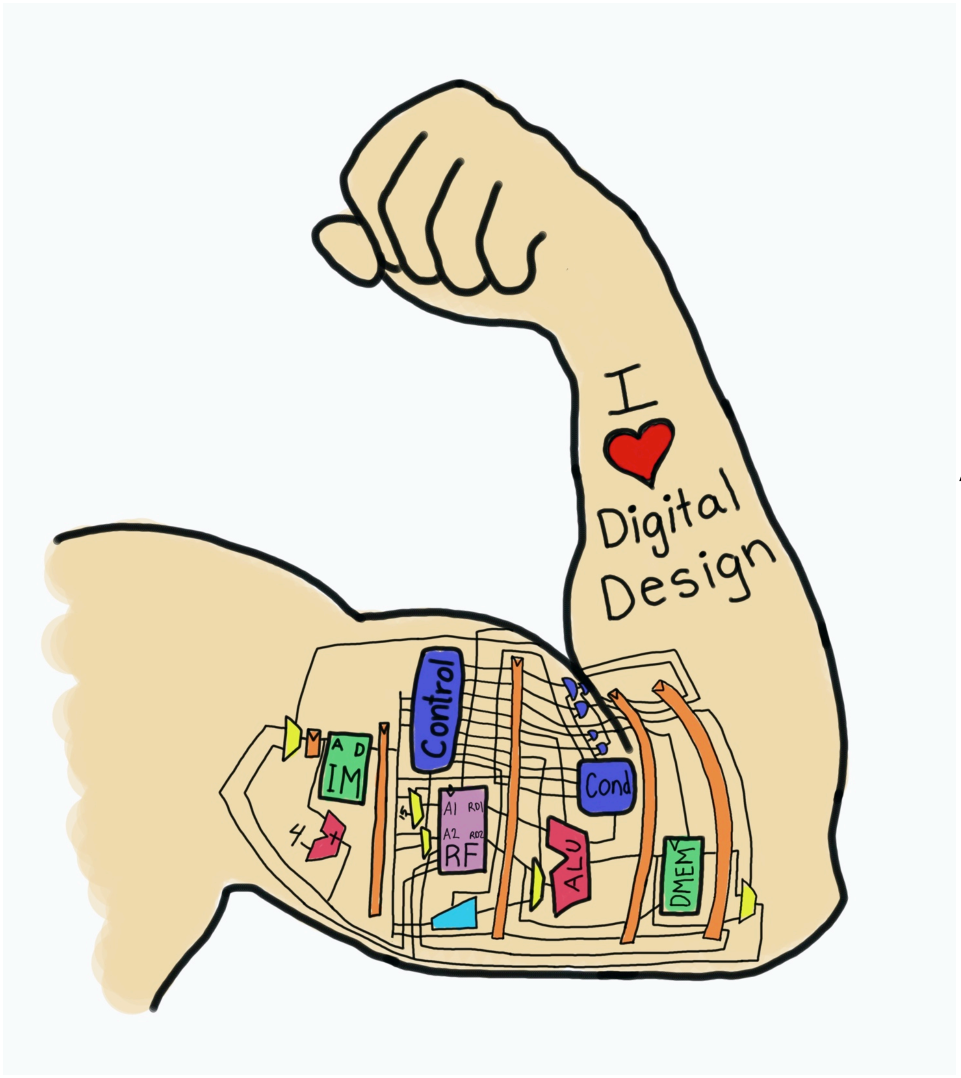
E85 Digital Design & Computer Engineering



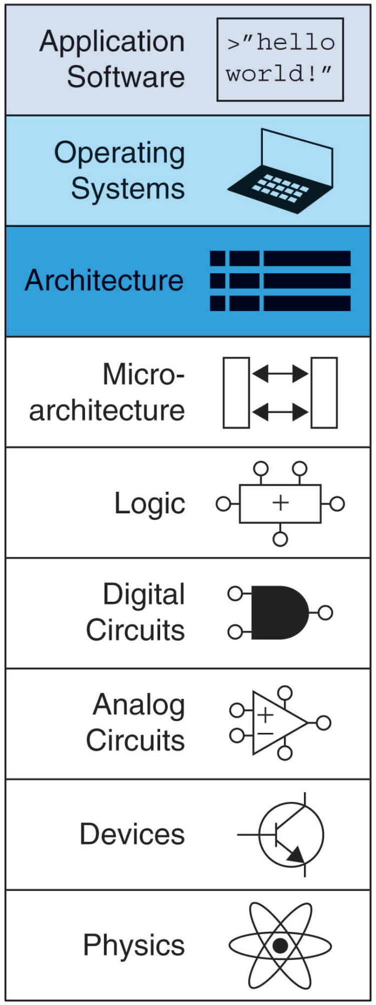
Lecture 17:

ARM Assembly Language

HARVEY MUDD COLLEGE

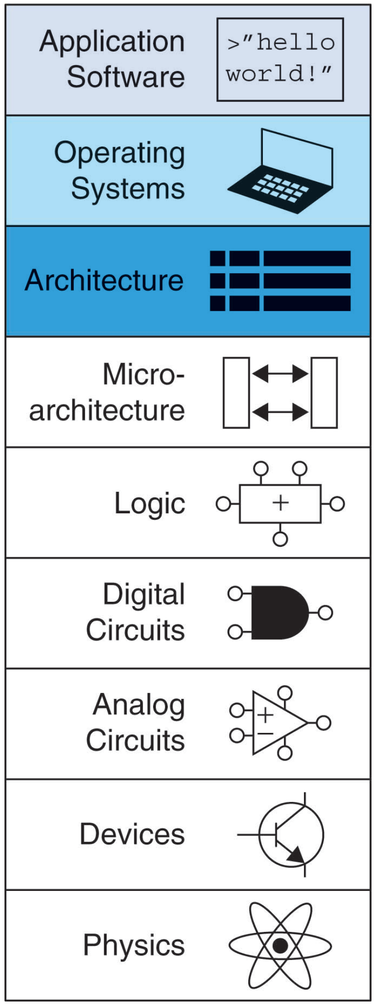
Lecture 17

- **Introduction**
- **Assembly Language**
- **Machine Language**
- **Programming**
- **Addressing Modes**



Introduction

- **Jumping up a few levels of abstraction**
  - **Architecture:** programmer's view of computer
    - Defined by **instructions** & **operand locations**
  - **Microarchitecture:** how to implement an architecture in hardware (covered in Chapter 7)



Instructions

- **Commands in a computer's language**
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)

ARM Architecture

- Developed in the 1980's by Advanced RISC Machines – now called ARM Holdings
- Nearly 10 billion ARM processors sold/year
- 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.

ARM Architecture

- Developed in the 1980's by Advanced RISC Machines – now called ARM Holdings
- Nearly 10 billion ARM processors sold/year
- 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.

Once you've learned one architecture, it's easier to learn others

Architecture Design Principles

Underlying design principles, as articulated by Hennessy and Patterson:

1. **Regularity supports design simplicity**
2. **Make the common case fast**
3. **Smaller is faster**
4. **Good design demands good compromises**

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

```
a = b - c;
```

ARM assembly code

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

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

Design Principle 1

Regularity supports design simplicity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Ease of encoding and handling in hardware

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
```

Design Principle 2

Make the common case fast

- ARM includes only simple, commonly used instructions
- Hardware to decode and execute instructions kept simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions

Design Principle 2

Make the common case fast

- ARM is a **Reduced Instruction Set Computer (RISC)**, with a small number of simple instructions
- Other architectures, such as Intel's x86, are **Complex Instruction Set Computers (CISC)**

Operand Location

Physical location in computer

- Registers
- Constants (also called *immediates*)
- Memory

Operands: Registers

- 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

Design Principle 3

Smaller is Faster

- ARM includes only a small number of registers

ARM Register Set

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

Operands: Registers

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

Operands: Registers

- **Registers used for specific purposes:**
  - **Saved registers:** R4-R11 hold variables
  - **Temporary registers:** R0-R3 and R12, hold intermediate values
  - Discuss others later

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
```

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
```

Constant must have < 8 bits of precision

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

Generating Constants

Generate larger constants using move (`MOV`) and or (`ORR`):

C Code

```
int a = 0x7EDC8765;
```

ARM Assembly Code

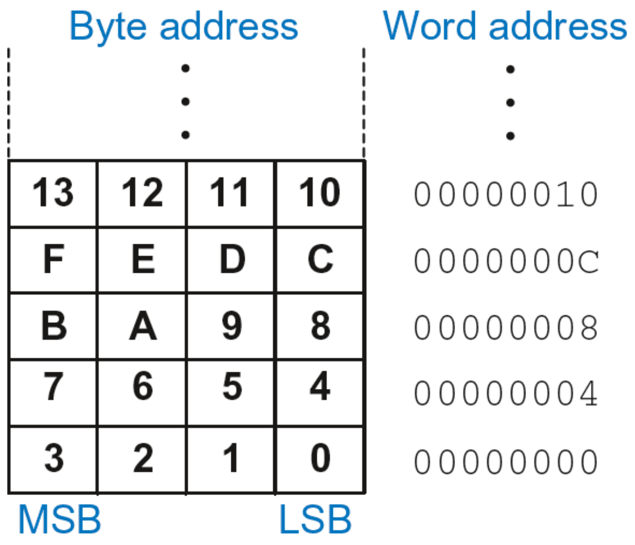
```
# R0 = a
MOV R0, #0x7E000000
ORR R0, R0, #0xDC0000
ORR R0, R0, #0x8700
ORR R0, R0, #0x65
```

Operands: Memory

- Too much data to fit in only 16 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables still kept in registers

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)

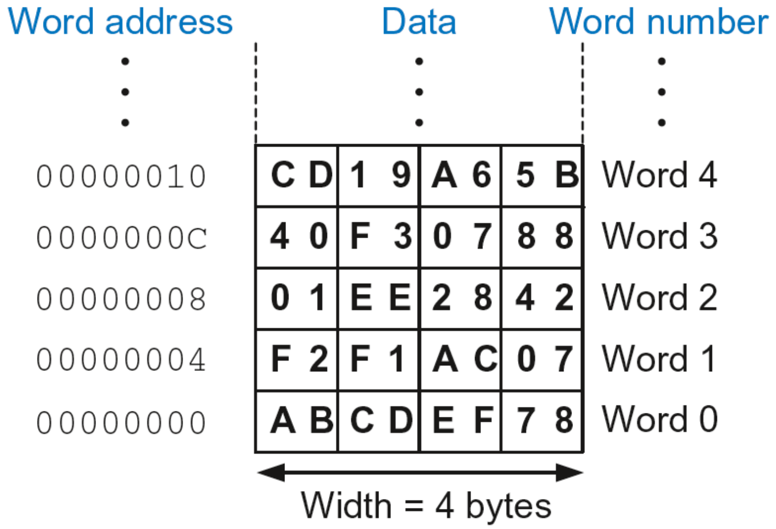
Any register may be used as base address

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`)

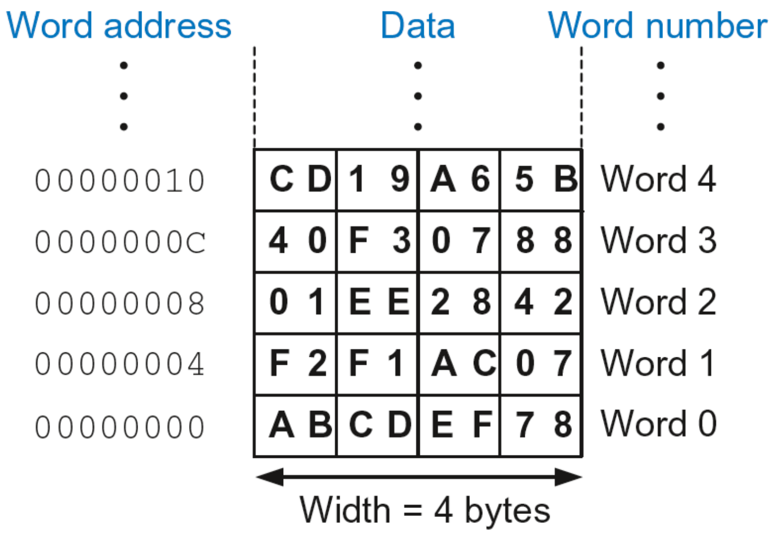
Writing Memory

- **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]
```

The offset can be written in decimal or hexadecimal

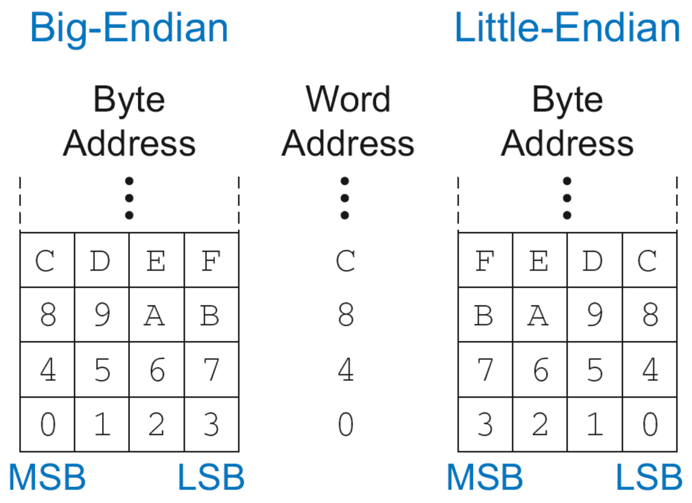


Recap: Accessing Memory

- Address of a memory **word** must be multiplied by 4
- **Examples:**
  – Address of memory word 2 = 2 $\times$ 4 = 8
  – Address of memory word 10 = 10 $\times$ 4 = 40

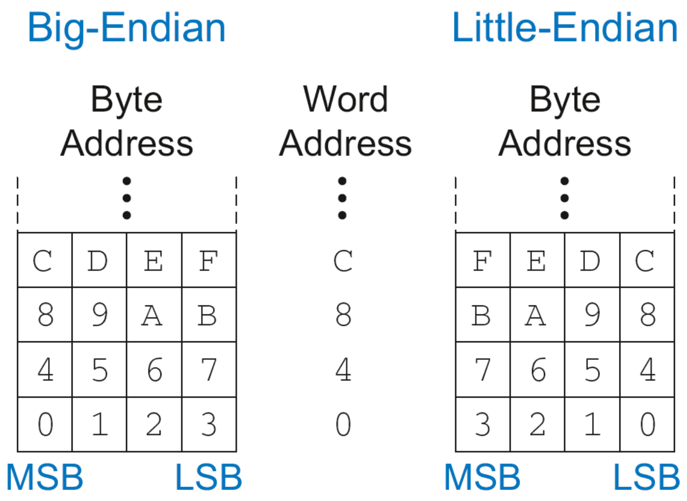
Big-Endian & Little-Endian Memory

- **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 & Little-Endian Memory

- **Jonathan Swift's *Gulliver's Travels*:** the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- **It doesn't really matter** which addressing type used – **except** when two systems **share data**



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 & Little-Endian Example

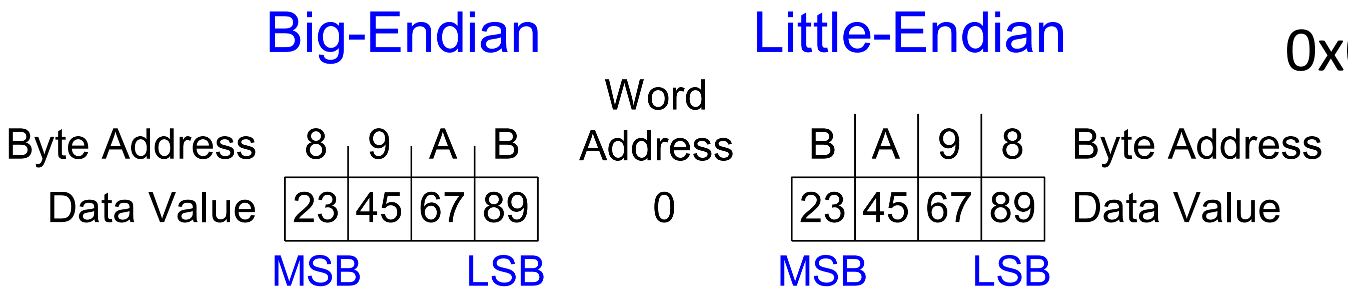
Suppose R2 and R5 hold the values 8 and 0x23456789

- 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 higher level of abstraction

Ada Lovelace, 1815-1852

- British mathematician
- Wrote the first computer program
- Her program calculated the Bernoulli numbers on Charles Babbage's Analytical Engine
- She was a child of the poet Lord Byron

Programming Building Blocks

- **Data-processing Instructions**
- **Conditional Execution**
- **Branches**
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Data-processing Instructions

- Logical operations
- Shifts / rotate
- Multiplication

Logical Instructions

- `AND`
- `ORR`
- `EOR` **(XOR)**
- `BIC` **(Bit Clear = A & ~B)**
- `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		Result			
`LSL R4, R8, R6`	R4	0110 1110	0111 0000	0000 0000	0000 0000
`ROR R5, R8, R6`	R5	1100 0001	0110 1110	0111 0000	1000 0001

Multiplication

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

  `MUL R1, R2, R3`

  Result: `R1 = (R2 x R3)`$_{31:0}$ (signed doesn't matter)

- **`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)

Programming Building Blocks

- Data-processing Instructions
- **Conditional Execution**
- Branches
- High-level Constructs:
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

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

Conditional Execution

Don't always want to execute code sequentially

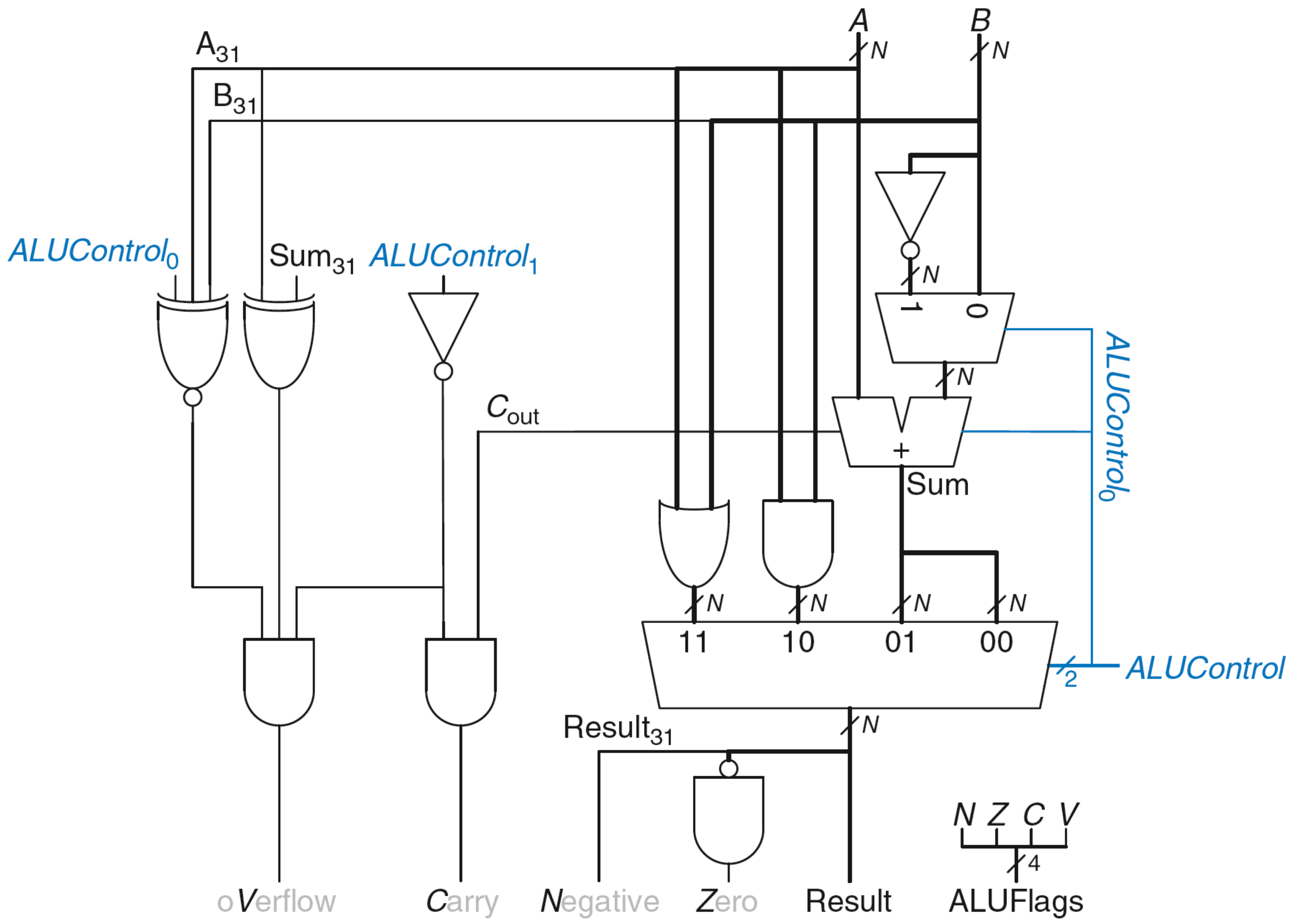
- For example:
  - if/else statements, while loops, etc.: only want to execute code *if* a condition is true
  - branching: jump to another portion of code *if* a condition is true
- ARM includes **condition flags** that can be:
  - set by an instruction
  - used to conditionally execute an instruction

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 (recall from Chapter 5)
- Held in *Current Program Status Register* (*CPSR*)

Review: ARM ALU



Setting the Condition Flags: *NZCV*

- **Method 1:** 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

Setting the Condition Flags: *NZCV*

- **Method 1:** Compare instruction: `CMP`

  Example: `CMP R5, R6`
  - Performs: R5-R6
  - Sets flags: If result is 0 (Z=1), negative (N=1), etc.
  - Does not save result
- **Method 2:** Append instruction mnemonic with **S**

  Example: `ADDS R1, R2, R3` (with the **S** in bold)
  - Performs: R2 + R3
  - Sets flags: If result is 0 (Z=1), negative (N=1), etc.
  - Saves result in R1

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**: not equal 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)

Programming Building Blocks

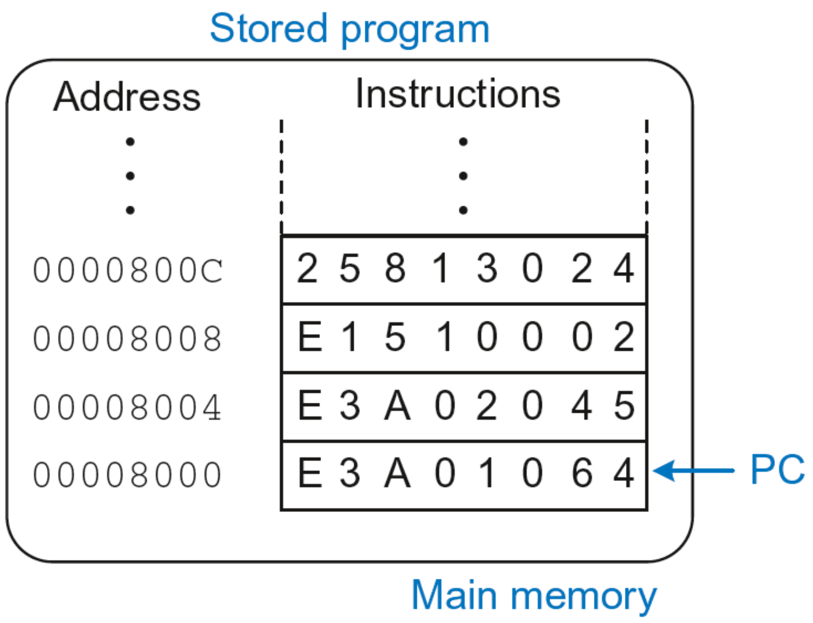
- Data-processing Instructions
- Conditional Execution
- Branches
- High-level Constructs:
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Branching

- Branches enable out of sequence instruction execution
- Types of branches:
  - **Branch (`B`)**
    - branches to another instruction
  - **Branch and link (`BL`)**
    - discussed later
- Both can be conditional or unconditional

The Stored Program

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



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.)

The Branch Not Taken

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
```

Programming Building Blocks

- Data-processing Instructions
- Conditional Execution
- Branches
- High-level Constructs:
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

if Statement

C Code

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



f = f - i;
```

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
```

if Statement

C Code | **ARM Assembly Code**

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

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

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

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

if Statement: Alternate Code

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
 ADDEQ R0, R1, R2   ; if (i==j) f = g + h
 SUB   R0, R0, R2   ; f = f - i
```

if Statement: Alternate Code

Original

```
 CMP R3, R4
 BNE L1
 ADD R0, R1, R2
L1
 SUB R0, R0, R2
```

Alternate Assembly Code

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

 CMP   R3, R4       ; set flags with R3-R4
 ADDEQ R0, R1, R2   ; if (i==j) f = g + h
 SUB   R0, R0, R2   ; f = f - i
```

if Statement: Alternate Code

Original

```
 CMP R3, R4
 BNE L1
 ADD R0, R1, R2
L1
 SUB R0, R0, R2
```

Alternate Assembly Code

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

 CMP   R3, R4       ; set flags with R3-R4
 ADDEQ R0, R1, R2   ; if (i==j) f = g + h
 SUB   R0, R0, R2   ; f = f - i
```

Useful for **short** conditional blocks of code

if/else Statement

C Code

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


else
  f = f - i;
```

ARM Assembly Code

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
```

if/else Statement: Alternate Code

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
 ADDEQ R0, R1, R2  ; if (i==j) f = g + h

 SUBNE R0, R0, R2  ; else f = f - i
```

if/else Statement: Alternate Code

Original

```
 CMP R3, R4
 BNE L1
 ADD R0, R1, R2
 B   L2
L1
 SUB R0, R0, R2
L2
```

Alternate Assembly Code

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

 CMP   R3, R4      ; set flags with R3-R4
 ADDEQ R0, R1, R2  ; if (i==j) f = g + h

 SUBNE R0, R0, R2  ; else f = f - i
```

while Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;


while (pow != 128) {

  pow = pow * 2;
  x = x + 1;
}
```

ARM Assembly Code

while Loops

C Code

```
// determines the power
// of x such that 2^x = 128
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              ; R0-128
  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
```

while Loops

C Code

```
// determines the power
// of x such that 2^x = 128
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             ; R0-128
  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
```

Assembly tests for the opposite case (`pow == 128`) of the C code (`pow != 128`).

for Loops

```
for (initialization; condition; loop operation)
    statement
```

- `initialization:` executes before the loop begins
- `condition:` is tested at the beginning of each iteration
- `loop operation:` executes at the end of each iteration
- `statement:` executes each time the condition is met

for Loops

C Code

```
// adds numbers from 1-9
int sum = 0;
int i;


for (i=1; i!=10; i=i+1)
   sum = sum + i;
```

ARM Assembly Code

for Loops

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=1; i!=10; i=i+1)
   sum = sum + i;
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV   R0, #1          ; i = 1
  MOV   R1, #0          ; sum = 0

FOR
  CMP R0, #10           ; R0-10
  BEQ DONE              ; if (i==10)
                        ; exit loop
  ADD R1, R1, R0        ; sum=sum + i
  ADD R0, R0, #1        ; i = i + 1
  B   FOR               ; repeat loop

DONE
```

for Loops: Decremented Loops

In ARM, decremented loop variables are more efficient

for Loops: Decremented Loops

In ARM, decremented loop variables are more efficient

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=9; i!=0; i=i-1)
   sum = sum + i;
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV   R0, #9             ; i = 9
  MOV   R1, #0             ; sum = 0

FOR
  ADD   R1, R1, R0         ; sum=sum + i
  SUBS  R0, R0, #1         ; i = i - 1
                           ; and set flags
  BNE   FOR                ; if (i!=0)
                           ; repeat loop
```

for Loops: Decremented Loops

In ARM, decremented loop variables are more efficient

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=9; i!=0; i=i-1)
   sum = sum + i;
```

ARM Assembly Code

```
; R0 = i, R1 = sum
    MOV   R0, #9          ; i = 9
    MOV   R1, #0          ; sum = 0

FOR
    ADD   R1, R1, R0      ; sum=sum + i
    SUBS  R0, R0, #1      ; i = i - 1
                          ; and set flags
    BNE   FOR             ; if (i!=0)
                          ; repeat loop
```

Saves 2 instructions per iteration:

- Decrement loop variable & compare: `SUBS R0, R0, #1`
- Only 1 branch – instead of 2

Programming Building Blocks

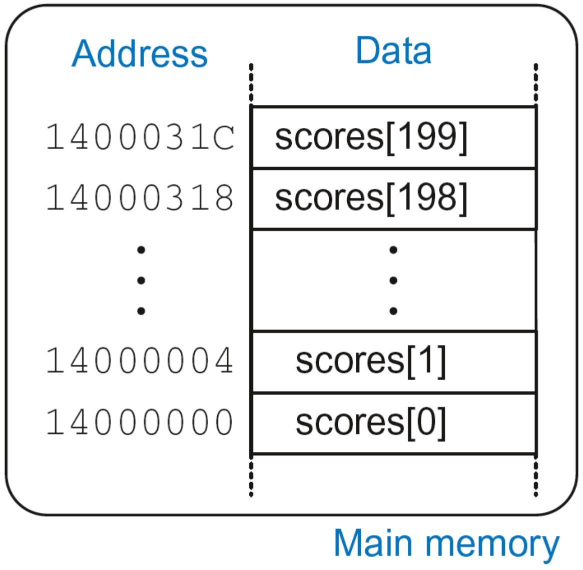
- Data-processing Instructions
- Conditional Execution
- Branches
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - **arrays**
  - function calls

Arrays

- Access large amounts of similar data
  - **Index:** access to each element
  - **Size:** number of elements

Arrays

- 200-element array
  - **Base address** = 0x14000000 (address of first element, scores[0])
  - Array elements accessed relative to base address



Accessing Arrays

C Code

```
int array[200];
array[0] = array[0] * 8;
array[1] = array[1] * 8;
```

ARM Assembly Code

```
; R0 = array 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
```

Arrays using for Loops

C Code

```
int array[200];
int i;
for (i=199; i >= 0; i = i - 1)
     array[i] = array[i] * 8;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
```

Arrays using for Loops

C Code

```
int array[200];
int i;

for (i=199; i >= 0; i = i - 1)
     array[i] = array[i] * 8;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
  MOV R0, 0x60000000
  MOV R1, #199

FOR
  LDR  R2, [R0, R1, LSL #2]      ; R2 = array(i)
  LSL  R2, R2, #3                ; R2 = R2<<3 = R3*8
  STR  R2, [R0, R1, LSL #2]      ; array(i) = R2
  SUBS R1, R1, #1                ; i = i - 1
                                 ; and set flags
  BPL  FOR                       ; if (i>=0) repeat loop
```

ASCII Code

- American Standard Code for Information Interchange
- Each text character has unique byte value
  - For example, S = 0x53, a = 0x61, A = 0x41
  - Lower-case and upper-case differ by 0x20 (32)

Cast of Characters

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	‘	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	–	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		