UC Berkeley
Teaching Professor
Dan Garcia

# CS61C

## Great Ideas in Computer Architecture (a.k.a. Machine Structures)

UC Berkeley
Professor
Bora Nikolić

# RISC-V Assembly Language

# Great Idea #1: Abstraction (Levels of Representation/Interpretation)

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

*Compiler*

Assembly Language Program (e.g., RISC-V)

```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

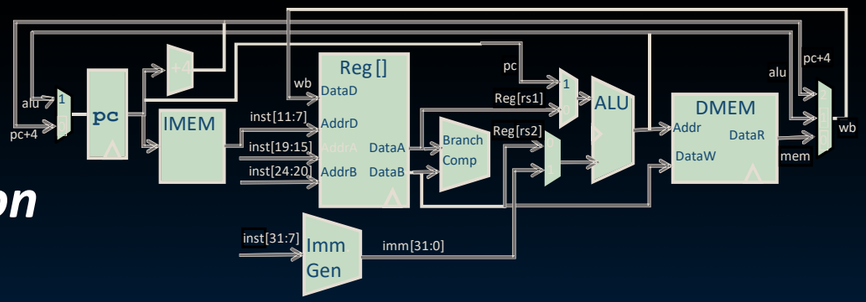
Anything can be represented as a number, i.e., data or instructions

*Assembler*

Machine Language Program (RISC-V)

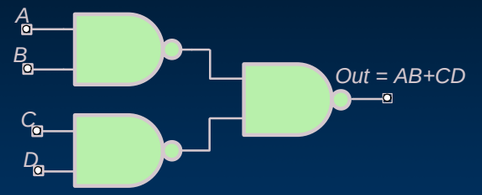
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

Hardware Architecture Description (e.g., block diagrams)




*Architecture Implementation*

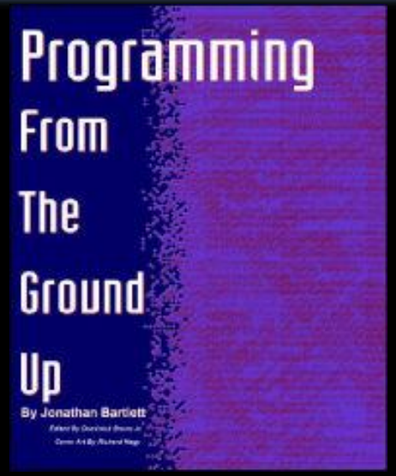
Logic Circuit Description (Circuit Schematic Diagrams)



# Assembly Language

- Basic job of a CPU: execute lots of *instructions*.
- Instructions are the primitive operations that the CPU may execute.
  - Like a sentence: operations (verbs) applied to operands (objects) processed in sequence …
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an *Instruction Set Architecture* (*ISA*).
  - Examples: ARM (cell phones), Intel x86 (i9, i7, i5, i3), IBM Power, IBM/Motorola PowerPC (old Macs), MIPS, RISC-V, …

# Book: Programming From the Ground Up




*"A new book was just released which is based on a new concept - teaching computer science through assembly language (Linux x86 assembly language, to be exact). This book teaches how the machine itself operates, rather than just the language. I've found that the key difference between mediocre and excellent programmers is whether or not they know assembly language.* ***Those that do tend to understand computers themselves at a much deeper level.*** *Although [almost!] unheard of today, this concept isn't really all that new -- there used to not be much choice in years past. Apple computers came with only BASIC and assembly language, and there were books available on assembly language for kids. This is why the old-timers are often viewed as 'wizards': they* ***had*** *to know assembly language programming."*

-- **slashdot.org** comment, 2004-02-05


# Instruction Set Architectures

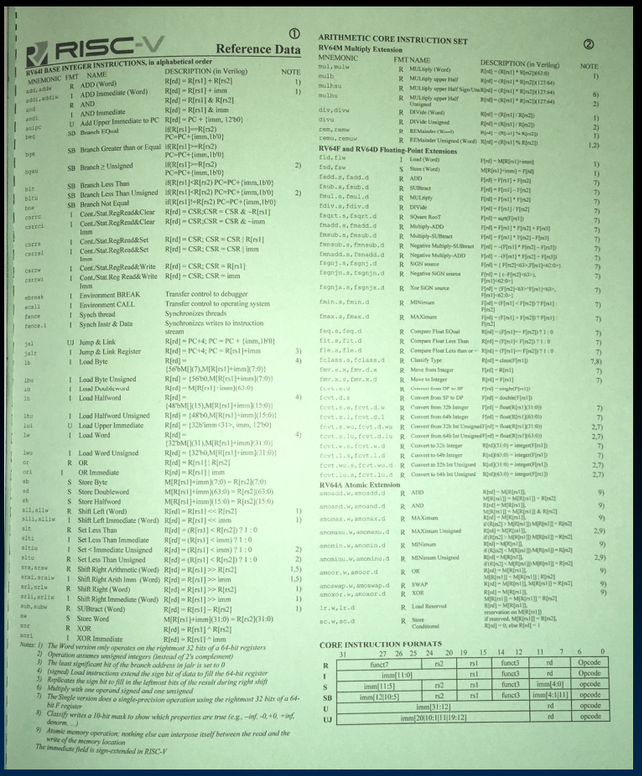
- Early trend was to add more and more instructions to new CPUs to do elaborate operations
  - VAX architecture had an instruction to multiply polynomials!
- RISC philosophy (Cocke IBM, Patterson, Hennessy, 1980s) – Reduced Instruction Set Computing
  - Keep the instruction set small and simple, makes it easier to build fast hardware.
  - Let software do complicated operations by composing simpler ones.
  - This went against the convention wisdom of the time. (he who laughs last, laughs best)

# Patterson and Hennessy win Turing!

# RISC-V Architecture

IBM 360 Green Card


- New open-source, license-free ISA spec
  - Supported by growing shared software ecosystem
  - Appropriate for all levels of computing system, from microcontrollers to supercomputers
  - 32-bit, 64-bit, and 128-bit variants (we're using 32-bit in class, textbook uses 64-bit)
- Why RISC-V instead of Intel 80x86?
  - RISC-V is simple, elegant. Don't want to get bogged down in gritty details.
  - RISC-V has exponential adoption



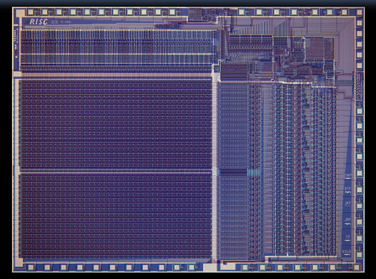

RISC-V Green Card

https://cs61c.org/resources/

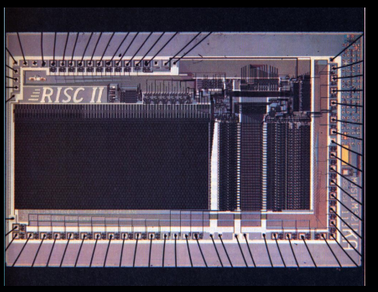
Garcia, Nikolić

# RISC-V Origins

- Started in Summer 2010 to support open research and teaching at UC Berkeley
  - Lineage can be traced to RISC-I/II projects (1980s)
- As the project matured, it migrated to RISC-V foundation (www.riscv.org )
- Many commercial and research projects based on RISC-V, open-source and proprietary
  - Widely used in education
- Read more:
  - https://riscv.org/risc-v-history/
    https://riscv.org/risc-v-genealogy/



RISC-I




RISC-II

# Elements of Architecture: Registers

# Instruction Set

**Preliminary discussion of the logical design of an electronic computing instrument**[1]

*Arthur W. Burks / Herman H. Goldstine / John von Neumann*

3.1. It is easy to see by formal-logical methods that there exist codes that are *in abstracto* adequate to control and cause the execution of any sequence of operations which are individually available in the machine and which are, in their entirety, conceivable by the problem planner. The really decisive considerations from the present point of view, in selecting a code, are more of a practical nature: simplicity of the equipment demanded by the code, and the clarity of its application to the actually important problems together with the speed of its handling of those problems. It would take us much too far afield to discuss these questions at all generally or from first principles. We will therefore restrict ourselves to analyzing only the type of code which we now envisage for our machine.

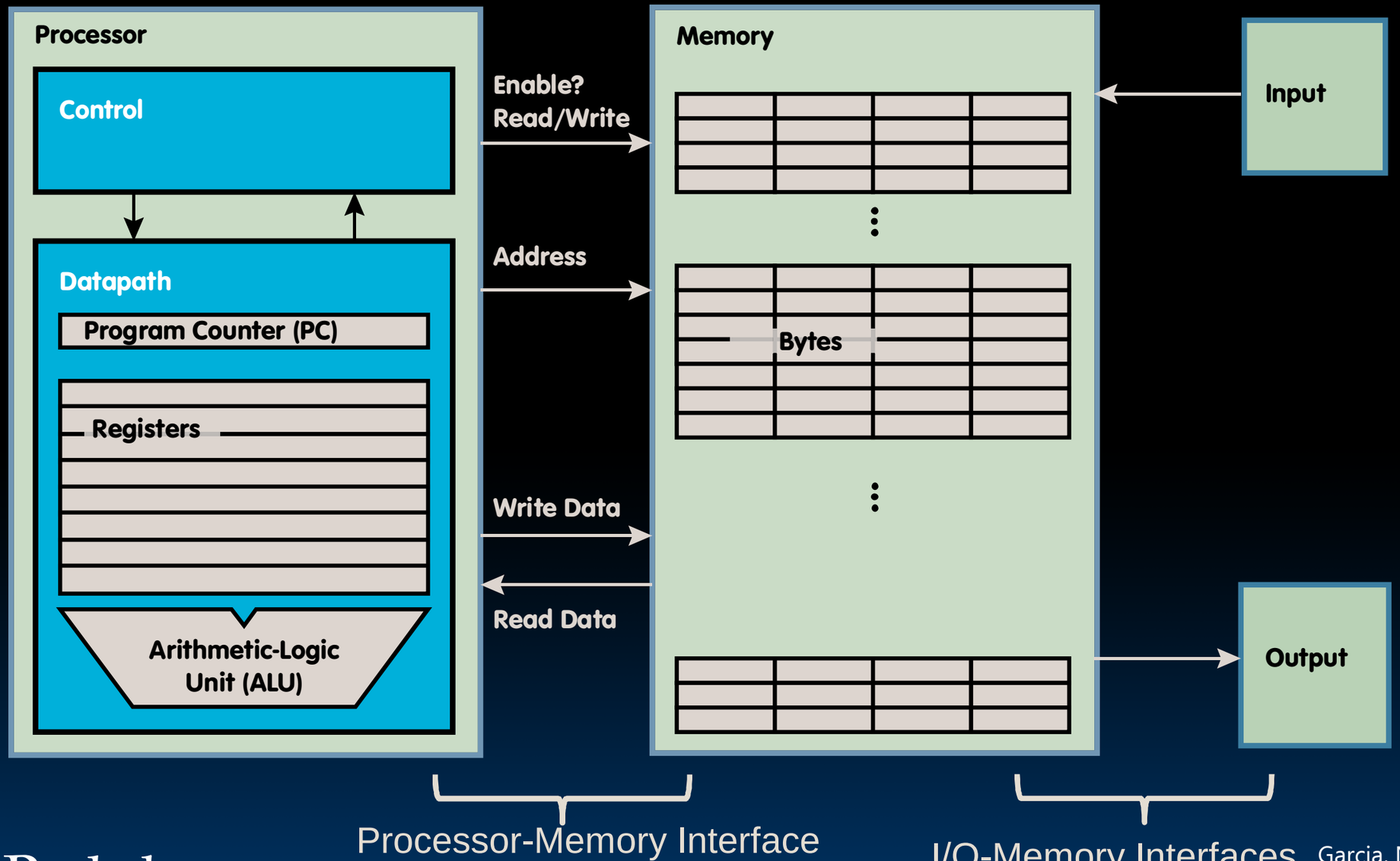
"instruction sets"

- Instruction set for a particular architecture (e.g. RISC-V) is represented by the Assembly language
- Each line of assembly code represents one instruction for the computer

# Assembly Variables: Registers (1/3)

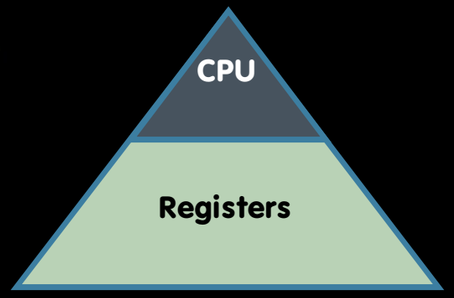
- Unlike HLL like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Assembly operands are registers
  - Limited number of special locations built directly into the hardware
  - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they're very fast (faster than 0.25ns)
  - Recall light is $3x10^8$m/s = 0.3m/ns = 30cm/ns = 10cm/0.3ns!!!... where 0.3ns is the clock period of a 3.33GHz computer

# Aside: Registers are Inside the Processor



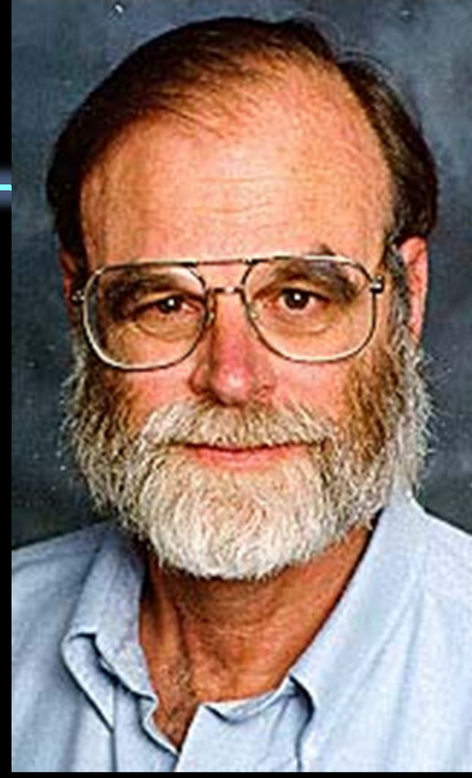
# Great Idea #3: Principle of Locality / Memory Hierarchy

Processor chip




Extremely fast
Extremely expensive
Tiny capacity

# Jim Gray's Storage Latency Analogy: How Far Away is the Data?



Jim Gray
Turing Award
B.S. Cal 1966
Ph.D. Cal 1969


| | | | |
|---|---|---|---|
| 1 [ns] | Registers | My Head | 1 min |

# Assembly Variables: Registers (2/3)

- Drawback: Since registers are in hardware, there is a predetermined number of them
  - Solution: RISC-V code must be very carefully put together to efficiently use registers
- 32 registers in RISC-V
  - Why 32?
    Smaller is faster, but too small is bad. Goldilocks principle ("This porridge is too hot; This porridge is too cold; this porridge is just right")
- Each RISC-V register is 32 bits wide (in RV32 variant)
  - Groups of 32 bits called a word in RV32
  - P&H textbook uses the 64-bit variant RV64

# Assembly Variables: Registers (3/3)

- Registers are numbered from 0 to 31
  - Referred to by number **`x0`** – **`x31`**
- **`x0`** is special, always holds value zero
  - So only 31 registers able to hold variable values
- Each register can be referred to by number or name
  - Will add names later

# C, Java variables vs. registers

- In C (and most high-level languages) variables declared first and given a type. E.g.,

```
int fahr, celsius;
char a, b, c, d, e;
```

- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match `int` and `char` variables).
- In assembly language, the registers have no type
  - Operation determines how register contents are treated

# Comments in Assembly

- Make your code more readable: comments!
- Hash (#) is used for RISC-V comments
  - anything from hash mark to end of line is a comment and will be ignored
  - This is just like the C99 //
- Note: Different from C.
  - C comments have format
    /* comment */
    so they can span many lines

# Aside: Apollo Guidance Computer

Margaret Hamilton
(Wikimedia commons)

```
179                     TC      BANKCALL        # TEMPORARY, I HOPE HOPE HOPE
180                     CADR    STOPRATE        # TEMPORARY, I HOPE HOPE HOPE
181                     TC      DOWNFLAG        # PERMIT X-AXIS OVERRIDE
```

```
245                     CAF     CODE500         # ASTRONAUT:    PLEASE CRANK THE
246                     TC      BANKCALL        #               SILLY THING AROUND
247                     CADR    GOPERF1
248                     TCF     GOTOP00H        # TERMINATE
249                     TCF     P63SPOT3        # PROCEED       SEE IF HE'S LYING
250
251     P63SPOT4        TC      BANKCALL        # ENTER         INITIALIZE LANDING RADAR
252                     CADR    SETPOS1
253
254                     TC      POSTJUMP        # OFF TO SEE THE WIZARD ...
255                     CADR    BURNBABY
```

Assembly code with comments
(ABC News, 2018)

# Assembly Instructions

- In assembly language, each statement (called an Instruction), executes exactly one of a short list of simple commands
- Unlike in C (and most other high-level languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, *, /) in C or Java
- Ok, enough already…gimme my RV32!

# RISC-V Add/Sub Instructions

# RISC-V Addition and Subtraction (1/4)

- Syntax of Instructions:
  - one two, three, four

```
add x1, x2, x3
```

  - where:
  - one = operation by name
  - two = operand getting result ("destination," **x1**)
  - three = 1st operand for operation ("source1," **x2**)
  - four = 2nd operand for operation ("source2," **x3**)
- Syntax is rigid:
  - 1 operator, 3 operands
  - Why? Keep hardware simple via regularity

# Addition and Subtraction of Integers (2/4)

- Addition in Assembly
  - Example: `add x1,x2,x3` (in RISC-V)
  - Equivalent to: a = b + c (in C)
  - where C variables ⇔ RISC-V registers are:
    a ⇔ **x1**, b ⇔ **x2**, c ⇔ **x3**
- Subtraction in Assembly
  - Example: `sub x3,x4,x5` (in RISC-V)
  - Equivalent to: d = e - f (in C)
  - where C variables ⇔ RISC-V registers are:
    d ⇔ **x3**, e ⇔ **x4**, f ⇔ **x5**

# Addition and Subtraction of Integers (3/4)

- 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
```

- Notice: A single line of C may break up into several lines of RISC-V.
- Notice: Everything after the hash mark on each line is ignored (comments).

# Addition and Subtraction of Integers (4/4)

- 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)
```

- A good compiler may do:

# RISC-V Immediates

# 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.

# 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, respectively

# 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!
```