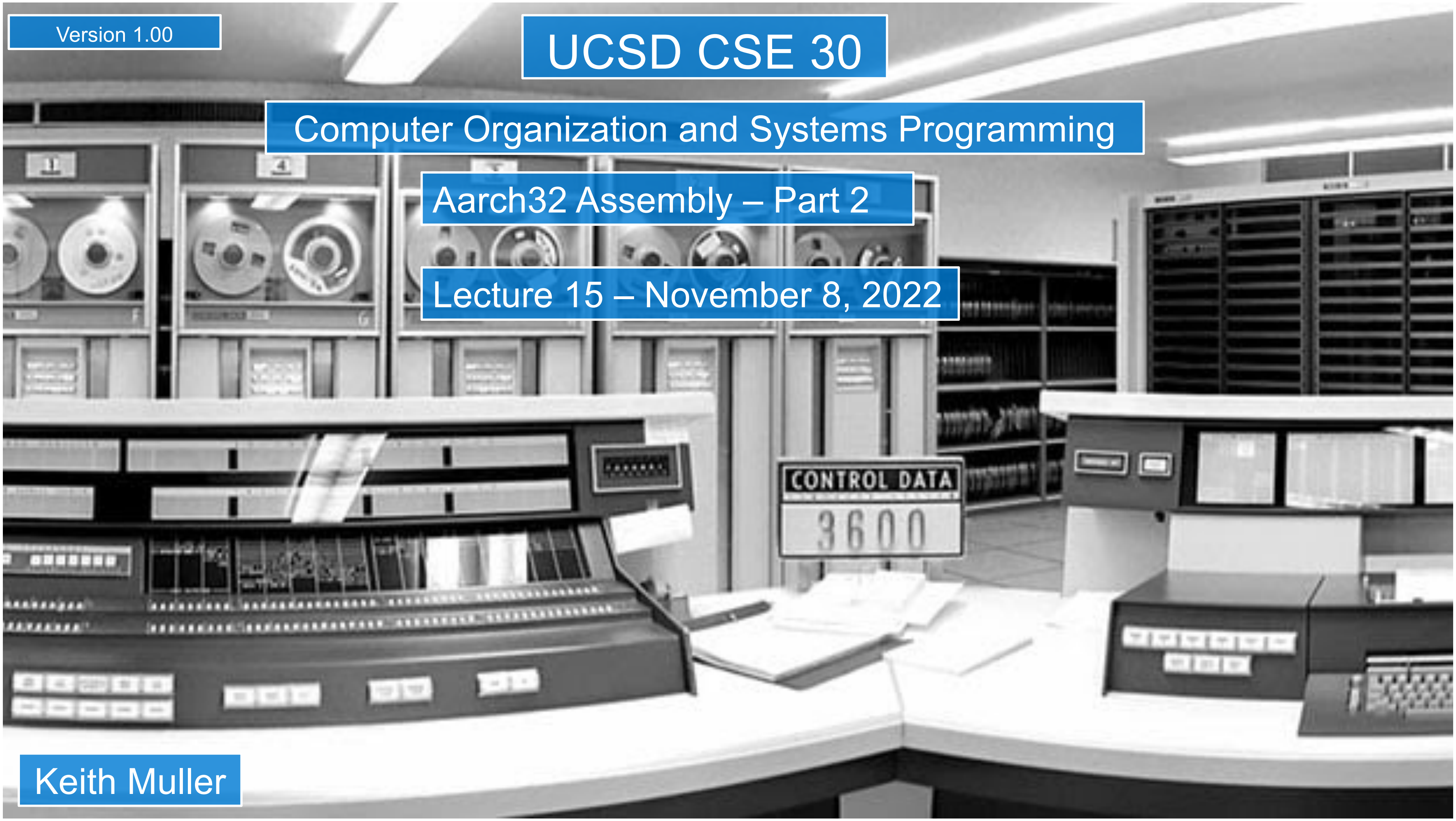
Version 1.00

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

```
        cmp     r3, r4      <- Branch guard
        beq     .Ldone
        // otherwise fall through do the add
        add     r1, r2, 1
        // now fall through to mov
.Ldone:
        mov     r3, r1
```

```
if (r3 != r4)
    r1 = r2 + 1;
r3 = r1;
```

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 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

x

Examples: Guards (Conditional Tests) and their Inverse

`Compare in C`	*"Inverse"* `Compare in C`
`==`	`!=`
`!=`	`==`
`>`	`<=`
`>=`	`<`
`<`	`>=`
`<=`	`>`

x

Conditional Branch: Changing the Next Instruction to Execute

cond	b	imm24

Branch instruction

```
bsuffix .Llabel
```

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

Condition	*Meaning*	*Flag Checked*
BEQ	Equal	Z = 1
BNE	Not equal	Z = 0
BGE	Signed ≥ ("Greater than or Equal")	N = V
BLT	Signed < ("Less Than")	N ≠ V
BGT	Signed > ("Greater Than")	Z = 0 && N = V
BLE	Signed ≤ ("Less than or Equal")	Z = 1 || N ≠ V
BHS	Unsigned ≥ ("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 ≤ ("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)	

x

Eliminate unnecessary branches and labels: use Fall Throughs

```
        mov   r2, 0
        b       .Lnext // not needed, slows code
.Lnext:
        add   r1, r2, r3
```

```
        mov   r2, 0
        add   r1, r2, r3
```

Use fall-through!
do not branch to a Label on the **next statement**!

x

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
```


x

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 <u>inverse</u> of the condition used in C**

C source Code	*Incorrect Assembly*	*Correct Assembly*

`int r0;`
`if (r0 == 5) {`
`    //code`
`}`
	`    cmp r0, 5`
	`    beq .Lendif`
	`    //code`
	`.Lendif:`
		`    cmp r0, 5`
		`    bne .Lendif`
		`    // code`
		`.Lendif:`

```
int r0;
if (r0 == 5) {
    /* code */
    /* then fall through */
}
/* branch around to this code */
```

If r0 == 5 true
then ***fall through*** to
the true block

```
    cmp r0, 5
    bne .Lendif
    /* code */
    /* then fall through */
.Lendif:
/* branch around to this code */
```

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

x

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:
```


x

When do you use a Signed or Unsigned Conditional Branch?

example with 4 bits
(arm is 32)

$0111_2 > 1100_2$?

unsigned

Compare
7 > 12 ?
(FALSE)

signed
2's complement

Compare
+7 > -4 ?
(TRUE)

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

x

If statement examples – Branch Around the True block!

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

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

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

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

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

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

```
        cmp     r0, r1
        bls     .Lendif
        mov     r1, r0
.Lendif:                    Fall through
        mov     r3, r2
```


x

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
```


x

Anatomy of a Conditional Branch: If - Else statement

Branch condition Test (branch guard)

```c
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)

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

condition true block

condition false block

x

Program Flow: If with an Else

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

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

```
    cmp r0, 5
    bne .Lelse
    /* cond. true block */

    /*
     * Now branch around the
     * condition false block
     */
    b .Lendif
.Lelse
    // condition false block
    // fall through
.Lendif:
    mov r1, 4
```

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

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

x

If with an Else Examples

Branch condition Test (branch guard)

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

If r0 < r1 false then branch *to* false block

```
       cmp r0, r1
       bge .Lelse

       mov r2, 1
       b .Lendif
.Lelse:
       mov   r2, 0
      /* fall through */

.Lendif:
       add   r4, r2, r4
```

condition true block

condition false block

x

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;
```

Same test
swapped blocks

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

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


x

Switch Statement

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

Approach 1 – *Branch Block*

```
    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:
```


x

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

```
        cmp r3, r4
        blt .Lendif
        beq .Lelse1
        bgt .Lelse2
.Lelse1:
        mov   r2, 0
        b .Lendif
.Lelse2:
        add r4, r3, r2
        b .Lesle1
.Lendif:
        add   r4, r2, r4
```

eliminate this branch replace with fall through

Upwards branch

eliminate this branch

```
        cmp r3, r4
        blt .Lendif
        beq .Lelse1
        add r4, r3, r2
.Lelse1:
        mov   r2, 0
.Lendif:
        add   r4, r2, r4
```


x

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();
```


x

Program Flow – If statements && compound tests - 1

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

```
    cmp r0, 5
    bne .Lendif

    cmp r1, 3
    ble .Lendif
    mov r2, r5 // true block
    // fall through
.Lendif:
    mov r4, r3
```

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

x

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;
```

```
    cmp r0, 5  // test 1
    bne .Lelse

    cmp r1, 3  // test 2
    ble .Lelse

    mov r2, r5 // true block
    // branch around else
    b .Lendif
.Lelse:
    mov r5, r2 //false block
    // fall through
.Lendif:
   mov r4, r3
```

if r0 == 5 false then short circuit branch ***to*** the false block

if r1 > 3 false then branch ***to*** the false block

x

Program Flow – If statements || compound tests - 1

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

```
        cmp r0, 5
        beq .Lthen

        cmp r1, 3
        ble .Lendif
        // fall through
.Lthen:
        mov r2, r5 // true block
        /* fall through */
.Lendif:
        mov r4, r3
```

If r0 == 5 true, then branch ***to*** true block

if r1 > 3 false then branch ***around*** true block

x

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 */
}
```

```
    cmp r0, 5
    beq .Lthen
    cmp r1, 3
    ble .Lelse
    // fall through
.Lthen:
    mov r2, r5 // true block
    // branch around else
    b .Lendif
.Lelse
    mov r5, r2 // false block
    // fall through
.Lendif:
```

If r0 == 5 true, then branch *to* the true block

if r1 > 3 false then branch *to* false block

x

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 loop guard

loop control variable

zero or more iterations

```
while (i < 10) {
    /* block */
    i++;
}
```

one or more iterations

```
do {
    /* block */
    i++;
} while (i < 10);
```

post-test loop guard

Pre-Test Guards - While Loop

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

pre-test loop guard

loop iteration

inverted test

```
.Lwhile:
    cmp r1, 10
    bge .Lendw
    // block
    add r1, r1, 1
    b .Lwhile
.Lendw:
    mov r2, r1
```

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

pre-test loop guard

if stmt guard

loop iteration

```
.Lwhile:
    cmp r1, 10
    bge .Lendw
    cmp r2, r1
    beq .Lnext
    // block
.Lnext:
    add r1, r1, 1
    b .Lwhile
.Lendw:
    mov r2, r1
```


x

Post-Test Guards – Do While Loop

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

r2 = r1;
```

loop iteration

post-test loop guard

```
.Ldo:
    // block
    add r1, r1, 1
    cmp r1, 10
    blt .Ldo

    mov r2, r1
```

test is not inverted

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

r2 = r1;
```

loop iteration

post-test loop guard

```
.Ldo:
    cmp r2, r1
    beq .Lnext
    // block
.Lnext
    add r1, r1, 1
    cmp r1, 10
    blt .Ldo

    move r2, r1
```


x

Program Flow – Counting (For) Loop

Pre loop setup | Pre-test Loop guard | Post loop update

```
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

- 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 → `mov r1, 0`
Pre-test loop guard → `cmp r1, 10` / `bge .Lendfr`
Post loop update → `add r1, r1, 1`

```
    mov r1, 0
.Lfor:
    cmp r1, 10
    bge .Lendfr
    // block code
    add r1, r1, 1
    b .Lfor
.Lendfr:
```

Alternative

Pre-loop setup → `mov r1, 0`
Pre-test loop guard → `cmp r1, 10` / `bge .Lendfr`
Post loop update → `add r1, r1, 1`
Post-test loop guard → `cmp r1, 10` / `blt .Ldo`

```
    mov r1, 0
    cmp r1, 10
    bge .Lendfr
.Ldo:
    // block code
    add r1, r1, 1
    cmp r1, 10
    blt .Ldo
.Lendfr:
```


x

Nested loops

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

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

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

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

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

    mov r0, 0

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

    cmp r1, 10   // loop guard
    blt .Ldo

    // fall through
    add r2, r2, r1

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


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**

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:
```


x

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 " "

x

Assembly Source File Template

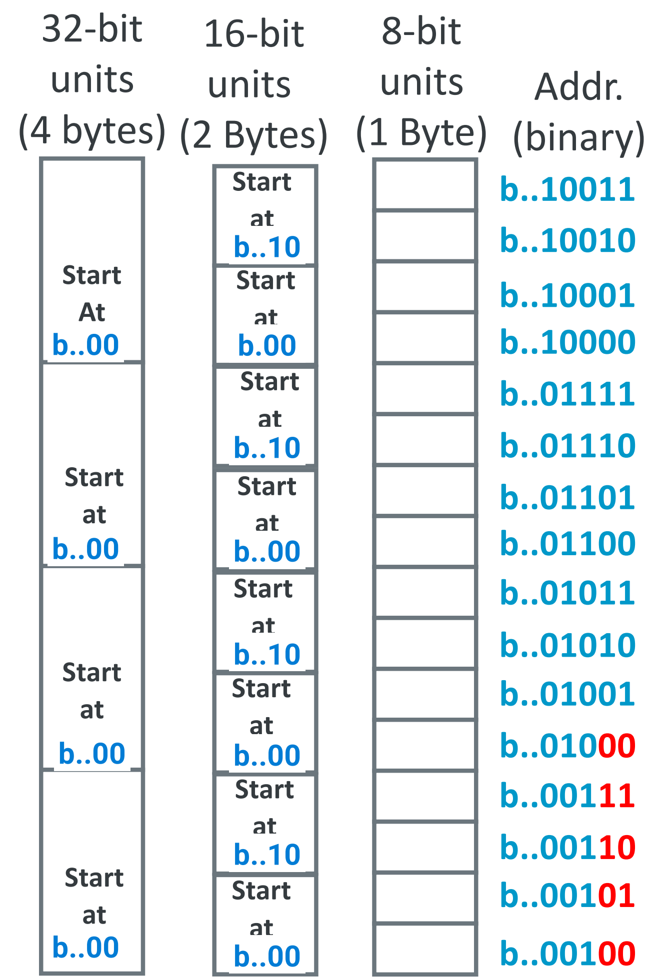
```
// File Header
        .arch armv6              // armv6 architecture instructions
        .arm                     // arm 32-bit instruction set
        .fpu vfp                 // floating point co-processor
        .syntax unified          // modern syntax

// 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

x

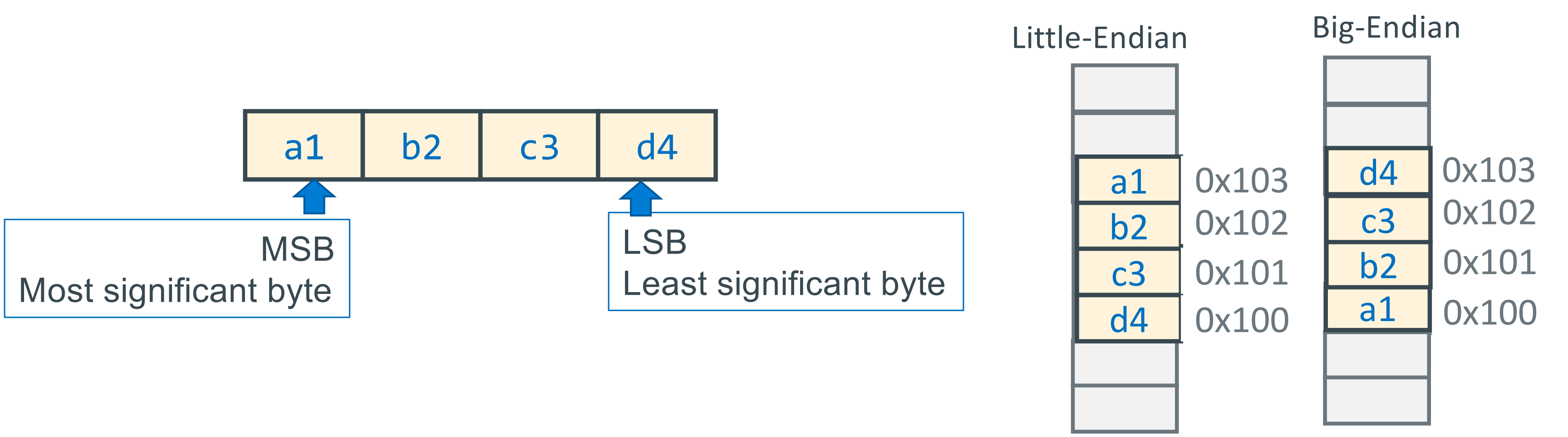
Memory Segment Data Alignment

- **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



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

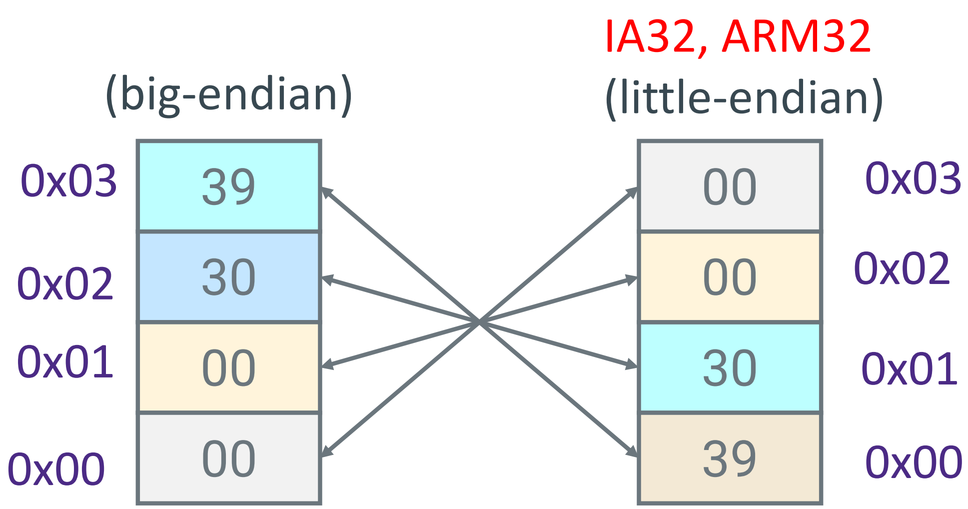


x

Byte Ordering Example

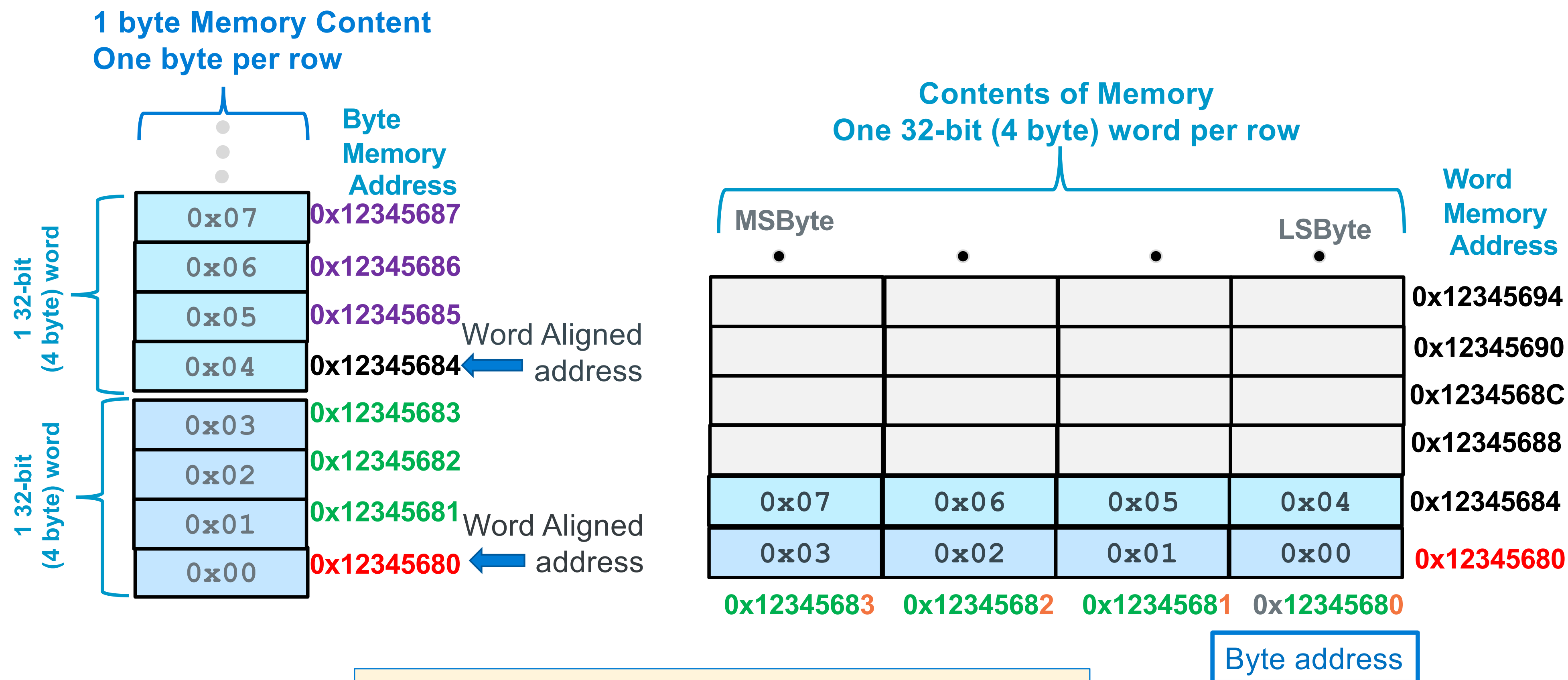
```
Decimal:   12345
Binary:     0011  0000  0011  1001
Hex:           3     0     3     9
```

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



x

Byte Addressable Memory Shown as 32-bit words



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

x

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

x

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`

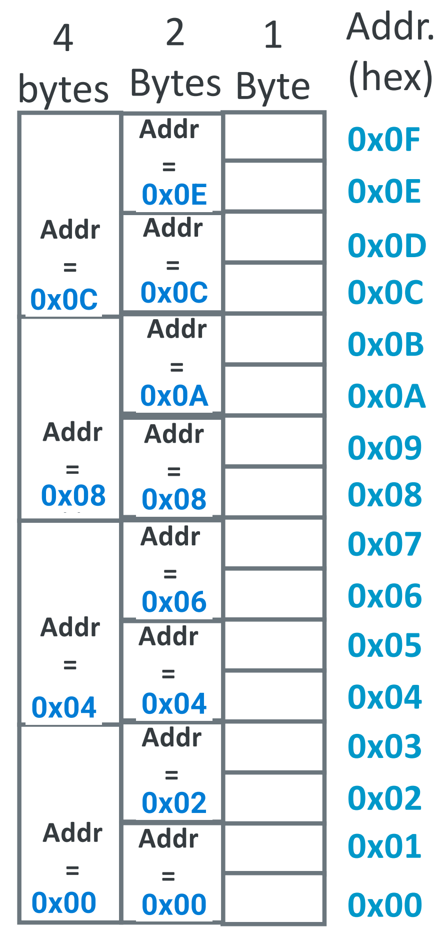
- Allocates `size` bytes, each of which contain the value `fill`
- Both `size` and `fill` are absolute expressions
- If the comma and `fill` are **omitted**, `fill` is assumed to be **zero**
- **`.bss section:`** Must be used **without a specified fill**

x

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



`.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

x

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 al1.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 al1.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 al1.S
  1                .data
  2 0000 02000000 xx:     .word 2
  3 0004 41424344 ch:     .byte 'A','B','C','D','E'
  3      45
  4 0009 000000           .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'
```


X

Load/Store: Register Base Addressing

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

32-bit memory ← register r1 (address)

↓

register r0

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

r1 is being used as a **<u>pointer</u>** to a location in memory

ldr requires the use of a <u>pointer</u> operand

```
str  r0, [r1]
```

register r0

↓

32-bit memory ← register r1 (address)

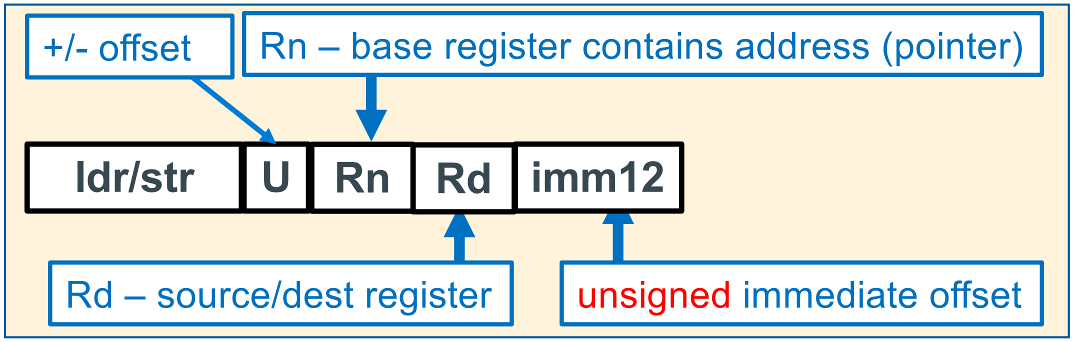
Copies all 32 bits of the value held in register r0 to the 32-bit memory location contained in register r1 (r1 pointer)

r1 is being used as a **<u>pointer</u>** to a location in memory

str requires the use of a <u>pointer</u> operand

x

LDR/STR – Base Register + Immediate Offset Addressing

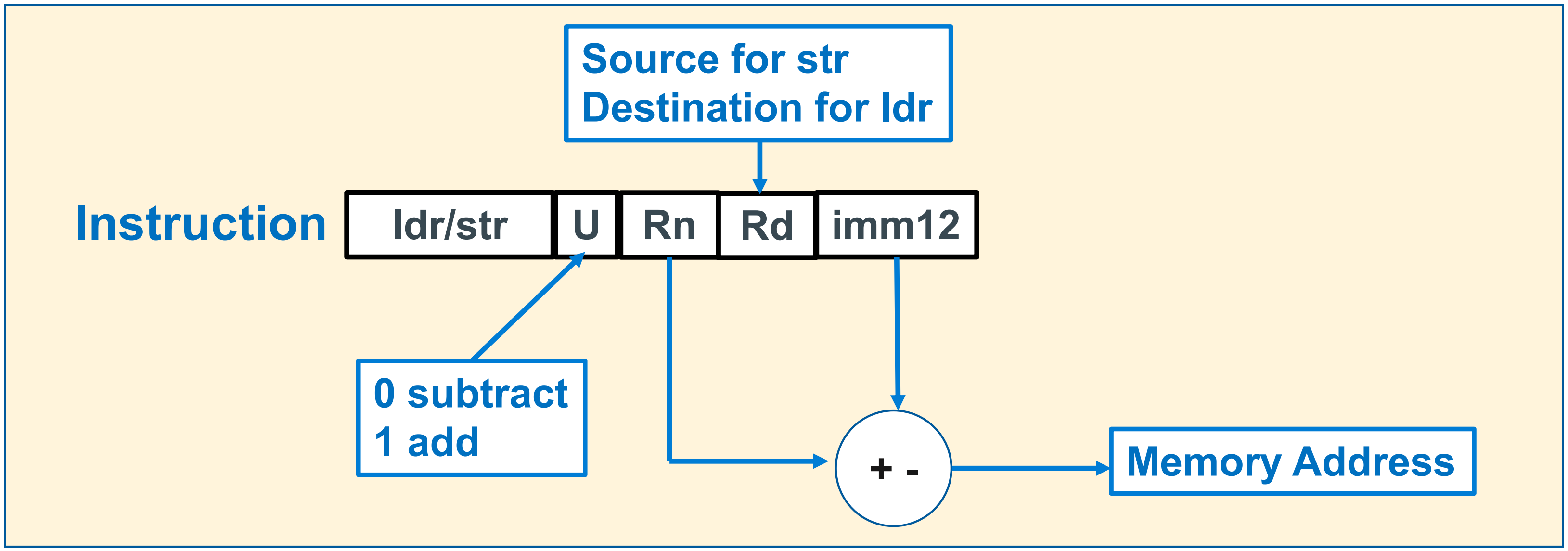


- **Register Base Addressing**:
  - Pointer Address: Rn; source/destination data: Rd
  - **Unsigned pointer address** in stored in the base register
- **Register Base + immediate offset Addressing:**
  - Pointer Address = register content + immediate offset
  - Unsigned offset integer immediate value (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)
```


x

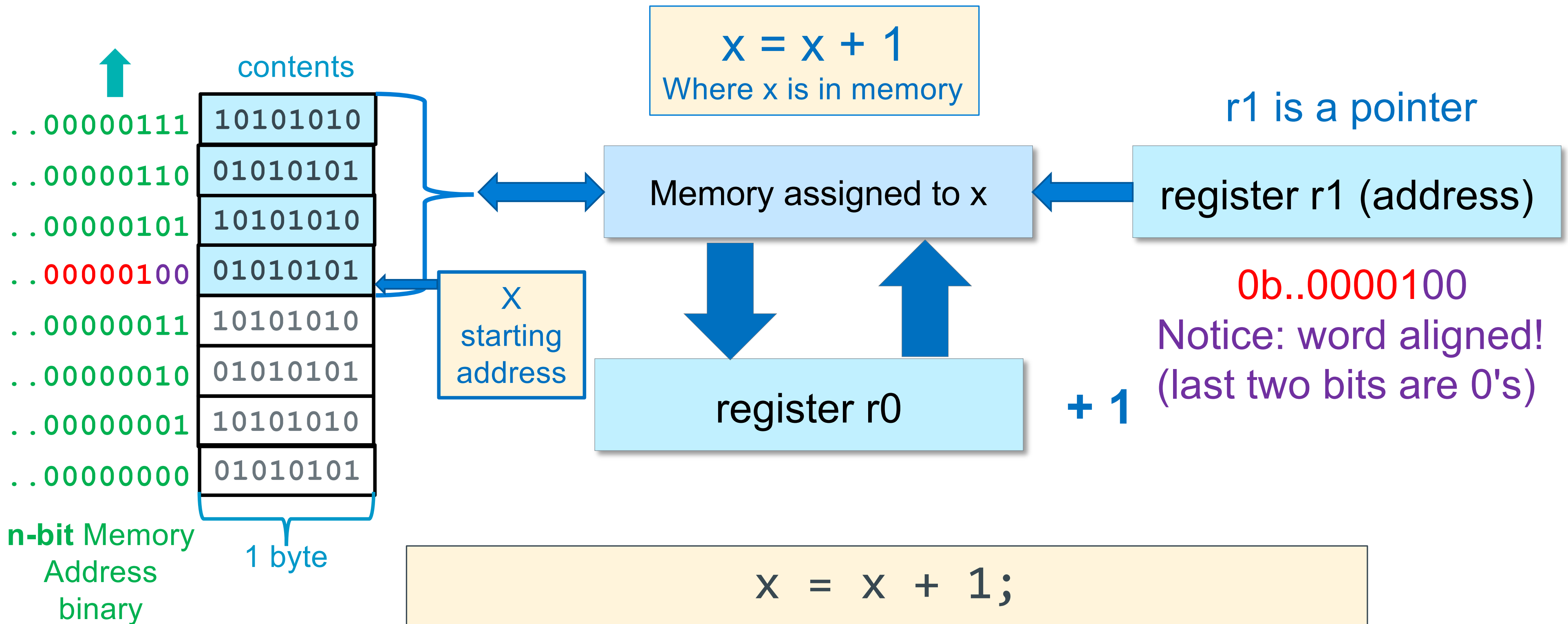
ldr/str Register Base and Register + Immediate Offset Addressing



Syntax	Address	Examples
ldr/str Rd, [Rn +/- constant] constant is in bytes	Rn + or – constant same	ldr r0, [r5,100] str r1, [r5, 0] str r1, [r5]

x

Example Base Register Addressing Load – Modify – Store



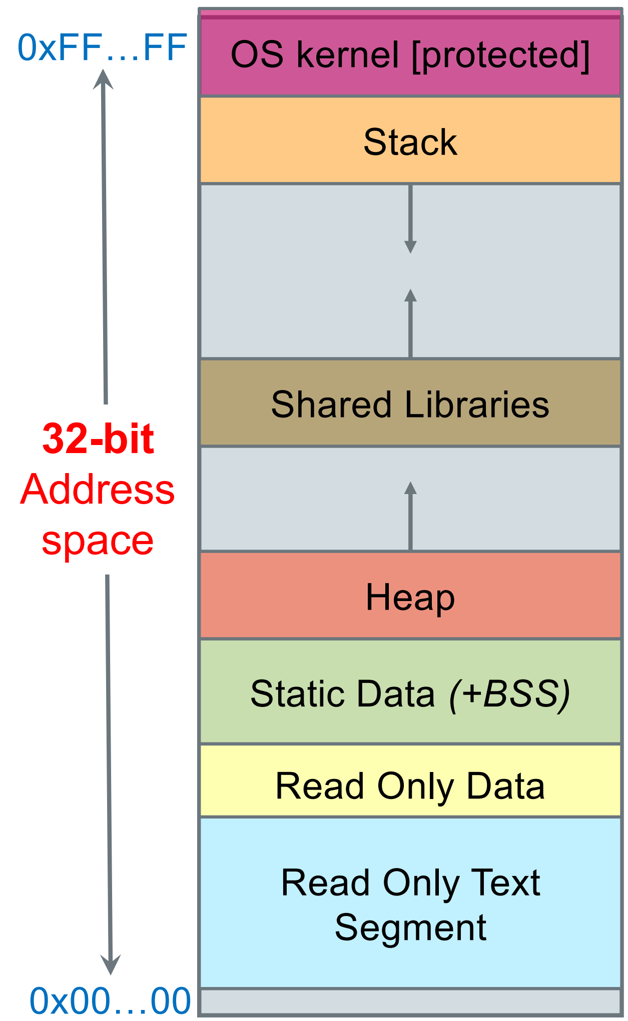

```
x = x + 1;
ldr r0, [r1]          // r0 = *r1 (read x)
add r0, r0, 1         // r0 = r0 + 1 (x++)
str r0, [r1]          // *r1 = r0 write x
```


x

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 4ç
       .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
```



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

1. load the pointer to the memory
2. read (load) from *pointer

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

1. load the pointer to the memory
2. 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
…
```


x

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?

mov	Rd	rot4	imm8

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
```


x

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
`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)

x

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

Unsigned

Zero-Extend: Add leading 0's

example `ldrb`

memory

0b 1110 0001

r0				
r0	0x00	0x00	0x00	0xe1

Overwrite the upper three bytes with 0

Signed (2's complement)

Sign-Extend: Replicate sign bit

example `ldrsb`

memory

0b 1110 0001

r0				
r0	0xff	0xff	0xff	0xe1

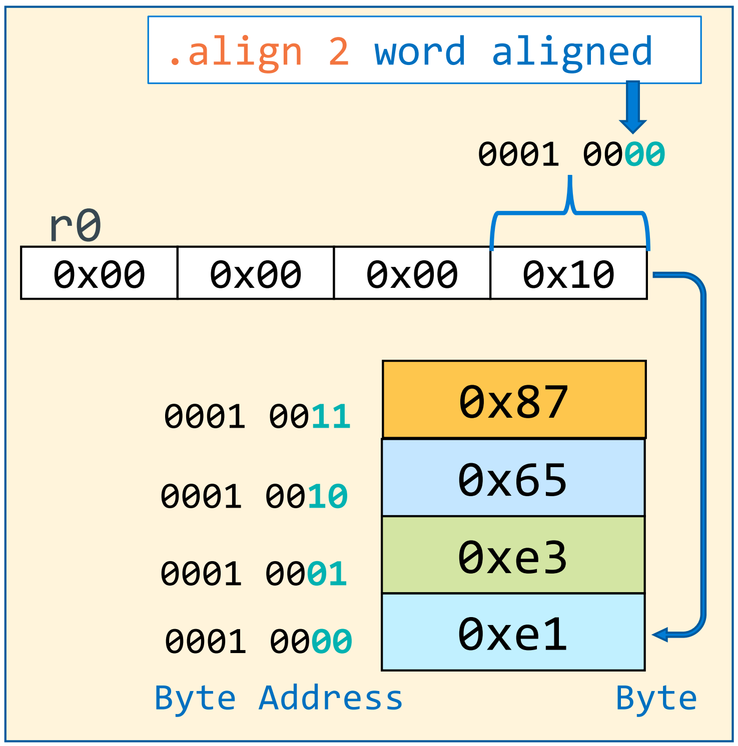
Overwrite the upper three bytes with 1

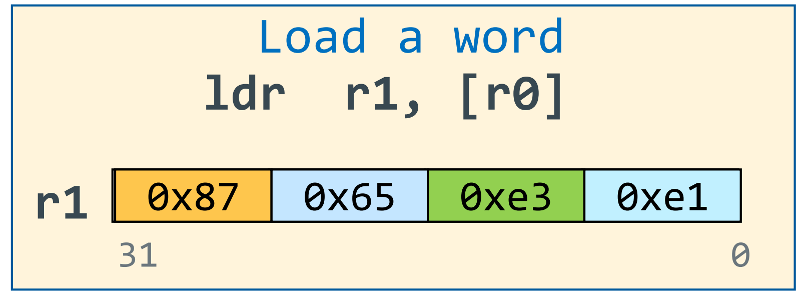
Instructions that zero-extend:
ldrb, ldrh

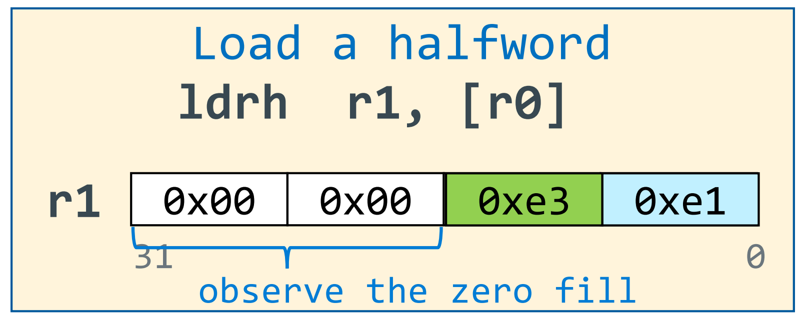
Instructions that sign-extend:
ldrsb, ldrsh

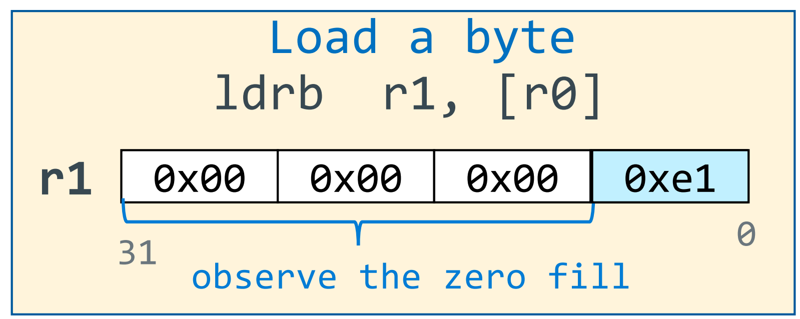
x

Load a Byte, Half-word, Word



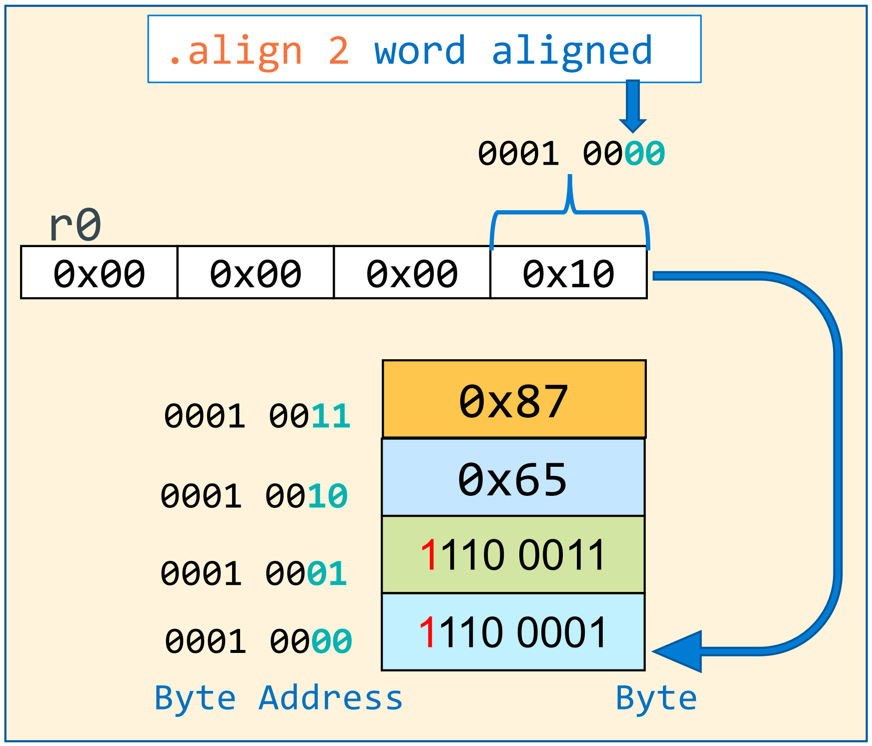


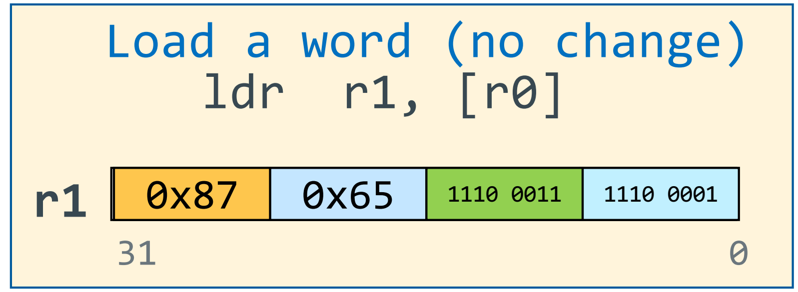


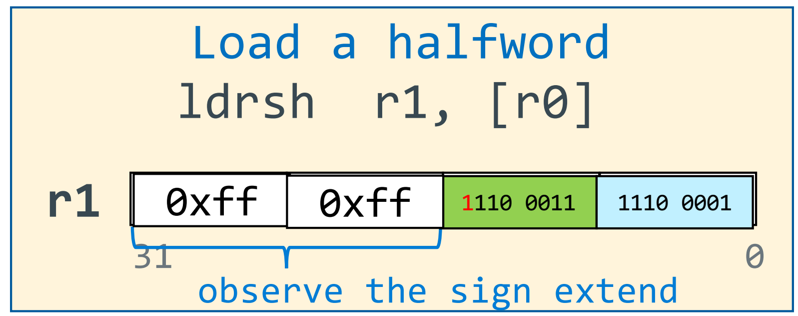


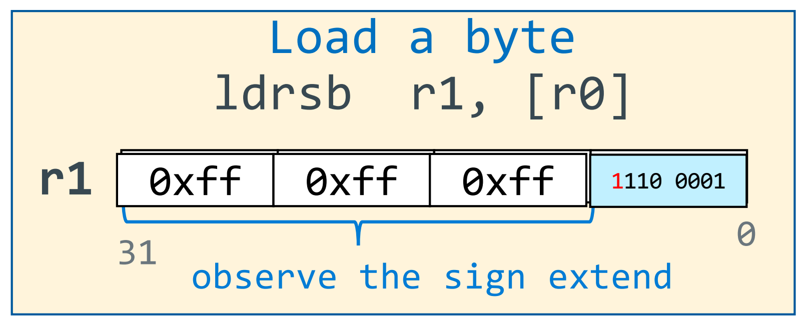
x

Signed Load a Byte, Half-word, Word



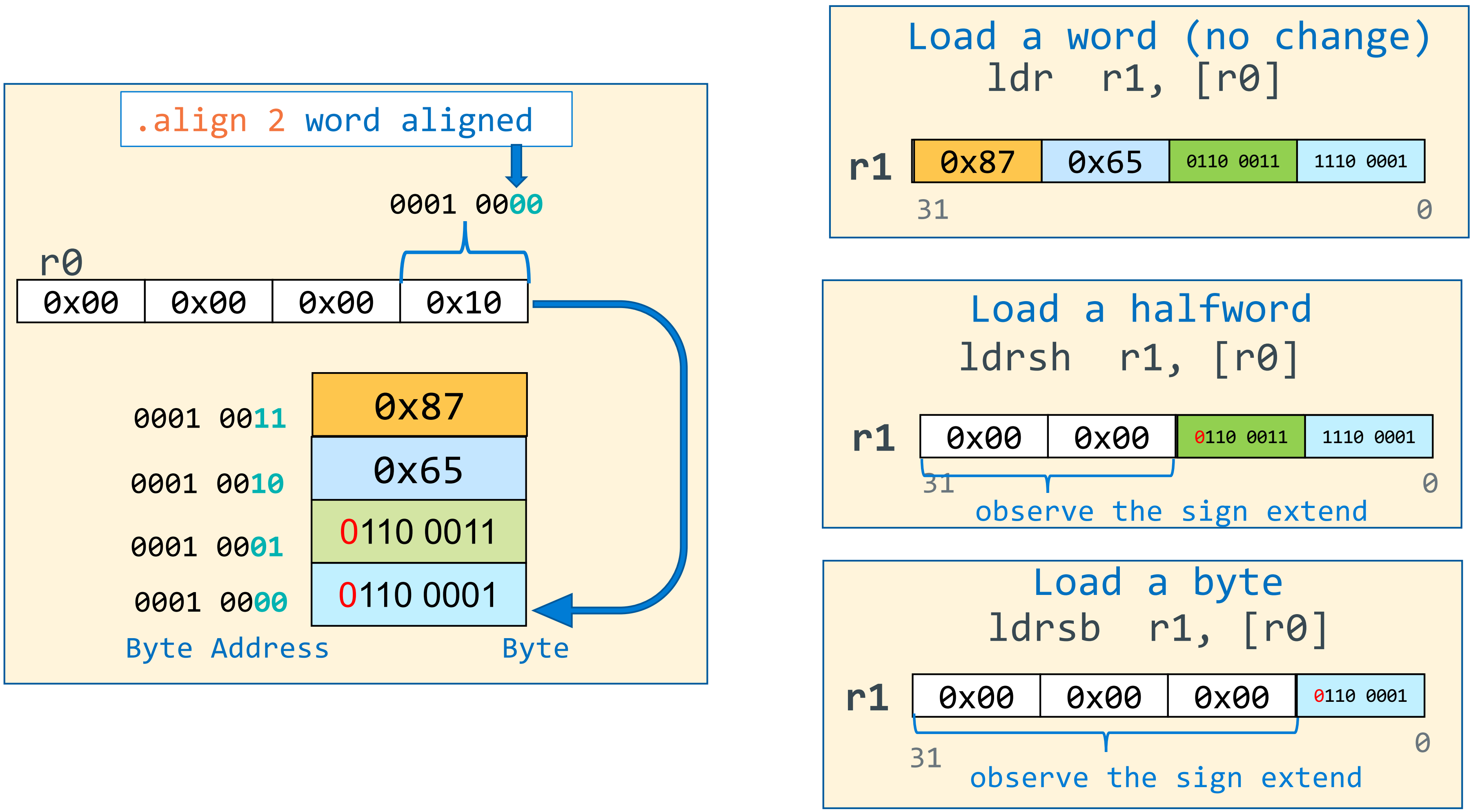






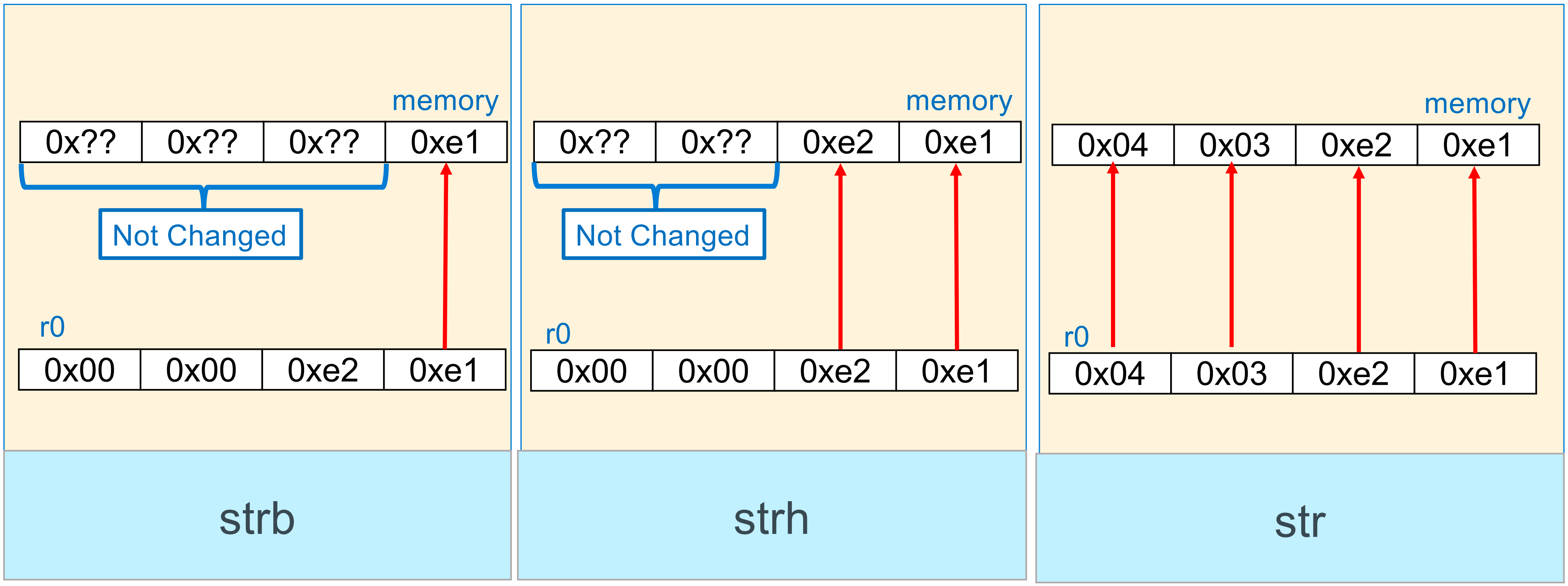
x

Signed Load a Byte, Half-word, Word



x

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



x

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]
```

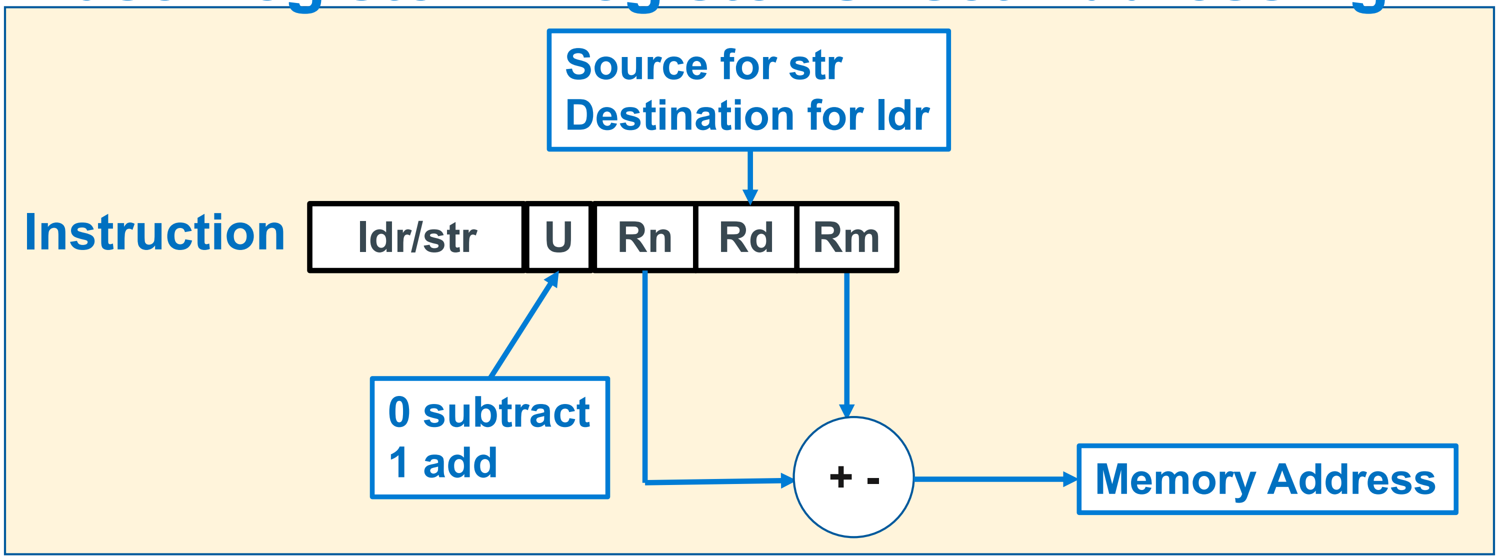
r1

0x87	0x65	0xe3	0xe1
31			0

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

x

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
`ldr/str Rd, [Rn +/- Rm ]`	`Rn + or – Rm`	`ldr r0, [r5, r4]` `str r1, [r5, r4]`

x

Reference: Addressing Mode Summary for use in CSE30

index Type	Example	Description
Pre-index immediate	`ldr r1, [r0]`	r1 ← memory[r0] r0 is unchanged
Pre-index immediate	`ldr r1, [r0, 4]`	r1 ← memory[r0 + 4] r0 is unchanged
Pre-index immediate	`str r1, [r0]`	memory[r0] ← r1 r0 is unchanged
Pre-index immediate	`str r1, [r0, 4]`	memory[r0 + 4] ← r1 r0 is unchanged
Pre-index register	`ldr r1, [r0, +-r2]`	r1 ← memory[r0 +- r2] r0 is unchanged
Pre-index register	`str r1, [r0, +-r2]`	memory[r0 +- r2] ← r1 r0 is unchanged

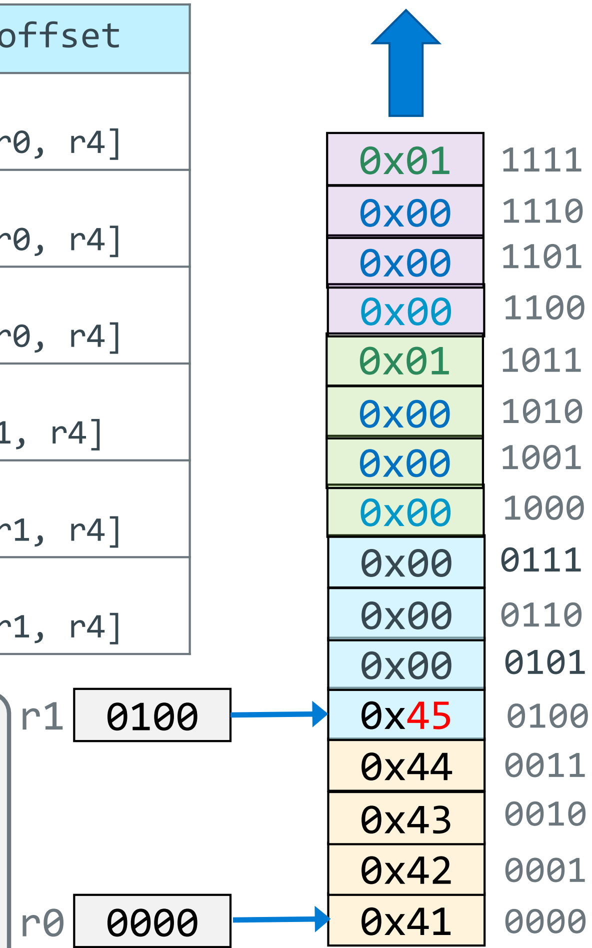
x

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
```

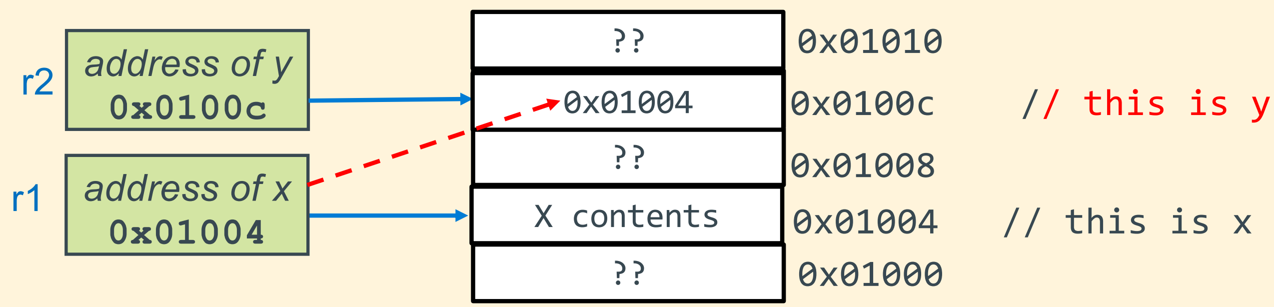


ldr/str practice - 1

`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
```

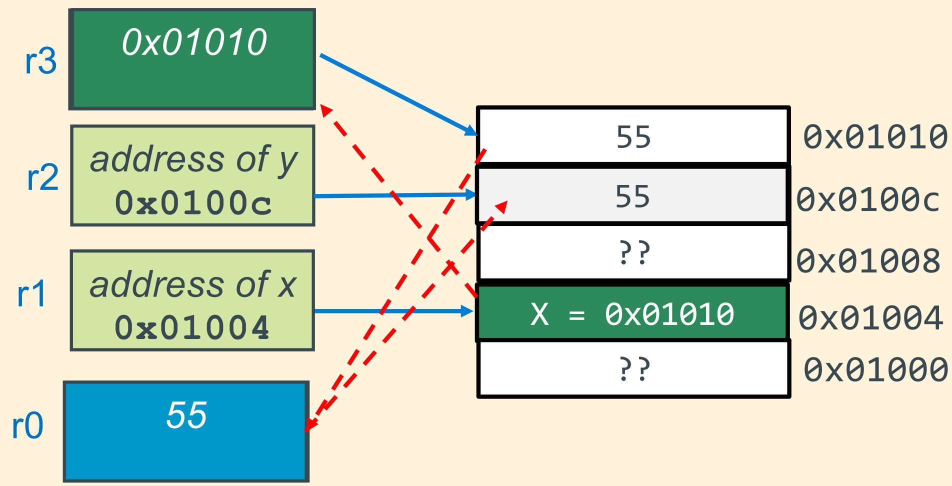

x

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;

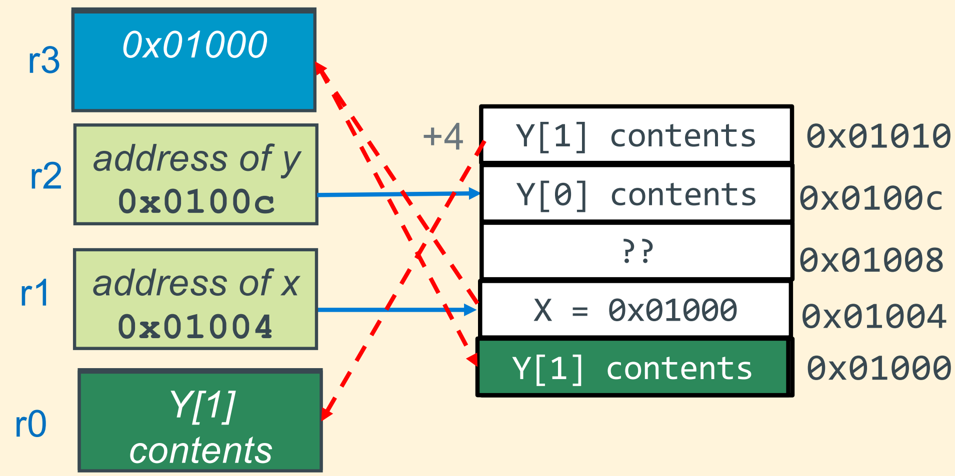



```
ldr   r3, [r1]  // r3 ← x (read 1)
ldr   r0, [r3]  // r0 ← *x (read 2)
str   r0, [r2]  // y ← *x
```


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];
```

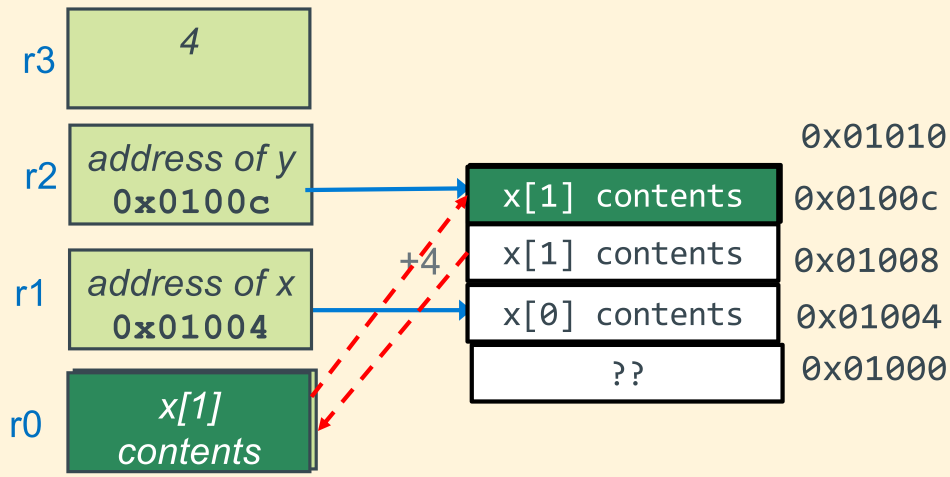



```
ldr    r0, [r2, 4]        // r0 ← y[1]

ldr    r3, [r1]           // r3 ← x

str    r0, [r3]           // *x ← y[1]
```


x

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]
```


x

Label (Address) Math

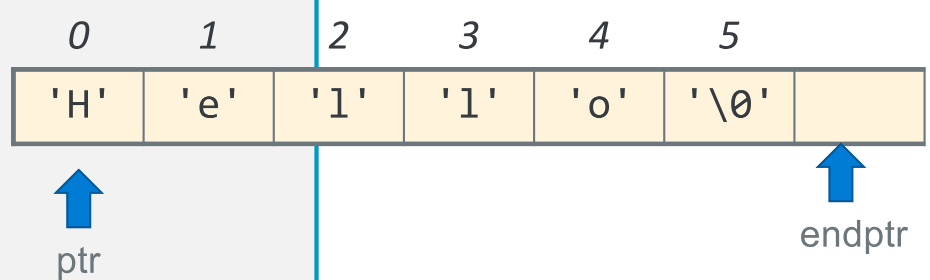
- You can have the assembler calculate some useful values for you
- One common use is calculating the distance in bytes between two labels
- The dot (.) refers to the address on the current line (the next byte after a previous space allocation)

```
       .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
```

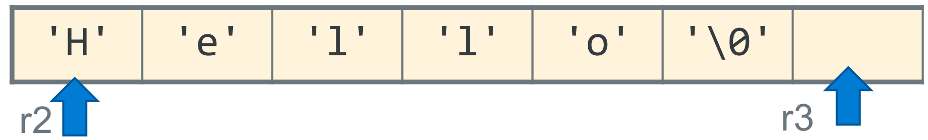

x

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
    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      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:
```


x

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

x

Example: Base Register + Register Offset Two Buffers

```c
#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;
}
```

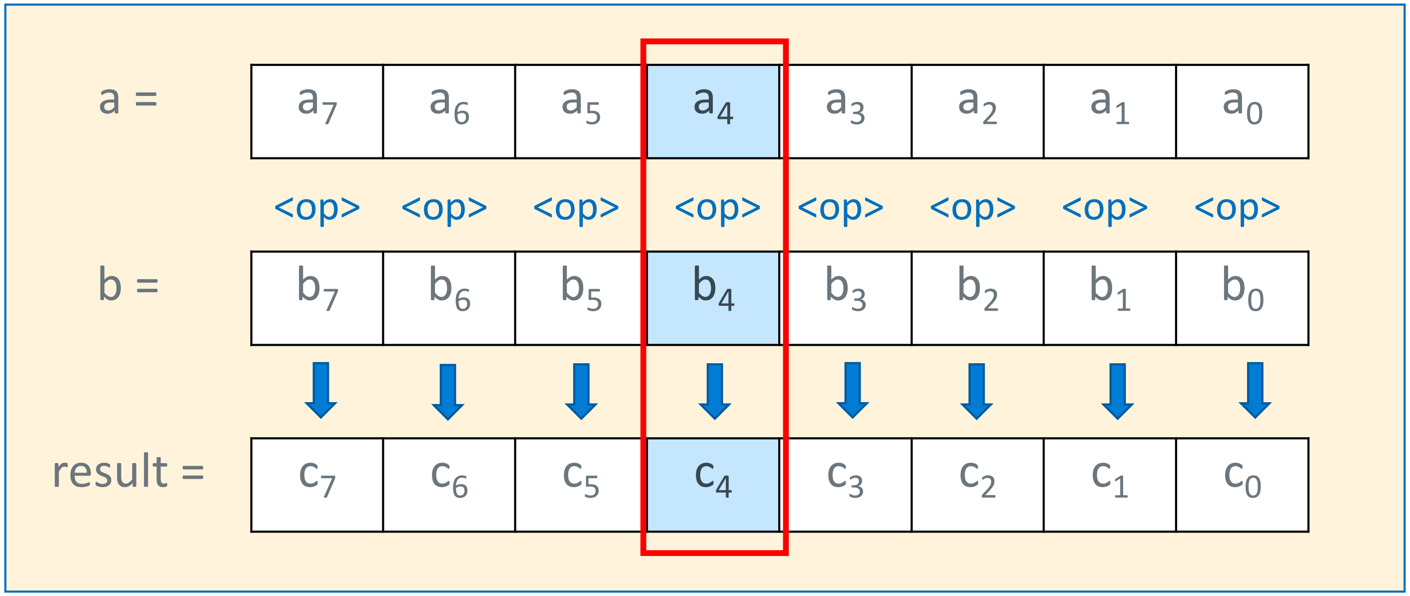
- Make sure to index by bytes and increment the index register by sizeof(int) = 4

```
    .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

X

What is a Bitwise Operation?



- Bitwise operators are applied 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 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

x

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
```

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 (~) of Register content between registers

mvn	Rd	rot4	imm8

destination register

constant (immediate value)

mvn	Rd	Rm

destination register

source register

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

~ 1100

0011

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
```

invert the bits

copy into 32 bits zero extend

r1 0xfffffffb ← 0x00000004 ← 0x4

r1 0x55555555 ← 0xaaaaaaaa r5

r1 0x11111111 ← 0x0

x

Bitwise (Bit to Bit) Operators in C

`output = ~a;`

a	~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

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

Bitwise AND

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

Bitwise OR

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

Bitwise EOR

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


x

Bitwise Instructions

<op>	Rd	Rn	rot4	imm8
	destination	operand 1	Operand 2 constant	

<op>	Rd	Rn	Rm
	destination	operand 1	Operand 2

```
<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 <= imm8 <= 255 + values from "rotating" rot 4 bits

Bitwise <op> description	*<op> Syntax*	*Operation*
Bitwise **AND**	and R_d, R_n, Op2	$R_d \leftarrow R_n$ & Op2
Bit Clear each bit in Op2 that is a 1, the same bit in R_d, is cleared	bic R_d, R_n, Op2	$R_d \leftarrow R_n$ & ~Op2
Bitwise **OR**	orr R_d, R_n, Op2	$R_d \leftarrow R_n$ | Op2
Exclusive **OR**	eor R_d, R_n, Op2	$R_d \leftarrow R_n$ ^ Op2

x

Bit Masks: Masking - 1

- Bit masks access/modify specific bits in memory
- Masking act of applying a mask to a value
- `or:` 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 force lower 16 bits to 1 "**mask on**" operation

```
orr  r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 ff ff lower half to 1
RSLT: r1 0xab ab ff ff
```

mask to invert the lower 8-bits "**bit toggle**" operation

```
eor  r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 00 ff flip LSB bits
RSLT: r1 0xab ab ab 88


MASK: r3 0x00 00 00 ff apply a 2nd time
RSLT: r1 0xab ab ab 77 original value!
```


x

Bit Masks: Masking - 2

mask to **extract top 8 bits** of r2 into r1

```
and   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0xff 00 00 00
RSLT: r1 0xab 00 00 00
```

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

```
and   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 00 01 is bit 0 set?
RSLT: r1 0x00 00 00 01 (0 if not set)
```

mask to force lower 8 bits to 0 "**mask off**" operation

```
and   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0xff ff ff 00 clear LSB
RSLT: r1 0xab ab ab 00
```

clear bit 5 to a 0 without changing the other bits

```
bic   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 00 20 clear bit 5 (0010)
RSLT: r1 0xab ab ab 57
```


x

Bit Masks: Masking - 3

mask to get **1's complement** operation

(like mvn)

```
eor   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0xff ff ff ff
RSLT: r1 0x54 54 54 88
```

remainder (mod): num % d where n ≥ 0 and d = 2^k

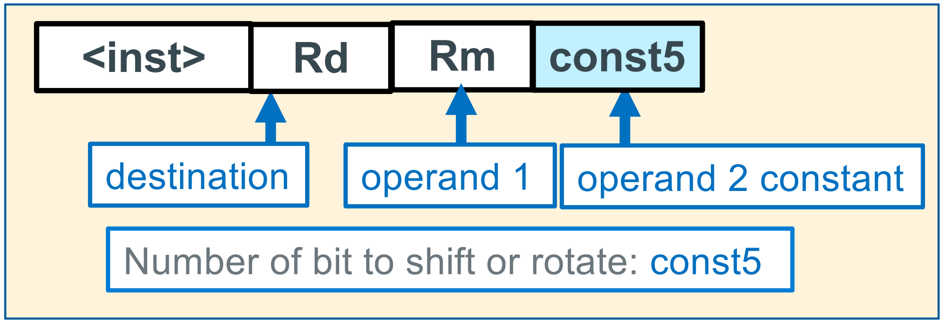
```
mask = 2^k  - 1 so for mod 2, mask = 2 -1 = 1
and   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 00 01 (mod 2 even or odd)
RSLT: r1 0x00 00 00 01 (odd)
```

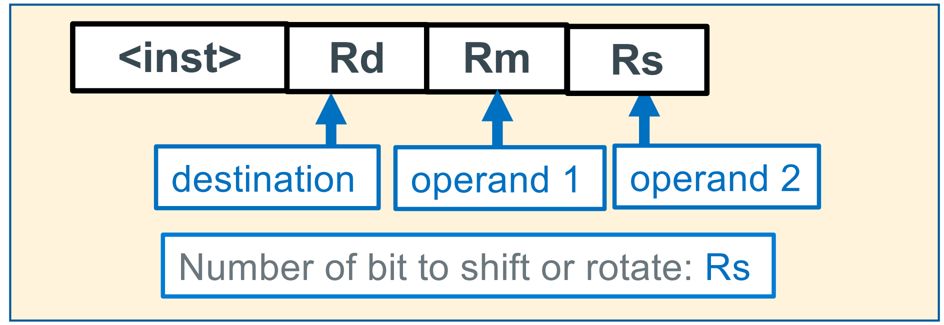
remainder (mod): num % d where n ≥ 0 and d = 2^k

```
mask = 2^k -1 so for mod 16, mask = 16 -1 = 15
and   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 00 0f (mod 16)
RSLT: r1 0xab 00 00 07 (if 0: divisible by)
```


x

Shift and Rotate Instructions



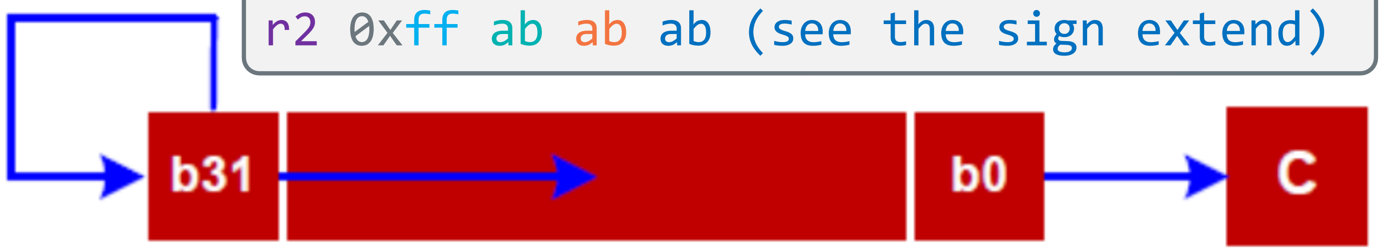


Instruction	*Syntax*	*Operation*	*Notes*	*Diagram*
Logical Shift Left	LSL R_d, R_m, *const5* LSL R_d, R_m, R_s	$R_d \leftarrow R_m$ << *const5* $R_d \leftarrow R_m$ << R_s	Zero fills shift: 0 - 31	
Logical Shift Right	LSR R_d, R_m, *const5* LSR R_d, R_m, R_s	$R_d \leftarrow R_m$ >> *const5* $R_d \leftarrow R_m$ >> R_s	Zero fills shift: 1 - 32	
Arithmetic Shift Right	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	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 & Rotate Operations

```
asr r2, r0, 8

r0 0xab ab ab 77
r2 0xff ab ab ab (see the sign extend)
```



Test for sign
-1 if r0 negative

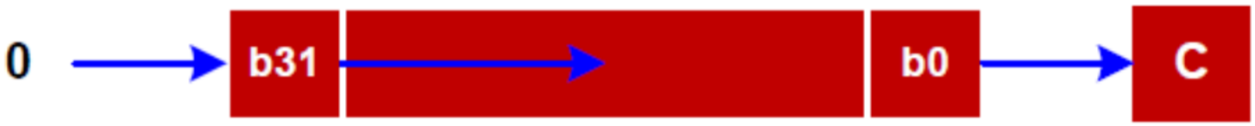
```
asr r2, r0, 31

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

Test for sign
0 if r0 positive

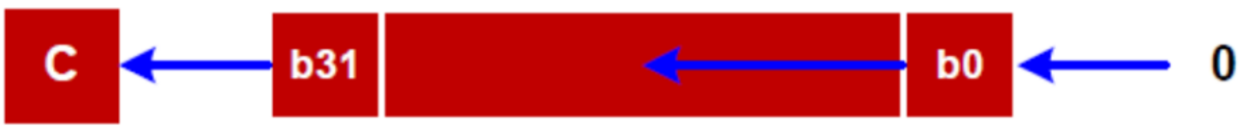
```
asr r2, r0, 31

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


x

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
```

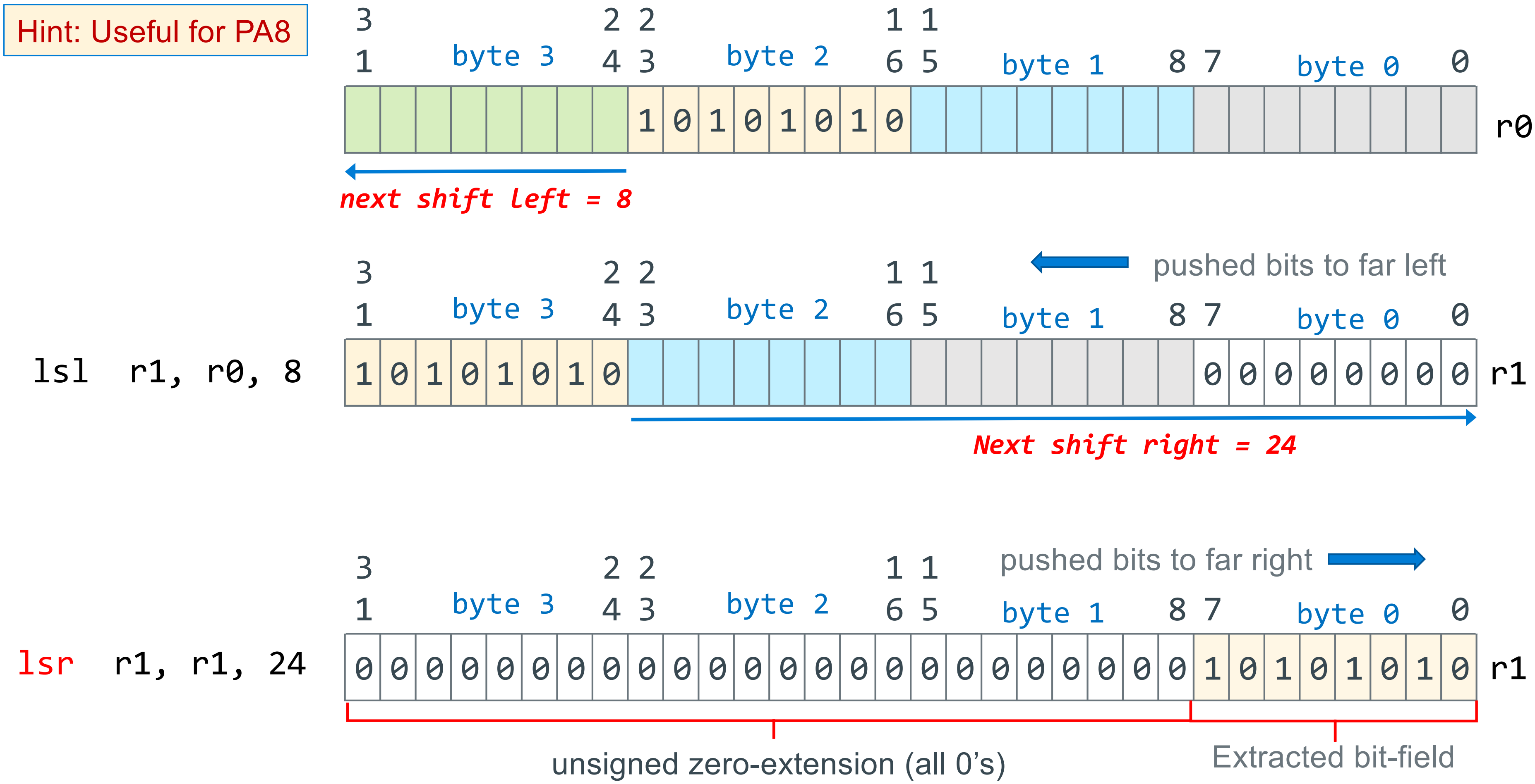

```
ror r2, r0, 8

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


x

Extracting Unsigned Bitfields

- Move byte 2 in r0 to byte 0 in r1

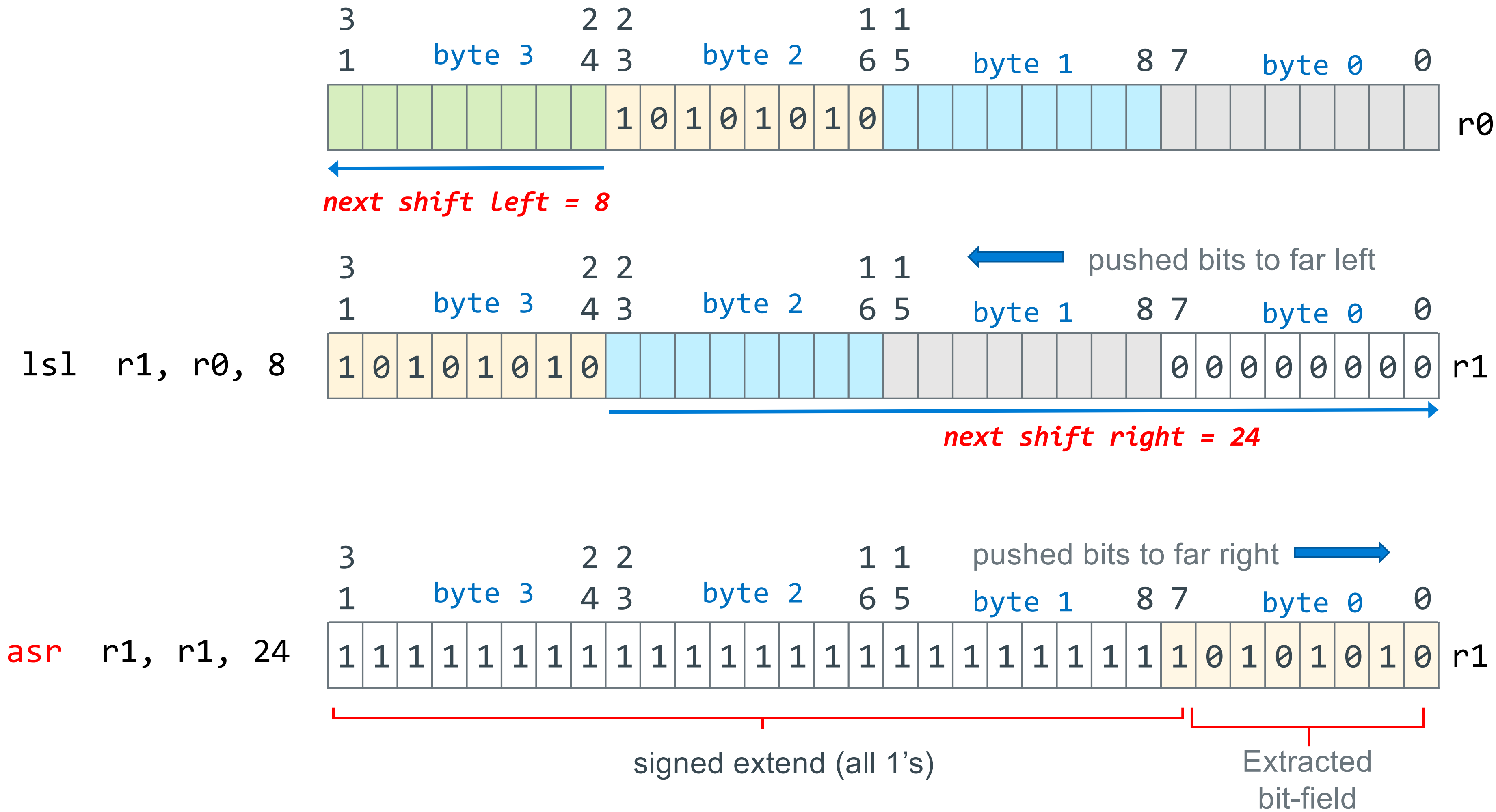
Hint: Useful for PA8



x

Extracting **Signed** Bitfields

- Move byte 2 in r0 to byte 0 in r1



x

Inserting Bitfields – Inserting Source Field into Destination Field

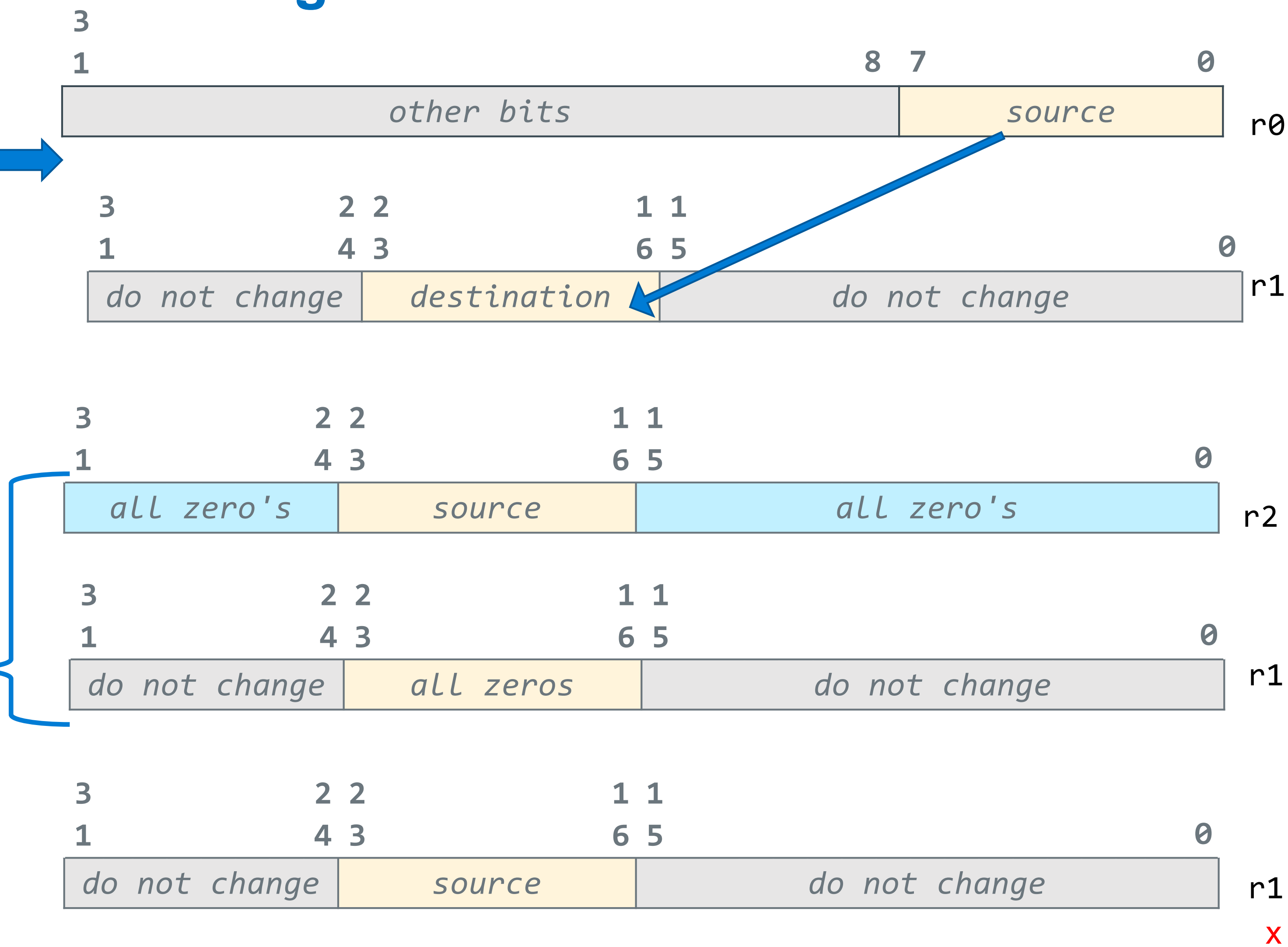
Task: Insert source into destination

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



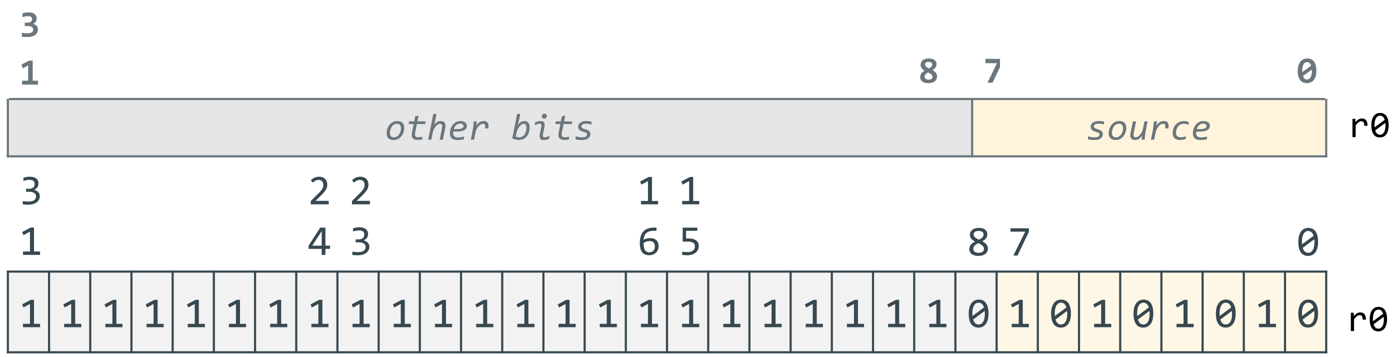
Creating a Mask

```
Create a 1 mask
ldr    r3, =0x00ffff0000


Create a 0 mask
ldr    r3, =0xff0000ffff
```

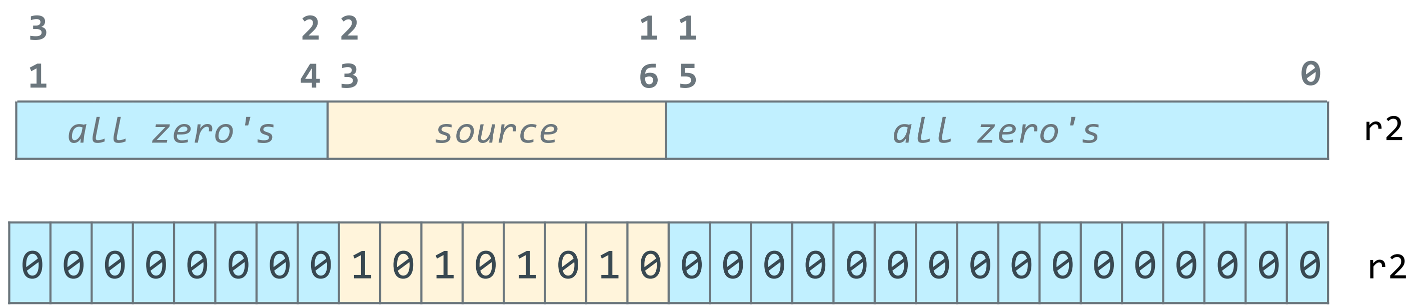
31							24	23							16	15							8	7							0
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

x

Inserting Bitfields – Isolating the Source Field

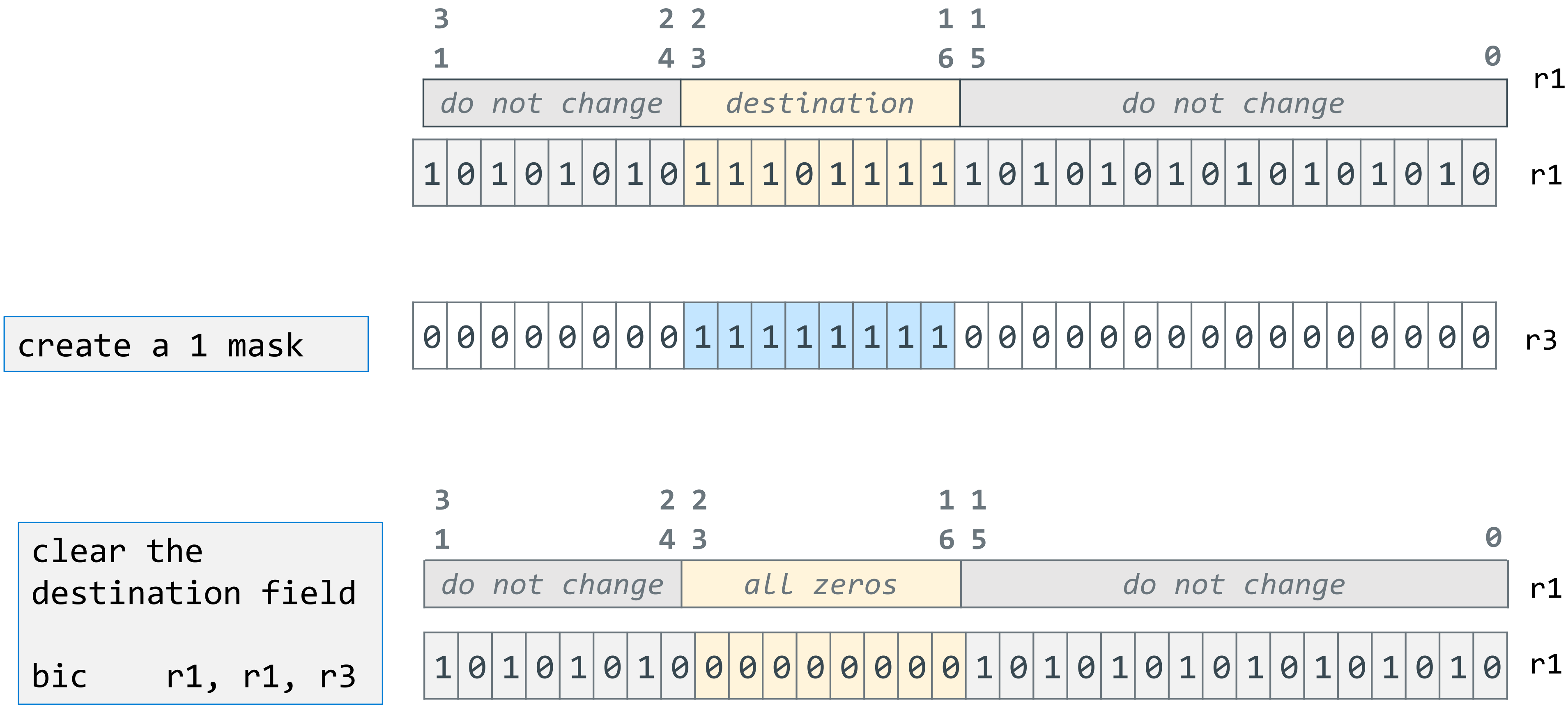



```
isolate source field

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



x

Inserting Bitfields – Clearing the Destination Field



x

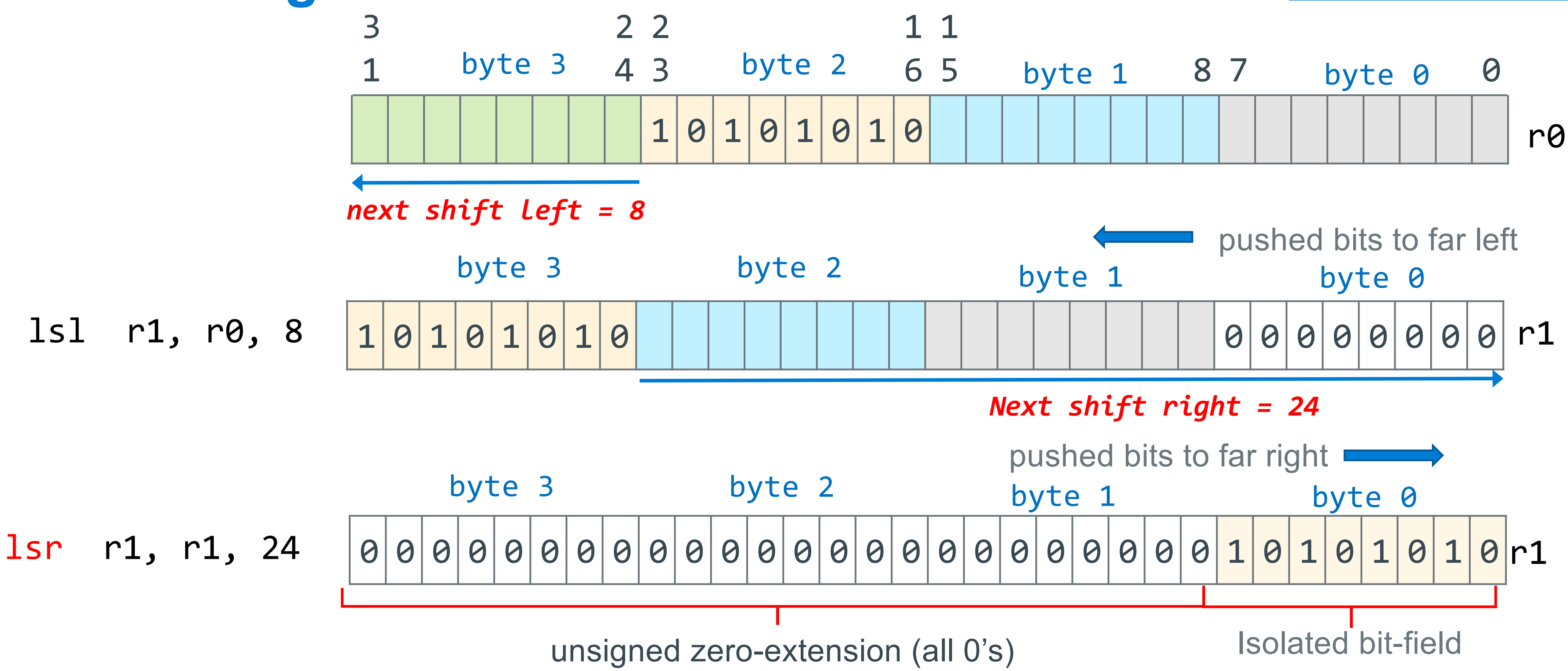
Inserting Bitfields –
Combining Isolated Source and Cleared Destination

	31 … 24	23 … 16	15 … 0	
isolated source	all zero's	source	all zero's	r2
	0 0 0 0 0 0 0 0	1 0 1 0 1 0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	r2
field cleared in destination	do not change	all zeros	do not change	r1
	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
inserted field orr r1, r1, r0	1 0 1 0 1 0 1 0	1 0 1 0 1 0 1 0	1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0	r1

x

Isolating Bitfields

Hint: Useful for PA8



- You can use ror to move the field to the desired location
- Alternative: If you can create an immediate value mask with a data operation like: movn, mov, add, or sub that is often faster

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

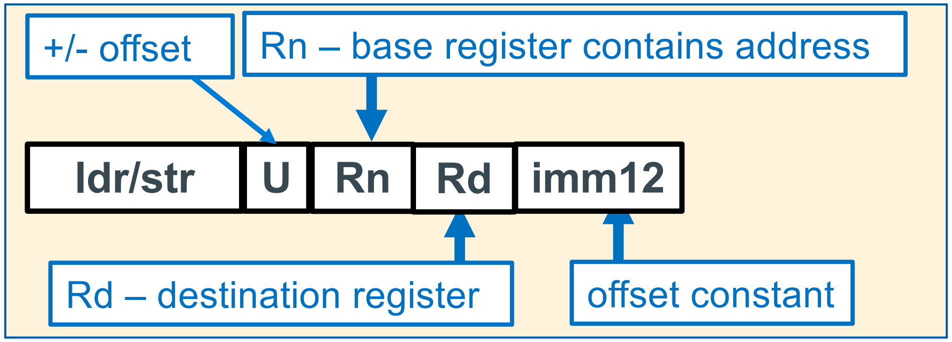
lsr to get this edge into lsb

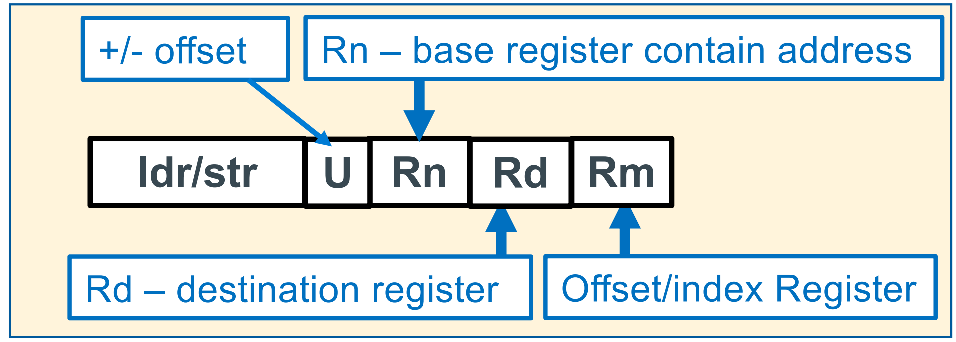
Insert a field: Use `orr` with fields surrounded by zeros

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

Extra Slides

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





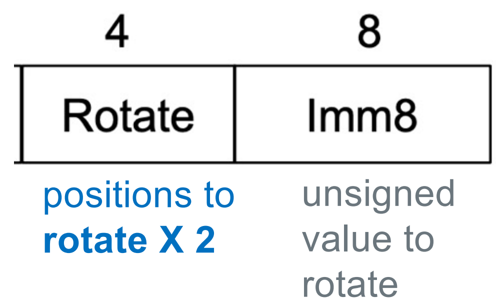

```
ldr/str  Rd,  [Rn, +- imm12] // base register pointer + offset  imm12 in bytes
                              -4095 <= imm12 <= 4095 (bytes)
ldr/str  Rd,  [Rn]            // base register pointer + 0 (imm12 is 0)
ldr/str  Rd,  [Rn, +- Rm]     // base register pointer +- offset register
```

```
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
```


x

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)



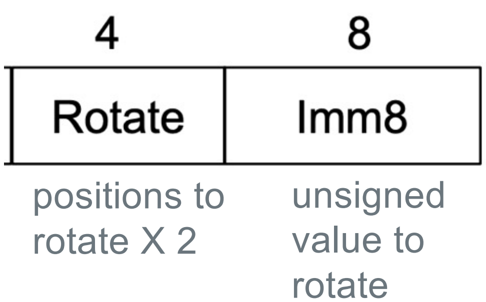
rot4 value	32-bit constant result
0000	0000 0000 0000 0000 0000 0000 1111 1111
0001 (2 bits)	1100 0000 0000 0000 0000 0000 0011 1111

results are interpreted as a 2's complement number

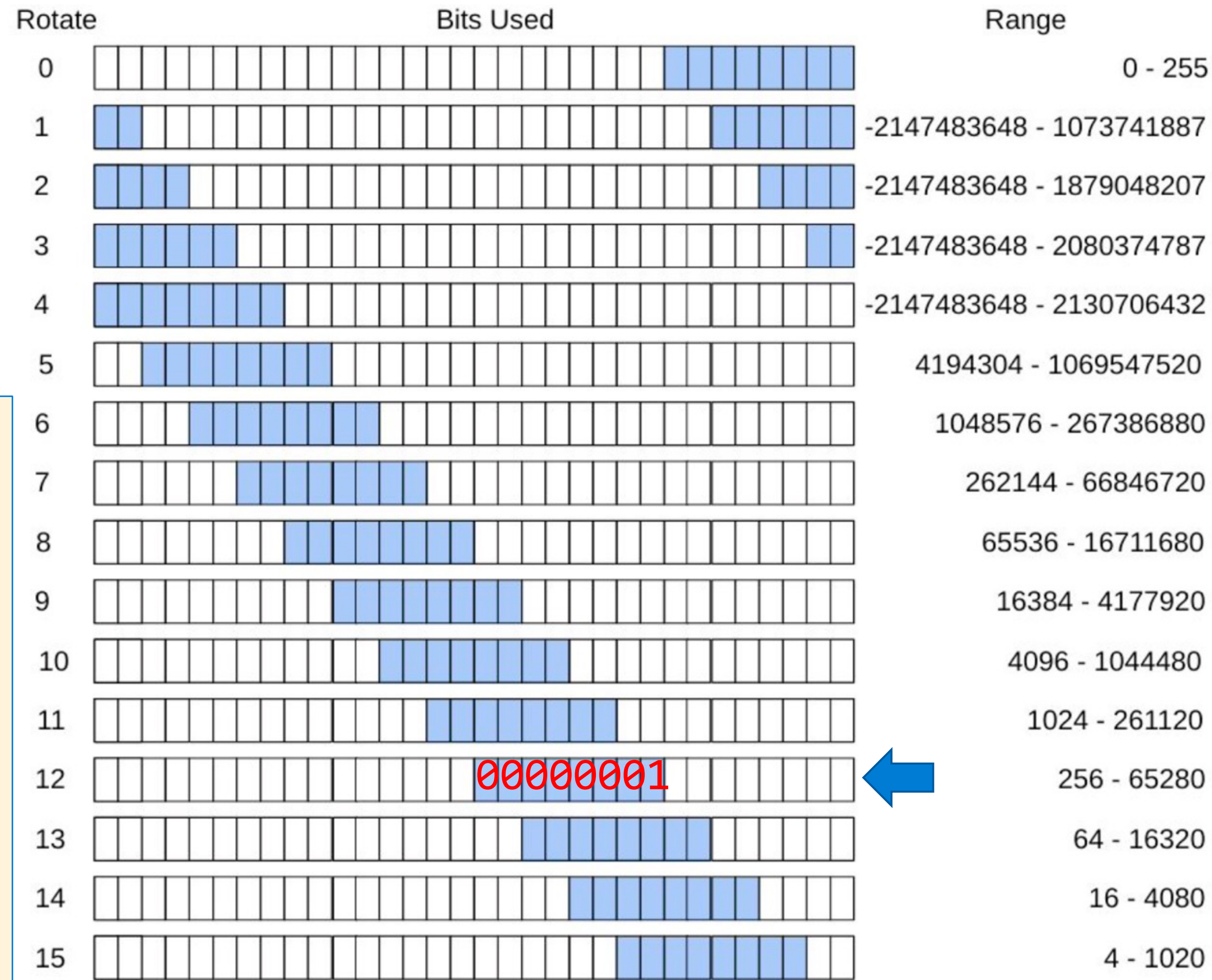
two bits rotated right

x

Rot4 - Imm8 Values



- 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

x

Branch Target Address (BTA): What Is imm24?

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: 1afffff5    bne 1042c <inloop>
```

BTA: + 2 instructions

- 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

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

imm24	0x 00 00 02

x

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`

	4	3	4	1	4	4	8	4
R type:	1110	000	Opcode	0	Rn	Rd	00000000	Rm

	4	3	4	1	4	4	4	8
I type:	1110	001	Opcode	0	Rn	Rd	Rotate	Imm

x

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;
```

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

special case: multiple branches from one cmp

- There are many other ways to do this

x

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

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  .Lmesg  // entry #3 32-bit address for .Lmesg
```


x

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  .Lmesg  // entry #3 32-bit address for .Lmesg
```

displacement1 - 8

displacement2 - 8

x

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)

X

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

X

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   .global myfunc            // make myfunc global for linking
Header     .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   .size myfunc, (. - myfunc)
Footer
```

`.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)`

X