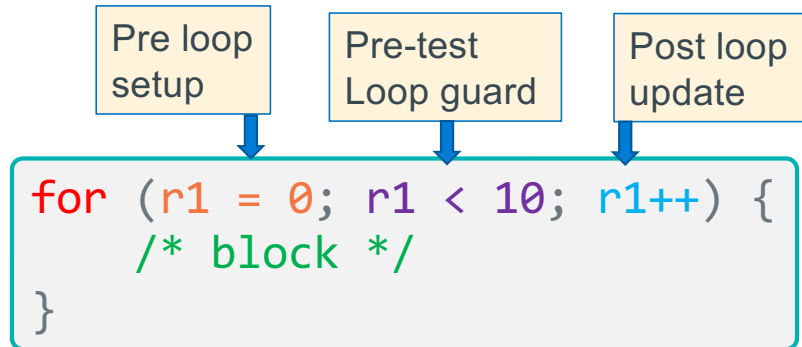
Lecture - 13

Keith Muller



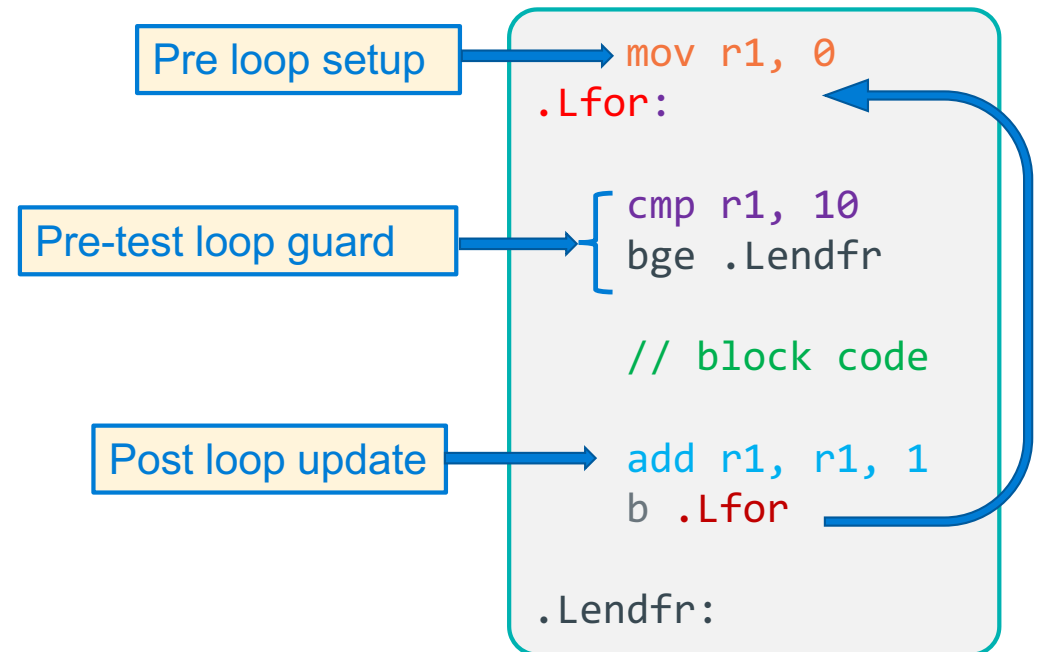


Program Flow – Counting (For) Loop Version 1

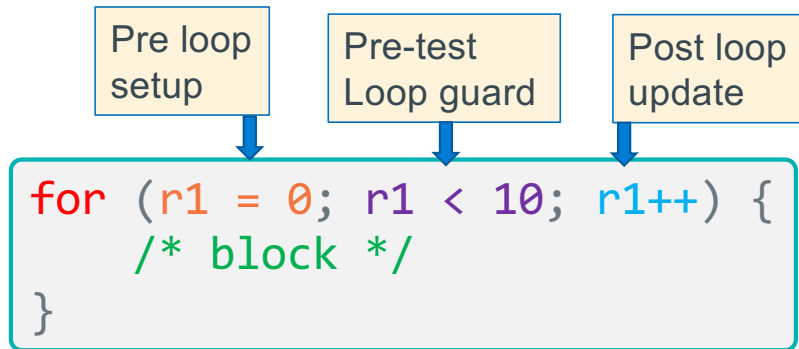


A **counting loop** has three parts:

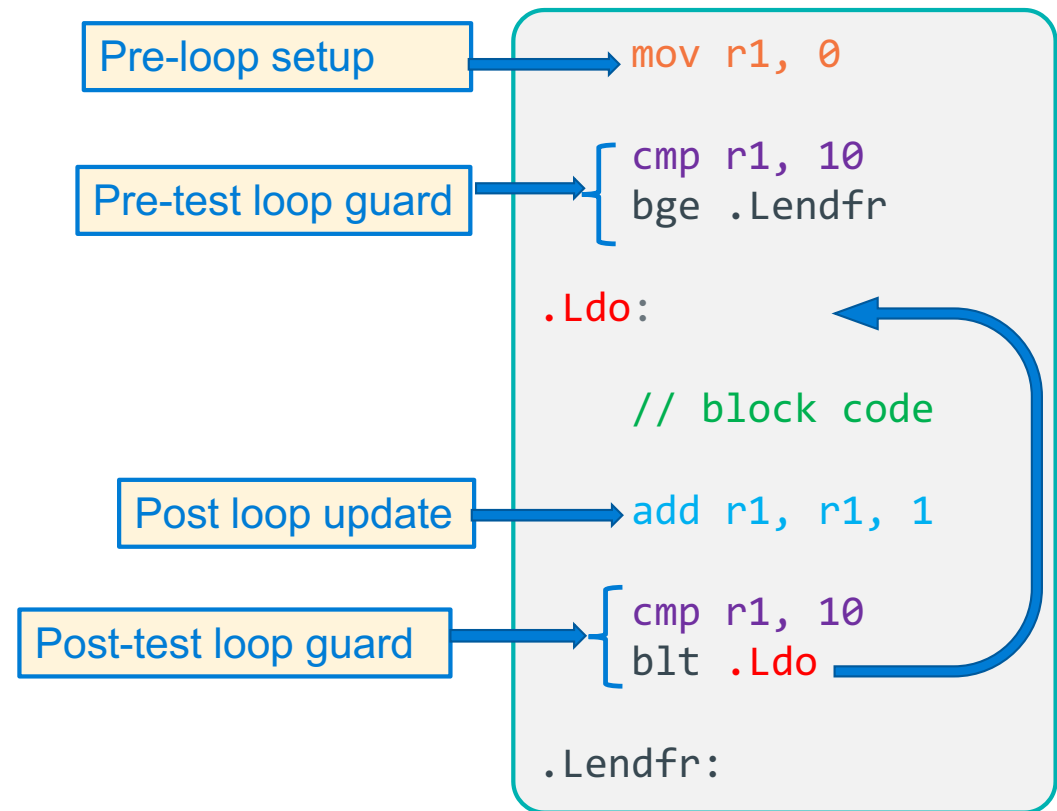
1. Pre-loop setup
2. Pre-test loop guard conditions
3. Post-loop update



Program Flow – Counting (For) Loop – Version 2




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



Nested loops

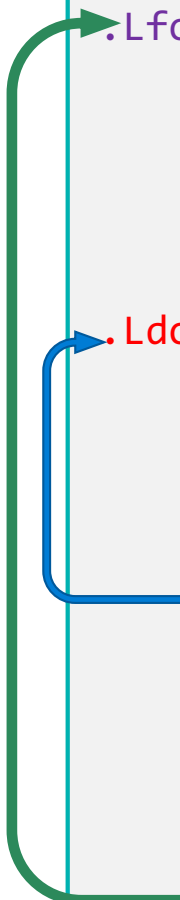
```
for (r3 = 0; r3 < 10; r3++) {  
    r0 = 0;  
  
    do {  
        r0 = r0 + r1++;  
    } while (r1 < 10);  
  
    // fall through  
    r2 = r2 + r1;  
}
```



r5 = r0;

- Nest loop blocks as you would in C or Java

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



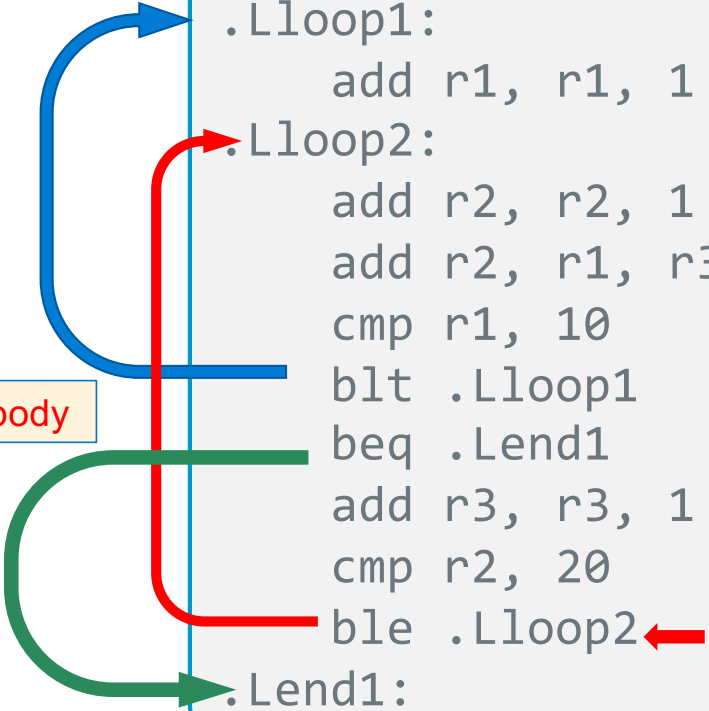
Keep loops Properly Nested: Do not branch into the middle of a loop

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

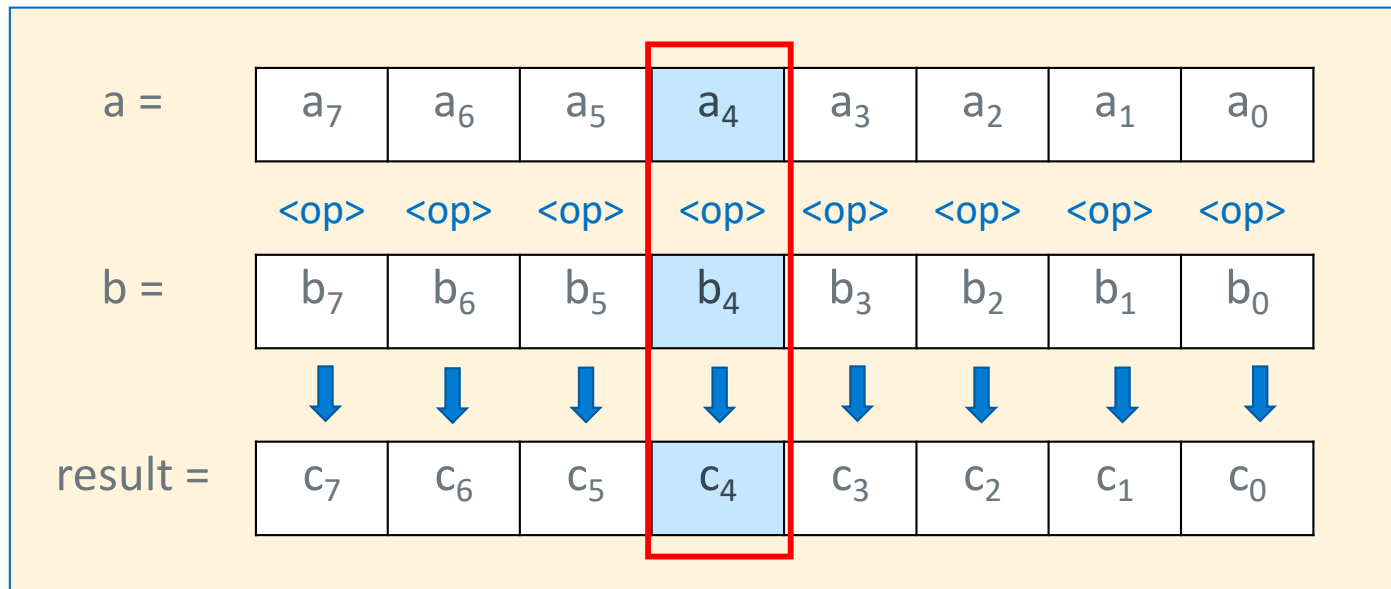
Bad practice: branch into loop body

Do not do the following:

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



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

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
```

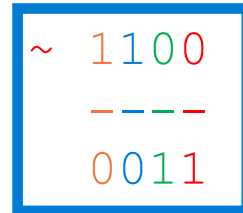
register r0

0x0c



0xffff fff3

Bitwise NOT



- A **bitwise NOT** operation

0x 0c

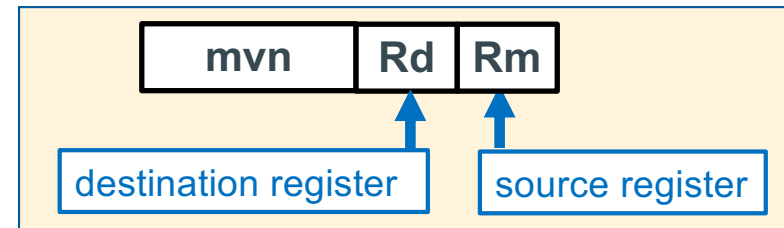
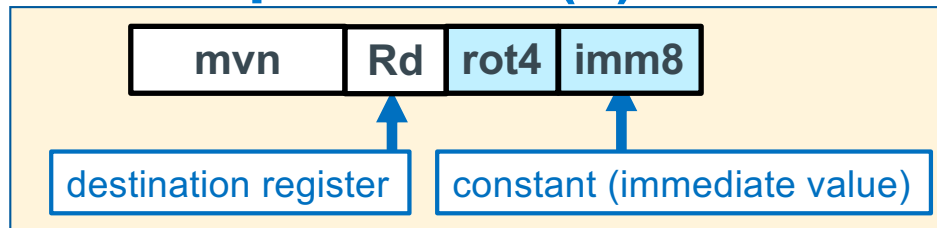
↓ imm8 expansion

0x0000000c

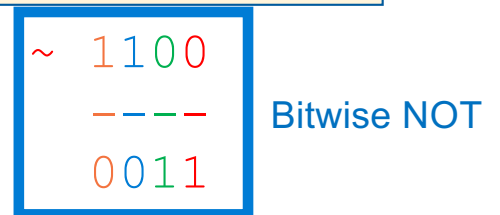
↓ bitwise not

0xfffffffff3

mvn – Copies NOT (~)

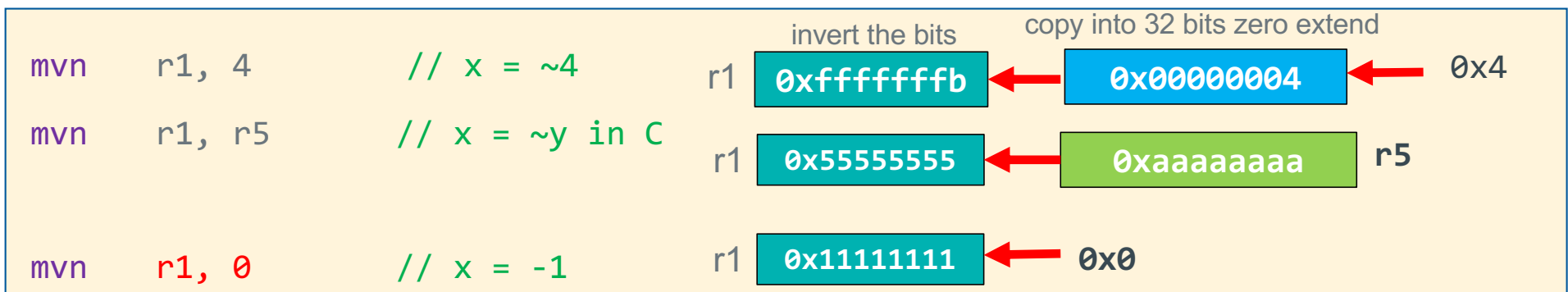


```
mvn Rd, constant // Rd = constant
mvn Rd, Rm       // Rd = Rm
```

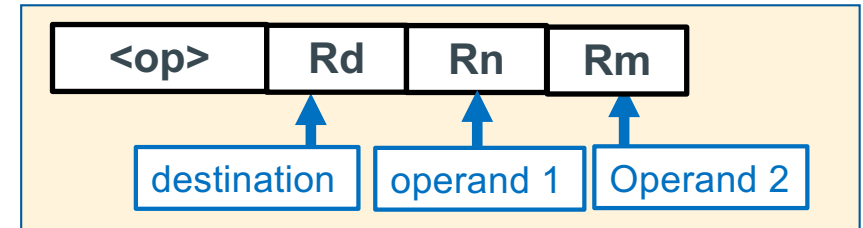
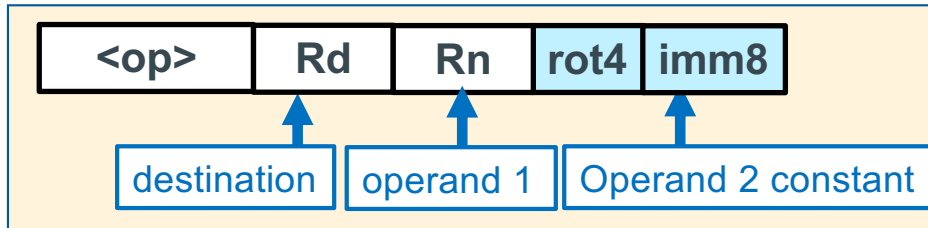


bitwise NOT operation. Immediate (constant) version copies to 32-bit register, then does a bitwise NOT

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 Instructions



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

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

Bitwise <code><op></code> description	C Syntax	Arm <code><op></code> Syntax <i>Op2: either register or constant value</i>	Operation
Bitwise AND	<code>a & b</code>	<code>and Rd, Rn, Op2</code>	$R_d = R_n \& Op2$
Bitwise OR	<code>a b</code>	<code>orr Rd, Rn, Op2</code>	$R_d = R_n Op2$
Exclusive OR	<code>a ^ b</code>	<code>eor Rd, Rn, Op2</code>	$R_d = R_n \wedge Op2$
Bitwise NOT	<code>a = ~b</code>	<code>mvn Rd, Rn</code>	$R_d = \sim R_n$

Bitwise versus C Boolean Operators

Boolean Operators

Bitwise Operators

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

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

bitwise & versus boolean &&

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

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

bitwise ~ versus boolean !

`~0x01 = 0xfffffffffe (bitwise)`

`!0x01 = 0x0 (Boolean)`

The act (operation) of *Masking*



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

unchanged

forces to a 0

RSLT: r1 0xab ab ab 00

Example: force lower 8 bits to 0

DATA: r2 0xab ab ab 77

and r1 r2, 0xffffffff00

r1 = r2 & 0xffffffff00; // in C

RSLT: r1 0xab ab ab 00

Extracting (Isolate) a Field of Bits with a mask

extract top 8 bits of r2 into r1

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

and r1, r2, r3

DATA: r2 0xab ab ab 77

and

MASK: r3 0xff 00 00 00

unchanged

RSLT: r1 0xab 00 00 00

forces to a 0

extract top 8 bits of r2 into r1

DATA: r2 0xab ab ab 77

and r1, r2, 0xff000000

RSLT: r1 0xab 00 00 00

r1 = r2 & 0xff000000; // in C

Finding if a bit is set

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

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

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

```
cmp r1, 0
```

```
beq .Lendif
```

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

```
.Lendif:
```

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

Example is bit 1 set

DATA: r2 0xab ab ab 77

and

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

forces to a 0 unchanged

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

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

Even/Odd

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

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

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

```
cmp r1, 0x01
```

```
bne .Lendif
```

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

```
.Lendif:
```

```
unsigned int r2;
```

```
// code
```

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

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

```
}
```

DATA: r2 0xab ab ab 77

and

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

forces to a 0 unchanged

RSLT: r1 0x00 00 00 01 (odd)

MOD %<power of 2>

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

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

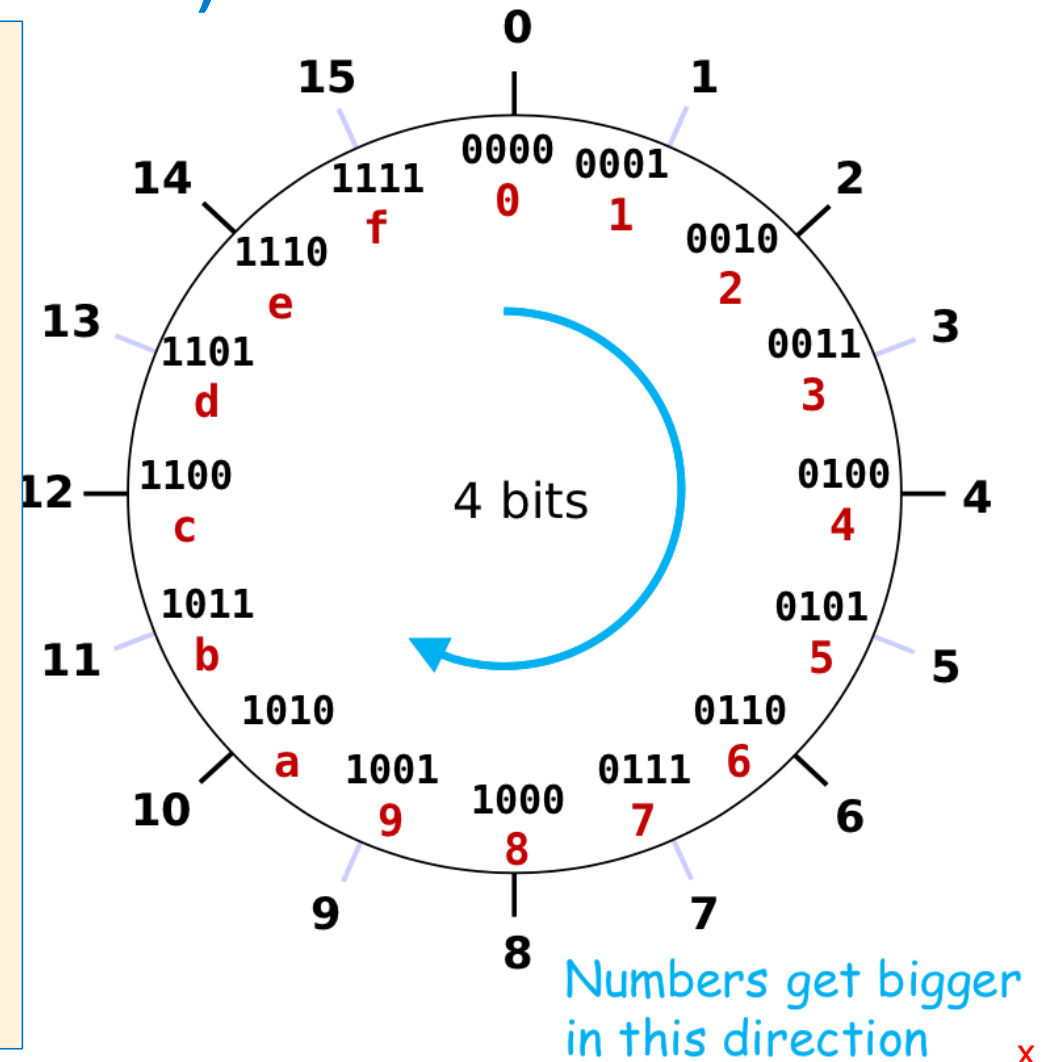
MASK: r3 0x00 00 00 **ff** **apply a 2nd time**

inverts (flips)

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

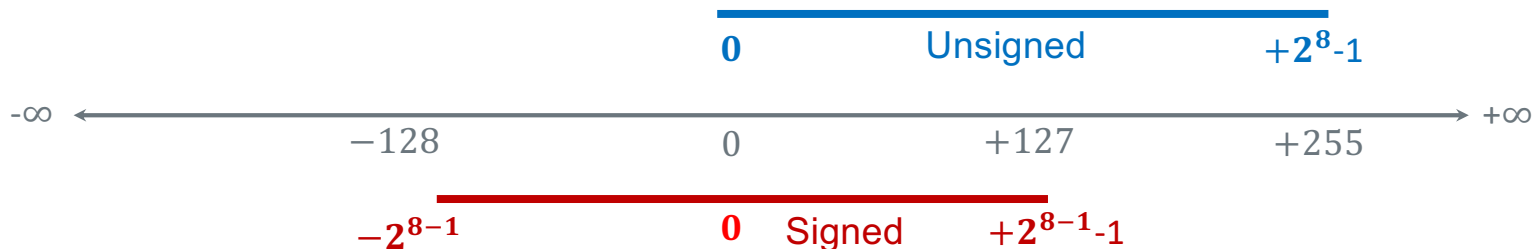
Unsigned Integers (positive numbers) with Fixed # of Bits

- 4 bits is $2^4 = \text{ONLY } 16$ 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 \rightarrow 0001 ... \rightarrow 1111 \rightarrow 0000
- Keep **subtracting** 1
 - wraps (**counter-clockwise**)
 - 1111 \rightarrow 1110 ... \rightarrow 0000 \rightarrow 1111
- Addition and subtraction use **normal** “**carry**” and “**borrow**” rules, just operate in binary



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)
- 2^n distinct bit patterns to encode positive and negative values
- Unsigned values:** $0 \dots 2^n - 1$ ← -1 comes from counting 0 as a "positive" number
- Signed values:** $-2^{n-1} \dots 2^{n-1} - 1$ (dividing the range in ~ half including 0)
- On a number line (below):** 8-bit integers – signed and unsigned (e.g., `char` in C)

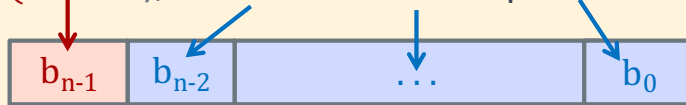


Same "width" (same number of encodings), just shifted in value

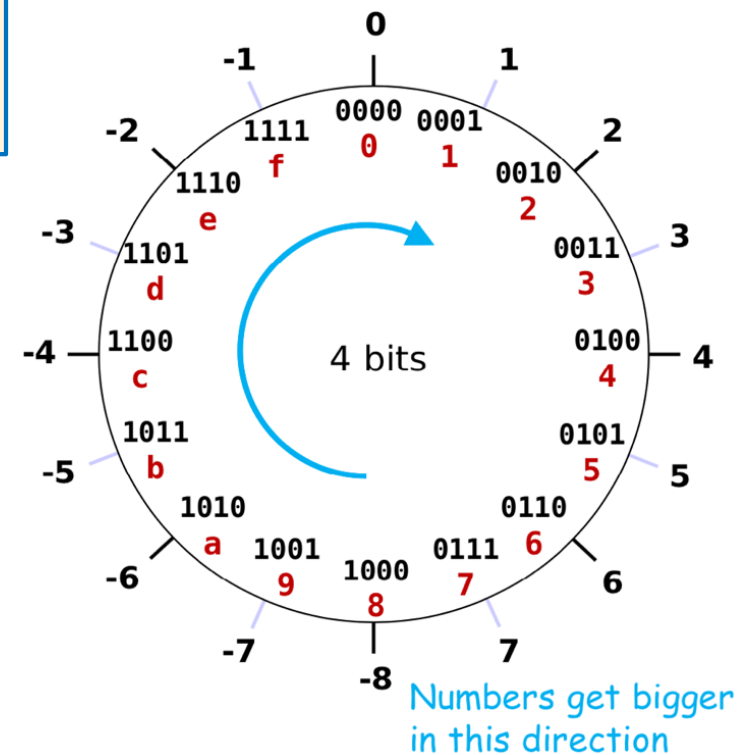
Two's Complement: The MSB Has a *Negative Weight*

$$2's\ Comp = -b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \dots + b_12^1 + b_02^0$$

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

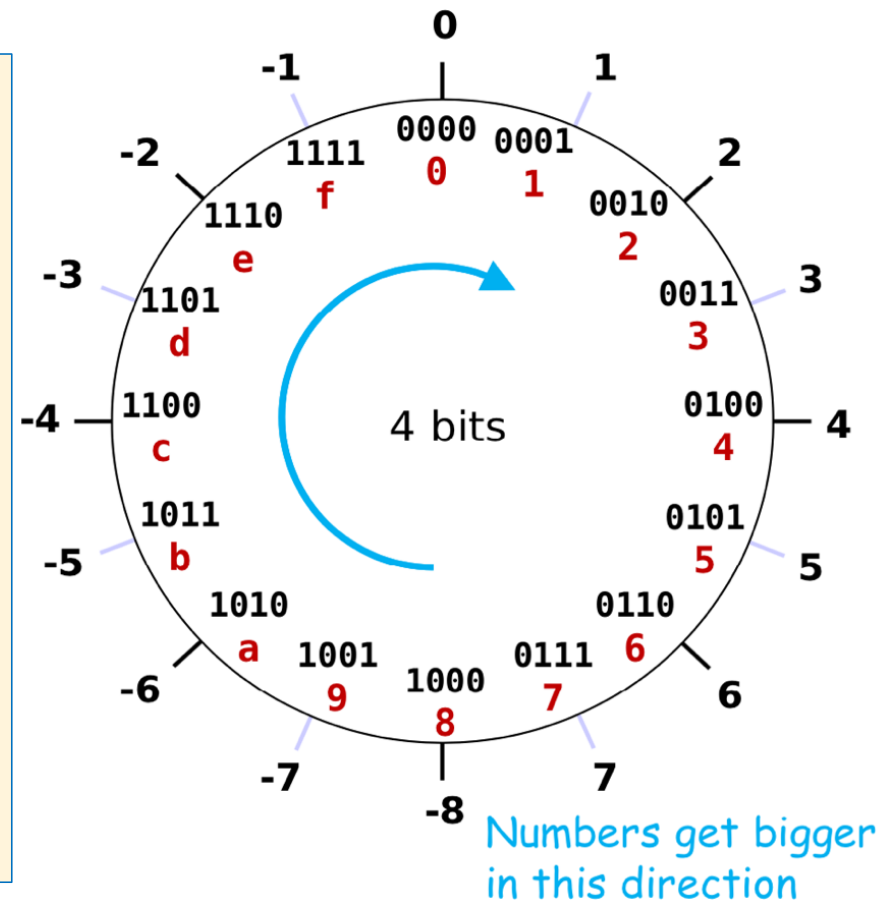


- 4-bit ($w = 4$) weight = $-2^{4-1} = -2^3 = -8$
 - 1010_2 **unsigned**:
 $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 10$
 - 1010_2 **two's complement**:
 $-1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = -8 + 2 = -6$
 - 8 in **two's complement**:
 $1000_2 = -2^3 + 0 = -8$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



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!*



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

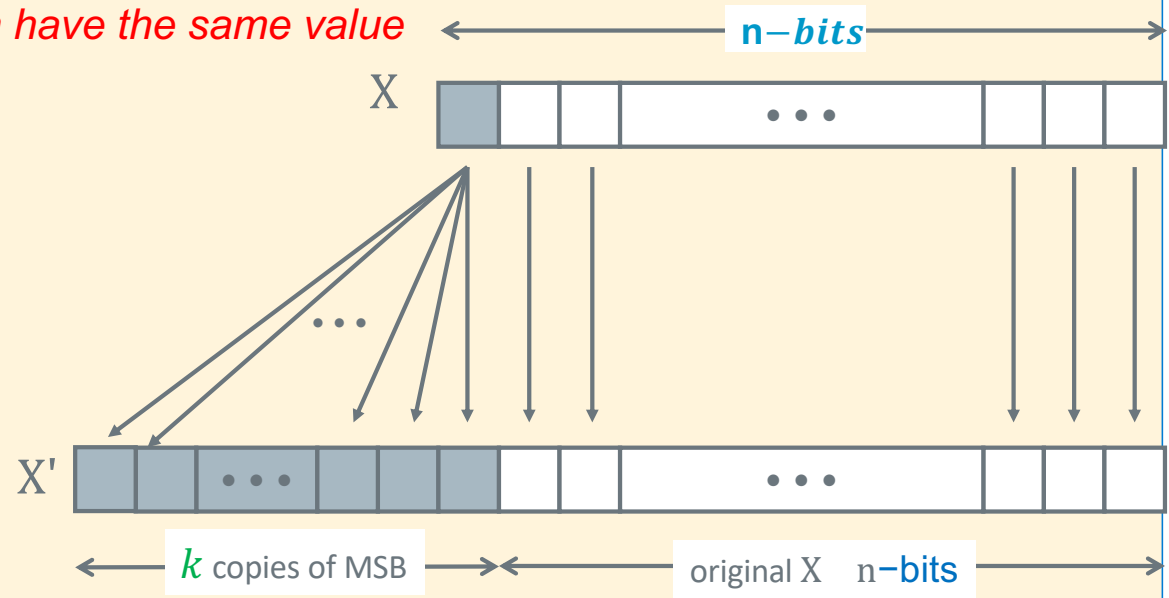
- Sign extension increases the number of bits:** n -bit wide signed integer X , **EXPANDS** to a **wider** n -bit + k -bit signed integer X' where **both have the same value**

Unsigned

- Just add leading zeroes to the left side

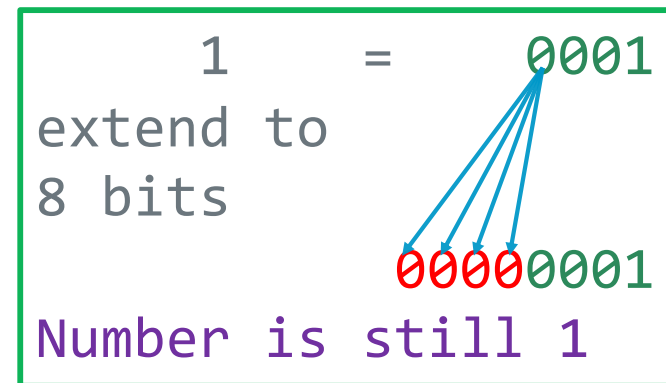
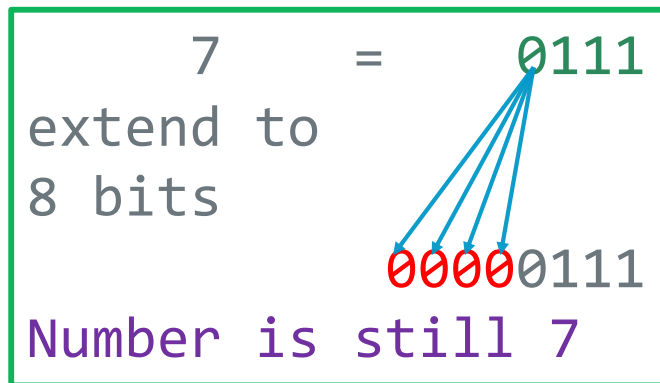
Two's Complement Signed:

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



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

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



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

$$\begin{aligned} 1001 &= -8 + 1 = -7 \\ \text{1111}1001 &= \\ (-128 + 64 + 32 + 16 + 8) + 1 &= \\ = -8 + 1 &= -7 \end{aligned}$$

7 = 00000111

invert = 11111000

add 1 + 1

-7 11111001

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)

```
#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

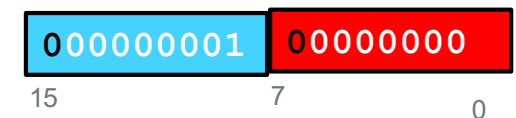
Spanning one or more contiguous bytes of memory

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



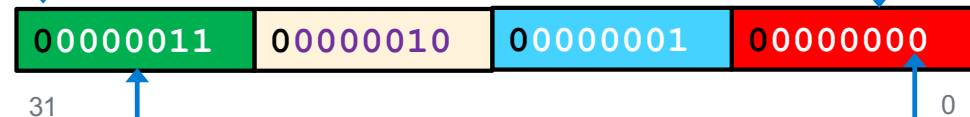
Half Word 16-bit integer uses 2 bytes



most significant bit (largest power of 2)

least significant byte

Word 32-bit integer uses 4 bytes

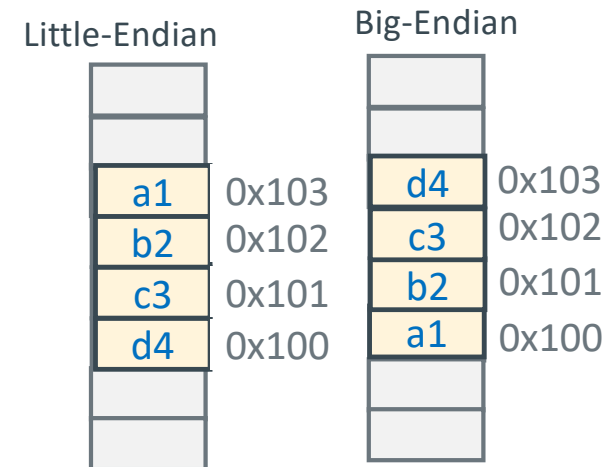
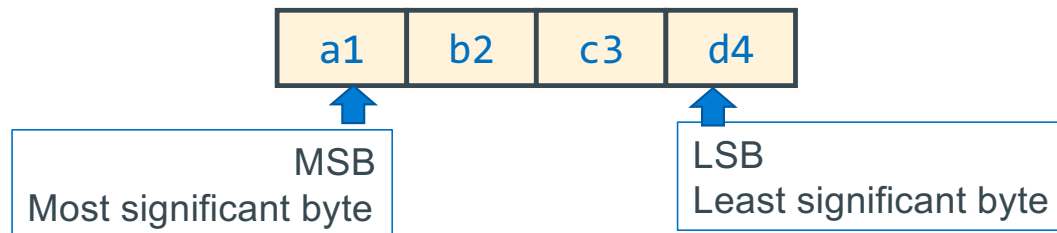


least significant bit (smallest power of 2)

most significant byte

Byte Ordering of Numbers In Memory: Endianness

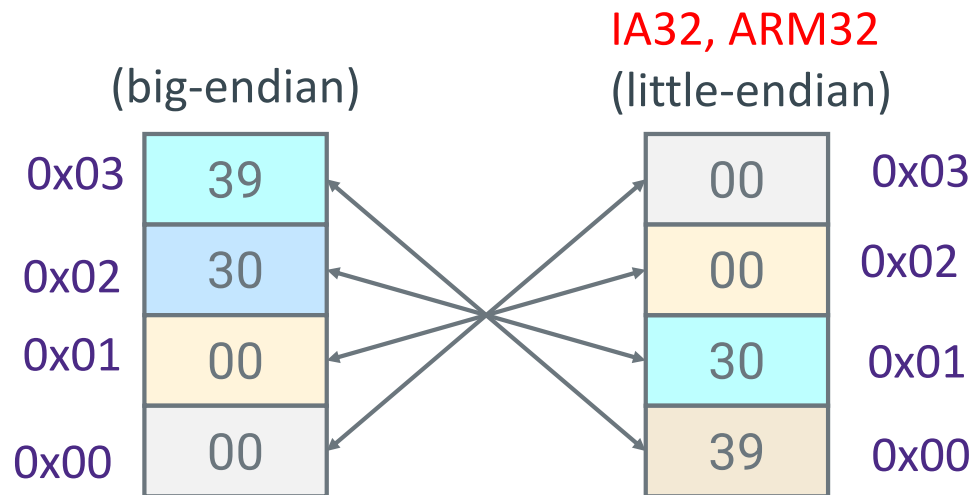
- Two different ways to place multi-byte integers in a **byte addressable** memory
- **Big-endian**: **Most** Significant Byte (“**big end**”) starts at the **lowest (starting)** address
- **Little-endian**: **Least** Significant Byte (“**little end**”) starts at the **lowest (starting)** address
- Example: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



Byte Ordering Example

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

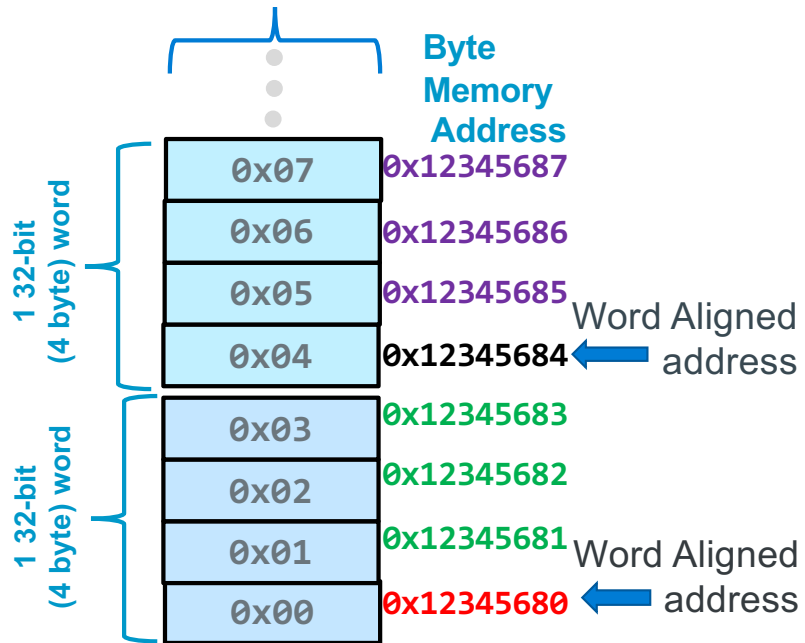
```
int x = 12345;  
// or x = 0x00003039; // show all 32 bits
```



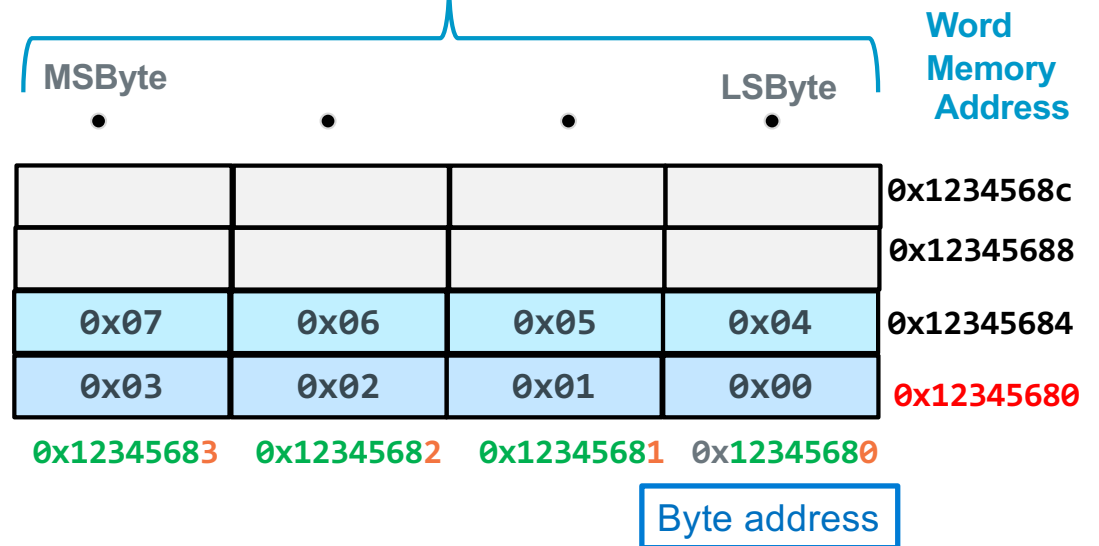
Byte Addressable Memory Shown as 32-bit words

1 byte Memory Content

One byte per row



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



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

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

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

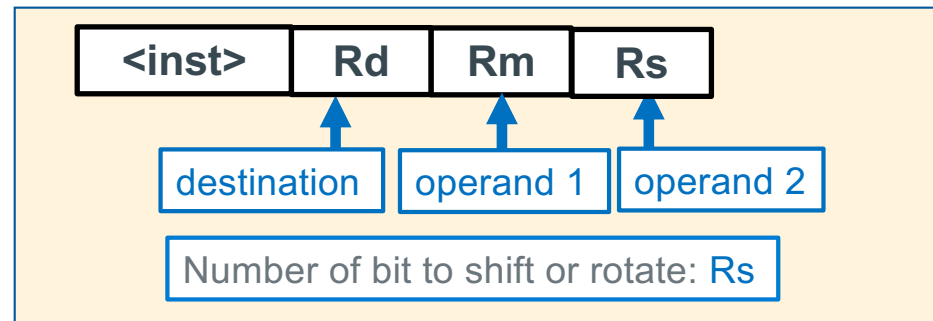
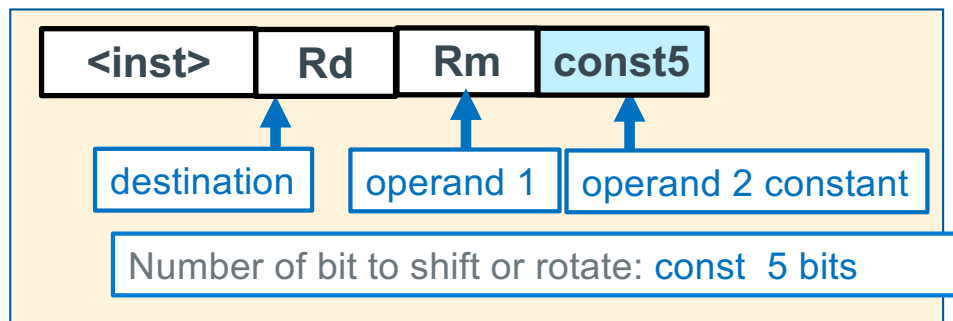
    for (int i = SZ-1; i >= 0; i--)
        printf("foo[%d]: %x\n", i, *(iptr + i));

    for (int i = sizeof(foo)-1; i >= 0; i--)
        printf("byte %d: %x\n", i, (unsigned int)*(chptr + i));
    return 0;
}
```

```
kmuller@keithm-pi4:~$ ./a.out
foo[1]: aabbccdd
foo[0]: 11223344
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

Shift and Rotate Instructions



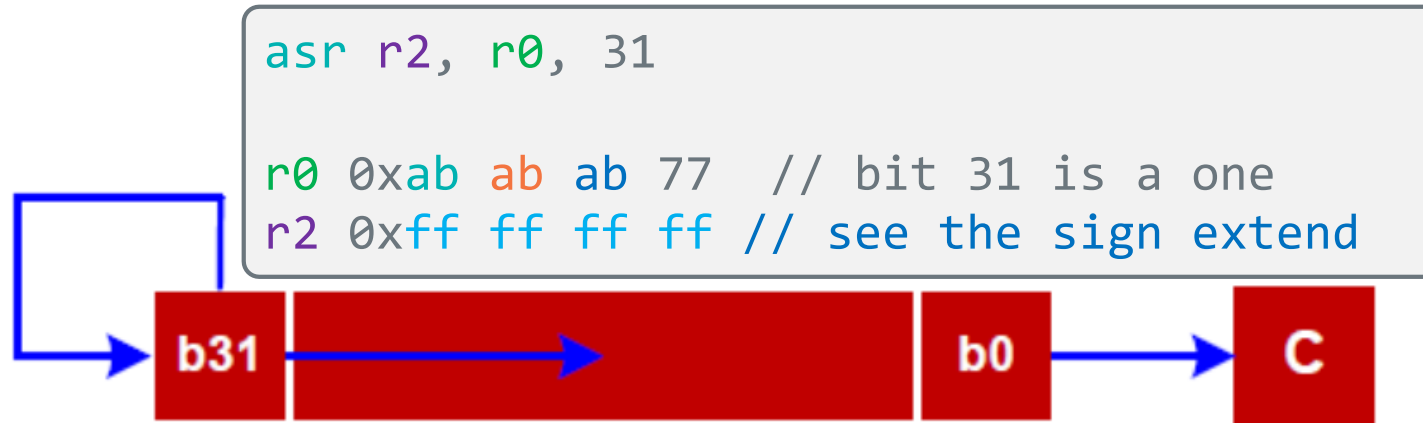
Instruction	Syntax	Operation	Notes	Diagram
Logical Shift Left <code>int x; or unsigned int x</code> <code>x << n;</code>	<code>lsl Rd, Rm, const5</code> <code>lsl Rd, Rm, Rs</code>	$R_d \leftarrow R_m \ll const5$ $R_d \leftarrow R_m \ll R_s$	Zero fills shift: 0 - 31	
Logical Shift Right <code>unsigned int x;</code> <code>x >> n;</code>	<code>lsr Rd, Rm, const5</code> <code>lsr Rd, Rm, Rs</code>	$R_d \leftarrow R_m \gg const5$ $R_d \leftarrow R_m \gg R_s$	Zero fills shift: 1 - 32	
Arithmetic Shift Right <code>int x;</code> <code>x >> n;</code>	<code>asr Rd, Rm, const5</code> <code>asr Rd, Rm, Rs</code>	$R_d \leftarrow R_m \gg const5$ $R_d \leftarrow R_m \gg R_s$	Sign extends shift: 1 - 32	
Rotate Right <code>unsigned int x;</code> <code>x = (x>>n) (x<<(32-n));</code>	<code>ror Rd, Rm, const5</code> <code>ror Rd, Rm, Rs</code>	$R_d \leftarrow R_m \text{ ror } const5$ $R_d \leftarrow R_m \text{ ror } R_s$	right rotate rot: 0 - 31	

Shift Operations in C

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



Arithmetic Shift Right Example: Testing Sign

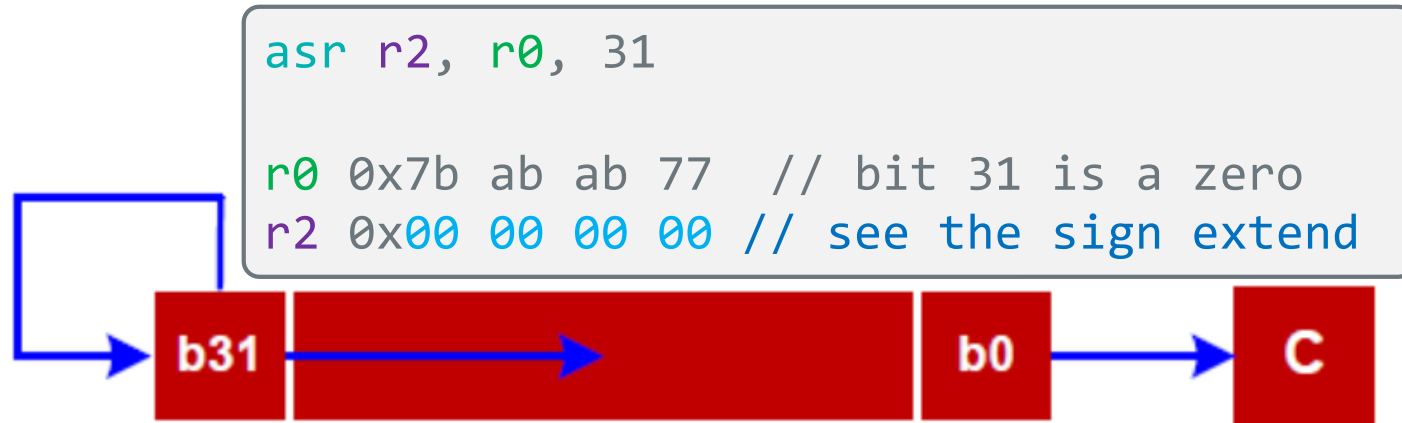


Test for sign
-1 if r0 negative

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

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

Arithmetic Shift Right Example: Testing Sign



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

Test for sign
0 if r0 positive

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

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
```



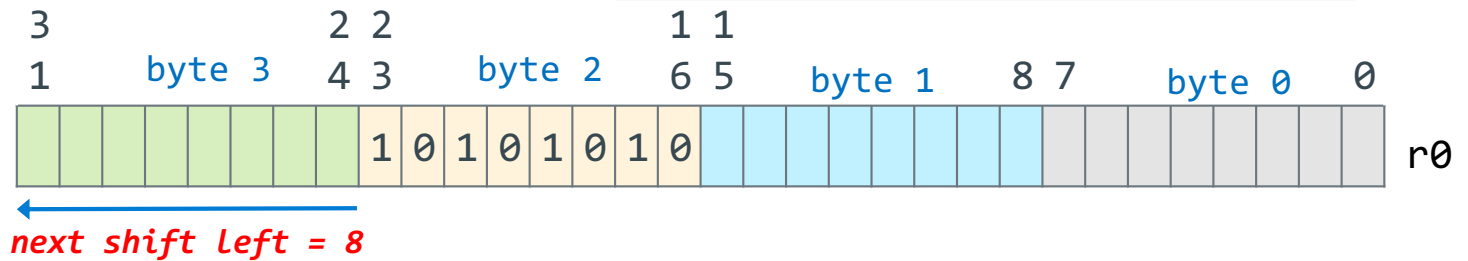
```
ror r2, r0, 8
```

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

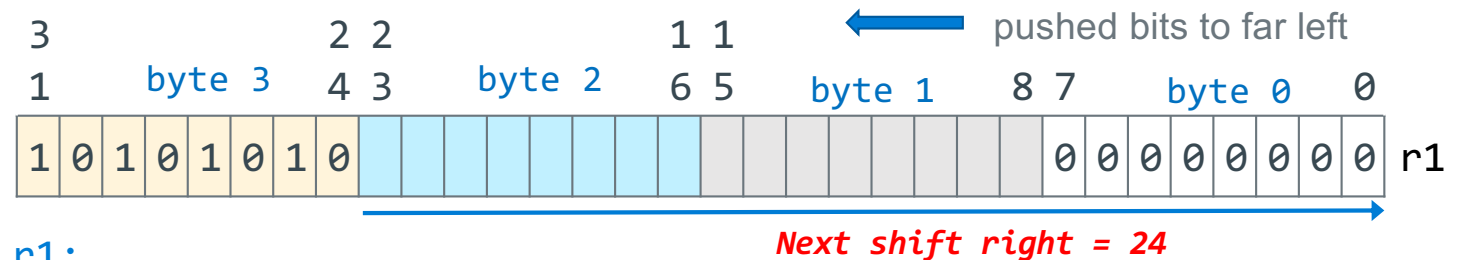
Extracting/Isolating Unsigned Bitfields

Hint: Useful for PA7

- Move byte 2 in r0 to byte 0 in r1



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

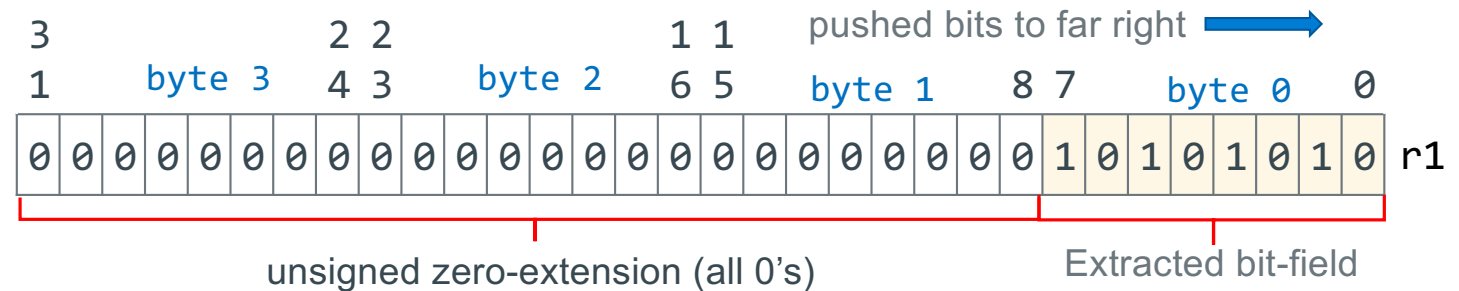


```
unsigned int r0,r1;
```

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

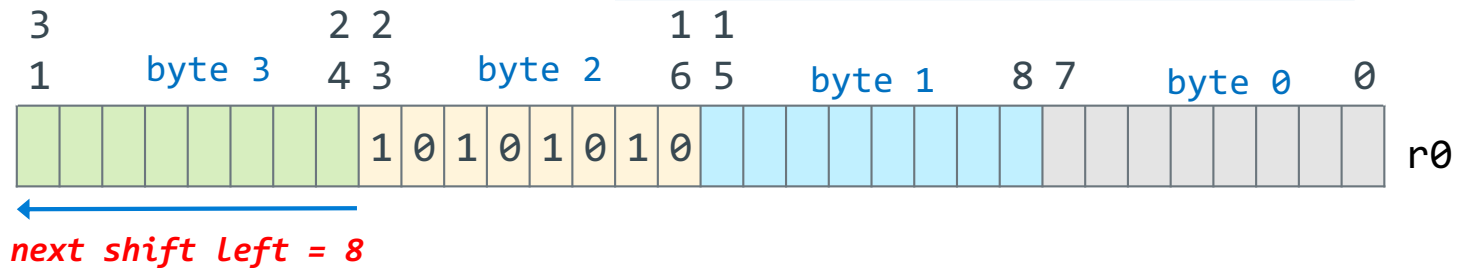
```
lsl r1, r1, 24
```

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

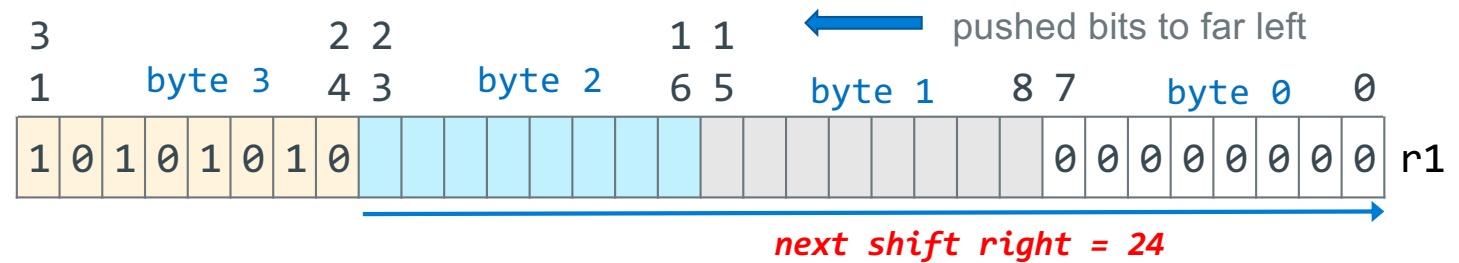


Extracting Signed Bitfields

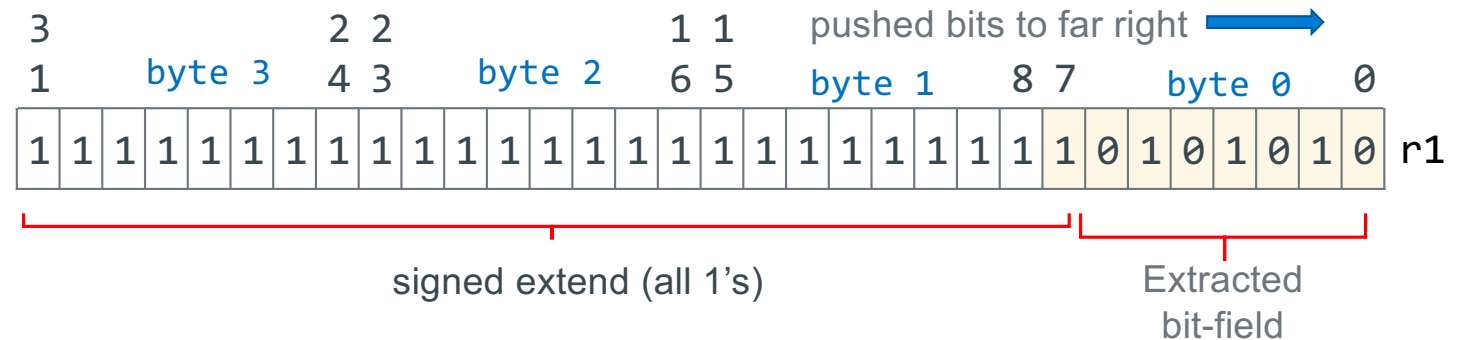
- Move byte 2 in r0 to byte 0 in r1



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



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



Inserting Bitfields – Inserting Source Field into Destination Field

Task: Insert source into destination

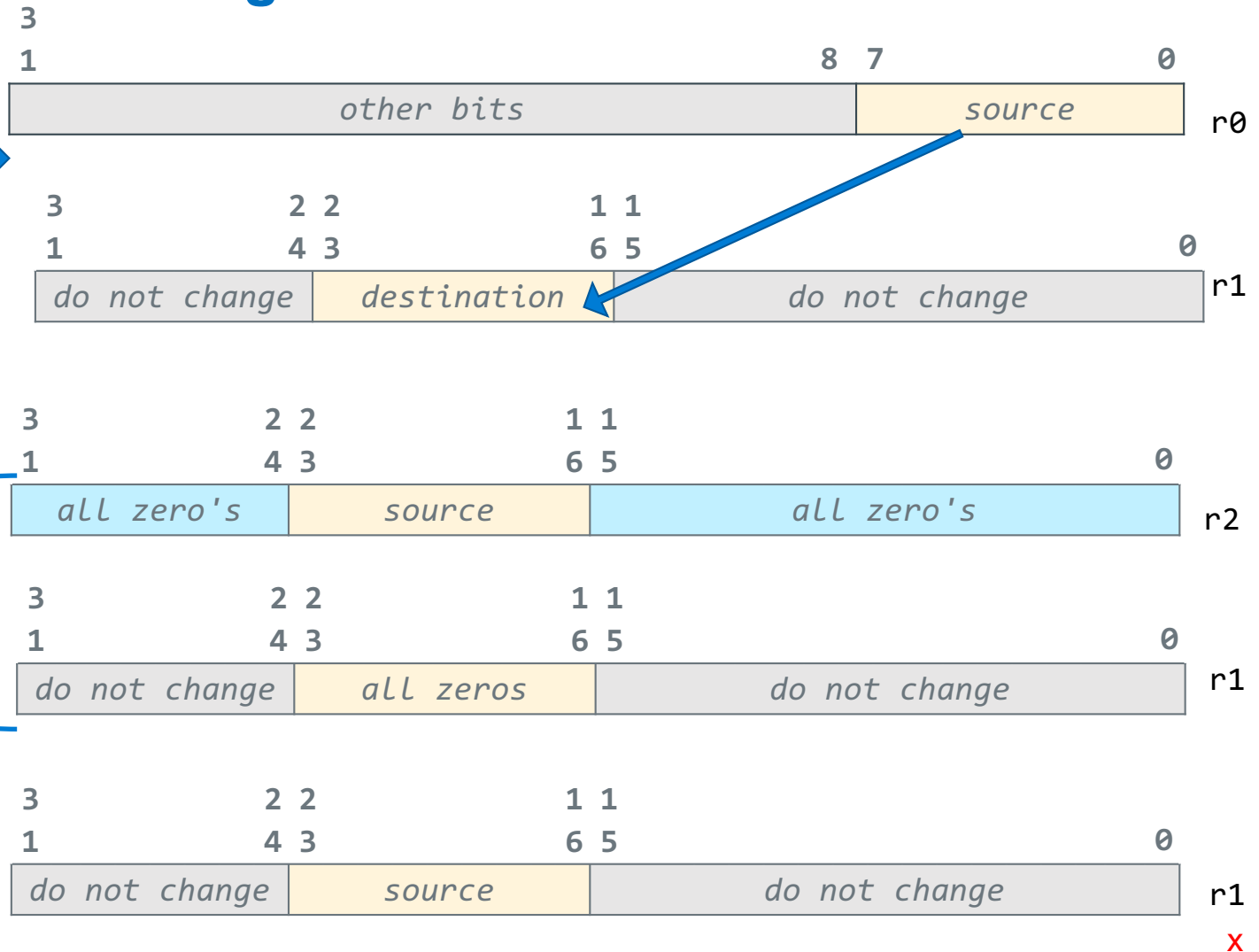
a	b	a b
0	0	0
0	1	1
1	0	1
1	1	1

Approach

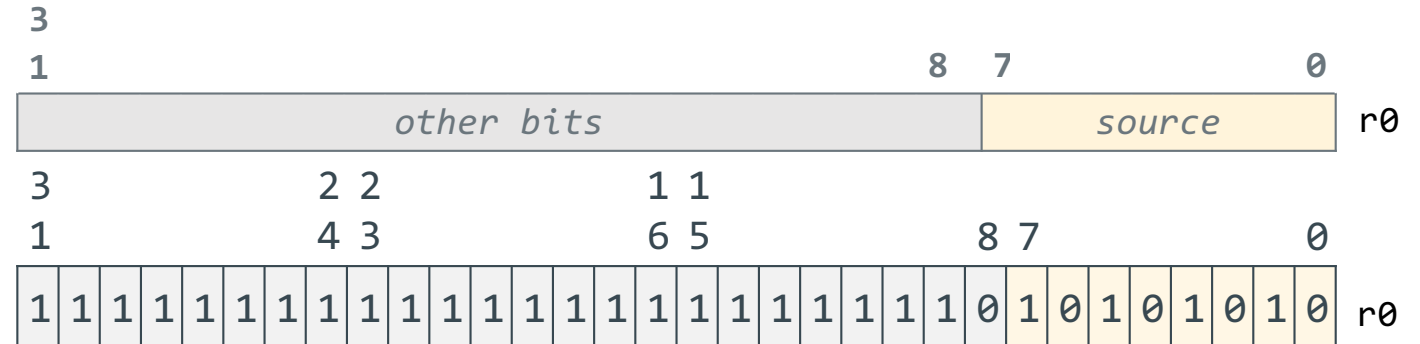
- (1) isolate source field
- (2) clear destination field
- (3) Bitwise **or** together

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

results in



Inserting Bitfields – Isolating the Source Field



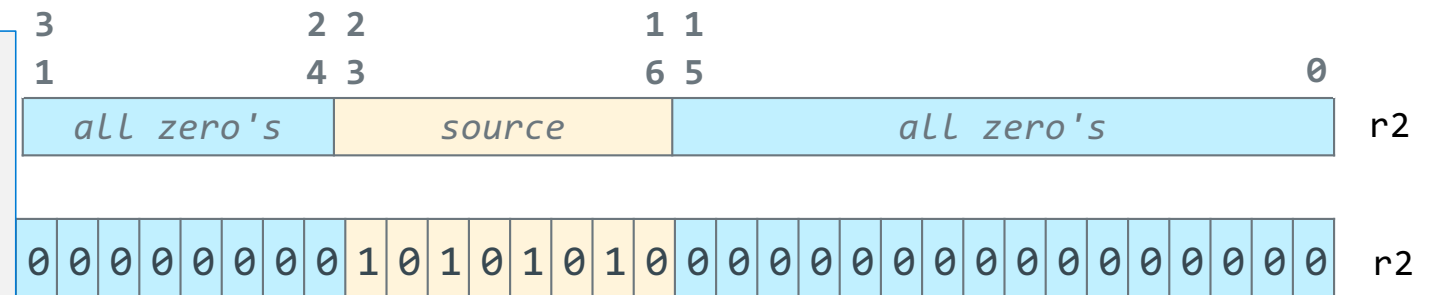
```
isolate source field
```

$$ls1 \quad r2, r0, 24$$

lsr r2, r2, 8

$$r_2 = r_0 \ll 24;$$

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

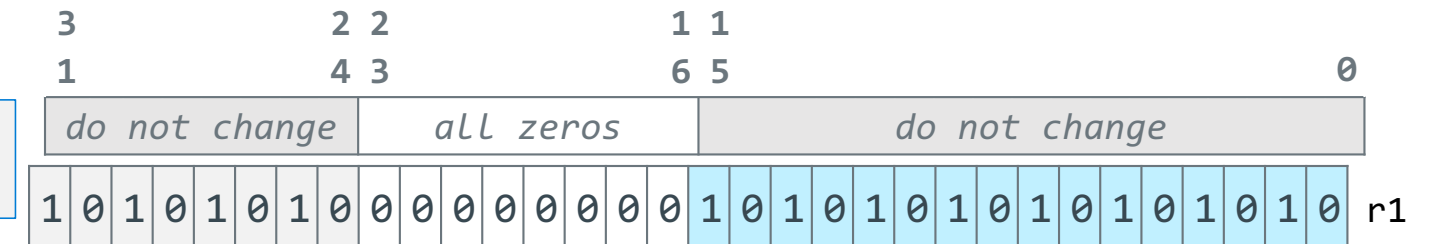
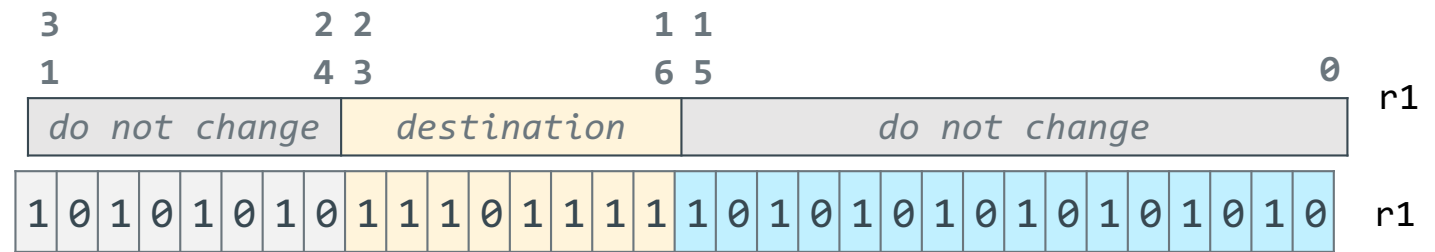


Inserting Bitfields – Clearing the Destination Field

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

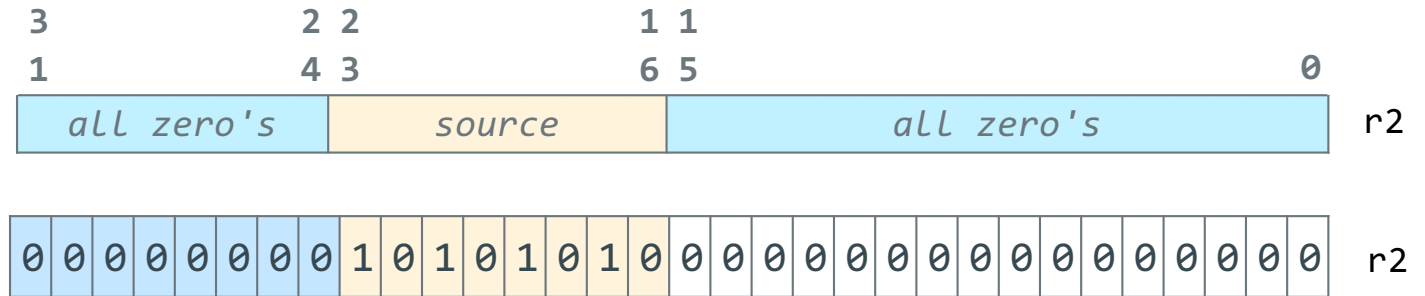
```
lsl    r1, r1, 8
r1 = r1 << 8;
```

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

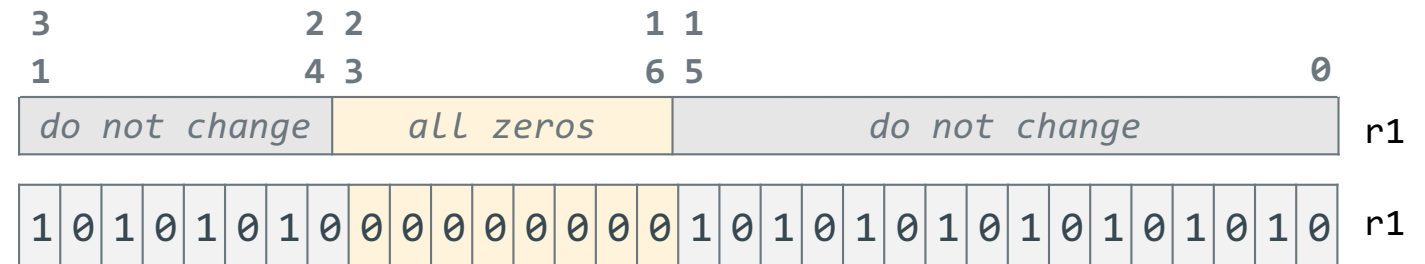


Inserting Bitfields – Combining Isolated Source and Cleared Destination

isolated source



field cleared in
destination



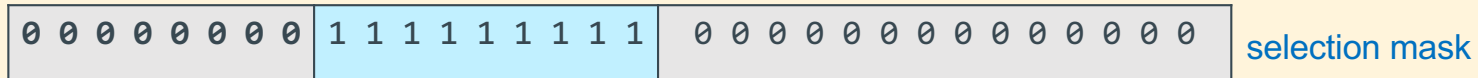
inserted field
orr r1, r1, r0
r1 = r1 | r0;



Masking Summary

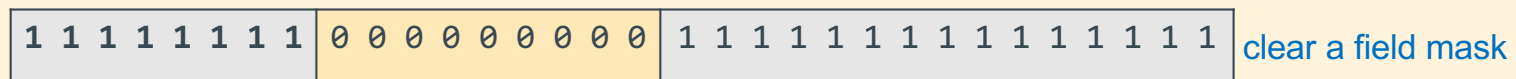
Select a field: Use **and** with a **mask** of one's surrounded by zero's to select the bits that have a 1 in the mask, all other bits will be set to zero

selects this field when used with and



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

clears this field when used with and



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



lsl to get this edge into msb

lsl to get this edge into lsb

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



Reference For PA7/8: C Stream Functions Opening Files

```
FILE *fopen(char filename[], const char mode[]);
```

- Opens a stream to the specified file in specified file access mode
 - returns NULL on failure – **always check the return value; make sure the open succeeded!**
- Mode is a string that describes the actions that can be performed on the stream:

"r" Open for reading.

The stream is positioned at the beginning of the file. Fail if the file does not exist.

"w" Open for writing.

The stream is positioned at the beginning of the file. Create the file if it does not exist.

"a" Open for writing.

The stream is positioned at the end of the file. Create the file if it does not exist.

Subsequent writes to the file will always be at current end of file.

- An optional "+" following "r", "w", or "a" opens the file for both reading and writing

Reference: C Stream Functions Closing Files and Usage

```
int fclose(FILE *stream) ;
```

- Closes the specified stream, forcing output to complete (eventually)
 - returns EOF on failure (often ignored as no easy recovery other than a message)
- Usage template for **fopen()** and **fclose()**
 1. Open a file with **fopen()** **always** checking the return value
 2. do i/o – keep calling stdio io routines
 3. close the file with **fclose()** when done with that I/O stream

C Stream Functions Array/block read/write

- These do not process contents they simply **transfer** a fixed number of bytes to and from a buffer passed to them
- `size_t fwrite(void *ptr, size_t size, size_t count, FILE *stream);`
 - Writes an array of *count elements* of *size* bytes from *stream*
 - *Updates the write file pointer forward by the number of bytes written*
 - returns number of elements written
 - error is short element count or 0
- `size_t fread(void *ptr, size_t size, size_t count, FILE *stream);`
 - Reads an array of *count elements* of *size* bytes from *stream*
 - *Updates the read file pointer forward by the number of bytes read*
 - returns number of elements read, **EOF is a return of 0**
 - error is short element count or 0
- **I almost always set size to 1 to return bytes read/written**

C fread/fwrite Example - 1

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define BFSZ      8192 /* size of read */
int main(void)
{
    unsigned char fbuf[BFSZ];
    FILE *fin, *fout;
    size_t readlen;
    size_t bytes_copied = 0;
    retval = EXIT_SUCCESS;
    if (argc != 3){
        fprintf(stderr, "%s requires two args\n", argv[0]);
        return EXIT_FAILURE;
    }
    /* Open the input file for read */
    if ((fin = fopen(argv[1], "r")) == NULL) {
        fprintf(stderr, "fopen for read failed\n");
        return EXIT_FAILURE;
    }
    /* Open the output file for write */
    if ((fout = fopen(argv[2], "w") == NULL) {
        fprintf(stderr, "fopen for write failed\n");
        fclose(fin);
        return EXIT_FAILURE;
    }
}
```

To handle
bytes moved

```
% ls -ls ZZZ
ls: ZZZ: No such file or directory
% ./a.out cp.c ZZZ
bytes copied: 1122
% ls -ls cp.c ZZZ
8 -rw-r--r--  1 kmuller  staff  1122 Jul  2 08:51 ZZZ
8 -rw-r--r--  1 kmuller  staff  1122 Jul  2 08:49 cp.c
```

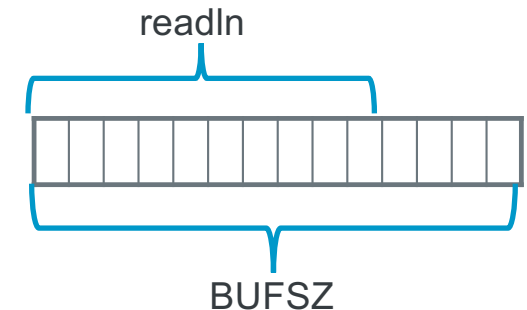
C fread/fwrite Example - 2

```
/* Read from the file, write to fout */  
while ((readlen = fread(fbuf, 1, BUFSIZ, fin)) > 0) {  
    if (fwrite(fbuf, 1, readlen, fout) != readlen) {  
        fprintf(stderr, "write failed\n");  
        retval = EXIT_FAILURE;  
        break;  
    }  
    bytes_copied += readlen; //running sum bytes copied  
}  
  
if (retval == EXIT_FAILURE)  
    printf("Failure Copy did not complete only ");  
printf("Bytes copied: %zu\n", bytes_copied);  
  
fclose(fin);  
fclose(fout);  
  
return retval;  
}
```

By using an element size of 1 with a char buffer, this is byte I/O

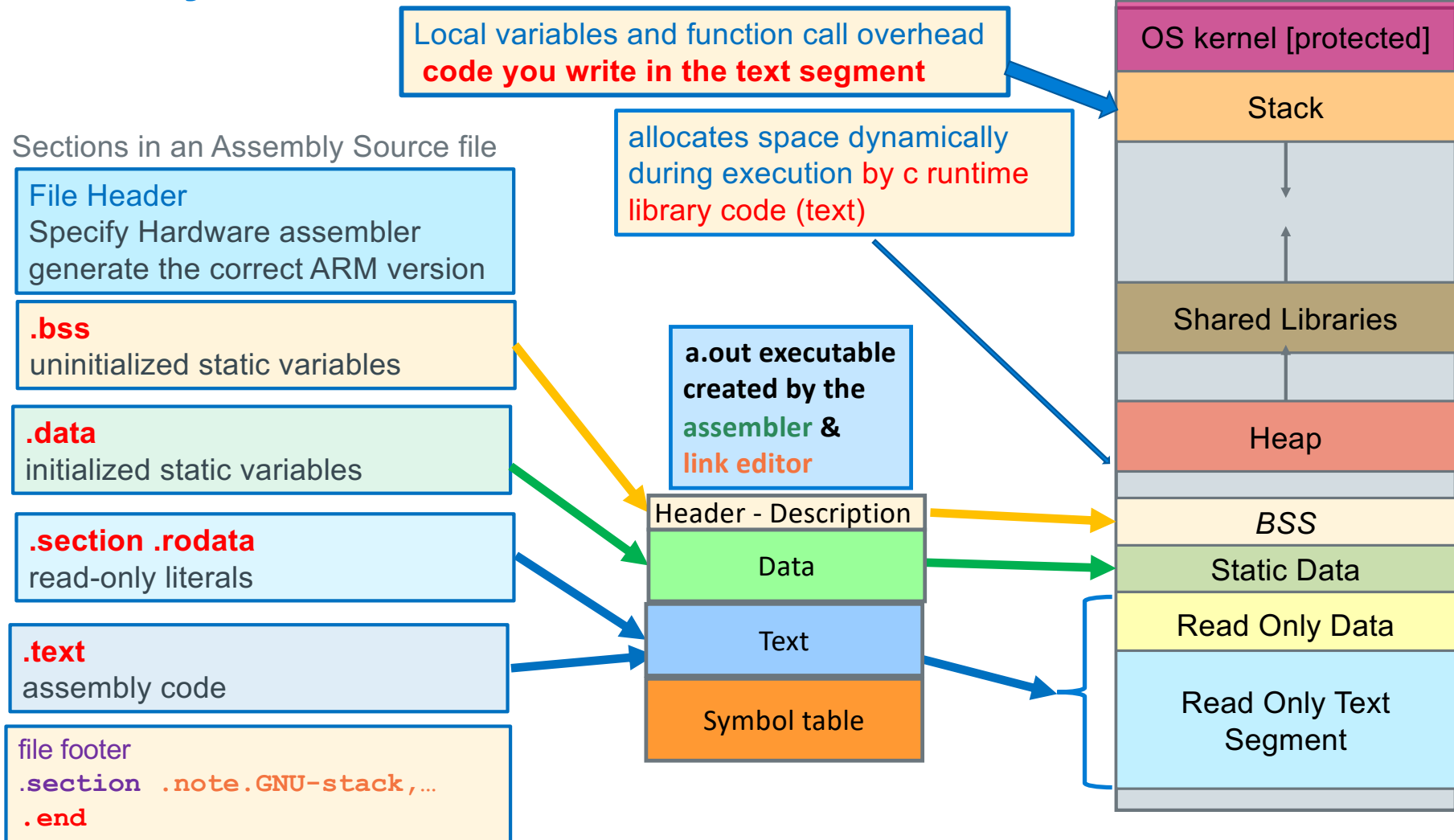
Capture the bytes read so you know how many bytes to write

unless the **input file length is an exact multiple of BUFSIZ**, last `fread()` will always read less than BUFSIZ which is why you write `readlen`



Jargon: the last record is often called the "runt"

Assembly Source File to Executable to Linux Memory



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)
```

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

Preview: Return Value and Passing Parameters to Functions

(Four parameters or less)

Register	Function Call Use
r0	1 st parameter
r1	2 nd parameter
r2	3 rd parameter
r3	4 th parameter

Register	Function Return Value Use
r0	8, 16 or 32-bit result, 32-bit address or least-significant half of a 64-bit result
r1	most-significant half of a 64-bit result

- Where **r0**, **r1**, **r2**, **r3** are arm registers, the function declaration is (first four arguments):

```
r0 = function(r0, r1, r2, r3)           // 32-bit return
```

```
r0, r1 = function(r0, r1, r2, r3)      // 64-bit return - long long
```
- Each **parameter** and **return value** is limited to data that **can fit in 4 bytes or less**
- You receive **up to the first four parameters in these four registers**
- You copy up to the first four parameters into these four registers before calling a function
- For parameter values using more than 4 bytes, a pointer to the parameter is passed (we will cover this later)
- You MUST ALWAYS assume** that the called function will **alter the contents of all four registers: r0-r3**
 - In terms of C runtime support, these registers contain the copies given to the called function
 - C allows the copies to be changed in any way by the called function

Preview: Writing an ARM32 function

```
#include <stdlib.h>
#include <stdio.h>
int sum4(int, int, int, int);
int main()
{
    int reslt;

    reslt = sum4(1,2,3,4);

    printf("%d\n", reslt);
    return EXIT_SUCCESS;
}
```

```
#ifndef SUM4_H
#define SUM4_H

#ifdef __ASSEMBLER__
int sum4(int, int, int, int);
#else
.extern sum4
#endif

#endif
```

two _

```
#include "sum4.h"
.arch armv6
.arm
.fpu vfp
.syntax unified
.global sum4
.type sum4, %function
.equ FP_OFF, 28
// r0 = sum4(r0, r1, r2, r3)
sum4:
    push    {r4-r9, fp, lr}
    add     fp, sp, FP_OFF

    add     r0, r0, r1
    add     r0, r0, r2
    add     r0, r0, r3

    sub     sp, fp, FP_OFF
    pop     {r4-r9, fp, lr}
    bx      lr

    .size sum4, (. - sum4)
    .section .note.GNU-stack,"",%progbits
.end
```

```
$ gcc -Wall -Wextra -c main.c
$ gcc -c sum4.S
$ gcc sum4.o main.o
$ ./a.out
10
```


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

32-bit memory



register r0

register r1 (address)



r1 is being used as a pointer to a location in memory

ldr requires the use of a pointer operand

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)

register r0



32-bit memory

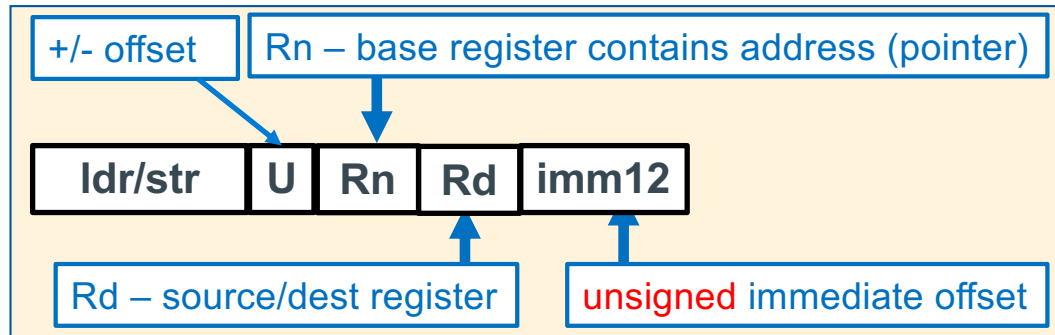
r1 is being used as a pointer to a location in memory

str requires the use of a pointer operand

register r1 (address)



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**

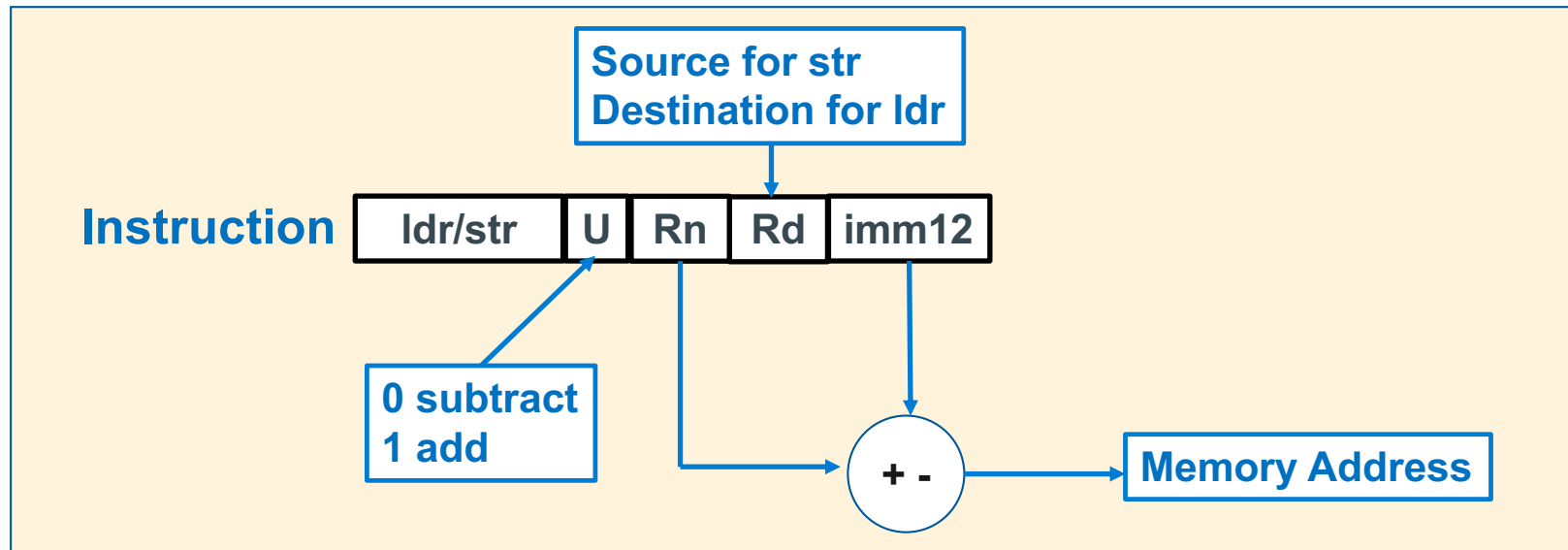
- **Register Base + immediate offset Addressing:**

- **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)

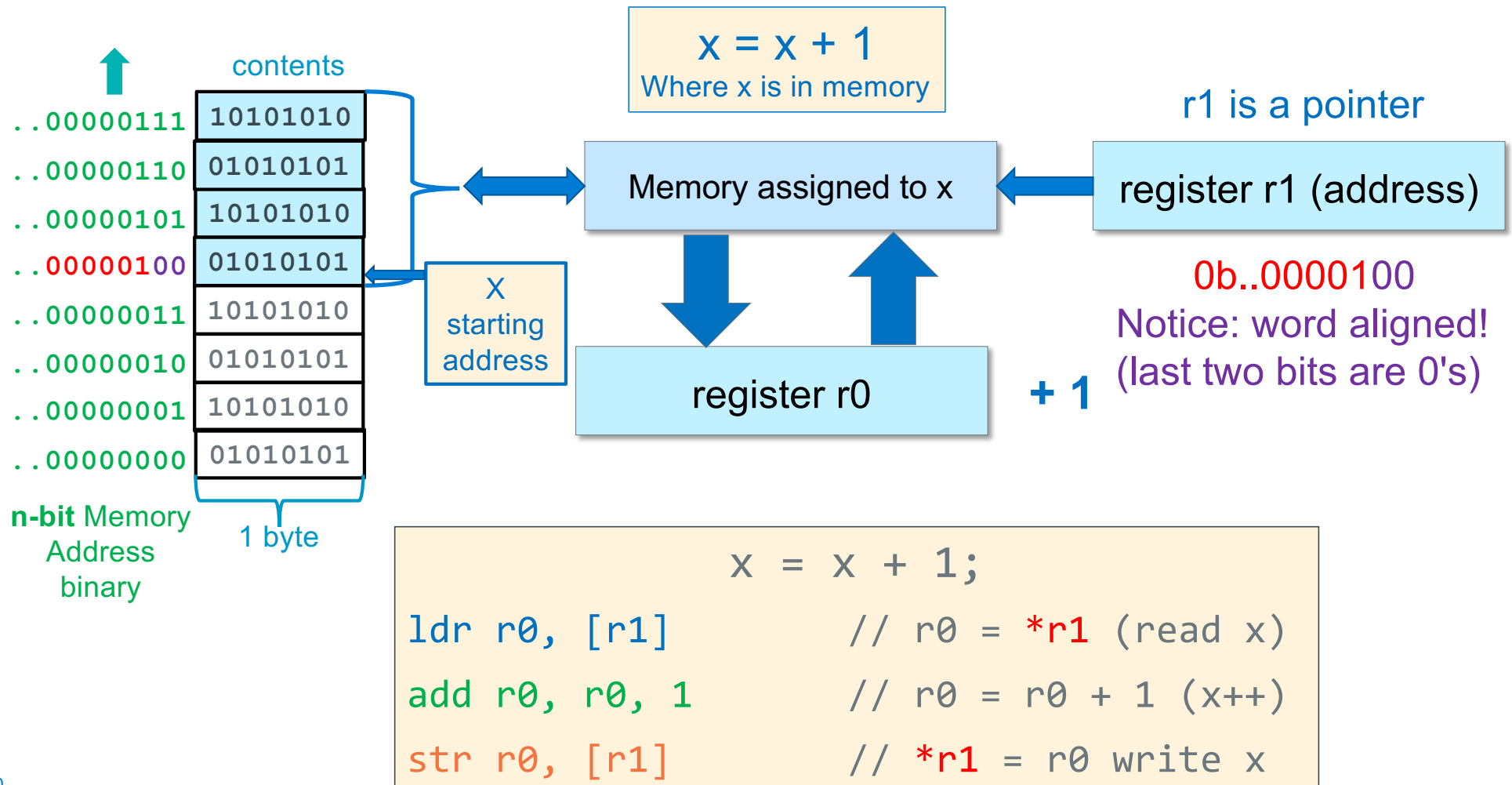
ldr/str  Rd,  [Rn]             // base register pointer + 0 offset (imm12 is 0)
```

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

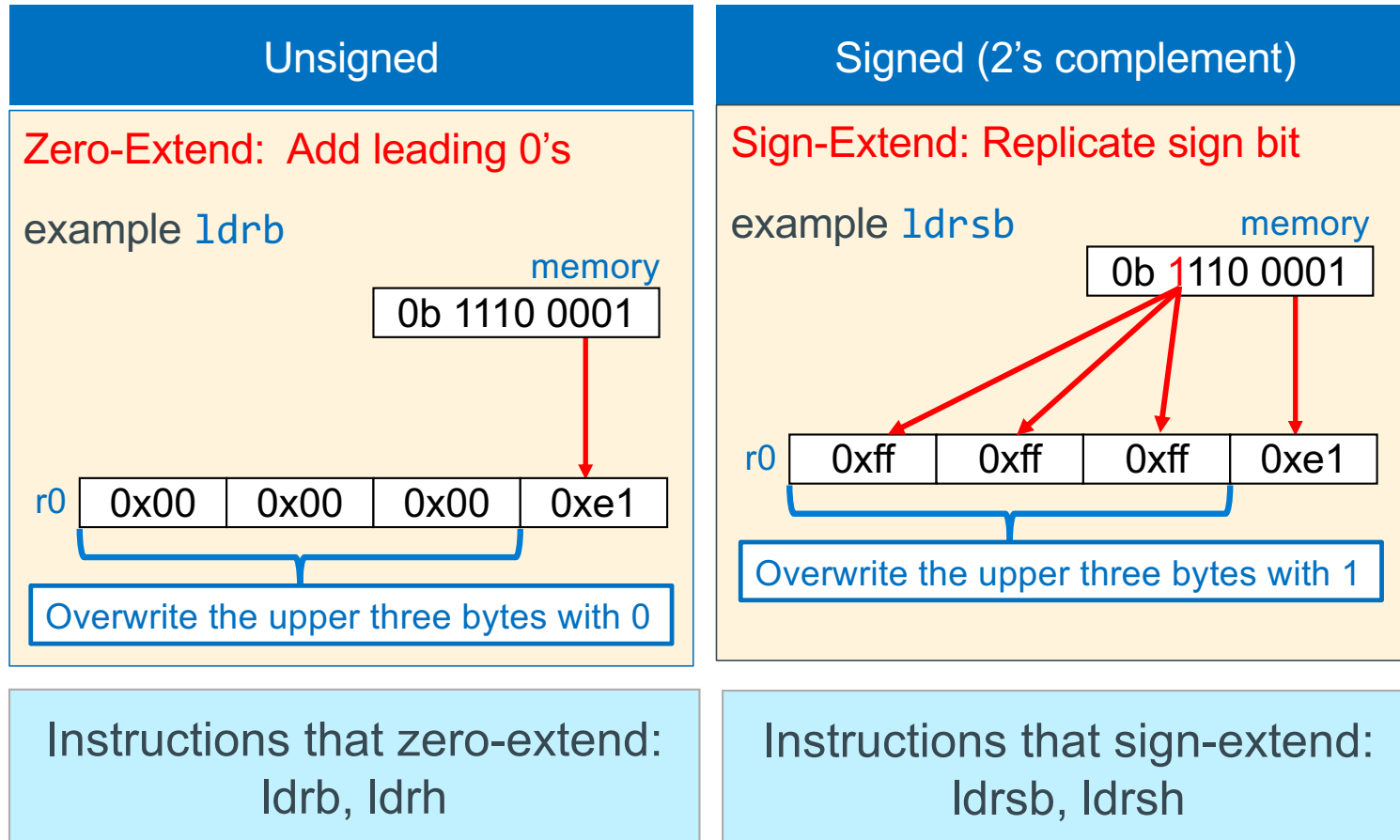


Loading and Storing: Variations List

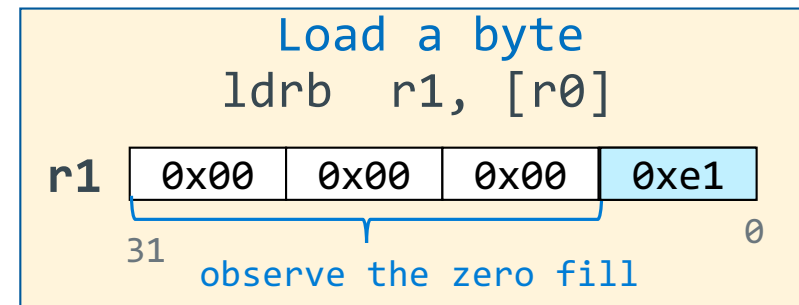
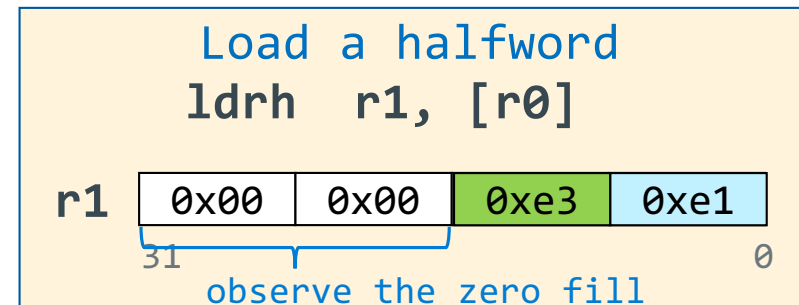
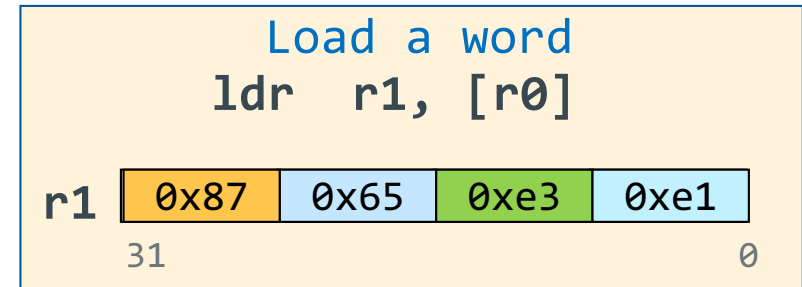
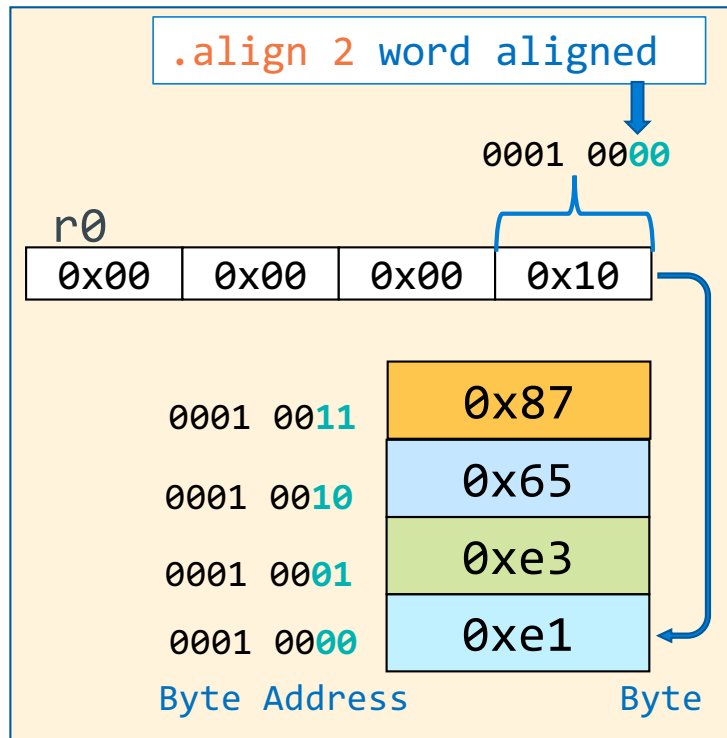
- Load and store have **variations** that move 8-bits, 16-bits and 32-bits
- Load into a register with less than 32-bits will **set the upper bits not filled from memory differently depending** on which **variation of the load instruction** is used
- Store will only select the lower 8-bit, lower 16-bits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

Instruction	Meaning	Sign Extension	Memory Address Requirement
ldrsb	load signed byte	sign extension	none (any byte)
ldrb	load unsigned byte	zero fill (extension)	none (any byte)
ldrsh	load signed halfword	sign extension	halfword (2-byte aligned)
ldrh	load unsigned halfword	zero fill (extension)	halfword (2-byte aligned)
ldr	load word	---	word (4-byte aligned)
strb	store low byte (bits 0-7)	---	none (any byte)
strh	store halfword (bits 0-15)	---	halfword (2-byte aligned)
str	store word (bits 0-31)	---	word (4-byte aligned)

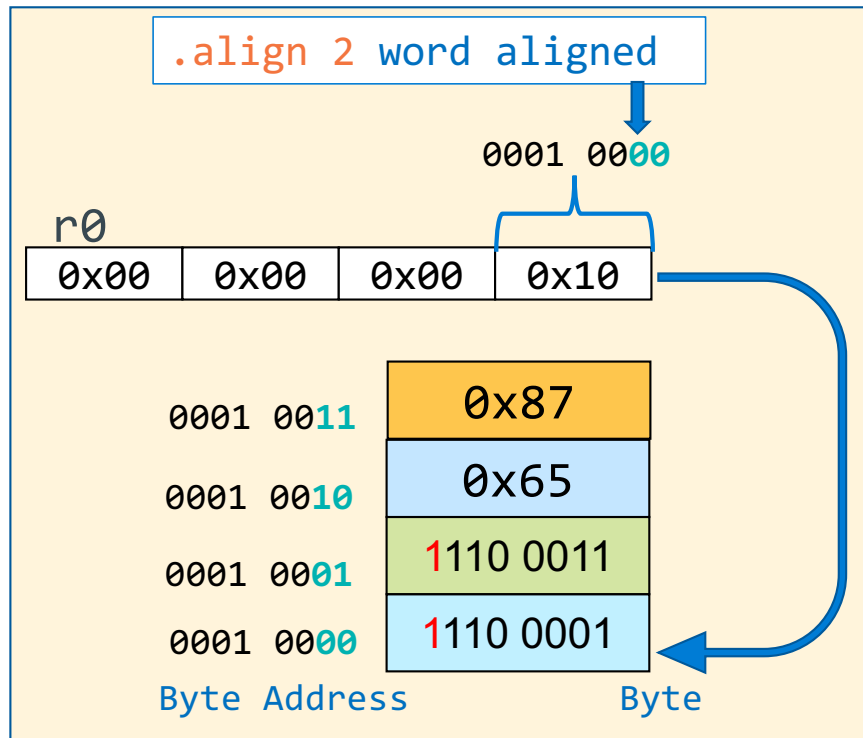
Loading 32-bit Registers From Memory Variables < 32-Bits Wide



Load a Byte, Half-word, Word

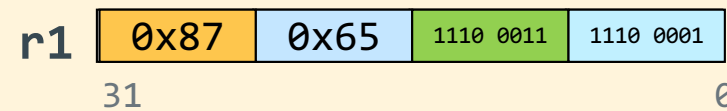


Signed Load a Byte, Half-word, Word



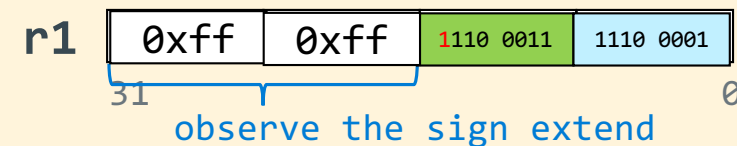
Load a word (no change)

```
ldr r1, [r0]
```



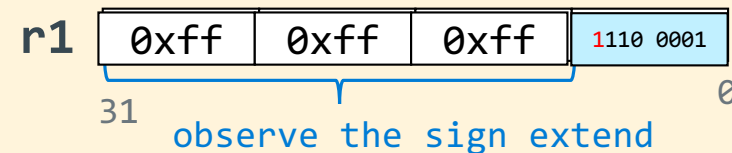
Load a halfword

```
ldrsh r1, [r0]
```

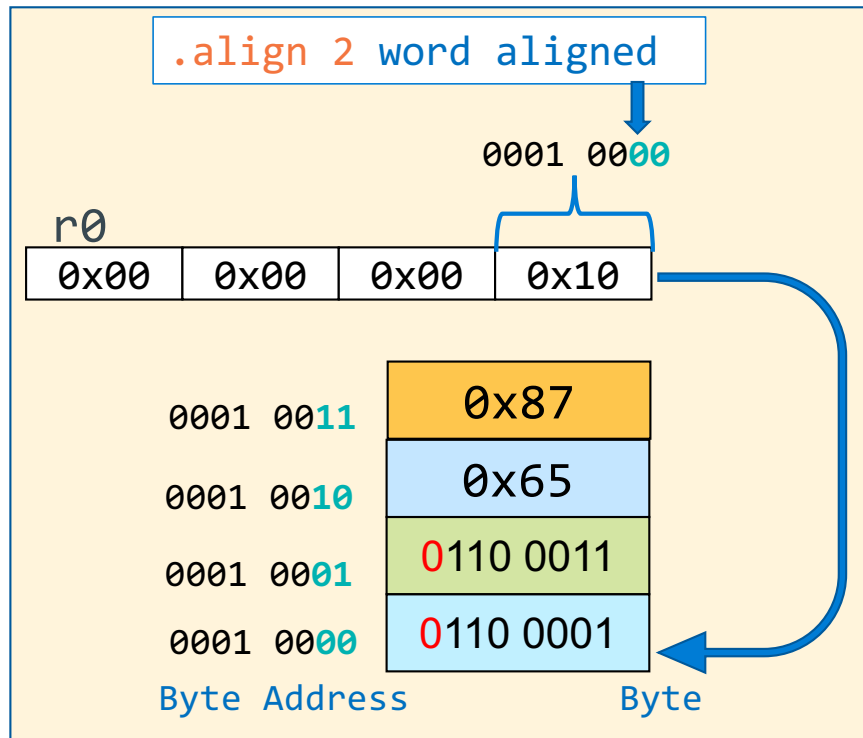


Load a byte

```
ldrshb r1, [r0]
```

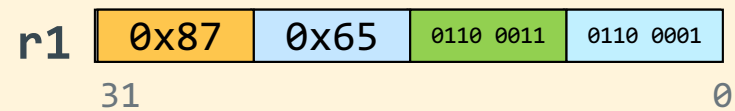


Signed Load a Byte, Half-word, Word



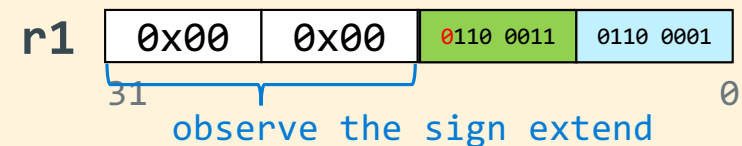
Load a word (no change)

ldr r1, [r0]



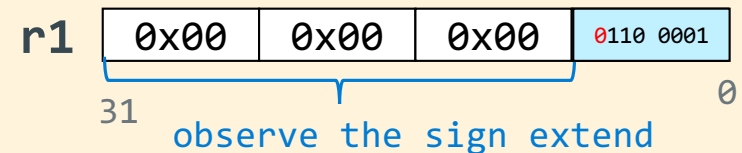
Load a halfword

ldrsh r1, [r0]

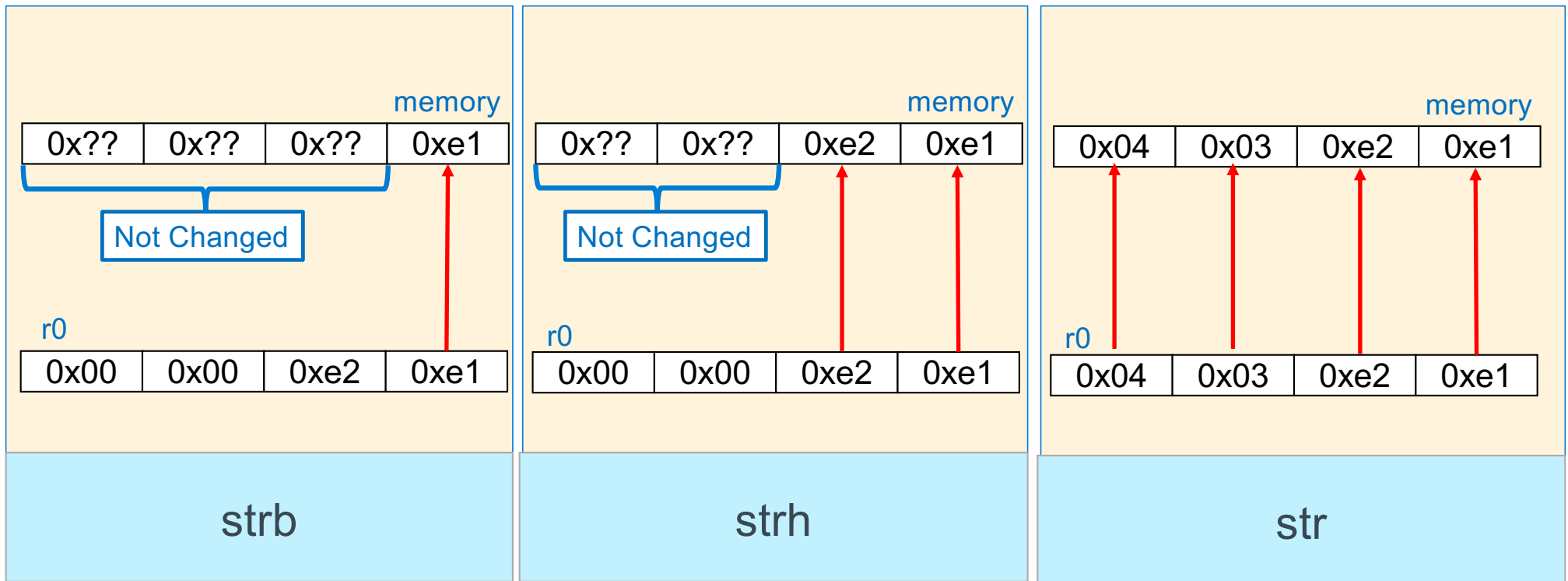


Load a byte

ldrsb 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: 31 | 0x87 | 0x65 | 0xe3 | 0xe1 | 0

Byte Address | Byte

0x20000003	0x33	observe other bytes NOT altered
0x20000002	0x22	
0x20000001	0x11	
0x20000000	0xe1	

Store a halfword
`strh r1, [r0]`

r1: 31 | 0x87 | 0x65 | 0xe3 | 0xe1 | 0

Byte Address | Byte

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

Store a word
`str r1, [r0]`

r1: 31 | 0x87 | 0x65 | 0xe3 | 0xe1 | 0

Byte Address | Byte

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