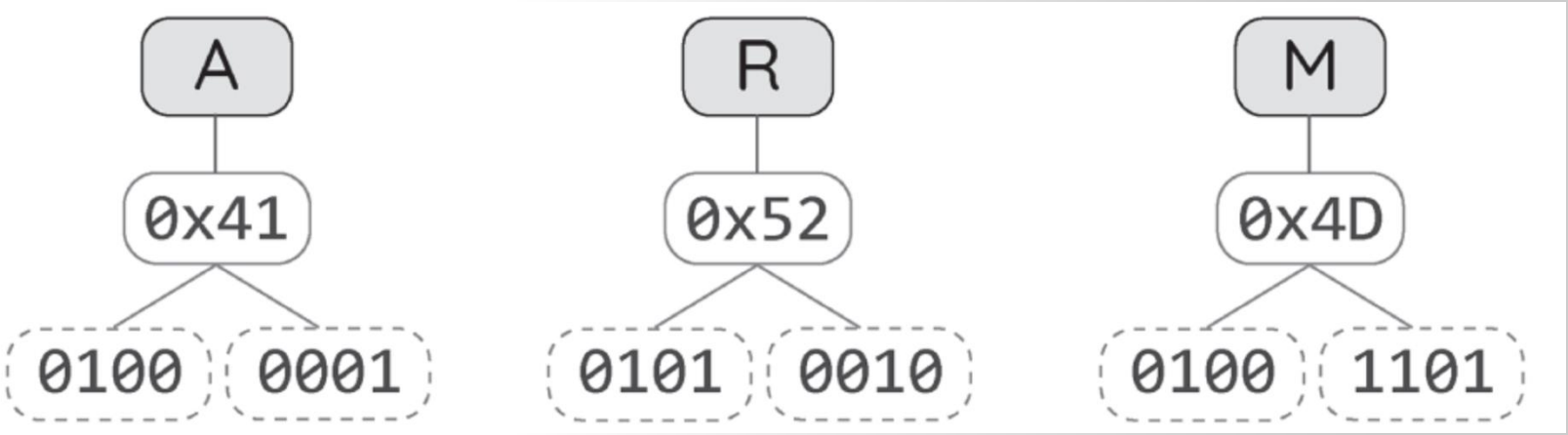
Class 2

CSC 590 ARM Architecture and Security
ARM Instruction Set

Dr. Si Chen (schen@wcupa.edu)

Introduction to Assembly

- Bits and Bytes
  - 1 Byte = ? Bits
    8
- Character Encoding



Little endian

- IA-32 processors use "little endian" as their byte order. This means that the bytes of a word are numbered starting from the least significant byte and that the least significant bit starts of a word starts in the least significant byte.

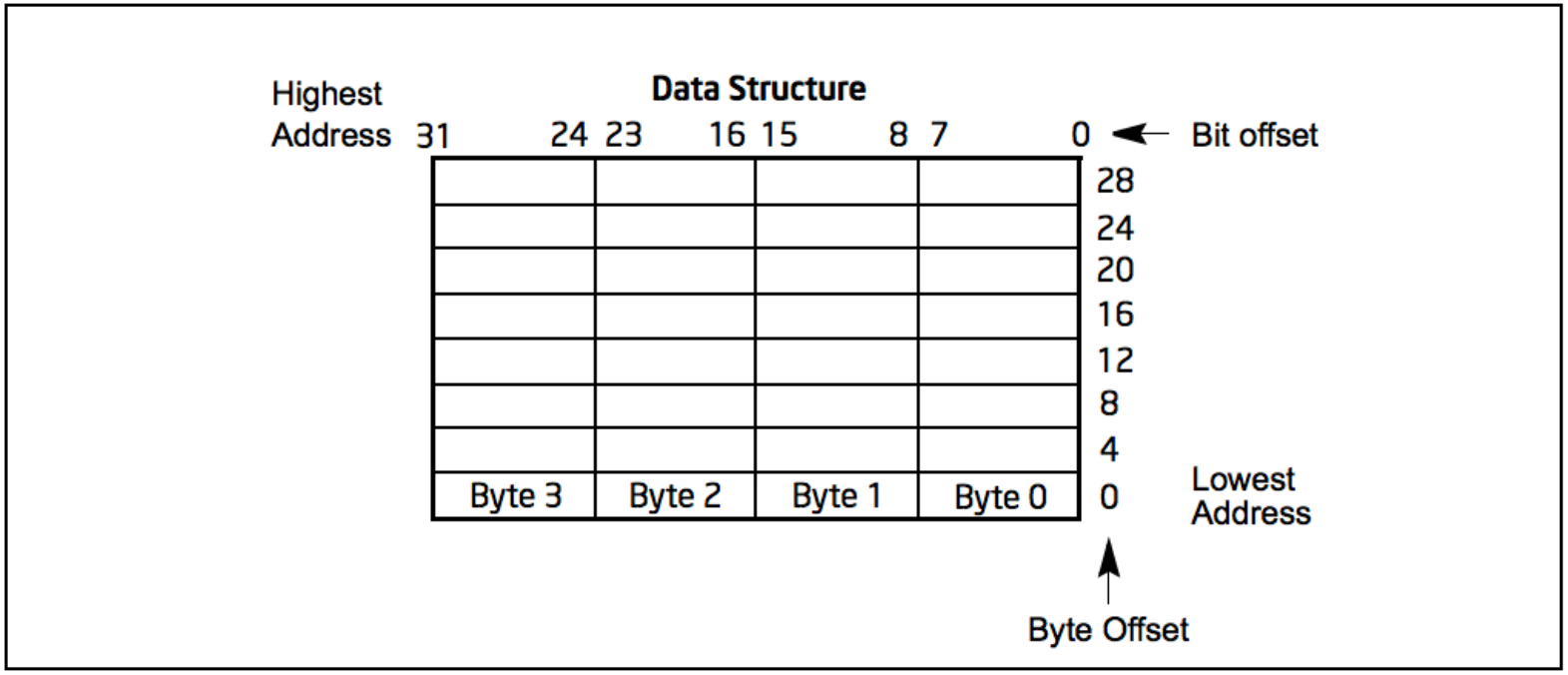
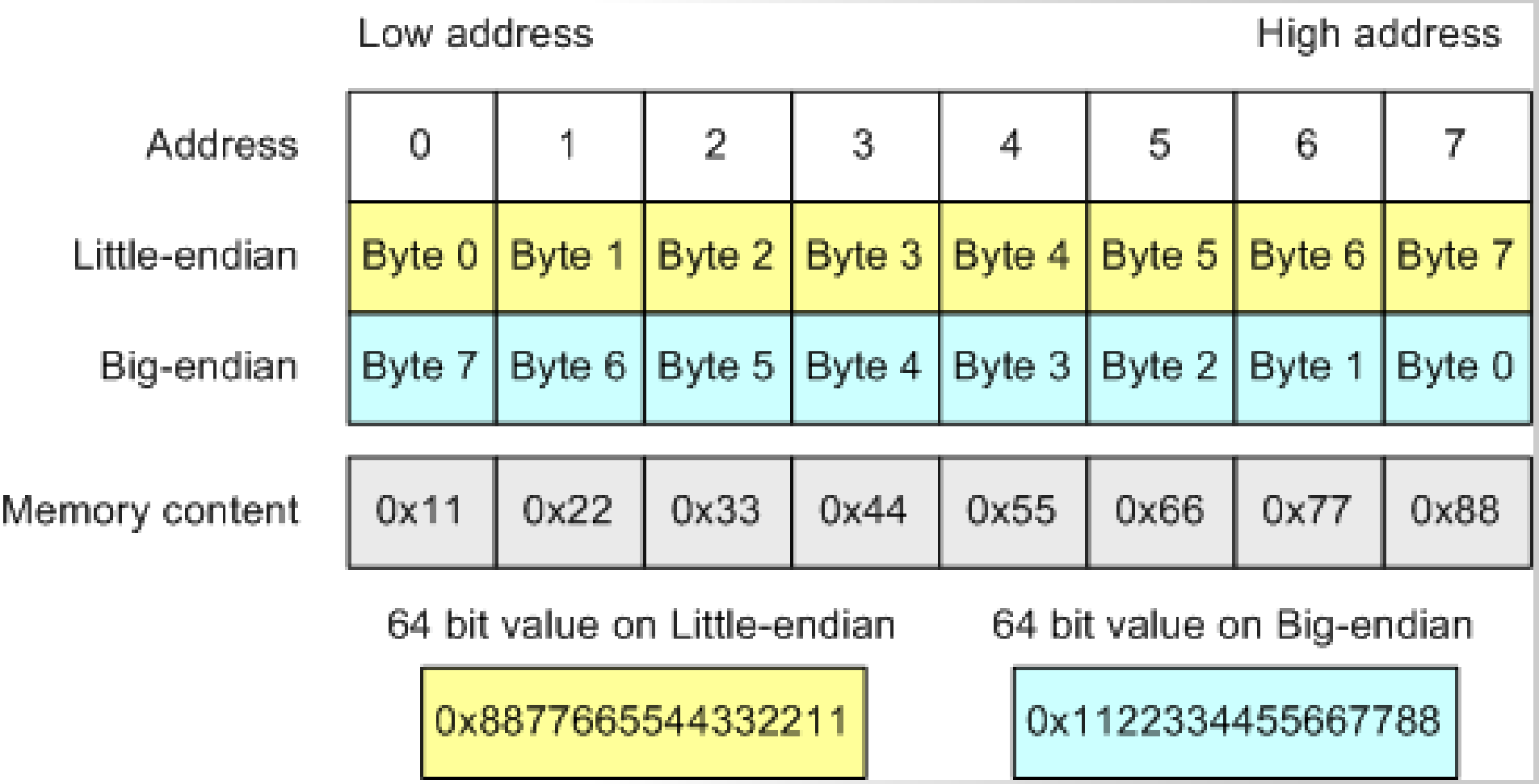


Figure 1-1. Bit and Byte Order

Byte Order


Low address
High address
Address
0
1
2
3
4
5
6
7
Little-endian
Byte 0
Byte 1
Byte 2
Byte 3
Byte 4
Byte 5
Byte 6
Byte 7
Big-endian
Byte 7
Byte 6
Byte 5
Byte 4
Byte 3
Byte 2
Byte 1
Byte 0
Memory content
0x11
0x22
0x33
0x44
0x55
0x66
0x77
0x88
64 bit value on Little-endian
0x8877665544332211
64 bit value on Big-endian
0x1122334455667788

Byte Order

- Both Apple silicon and Intel-based Mac computers use the little-endian format for data, so you don’t need to make endian conversions in your code.
- In general, the ARM architecture was **little-endian before version 3**. Since then ARM processors became **BI-endian** and feature a setting which **allows for switchable endianness**.

Porting Your macOS Apps to Apple Silicon:
https://developer.apple.com/documentation/apple-silicon/porting-your-macos-apps-to-apple-silicon

little_endian.s

```
1       .section .data
2       .align 4
3   value:
4       .word 0x04030201
5
6       .section .text
7       .globl _start
8   _start:
9       ldr r0, =value      ; Load the address of value into r0
10      ldr r1, [r0]        ; Load the 4-byte value from memory into r1
```

What does the memory content at [r0] look like?

little_endian.s

```
[ Legend: Modified register | Code | Heap | Stack | String ]
registers
$r0  : 0x11080
$r1  : 0x4030201
$r2  : 0x0
$r3  : 0x0
$r4  : 0x0
$r5  : 0x0
$r6  : 0x0
$r7  : 0x0
$r8  : 0x0
$r9  : 0x0
$r10 : 0x11080
$r11 : 0x0
$r12 : 0x0
$sp  : 0x408005c0
$lr  : 0x0
$pc  : 0x1007c
$cpsr: [negative zero carry overflow interrupt fast thumb]
stack
[!] Unmapped address: '0x408005c0'
code:arm:ARM
      0x10070                 andeq   r1,  r0,  r0
      0x10074 <_start+0>      ldr     r0,  [pc,  #-0]   @ 0x1007c <_start+8>
      0x10078 <_start+4>      ldr     r1,  [r0]
 →    0x1007c <_start+8>      andeq   r1,  r1,  r0,  lsl #1     NOT taken [Reason: !(Z)]
      0x10080                 streq   r0,  [r3],  #-513 @ 0xfffffdff
      0x10084                 andeq   r1,  r0,  r1,  asr #2
      0x10088                 cmnvs   r5,  r0,  lsl #2
      0x1008c                 tsteq   r0,  r2,  ror #18
      0x10090                 andeq   r0,  r0,  r7
memory:0x11080
0x00011080 <value+0000>    01 02 03 04 41 11 00 00 00 61    ....A....a
threads
[#0] Id 1, stopped 0x1007c in _start (), reason: SINGLE STEP
trace
[#0] 0x1007c → _start()
gef➤
```

```
1      .section .data
2      .align 4
3  value:
4      .word 0x04030201
5
6      .section .text
7      .globl _start
8  _start:
9      ldr r0, =value      ; Load the address of value into r0
10     ldr r1, [r0]        ; Load the 4-byte value from memory into r1
```

```
0x11080:        0x01
(remote) gef➤  x/16xb 0x0011080
0x11080:        0x01    0x02    0x03    0x04    0x41    0x11    0x00    0x00
0x11088:        0x00    0x61    0x65    0x61    0x62    0x69    0x00    0x01
```

big_endian.s

```
1     .section .data
2     .align 4
3   value:
4     .word 0x04030201
5
6     .section .text
7     .globl _start
8   _start:
9     ldr r0, =value  @ Load the address of value into r0
10    ldr r1, [r0]    @ Load the 4-byte value from memory into r1
11    rev r2, r1      @ Reverse the byte order
```

big_endian.s

```
1     .section .data
2     .align 4
3   value:
4     .word 0x04030201
5
6     .section .text
7     .globl _start
8   _start:
9     ldr r0, =value  @ Load the address of value into r0
10    ldr r1, [r0]    @ Load the 4-byte value from memory into r1
11    rev r2, r1      @ Reverse the byte order
```

big_endian.s

```
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────── registers ────
$r0   : 0x11090
$r1   : 0x4030201
$r2   : 0x1020304
$r3   : 0x0
$r4   : 0x0
$r5   : 0x0
$r6   : 0x0
$r7   : 0x0
$r8   : 0x0
$r9   : 0x0
$r10  : 0x11090
$r11  : 0x0
$r12  : 0x0
$sp   : 0x408005d0
$lr   : 0x0
$pc   : 0x10080
$cpsr: [negative zero carry overflow interrupt fast thumb]
──────────────────────────────────────────────────────────── stack ────
[!] Unmapped address: '0x408005d0'
───────────────────────────────────────────────────── code:arm:ARM ────
      0x10074 <_start+0>       ldr    r0,  [pc,  #4]    @ 0x10080 <_start+12>
      0x10078 <_start+4>       ldr    r1,  [r0]
      0x1007c <_start+8>       rev    r2,  r1
 →    0x10080 <_start+12>      muleq  r1,  r0,  r0      NOT taken [Reason: !(Z)]
      0x10084                  andeq  r0,  r0,  r0
      0x10088                  andeq  r0,  r0,  r0
      0x1008c                  andeq  r0,  r0,  r0
      0x10090                  streq  r0,  [r3],  #-513 @ 0xfffffdff
      0x10094                  andeq  r1,  r0,  r1,  asr #6
────────────────────────────────────────────────── memory:0x11090 ────
0x00011090 <value+0000>    01 02 03 04 41 13 00 00 00 61    ....A....a
────────────────────────────────────────────────────────── threads ────
[#0] Id 1, stopped 0x10080 in _start (), reason: SINGLE STEP
──────────────────────────────────────────────────────────── trace ────
[#0] 0x10080 → _start()
───────────────────────────────────────────────────────────────────────
gef➤
```

```
    .section .data
    .align 4
value:
    .word 0x04030201

    .section .text
    .globl _start
_start:
    ldr r0, =value  @ Load the address of value into r0
    ldr r1, [r0]    @ Load the 4-byte value from memory into r1
    rev r2, r1      @ Reverse the byte order
```

ARM programmer model

- The state of an ARM system is determined by the content of visible registers and memory.
- A user-mode program can see 15 32-bit general purpose registers (R0 -R14), program counter R15 (PC) and CPSR.
- Instruction set defines the operations that can change the state.

#	Alias	Purpose
R0	–	General purpose
R1	–	General purpose
R2	–	General purpose
R3	–	General purpose
R4	–	General purpose
R5	–	General purpose
R6	–	General purpose
R7	–	Holds Syscall Number
R8	–	General purpose
R9	–	General purpose
R10	–	General purpose
R11	FP	Frame Pointer
Special Purpose Registers		
R12	IP	Intra Procedural Call
R13	SP	Stack Pointer
R14	LR	Link Register
R15	PC	Program Counter
CPSR	–	Current Program Status Register

ARM (AArch32) REGISTERS

#	Alias	Purpose
R0	–	General purpose
R1	–	General purpose
R2	–	General purpose
R3	–	General purpose
R4	–	General purpose
R5	–	General purpose
R6	–	General purpose
R7	–	Holds Syscall Number
R8	–	General purpose
R9	–	General purpose
R10	–	General purpose
R11	FP	Frame Pointer
Special Purpose Registers		
R12	IP	Intra Procedural Call
R13	SP	Stack Pointer
R14	LR	Link Register
R15	PC	Program Counter
CPSR	–	Current Program Status Register

ARM	Description	x86
R0	General Purpose	EAX
R1-R5	General Purpose	EBX, ECX, EDX, ESI, EDI
R6-R10	General Purpose	–
R11 (FP)	Frame Pointer	EBP
R12	Intra Procedural Call	–
R13 (SP)	Stack Pointer	ESP
R14 (LR)	Link Register	–
R15 (PC)	<- Program Counter / Instruction Pointer ->	EIP
CPSR	Current Program State Register/Flags	EFLAGS

Features of ARM instruction set

- **Load-Store Architecture Characteristics:**

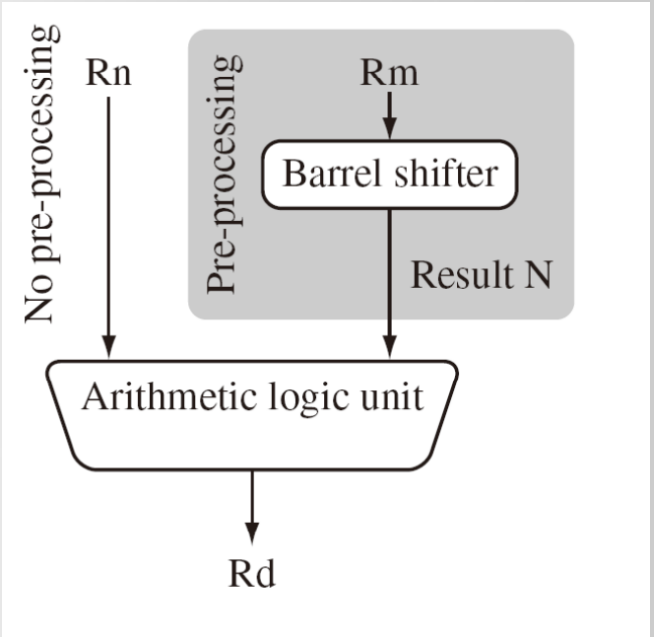
1. **Three-Address Instructions**: The architecture employs three-address instruction formats, allowing operations to specify two source operands and one destination operand explicitly.
2. **Conditional Execution of Instructions**: Each instruction can be conditionally executed based on the values in specific condition registers or flags.
3. **Multiple Register Load/Store Capability**: The architecture supports instructions that can simultaneously load or store multiple registers.
4. **Combined Shift and ALU Operations**: It is possible to perform a shift operation in conjunction with an Arithmetic Logic Unit (ALU) operation within a single instruction.

Instruction set

- Data Processing
- Data Movement & Memory Access
- Flow Control

Data processing

- Types of Instructions: Move, arithmetic, logical, comparison, and multiply instructions.
- Data Processing: Most data processing instructions can process one of their operands using the barrel shifter.
- General Rules:
  - All operands are 32-bit, coming from registers or literals.
  - The result, if any, is 32-bit and placed in a register (with the exception for long multiply, which produces a 64-bit result).
  - Three-address format.



MOV

immediate,register,shift

MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$

- **MOV R0, R2 @ R0 = R2**
- **MVN R0, R2 @ R0 = ~R2**

Two-Operand Instructions

```
PRE     r5 = 5
        r7 = 8
        MOV     r7, r5      ; let r7 = r5
POST    r5 = 5
        r7 = 5
```

MVN

immediate,register,shift

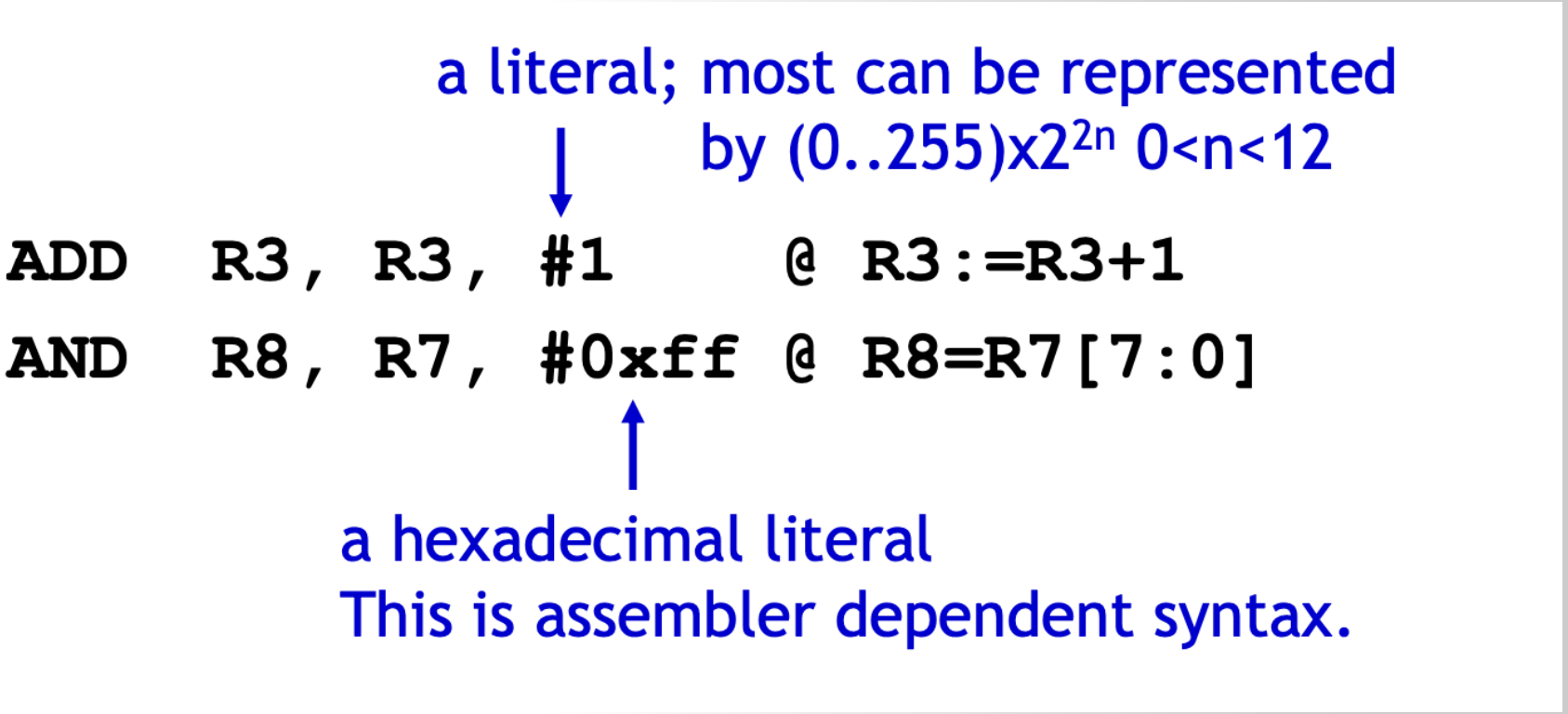
MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$

```
1          .section .text
2          .global _start
3
4   _start:
5          MOV R1, #0x0000000A  ; Load immediate value 0x0000000A into R1
6          MVN R0, R1          ; Bitwise negate R1 and store the result in R0
7
8          /* Exit Program */
9          MOV R7, #1           ; Load the exit syscall number into R7
10         SWI 0                ; Make the syscall to exit
```

What's the value in R0?

Addressing modes (ADD & AND)

Register operands ADD R0 R1 R2



Three-Operand Instructions

Single register load/store

```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing[1]
        LDR{<cond>}SB|H|SH Rd, addressing[2]
        STR{<cond>}H Rd, addressing[2]
```

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*

LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

Single register load/store

- The data items can be a 8-bit byte, 16-bit halfword or 32-bit word Addresses must be bit word. Addresses must be boundary aligned. (e.g. 4's multiple for LDR/STR)

```
LDR   R0, [R1]    @ R0 := mem32[R1]
STR   R0, [R1]    @ mem32[R1] := R0
```

LDR, LDRH, LDRB for 32, 16, 8 bits

STR, STRH, STRB for 32, 16, 8 bits

Example (ldr_str_example.s)

```
1       .section .data
2   value:  .word 42
3
4       .section .text
5       .global _start
6
7   _start:
8       LDR R0, =value  ; Load the address of 'value' into R0
9       LDR R1, [R0]    ; Load the value from the address in R0 into R1
10
11      ADD R1, R1, #1   ; Increment the value in R1 by 1
12
13      STR R1, [R0]     ; Store the updated value back into the address in R0
14
15      /* Exit Program */
16      MOV R7, #1       ; Load the exit syscall number into R7
17      SWI 0            ; Make the syscall to exit
```

what's the value inside R1, R0 and [R0]?

1. Save the code to a file named ldr_str_example.s.
2. Assemble it: ***arm-linux-gnueabi-as -o ldr_str_example.o ldr_str_example.s***
3. Link it: ***arm-linux-gnueabi-ld -o ldr_str_example ldr_str_example.o***

How to DEBUG

1. **Open a Terminal and Start Tmux**

   Type `tmux` and hit enter.

2. **Create the First Pane (Window)**

   By default, you should already be in a new Tmux pane.

3. **Split the Pane**

   - For a horizontal split, press `Ctrl-b` followed by `%`
   - For a vertical split, press `Ctrl-b` followed by `"`

   This will split your current pane into two separate "panes."

4. **Run QEMU in One of the Panes** qemu-arm –g 1234 /path/to/your/binary

   In one pane, enter the command you use to start QEMU.

5. **Run GDB in the Other Pane** gdb-multiarch /path/to/your/binary

   In the other pane, enter the command to start GDB.

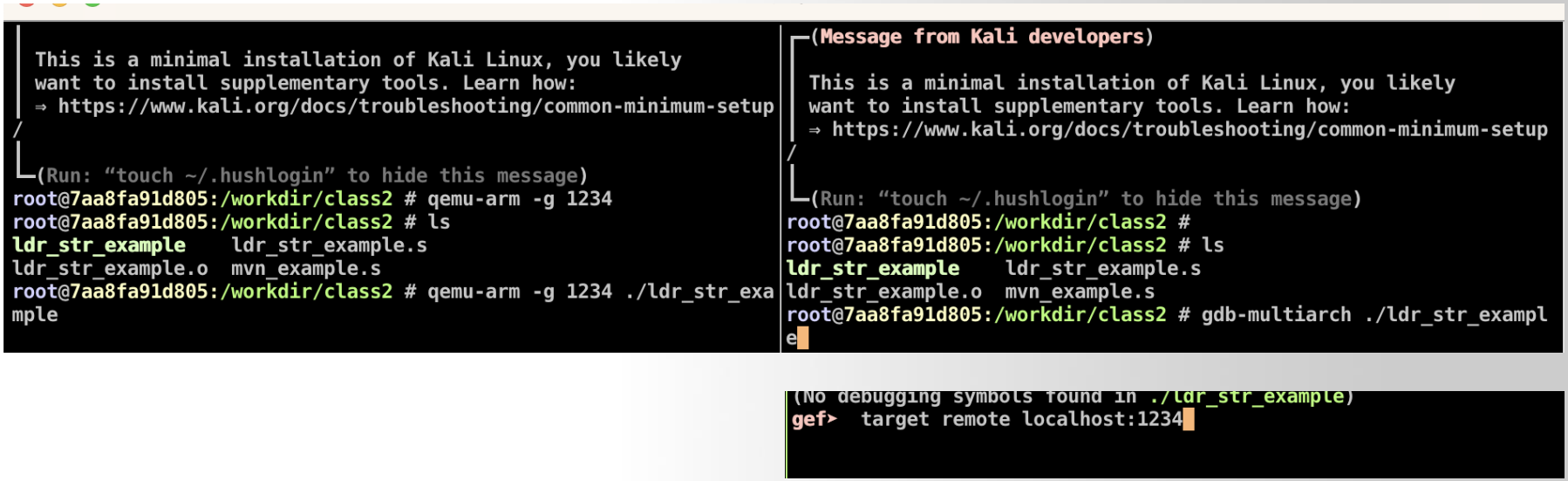
6. **Connect GDB to QEMU**

   If you've started QEMU with a GDB server (usually with the `-s -S` options), you can connect to it in GDB with `target remote :1234` (or whatever port you set).

7. **Switch Between Panes**

   You can switch between the panes using `Ctrl-b` followed by the arrow keys.

West Chester University


This is a minimal installation of Kali Linux, you likely
want to install supplementary tools. Learn how:
⇒ https://www.kali.org/docs/troubleshooting/common-minimum-setup
(Run: "touch ~/.hushlogin" to hide this message)
root@7aa8fa91d805:/workdir/class2 # qemu-arm -g 1234
root@7aa8fa91d805:/workdir/class2 # ls
ldr_str_example ldr_str_example.s
ldr_str_example.o mvn_example.s
root@7aa8fa91d805:/workdir/class2 # qemu-arm -g 1234 ./ldr_str_example
(Message from Kali developers)
This is a minimal installation of Kali Linux, you likely
want to install supplementary tools. Learn how:
⇒ https://www.kali.org/docs/troubleshooting/common-minimum-setup
(Run: "touch ~/.hushlogin" to hide this message)
root@7aa8fa91d805:/workdir/class2 #
root@7aa8fa91d805:/workdir/class2 # ls
ldr_str_example ldr_str_example.s
ldr_str_example.o mvn_example.s
root@7aa8fa91d805:/workdir/class2 # gdb-multiarch ./ldr_str_example
(No debugging symbols found in ./ldr_str_example)
gef➤ target remote localhost:1234

hello_world.s

```
1   .data                   /* section declaration */
2   hello_string:
3       .asciz "Hello, World!\n"  /* null-terminated string */
4
5   .text                   /* section declaration */
6   .global _start          /* entry point for the program */
7
8   _start:                 /* _start label */
9       /* Write syscall */
10      mov r0, #1          /* file descriptor (stdout) */
11      ldr r1, =hello_string /* pointer to the string */
12      mov r2, #14         /* string length */
13      mov r7, #4          /* syscall number for sys_write */
14      swi 0               /* invoke syscall */
15
16      /* Exit syscall */
17      mov r0, #0          /* exit status */
18      mov r7, #1          /* syscall number for sys_exit */
19      swi 0               /* invoke syscall */
```

hello_world.s

```
1   .data                   /* section declaration */
2   hello_string:
3       .asciz "Hello, World!\n"  /* null-terminated string */
4
5   .text                   /* section declaration */
6   .global _start          /* entry point for the program */
7
8   _start:                 /* _start label */
9       /* Write syscall */
10      mov r0, #1          /* file descriptor (stdout) */
11      ldr r1, =hello_string /* pointer to the string */
12      mov r2, #14         /* string length */
13      mov r7, #4          /* syscall number for sys_write */
14      swi 0               /* invoke syscall */
15
16      /* Exit syscall */
17      mov r0, #0          /* exit status */
18      mov r7, #1          /* syscall number for sys_exit */
19      swi 0               /* invoke syscall */
```

```
schen@molly:~$ arm-linux-gnueabi-as -o hello_world.o hello_world.s
schen@molly:~$ arm-linux-gnueabi-ld -o hello_world hello_world.o
schen@molly:~$ qemu-arm hello_world
Hello, World!
```

https://arm.syscall.sh/

Stdin, stdout, and stderr

Name	File descriptor	Abbreviation
Standard input	0	stdin
Standard output	1	stdout
Standard error	2	stderr

Q & A