Assembly: load/store operations; arithmetic and bitwise operations; translating assembly code to low-level C code

COSC 208, Introduction to Computer Systems, 2022-03-22

Announcements

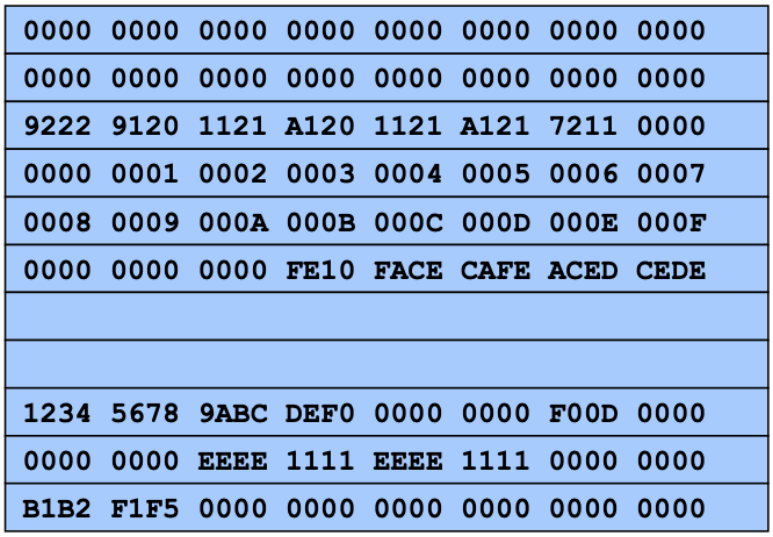
- Project 2 due Thursday, Mar 31

Outline

- Warm-up
- Assembly
- Load/store operations
- Arithmetic and bitwise operations
- Translating assembly code to low-level C code

Before break

Language forms



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 Architecture Types

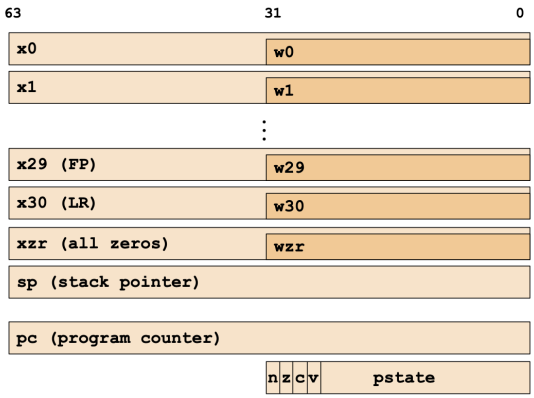
- Instruction set architectures (ISAs) --- defines the low-level instructions a central processing unit (CPU), or processor, can execute
    - Most popular Complex Instruction Set Computer (CISC) architecture: x86 (includes IA32 and x86-64)
    - Most popular Reduced Instruction Set Computer (RISC) architecture: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

New

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` vs `#12` (hexa vs decimal)
- Memory
  - Dereference --- e.g., `[x1]`
  - Add to (offset from) memory address, then dereference --- e.g., `[sp,#16]`

Registers



Load/store operations

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

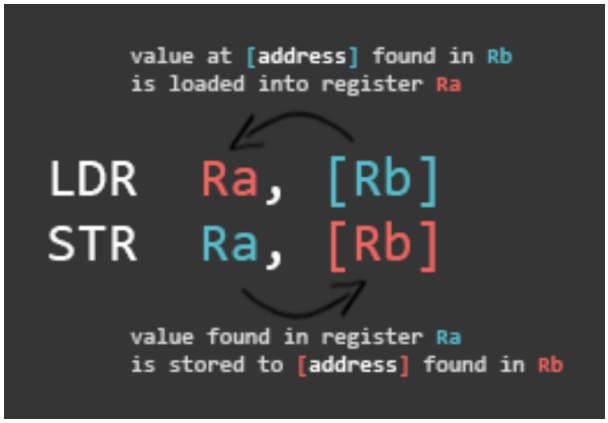
On RISC, this pattern is enforced

- 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. Specifically,

- LDR operation: loads the value at the [memory address] found in Rb to the destination register Ra.
- STR operation: stores the value found in Ra to the [memory address] found in Rb.



- Load (`ldr`) — copies a value from a specific location in main memory into a specific register on the CPU
  - Can copy either 64-bits or 32-bits at a time, depending on whether the instruction specifies `x` or `w` register, respectively
  - Similar to dereferencing a pointer and storing the value in a local variable
    - For example, if we treat registers as if they were variable names, `ldr x0, [x1]` is similar to `x0 = *x1`, where `x0` has type `long` and `x1` has type `long *`
- Store (`str`) — copies a value from a specific register on the CPU into a specific location in main memory
  - Again, can copy 64-bits or 32-bits at a time, depending on whether the instruction specifies `x` or `w` register, respectively
  - Similar to storing the value in a local variable at a memory location pointed-to by a pointer
    - For example, if we treat registers as if they were variable names, `str x0, [x1]` is similar to `*x1 = x0`, where `x0` has type `long` and `x1` has type `long *`

Warm-up: Load/store operations

- Q1: *Write the C code equivalent for each line of assembly, treating registers as if they were variable names.*
  - `ldr x0, [sp]`

    ```
    x0 = *sp
    ```

  - `str w0, [sp]`

    ```
    *sp = w0
    ```

  - `ldr x1, [sp,#12]`

    ```
    x1 = *(sp + 12)
    ```

  - `str x2, [x3,#0x10]`

    ```
    *(x3 + 0x10) = x2
    ```

Arithmetic and logical operations

- (Most) arithmetic and bitwise operations take three operands
    - Register into which to store the result of the operation
    - Register containing the first operand
    - Constant value or register containing the second operand
- Q2: *Write the C code equivalent for each line of assembly, treating registers as if they were variable names.*
    - `lsl w0, w1, w2`

      ```
      w0 = w1 << w2

      //equivalent to 2^n
      ```

    - `and w3, w4, #20`

      ```
      w3 = w4 & 20
      ```

    - `mul w5, w6, #0x11`

      ```
      w5 = w6 * 0x11
      ```

    - `sdiv x7, x8, x9`

      ```
      x7 = x8 / x9
      //signed integer division used 64-bit value, hence x registers
      ```

The `udiv` and `sdiv` instructions operate on 32-bit and 64-bit data respectively. Note that you cannot multiply 32-bit registers with 64 bit registers.

Mapping assembly code to C code

- Q3: *The following C code was compiled into assembly.*

```c
#include <stdio.h>
int years_to_double(int rate) {
    int ruleof72 = 72;
    int years = ruleof72 / rate;
    return years;
}
int main() {
    int r = 10;
    int y = years_to_double(r);
    printf("With an interest rate of %d%% it will take ~%d
        years to double your money\n", r, y);
}
```

For each line of assembly, indicate which original line of C code (above) the assembly instruction was derived from.

```
000000000000076c <years_to_double>:
    76c:    d10083ff    sub sp, sp, #0x20   // 2
    770:    b9000fe0    str w0, [sp, #12]   // 2
    774:    52800900    mov w0, #0x48       // 3
    778:    b9001be0    str w0, [sp, #24]   // 3
    77c:    b9401be1    ldr w1, [sp, #24]   // 4
    780:    b9400fe0    ldr w0, [sp, #12]   // 4
    784:    1ac00c20    sdiv    w0, w1, w0  // 4
    788:    b9001fe0    str w0, [sp, #28]   // 4
    78c:    b9401fe0    ldr w0, [sp, #28]   // 5
    790:    910083ff    add sp, sp, #0x20   // 5
    794:    d65f03c0    ret                 // 5
```

  - *What do each of the columns contain?*
    - Code memory address
    - Bytes corresponding to instruction
    - Operation
    - Operands

- Viewing assembly code
  - Compile: `gcc -fomit-frame-pointer -o years years.c`
  - Disassemble executable: `objdump -d years > years.txt`

- Assembly code (excerpt from `years.txt`)

Translating assembly code to low-level C code

The following C program (`operands.c`) has been compiled into assembly:

```c
int operandsA(int a) {
    return a;
}

long operandsB(long b) {
    return b;
}

int operandsC(int *c) {
    return *c;
}

long operandsD(long *d) {
    return *d;
}

int main() {
    operandsA(5);
    operandsB(5);
    int x = 5;
    operandsC(&x);
    long y = 5;
    operandsD(&y);
}
```

- Q4: *Write the C code equivalent for each line of assembly, treating registers as if they were variable names. The assembly code for the `operandsA` function has already been translated into C code.*

```
00000000000007ec <operandsA>:
    7ec:    d10043ff    sub sp, sp, #0x10   // sp = sp - 0x10
    7f0:    b9000fe0    str w0, [sp, #12]   // *(sp + 12) = w0
    7f4:    b9400fe0    ldr w0, [sp, #12]   // w0 = *(sp + 12)
    7f8:    910043ff    add sp, sp, #0x10   // sp = sp + 0x10
    7fc:    d65f03c0    ret                 // return

0000000000000800 <operandsB>:
    800:    d10043ff    sub sp, sp, #0x10   // sp = sp - 0x10
    804:    f90007e0    str x0, [sp, #8]    // *(sp + 8) = w0
    808:    f94007e0    ldr x0, [sp, #8]    // x0 = *(sp + 8)
    80c:    910043ff    add sp, sp, #0x10   // sp = sp + 0x10
    810:    d65f03c0    ret                 // return

0000000000000814 <operandsC>:
    814:    d10043ff    sub sp, sp, #0x10   // sp = sp - 0x10
    818:    f90007e0    str x0, [sp, #8]    // *(sp + 8) = x0
    81c:    f94007e0    ldr x0, [sp, #8]    // x0 = *(sp + 8)
    820:    b9400000    ldr w0, [x0]        // w0 = *x0
    824:    910043ff    add sp, sp, #0x10   // sp = sp + 0x10
    828:    d65f03c0    ret                 // return

000000000000082c <operandsD>:
    82c:    d10043ff    sub sp, sp, #0x10   // sp = sp - 0x10
    830:    f90007e0    str x0, [sp, #8]    // *(sp + 8) = x0
    834:    f94007e0    ldr x0, [sp, #8]    // x0 = *(sp + 8)
    838:    f9400000    ldr x0, [x0]        // x0 = *x0
```

```
    83c:    910043ff    add sp, sp, #0x10   // sp = sp + 0x10
    840:    d65f03c0    ret                 // return
```

- Q5: *How does the assembly code for `operandsA` and `operandsB` differ? Why?*
    - `operandsA` takes and returns an int, which is 32-bits,
      whereas `operandsB` takes and returns a long, which is 64-bits, so:
        - `operandsA` uses `w0` while `operandsB` uses `x0`
        - `operandsA` stores the parameter at `sp + 12` while `operandsB` stores the parameter at `sp + 8`
- Q6: *How does the assembly code for `operandsB` and `operandsD` differ? Why?*
    - `operandsB` takes and returns a long, whereas `operandsD` takes a pointer to a long and returns a long, so:
    - `operandsD` must deference the pointer (`ldr x0, [x0]`) before returning
- Q7: *How does the assembly code for `operandsC` and `operandsD` differ? Why?*
    - `operandsC` takes a pointer to an int and returns and int,
      whereas `operandsD` takes a pointer to a long and returns a long
    - both take a memory address (a 64-bit value), which is initially in `x0` and stored at `sp + 8`
    - the dereference of the pointer is a 32-bit value in `operandsC` and
      a 64-bit value in `operandsD`, so the value is loaded into
      `w0` in `operandsC` and `x0` in `operandsD`