Function Calls, Memory
Instruction Set Architectures

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

Announcements

- Homework 3 due Next Monday on Gradescope
- Midterm 1 Friday in class
  - Written, closed notes
  - If you have SDAC, please schedule ASAP

Exercises

Q6 Coding ToyISA

1 Point

In our example instruction set (Toy ISA) from class, encoding an operation may mean writing one or more instructions that collectively have exactly the same result as the operation we want. For example, to encode the operation `x = ~y`, we may encode it as `x = y; x = ~x;` (icode 0 then icode 3.0). However, while `y = ~y; x = y;` (icode 3.0 then icode 0) has the same effect, it would also modify y as well as x.

The source code operation `x = y + z;` could be implemented (efficiently using our instruction set as:

- One instruction
- Two instructions
- Three or more instructions
- It cannot be implemented with the Toy ISA instructions

Exercises

Q7 ToyISA Encoding

2 Points

In our example instruction set (Toy ISA) from class, which of the following programs will compute `r0 = r0 - r2`?

- 3a 02
- 06 36 21
- 3a 12 00
- 06 34 21
- None of the above

Suppose we extended the ISA simulator you wrote in Lab 4 with the following code:

```
if (reserved == 1 && icode == 7) {
    R[a] = R[b] >> M[oldPC + 1];
    return oldPC + ___;
}
```

1. [4 points] What is the value that we will need to increment the `oldPC`? (Fill in the blank below.)

```
return oldPC + _____;
```

2. [10 points] Using the new instruction above **at least once**, write a program that determines if both of the hexadecimal digits representing the byte in register 1 are odd, storing a `1` in register 0 if both are odd and a `0` if either are even. For example, if `r1` is `0x73`, store a `0x01` in `r0`; if `r1` is `0xE9`, store a `0x00` in `r0`. Do not change the values stored in `r1`, `r2`, or `r3`. Answer in hexadecimal bytes, separated by spaces. *Hint: You may need to write additional instructions.*

Answer: _____

3. [12 points] Complete the table below listing all the register values as hex digits after the following code executes. Assume that all registers start with value `0x00` and that the first instruction is at address `0x00`.

```
64 05 09 26 4E 62 F8 14 80
```

Register	Value
0	
1	
2	
3	

5. [15 points] Answer the following questions assuming 8-bit two's-complement numbers.

   A. Compute the following sum, showing your work (such as carry bits, etc).

```
    0  0  1  1  1  0  1  1
+   0  1  1  0  0  1  1  1
-------------------------
```

   B. Is your result a positive or negative number? (circle one)

   Positive **Negative**

   C. Convert your solution to decimal.

Answer

7. [8 points] The following number is encoded as an 8-bit floating point number assuming a 4-bit exponent value.

```
01001111
```

A. Write the value in binary scientific notation.

Answer

B. Write the value in decimal.

Answer

10. [14 points] Using only 2-input `and`, `or`, `xor` gates, 1-input `not` gates, and constants `0` and/or `1`, draw a circuit that has 3-bit input a with bits labeled a_2, a_1, and a_0; 3-bit output with bits labeled b_2, b_1, b_0; which computes $b = -a$ in 3-bit two's complement. *Clearly label your inputs and output bits individually.*

High-level Instructions

In general, 3 kinds of instructions

- **moves** - move values around without doing "work"
- **math** - broadly doing "work"
- **jumps** - jump to a new place in the code

Memory

What kinds of things do we put in memory?

- Code: binary code like instructions in our example ISA
  - Intel/AMD compatible: x86_64
  - Apple Mx and Ax, ARM: ARM
  - And others!
- Variables: we may have more variables that will fit in registers
- Data Structures: organized data, collection of data
  - Arrays, lists, heaps, stacks, queues, ...

Dealing with Variables and Memory

What if we have many variables? Compute: x += y

Arrays

Array: a sequence of values (collection of variables)

In Java, arrays have the following properties:

- Fixed number of values
- Not resizable
- All values are the same type

Arrays

Array: a sequence of values (collection of variables)

In Java, arrays have the following properties:

- Fixed number of values
- Not resizable
- All values are the same type

How do we store them in memory?

Arrays

Storing Arrays

In memory, store array sequentially

- Pick address to store array
- Subsequent elements stored at following addresses
- Access elements with math

Example: Store array ***arr*** at **0x90**

- Access ***arr*****[3]** as **0x90 + 3** assuming 1-byte values

What's Missing?

What are we missing?

- Nothing says "this is an array" in memory
- Nothing says how long the array is

Instructions

icode	b	meaning
0		`rA = rB`
1		`rA &= rB`
2		`rA += rB`
3	0	`rA = ~rA`
	1	`rA = !rA`
	2	`rA = -rA`
	3	`rA = pc`
4		`rA` = read from memory at address `rB`
5		write `rA` to memory at address `rB`
6	0	`rA` = read from memory at `pc + 1`
	1	`rA &=` read from memory at `pc + 1`
	2	`rA +=` read from memory at `pc + 1`
	3	`rA` = read from memory at the address stored at `pc + 1`
		For icode 6, increase `pc` by 2 at end of instruction
7		Compare `rA` as 8-bit 2's-complement to 0
		if `rA <= 0` set `pc = rB`
		else increment `pc` as normal

Instructions Set Architecture

Instruction Set Architecture (ISA) is an abstract model of a computer defining how the CPU is controlled by software

- Conceptually, set of instructions that are possible and how they should be encoded
- Results in many different machines to implement same ISA
  - Example: How many machines implement our example ISA?
- Common in how we design hardware

Instructions Set Architecture

Instruction Set Architecture (ISA) is an abstract model of a computer defining how the CPU is controlled by software

- Provides an abstraction layer between:
  - Everything computer is really doing (hardware)
  - What programmer using the computer needs to know (software)
- Hardware and Software engineers have freedom of design, if conforming to ISA
- Can change the machine without breaking any programs

Instructions Set Architecture

Instruction Set Architecture (ISA) is an abstract model of a computer defining how the CPU is controlled by software

- Provides an abstraction layer between:
  - Everything computer is really doing (hardware)
  - What programmer using the computer needs to know (software)

CSO: covering many of the times we'll need to think across this barrier

Instructions Set Architecture

Backwards compatibility

- Include flexibility to add additional instructions later
- Original instructions will still work
- Same program can be run on PC from 10+ years ago and new PC today

Most manufacturers choose an ISA and stick with it

- Notable Exception: Apple

Our Instructions Set Architecture

What about our ISA?

- Enough instructions to compute what we need
- As is, lot of things that are painful to do
  - This was on purpose! So we can see limitations of ISAs early

Our Instructions Set Architecture

What about our ISA?

- Enough instructions to compute what we need
- As is, lot of things that are painful to do
  - This was on purpose! So we can see limitations of ISAs early
- Add any number of new instructions using the reserved bit (7)

Our Instructions Set Architecture

What about our ISA?

- Enough instructions to compute what we need
- As is, lot of things that are painful to do
  - This was on purpose! So we can see limitations of ISAs early
- Add any number of new instructions using the reserved bit (7)
- Missing something important: *Help to put variables in memory*

Storing Variables in Memory

So far... we/compiler chose location for variable

Consider the following example:

```
f(x):
    a=x
    if (x <= 0) return 0
    else return f(x-1) + a
```

Recursion

- The formal study of a function that calls itself

Storing Variables in Memory

```
f(x):
    a=x
    if (x <= 0) return 0
    else return f(x-1) + a
```

Where do we store a?

The Stack

Stack - a last-in-first-out (LIFO) data structure

- The solution for solving this problem

rsp - Special register - the *stack* pointer

- Points to a special location in memory
- Two operations most ISAs support:
  – push - put a new value on the stack
  – pop - return the top value off the stack

The Stack: Push and Pop

push r0

- Put a value onto the “top” of the stack
  - rsp -= 1
  - M[rsp] = r0

pop r2

- Read value from “top”, save to register
  - r2 = M[rsp]
  - rsp += 1

The Stack: Push and Pop

The Stack: Push and Pop

What about real ISAs?