Computer Systems

Lecture 7

Computer Architecture

Dr José Cano, Dr Lito Michala
School of Computing Science
University of Glasgow
Spring 2020

Outline

- Some history: early computing machines
- Computer Architecture
- Instructions
- Memory

How computers developed

- Adding machines
- Machines that could add, subtract, multiply, divide
- Machines that could do fixed sequences of arithmetic
- Machines that could look at the results of arithmetic and then decide what to do next

Machine arithmetic

- Machines have been used to help with arithmetic for along time
  - Some assist the human (abacus)
  - Others perform arithmetic mechanically (Pascal's adder)

- Arithmetic with gears (19th century technology)
  - Video showing how carry propagation can be done with gears

    http://www.youtube.com/watch?v=YXMuJco8onQ
  - Video of Pascal calculator

    http://www.youtube.com/watch?v=3h71HAJWnVU

- Binary arithmetic with marbles (just for fun!)
  - http://www.youtube.com/watch?v=GcDshWmhF4A
  - http://www.youtube.com/watch?v=md0TlSjIags

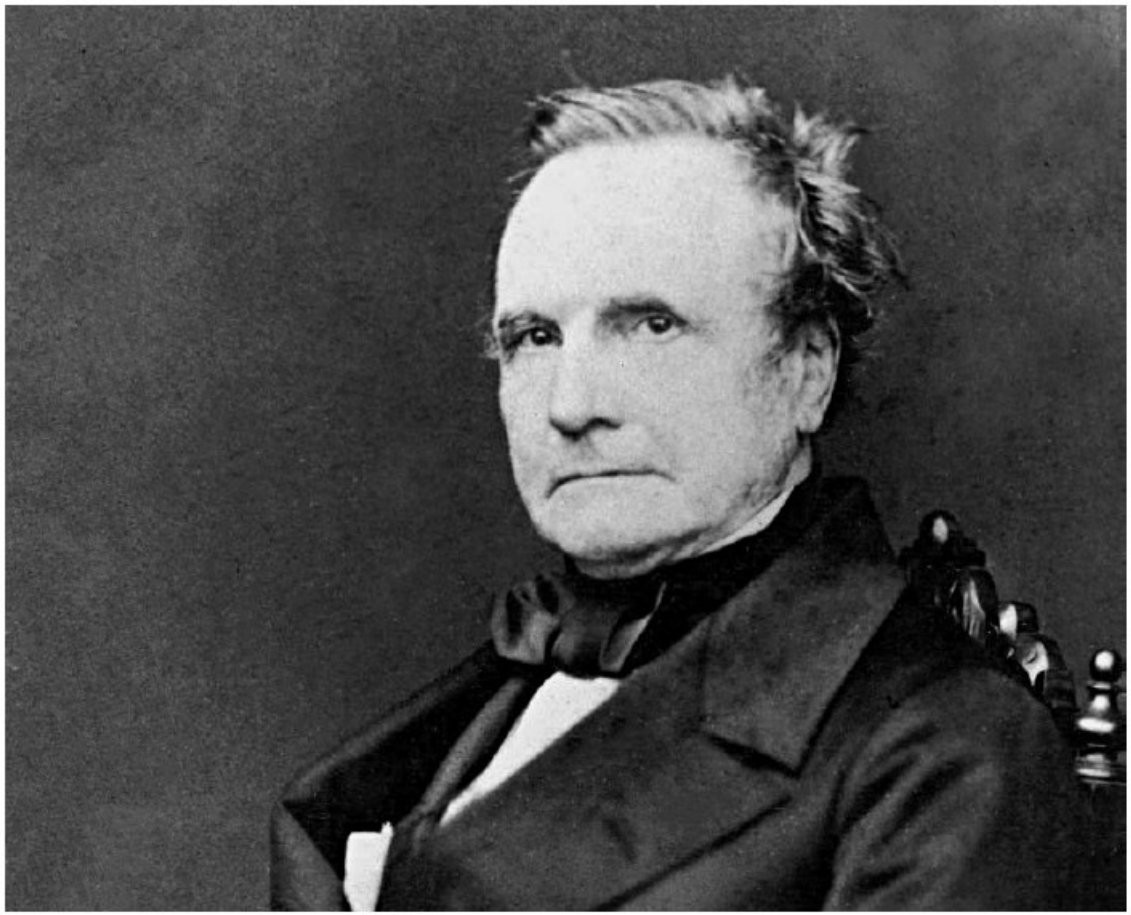
Three key ideas

- Make a machine that performs a fixed sequence of arithmetic operations

- Provide a way to change that sequence
  - Configure the machine for a specific problem by defining the sequence of operations to perform
  - **Jacquard loom**: used punched cards
  - **Babbage's Difference Engine**: used fixed sequences of arithmetic operations to calculate functions for nautical tables

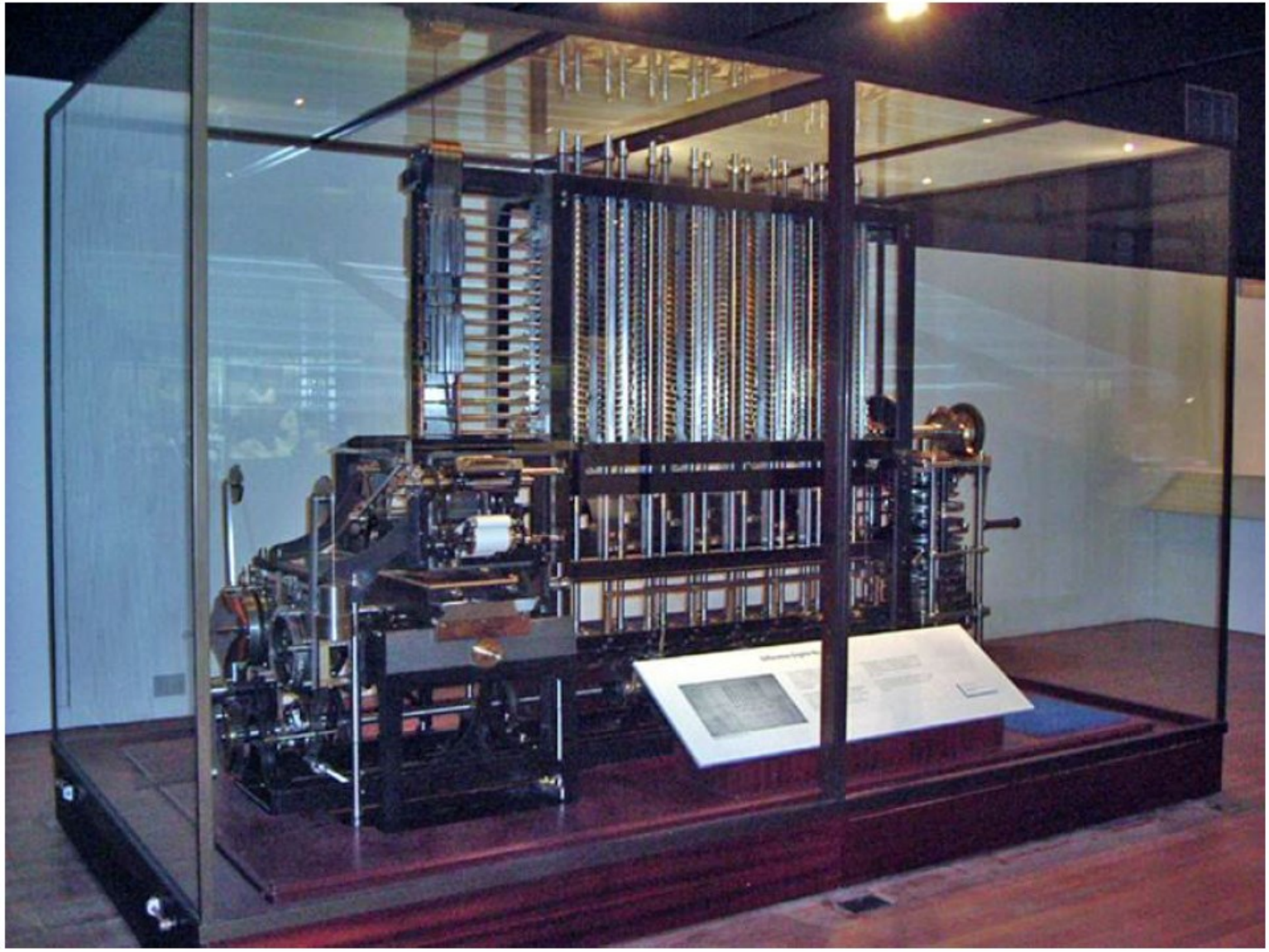
- Make it possible to compare two numbers, and then decide what to do next
  - **Babbage's Analytical Engine**: first general "Turing-complete" computer
  - Instead of computing a fixed sequence of operations, there is something like a conditional if x<y then ... else ...

Charles Babbage (1791-1871)



"Inventor of the computer"

Babbage's Difference Engine

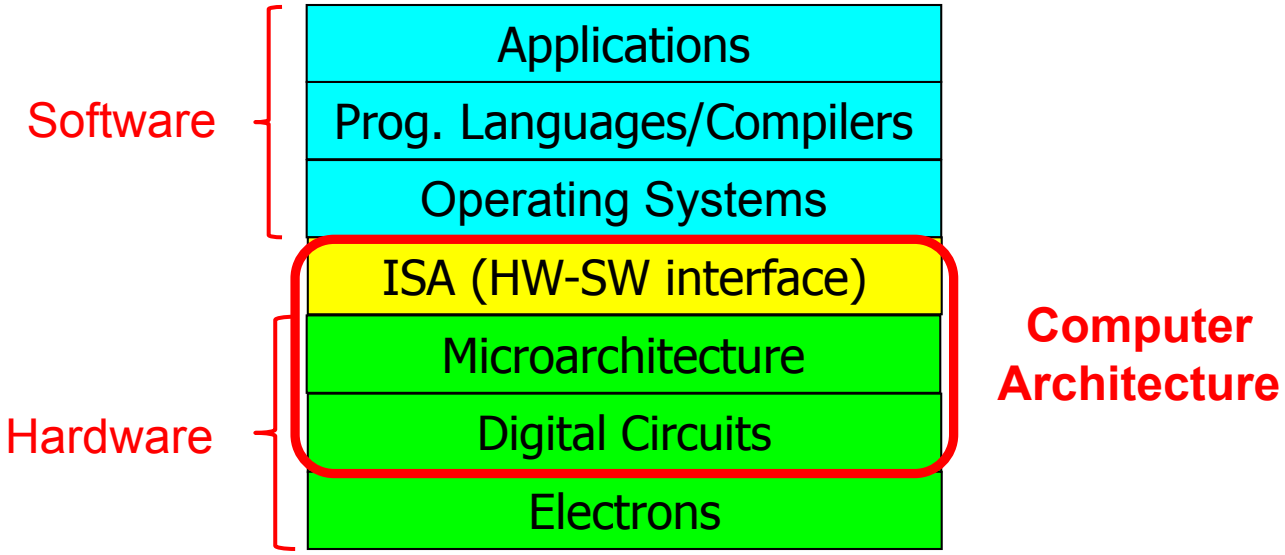


Outline

- Some history: early computing machines
- Computer Architecture
- Instructions
- Memory

Computer Architecture

- CA defines the structure and the machine language of a computer



- Understanding the principles of CA is central to computer science

- **Practical benefit**: understanding machine language will give you a deeper understanding of programming languages

The Language of Machines

- There are many programming languages
  - Python, Java, C, etc
- Each machine has one fixed machine language
- Why are there many programming languages for a given machine?
  - Each PL is translated to machine language by software called a compiler
- What is a computer?
  - A digital circuit (a piece of hardware, a machine) that executes programs

Machine language

- A fixed digital circuit can execute one fixed machine language

- Examples
  - X86 (Intel, AMD)
  - ARM
  - MIPS
  - Sparc
  - … and many more

- The details of different machine languages are quite different

- But we will focus on principles common to all of them

What is machine language like?

- Very different from Python, Java, C, …

- The designer of a machine language has to "look both up and down"
  - **Looking up**: the machine language must be powerful enough to provide the foundation for operating systems and programming languages
  - **Looking down**: the machine language must be simple enough so that a digital circuit can execute it (example: RTM circuit)

- Machine languages are also designed to make high performance possible

- But they are not intended to make programming easy!

- Today we'll see assembly language: a simplified notation of machine language

Outline

- Some history: early computing machines
- Computer Architecture
- Instructions
- Memory

Instructions

- A machine language provides instructions

- Analogous to statements in a programming language, with some differences
  - Statements can be complex: x := 2 * (a + b/c)
  - Instructions are simple: R2 := R1 + R3
  - Each instruction performs just one operation

Sigma16

- In this course we will study an architecture called Sigma16
- Designed to support several research projects at the University of Glasgow
- It's a research machine, not a commercial product
- There is a complete design, including a full digital circuit
- It hasn't been manufactured, but there are two implementations in software
  - An emulator, which you will use in this course
  - A simulator for the circuit

Why use Sigma16?

- Our focus is on ideas and principles
- Sigma16 illustrates all the main ideas, but avoids unnecessary complexity
- Example
  - Sigma16 has just one word size (16 bits) while commercial machines provide many
  - Most commercial computers have backward compatibility with previous versions, leading to great complexity
- Legacy architectures use an approach called complex instruction set (CISC)
- Simpler reduced instruction set (RISC) gives better performance
  - Sigma16 uses this

Structure of a computer

- All computers have several main subsystems
  - The register file is a set or array of “registers”
    - The RTM circuit has 4, sigma16 has 16 (R0, R1, R2, …, R15)
    - A register is a circuit that can remember a 16-bit word
  - The ALU (arithmetic and logic unit) is a circuit that can do arithmetic, such as addition, subtraction, comparison, and some other operations
  - The memory can hold a large number of words: similar to the register file, but significantly slower and much larger
  - The Input/Output can transfer data from the outside world to/from the memory

Register file

- There are 16 registers
- Each register holds a 16-bit word
- We'll write the words using hexadecimal

R0	0000
R1	fffe
R2	13c4
⋮	⋮
R14	03c8
R15	0020

What are the registers actually?

- Recall the register transfer machine circuit
- It contains four registers, each holding 4 bits
- Sigma16 is just the same, but with 16 registers and each holds 16 bits
- Each 16-bit register is 16 copies of the reg1 circuit
- Why program with registers, not variables like sum, count, x, etc?
  - In machine language we are programming directly with the hardware in the computer

The RTM instructions

- The RTM circuit can execute two instructions

```
R2 := R1 + R0        ; add two registers and load result
R1 := 8              ; load a constant
```

- We'll begin with the corresponding Sigma16 instructions

The add instruction

- Think of the registers as variables

- Examples

```
add R5,R2,R3      ; means R5 := R2 + R3
add R12,R1,R7     ; means R12 := R1 + R7
```

- General form:
  - add dest,op1,op2 where dest, op1, op2 are registers
  - The two operands are added, the result is placed in the destination
  - Meaning: dest := op1 + op2

- Everything after a semicolon ; is a comment

Registers can hold variables

- We often think of a variable as a box that can hold a number
  - A register can hold a variable!

- An add instruction (or sub, mul, div) is like an assignment statement
  - add R2,R8,R2 means R2 := R8 + R2
    - Evaluate the right hand side R8 + R2
    - The operands (R8, R2) are not changed
    - Overwrite the left hand side R2 (destination) with the result
    - The old value of the destination is destroyed
  - It is not a mathematical equation, it is a command to do an operation and put the result into a register, overwriting the previous contents

- Assignment is often written R2 := R8 + R2
  - The := operator means assign, and does not mean equals

Notation and terminology

- Why write a notation like add R5,R2,R3 instead of R5 := R2 + R3?
  - It's actually more consistent because every instruction will be written in this form: a keyword for the operation, followed by the operands
  - The notation is related closely to the way instructions are represented in memory, which we'll see later

A simple program

- **Problem**: given three integers in R1, R2, R3
- **Goal**: calculate the sum R1+R2+R3 and put it in R4
- **Solution**

```
add R4,R1,R2        ; R4 := R1+R2 (this is a comment)
add R4,R4,R3        ; R4 := (R1+R2) + R3
```

More arithmetic instructions

- There are instructions for the basic arithmetic operations

```
add R4,R11,R0      ; R4 := R11 + R0
sub R8,R2,R5       ; R8 := R2 - R5
mul R10,R1,R2      ; R10 := R1 * R2
div R7,R2,R12      ; R7 := R2 / R12
```

- Every arithmetic operation takes its operands from registers, and loads the result into a register

Example

- Suppose we have variables a, b, c, d
- R1=a, R2=b, R3=c, R4=d
- We wish to compute R5 = (a+b) * (c-d)

```
add R6,R1,R2      ; R6 := a + b
sub R7,R3,R4      ; R7 := c - d
mul R5,R6,R7      ; R5 := (a+b) * (c-d)
```

- Good comments make the code easier to read!

General form of arithmetic instruction

- General form: op d,a,b

  op: operation (+, −, ×, ÷)

  d: destination register (where the result goes)

  a: first operand register

  b: second operand register

- **Meaning**: Rd := Ra (op) Rb

- **Example**: add R5,R2,R12 ; R5 := R2+R12

Register R0 and R15 are special!

- You should not use R0 or R15 to hold ordinary variables!

- R0 always contains 0
  - Any time you need the number 0, it's available in R0
  - You cannot change the value of R0
  - add R0,R2,R3 ; does nothing (R0 will not change)
  - add R5,R2,R3 ; you can change all other registers
  - It is legal to use R0 as the destination, but it will still be 0 after you do it!

- R15 holds status information
  - Some instructions place additional information in R15 (is the result negative? was there an overflow?)
  - Therefore the information in R15 is transient
  - R15 is for temporary information (not a safe place to keep long-term data)

Limitation of register file: it's small

- The register file is used to perform calculations
- In computing something like x := (2*a + 3*b) / (x-1)
  - All the arithmetic will be done using the register file
- But it has a big limitation
  - There are only 16 registers
  - And most programs need more than 16 variables!
- **Solution**: the memory is large and can hold far more data than the register file

Outline

- Some history: early computing machines
- Computer Architecture
- Instructions
- Memory

Memory

- The memory is similar to the register file
  - It is a large collection of words
- A variable name (x, sum, count) refers to a word in memory
- Some differences between memory and register file
  - The memory is much larger: 65,536 locations (the register file has only 16)
  - The memory cannot do arithmetic
- So our strategy in programming
  - Keep data permanently in memory
  - When you need to do arithmetic, copy a variable from memory to a register
  - When finished, copy the result from a register back to memory

Registers and memory

- Register file
  - 16 registers
  - Can do arithmetic, but too small to hold all your variables
  - Each register holds a 16-bit word
  - Names are R0, R1, R2, ..., R15
  - You can do arithmetic on data in the registers
  - Use registers to hold data temporarily that you're doing arithmetic on

- Memory
  - 65,536 memory locations
  - Each memory location holds a 16-bit word
  - Each memory location has an address 0, 1, 2, …, 65,535
  - The machine cannot do arithmetic on a memory location
  - Use memory locations to store program variables permanently and the program itself

Copying a word between memory and register

- There are two instructions for accessing the memory
  - load copies a variable from memory to a register
    - load R2,x[R0] copies the variable x from memory to register R2
    - R2 := x
    - R2 is changed; x is unchanged
  - store copies a variable from a register to memory
    - store R3,y[R0] copies the word in register R3 to variable y in memory
    - y := R3
    - y is changed; R3 is unchanged

- Notice that we write [R0] after a variable name
  - Later we'll see the reason

An assignment statement in machine language

- x := a+b+c

```
load R1,a[R0]      ; R1 := a
load R2,b[R0]      ; R2 := b
add R3,R1,R2       ; R3 := a+b
load R4,c[R0]      ; R4 := c
add R5,R3,R4       ; R5 := (a+b) + c
store R5,x[R0]     ; x := a+b+c
```

- Use load to copy variables from memory to registers
- Do arithmetic with add, sub, mul, div
- Use store to copy result back to memory

Why do we have registers and memory?

- The programmer has to keep track of which variables are currently in registers
- You have to use load and store instructions to copy data between the registers and memory
- Wouldn't it be easier just to get rid of the distinction between registers and memory?
  - Do all the arithmetic on memory
- Short answer
  - Yes, it's possible to design a computer that way
  - But it makes the computer very much slower!
  - A computer without load and store instructions (where you do arithmetic on memory locations) would run between 100 and 1,000 times slower

Constants: the lea instruction

- The RTM has an instruction that loads a constant into a register

- Use the lea instruction
  - lea R2,57[R0] loads the constant 57 into R2 ; R2 := 57
  - Actually lea does more (later we'll see some advanced applications)
  - General form: lea Rd,const[R0]
  - Write [R0] after the constant; we'll see the reason for this later on

- **Example**: R3 := R1 + 39*R2

```
lea R4,39[R0]      ; R4 := 39
mul R3,R4,R2       ; R3 := 39 * R2
add R3,R1,R3       ; R3 := R1 + (39*R2)
```

Stopping the program

- The last instruction of every program should be

```
trap R0,R0,R0        ; halt
```

- This tells the computer to halt, it stops execution of the program

Defining variables

- To define variables x, y, z and give them initial values

```
x data 34          ; x is a variable with initial value 34
y data 9           ; y is initially 9
z data 0           ; z is initially 0
abc data $02c6     ; specify initial value as hex
```

- The data statements should come after all the instructions in the program

  Reason: the booter

A complete example program

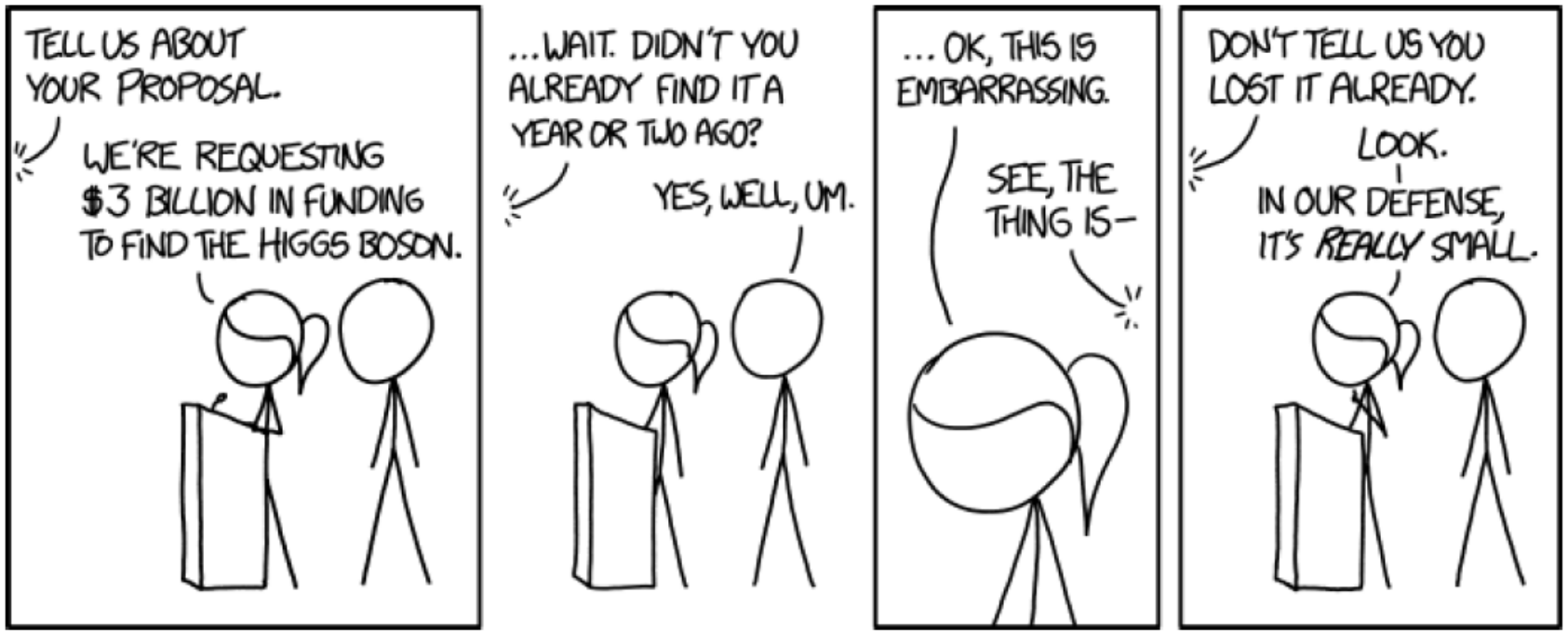
- ; Program Add
- ; A minimal program that adds two integer variables

- ; Execution starts at location 0, where the first instruction will be
- ; placed when the program is executed

```
load R1,x[R0]       ; R1 := x
load R2,y[R0]       ; R2 := y
add R3,R1,R2        ; R3 := x + y
store R3,z[R0]      ; z := x + y
trap R0,R0,R0       ; terminate
```

- ; Static variables are placed in memory after the program

```
x data 23
y data 14
z data 99
```



https://xkcd.com/1437/