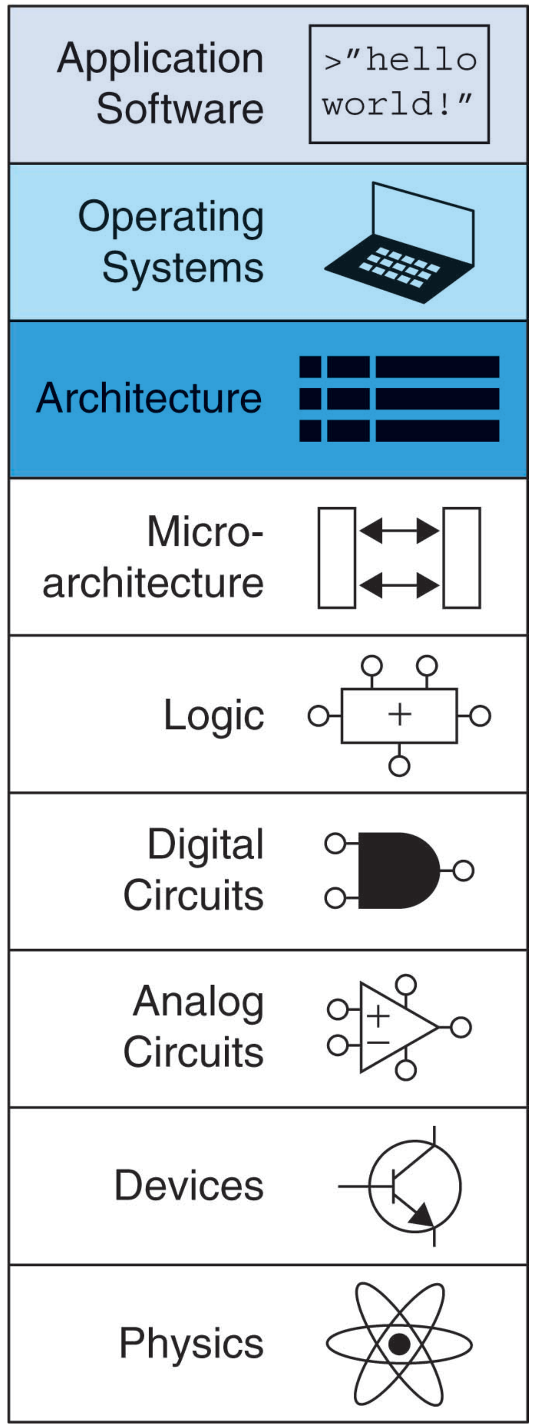
Architecture

Digital Computer Design

Architecture

- The architecture is the **programmer's view** of a computer.
  - It is defined by the
    - **instruction set** (language) and
    - **operand locations** (registers and memory).
- Many different architectures exist, such as
  - ARM, x86, MIPS, SPARC, and PowerPC.

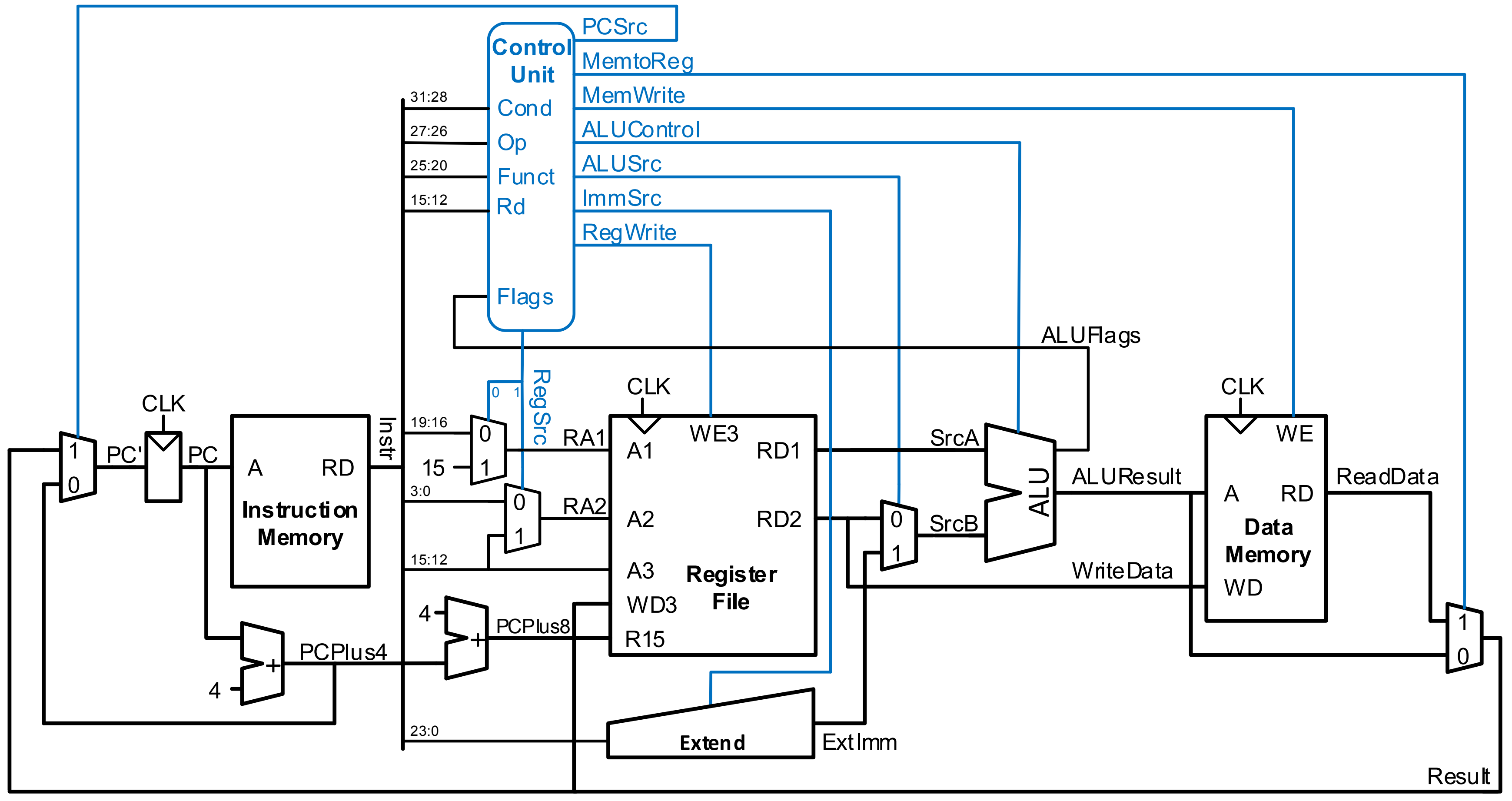


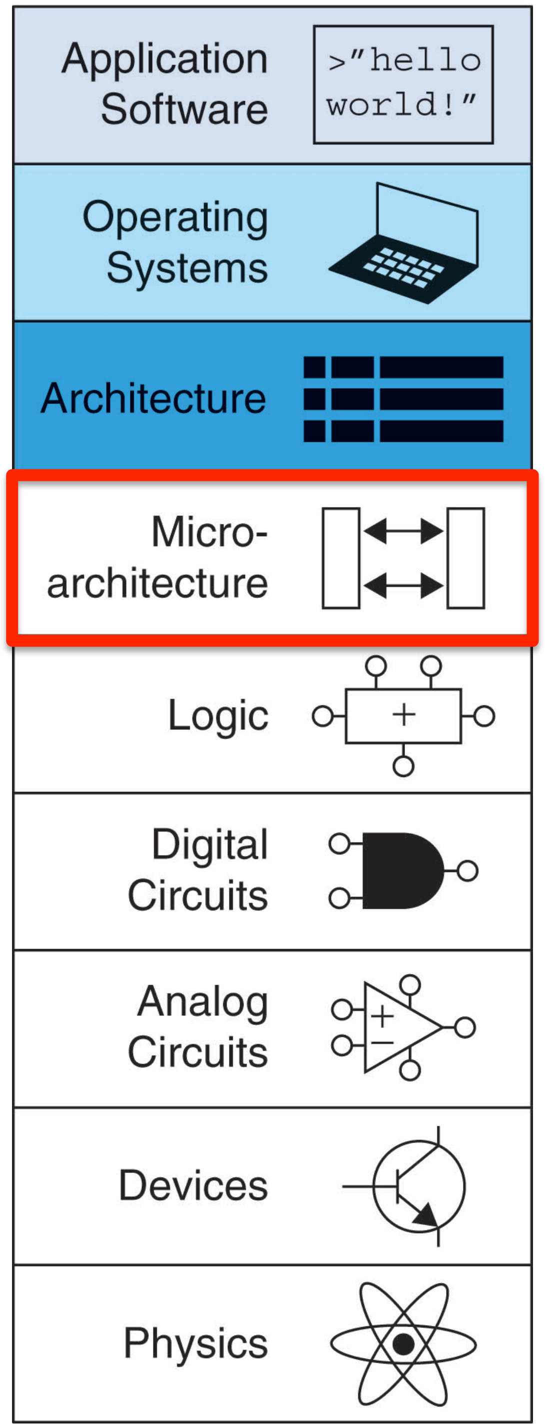
Understanding Computer Architecture

- The first step in understanding any computer architecture is to learn its language:
  - The words in a computer's language are called **instructions**.
  - The computer's vocabulary is called the **instruction set**.
- All programs running on a computer **use the same instruction set**.
  - All applications are eventually compiled into a series of simple instructions:
    - such as add, subtract, and branch.

Microarchitecture

- A computer architecture **does not define** the underlying hardware implementation.
  - Registers, memories, ALUs, and other building blocks to form a microprocessor is called the **microarchitecture.**





Microarchitecture

- Different microarchitectures may exist for a single architecture.
  - Intel and Advanced Micro Devices (AMD) both sell various microprocessors belonging to the same x86 architecture.
  - They all can run the same programs,
  - But they use different underlying hardware
    - Offer different trade-offs in performance, price, and power.
- We will explore microarchitecture in the following weeks!

ARM Architecture

- Developed in the 1980's by Advanced RISC Machines – now called ARM Holdings
- Almost all cell phones and tablets have multiple ARM processors
  - Over 75% of humans use products with an **ARM processor**
    - Used in servers, cameras, robots, cars, pinball machines, etc.

Machine Language

- Computer hardware understands only 1’s and 0’s
  - Instructions are encoded as binary numbers in a format called **machine language**.
  - The ARM architecture represents each instruction as a 32-bit word.

Assembly Language

- However, humans consider reading machine language to be tedious
  - We prefer to represent the instructions in a symbolic format called assembly language.
  - Each **assembly language instruction** specifies the **operation** to perform and the **operands** on which to operate

Instruction: Addition

C Code

```
a = b + c;
```

ARM Assembly Code

```
ADD a, b, c
```

- **ADD:** **mnemonic** – indicates operation to perform
- **b, c:** source operands
- **a:** destination operand

Instruction: Subtraction

Similar to addition - only mnemonic changes

C Code	**ARM assembly code**
`a = b - c;`	`SUB a, b, c`

- **SUB:** mnemonic
- **b, c:** source operands
- **a:** destination operand

Multiple Instructions

More complex code handled by multiple ARM instructions

C Code

```
a = b + c - d;
```

ARM assembly code

```
ADD t, b, c  ; t = b + c
SUB a, t, d  ; a = t - d
```

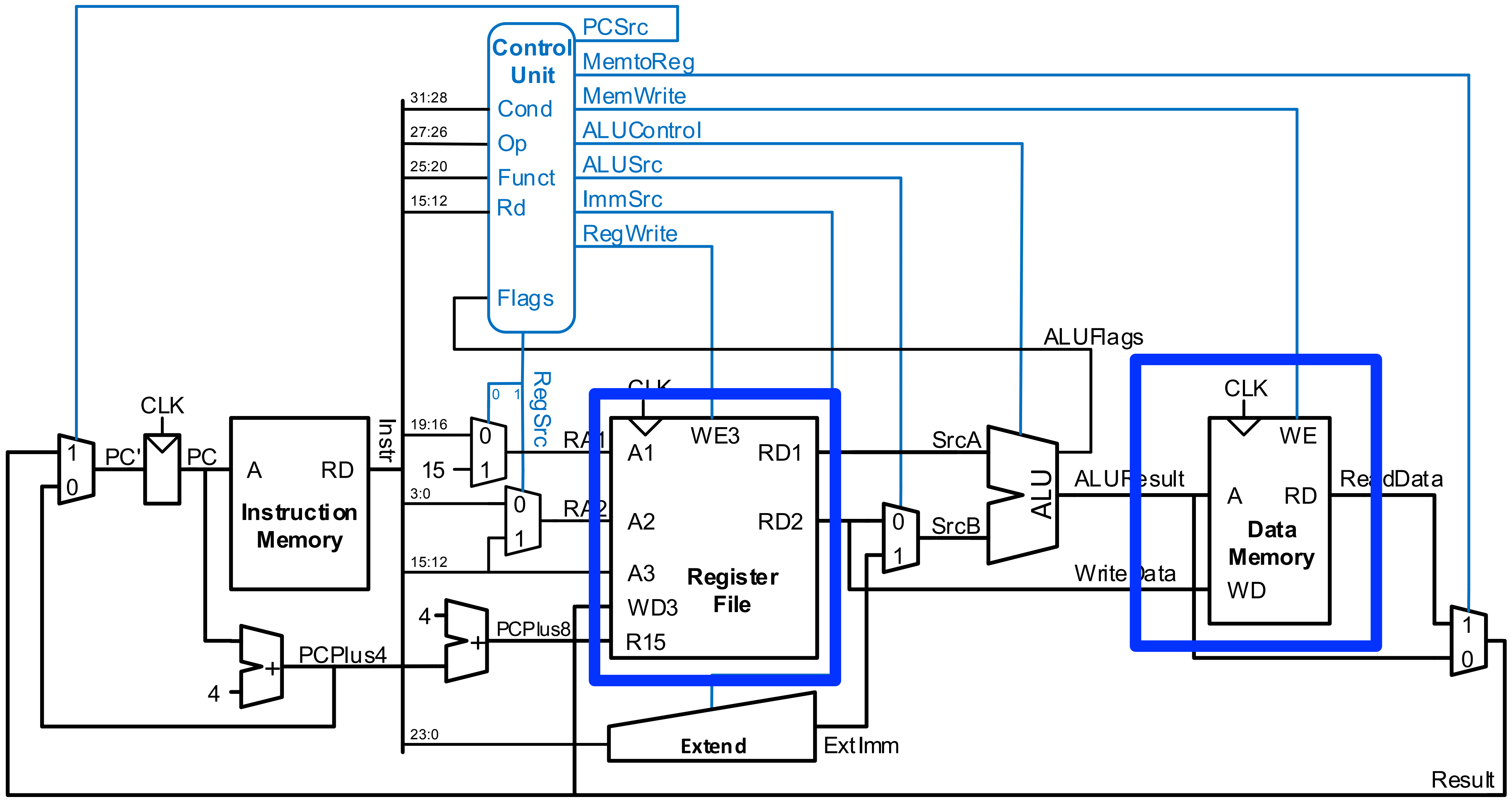
Operands

- An instruction operates on operands .

```
SUB a, b, c
```

  - The variables a, b, and c are all operands.
  - But computers operate on 1’ s and 0’ s, not variable names.
- The instructions need a **physical location** from which to **retrieve** the binary data.
- Operands can be stored in
  - Registers
  - Memory
  - Constants stored in the instruction itself **(immediates)**.

Registers and Memory



Operands: Registers

- Instructions need to access operands quickly so that they can run fast.
  - But operands stored in memory take **a long time** to retrieve.
- Therefore, most architectures specify a small number of **registers** that hold commonly used operands.
- ARM has 16 registers
  - Registers are **faster** than memory
  - Each register is **32 bits**
  - ARM is called a "**32-bit architecture**" because it operates on 32-bit data

ARM Register Set

- **Registers:**
  - R before number, all capitals
  - Example: “R0” or “register zero” or “register R0”

Name	Use
R0	Argument / return value / temporary variable
R1-R3	Argument / temporary variables
R4-R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Instructions with Registers

Revisit `ADD` instruction

C Code

```
a = b + c
```

ARM Assembly Code

```
; R0 = a, R1 = b, R2 = c
ADD R0, R1, R2
```

Operands: Constants\Immediates

- Many instructions can use constants or *immediate* operands
- For example: ADD and SUB
- Value is *immediate*ly available from instruction

C Code

```
a = a + 4;
b = a - 12;
```

ARM Assembly Code

```
; R0 = a, R1 = b
ADD R0, R0, #4
SUB R1, R0, #12
```

Generating Constants

Generating small constants using move (`MOV`):

C Code

```
//int: 32-bit signed word
int a = 23;
int b = 0x45;
```

ARM Assembly Code

```
; R0 = a, R1 = b
MOV R0, #23
MOV R1, #0x45
```

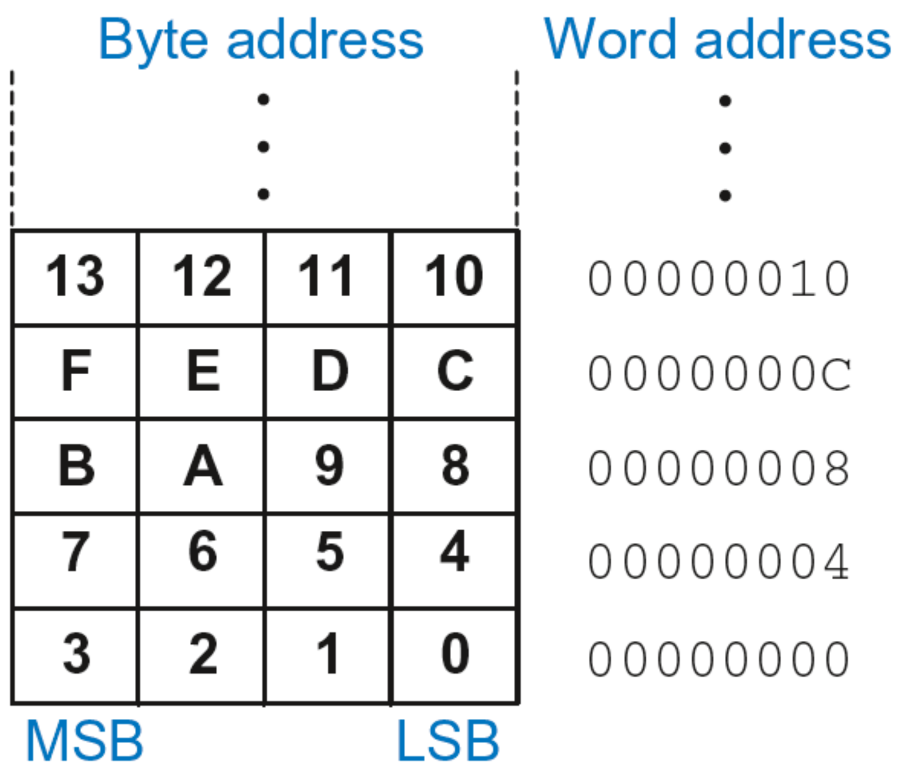
Note: MOV can also use 2 registers: `MOV R7,R9`

Operands: Memory

- If registers were the only storage space for operands
  - Simple programs with no more than 15 variables.
- However, data can also be stored in memory.
  - Whereas the register file is small and fast, memory is **larger and slower**.
  - For this reason, **frequently used variables** are kept in registers.
- In the ARM architecture, instructions operate **exclusively** on registers
  - so data stored in memory **must be moved** to a register before it can be processed.

Byte-Addressable Memory

- ARM uses a **byte-addressable** memory.
- Each data byte has **unique address**
  - 32-bit word = 4 bytes, so word address increments by 4



Reading Memory

- Memory read called **load**
  - **Mnemonic:** *load register* (LDR)
  - **Format:**

```
LDR R0, [R1, #12]
```

  - **Address calculation:**
    - add **base address** (R1) to the **offset** (12)
    - address = (R1 + 12)
  - **Result:**
    - R0 holds the data at memory address (R1 + 12)

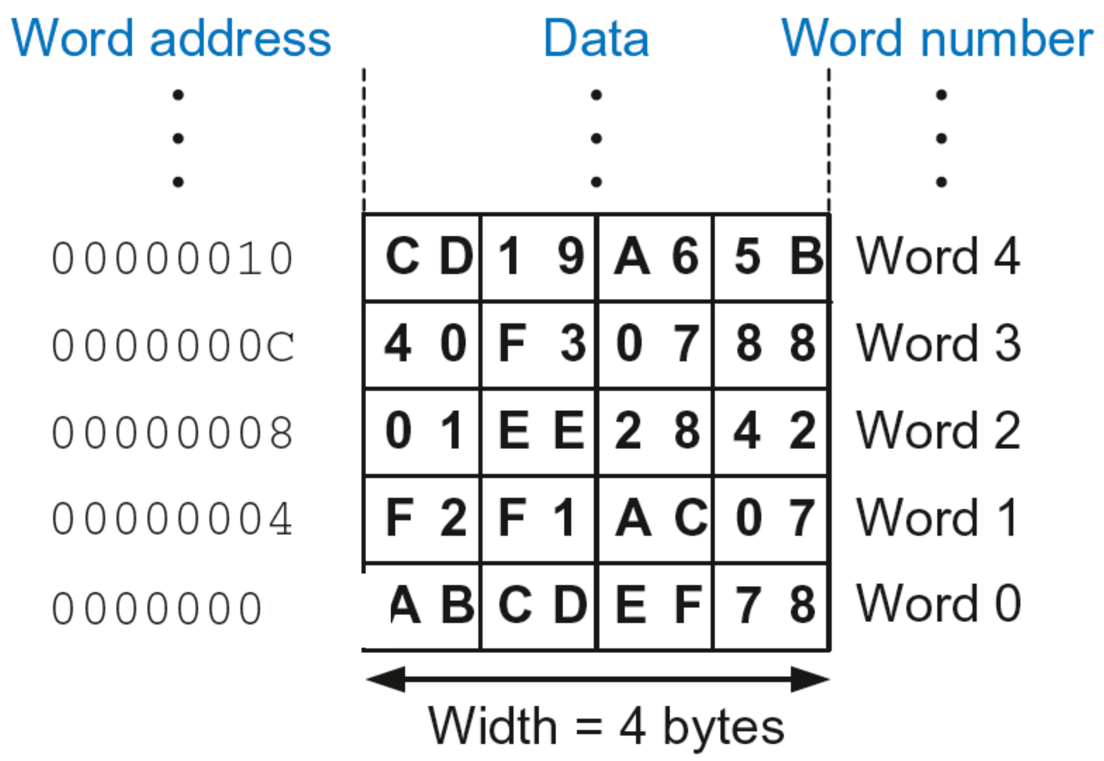
Any register may be used as base address.

Reading Memory

- **Example:** Read a word of data at memory address 8 into R3
  - Address = (R2 + 8) = 8
  - R3 = 0x01EE2842 after load

ARM Assembly Code

```
MOV R2, #0
LDR R3, [R2, #8]
```

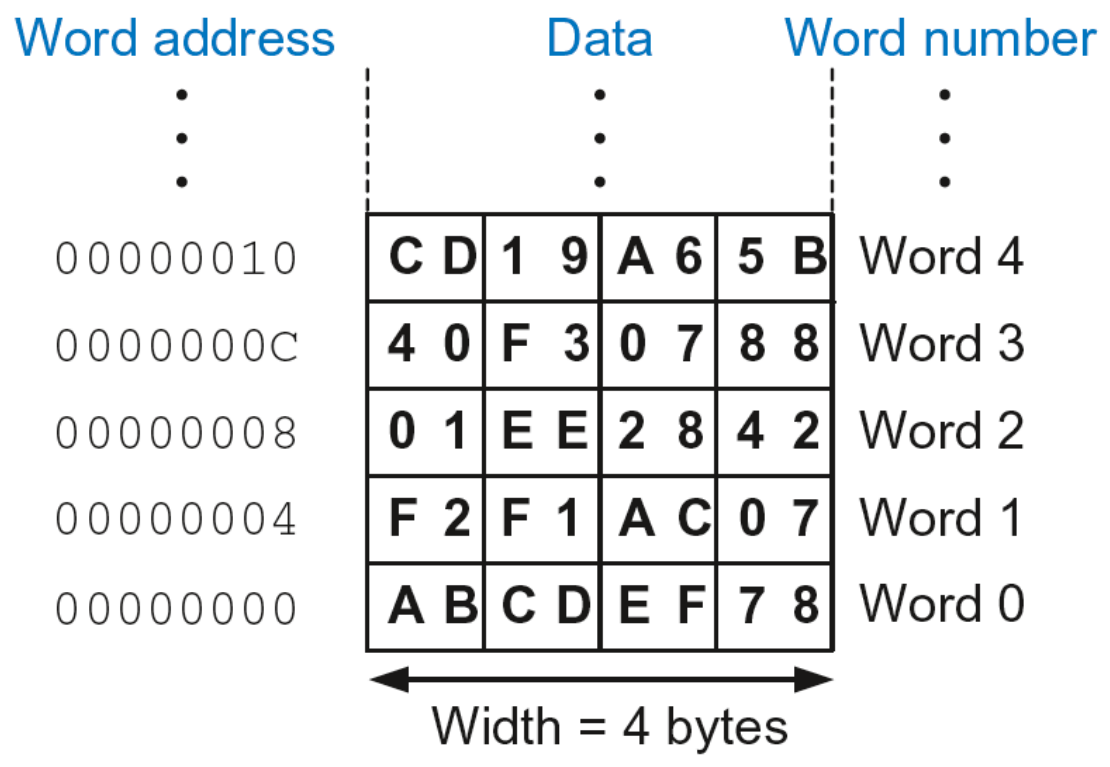


Writing Memory

- Memory write are called **stores**
  - **Mnemonic:** store register (`STR`)
- Example: Store the value held in R7 into memory word 21.
  - Memory address = 4 x 21 = 84 = 0x54

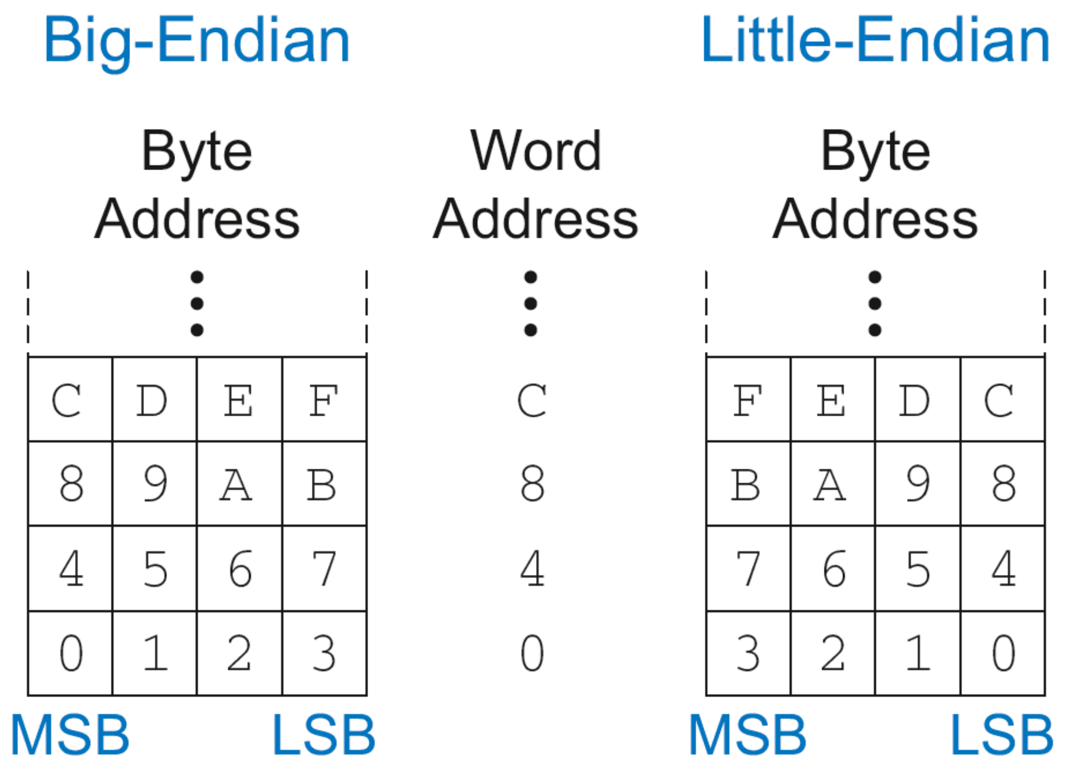
ARM assembly code

```
MOV R5, #0
STR R7, [R5, #0x54]
```



Recap: Accessing Memory

- How to number bytes within a word?
  - **Little-endian:** byte numbers start at the **little** (least significant) end
  - **Big-endian:** byte numbers start at the **big** (most significant) end



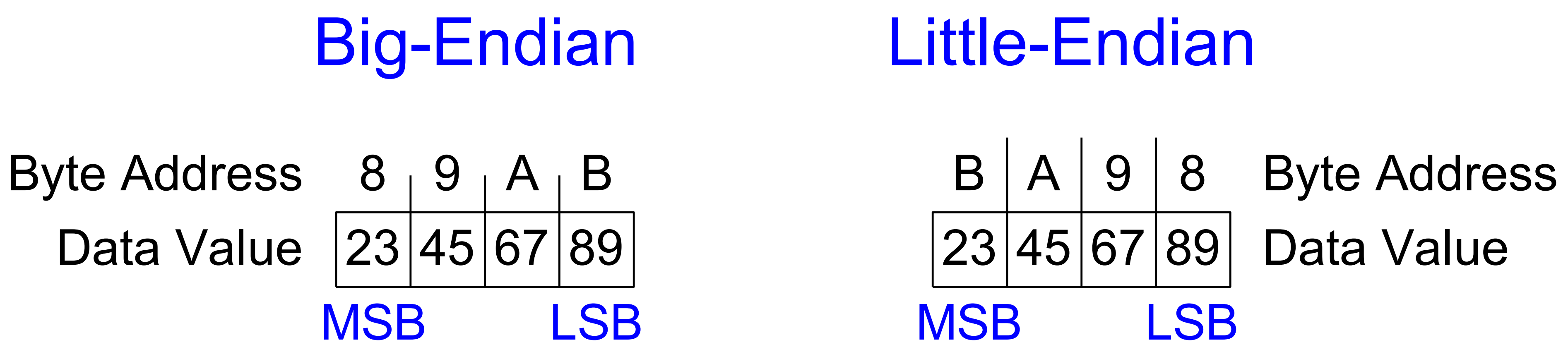
Big-Endian Byte Address				Word Address	Little-Endian Byte Address			
⋮				⋮	⋮			
C	D	E	F	C	F	E	D	C
8	9	A	B	8	B	A	9	8
4	5	6	7	4	7	6	5	4
0	1	2	3	0	3	2	1	0
MSB			LSB		MSB			LSB

Big-Endian & Little-Endian Example

Suppose R2 and R5 hold the values 8 and 0x23456789

- After following code runs on big-endian system, what value is in `R7`?
- In a little-endian system?

```
STR  R5, [R2, #0]
LDRB R7, [R2, #1]
```



Big-endian: 0x00000045

Little-endian: 0x00000067

Programming

- **High-level languages**
  - e.g., C, Java, Python:
  - Written at a more abstract level than assembly
- Many high-level languages use common software constructs
  - such as arithmetic and logical operations
  - conditional execution, if/else statements
  - for and while loops
  - array indexing
  - function calls.

Data-processing Instructions

- Logical operations
- Shifts / rotate
- Multiplication

Logical Instructions

- These each operate **bitwise** on **two sources** and write the result to a **destination register**.
  - The first source is always a register and the second source is either an immediate or another register.
- `AND`
- `ORR`
- `EOR` **(XOR)**
- `BIC` **(Bit Clear)**
- `MVN` **(MoVe and NOT)**

Logical Instructions: Examples

Source registers

R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code		Result			
`AND R3, R1, R2`	R3	0100 0110	1010 0001	0000 0000	0000 0000
`ORR R4, R1, R2`	R4	1111 1111	1111 1111	1111 0001	1011 0111
`EOR R5, R1, R2`	R5	1011 1001	0101 1110	1111 0001	1011 0111
`BIC R6, R1, R2`	R6	0000 0000	0000 0000	1111 0001	1011 0111
`MVN R7, R2`	R7	0000 0000	0000 0000	1111 1111	1111 1111

Logical Instructions: Uses

- `AND` or `BIC`: useful for **masking** bits

  Example: Masking all but the least significant byte of a value

  0xF234012F `AND` 0x000000FF = 0x0000002F

  0xF234012F `BIC` 0xFFFFFF00 = 0x0000002F

- `ORR`: useful for **combining** bit fields

  Example: Combine 0xF2340000 with 0x000012BC:

  0xF2340000 `ORR` 0x000012BC = 0xF23412BC

Shift Instructions

- `LSL`: logical shift left

Example: `LSL R0, R7, #5  ; R0=R7 << 5`

- `LSR`: logical shift right

Example: `LSR R3, R2, #31 ; R3=R2 >> 31`

- `ASR`: arithmetic shift right

Example: `ASR R9, R11, R4 ;` $R9=R11 >>> R4_{7:0}$

- `ROR`: rotate right

Example: `ROR R8, R1, #3  ; R8=R1 ROR 3`

Shift Instructions: Example 1

- **Immediate** shift amount (5-bit immediate)
- Shift amount: 0-31

Source register

R5	1111 1111	0001 1100	0001 0000	1110 0111

Assembly Code		Result			
`LSL R0, R5, #7`	R0	1000 1110	0000 1000	0111 0011	1000 0000
`LSR R1, R5, #17`	R1	0000 0000	0000 0000	0111 1111	1000 1110
`ASR R2, R5, #3`	R2	1111 1111	1110 0011	1000 0010	0001 1100
`ROR R3, R5, #21`	R3	1110 0000	1000 0111	0011 1111	1111 1000

Shift Instructions: Example 2

- **Register** shift amount (uses low 8 bits of register)
- Shift amount: 0-255

Source registers

R8	0000 1000	0001 1100	0001 0110	1110 0111
R6	0000 0000	0000 0000	0000 0000	0001 0100

Assembly code

```
LSL R4, R8, R6
ROR R5, R8, R6
```

Result

R4	0110 1110	0111 0000	0000 0000	0000 0000
R5	1100 0001	0110 1110	0111 0000	1000 0001

Multiplication

- `MUL`: 32 × 32 multiplication, 32-bit result

  ```
  MUL R1, R2, R3
  ```

  Result: $R1 = (R2 \times R3)_{31:0}$

- `UMULL`: Unsigned multiply long: 32 × 32 multiplication, 64-bit result

  ```
  UMULL R1, R2, R3, R4
  ```

  Result: `{R1,R4} = R2 x R3` (`R2`, `R3` unsigned)

- `SMULL`: Signed multiply long: 32 × 32 multiplication, 64-bit result

  ```
  SMULL R1, R2, R3, R4
  ```

  Result: `{R1,R4} = R2 x R3` (`R2`, `R3` signed)

Conditional Execution

Don’t always want to execute code sequentially

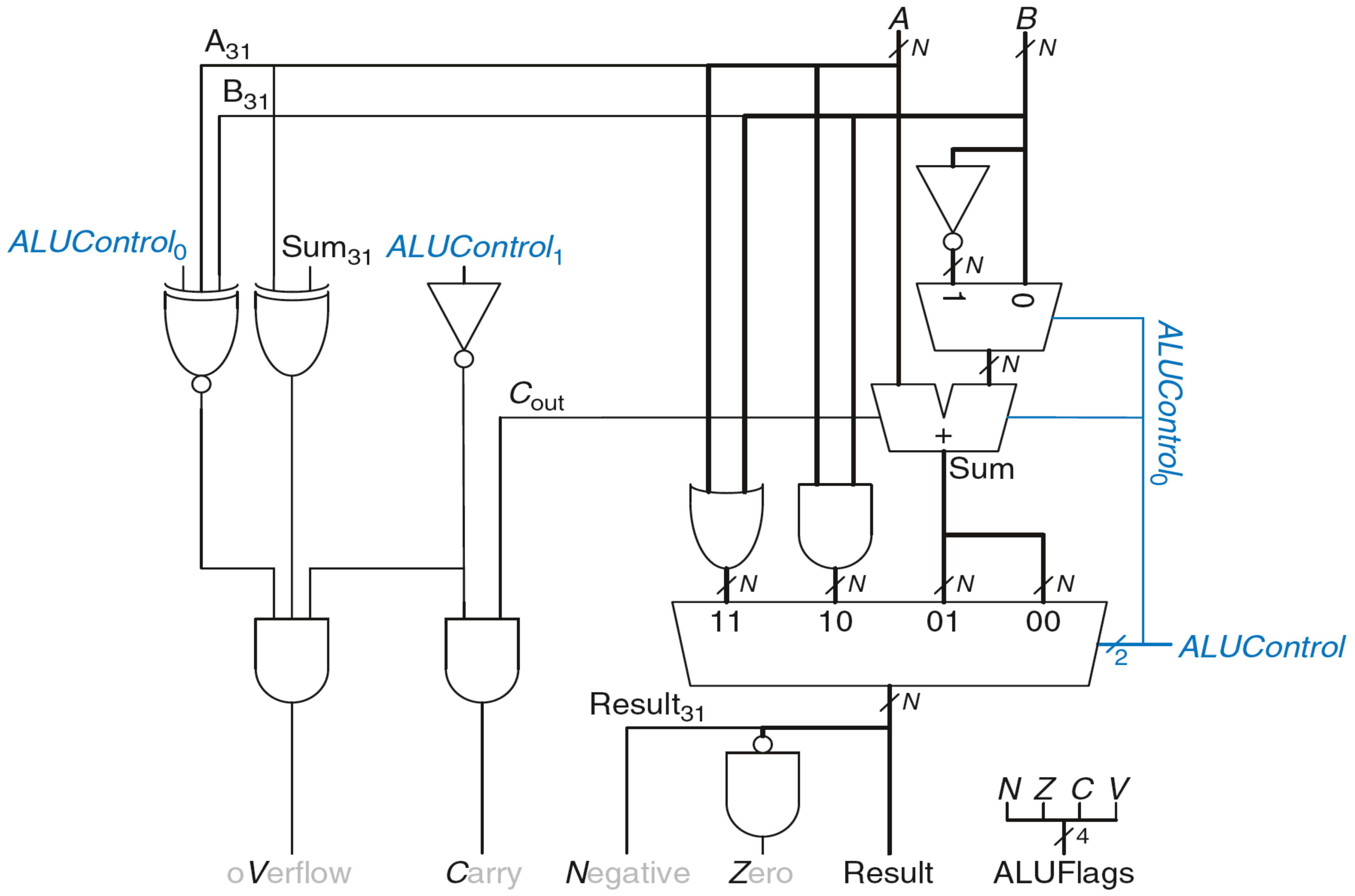
- For example:
  - if/else statements, while loops, etc.: only want to execute code *if* a **condition** is true
  - branching: jump to another portion of code *if* a condition is true
- ARM includes **condition flags** that can be:
  - set by an instruction
  - used to conditionally execute an instruction

ARM Condition Flags

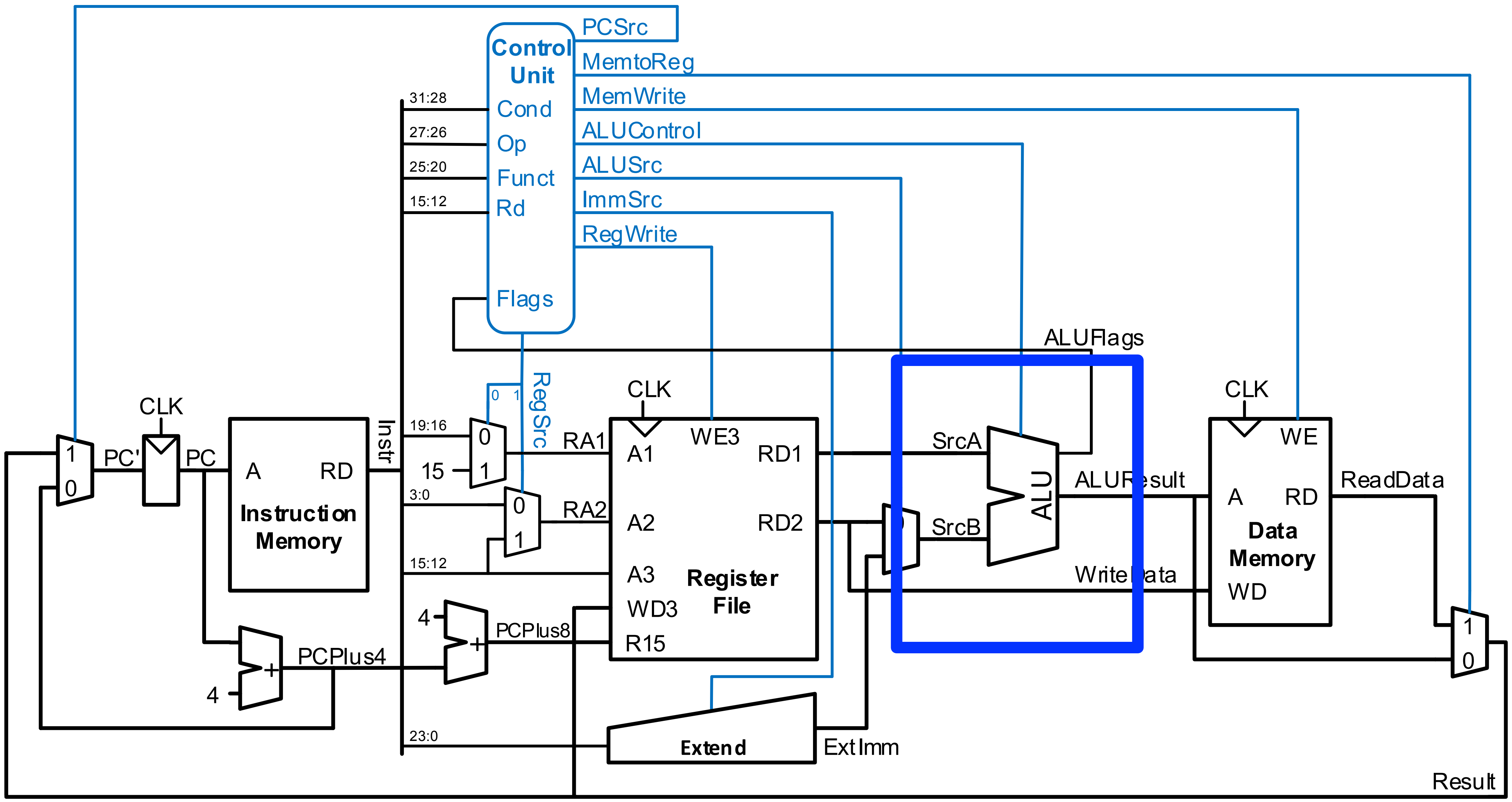
Flag	Name	Description
N	**N**egative	Instruction result is negative
Z	**Z**ero	Instruction results in zero
C	**C**arry	Instruction causes an unsigned carry out
V	o**V**erflow	Instruction causes an overflow

- Set by ALU
- Held in *Current Program Status Register* **(*CPSR*)**

Review: ARM ALU



Review: ALU



PCSrc
Control Unit
MemtoReg
MemWrite
ALUControl
ALUSrc
ImmSrc
RegWrite
31:28
27:26
25:20
15:12
Cond
Op
Funct
Rd
Flags
ALUFlags
RegSrc
CLK
CLK
CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
15
3:0
15:12
RA1
RA2
A1
A2
A3
WD3
R15
WE3
RD1
RD2
Register File
SrcA
SrcB
ALU
ALUResult
WriteData
A
RD
WE
WD
Data Memory
ReadData
PCPlus4
PCPlus8
4
4
Extend
ExtImm
23:0
Result

Setting the Condition Flags: *NZCV*

- Compare instruction: `CMP`

  Example: `CMP R5,R6`

  - Performs: R5 - R6
  - Does not save result
  - Sets flags. If result:

• Is 0,	*Z*=1
• Is negative,	*N*=1
• Causes a carry out,	*C*=1
• Causes a signed overflow,	*V*=1

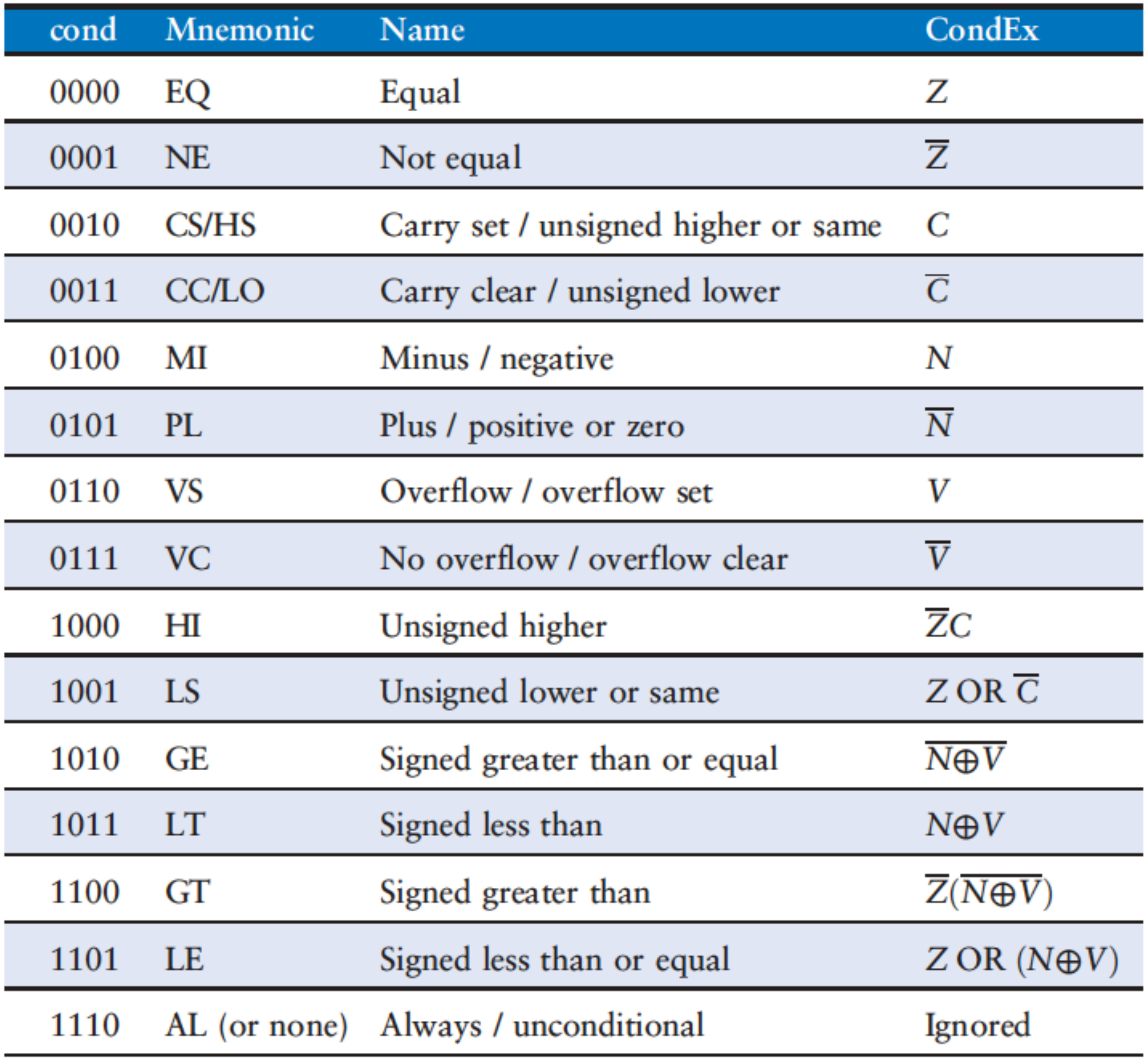
Condition Mnemonics

- Instruction may be **conditionally executed** based on the condition flags
- Condition of execution is encoded as a **condition mnemonic** appended to the instruction mnemonic

Example:
```
CMP    R1, R2
SUBNE R3, R5, R8
```

- **NE:** condition mnemonic
- `SUB` will only execute if R1 ≠ R2 (i.e., Z = 0)

Condition Mnemonics



cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\overline{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\overline{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\overline{N}$
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	$\overline{V}$
1000	HI	Unsigned higher	$\overline{Z}C$
1001	LS	Unsigned lower or same	Z OR $\overline{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	Z OR $(N \oplus V)$
1110	AL (or none)	Always / unconditional	Ignored

Conditional Execution

Example:

```
CMP   R5, R9      ; performs R5-R9
                  ; sets condition flags

SUBEQ R1, R2, R3    ; executes if R5==R9 (Z=1)
ORRMI R4, R0, R9    ; executes if R5-R9 is
                    ; negative (N=1)
```

Suppose R5 = 17, R9 = 23:

`CMP` performs: 17 – 23 = -6 (Sets flags: *N*=1, *Z*=0, *C*=0, *V*=0)

`SUBEQ` **doesn't execute** (they aren't equal: *Z*=0)

`ORRMI` **executes** because the result was negative (*N*=1)

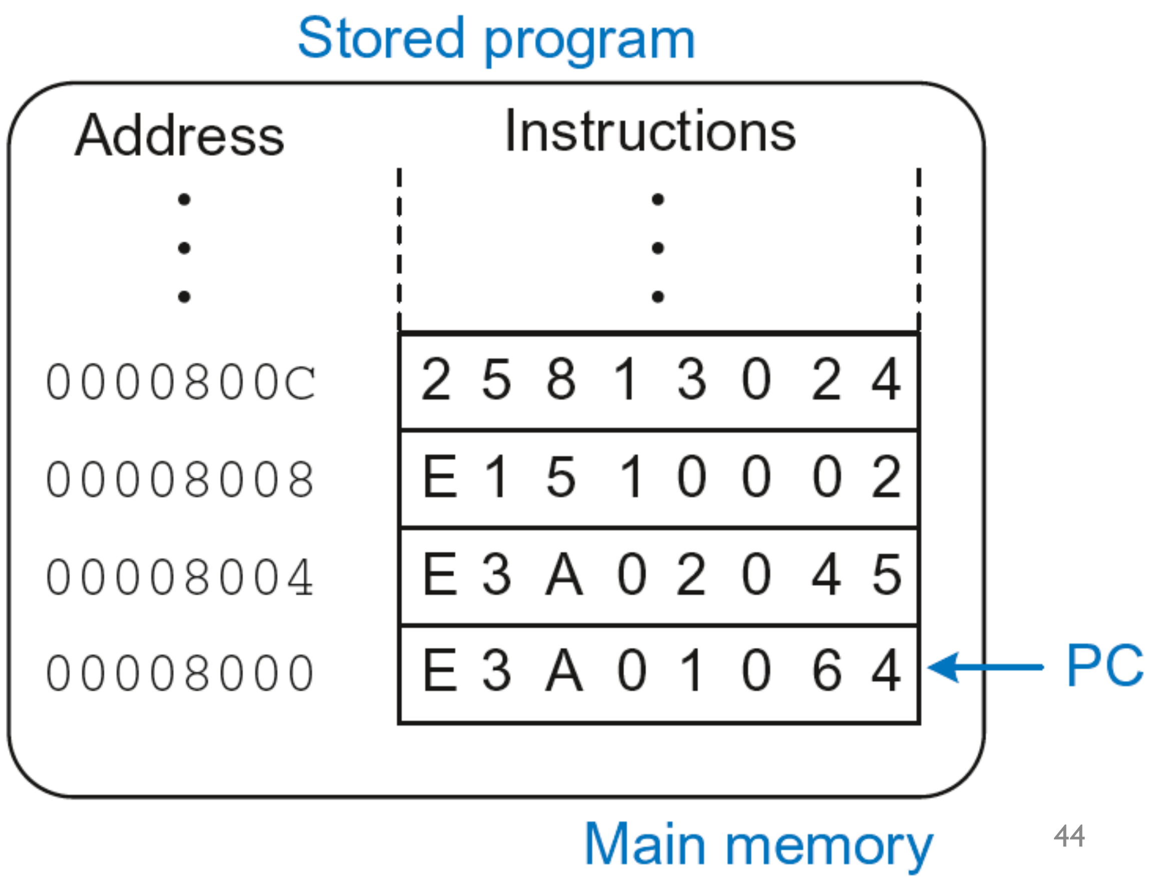
Branching

- Branches enable out of sequence instruction execution
  - ARM use branch instructions to skip over sections of code or repeat code.
- Types of branches:
  - **Branch (`B`)**
    - branches to another instruction
  - **Branch and link (`BL`)**
- Both can be conditional or unconditional

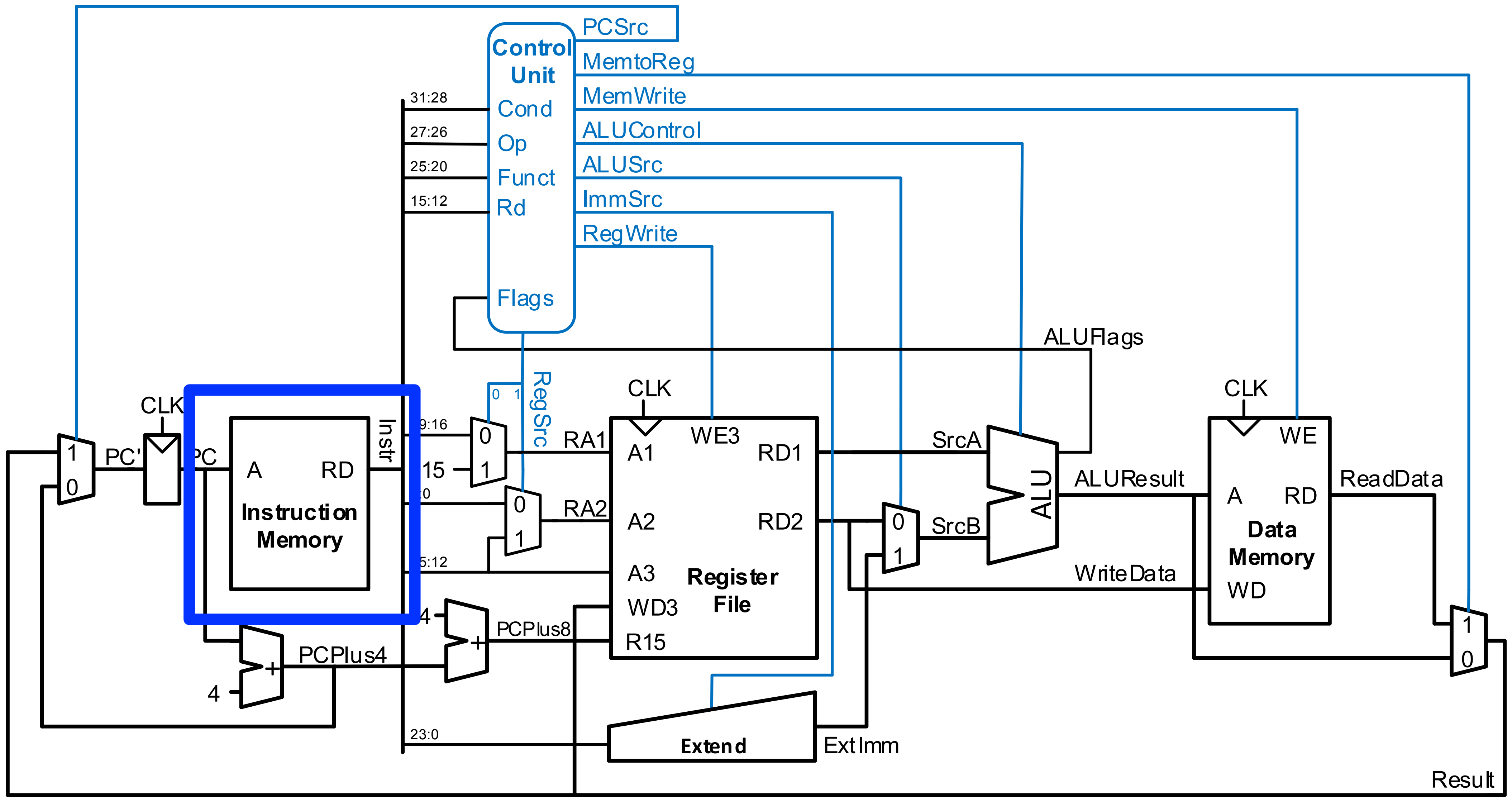
The Stored Program

- A program usually executes in sequence, with the **program counter (PC)** incrementing by 4 after each instruction to point to the next instruction.
  - Recall that instructions are 4 bytes long and ARM is a byte-addressed architecture.
- Branch instructions **change** the program counter.

Assembly code	Machine code
`MOV R1, #100`	`0xE3A01064`
`MOV R2, #69`	`0xE3A02045`
`CMP R1, R2`	`0xE1510002`
`STRHS R3, [R1, #0x24]`	`0x25813024`



Review: Stored Program



Unconditional Branching (B)

ARM assembly

```
   MOV R2, #17        ; R2 = 17
   B   TARGET         ; branch  to  target
   ORR R1, R1, #0x4   ; not executed

TARGET
   SUB R1, R1, #78    ; R1 = R1 + 78
```

Labels (like `TARGET`) indicate instruction location. Labels can't be reserved words (like `ADD`, `ORR`, etc.)

Conditional Branching

ARM Assembly

```
   MOV   R0, #4         ; R0 = 4
   ADD   R1, R0, R0     ; R1 = R0+R0 = 8
   CMP   R0, R1         ; sets flags with R0-R1
   BEQ   THERE          ; branch not taken (Z=0)
   ORR   R1, R1, #1     ; R1 = R1 OR R1 = 9
THERE
   ADD R1, R1, 78       ; R1 = R1 + 78 = 87
```

if Statement

C Code

```
if (i == j)
  f = g + h;

f = f - i;
```

ARM Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP R3, R4         ; set flags with R3-R4
 BNE L1             ; if i!=j, skip if block
 ADD R0, R1, R2     ; f = g + h

L1
 SUB R0, R0, R2     ; f = f - i
```

Assembly tests opposite case (**i != j**) of high-level code (**i == j**)

if/else Statement

C Code

```
if (i == j)
   f = g + h;


else
   f = f - i;
```

ARM Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP R3, R4       ; set flags with R3-R4
 BNE L1           ; if i!=j, skip if block
 ADD R0, R1, R2   ; f = g + h
 B   L2           ; branch past else block
L1
 SUB R0, R0, R2   ; f = f - i
L2
```

while Loops

High-Level Code

```
int pow = 1;
int x = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

ARM Assembly Code

```
; R0 = pow, R1 = x
  MOV R0, #1        ; pow = 1
  MOV R1, #0        ; x = 0

WHILE
  CMP R0, #128      ; pow != 128 ?
  BEQ DONE          ; if pow == 128, exit loop
  LSL R0, R0, #1    ; pow = pow * 2
  ADD R1, R1, #1    ; x = x + 1
  B   WHILE         ; repeat loop
DONE
```

For Loops

High-Level Code

```
int i;
int sum = 0;



for (i = 0; i < 10; i = i + 1) {
   sum = sum + i;
}
```

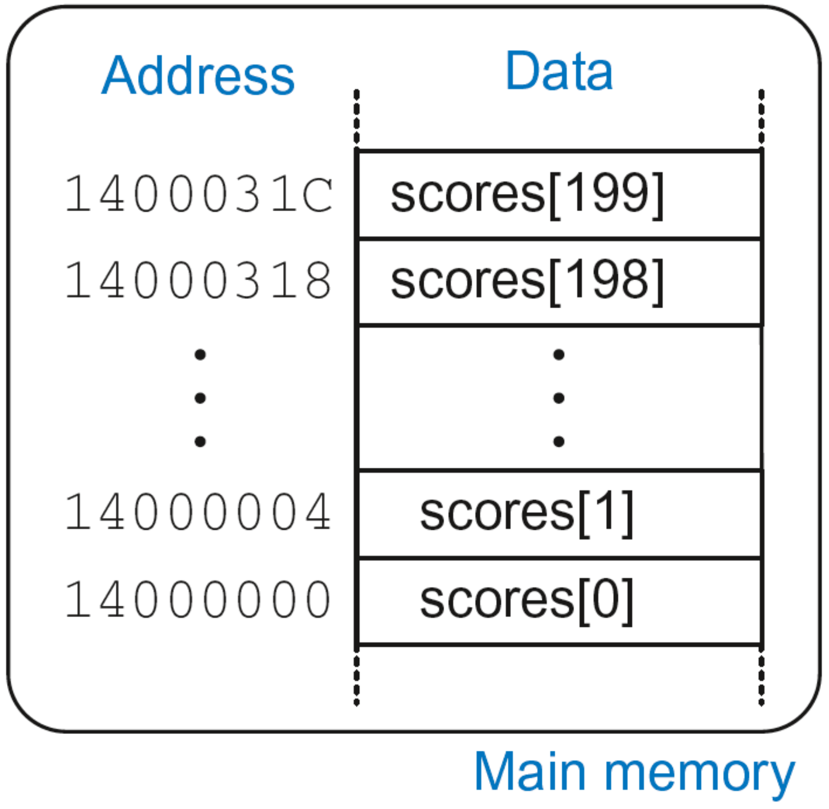
ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV R1, #0          ; sum = 0
  MOV R0, #0          ; i = 0           loop initialization

FOR
  CMP R0, #10         ; i < 10 ?        check condition
  BGE DONE            ; if (i >= 10) exit loop
  ADD R1, R1, R0      ; sum = sum + i   loop body
  ADD R0, R0, #1      ; i = i + 1       loop operation
  B   FOR             ; repeat loop
DONE
```

Arrays

- Access large amounts of similar data
  - **Index:** access to each element
  - **Size:** number of elements
- 5-element array
  - **Base address** = 0x14000000 (address of first element, scores[0])
  - Array elements accessed **relative to** base address



Accessing Arrays

C Code

```
int array[5];
array[0] = array[0] * 8;
array[1] = array[1] * 8;
```

ARM Assembly Code

```
; R0 = array base address
MOV R0, #0x60000000    ; R0 = 0x60000000

LDR R1, [R0]           ; R1 = array[0]
LSL R1, R1, 3          ; R1 = R1 << 3 = R1*8
STR R1, [R0]           ; array[0] = R1

LDR R1, [R0, #4]       ; R1 = array[1]
LSL R1, R1, 3          ; R1 = R1 << 3 = R1*8
STR R1, [R0, #4]       ; array[1] = R1
```

ACCESSING ARRAYS USING A FOR LOOP

High-Level Code

```
int i;
int scores[200];
...



for (i = 0; i < 200; i = i + 1)


  scores[i] = scores[i] + 10;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
; initialization code          ...
  MOV R0, #0x14000000          ; R0 = base address
  MOV R1, #0                   ; i = 0

LOOP
  CMP R1, #200                 ; i < 200?
  BGE L3                       ; if i ≥ 200, exit loop
  LSL R2, R1, #2               ; R2 = i * 4
  LDR R3, [R0, R2]             ; R3 = scores[i]
  ADD R3, R3, #10              ; R3 = scores[i] + 10
  STR R3, [R0, R2]             ; scores[i] = scores[i] + 10
  ADD R1, R1, #1               ; i = i + 1
  B   LOOP                     ; repeat loop
L3
```

ACCESSING ARRAYS USING A FOR LOOP

High-Level Code

```
int i;
int scores[200];
...


for (i = 0; i < 200; i = i + 1)
        scores[i] = scores[i] + 10;
```

ARM Assembly Code

```
; R0 = array base address
; initialization code          ...
  MOV R0, #0x14000000          ; R0 = base address
  ADD R1, R0, #800             ; R1 = base address + (200*4)

LOOP
  CMP R0, R1                   ; reached end of array?
  BGE L3                       ; if yes, exit loop
  LDR R2, [R0]                 ; R2 = scores[i]
  ADD R2, R2, #10              ; R2 = scores[i] + 10
  STR R2, [R0], #4             ; scores[i] = scores[i] + 10
                               ; then R0 = R0 + 4
  B   LOOP                     ; repeat loop
 L3
```

Function Calls

Caller:

– passes arguments to callee
– jumps to callee

Callee:

– performs the function
– returns result to caller
– returns to point of call
– must not overwrite registers or memory needed by caller

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

ARM Function Conventions

- **Call Function:** branch and link

  `BL`

  - it stores the return address of the next instruction in the **link register (LR)**, and it branches to the target instruction.
- **Return** from function: move the link register to PC:

  `MOV PC, LR`
- **Arguments:** R0-R3
- **Return value:** R0

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

ARM Assembly Code

```
0x00000200 MAIN      BL  SIMPLE
0x00000204           ADD R4, R5, R6
...

0x00401020 SIMPLE    MOV PC, LR
```

`BL` `(branch and link)` branches to `SIMPLE`
`LR` = `PC` + 4 = 0x00000204

`MOV PC, LR` makes `PC` = `LR`
(the next instruction executed is at 0x00000200)

Input Arguments and Return Value

ARM conventions:

- Argument values: `R0` - `R3`
- Return value: `R0`

Input Arguments and Return Value

C Code

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;               // return value
}
```

Input Arguments and Return Value

ARM Assembly Code

```
; R4 = y
MAIN
  ...
  MOV R0, #2          ; argument 0 = 2
  MOV R1, #3          ; argument 1 = 3
  MOV R2, #4          ; argument 2 = 4
  MOV R3, #5          ; argument 3 = 5
  BL DIFFOFSUMS       ; call function
  MOV R4, R0          ; y = returned value
  ...
; R4 = result
DIFFOFSUMS
  ADD R8, R0, R1      ; R8 = f + g
  ADD R9, R2, R3      ; R9 = h + i
  SUB R4, R8, R9      ; result = (f + g) - (h + i)
  MOV R0, R4          ; put return value in R0
  MOV PC, LR          ; return to caller
```

Further Reading

- You can read **Chapter 6** of your book
  - **Till Section 6.4**