Chapter 2

Instructions: Language of the Computer

Instruction Set

- The collection of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

The ARM Instruction Set

- Used as the example in chapters 2 and 3
- Most popular 32-bit instruction set in the world (www.arm.com)
- 4 Billion shipped in 2008
- Large share of embedded core market
  - Applications include mobile phones, consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern RISC ISAs
  - See ARM Assembler instructions, their encoding and instruction cycle timings in appendixes B1,B2 and B3

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
ADD a, b, c  ; a gets b + c
```

- All arithmetic operations have this form

Simplicity Favours Regularity

- Hardware for a variable number of operands is more complicated than for a fixed number
- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

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

- Compiled ARM code:

```
ADD t0, g, h    ; temp t0 = g + h
ADD t1, i, j    ; temp t1 = i + j
SUB f, t0, t1   ; f = t0 - t1
```

Operands in Real Hardware

- Operands in instructions are **registers** in hardware. i.e. Register operands
- Unlike in an high level languages (variables) the number of operands in instructions (registers) are limited

Register Operands

- ARM has a 16 × 32-bit register file
  - Use for frequently accessed data
  - Registers numbered 0 to 15 (r0 to r15)
  - 32-bit data called a “word”

Smaller is Faster

- Very large number of registers increase the clock cycle time
  - Because electronic signals has to travel farther
- *Design Principle 2:* Smaller is faster
  - Designer should balance the craving of programs for more registers with the desire for fast clock cycle

Register Operand Example

- C code:

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

  - *f, g, h, I, j in registers r0, r1, r2, r3, r4*
  - *r5 and r6 are temporary registers*

- Compiled ARM code:

```
ADD r5,r1,r2 ;register r5 contains g + h
ADD r6,r3,r4 ;register r6 contains i + j
SUB r0,r5,r6 ;r0 (f in register r0) gets r5-r6
```

Lets go to practical assembly programming

Cross Compiler

- **gcc** you used so far targeted Intel x86_64 architecture : can’t compile ARM assembly

- Use a **cross compiler**
  - Compiler runs on your Intel machine. But compiles for ARM.
  - On Linux *sudo apt-get install gcc-arm-linux-gnueabi*

Emulator

- At the moment we do not have a computer with an ARM processor.

- We use a emulator called **qemu** that runs on your Intel machine
  - On Linux *sudo apt-get install qemu-user*

Assembly Program

- Save with .s extension
- Comments starts with @ (even // would do)
- Things starting with '.' are called assembler directives (eg : .text , .global)
  - Assembler directives directs the assembler
  - Assembler : program that convers instructions to machine code
- One instruction per one line

Assembling and Running

- Assemble
  - arm-linux-gnueabi-gcc -Wall example.s -o example
- Run
  - qemu-arm -L /usr/arm-linux-gnueabi example

Exercise

- Assemble and run the given hello world example

- Complete ex1.s to do the calculation
  - f=a+b-c-d+e
  - a,b,c,d,e in r0,r1,r2,r3,r4 respectively
  - Put f to r5

Back to Theory …

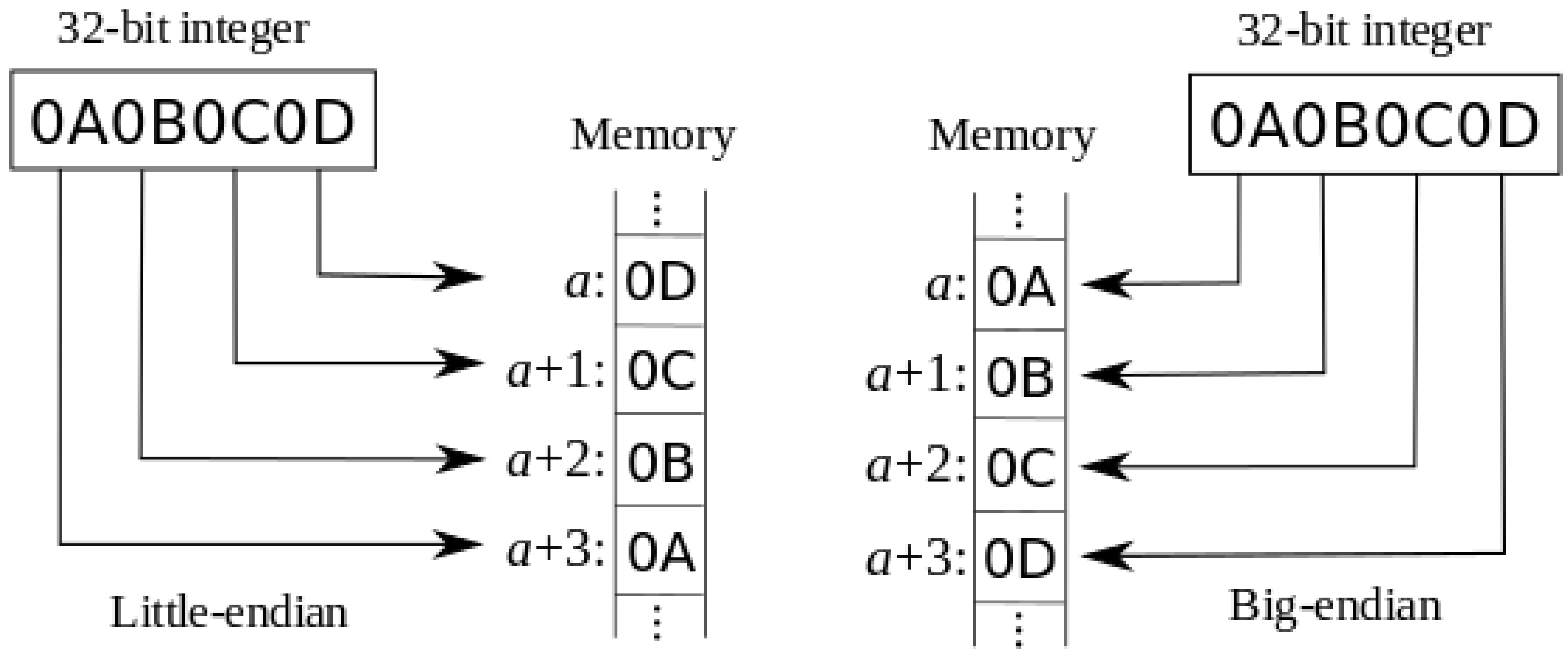
Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
  - Registers not adequate
  - Hence stored in memory
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory

Memory Operands

- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory (called **alignment restriction**)
  - Address must be a multiple of 4
- ARM is Little Endian
  - Least-significant byte at least address
  - *c.f.* Big Endian: Most-significant byte at least address of a word

Little Endian vs Big Endian



LDR and STR Instructions

- LDR : Load word into register
- STR : Store word from register

Memory Operand Example 1

- C code:

```
g = h + A[8];
```

  - *g in r1, h in r2, base address of A in r3*
  - *r5 is temporary register*
- Compiled ARM code:
  - Index 8 requires offset of 32 (4 bytes per word)

```
LDR  r5,[r3,#32] ; reg r5 gets A[8]
ADD r1, r2, r5 ; g = h + A[8]
```

base register

offset

Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

  - *h in r2, base address of A in r3*
  - *r5 is temporary register*
- Compiled ARM code:
  - Index 8 requires offset of 32

```
LDR  r5,[r3,#32] ; reg r5 gets A[8]
ADD  r5, r2, r5  ; reg r5 gets h+A[8]
STR  r5,[r3,#48] ; Stores h+A[8]into A[12]
```

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only **spill** to memory for less frequently used variables
  - Register optimization is important!

Exercise

- Complete ex2.s to do the following
  - a[2] = a[0] + a[1] - b
  - base address of a in r0
  - b in r1

Immediate Operands

- Constant data specified in an instruction

```
ADD r3,r3,#4 ; r3 = r3 + 4
```

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Exercise

- Change your solution in ex2 to do the following computation
- a[2] = 2 + a[0] + a[1] - 7

Exercise

- Complete the code in ex3.s to do the following computation
  - b[4] = 6 + a[9] - a[3] + b[1] – (c + d - e)
  - base address of a in r0
  - base address of b in r1
  - c,d,e in r2,r3,r4 respectively