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

Lecture 6: RISC-V Instructions

Instructors: Rosalie Fang, **Charles Hong**, Jero Wang

Announcements

- Labs 1 and 2: due Thursday, June 29th, 11:59 PM PT
- Homework 1: due today, June 28th, 11:59 PM PT
  - Worth twice as much as a normal homework (6 course points as opposed to 3).
- Homework 2 will be released this week and is due Wednesday, July 5th, 11:59 PM PT
  - Short, only lecture 5 (FP) concepts
- Project 1: Due Friday, June 30th, 11:59 PM PT
- Labs moved from Soda 271 to Soda 330 (more details to come)

Last Time

- Floating Point

This Time (and moving forward)

- Lecture 6 (Today)
  - Intro to Assembly Languages, RISC-V
- Lecture 7 (Tomorrow)
  - How to write RISC-V code
- Lecture 8 (Next Wednesday)
  - Translating RISC-V to binary

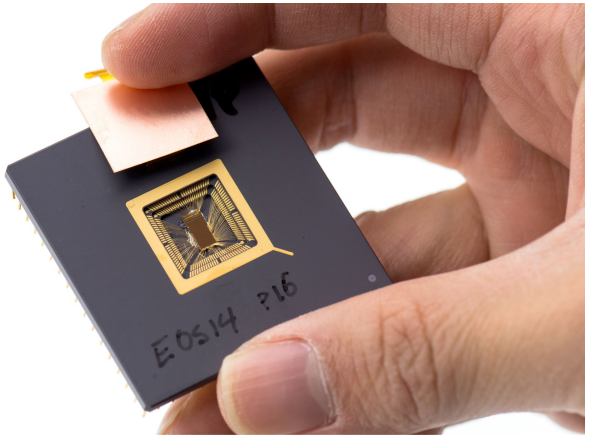
Intro to Assembly Languages

CPUs

How do we run this code:

```c
#define SPOCK 1701
int KIRK = 1701;
int sulu(int scotty) {
    return scotty * scotty;
}
int main(int argc, char *argv[])  {
    int *chekov = malloc(sizeof(int) * 1701);
    if (chekov) free(chekov);
    sulu(SPOCK); // ← snapshot just before it returns
    return 0;
}
```

On this piece of metal?



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

CS61C

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)

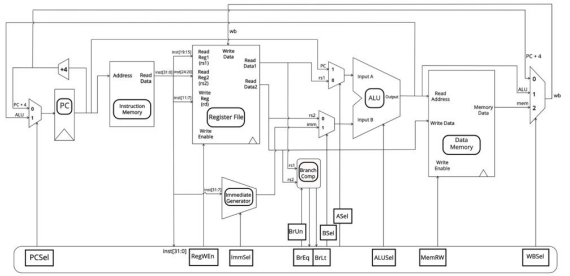
```
lw   x3, 0(x10)
lw   x4, 4(x10)
sw   x4, 0(x10)
sw   x3, 4(x10)
```

Assembler

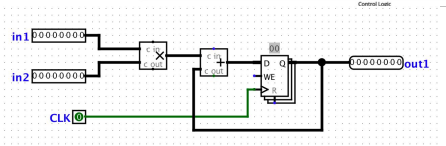
Machine Language Program (RISC-V)

```
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)



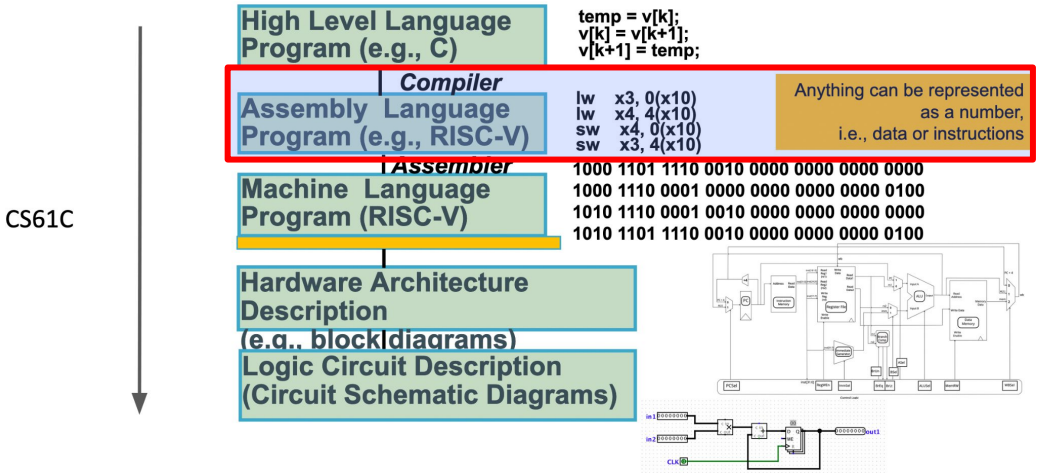
Logic Circuit Description (Circuit Schematic Diagrams)



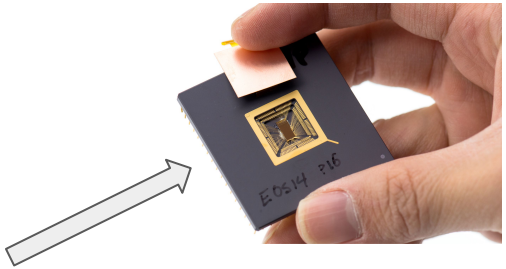
Anything can be represented as a number, i.e., data or instructions

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

- Computers generally consist of binary circuits
- Directly programming circuits to run complex operations would be very difficult
- Instead, expose higher levels of abstraction to the user



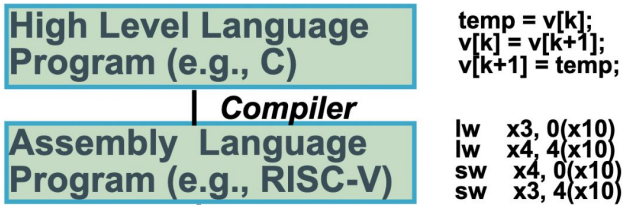
Assembly Language (1/2)



- It’s hard to change circuits after building them
- Instead, build a circuit that can run a set of building block operations that when combined, can describe almost any behavior - **central processing unit (CPU)**
- Different CPUs implement different sets of operations (AKA instructions) — called Instruction Set Architecture (ISA)
- The programming language defined by the ISA is known as an assembly language.
- ISA Examples?
  - ARM (cell phones), Intel x86 (i9, i7, i5, i3), IBM/Motorola PowerPC (old Macs), MIPS, RISC-V, ...

arm intel AMD RISC-V®

Assembly Language (2/2)



- C is a “lower-level” language than Java or Python, because it is closer to assembly language
- Still more complex and automated than most (all?) ISAs. How?
  - 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 call a function, and use local variables that don’t overwrite existing ones.
  - C lets you name variables, and will keep track of that name, and even the type of variable that name refers to.
- With assembly languages, almost everything is explicitly handled 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 (speed often limited by the slowest instruction)

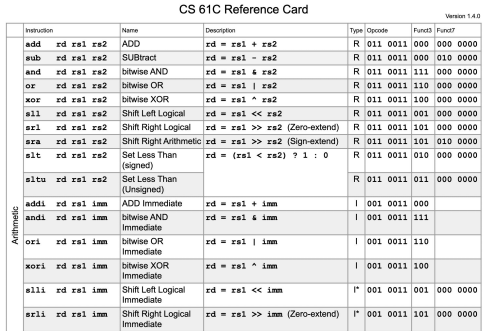
Turing Award winner (for RISC)!

RISC-V

- For the purposes of this class, we'll be learning RISC-V as our assembly language
- Why?
  - RISC-V is relatively simple.
  - There are only a few instructions in the base instruction set.
  - RISC-V instructions themselves follow a consistent format.
  - Project 3: Build a complete RISC-V CPU
  - x86, ARM are more popular languages (base CPU for most phones/laptops/desktops), but are more complex
  - 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/su22/pdfs/resources/reference-card.pdf
  - Lists out the entire base architecture
- Venus
  - https://venus.cs61c.org/
  - Online RISC-V simulator



CS 61C Reference Card

Version 1.4.0

	Instruction	Name	Description	Type	Opcode	Funct3	Funct7
Arithmetic	`add rd rs1 rs2`	ADD	`rd = rs1 + rs2`	R	011 0011	000	000 0000
	`sub rd rs1 rs2`	SUBtract	`rd = rs1 - rs2`	R	011 0011	000	010 0000
	`and rd rs1 rs2`	bitwise AND	`rd = rs1 & rs2`	R	011 0011	111	000 0000
	`or rd rs1 rs2`	bitwise OR	`rd = rs1 | rs2`	R	011 0011	110	000 0000
	`xor rd rs1 rs2`	bitwise XOR	`rd = rs1 ^ rs2`	R	011 0011	100	000 0000
	`sll rd rs1 rs2`	Shift Left Logical	`rd = rs1 << rs2`	R	011 0011	001	000 0000
	`srl rd rs1 rs2`	Shift Right Logical	`rd = rs1 >> rs2` (Zero-extend)	R	011 0011	101	000 0000
	`sra rd rs1 rs2`	Shift Right Arithmetic	`rd = rs1 >> rs2` (Sign-extend)	R	011 0011	101	010 0000
	`slt rd rs1 rs2`	Set Less Than (signed)	`rd = (rs1 < rs2) ? 1 : 0`	R	011 0011	010	000 0000
	`sltu rd rs1 rs2`	Set Less Than (Unsigned)		R	011 0011	011	000 0000
	`addi rd rs1 imm`	ADD Immediate	`rd = rs1 + imm`	I	001 0011	000	
	`andi rd rs1 imm`	bitwise AND Immediate	`rd = rs1 & imm`	I	001 0011	111	
	`ori rd rs1 imm`	bitwise OR Immediate	`rd = rs1 | imm`	I	001 0011	110	
	`xori rd rs1 imm`	bitwise XOR Immediate	`rd = rs1 ^ imm`	I	001 0011	100	
	`slli rd rs1 imm`	Shift Left Logical Immediate	`rd = rs1 << imm`	I*	001 0011	001	000 0000
	`srli rd rs1 imm`	Shift Right Logical Immediate	`rd = rs1 >> imm` (Zero-extend)	I*	001 0011	101	000 0000

RISC-V Registers

Overarching view of RISC-V

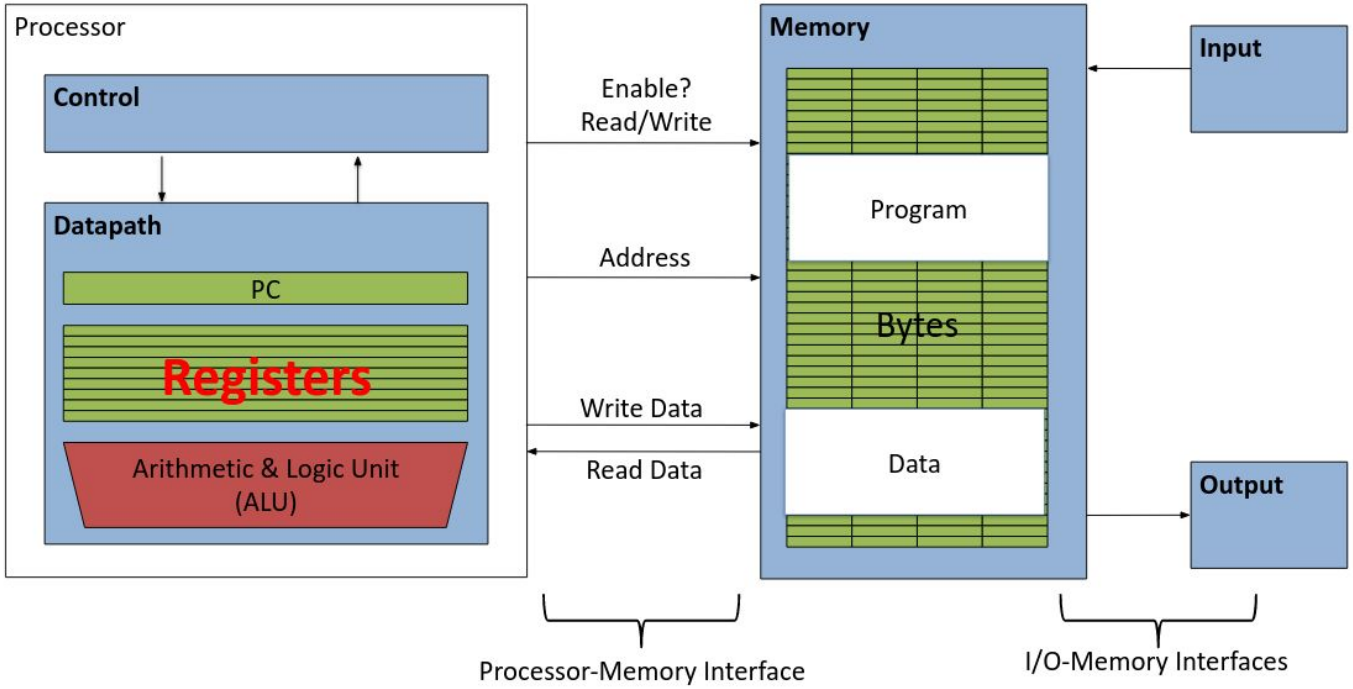
- A RISC-V system is composed of two main parts:
  - The CPU, which is responsible for computing
  - Main memory, which is responsible for long-term data storage
- The CPU is designed to be extremely fast, often completing multiple instructions every nanosecond
- Going to main memory often takes hundreds or even thousands of nanoseconds.
- The CPU can store a small amount of data, through components called registers.

Registers (1/2)

Reg	Name	Reg	Name	Reg	Name	Reg	Name
x0		x8		x16		x24	
x1		x9		x17		x25	
x2		x10		x18		x26	
x3		x11		x19		x27	
x4		x12		x20		x28	
x5		x13		x21		x29	
x6		x14		x22		x30	
x7		x15		x23		x31	

- A register is a CPU component 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). In this class, we use 32-bit only.
- RISC-V gives access to 32 integer registers
- The size and number of registers is fixed (you can think of them as being implemented in hardware)
    - Not entirely true – take CS 152 to learn more!

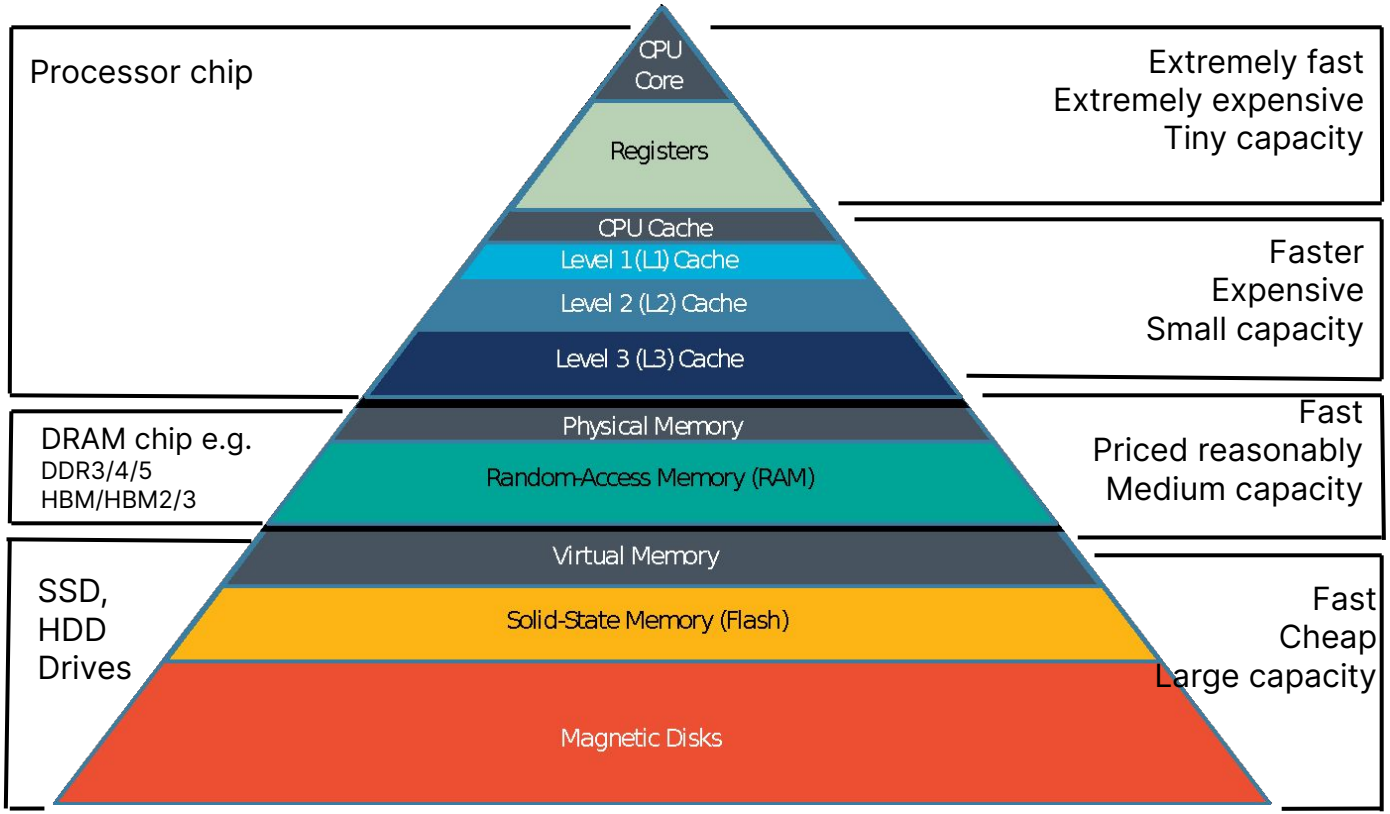
Aside: Registers are Inside the Processor



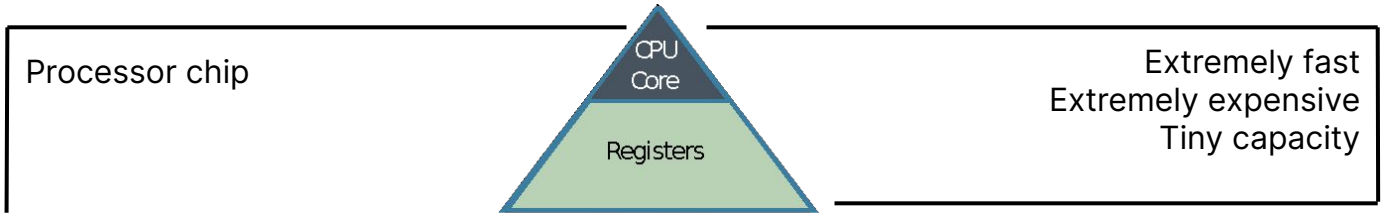
Accessing registers is very fast – faster than 0.25 ns!

Can be accessed in one “clock cycle” (to be covered) of a 4 GHz CPU.

Great Idea #3: Principles of Locality / Memory Hierarchy



Great Idea #3: Principles of Locality / Memory Hierarchy



Registers (2/2)

Reg	Name	Reg	Name	Reg	Name	Reg	Name
x0		x8		x16		x24	
x1		x9		x17		x25	
x2		x10		x18		x26	
x3		x11		x19		x27	
x4		x12		x20		x28	
x5		x13		x21		x29	
x6		x14		x22		x30	
x7		x15		x23		x31	

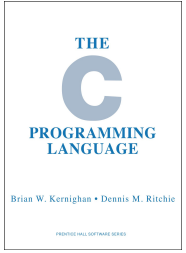
- RISC-V gives access to 32 integer registers
- Registers are numbered from 0 to 31
  - Referred to by number: x0 – x31
- The register x0 is special and always stores 0 (trying to write data to that register results in the write being ignored). As such, we have 31 registers available for data storage
- 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 how each register is used

Intro to RISC-V Instructions

Instructions

- Each line of RISC-V code is a single instruction, which executes a simple operation on registers.
- Arithmetic instruction syntax:

- Ex. “add x5 x6 x7” means “Add the values stored in x6 and x7, and store the result in x5”
  - Commas are optional



- Variables are declared and given a type

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

- **Variable types determine operation**, e.g. you (usually) do math with numerical types

int *p = ...;
p = p + 2;
int x = 42;
x = 3 * x;

RISC-V®

- Assembly operands are registers
- Registers have **no types** - they are just bits
- Registers cannot be given a name (comments are **very** important!)

- **Operation determines type**
  - Whether you treat bits as a value, memory address, etc.

Addition and Subtraction of Integers (1/3)

- Addition in Assembly:

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

Equivalent to: `a = b + c` (in C)

where `a` ⇔ `x1`, `b` ⇔ `x2`, `c` ⇔ `x3`

- Subtraction in Assembly

```
sub x3,x4,x5
```

Equivalent to: `d = e - f` (in C)

where `d` ⇔ `x3`, `e` ⇔ `x4`, `f` ⇔ `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
```

- 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 (3/3)

- How do we do this? Say **`g,h,i,j are in x20,x21,x22,x23`**

  `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: The data in destination registers is overwritten. So, generally keep a few registers free (i.e. not containing any important data) to help with intermediate calculations.

Pseudoinstructions

- Minimize # of instructions → minimize circuitry needed in CPU
- There are some commands which can be written using other instructions, but we want a shorthand for.
- “Pseudo” (false) instructions are created, which get turned into regular instructions in software.
- Ex. “**`mv x5 x6`**” is a pseudoinstruction that means “Set x5 to x6”
- How can we do this operation using normal instructions we already learned?

  `add x5 x6 x0`

Immediates

- Immediates are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:

```
addi x3,x4,10
```

  - `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, sub, and addi, but no subi
- 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), is useful for many instructions
- So the register zero (**`x0`**) is ‘hard-wired’ to value 0; e.g.

`add x3,x4,x0`

is equivalent to **`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!

RISC-V Instruction Types

Instruction types

- Instructions fall in 5 categories:
- Arithmetic Operators

Arithmetic Operators

- For doing math between registers or between a register and an immediate.
- The following operations are available in the base RISC-V instruction set:
  - Addition/Subtraction: **add**, **sub**, **addi**
  - Bitwise: **and**, **or**, **xor**, **andi**, **ori**, **xori**
  - Shifts: **sll** (Shift Left Logical), **srl** (Shift Right Logical), **sra** (Shift Right Arithmetic), **slli**, **srli**, **srai**
  - Set Less Than: **slt**, **sltu**, **slti**, **sltiu**
- Note on Set Less Than:
  - **slt x5 x6 x7** means:
  - If the x6 < x7, then set x5 to 1. Otherwise, set x5 to 0
  - Comes in two versions:
    - slt (treat the operands as signed numbers)
    - sltu (treat the operands as unsigned numbers)
    - slti and sltiu are the same but with an immediate instead of x7.
- Note: Multiplication/Division is NOT within the base instruction set
  - Takes $O(n^2)$ circuit components, so it gets put in an optional extension instead

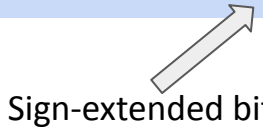
Arithmetic vs Logical Shifts (1/2)

- Logical Left shift: Move all bits to the left, appending zeros as needed
- Logical Right shift: Move all bits to the right, prepending zeros as needed
- Recall: For unsigned numbers, these are equivalent to multiplying by a power of 2 and dividing by a power of 2 (rounding down), respectively.
- What happens when we shift signed numbers? Is it different?
  - Shift left:
    - Multiplies by power of 2
  - Shift right:
    - Divides by power of 2?
    - What happens when you add 0 to the front of a negative two's complement integer?

Arithmetic vs Logical Shifts (2/2)

- **What happens when we shift signed numbers?**
- If we prepend 0s, we'll flip the sign of negative numbers.
- Solution: If you right-shift a negative number, prepend with 1s instead to keep the number negative. Example:

- `0b110101 >>> 2 -> 0b111101` `-11 -> -3`



Sign-extended bits!

- Conclusion: To right-shift (or extend) a signed number, fill in the extra bits with the MSB to keep the same represented value. This is known as "sign-extension."

Logical Shifting

- Shift Left Logical (**`sll`**) and immediate (**`slli`**):

```
slli x11,x12,2 #x11=x12<<2
```

- Store in x11 the value from x12 shifted by 2 bits to the left (they fall off end), inserting 0's on right.
- Equivalent to << in C.
- **Before:** 0d2
  x12 ⟶ 0b0000 0000 0000 0000 0000 0000 0000 00**10**
- **After:** 0d8
  x11 ⟶ 0b0000 0000 0000 0000 0000 0000 0000 **1000**
- Shift Right Logical (**`srl`**) and immediate (**`srli`**): Bits fall of right, prepend 0s on left.

Arithmetic Shifting

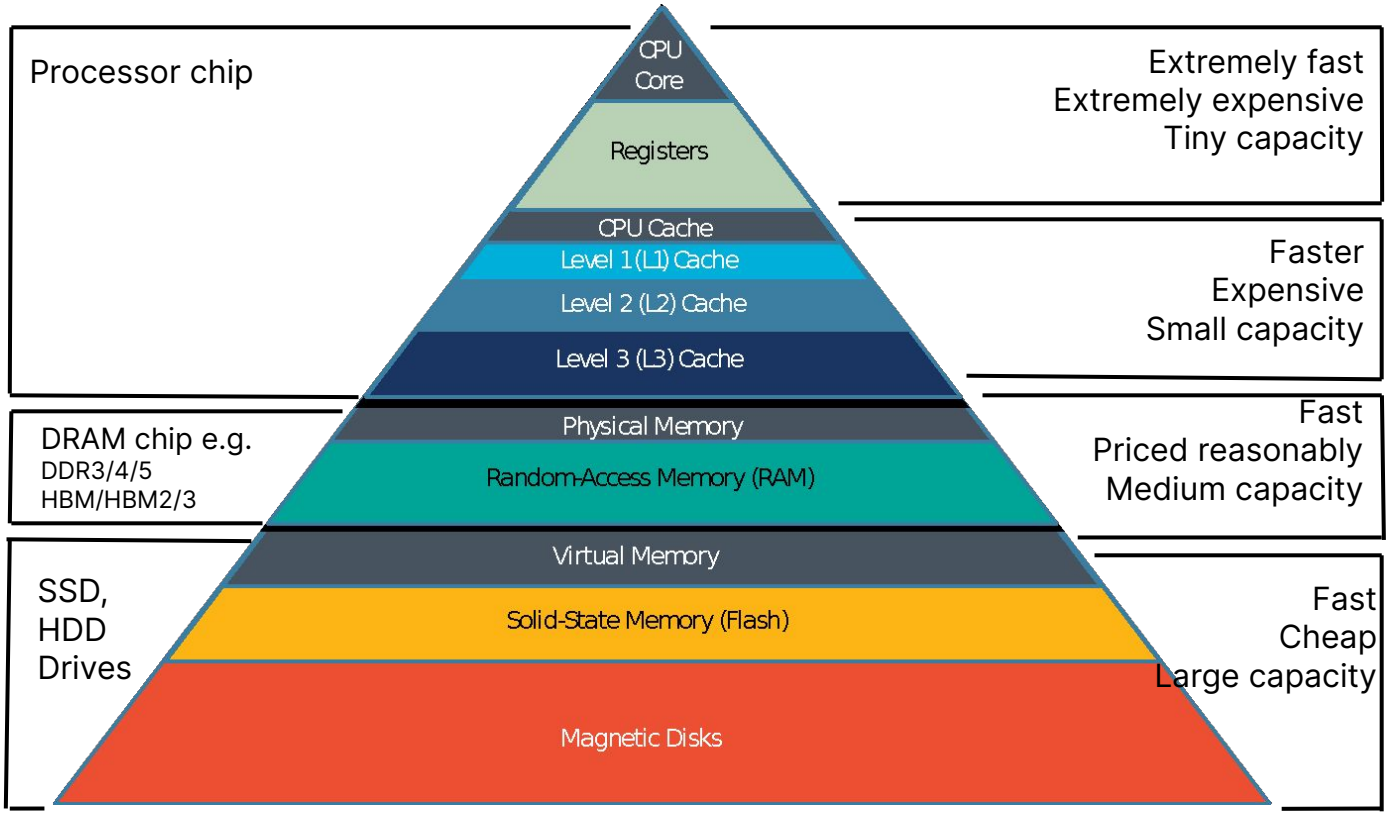
- Shift right arithmetic (sra, srai) moves n bits to the right, insert MSB into empty bits
- For example, if register x10 contained
  x10 ⟶ 0b1111 1111 1111 1111 1111 1111 111**0 0111** = -25

  `srai x10, x10, 4`
  x10 ⟶ 0b1111 1111 1111 1111 1111 1111 1111 111**0** = -2
- Different from C arithmetic
  - Rounds towards negative infinity, not 0
- Note: There's no "sla" because left shifts work the same regardless of if we use signed or unsigned numbers.

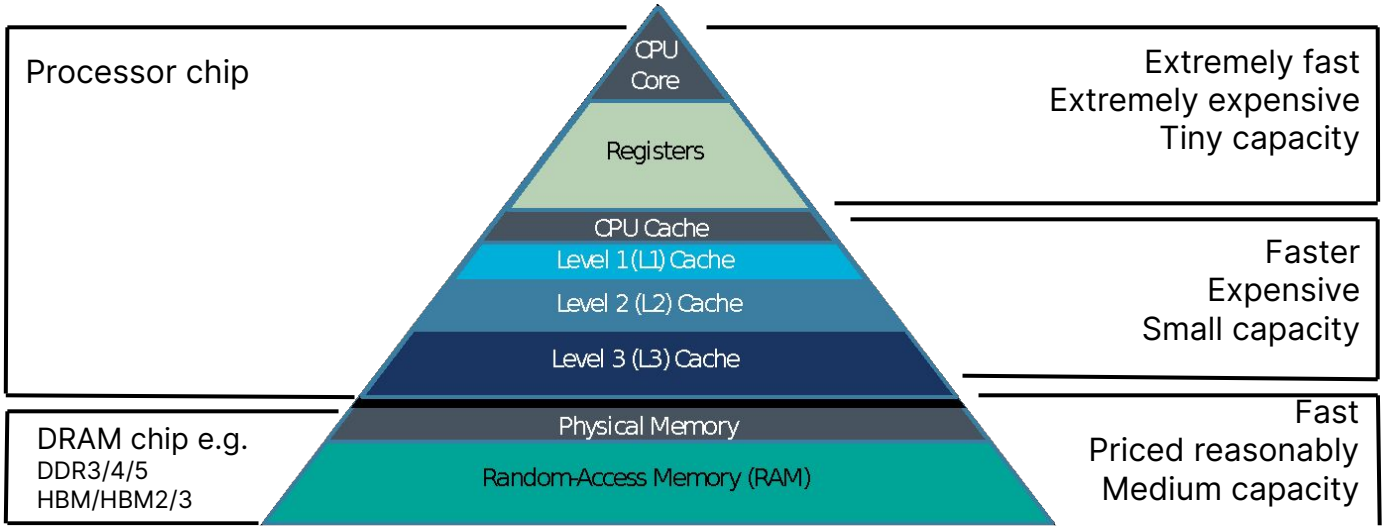
Instruction types

- Instructions fall in 5 categories:
- Arithmetic Operators
- Memory

Great Idea #3: Principles of Locality / Memory Hierarchy



Great Idea #3: Principles of Locality / Memory Hierarchy

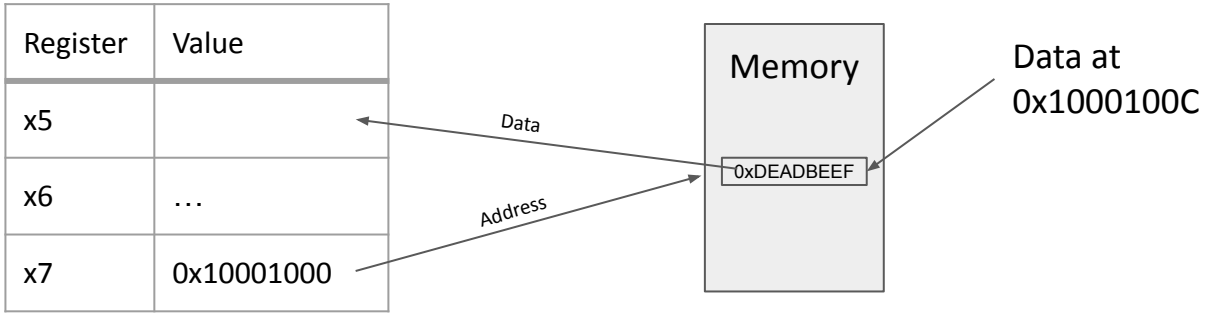


Memory Instructions

- Allows data to be transferred from main memory to registers and vice versa.
- Has a different syntax compared to arithmetic instructions
- Example:

```
lw x5 12(x7)
```

- Meaning: Get address in x7 and add 12. Retrieve the next word of data (4 bytes) from memory, and save the result in x5.

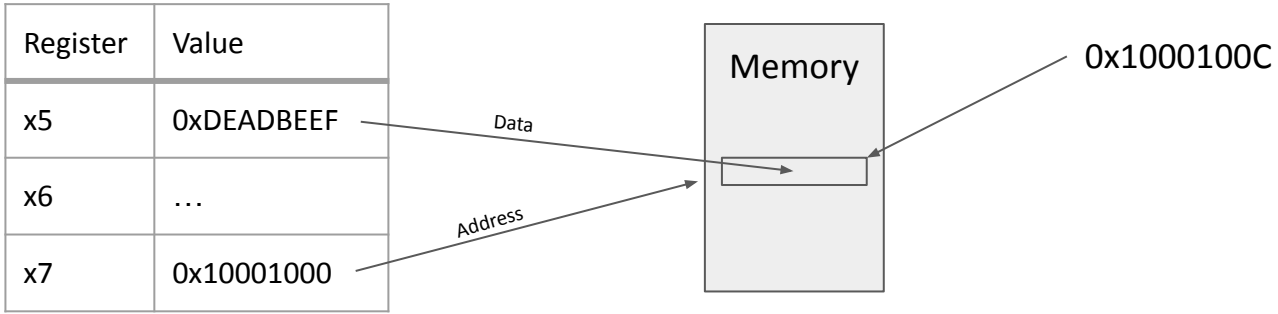


Memory Instructions

- Example:

```
sw x5 12(x7)
```

- Meaning: Interpret x7 as an address and add 12. Overwrite the 4 bytes of memory at that location with the value currently stored in x5.



RISC-V Example: Load from Memory to Register

- C:

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

- Setup:
  - x11 = g, x12 = h
  - x15 = address of A[0]
  - 12 = offset in bytes
- RISC-V:

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

RISC-V Example: Store from Register to Memory

- C:

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

- Setup:
  - x12 = h
  - x15 = address of A[0]
- 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]
```

- x15+12 and x15+40 should be multiples of 4, otherwise accesses are “unaligned”

Loading and Storing Bytes

- In addition to word data transfers (lw, sw), RISC-V has byte data transfers:
  - load byte: **lb**
  - store byte: **sb**
  - load/store half-word: **lh**, **sh**
- Same format as lw, sw
- Example:

```
lb x10,3(x11)
```

- Meaning: Interpret x11 as an address and add 3. Retrieve the next byte of data, and save the result in x10.
- RISC-V also has “unsigned byte” loads (lbu, lhu) which zero-extends to fill register (normal lb and lh sign-extends). Why no unsigned store byte sbu?

Your turn. What's in x12?

```
addi x11,x0,0x3F5

sw x11,0(x5)

lb x12,1(x5)
```

Instruction types

- Instructions fall in 5 categories:
- Arithmetic Operators
- Memory
- Control

Control

- Normally, when running RISC-V code, we always move to the next line of code
- Control instructions instead specify a different line of code to run next.
- Comes in two variants:

    1. Conditional jumps (also called branches), which jump to the specified line only if a certain condition is met, and otherwise move to the next line

        i.e., if statement

    2. Unconditional jumps, which jump to the specified line no matter what.

Labels

- When working with control instructions, we often need to specify a line of RISC-V code to jump to.
- Label: A human-readable identifier for a particular line of code.

Ex.

```
        addi x5 x0 0
        addi x6 x0 10
Loop:   add x5 x5 x6
        addi x5 x5 -1
        bne x5 x0 Loop
```

Types of Branches

- **beq**: Branch if the two registers have equal value
- **bne**: Branch if non-equal
- **blt**, **bge**, **bltu**, **bgeu**: Less than and “greater or equal”, along with unsigned variants.
- Note: bgt is not an instruction, because if we wanted to do bgt x5 x6 Label, we could do blt x6 x5 Label instead.

Unconditional Jumps

- Two versions:
  - jump to a label
  - jump to an address saved in a register.
- These instructions also reference the current Program Counter (or PC), which stores the address of the current line of code being run.
- Ex. **`jal x1 Label`** (Jump and Link)
  - Meaning: Set x1 to PC+4 (the line of code immediately after the current line), then jump to Label
- Ex. **`jalr x1 x5 0`** (Jump and Link Register)
  - Meaning: Set x1 to PC+4, then jump to the line of code at address x5+0
- Pseudoinstructions exist for when we don't need the link
  - "**`j Label`**" and "**`jr x1`**"

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=0
    add x11, x0, x0 # i=0
    addi x13, x0, 20 # x13=20
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:
```

Why We Link

- Ex. jal x1 Label (Jump and Link)
- Meaning: Set x1 to PC+4, then jump to Label
- Links allow us to “return” to where we were before we jumped
- Often used to mimic the behavior of functions:

```
main:  addi x10 x0 5
       jal x1 foo
       ...
foo:   ...
       jr x1
```

1. We jump to foo, and store the address of the next line into x1
2. Later, foo runs `jr x1`, returning to the point after the “function call.”

- This lets foo act as a function (though there are critical differences)
- More info tomorrow

Instruction types

- Instructions fall in 5 categories:
- Arithmetic Operators
- Memory
- Control
- Miscellaneous

Miscellaneous Instructions

- **lui**, **auipc**: “helpers” to let the pseudoinstructions “li” and “la” work. Rarely used independently.
  - **li x5 0xDEADBEEF**: Sets the register x5 to immediate value 0xDEADBEEF
  - **la x5 Label**: Sets x5 to the address of the line of code pointed to by Label
  - Because immediates are less than 32 bits – will make more sense when we talk about how instructions are encoded as bits

- **ebreak**: Debugger-specific behavior. In Venus, acts like a breakpoint.
- **ecall**: OS-specific behavior, such as printing out data, requesting memory via malloc. In this class, we wrap ecalls with “functions” so you don’t need to call ecall directly.

Instruction types

- Instructions fall in 5 categories:
- Arithmetic Operators
- Memory
- Control
- Miscellaneous
- Extensions

RISC-V Extensions

- RISC-V offers a number of extensions which may optionally be implemented by a CPU
  - Multiplication/Division extension (much slower than other arithmetic instructions)
  - Float extension (requires significantly more circuitry)
  - Double extension (requires more circuitry AND 64-bit registers)
  - Other extensions for different use cases
- For the purposes of this class, we allow only one instruction from an extension: **`mul`**
- **`mul x5 x6 x7`**: Multiply x6 and x7 together, and store the result in x5
- Note that we do not include muli, or any division instructions.

Summary

- Today was a general overview of all the instructions available in RISC-V.
- Everything that we normally use when programming (variables, functions, loops) need to be constructed out of these instructions
- Assembly is easy to get wrong, so to ensure that everyone's code is compatible, we need to follow a set of conventions.

- **Tomorrow:** Common constructs from C, written in RISC-V + General conventions when writing code in RISC-V
- Aside: Please give me feedback!