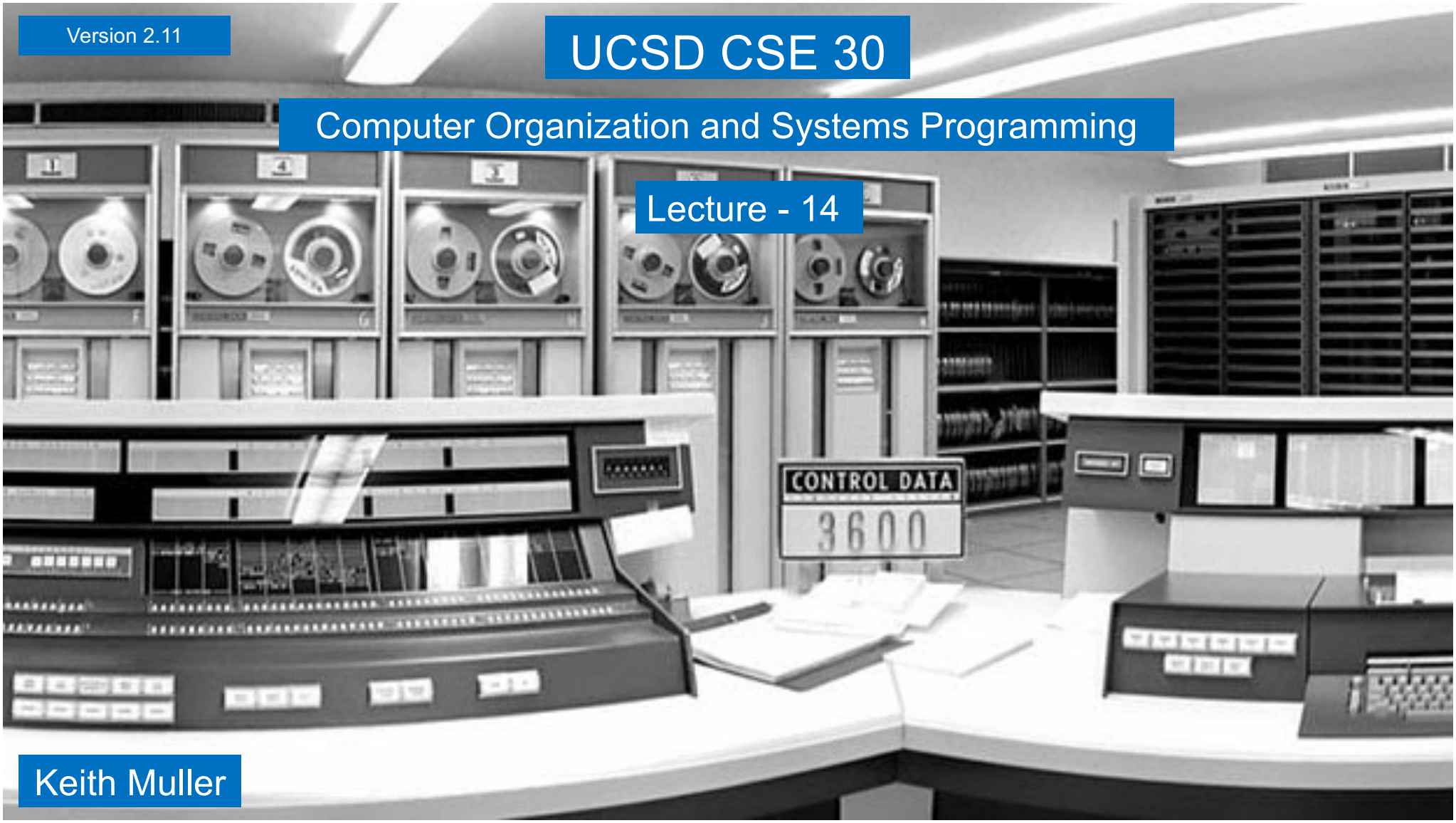
Version 2.11

UCSD CSE 30

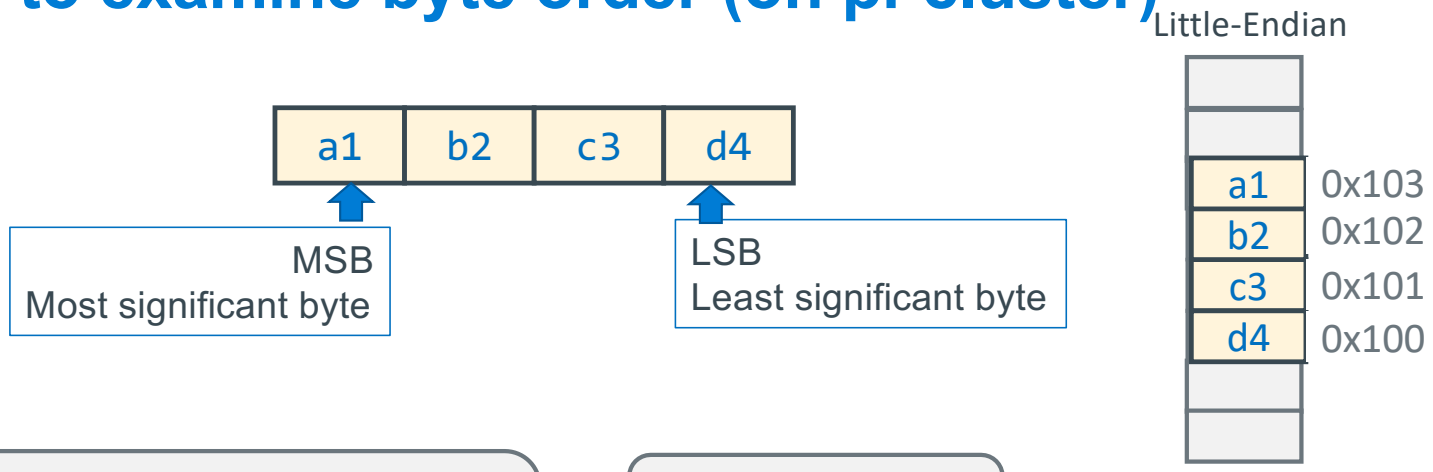
Computer Organization and Systems Programming

Lecture - 14



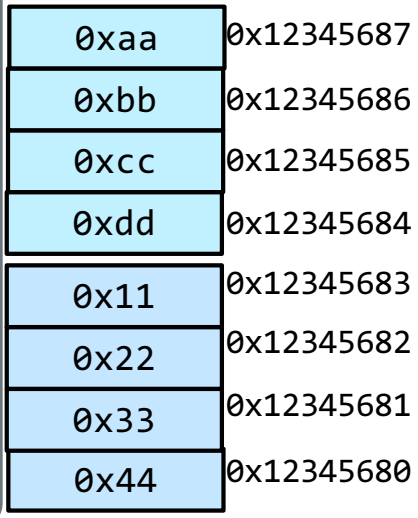
Keith Muller

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



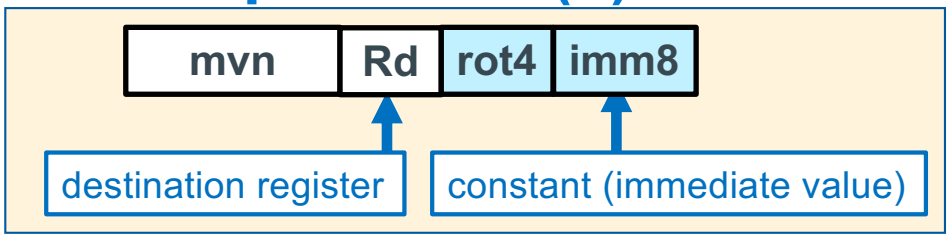

```c
#include <stdio.h>
#include <stdlib.h>
#define SZ 2
int main()
{
    unsigned int foo[SZ] = {0x11223344, 0xaabbccdd};
    unsigned char *chptr = (unsigned char *)foo;

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

    return EXIT_SUCCESS;
}
```

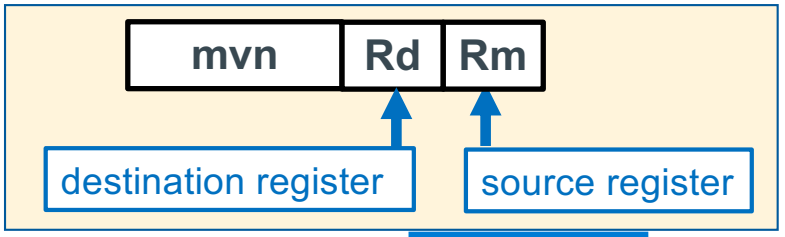
```
$ ./a.out
byte 7: aa
byte 6: bb
byte 5: cc
byte 4: dd
byte 3: 11
byte 2: 22
byte 1: 33
byte 0: 44
```



x

mvn – Copies NOT (~)






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

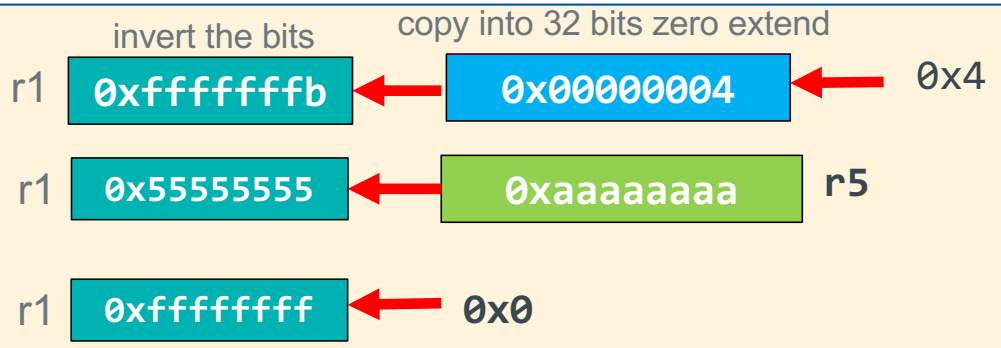
```
~ 1100
  ----
  0011
```

Bitwise NOT

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

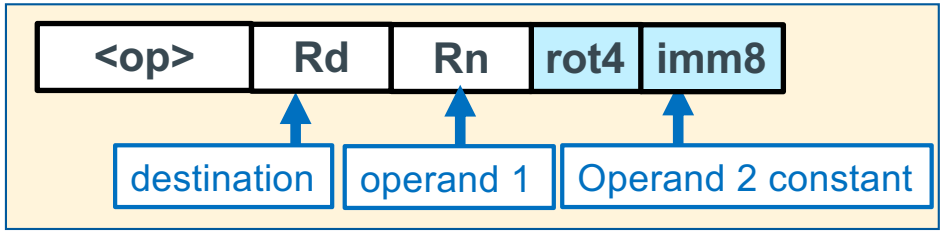
imm8	extended imm8	inverted imm8	signed base 10
0x00	0x00 00 00 00	0xff ff ff ff	-1
0xff	0x00 00 00 ff	0xff ff ff 00	-256

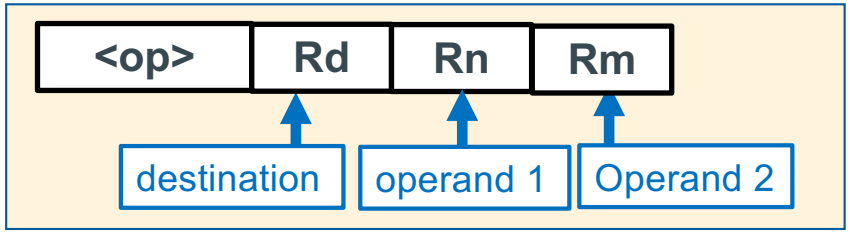
```
mvn    r1, 4          // x = ~4
mvn    r1, r5         // x = ~y in C
mvn    r1, 0          // x = -1
```



x

Bitwise Instructions






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

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

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

x

Inserting Bitfields – Inserting Source Field into Destination Field

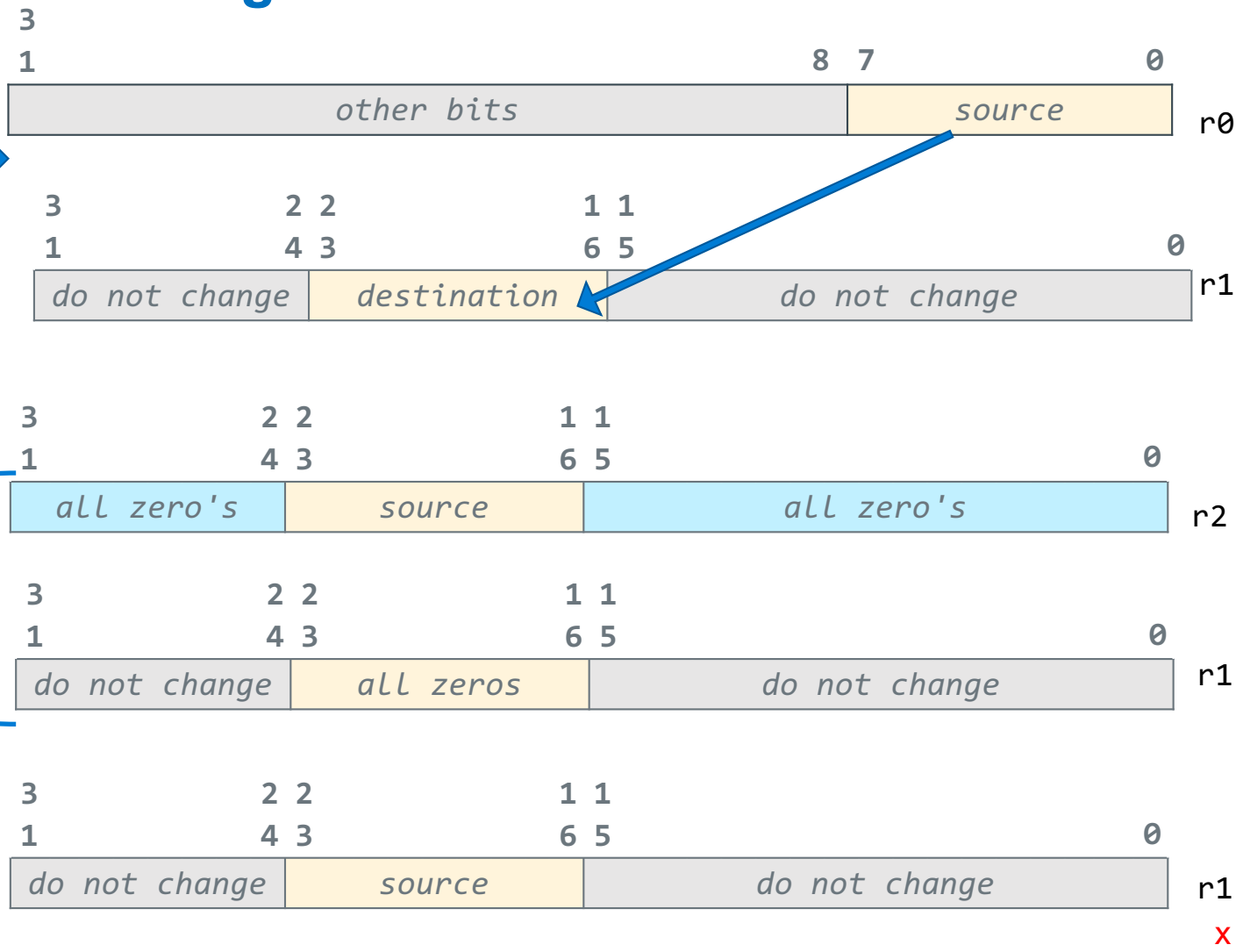
Task: Insert source into destination

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

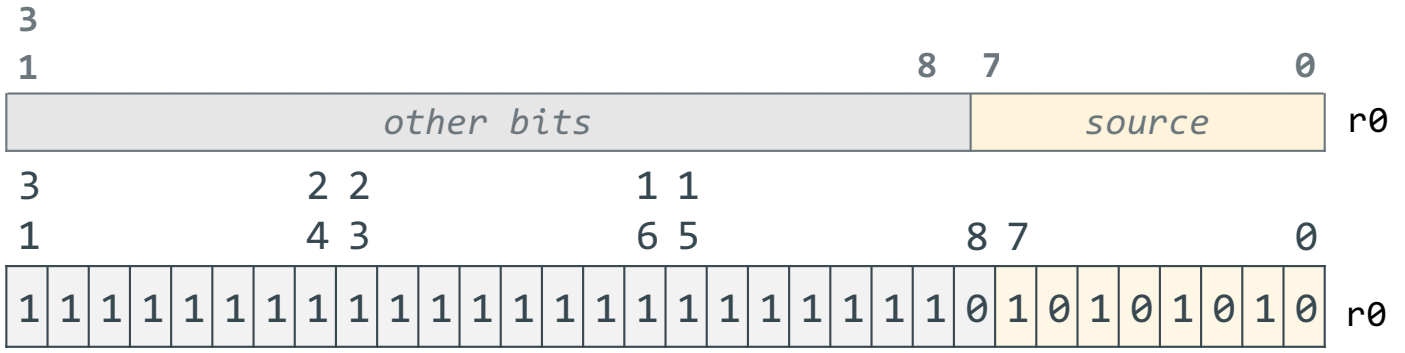
Approach
(1) isolate source field
(2) clear destination field
(3) Bitwise or together

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

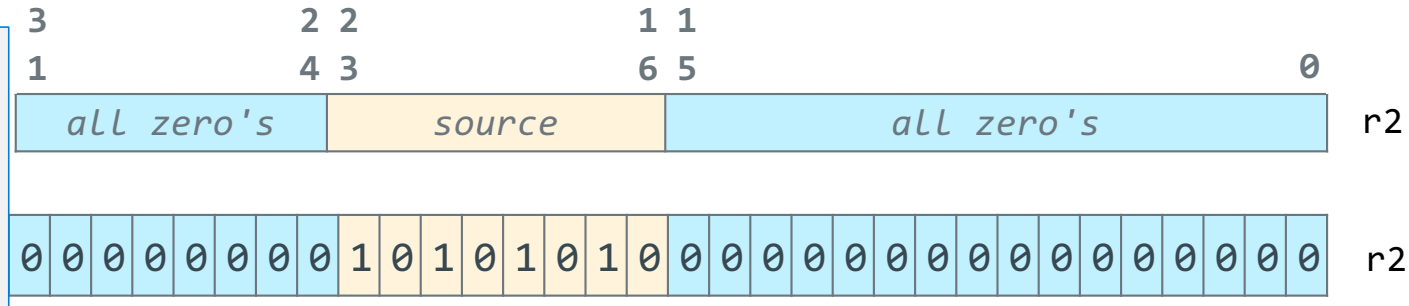
results in



Inserting Bitfields – Isolating the Source Field

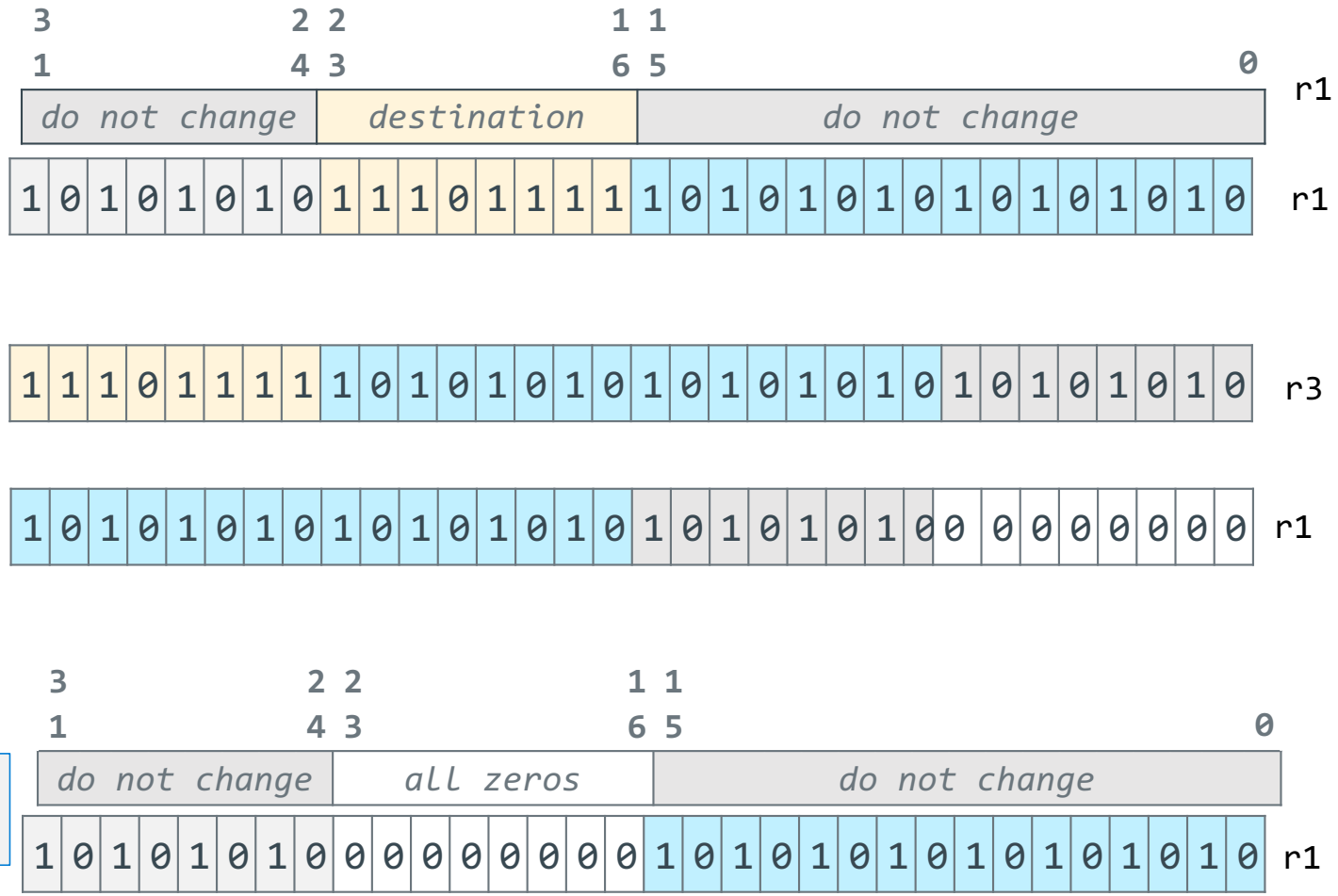



```
isolate source field

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

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



x

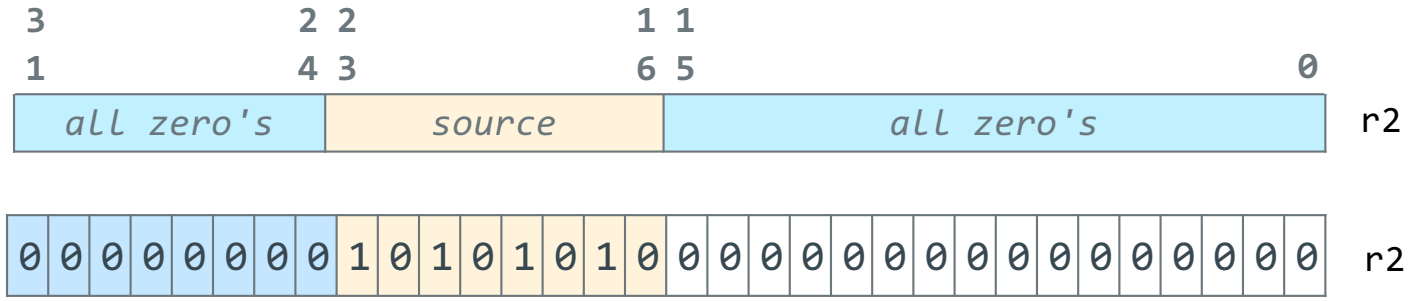
Inserting Bitfields – Clearing the Destination Field



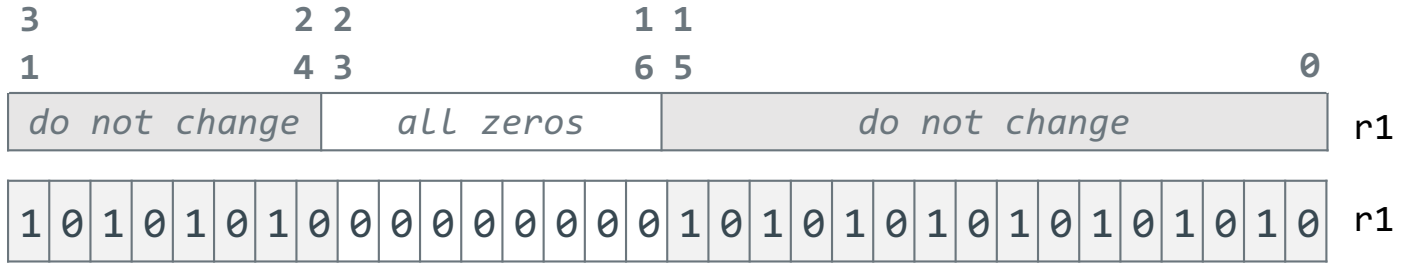
x

Inserting Bitfields – Combining Isolated Source and Cleared Destination

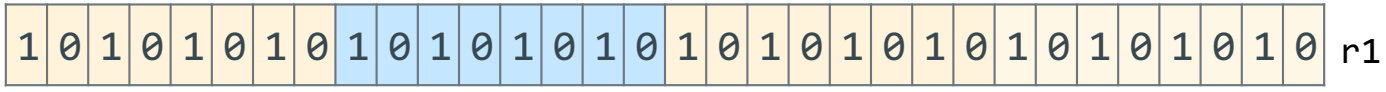
```
isolated source
```




```
field cleared in
destination
```

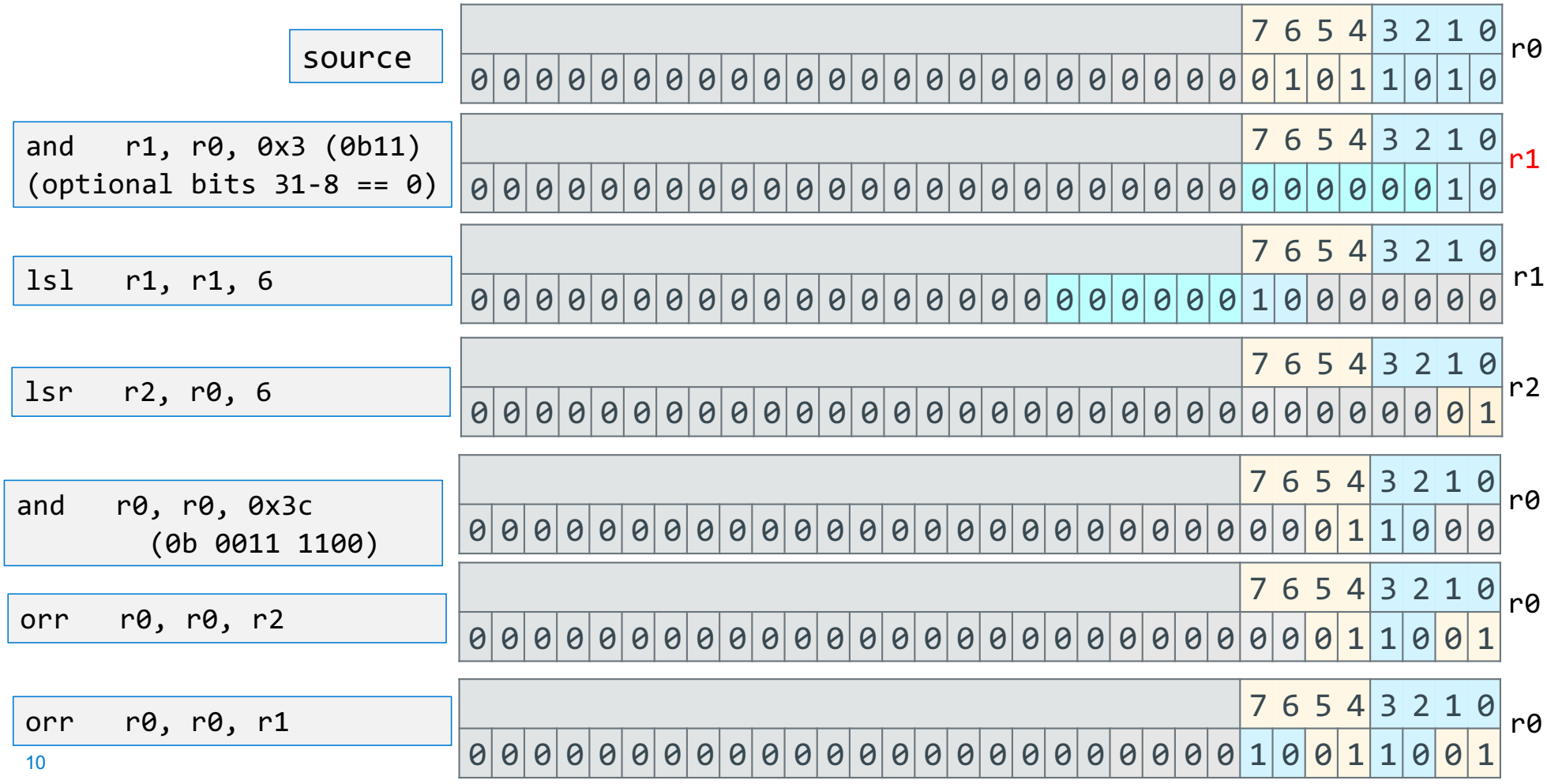



```
inserted field
orr    r1, r1, r2
r1 = r1 | r2;
```



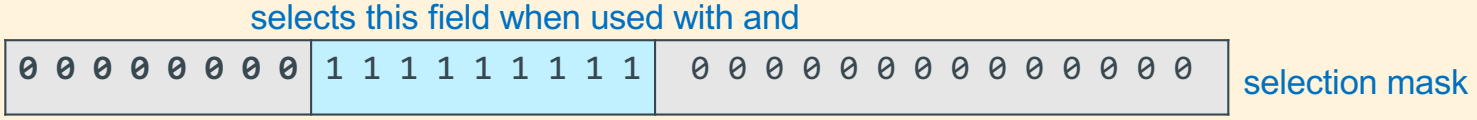
x

Example: Swapping bits7,6 with bits 1,0

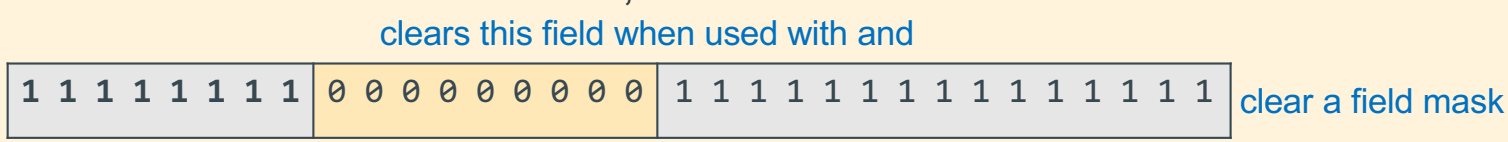


Masking Summary

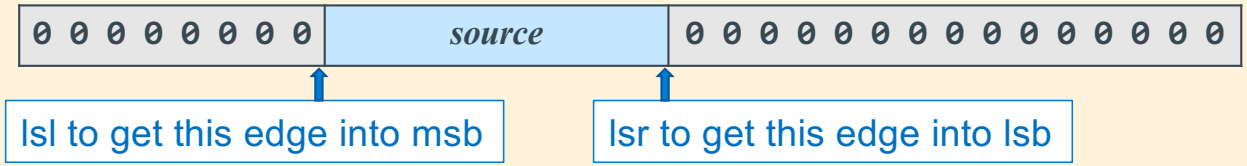
Select a field: Use `and` with a mask of one's surrounded by zero's to select the bits that have a 1 in the mask, all other bits will be set to zero



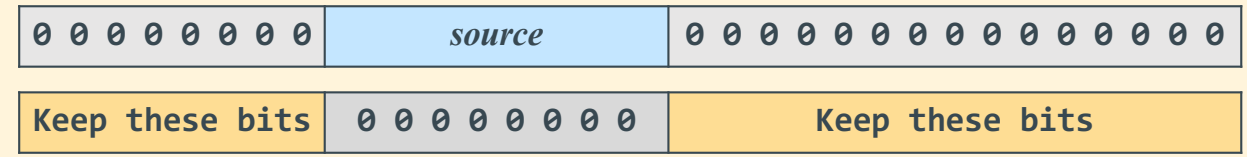
Clear a field: Use `and` with a mask of zero's surrounded by one's to select the bits that have a 1 in the mask, all other bits will be set to zero



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



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



x

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

x

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

x

C Stream Functions Array/block read/write

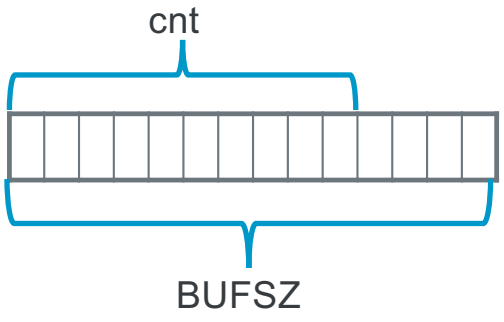
- These do not process contents they simply **transfer** a fixed number of bytes to and from a buffer passed to them
- `size_t fwrite(void *ptr, size_t size, size_t count, FILE *stream);`
  - Writes an array of *count* ***elements*** of ***size*** bytes from **stream**
  - *Updates the write file pointer 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**

x

C fread() and fwrite()

element size of 1 with a char buffer is byte I/O
Capture 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 cnt



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

```
size_t
fread(void *ptr, size_t size, size_t count, FILE *stream)
```

- Reads an array of ***count elements*** of ***size*** bytes from *stream*

```
size_t
fwrite(void *ptr, size_t size, size_t count, FILE *stream
```

- Writes an array of ***count elements*** of ***size*** bytes to *stream*

```c
#define BUFSZ   128

int copy(FILE *infp, FILE *outfp) {
    unsigned char buf[BUFSZ];
    size_t cnt;

    while ((cnt = fread(buf, 1, BUFSZ, infp)) > 0) {
        fprintf(stderr, "bytes: %u\n", cnt);

        if (fwrite(buf, 1, cnt, outfp) != cnt)
            return -1;
    }
    return 0;
}
```

```
% ls -l a
4 -rw-r--r-- 1 kmuller 1104 May 15 09:45 a
% ./a.out a b
bytes: 128
bytes: 128
bytes: 128
bytes: 128
bytes: 128
bytes: 128
bytes: 128
bytes: 128
bytes: 80
```

128 * 8 + 80 = 1104

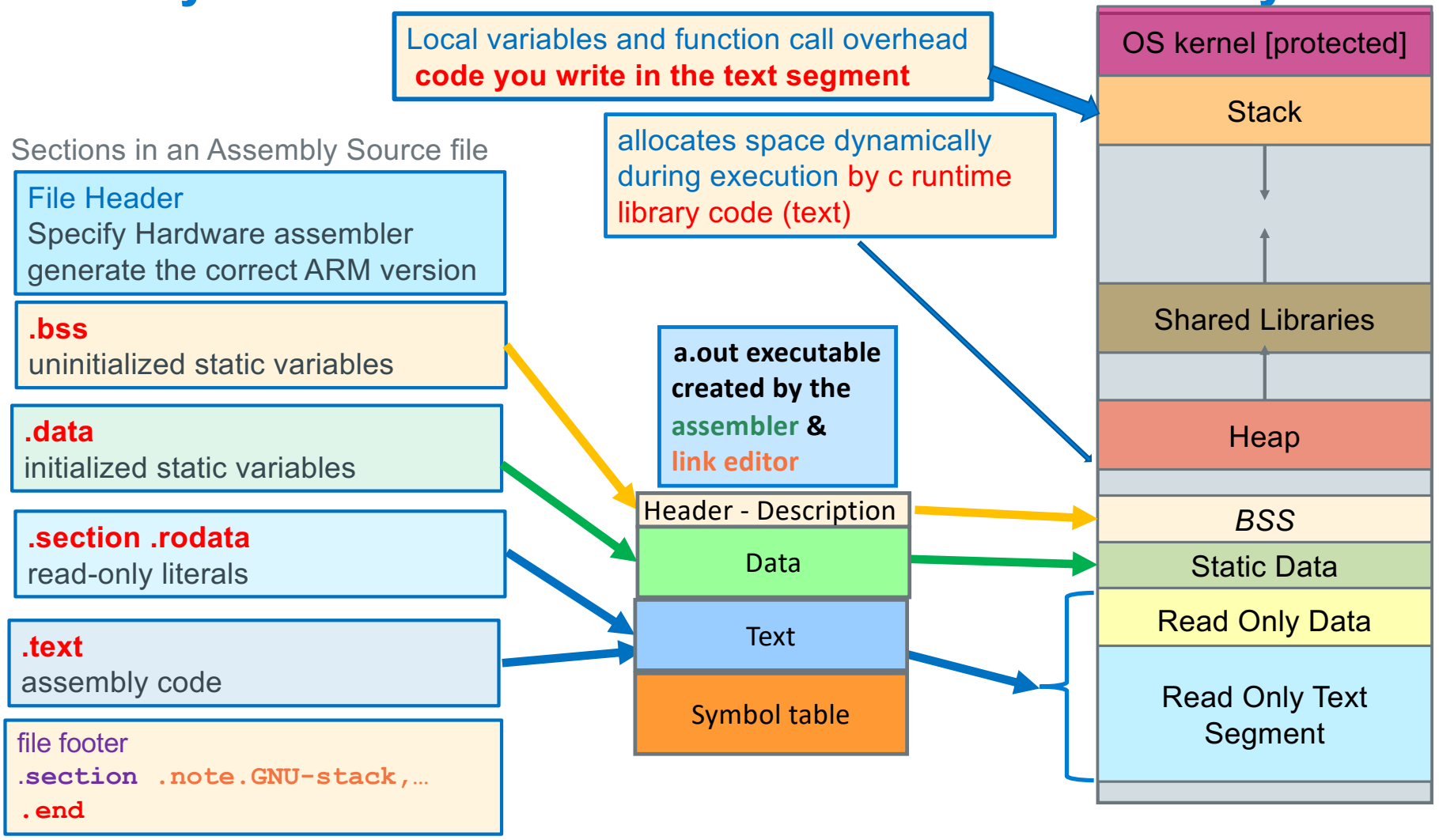
x

Using fopen() and fclose()

```
int main(int argc, char **argv)
{
    FILE *infp;
    FILE *outfp;
    int reslt;

    if (argc != 3) {
        fprintf(stderr, "%s requires two args\n", argv[0]);
        return EXIT_FAILURE;
    }
    // Open the input file for read
    if ((infp = fopen(*(argv+1), "r")) == NULL) {
        fprintf(stderr,"fopen for read failed\n");
        return EXIT_FAILURE;
    }
    //  Open the output file for write
    if ((outfp = fopen(*(argv+2), "w")) == NULL) {
        fprintf(stderr, "fopen for write failed\n");
        fclose(infp);
        return EXIT_FAILURE;
    }
    reslt = copy(infp, outfp);
    fclose(infp);
    fclose(outfp);
    if (reslt != 0) {
        fprintf(stderr, "copy %s to %s failed\n", *(argv+1), *(argv+2));
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```


x

Assembly Source File to Executable to Linux Memory



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)
```


x

```
// 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 Source File Template

- 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

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

Assembler Directives: .equ and .equiv

```
        .equ    BLKSZ,  10240     // buffer size in bytes
        .equ    BUFCNT, 100*4     // buffer for 100 ints
        .equ    BLKSZ,  STRSZ * 4 // redefine BLKSZ from here
```

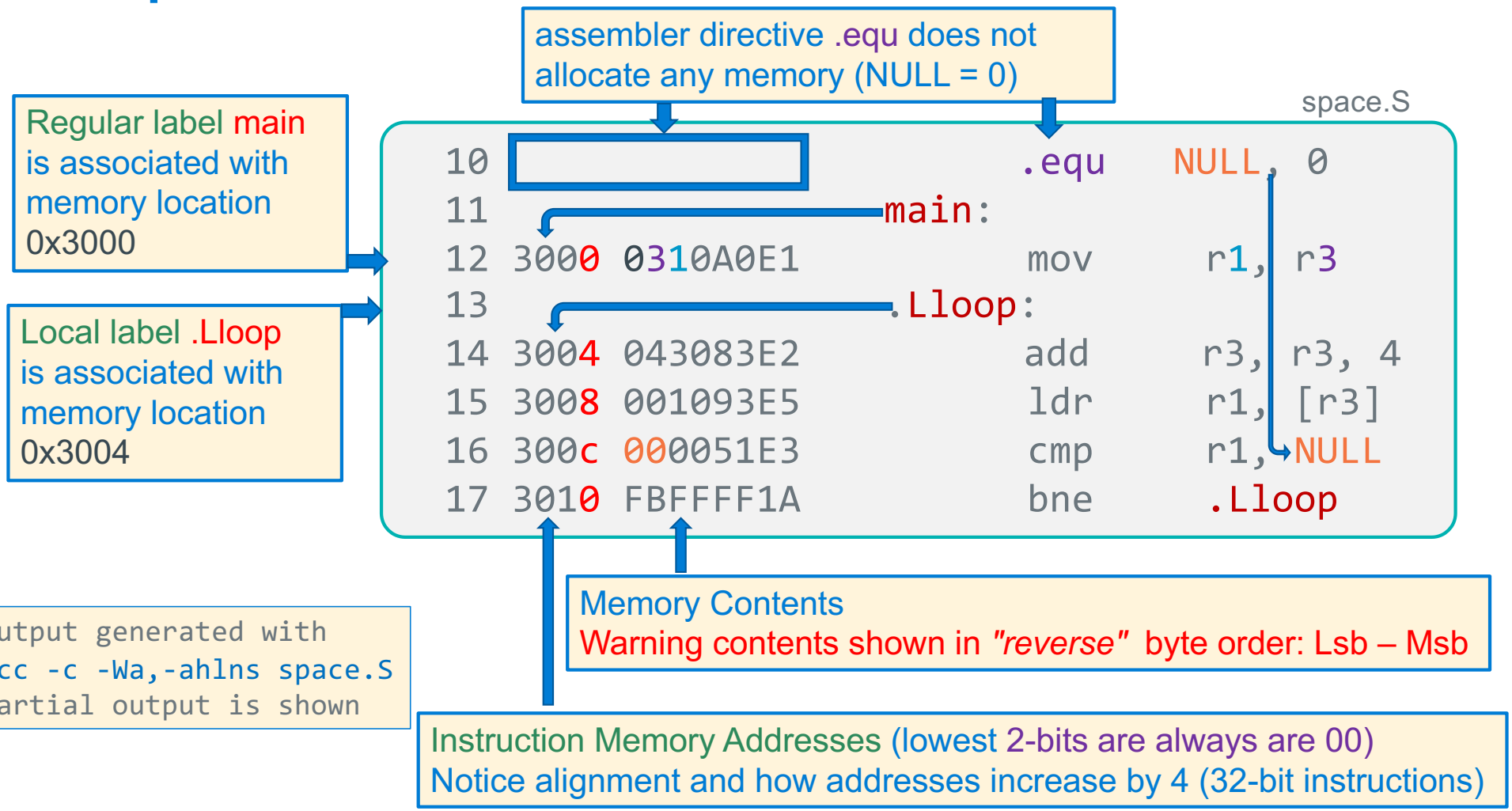
`.equ <symbol>, <expression>`

- Defines and sets the value of a symbol to the evaluation of the expression
- Used for specifying constants, like a `#define` in C
- You can (re)set a symbol many times in the file, last one seen applies

```
        .equ    BLKSZ,  10240       // buffer size in bytes
        // other lines
        .equ    BLKSZ,  1024        // buffer size in bytes
```


x

Example: Assembler Directive and Instructions



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)**

```
.size name, (. - name)
```


x

Function Prologue and Epilogue: Stack Frame Management Minimum Sized stack frame shown

```
        .text
        .global myfunc              // make myfunc global for linking
        .type   myfunc, %function // define myfunc to be a function
        .equ    FP_OFF,  4          // fp offset in main stack frame
myfunc:
        // function prologue, stack frame setup - (later slides)
        push    {fp, lr}
        add     fp, sp, FP_OFF
        // your code

        // function epilogue, stack frame teardown, return - (later slides)
        sub     sp, fp, FP_OFF
        pop     {fp, lr}
        bx      lr
        .size myfunc, (. - myfunc)
```

Function Prologue

Function Epilogue

x

Preview: Return Value and Passing Parameters to Functions

(Four parameters or less)

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

Assembler Directives: Label Scope Control (Normal Labels only)

```
.extern printf
.extern fgets
.extern strcpy
.global fbuf
```

`.extern <label>`

- **Imports** label (function name, symbol or a static variable name);
- An address associated with the label from another file can be used by code in this file

`.global <label>`

- **Exports** label (or symbol) to be visible outside the source file boundary (other assembly or c source)
- label is either a function name or a global variable name
- Only use with function names or static variables

- **Without** .global, labels are usually **local to the file** from the point where they are defined

x

Preview: Writing an ARM32 function

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

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

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

```
#ifndef SUM4_H
#define SUM4_H

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

#endif
```

```
#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
```

Variable Alignment In Memory and Performance

Accessing **address aligned** memory on many systems based on data type has the best performance (due to hardware implementation)

char

1 any address

short

2 bytes addresses that end in 0b0

integer

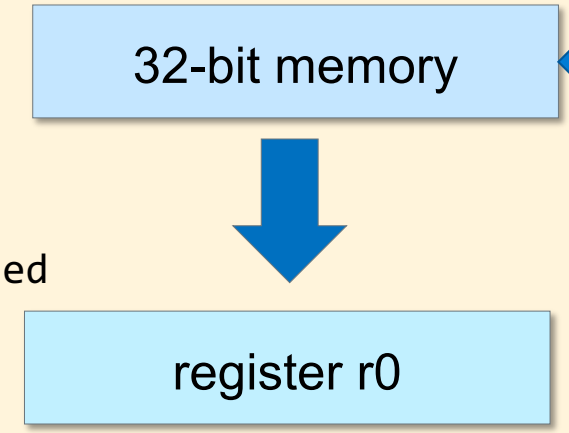
4 bytes addresses that end in 0b00

x

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

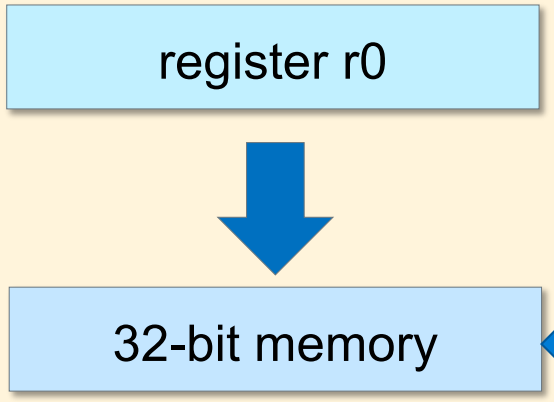


r1 is being used as a **pointer** to a location in memory

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)

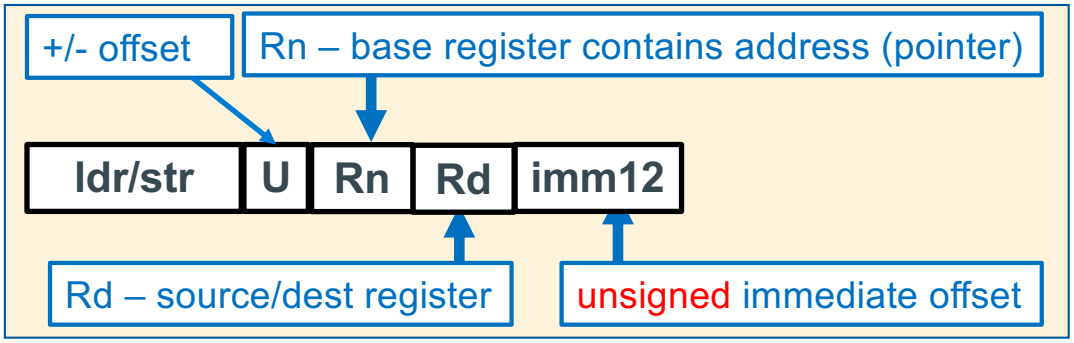


r1 is being used as a **pointer** to a location in memory

str requires the use of a pointer operand

x

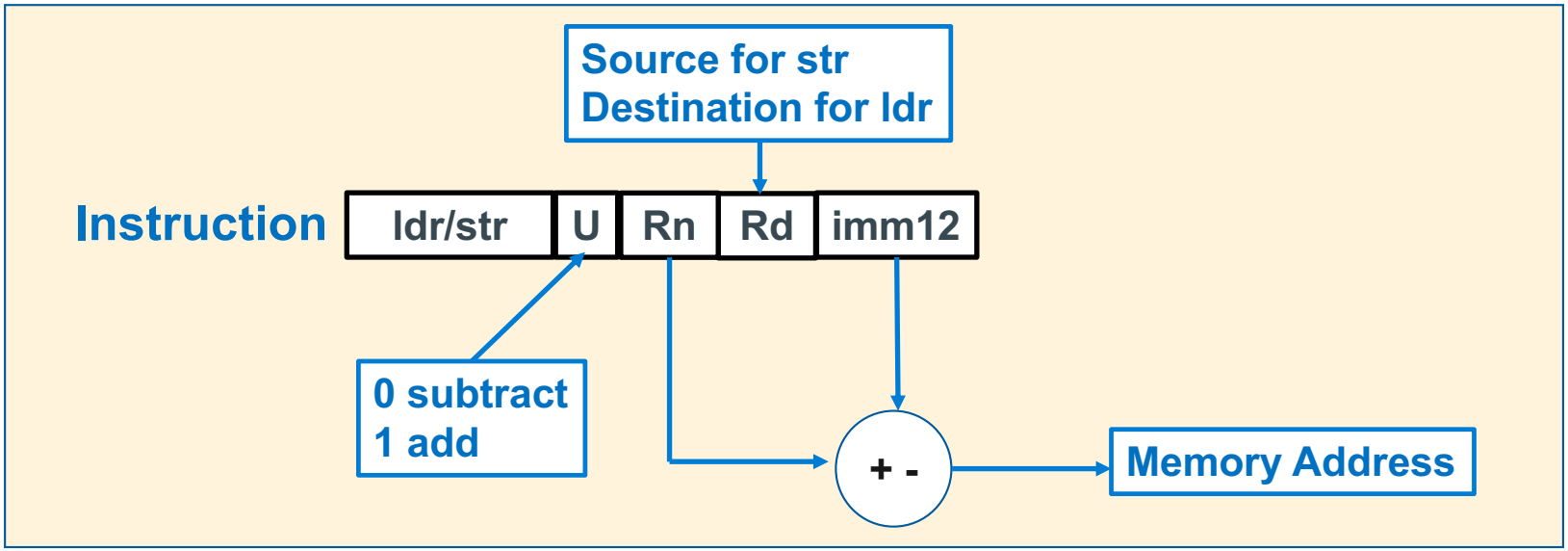
LDR/STR – Base Register + Immediate Offset Addressing



- **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 `-4095 <= imm12 <= 4095 (bytes)`
  - 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

x

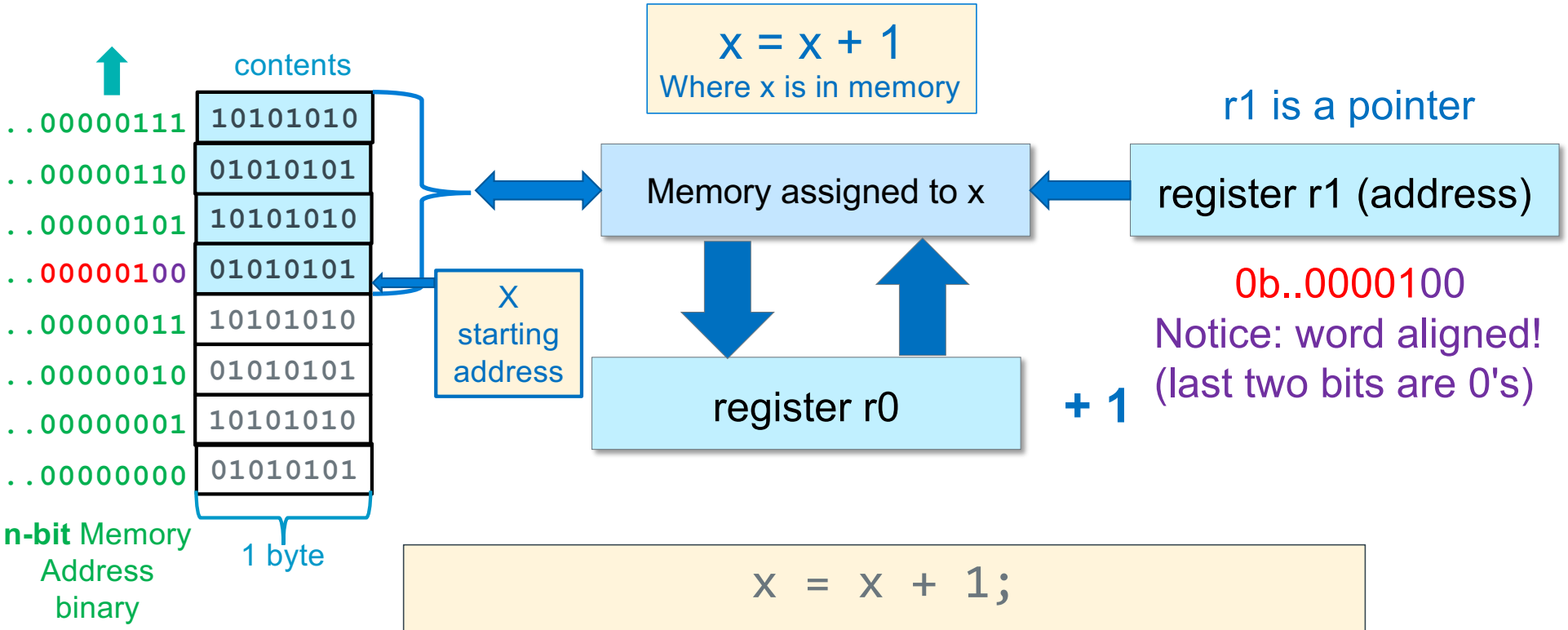
ldr/str Register Base + Immediate Offset Addressing



Syntax	Address	Examples
ldr/str Rd, [Rn, +/- constant] constant is in bytes ldr/str Rd, [Rn]	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

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)

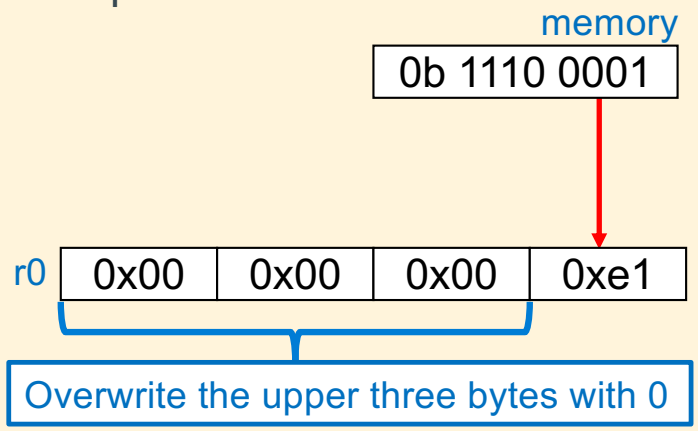
x

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

Unsigned

Zero-Extend: Add leading 0's

example `ldrb`

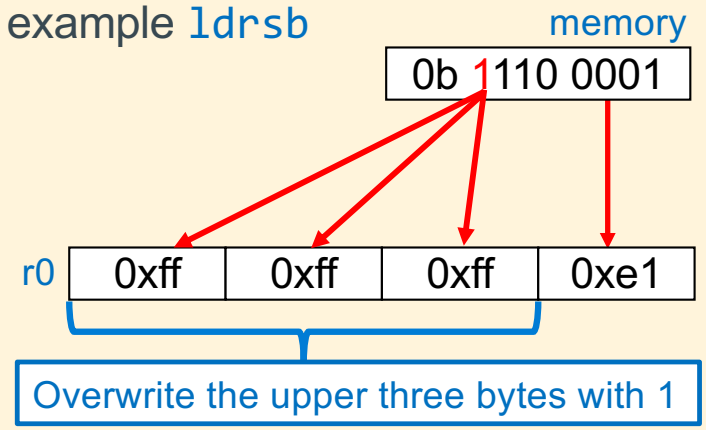


Instructions that zero-extend:
ldrb, ldrh

Signed (2's complement)

Sign-Extend: Replicate sign bit

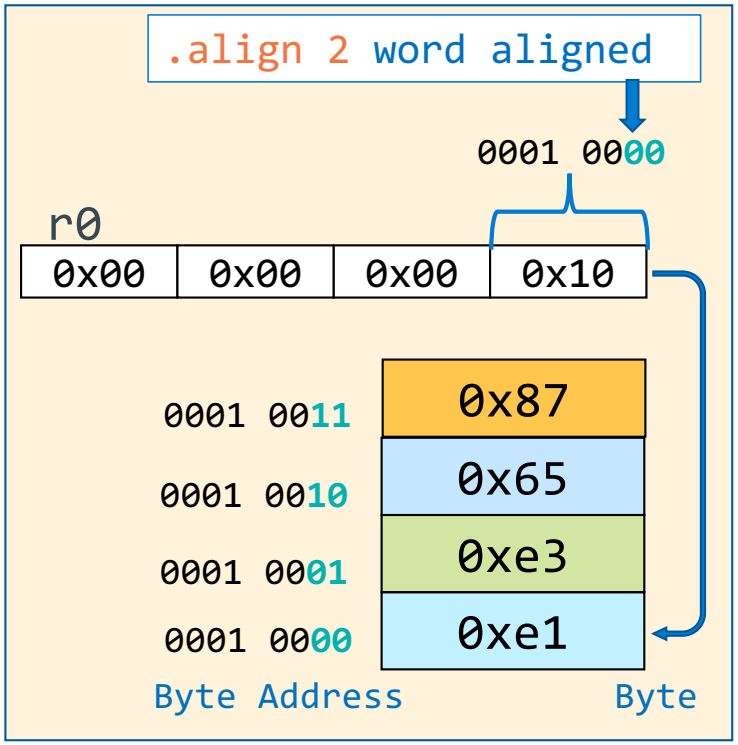
example `ldrsb`

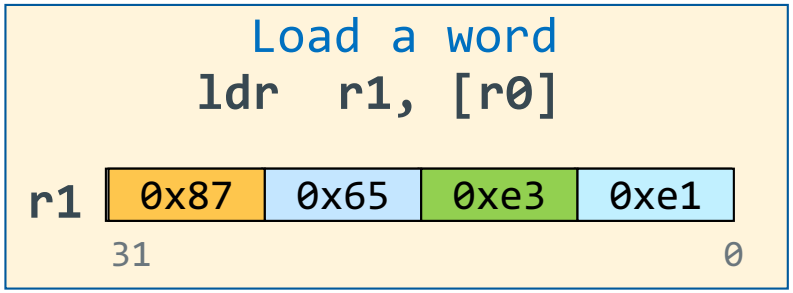


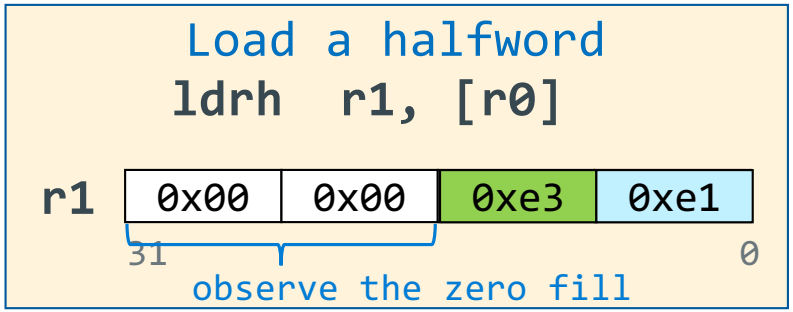
Instructions that sign-extend:
ldrsb, ldrsh

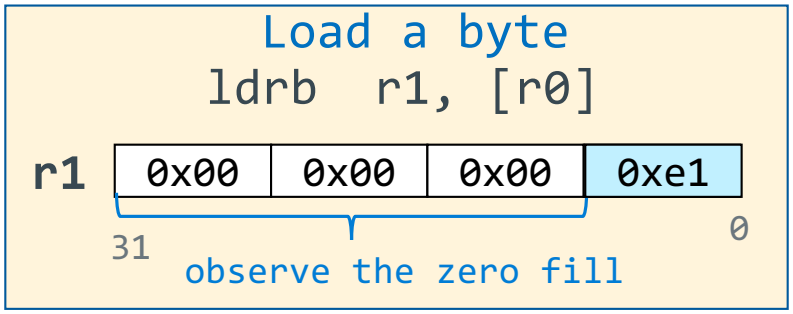
x

Load a Byte, Half-word, Word



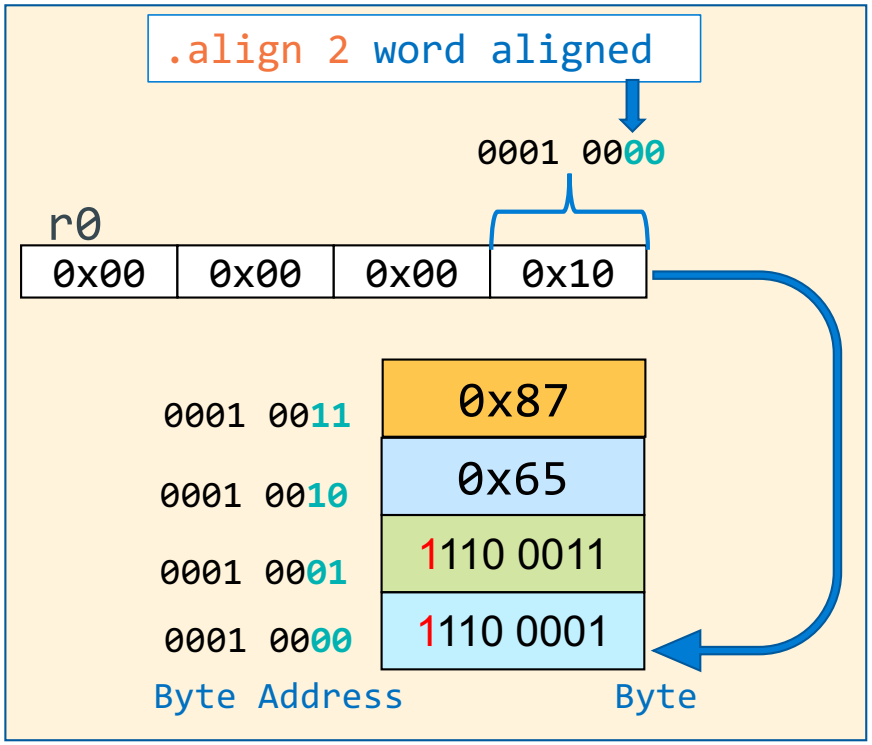






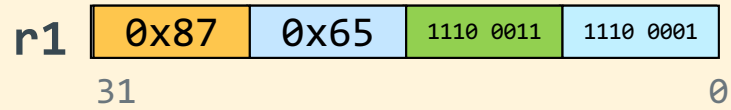
x

Signed Load a Byte, Half-word, Word



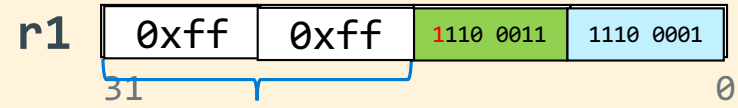
Load a word (no change)

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



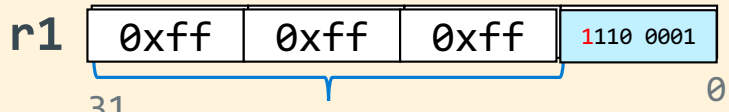
Load a halfword

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



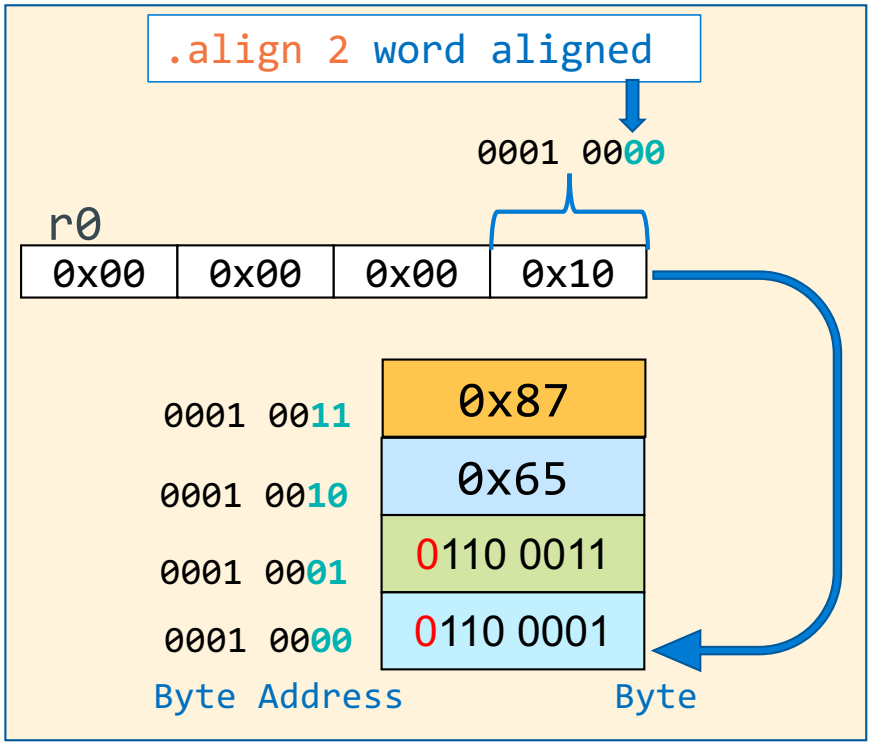
Load a byte

```
ldrsb  r1, [r0]
```



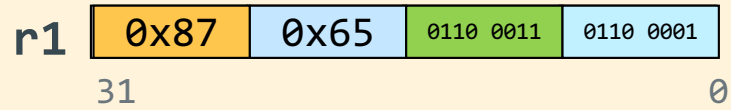
x

Signed Load a Byte, Half-word, Word



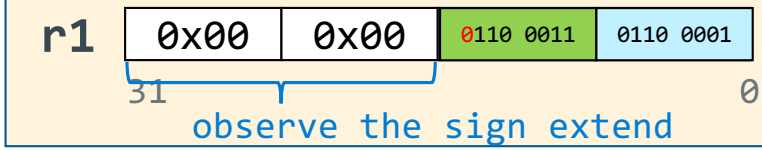
Load a word (no change)

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



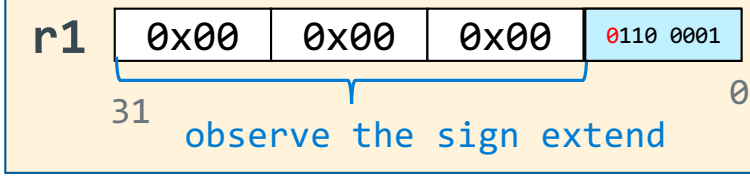
Load a halfword

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



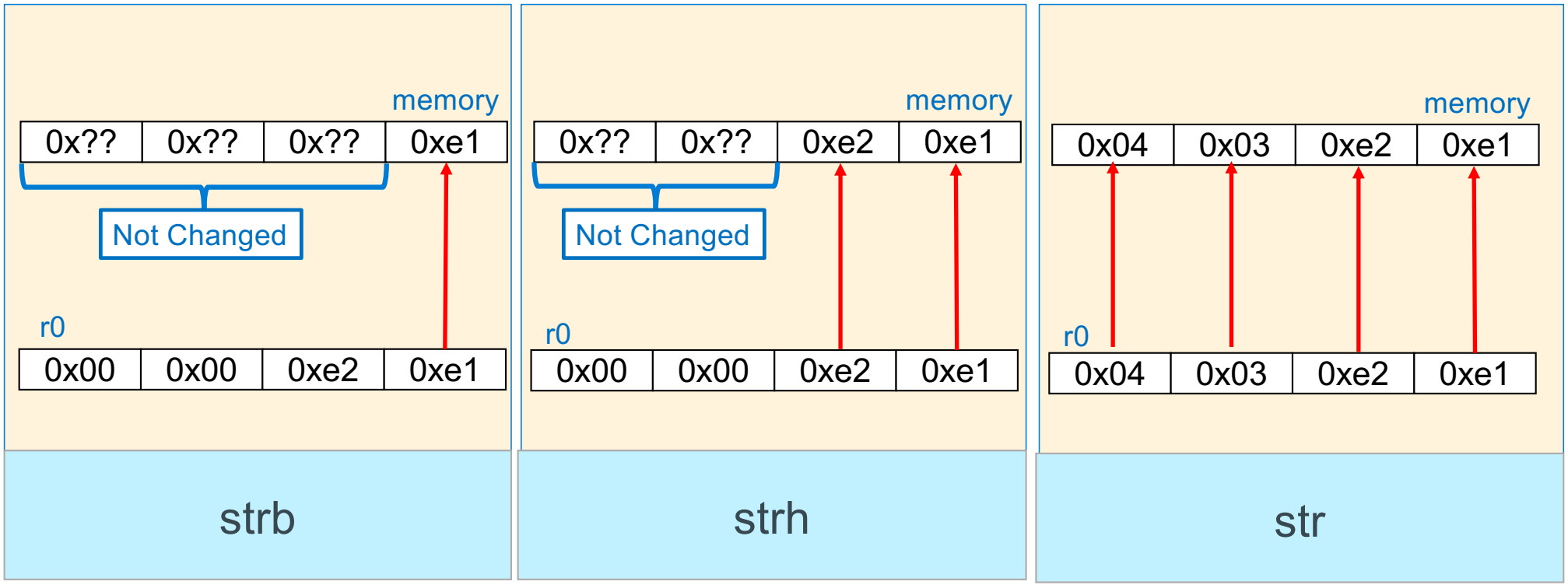
Load a byte

```
ldrsb  r1, [r0]
```



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

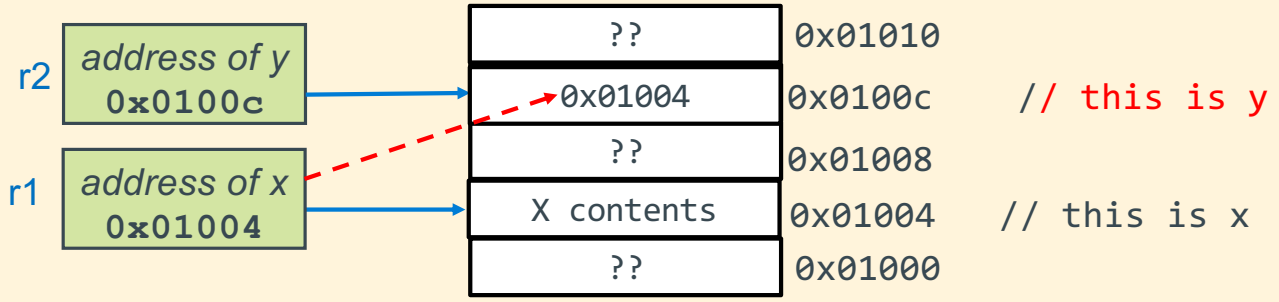
x

ldr/str 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
```

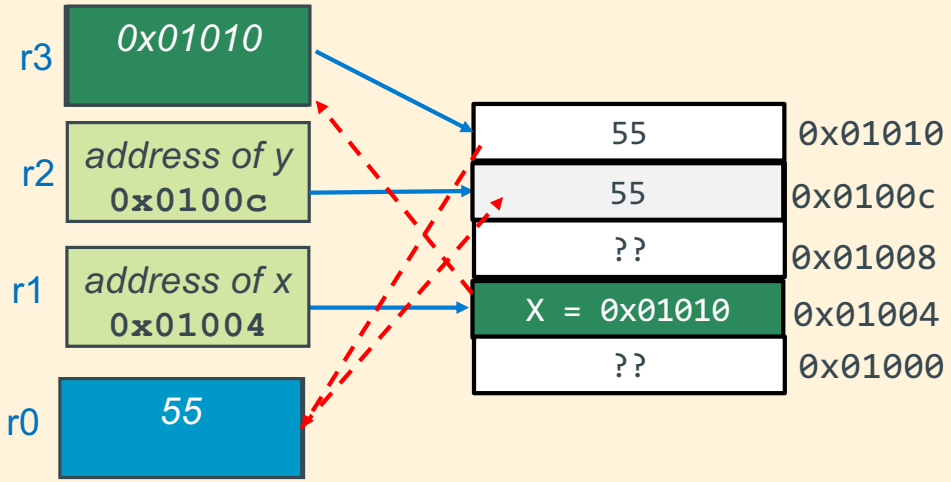

x

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

using ldr/str: array copy

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

void icpy(int *, int *, int);

int main(void)
{
    int  src[SZ] = {1, 2, 3, 4, 5, 6};
    int  dst[SZ];

    icpy(src, dst, SZ);
    for (int i = 0; i < SZ; i++)
        printf("%d\n", *(dst + i));

    return EXIT_SUCCESS;
}
```

```
void icpy(int *src, int *dst, int cnt)
{
    for (int i = 0; i < cnt; i++)
        *dst++ = *src++;

    return;
}
```


x

Base Register version

```
    .arch armv6
    .arm
    .fpu vfp
    .syntax unified
    .text
    .global icpy
    .type   icpy, %function
    .equ    FP_OFF, 12

    // r0 contains int *src
    // r1 contains int *dst
    // r2 contains int cnt
    // r3 use as loop term pointer
    // r4 use as temp

icpy:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
// see right ->
    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx      lr
    .size icpy, (. - icpy)
    .end
```

```
    cmp     r2, 0
    ble     .Ldone

    lsl     r2, r2, 2  //convert cnt to int size
    add     r3, r0, r2 // loop term pointer
.Ldo:
    ldr     r4, [r0]   // load from src
    str     r4, [r1]   // store to dest

    add     r0, r0, 4  // src++
    add     r1, r1, 4  // dst++

    cmp     r0, r3     // src < term pointer?
    blt     .Ldo

.Ldone:
```

pre loop guard

loop guard

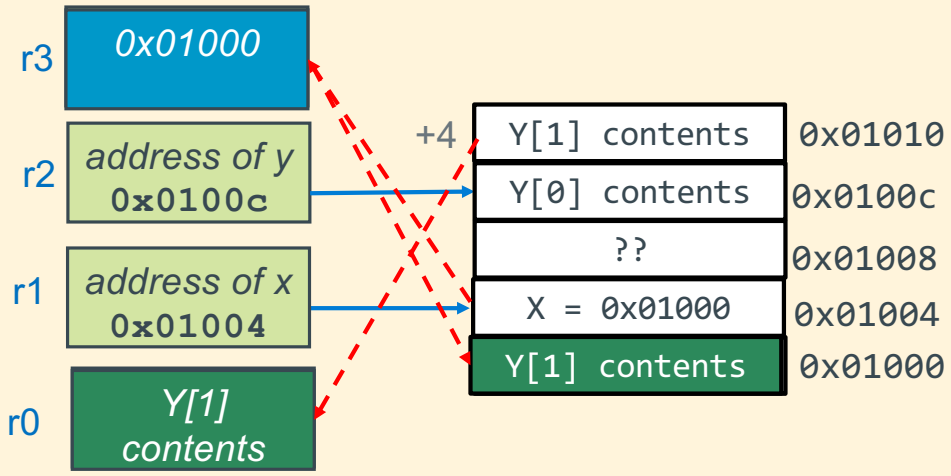
x

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])

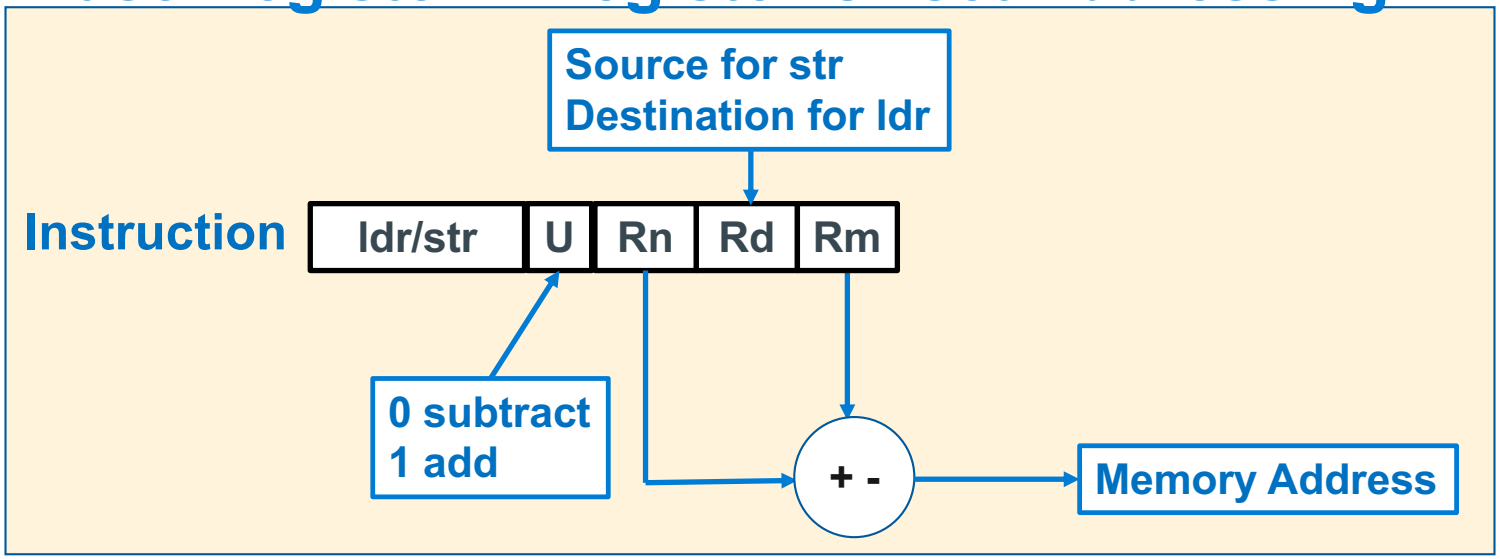
write *X = Y[1];




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


x

ldr/str 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

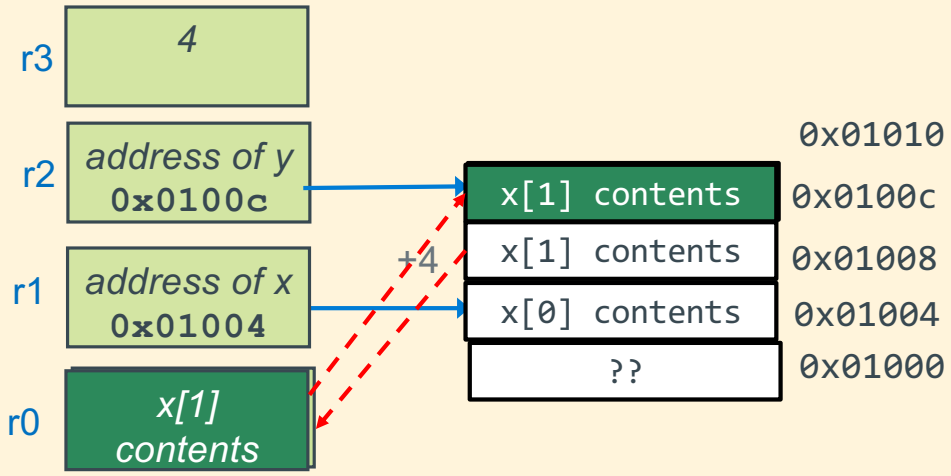
ldr/str practice - 4

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

Base Register + Register Offset Version

```
    .arch armv6
    .arm
    .fpu vfp
    .syntax unified
    .text
    .global icpy
    .type   icpy, %function
    .equ    FP_OFF, 12
    // r0 contains int *src
    // r1 contains int *dst
    // r2 contains int cnt
    // r3 use as loop counter
    // r4 use as temp

icpy:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
// see right ->
    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx      lr
    .size icpy, (. - cpy)
    .end
```

```
    cmp     r2, 0
    ble     .Ldone

    lsl     r2, r2, 2       //convert cnt to int size
    mov     r3, 0           // initialize counter
.Ldo:
    ldr     r4, [r0, r3]    // load from src
    str     r4, [r1, r3]    // store to dest
    add     r3, r3, 4       // counter++
    cmp     r3, r2          // count < r3
    blt     .Ldo

.Ldone:
```

pre loop guard

loop guard

one increment
covers both arrays

x

Base Register + Register Offset With chars

```
#include <stdio.h>
#include <stdlib.h>
#define SZ 6
void cpy(char *, char *, int);
int main(void)
{
    char src[SZ] =
       {'a', 'b', 'c', 'd', 'e', '\0'};
    char dst[SZ];


    cpy(src, dst, SZ);
    printf("%s\n", dst);
    return EXIT_SUCCESS;
}
```

```
    cmp     r2, 0
    ble     .Ldone

    mov     r3, 0           // initialize counter
.Ldo:
    ldrb     r4, [r0, r3]   // load from src
    strb     r4, [r1, r3]   // store to dest
    add     r3, r3, 1       // counter++
    cmp     r3, r2          // count < r3
    blt     .Ldo

.Ldone:
```


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

Base Register Addressing + Offset register

```
#include <stdio.h>
#include <stdlib.h>
int count(char *, int);
int main(void)
{
    char msg[] ="Hello CSE30! We Are CountinG UpPER cASe letters!";

    printf("%d\n", count(msg, sizeof(msg)/sizeof(*msg)));
    return EXIT_SUCCESS;
}
```

```
int count(char *ptr, int len)
{
    int cnt = 0;
    int i;

    for (i = 0; i < len; i++) {
        if ((ptr[i] >= 'A') && (ptr[i] <= 'Z'))
            cnt++;
   }
    return cnt;
}
```


x

Base Register + Offset register

```
    .arch armv6
    .arm
    .fpu vfp
    .syntax unified
    .text
    .global count
    .type   count, %function
    .equ    FP_OFF, 12
    // r0 contains char *ptr
    // r1 contains int len
    // r2 contains int cnt
    // r3 contains int i
    // r4 contains char

count:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
// see right ->
    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx      lr
    .size count, (. - count)
    .end
```

byte array
Also use ldrb here
offsets are 0,1,2,…

```
count:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF

    mov     r2, 0
    cmp     r1, 0
    ble     .Ldone
    mov     r3, 0
.Lfor:
    cmp     r3, r1
    bge     .Ldone

    ldrb    r4, [r0, r3]
    cmp     r4, 'A'
    blt     .Lendif
    cmp     r4, 'Z'
    bgt     .Lendif
    add     r2, r2, 1

.Lendif:
    add     r3, r3, 1
    b       .Lfor

.Ldone:
    mov     r0, r2
```

loop guard

x