

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;
```

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

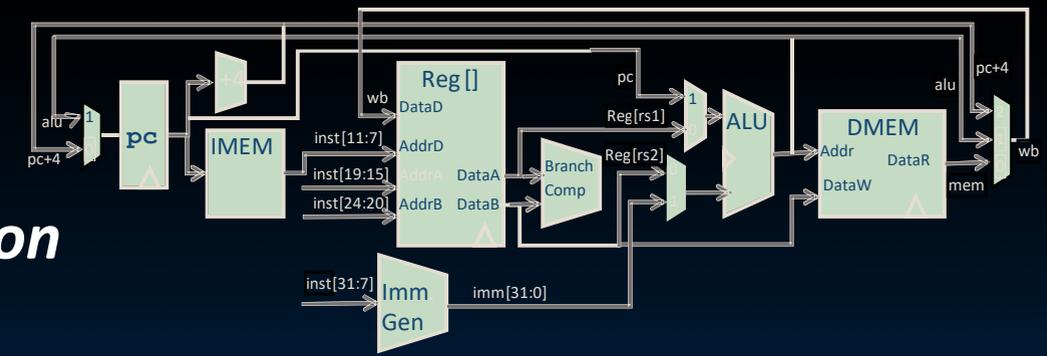
```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

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

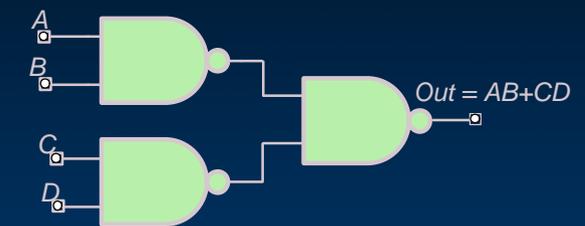
Machine Language  
Program (RISC-V)

```
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
```

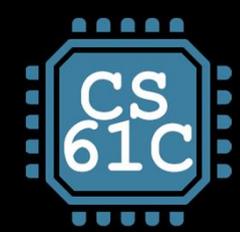
Hardware Architecture Description  
(e.g., block diagrams)



Logic Circuit Description  
(Circuit Schematic Diagrams)



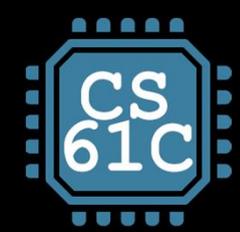
Architecture Implementation



# Assembly Language

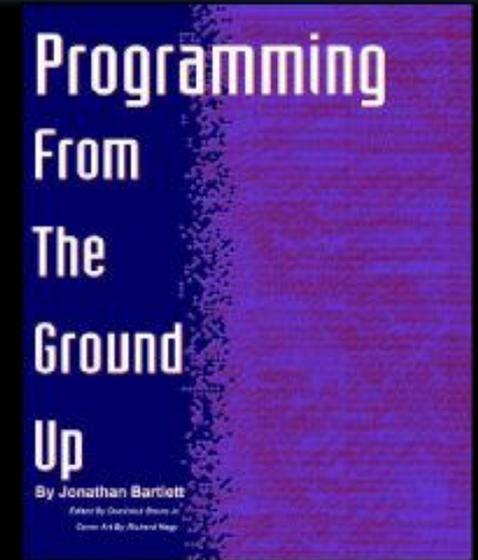
---

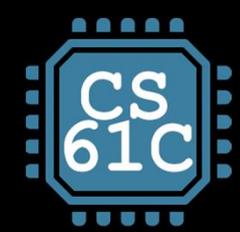
- Basic job of a CPU: execute lots of *instructions*.
- Instructions are the primitive operations that the CPU may execute.
  - Like a sentence: operations (verbs) applied to operands (objects) processed in sequence ...
- Different CPUs implement different sets of instructions. The set of instructions 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.”*

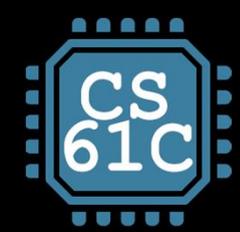




# Instruction Set Architectures

---

- 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!





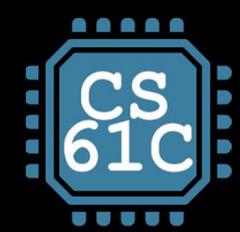
# RISC-V Architecture

## IBM 360 Green Card

Address	Instruction	Operation	Notes
0000	ADD	ADD	
0001	ADDI	ADDI	
0002	ADDI16	ADDI16	
0003	ADDI32	ADDI32	
0004	ADDI64	ADDI64	
0005	ADDI128	ADDI128	
0006	ADDI256	ADDI256	
0007	ADDI512	ADDI512	
0008	ADDI1024	ADDI1024	
0009	ADDI2048	ADDI2048	
0010	ADDI4096	ADDI4096	
0011	ADDI8192	ADDI8192	
0012	ADDI16384	ADDI16384	
0013	ADDI32768	ADDI32768	
0014	ADDI65536	ADDI65536	
0015	ADDI131072	ADDI131072	
0016	ADDI262144	ADDI262144	
0017	ADDI524288	ADDI524288	
0018	ADDI1048576	ADDI1048576	
0019	ADDI2097152	ADDI2097152	
0020	ADDI4194304	ADDI4194304	
0021	ADDI8388608	ADDI8388608	
0022	ADDI16777216	ADDI16777216	
0023	ADDI33554432	ADDI33554432	
0024	ADDI67108864	ADDI67108864	
0025	ADDI134217728	ADDI134217728	
0026	ADDI268435456	ADDI268435456	
0027	ADDI536870912	ADDI536870912	
0028	ADDI1073741824	ADDI1073741824	
0029	ADDI2147483648	ADDI2147483648	
0030	ADDI4294967296	ADDI4294967296	
0031	ADDI8589934592	ADDI8589934592	
0032	ADDI17179869184	ADDI17179869184	
0033	ADDI34359738368	ADDI34359738368	
0034	ADDI68719476736	ADDI68719476736	
0035	ADDI137438953472	ADDI137438953472	
0036	ADDI274877906944	ADDI274877906944	
0037	ADDI549755813888	ADDI549755813888	
0038	ADDI1099511627776	ADDI1099511627776	
0039	ADDI2199023255552	ADDI2199023255552	
0040	ADDI4398046511104	ADDI4398046511104	
0041	ADDI8796093022208	ADDI8796093022208	
0042	ADDI17592186444416	ADDI17592186444416	
0043	ADDI35184372888832	ADDI35184372888832	
0044	ADDI70368745777664	ADDI70368745777664	
0045	ADDI140737491553280	ADDI140737491553280	
0046	ADDI281474983106560	ADDI281474983106560	
0047	ADDI562949966213120	ADDI562949966213120	
0048	ADDI1125899932266240	ADDI1125899932266240	
0049	ADDI2251799864532480	ADDI2251799864532480	
0050	ADDI4503599729064960	ADDI4503599729064960	
0051	ADDI9007199458129920	ADDI9007199458129920	
0052	ADDI18014398916259840	ADDI18014398916259840	
0053	ADDI36028797832519680	ADDI36028797832519680	
0054	ADDI72057595665039360	ADDI72057595665039360	
0055	ADDI144115191330078720	ADDI144115191330078720	
0056	ADDI288230382660157440	ADDI288230382660157440	
0057	ADDI576460765320314880	ADDI576460765320314880	
0058	ADDI1152921530640289600	ADDI1152921530640289600	
0059	ADDI2305843061280579200	ADDI2305843061280579200	
0060	ADDI4611686122561158400	ADDI4611686122561158400	
0061	ADDI9223372245122316800	ADDI9223372245122316800	
0062	ADDI18446744490244273600	ADDI18446744490244273600	
0063	ADDI36893488980488547200	ADDI36893488980488547200	
0064	ADDI73786977960977094400	ADDI73786977960977094400	
0065	ADDI147573959921954188800	ADDI147573959921954188800	
0066	ADDI295147919843908377600	ADDI295147919843908377600	
0067	ADDI590295839687816755200	ADDI590295839687816755200	
0068	ADDI1180591679375633510400	ADDI1180591679375633510400	
0069	ADDI2361183358751267020800	ADDI2361183358751267020800	
0070	ADDI4722366717502534041600	ADDI4722366717502534041600	
0071	ADDI9444733435005068083200	ADDI9444733435005068083200	
0072	ADDI18889466870010136166400	ADDI18889466870010136166400	
0073	ADDI37778933740020272332800	ADDI37778933740020272332800	
0074	ADDI75557867480040544665600	ADDI75557867480040544665600	
0075	ADDI151115734960081093312000	ADDI151115734960081093312000	
0076	ADDI302231469920162186624000	ADDI302231469920162186624000	
0077	ADDI604462939840324373248000	ADDI604462939840324373248000	
0078	ADDI120892579968064874656000	ADDI120892579968064874656000	
0079	ADDI2417851599360129751328000	ADDI2417851599360129751328000	
0080	ADDI4835703198720259402656000	ADDI4835703198720259402656000	
0081	ADDI96714063974405188153120000	ADDI96714063974405188153120000	
0082	ADDI193428127948803573506240000	ADDI193428127948803573506240000	
0083	ADDI386856255897607147012480000	ADDI386856255897607147012480000	
0084	ADDI773712511795214344024960000	ADDI773712511795214344024960000	
0085	ADDI1547425023590428688049280000	ADDI1547425023590428688049280000	
0086	ADDI3094850047180857376098560000	ADDI3094850047180857376098560000	
0087	ADDI618970009436171475219712000000	ADDI618970009436171475219712000000	
0088	ADDI123794001872323430439424000000	ADDI123794001872323430439424000000	
0089	ADDI247588003744646860878848000000	ADDI247588003744646860878848000000	
0090	ADDI495176007489293721757696000000	ADDI495176007489293721757696000000	
0091	ADDI9903520149785874435153920000000	ADDI9903520149785874435153920000000	
0092	ADDI19807040319571752870318400000000	ADDI19807040319571752870318400000000	
0093	ADDI39614080639143505740636800000000	ADDI39614080639143505740636800000000	
0094	ADDI79228161278287011481212736000000000	ADDI79228161278287011481212736000000000	
0095	ADDI158456322565740222824244672000000000	ADDI158456322565740222824244672000000000	
0096	ADDI3169126451314804444448489440000000000	ADDI3169126451314804444448489440000000000	
0097	ADDI63382529026296088888969788800000000000	ADDI63382529026296088888969788800000000000	
0098	ADDI1267650580525921777779395776000000000000	ADDI1267650580525921777779395776000000000000	
0099	ADDI25353011610518435555587915520000000000000	ADDI25353011610518435555587915520000000000000	
0100	ADDI507060232210368711111158310400000000000000	ADDI507060232210368711111158310400000000000000	

- 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 Reference Data		ARITHMETIC CORE INSTRUCTION SET	
RV64M Integer Instructions	RV64M Floating-Point Extensions	Mnemonic	FMNEMONIC
add	addw	add	addw
addi	addiw	addi	addiw
addi16	addi16w	addi16	addi16w
addi32	addi32w	addi32	addi32w
addi64	addi64w	addi64	addi64w
addi128	addi128w	addi128	addi128w
addi256	addi256w	addi256	addi256w
addi512	addi512w	addi512	addi512w
addi1024	addi1024w	addi1024	addi1024w
addi2048	addi2048w	addi2048	addi2048w
addi4096	addi4096w	addi4096	addi4096w
addi8192	addi8192w	addi8192	addi8192w
addi16384	addi16384w	addi16384	addi16384w
addi32768	addi32768w	addi32768	addi32768w
addi65536	addi65536w	addi65536	addi65536w
addi131072	addi131072w	addi131072	addi131072w
addi262144	addi262144w	addi262144	addi262144w
addi524288	addi524288w	addi524288	addi524288w
addi1048576	addi1048576w	addi1048576	addi1048576w
addi2097152	addi2097152w	addi2097152	addi2097152w
addi4194304	addi4194304w	addi4194304	addi4194304w
addi8388608	addi8388608w	addi8388608	addi8388608w
addi16777216	addi16777216w	addi16777216	addi16777216w
addi33554432	addi33554432w	addi33554432	addi33554432w
addi67108864	addi67108864w	addi67108864	addi67108864w
addi134217728	addi134217728w	addi134217728	addi134217728w
addi268435456	addi268435456w	addi268435456	addi268435456w
addi536870912	addi536870912w	addi536870912	addi536870912w
addi1073741824	addi1073741824w	addi1073741824	addi1073741824w
addi2147483648	addi2147483648w	addi2147483648	addi2147483648w
addi4294967296	addi4294967296w	addi4294967296	addi4294967296w
addi8589934592	addi8589934592w	addi8589934592	addi8589934592w
addi17179869184	addi17179869184w	addi17179869184	addi17179869184w
addi34359738368	addi34359738368w	addi34359738368	addi34359738368w
addi68719476736	addi68719476736w	addi68719476736	addi68719476736w
addi137438953472	addi137438953472w	addi137438953472	addi137438953472w
addi274877906944	addi274877906944w	addi274877906944	addi274877906944w
addi549755813888	addi549755813888w	addi549755813888	addi549755813888w
addi1099511627776	addi1099511627776w	addi1099511627776	addi1099511627776w
addi2199023255552	addi2199023255552w	addi2199023255552	addi2199023255552w
addi4398046511104	addi4398046511104w	addi4398046511104	addi4398046511104w
addi8796093022208	addi8796093022208w	addi8796093022208	addi8796093022208w
addi17592186444416	addi17592186444416w	addi17592186444416	addi17592186444416w
addi35184372888832	addi35184372888832w	addi35184372888832	addi35184372888832w
addi70368745777664	addi70368745777664w	addi70368745777664	addi70368745777664w
addi140737491553280	addi140737491553280w	addi140737491553280	addi140737491553280w
addi281474983106560	addi281474983106560w	addi281474983106560	addi281474983106560w
addi562949966213120	addi562949966213120w	addi562949966213120	addi562949966213120w
addi1125899932266240	addi1125899932266240w	addi1125899932266240	addi1125899932266240w
addi2251799864532480	addi2251799864532480w	addi2251799864532480	addi2251799864532480w
addi4503599729064960	addi4503599729064960w	addi4503599729064960	addi4503599729064960w
addi9007199458129920	addi9007199458129920w	addi9007199458129920	addi9007199458129920w
addi18014398916259840	addi18014398916259840w	addi18014398916259840	addi18014398916259840w
addi36028797832519680	addi36028797832519680w	addi36028797832519680	addi36028797832519680w
addi72057595665039360	addi72057595665039360w	addi72057595665039360	addi72057595665039360w
addi144115191330078720	addi144115191330078720w	addi144115191330078720	addi144115191330078720w
addi288230382660157440	addi288230382660157440w	addi288230382660157440	addi288230382660157440w
addi576460765320314880	addi576460765320314880w	addi576460765320314880	addi576460765320314880w
addi115292153064028960	addi115292153064028960w	addi115292153064028960	addi115292153064028960w
addi230584306128057920	addi230584306128057920w	addi230584306128057920	addi230584306128057920w
addi461168612256115840	addi461168612256115840w	addi461168612256115840	addi461168612256115840w
addi922337224512231680	addi922337224512231680w	addi922337224512231680	addi922337224512231680w
addi1844674449024427360	addi1844674449024427360w	addi1844674449024427360	addi1844674449024427360w

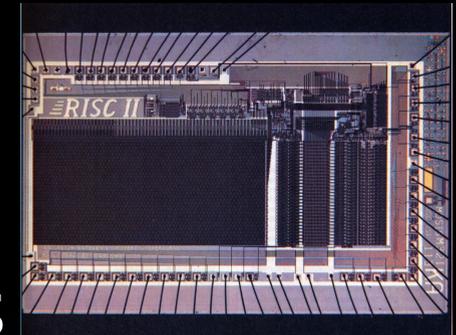


# RISC-V Origins

- 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](http://www.riscv.org))
- Many commercial and research projects based on RISC-V, open-source and proprietary
  - Widely used in education
- Read more:



RISC-I



RISC-II

- <https://riscv.org/risc-v-history/>
- <https://riscv.org/risc-v-genealogy/>



# Elements of Architecture: Registers

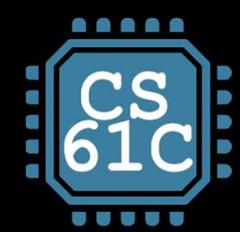
Preliminary discussion of the logical design of an electronic computing instrument<sup>1</sup>

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

“instruction sets”

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.

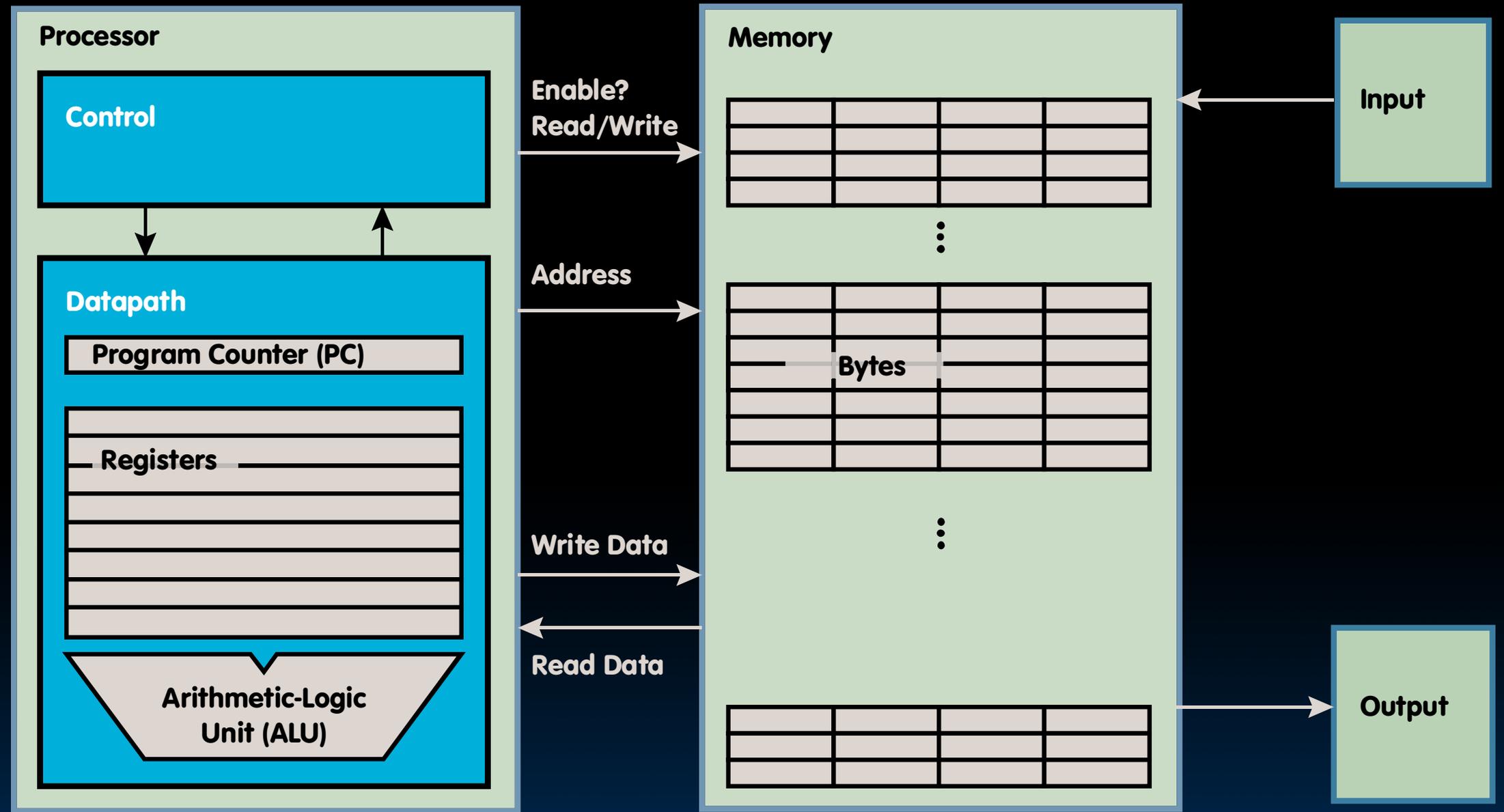
- Instruction set for a particular architecture (e.g. RISC-V) is represented by the Assembly language
- Each line of assembly code represents one instruction for the computer



# Assembly Variables: Registers (1/3)

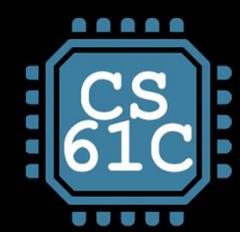
- Unlike HLL like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Assembly operands are registers
  - Limited number of special locations built directly into the hardware
  - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they're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

# Aside: Registers are Inside the Processor

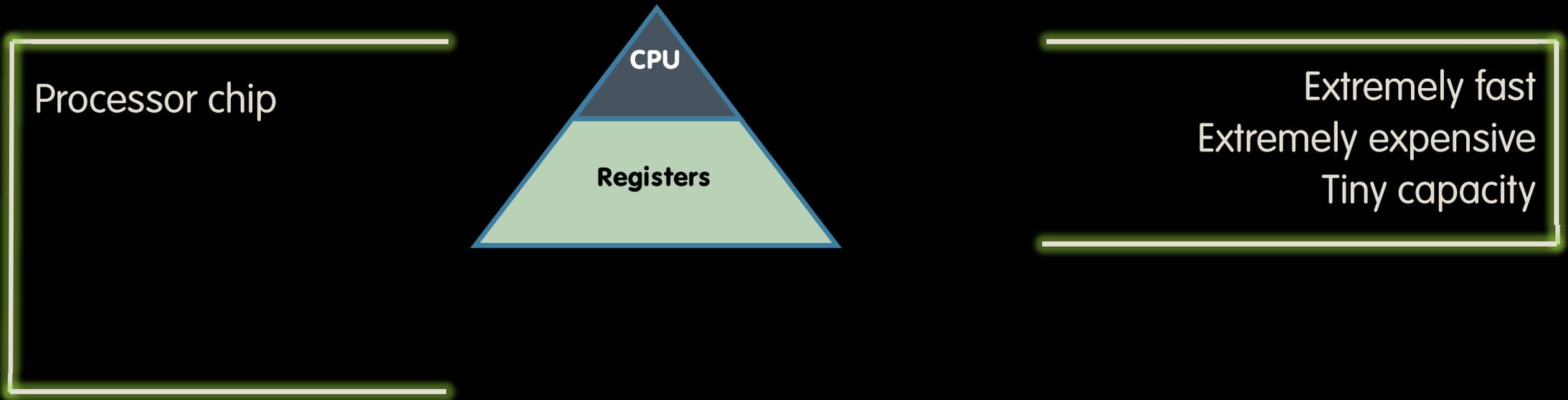


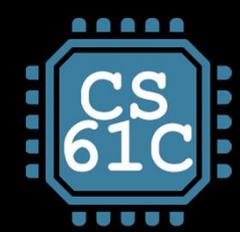
Processor-Memory Interface

I/O-Memory Interfaces

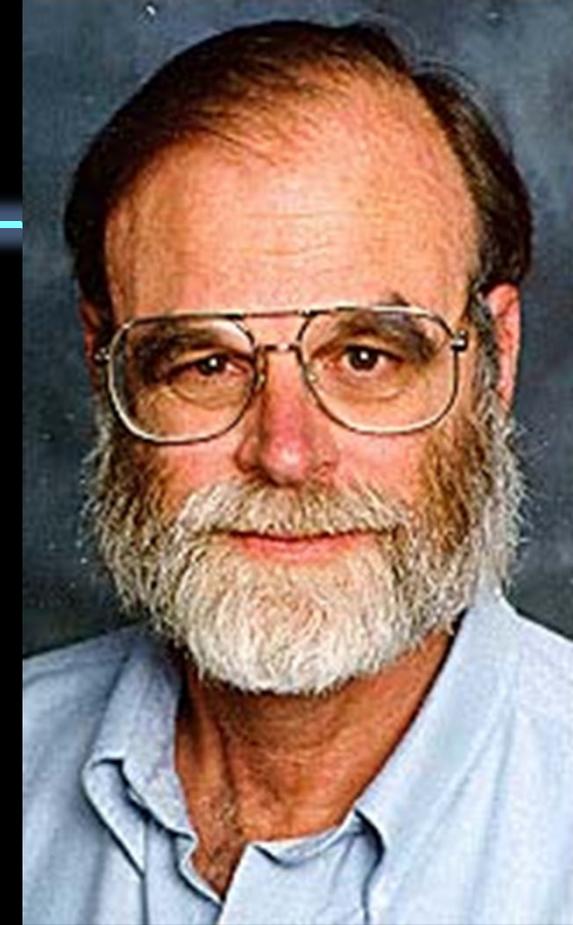


# Great Idea #3: Principle of Locality / Memory Hierarchy





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

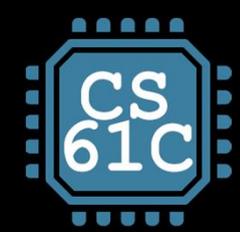


**Jim Gray**  
Turing Award  
B.S. Cal 1966  
Ph.D. Cal 1969



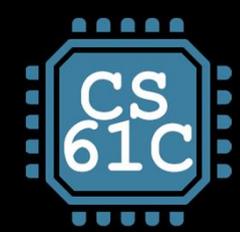
**1**  
[ns] **Registers**

 **My Head** **1 min**



# Assembly Variables: Registers (2/3)

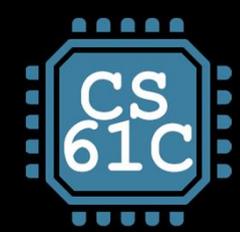
- Drawback: Since registers are in hardware, there is a predetermined number of them
  - Solution: RISC-V code must be very carefully put together to efficiently use registers
- 32 registers in RISC-V
  - Why 32?  
Smaller is faster, but too small is bad. Goldilocks principle ("This porridge is too hot; 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



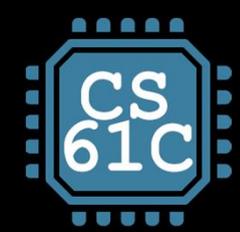
# C, Java variables vs. registers

---

- 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)



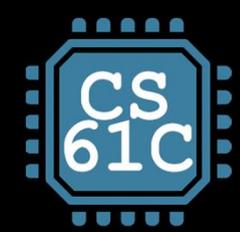
```

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
  
```

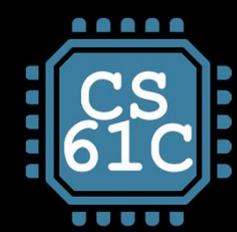
Assembly code with comments  
(ABC News, 2018)



# Assembly Instructions

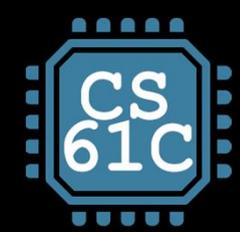
- In assembly language, each statement (called an **Instruction**), executes exactly one of a short list of simple commands
- Unlike in C (and most other high-level languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, \*, /) in C or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



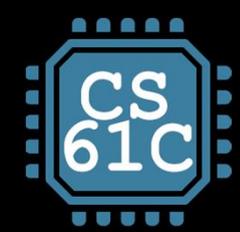
# Addition and Subtraction of Integers (2/4)

- Addition in Assembly

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

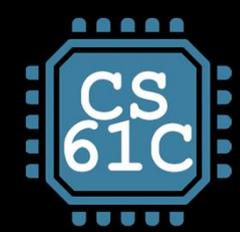
```
add x6, x22, x23 # b_temp = i + j
```

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

- A good compiler may do:

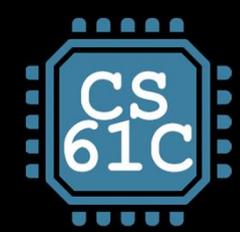
# RISC-V

## Immediates



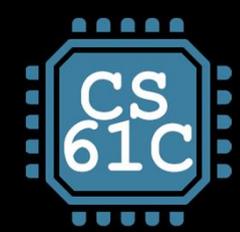
# Immediates

- 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!

# Storing Data in Memory



# RV32 So Far...

- Addition/subtraction

**add rd, rs1, rs2**

$$R[rd] = R[rs1] + R[rs2]$$

**sub rd, rs1, rs2**

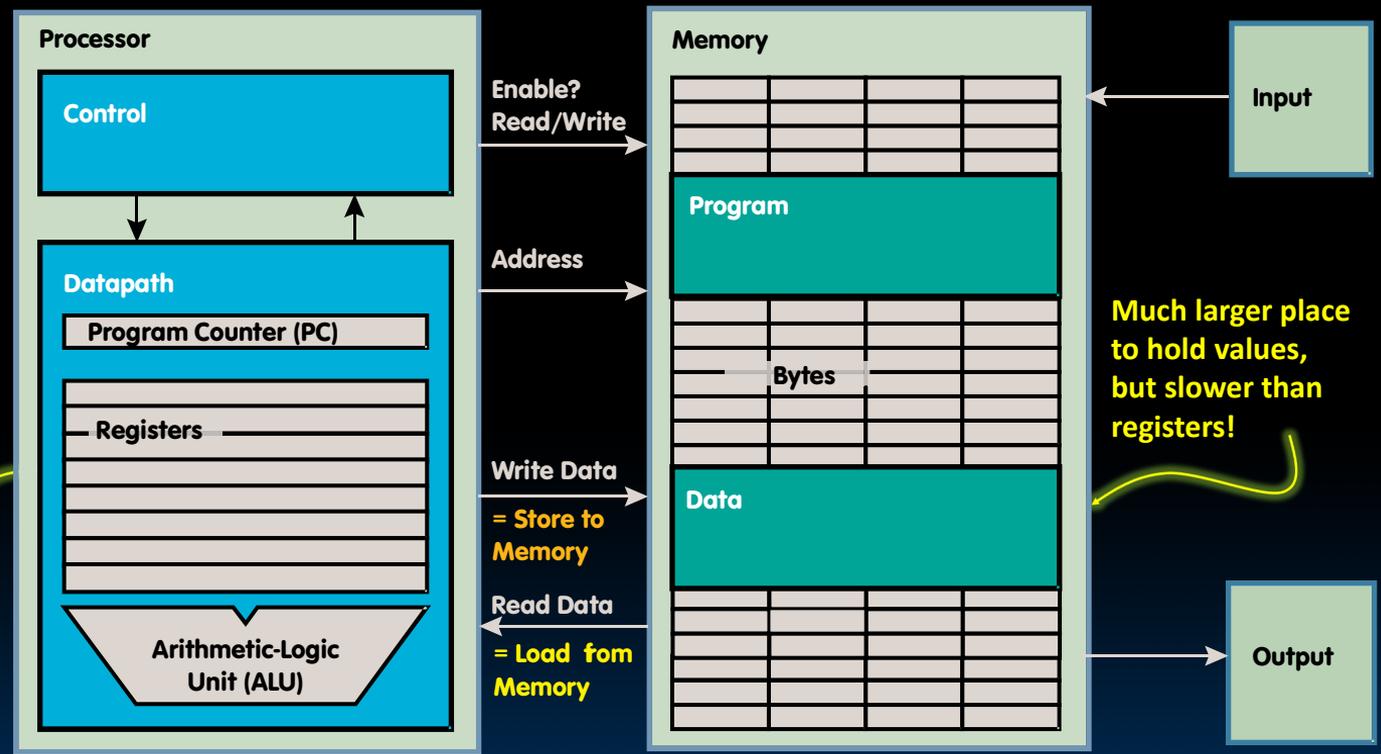
$$R[rd] = R[rs1] - R[rs2]$$

- Add immediate

**addi rd, rs1, imm**

$$R[rd] = R[rs1] + imm$$

# Data Transfer: **Load from** and **Store to** memory



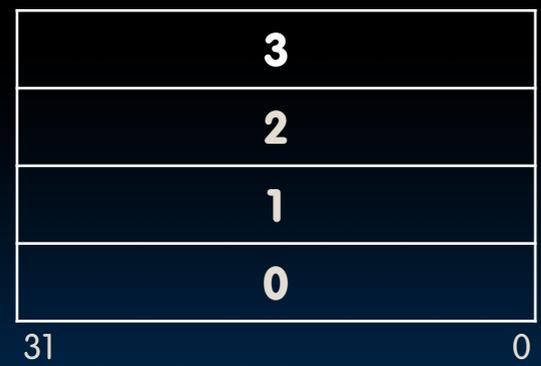
Very fast,  
but limited space to hold values!

Much larger place  
to hold values,  
but slower than  
registers!



# Memory Addresses are in Bytes

- Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type)—works fine if everything is a multiple of 8 bits
- 8 bit chunk is called a **byte** (1 word = 4 bytes)
- Memory addresses are really in **bytes**, not words
- Word addresses are 4 bytes apart
  - Word address is same as address of rightmost byte – least-significant byte (i.e. **Little-endian** convention)



# Memory Addresses are in Bytes

- Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type)—works fine if everything is a multiple of 8 bits
- 8 bit chunk is called a **byte** (1 word = 4 bytes)
- Memory addresses are really in **bytes**, not words
- Word addresses are 4 bytes apart
  - Word address is same as address of rightmost byte – least-significant byte (i.e. **Little-endian** convention)

Least-significant byte  
in a word  
↓

<b>15</b>	<b>14</b>	<b>13</b>	<b>12</b>				
<b>11</b>	<b>10</b>	<b>9</b>	<b>8</b>				
<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>				
<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>				
31	24	23	16	15	8	7	0

Least-significant byte  
gets the smallest address



# Big Endian vs. Little Endian

The adjective endian has its origin in the writings of 18th century writer Jonathan Swift. In the 1726 novel Gulliver's Travels, he portrays the conflict between sects of Lilliputians divided into those breaking the shell of a boiled egg from the big end or from the little end. He called them the "Big-Endians" and the "Little-Endians".

- The order in which BYTES are stored in memory
- Bits always stored as usual (E.g., 0xC2=0b 1100 0010)

Consider the number 1025 as we typically write it:

BYTE3    BYTE2    BYTE1    BYTE0  
00000000 00000000 00000100 00000001

## Big Endian

ADDR3	ADDR2	ADDR1	ADDR0
BYTE0	BYTE1	BYTE2	BYTE3
00000001	00000100	00000000	00000000

## Examples

- Names in China or Hungary (e.g., Nikolić Bora)
- Java Packages: (e.g., org.mypackage.HelloWorld)
- Dates in ISO 8601 YYYY-MM-DD (e.g., 2020-09-07)
- Eating Pizza crust first

## Little Endian

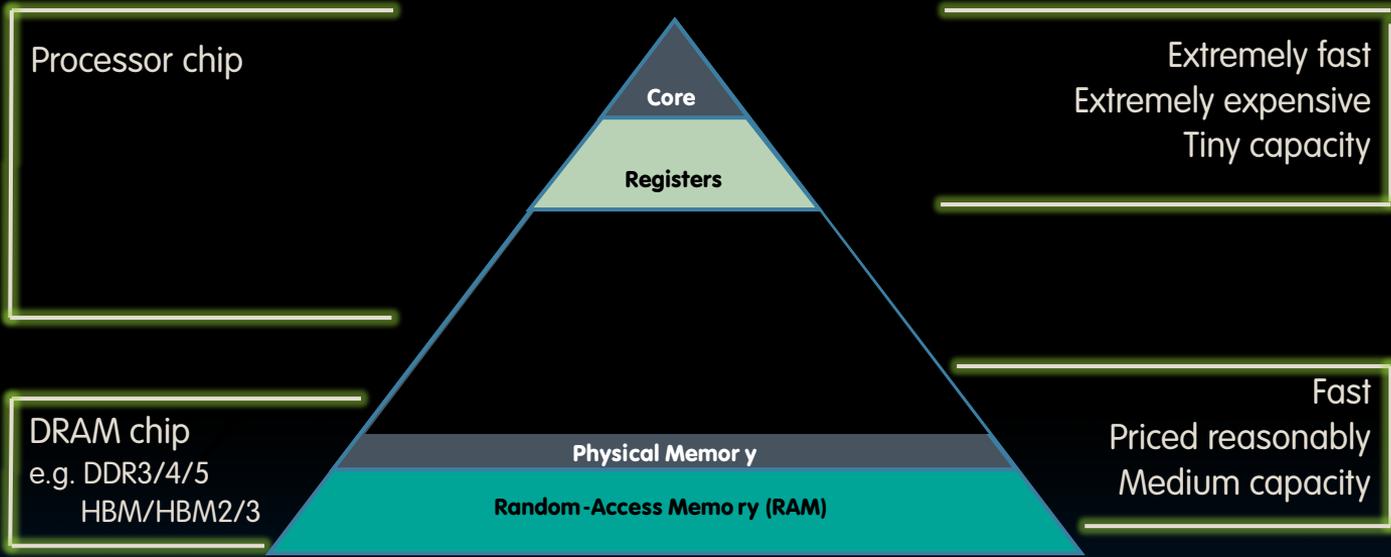
ADDR3	ADDR2	ADDR1	ADDR0
BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

## Examples

- Names in the US (e.g., Bora Nikolić)
- Internet names (e.g., cs.berkeley.edu)
- Dates written in Europe DD/MM/YYYY (e.g., 07/09/2020)
- Eating Pizza skinny part first

# Data Transfer Instructions

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





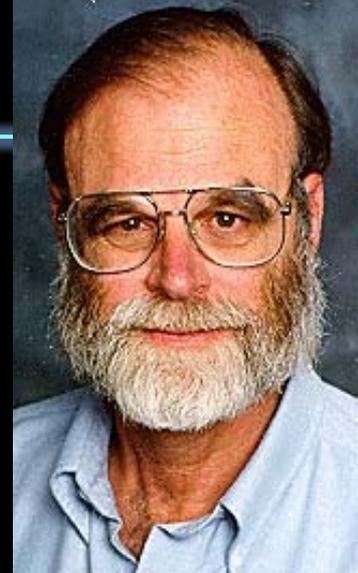
# Speed of Registers vs. Memory

---

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory (DRAM): Billions of bytes (2 GB to 64 GB on laptop)
- and physics dictates...
  - Smaller is faster
- How much faster are registers than DRAM??
  - About 50-500 times faster! (in terms of **latency** of one access - tens of ns)
    - But subsequent words come every few ns



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



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



# Load from Memory to Register

- C code

```
int A[100];
g = h + A[3];
```



- Using Load Word (`lw`) in RISC-V:

```
lw x10,12(x15) # Reg x10 gets A[3]
add x11,x12,x10 # g = h + A[3]
```

Note: x15 – base register (pointer to A[0])  
 12 – offset in bytes

**Offset must be a constant known at assembly time**

# Store from Register to Memory

- C code

```
int A[100];
A[10] = h + A[3];
```

- Using Store Word (**sw**) in RISC-V:

```
lw x10,12(x15) # Temp reg x10 gets A[3]
add x10,x12,x10 # Temp reg x10 gets h + A[3]
sw x10,40(x15) # A[10] = h + A[3]
```



Note: x15 – base register (pointer)  
 12,40 – offsets in bytes  
 x15+12 and x15+40 must be multiples of 4

# Loading and Storing Bytes

- In addition to word data transfers (**lw**, **sw**), RISC-V has **byte** data transfers:
  - load byte: **lb**
  - store byte: **sb**
- Same format as **lw**, **sw**
- E.g., **lb x10, 3(x11)**
  - contents of memory location with address = sum of "3" + contents of register **x11** is copied to the low byte position of register **x10**.

RISC-V also has "unsigned byte" loads (**lbu**) which zero extends to fill register. Why no unsigned store byte '**sbu**'?



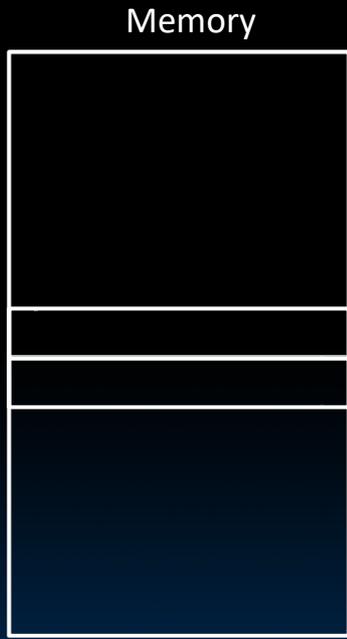


# Example: What is in x12 ?

```
addi x11, x0, 0x3F5
```

```
sw x11, 0(x5)
```

```
lb x12, 1(x5)
```





# Substituting addi

The following two instructions:

```
lw x10,12(x15) # Temp reg x10 gets A[3]
add x12,x12,x10 # reg x12 = reg x12 + A[3]
```

Replace addi:

```
addi x12, value # value in A[3]
```

But involve a load from memory!

Add immediate is so common that it deserves its own instruction!

# Decision Making



# RV32 So Far...

---

- Addition/subtraction

```
add rd, rs1, rs2
```

```
sub rd, rs1, rs2
```

- Add immediate

```
addi rd, rs1, imm
```

- Load/store

```
lw rd, rs1, imm
```

```
lb rd, rs1, imm
```

```
lbu rd, rs1, imm
```

```
sw rs1, rs2, imm
```

```
sb rs1, rs2, imm
```

# Computer Decision Making

- Based on computation, do something different
- In programming languages: *if*-statement
  
- RISC-V: *if*-statement instruction is
  - beq reg1 , reg2 , L1**
  - means: go to statement labeled L1  
if (value in reg1) == (value in reg2)  
...otherwise, go to next statement
  - **beq** stands for *branch if equal*
  - Other instruction: **bne** for *branch if not equal*

# Types of Branches

- Branch – change of control flow
  
- Conditional Branch – change control flow depending on outcome of comparison
  - branch *if* equal (**beq**) or branch *if not* equal (**bne**)
  - Also branch if less than (**blt**) and branch if greater than or equal (**bge**)
  - And unsigned versions (**bltu**, **bgeu**)
  
- Unconditional Branch – always branch
  - a RISC-V instruction for this: *jump* (**j**), as in **j label**

# Example *if* Statement

- Assuming translations below, compile *if* block

$f \rightarrow x10$                        $g \rightarrow x11$                        $h \rightarrow x12$   
 $i \rightarrow x13$                                $j \rightarrow x14$

```

if (i == j)                              bne x13,x14,Exit
    f = g + h;                            add x10,x11,x12

```

**Exit:**

- May need to negate branch condition

# Example *if-else* Statement

- Assuming translations below, compile

$f \rightarrow x10$

$g \rightarrow x11$

$h \rightarrow x12$

$i \rightarrow x13$

$j \rightarrow x14$

```

if (i == j)                bne x13,x14,Else
    f = g + h;             add x10,x11,x12
else                        j Exit
    f = g - h; Else:sub x10,x11,x12
                          Exit:

```



# Magnitude Compares in RISC-V

- General programs need to test < and > as well.
- RISC-V magnitude-compare branches:

“Branch on Less Than”

Syntax: `blt reg1,reg2, Label`

Meaning: `if (reg1 < reg2) goto Label;`

“Branch on Less Than Unsigned”

Syntax: `bltu reg1,reg2, Label`

Meaning: `if (reg1 < reg2) // treat registers  
as unsigned integers  
goto label;`

Also “Branch on Greater or Equal” `bge` and `bgeu`

Note: No `bgt` or `ble` instructions





# Loops in C/Assembly

---

- There are three types of loops in C:
  - **while**
  - **do ... while**
  - **for**
- Each can be rewritten as either of the other two, so the same branching method can be applied to these loops as well.
- Key concept: Though there are multiple ways of writing a loop in RISC-V, the key to decision-making is conditional branch



# C Loop Mapped to RISC-V Assembly

```
int A[20];  
int sum = 0;  
for (int i=0; i < 20; i++)  
    sum += A[i];
```

```
add x9, x8, x0 # x9=&A[0]  
add x10, x0, x0 # sum  
add x11, x0, x0 # i  
addi x13,x0, 20 # x13
```

Loop:

```
bge x11,x13,Done  
lw x12, 0(x9) # x12 A[i]  
add x10,x10,x12 # sum  
addi x9, x9,4 # &A[i+1]  
addi x11,x11,1 # i++  
j Loop
```

Done: