

CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 8: RISC-V Part 1

Instructors: Lisa Yan, Justin Yokota

Computing in the News

- [Memory safe programming languages are on the rise. Here's how developers should respond | ZDNET](#)

Agenda

- Warm-Up: Floating Point
- Intro to Assembly Languages
- RISC-V Programming Paradigm
 - add and sub
 - Immediates: The `addi` instruction
- Demo: Venus

Agenda

- **Warm-Up: Floating Point**
- Intro to Assembly Languages
- RISC-V Programming Paradigm
 - add and sub
 - Immediates: The `addi` instruction
- Demo: Venus

For what positive integers n does $(1.0/n) + (1.0/n) + \dots$ (n times) $= 1.0$?



All n

(A)

Most n

(B)

A few n

(C)

No n

(D)



For what positive integers n does $(1.0/n) + (1.0/n) + \dots$ (n times) $= 1.0$?



(A) All n

0%

(B) Most n

0%

(C) A few n

0%

(D) No n

0%



For what positive integers n does $(1.0/n) + (1.0/n) + \dots$ (n times) $= 1.0$?



(A) All n

0%

(B) Most n

0%

(C) A few n

0%

(D) No n

0%



Agenda

- Warm-Up: Floating Point Questions
- **Intro to Assembly Languages**
- RISC-V Programming Paradigm
 - add and sub
 - Immediates: The `addi` instruction
- Demo: Venus

Building a computer from the ground up

- Any program we write needs to run on a circuit in order to be useful.
- However, circuit-level programming is highly restrictive.
 - With C, a human designed the programming language
 - With a circuit, silicon dictates what language we can design.
- Early computers essentially had to be rebuilt for every program you wanted to run, since different computations required different circuits.
- One major advancement in computer science was the creation of software-based languages.
 - Instead of making a new circuit for every problem you want to solve, make a circuit (called a CPU) that solves the problem “Carry out a sequence of instructions stored as binary data”. Then solve your problem by writing instructions in binary data.

Assembly Language

- We don't want to change the CPU after we build it, so when designing our CPU, we need to decide on a specific set of instructions that will be supported by the CPU, along with a way to translate each instruction to a binary form.
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an Instruction Set Architecture (ISA), and the programming language defined by the ISA is commonly known as an assembly language.
 - Examples: ARM (cell phones), Intel x86 (i9, i7, i5, i3), IBM/Motorola PowerPC (old Macs), MIPS, RISC-V, ...

Assembly Language

- C is generally considered a “lower-level” language than Java or Python, because it’s “closer” to the underlying CPU
 - Less is automated by the language, so you get faster runtimes, but have to keep track of more things
- Ultimately, though, C is still considered a high-level language, because there’s still a lot done for you.
 - C lets you write a bunch of operations in a single line of code, and splits it up for you
 - C sets up the stack for you and keeps track of where it stored local variables
 - C lets you just call a function, and you can expect that calling functions won’t affect any local variables you have.
 - C lets you name variables, and will keep track of that name, and even the type of variable that name refers to.
- Once you start working with assembly languages, almost everything is the result of an explicit instruction by the programmer.

Instruction Set Architectures

- Early trend in ISA design was to add more and more instructions to new CPUs to do elaborate operations
 - VAX architecture had an instruction to multiply polynomials!
- RISC philosophy (Cocke IBM, Patterson, Hennessy, 1980s) – Reduced Instruction Set Computing
 - Keep the instruction set small and simple.
 - Let software do complicated operations by composing simpler ones.
 - A simpler CPU is easier to iterate on (allowing for faster development), and can generally be made faster than a complex CPU (we're often limited by the slowest instruction we decide to implement)

RISC-V

- For the purposes of this class, we'll be learning RISC-V as our assembly language
- Why?
 - RISC-V is relatively simple, in that there's only a few instructions in the base instruction set, and that instructions themselves follow a consistent format.
 - x86 is a more popular language (base CPU for most laptops/desktops), but is a CISC language that Huffman encodes its instructions
 - Project 3: Build a complete RISC-V CPU
 - RISC-V is relatively popular, open-source, and growing in popularity
 - RISC-V was invented in Berkeley in 2010.

RISC-V Resources

- CS 61C Reference Card
 - <https://cs61c.org/sp24/pdfs/resources/reference-card.pdf>
 - Lists out the entire base architecture
- Venus
 - <https://venus.cs61c.org/>
 - Online RISC-V simulator

Agenda

- Warm-Up: Floating Point
- Intro to Assembly Languages
- **RISC-V Programming Paradigm**
 - add and sub
 - Immediates: The `addi` instruction
- Demo: Venus

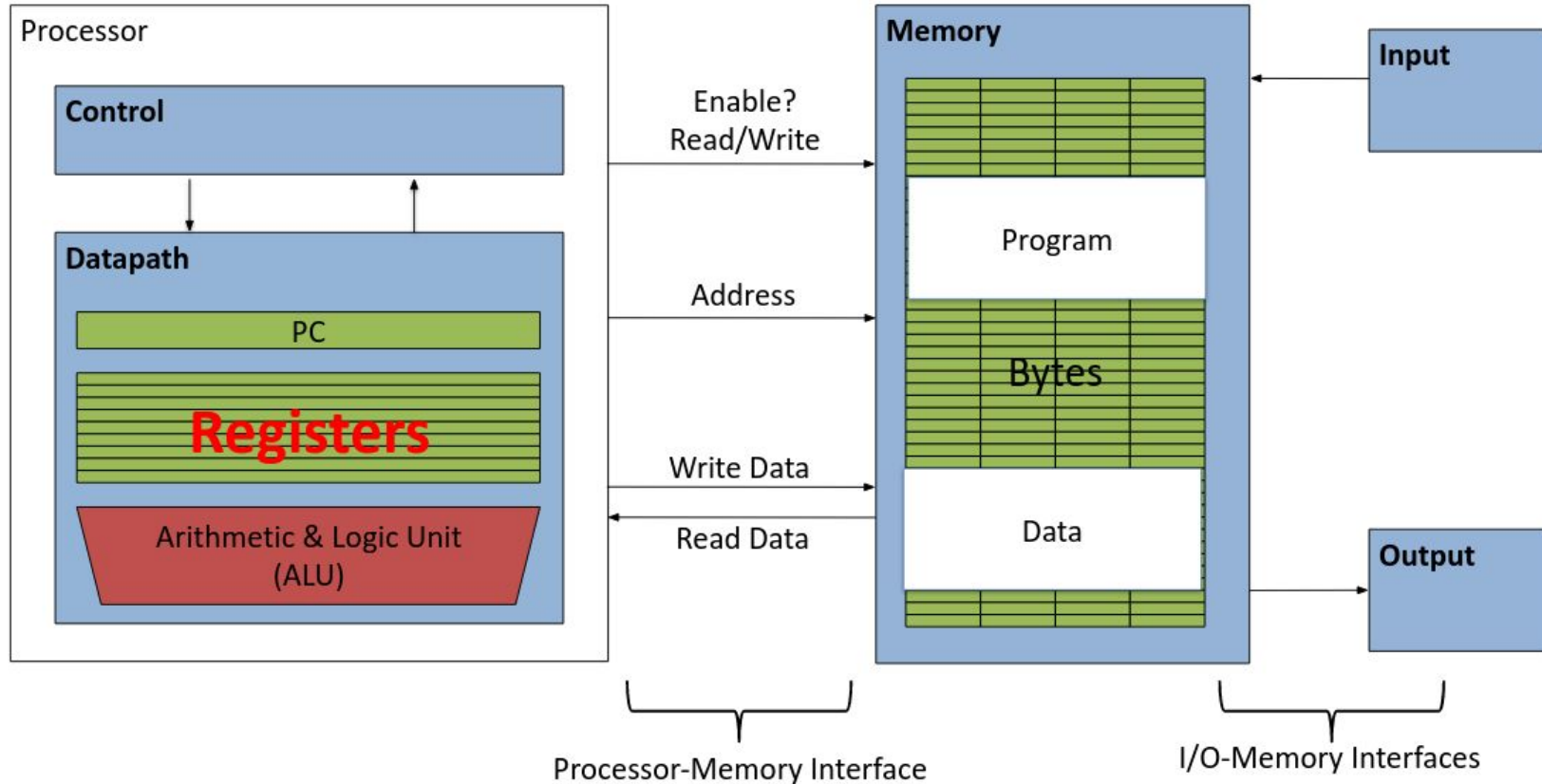
Overarching view of RISC-V

- A RISC-V system is composed of two main parts:
 - The CPU, which is responsible for computing
 - Main memory, which is responsible for long-term data storage (Monday)
- The CPU is designed to be extremely fast, often completing one instruction every nanosecond or faster
 - Note: Light travels 30 cm in 1 nanosecond. In other words, it takes longer for light to travel from one end of my laptop to the other, than it does for a CPU to finish one instruction.
- Going to main memory often takes hundreds or even thousands of times longer.
- The CPU can store a small amount of memory, through components called registers.

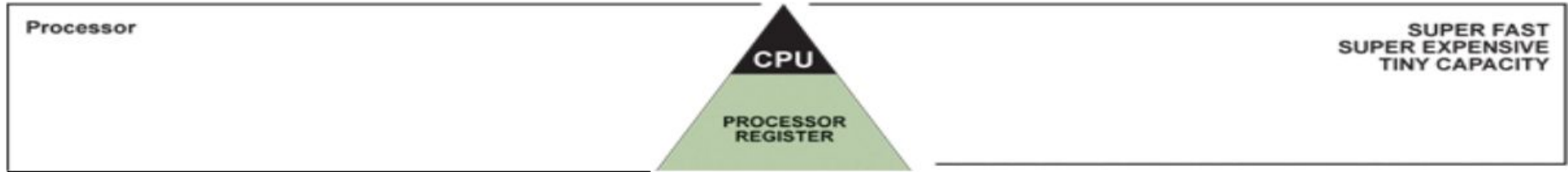
Registers

- A register is a CPU component specifically designed to store a small amount of data. Each register stores 32 bits of data (for a 32-bit system) or 64 bits of data (for a 64-bit system). For the purposes of this class, we consider RV32 only (which uses 32-bit registers)
- This data is purely binary; types do not exist at the assembly level
 - It's the programmer's responsibility to keep track of that register and its type.
- Registers are a hardware component, so once you make the CPU, you can't change the number of registers available.
 - Can't "make" a new register when defining a new variable; have to delete an existing register for each variable you make.
- RISC-V gives access to 32 integer registers

Aside: Registers are Inside the Processor



Great Idea #3: Principle of Locality / Memory Hierarchy



Registers

- Registers are a hardware component, so once you make the CPU, you can't change the number of registers available.
- RISC-V gives access to 32 integer registers
- Registers are numbered from 0 to 31
 - Referred to by number: x0 – x31
- The register x0 is special and always stores 0 (More on this later). So only 31 registers are available to hold variables
- The other 31 registers are all identical in behavior; the only difference between different registers is the conventions we follow when using them.
- Later, we'll give them names to hint at what conventions get used on which registers.

Instructions

- Each line of RISC-V code is a single instruction, which executes a simple operation on registers.
- Instructions are generally written in the format
 - <instruction name> <destination register> <operands>
 - Ex. “add x5 x6 x7” means “Add the values stored in x6 and x7, and store the result in x5”
 - Commas can be added between registers (“add x5, x6, x7”), but this is optional.

Instructions

- Comments are written using the # symbol.
 - Anything written after a # on a line gets ignored
 - Similar to Python comment syntax
- Important: Comments are far more important in RISC-V than in other languages. In a higher level language, you can sometimes get away with choosing variable names so that the code is self-documenting. **In RISC-V, we don't have variable names!**
- Uncommented RISC-V code is practically impossible to debug properly. If you don't comment your code, we may not be able to help debug in office hours.

Addition and Subtraction of Integers (1/3)

Addition in Assembly

- Example: `add x1, x2, x3` (in RISC-V)
- Equivalent to: `a = b + c` (in C)
- where C variables \Leftrightarrow RISC-V registers are:

`a \Leftrightarrow x1, b \Leftrightarrow x2, c \Leftrightarrow x3`

Subtraction in Assembly

- Example: `sub x3, x4, x5` (in RISC-V)
- Equivalent to: `d = e - f` (in C)
- where C variables \Leftrightarrow RISC-V registers are:

`d \Leftrightarrow x3, e \Leftrightarrow x4, f \Leftrightarrow x5`

Addition and Subtraction of Integers (2/3)

How to do the following C statement?

```
a = b + c + d - e;
```

- Break into multiple instructions

```
add x10, x1, x2 # a_temp = b + c
```

```
add x10, x10, x3 # a_temp = a_temp + d
```

```
sub x10, x10, x4 # a = a_temp - e
```

Note: A single line of C may break up into several lines of RISC-V.

Note: Everything after the hash mark on each line is ignored (comments).

Addition and Subtraction of Integers (3/3)

How do we do this?

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

- Use intermediate temporary register

```
add x5, x20, x21 # a_temp = g + h
```

```
add x6, x22, x23 # b_temp = i + j
```

```
sub x19, x5, x6 # f = (g + h) - (i + j)
```

Note: By using x5 and x6 in this way, we overwrite any data that used to be in those registers.

Note: We could also have written this as $f = g + h - i - j$; which allows us to compute this without any temporary registers. A smart compiler may write code this way instead.

Immediates

- Immediates are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:
 - `addi x3,x4,10` (in RISC-V)
 - `f = g + 10` (in C)
 - where RISC-V registers `x3,x4` are associated with C variables `f,g`
- Syntax similar to add instruction, except that last argument is a number instead of a register.
- Common mistake: `addi x3,x4,x5` / `add x3,x4,10` are both invalid RISC-V instructions; be careful with using the register version vs the immediate version of an instruction!

Immediates

- There is no Subtract Immediate in RISC-V: Why?
 - There are add and sub, but no addi counterpart
- Limit types of operations that can be done to absolute minimum
 - if an operation can be decomposed into a simpler operation, don't include it
 - `addi ..., -X = subi ..., X` => so no subi
 - `addi x3, x4, -10` (in RISC-V)
 - `f = g - 10` (in C)
- where RISC-V registers `x3, x4` are associated with C variables `f, g`

Register Zero

- One particular immediate, the number zero (0), appears very often in code.
- So the register zero (**x0**) is ‘hard-wired’ to value 0; e.g.

`add x3, x4, x0` (in RISC-V)

`f=g` (in C)

- where RISC-V registers **x3**, **x4** are associated with C variables `f`, `g`
- Defined in hardware, so an instruction
`add x0, x3, x4` will not do anything!

Agenda

- Warm-Up: Floating Point
- Intro to Assembly Languages
- RISC-V Programming Paradigm
 - add and sub
 - Immediates: The `addi` instruction
- **Demo: Venus**

Venus Demo

- Done in lecture
- Add 1+2+3+4+5

```
addi x5 x0 0
addi x6 x0 5
add x5 x5 x6
addi x6 x6 -1
add x5 x5 x6
addi x6 x6 -1
add x5 x5 x6
addi x6 x6 -1
add x5 x5 x6
addi x6 x6 -1
```