Assembly: instruction formats; load/store

COSC 208, Introduction to Computer Systems, 2021-10-06

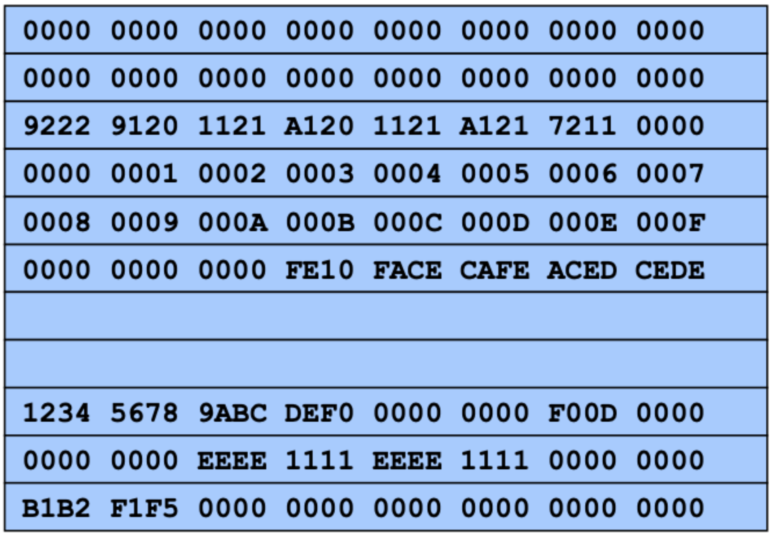
Announcements

- Exam1 Q5

Outline

- Assembly
- Operands
- Load/store

Language forms



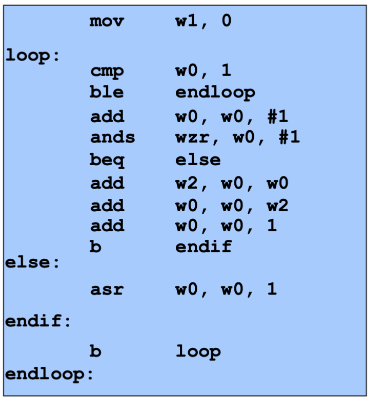
```
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
9222 9120 1121 A120 1121 A121 7211 0000
0000 0001 0002 0003 0004 0005 0006 0007
0008 0009 000A 000B 000C 000D 000E 000F
0000 0000 0000 FE10 FACE CAFE ACED CEDE


1234 5678 9ABC DEF0 0000 0000 F00D 0000
0000 0000 EEEE 1111 EEEE 1111 0000 0000
B1B2 F1F5 0000 0000 0000 0000 0000 0000
```

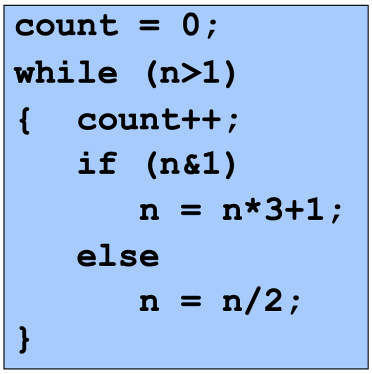
Machine language

- Not portable
  - Specific to hardware
- Simple
  - Each instruction does a simple task – poor ratio of functionality to code size
- Not human readable
  - Not structured
  - Requires lots of effort!
  - Requires tool support

We need assembly languages!



```
        mov     w1, 0
loop:
        cmp     w0, 1
        ble     endloop
        add     w0, w0, #1
        ands    wzr, w0, #1
        beq     else
        add     w2, w0, w0
        add     w0, w0, w2
        add     w0, w0, 1
        b       endif
else:
        asr     w0, w0, 1
endif:
        b       loop
endloop:
```



```
count = 0;
while (n>1)
{   count++;
    if (n&1)
        n = n*3+1;
    else
        n = n/2;
}
```

High-level language: C code

- Portable
- To varying degrees
- Complex
- One statement can do much work – good ratio of functionality to code size
- Human readable
- Structured – `if()`, `for()`, `while()`, etc.

Assembly Languages

- Not portable
  - Each assembly lang instruction maps to one machine lang instruction
- Simple
  - Each instruction does a simple task
- Human readable (In the same sense that Polish is human readable, if you know Polish.)

Why learn enough assembly?

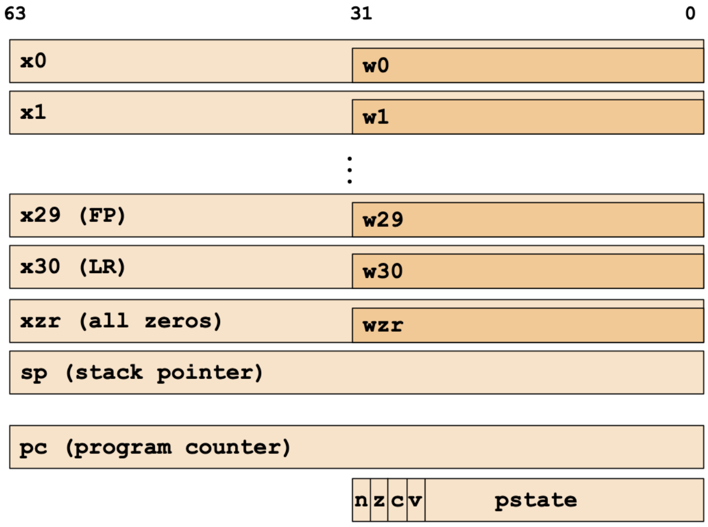
Knowing assembly language helps you:

- Write faster code
  - In assembly language
  - In a high-level language!
- Write safer code
  - Understanding mechanism of potential security problems helps you avoid them – even in high-level languages
- Understand what's happening *under the hood*
  - Someone needs to develop future computer systems
  - Intuition; some part for OS are written in assembly
- Become more comfortable with levels of abstraction
  - Become a better programmer!

Assembly

- Instruction set architectures (ISAs) --- defines the low-level instructions a central processing unit (CPU), or processor, can execute
    - Most popular Complex Instruction Set Computer (CISC) architecture: x86 (includes IA32 and x86-64)
    - Most popular Reduced Instruction Set Computer (RISC) architecture: ARM
    - Other common RISC architectures: SPARC (Scalable Processor Architecture), MIPS (Microprocessor without Interlocked Pipelined Stages), PowerPC, ARC (Argonaut RISC Core)

Registers



- Typical pattern:
    - Load data from RAM to registers
    - Manipulate data in registers
    - Store data from registers to RAM

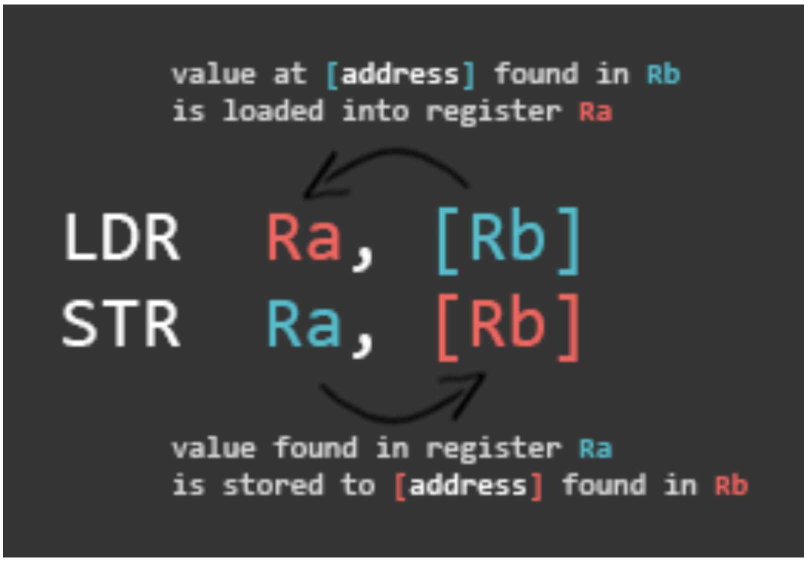
On RISC, this pattern is enforced

- Manipulation instructions can *only* access registers
- Known as a **Load/store architecture**
- CISC does more at once, compact instructions, but slower. Tradeoff

LDR is used to load something from memory into a register, and STR is used to store something from a register to a memory address.

- LDR operation: loads the value at the address found in R0 to the destination register R2.

- STR operation: stores the value found in R2 to the memory address found in R1.



value at [address] found in Rb
is loaded into register Ra

```
LDR  Ra, [Rb]
STR  Ra, [Rb]
```

value found in register Ra
is stored to [address] found in Rb

Example

- From high-level to low-level: mapping
    - C code

```
1  #include <stdio.h>
2  int deref(int *p) {
3      int v = *p;
4      return v;
5  }
6  int main() {
7      int x = 2;
8      int *y = &x;
9      int z = deref(y);
10     printf("deref(y) = %d\n", z);
11     return 0;
12 }
```

    - Viewing assembly code
        - Compile: `clang -o deref deref.c`
        - Dissasseamble executable: `objdump -d deref > deref_dump.txt`
    - Assembly code

```
0000000000400584 <deref>:
    400584: d10043ff    sub sp, sp, #0x10
    400588: f90007e0    str x0, [sp, #8]
    40058c: f94007e8    ldr x8, [sp, #8]
    400590: b9400109    ldr w9, [x8]
    400594: b90007e9    str w9, [sp, #4]
    400598: b94007e0    ldr w0, [sp, #4]
    40059c: 910043ff    add sp, sp, #0x10
    4005a0: d65f03c0    ret
```

        - *What do each of the columns contain?*
            - Code memory address
            - Bytes corresponding to instruction
            - Operation
            - Operands
    - Mapping between assembly and C code

```
0000000000400584 <deref>:
    400584: d10043ff    sub sp, sp, #0x10    // Line 2
    400588: f90007e0    str x0, [sp, #8]     //   V
    40058c: f94007e8    ldr x8, [sp, #8]     // Line 3
    400590: b9400109    ldr w9, [x8]         //   |
    400594: b90007e9    str w9, [sp, #4]     //   V
    400598: b94007e0    ldr w0, [sp, #4]     // Line 4
    40059c: 910043ff    add sp, sp, #0x10    //   |
    4005a0: d65f03c0    ret                  //   V
```

Operands

- Registers
  - General purpose: `w0` through `w30` (32-bit) and `x0` through `x30` (64-bit)
  - Stack pointer (top of current stack frame): `sp`
- Constant -- e.g., `#0x20`
- Memory
  - Dereference --- e.g., `[x1]`
  - Add to memory address, then dereference --- e.g., `[sp,#16]`

Load/store

- *What is the C code equivalent for* `str x0, [x1]`*, treating registers as if they were variable names?* — *x1 = x0
- *What is the C code equivalent for* `ldr x2, [x3]`*, treating registers as if they were variable names?* — x2 = *x3
- Q1: *Write the C code equivalent for each line of assembly, treating registers as if they were variable names. For example, the C code equivalent for* `sub sp, sp, #0x20` is `sp = sp - 0x20`

```
0000000000400584 <deref>:
    400584: d10043ff    sub sp, sp, #0x10   // sp = sp - 0x10
    400588: f90007e0    str x0, [sp, #8]    // *(sp + 8) = x0
    40058c: f94007e8    ldr x8, [sp, #8]    // x8 = *(sp + 8)
    400590: b9400109    ldr w9, [x8]        // w9 = *(x8)
    400594: b90007e9    str w9, [sp, #4]    // *(sp + 4) = w9
    400598: b94007e0    ldr w0, [sp, #4]    // w0 = *(sp + 4)
    40059c: 910043ff    add sp, sp, #0x10   // sp = sp + 0x10
    4005a0: d65f03c0    ret
```