Class 3

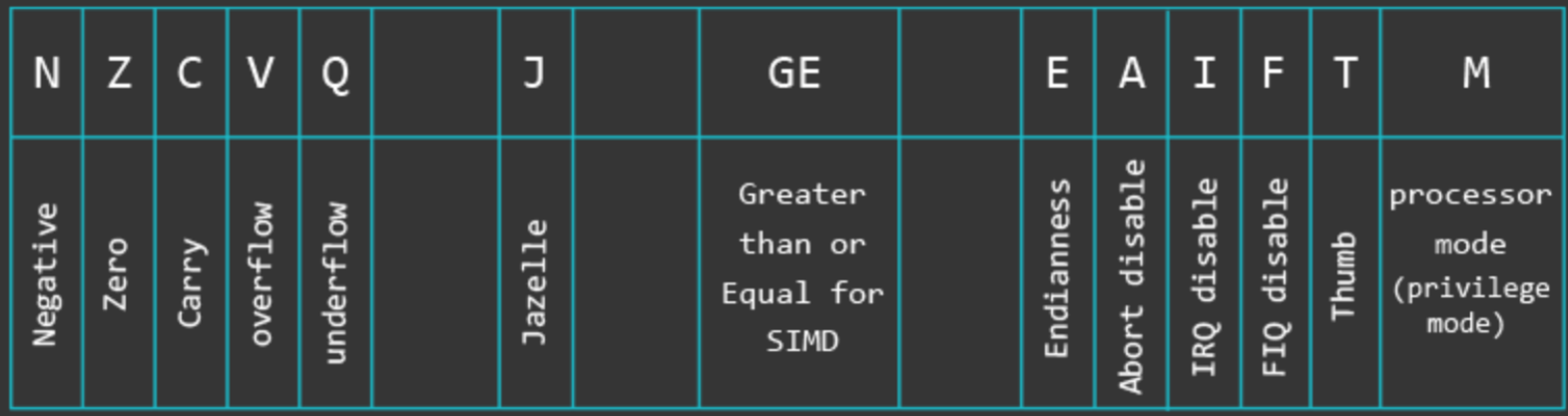
CSC 590 ARM Architecture and Security
ARM Instruction Set (2): str/ldr, B, BL

Dr. Si Chen (schen@wcupa.edu)

CPSR

CPSR

(Current Program Status Register)



Quick Review

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

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	$Rd = \sim N$

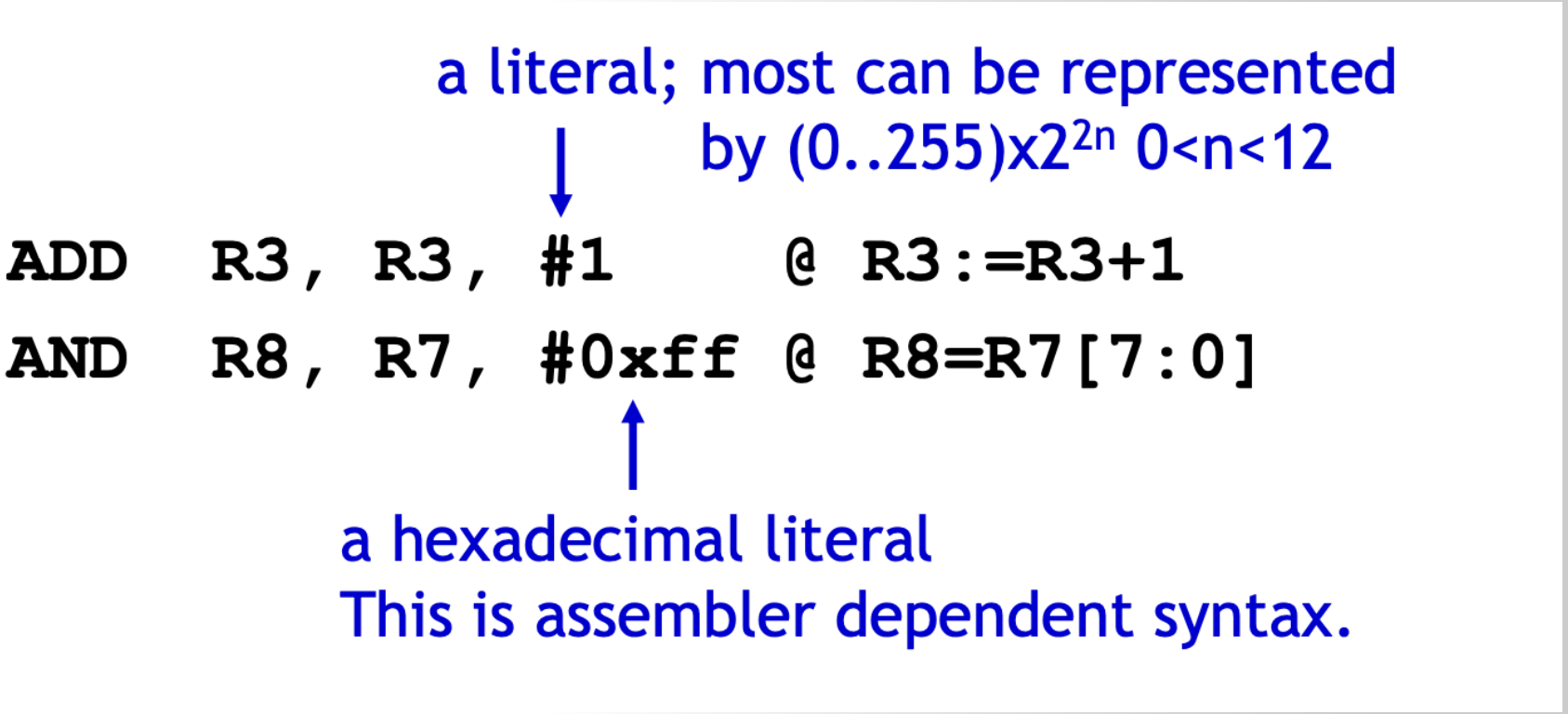
- **MOV R0, R2 @ R0 = R2**
- **MVN R0, R2 @ R0 = ~R2**

Two-Operand Instructions

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

Addressing modes (ADD & AND)

Register operands ADD R0 R1 R2



Three-Operand Instructions

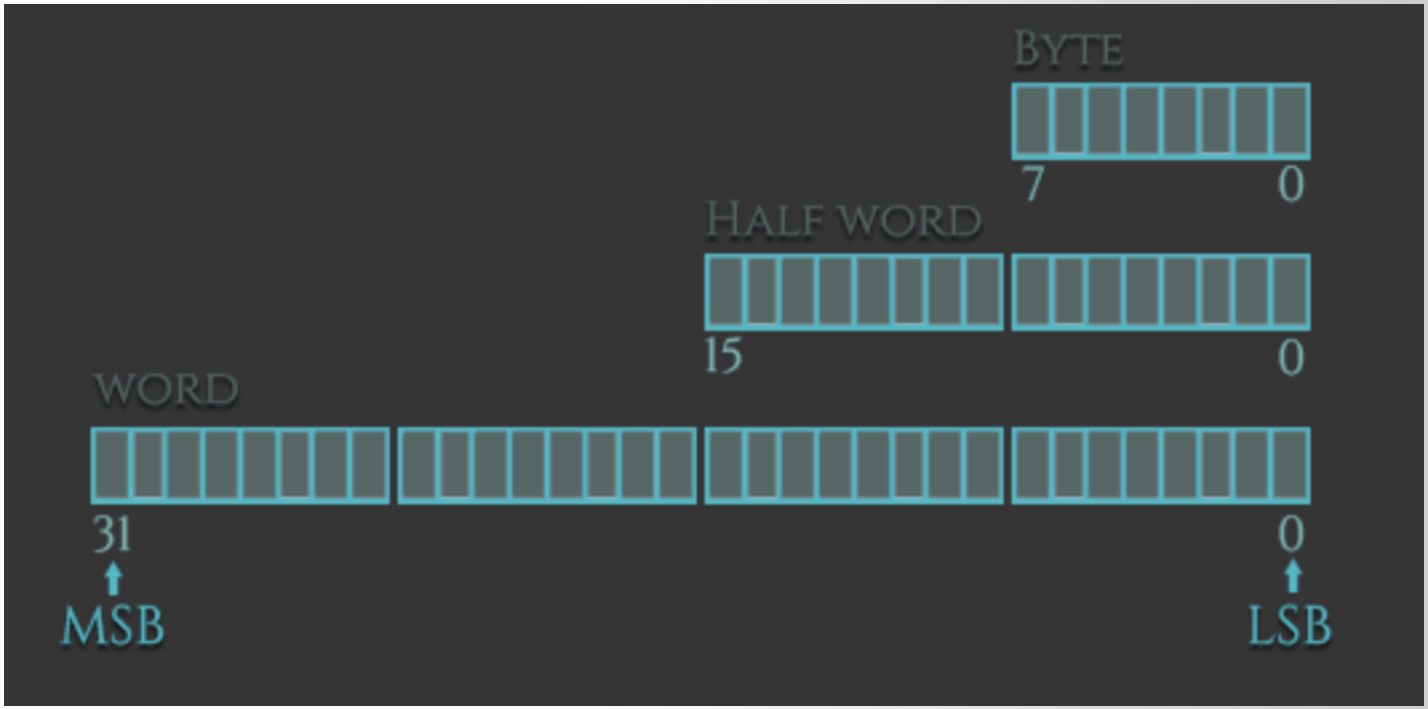
Single register load/store

```
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])*

DATA TYPES



Similar to high level languages, ARM supports operations on different datatypes.

The data types we can load (or store) can be signed and unsigned words, halfwords, or bytes. The extensions for these data types are: -h or -sh for halfwords, -b or -sb for bytes, and no extension for words.

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.

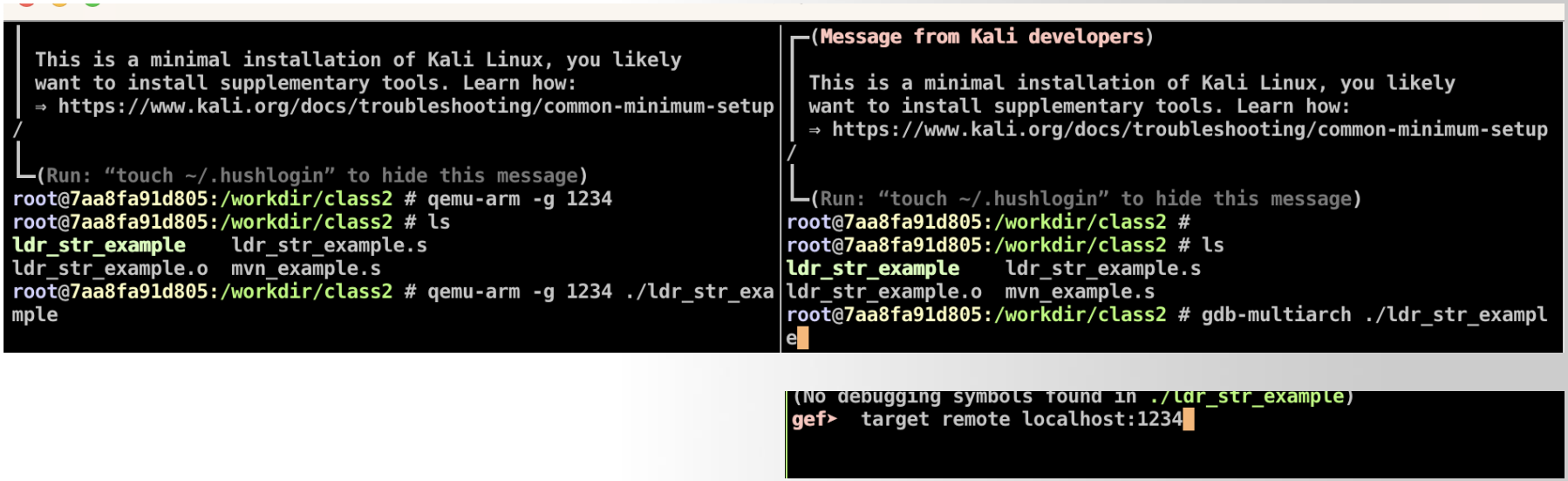
6. **Connect GDB to QEMU**

   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

Comparison Between x86 and ARM Instructions

- ARM's **RISC (Reduced Instruction Set Computer)** architecture is different from Intel X86's **CISC (Complex Instruction Set Computer)** design.
- E.g., In the x86 architecture, you can indeed **directly modify data in memory**.
  - For example, using the **MOV** instruction, you can directly move data from a register to a memory address, or from a memory address to a register.
  - Such operations in the ARM architecture usually require **two steps**:
    - first, use the **LDR** instruction to load data from memory into a register,
    - then operate on that register, and finally use the **STR** instruction to store the data back into memory from the register.

Comparison Between x86 and ARM Instructions

- E.g., In the x86 architecture, you can indeed **directly modify data in memory**.
  - For example, using the **MOV** instruction, you can directly move data from a register to a memory address, or from a memory address to a register.
  - Such operations in the ARM architecture usually require **two steps**:
    - first, use the **LDR** instruction to load data from memory into a register,
    - then operate on that register, and finally use the **STR** instruction to store the data back into memory from the register.

x86 Instruction

```
MOV [0x30008000], EAX
; Store the contents of the EAX register directly into the memory address 0x30008000
```

ARM Instruction

```
LDR R0, [R1]  ; Load data from the memory address pointed to by the R1 register into the R0 register
STR R0, [R2]  ; Store the contents of the R0 register into the memory address pointed to by the R2 register
```

The B Instruction

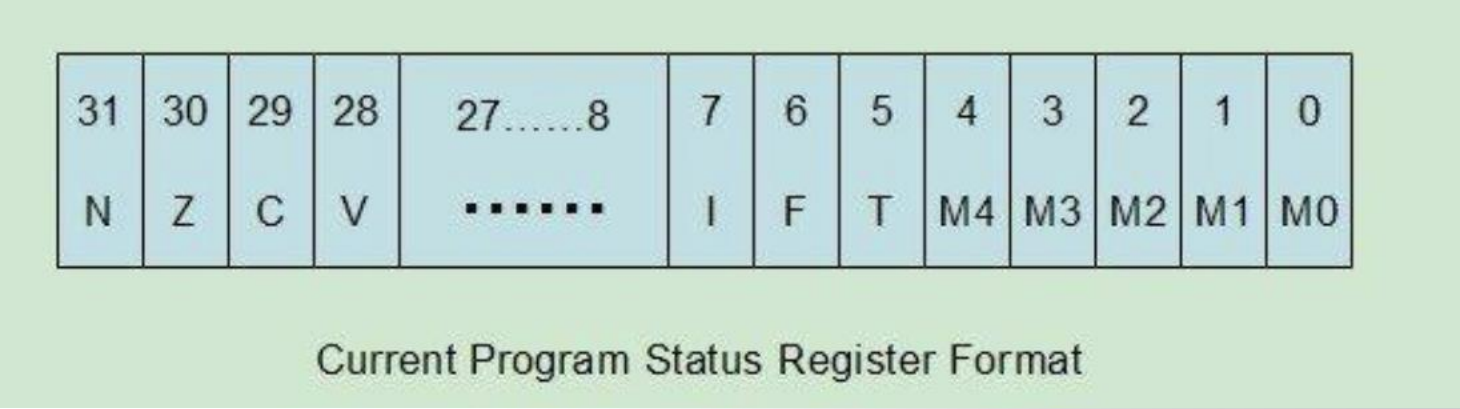
- The B instruction is one of the most fundamental branching instructions in ARM assembly language. Its basic syntax is as follows:

```
B label
```

- This statement indicates that the program should jump to the location specified by the label.
- The B instruction is a simple, relative jump, meaning it jumps to an address relative to the current Program Counter (PC).
- It can jump to an address within a **32MB** range, both forward and backward, from the current PC. Therefore, the B instruction is typically used for jumps between nearby code blocks or labels.

The B Instruction

- **Conditional Execution**
- Like other ARM instructions, the B instruction can be conditionally executed based on the flags in the CPSR (Current Program Status Register).
- This allows for more efficient code by reducing the number of instructions and increasing code density.
- For example, **BNE** and **BEQ** are conditional jumps that occur when the result is **not equal** or **equal**, respectively.



Current Program Status Register Format

Condition Code

Condition Code	Meaning (for cmp or subs)	Status of Flags
EQ	Equal	Z==1
NE	Not Equal	Z==0
GT	Signed Greater Than	(Z==0) && (N==V)
LT	Signed Less Than	N!=V
GE	Signed Greater Than or Equal	N==V
LE	Signed Less Than or Equal	(Z==1) || (N!=V)
CS or HS	Unsigned Higher or Same (or Carry Set)	C==1
CC or LO	Unsigned Lower (or Carry Clear)	C==0
MI	Negative (or Minus)	N==1
PL	Positive (or Plus)	N==0
AL	Always executed	–
NV	Never executed	–
VS	Signed Overflow	V==1
VC	No signed Overflow	V==0
HI	Unsigned Higher	(C==1) && (Z==0)
LS	Unsigned Lower or same	(C==0) || (Z==0)

The B Instruction: Example

```
1    LOOP
2        SUB R0,R0,#1
3        ...
4        CMP R0,#0
5        BNE LOOP
```

Example: Loop Structure

```
        MOV R1,#10
        MOV R2,#20
        CMP R1,R2
        BEQ HERE
        ...
        B END
HERE
        ...
END
        ...
```

Example: Conditional Structure

Exercise: Simple Loop

```
.section .text
.global _start

_start:
    mov r0, #10     /* Number of iterations */
    mov r1, #0      /* Sum */

loop1:
    add r1, r1, r0  /* Add current number to sum */
    subs r0, r0, #1 /* Decrement counter */
    bne loop1       /* If counter is not zero, loop */

    /* Exit the program */
    mov r0, #0  /* Exit code */
    mov r7, #1  /* sys_exit */
    swi 0       /* Invoke syscall */
```

What is the value stored in register r1 before the program exits?

Exercise: Conditional Jump

```
.section .text
.global _start

_start:
    mov r0, #5
    mov r1, #10

    cmp r0, r1
    blt less
    bge greater_or_equal

less:
    mov r2, #0
    b end

greater_or_equal:
    mov r2, #1

end:
    /* Exit the program */
    mov r0, #0  /* Exit code */
    mov r7, #1  /* sys_exit */
    swi 0       /* Invoke syscall */
```

What is the value stored in register r2 before the program exits?

The BL Instruction

- The BL (Branch with Link) instruction in ARM assembly language serves a different purpose than the simple B (Branch) instruction.
- The BL instruction performs a jump to a label, **but before doing so, it saves the address of the next instruction to the Link Register (LR).**

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

The BL Instruction

- **Key Features**
  - **Return Address**: Before jumping, the BL instruction saves the address of the next instruction in the LR register.
  - **Subroutine Calls**: The BL instruction is commonly used for **calling subroutines**, whether in assembly or in C language.
  - **Return to Main Program**: After the subroutine is executed, the address saved in the LR register can be moved back to the Program Counter (PC) to continue the main program's execution.

The BL Instruction

Example in Assembly

```
BEGIN
    MOV R0,#SRC
    MOV R1,#DST
    MOV R2,#100
    BL COPY
    NOP
    ...
COPY
    SUB R2,R2,#1
    LDR R3,[R0],#1
    STR R3,[R1],#1
    CMP R2,#0
    BNE COPY
    MOV PC,LR
```

Commonly Used Registers:

- FP (Frame Pointer): Alias for R12
- SP (Stack Pointer): Alias for R13
- LR (Link Register): Alias for R14
- PC (Program Counter): Alias for R15

The BL Instruction

C Language Example

```c
int main(void)
{
    func();
    printf("Hello zhaixue.cc!\n");
    return 0;
}
```

```
main
    BL func
    BL printf
func
    PUSH LR
    ...
    POP PC
```

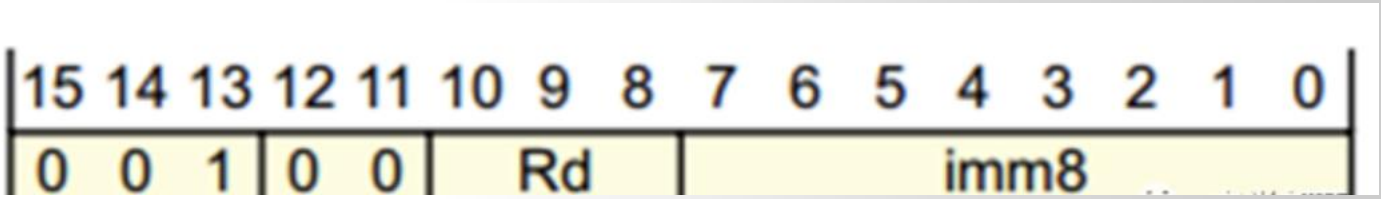
Commonly Used Registers:

- FP (Frame Pointer): Alias for R12
- SP (Stack Pointer): Alias for R13
- LR (Link Register): Alias for R14
- PC (Program Counter): Alias for R15

Jump with the MOV Instruction

- The MOV instruction in ARM assembly language is primarily used for transferring data between registers or moving an immediate value into a register. → Is it possible to use the MOV instruction to jump to any memory location??
- However, it has a limitation: the **immediate value that can be transferred is restricted to 8 bits**. This is because ARM follows the RISC (Reduced Instruction Set Computer) architecture, where each **instruction is typically 32 bits long**.
- **Understanding the Limitation**

In a 32-bit ARM instruction, the **opcode for the MOV operation itself takes up a few bits**, and the **destination register (Rd) encoding also consumes some bits**. This leaves limited space for the immediate value, which is why it's restricted to 8 bits.



Jump with the MOV Instruction

- **Why is this a Problem?**
- In a 32-bit program, the addresses for variables and functions are generally 32 bits long. If you try to use the MOV instruction to move such an address directly into the Program Counter (PC), you'll run into issues. For example:

MOV PC, #0x30008000

- Here, the immediate value **#0x30008000** is already 32 bits long. Adding the opcode for the MOV instruction would exceed the 32-bit limit, making it impossible for the compiler to translate the instruction.

Workarounds?

Jump with the MOV Instruction

Use Multiple Instructions: You can use a combination of instructions to load a 32-bit value into a register. For example, you can use LDR to load from a memory location where the 32-bit value is stored.

LDR PC, =0x30008000

Use Register Indirection: If the 32-bit address is already stored in another register, you can use that register to update the PC.

MOV R0, #0x30008000

MOV PC, R0

Use Link Register (LR): For return addresses, as seen in the BL instruction, you can use **MOV PC, LR** to move the saved return address from the LR register back to the PC.

Jump with the MOV Instruction

Use Multiple Instructions: You can use a combination of instructions to load a 32-bit value into a register. For example, you can use LDR to load from a memory location where the 32-bit value is stored.

LDR PC, =0x30008000

- **Pseudo-Instruction**: The presence of an equal sign "=" indicates that this is a **pseudo-instruction**, not a standard ARM instruction.
- **Memory Loading**: In addition to its role as a pseudo-instruction, LDR is also a standard ARM instruction used to load data from memory into a register.

Compiler Translation: Pseudo-instructions like LDR are not part of the standard ARM instruction set. They are mnemonics defined by the compiler to make programming easier. During compilation, these **pseudo-instructions are translated into standard ARM instructions**.
Optimization: If the immediate value in the LDR pseudo-instruction is less than 8 bits, the compiler may optimize it to a MOV instruction.

LDR R0,=200
MOV R0,#200

Exercise

```
.section .text
.global _start

_start:
    mov r0, #10     /* Number of iterations */
    mov r1, #0      /* Sum */
    ldr r2, =0x30008000  /* Memory address to store the sum */

    bl calculate_sum  /* Call subroutine to calculate sum */

    /* Store the sum in memory */
    str r1, [r2]

    /* Exit the program */
    mov r0, #0  /* Exit code */
    mov r7, #1  /* sys_exit */
    swi 0       /* Invoke syscall */

/* Subroutine to calculate sum */
calculate_sum:
    push {lr}  /* Save return address */

loop1:
    add r1, r1, r0  /* Add current number to sum */
    subs r0, r0, #1 /* Decrement counter */
    bne loop1       /* If counter is not zero, loop */

    pop {pc}  /* Return to caller */
```

Explanation and Questions for Students

1.The program starts with r0 set to 10 (the number of iterations) and r1 set to 0 (the sum).

2.The bl calculate_sum instruction calls a subroutine that calculates the sum of numbers from 1 to 10.

3.Inside the subroutine, the loop1 label is where the actual addition happens.

4.The sum is then stored in the memory address 0x30008000 using the STR instruction.

Questions

1.What will be the value in r1 after the subroutine calculate_sum is executed?

2.What will be stored at the memory address 0x30008000?

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