CS/COE 0447

MIPS: Programs, Instructions, and Registers

wilkie (with content borrowed from:

Jarrett Billingsley

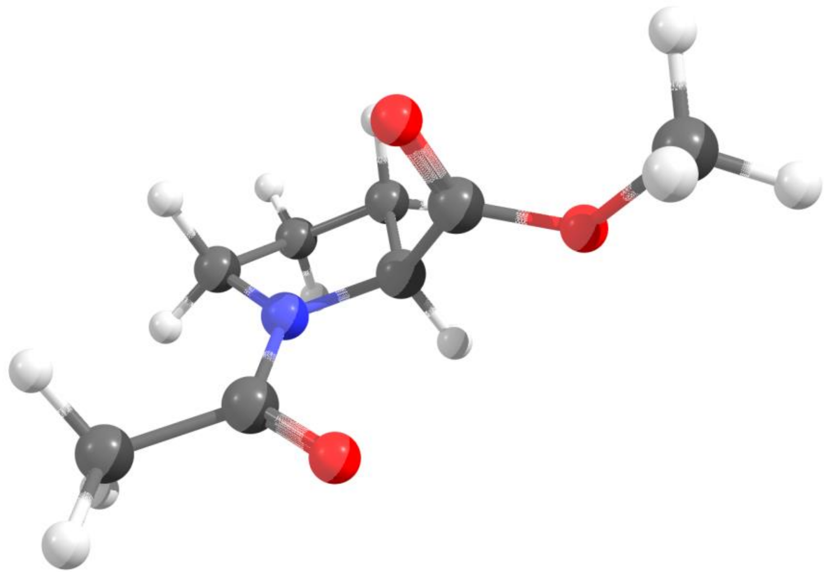
Dr. Bruce Childers)

Programs and Instructions

Programs? What are they…

Programs? Instructions? What are they?

- Chemistry/Physics Analogy:
  - *Instructions* are “atoms” and *programs* are “molecules”
  - Molecules have structure…
    - And that structure is often functional
    - And potentially nonsense… but whatever.



What are they really, though?

- **Instructions** are single, simple operations that a computer can carry out.
  - "add two numbers together"
  - "move a number from here to there"
  - "go to this place in the program" (jump)
  - "search this string for a character" (they can be complex)

- **Programs** are a series of these *tiny* instructions.
  - Well, how do we create these instructions?
  - How does a computer understand them?
    - (again… how does a computer understand *anything*???)

Assembly vs. Machine Language

- **machine language instructions** are the patterns of bits that a processor reads to know what to do
- **assembly language** (or "**asm**") is a human-readable, textual representation of machine language

MIPS asm	MIPS machine language
lw t0, 1200(t1)	100011 01001 01000 0000010010110000 lw t1 t0 1200
add t2, s2, t0	000000 10010 01000 01010 00000 100000 <math> s2 t0 t2 n/a add
sw t2, 1200(t1)	101011 01001 01010 0000010010110000 sw t1 t2 1200

What about "compilers"?

```
#include <stdio.h>

int main() {
  printf("hello!");
  return 0;
}
```

Source Code

`gcc`

Compiler

`hello.o`
`libc.a`

Object Files

`ld`

Linker

`hello`

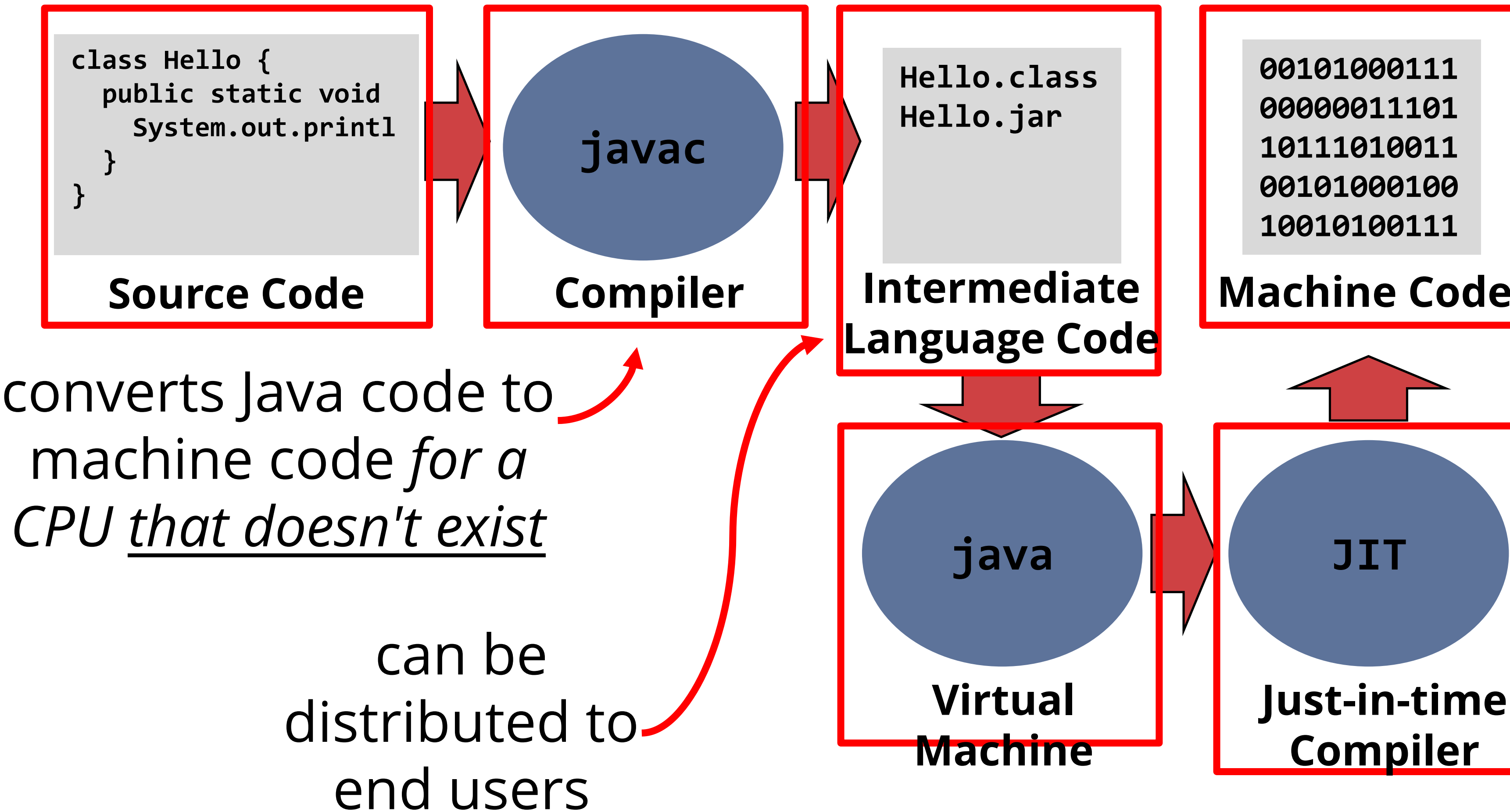
Executable

converts C code to machine code

sticks pieces of machine code together to make a program

(for more information, take CS0449!)

Virtual Machines?



Virtual Machines: Why learning asm matters!

- Developing Virtual Machine “intermediate languages” is like developing a processor.
- It’s a great way to make your software more portable!
- Pioneered by Infocom for text adventures
  - Lots of different computers with little standardization…
  - So just create your many games for *hypothetical hardware*
  - Then you just port the game engine. Genius!
- You develop your own **ISA** (instruction set architecture)

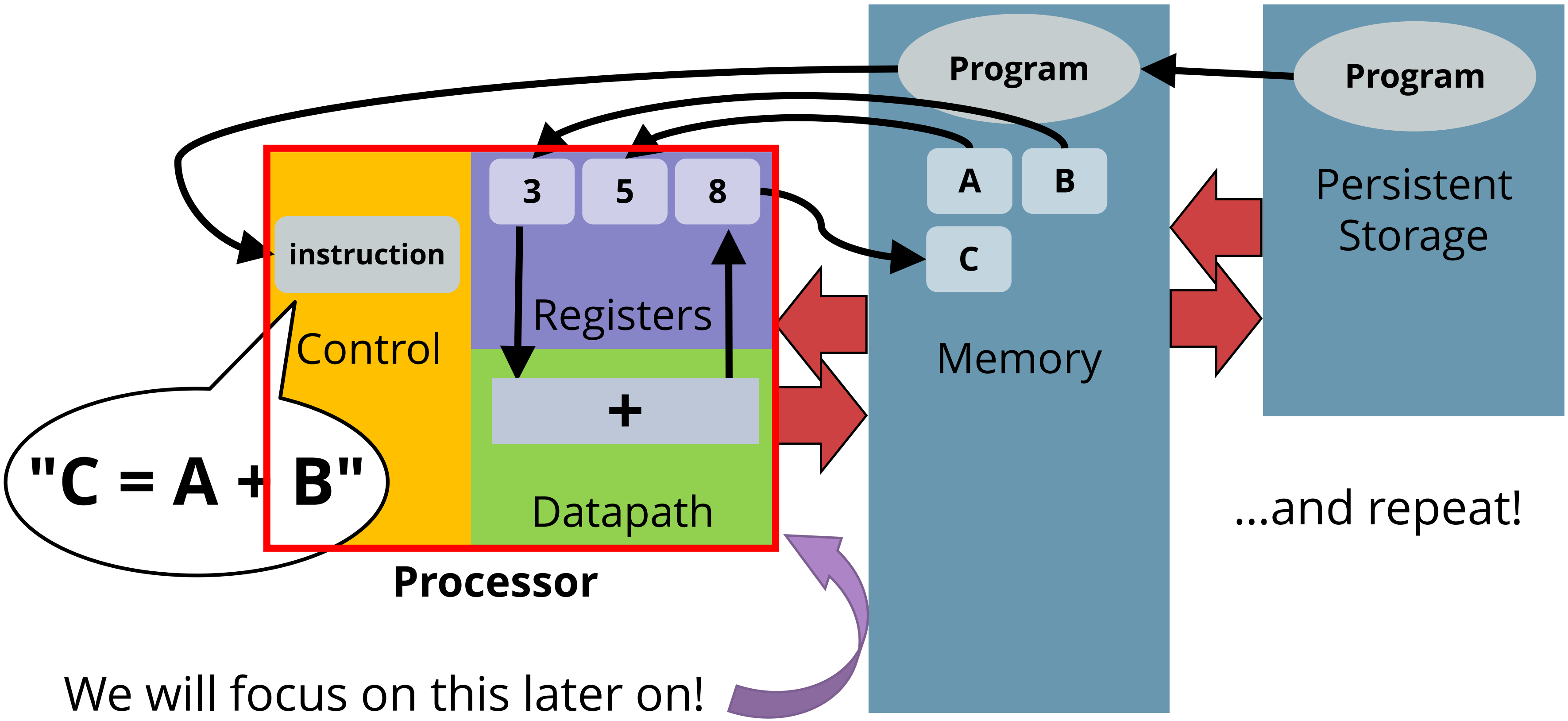
- Aside: https://ryiron.wordpress.com/2017/02/01/finding-the-lost-vikings-reversing-a-virtual-machine/ (Lost Vikings)

How a CPU runs a program!

- Step 1: Read next instruction
- Step 2: Do the instruction
- Step 3: Go to step 1

- Ok… maybe there’s a little more to it…

How a CPU runs a program!



We will focus on this later on!

How do we construct a CPU?

Not like this

ISAs

Instruction Set Architecture

Instruction Set Architecture (ISA)

- An **ISA** is **the interface that a CPU presents to the programmer**
  - when we say "architecture," *this* is what we mean
- It defines:
  - what the CPU **can do** (add, subtract, call functions, etc.)
  - what **registers** it has (we'll get to those)
  - the **machine language**
    - that is, the bit patterns used to encode instructions
- It **does not** define:
  - how to design the hardware!
    - ...if there's any hardware at all (remember: virtual/hypothetical ISAs)

ISA Example: X86

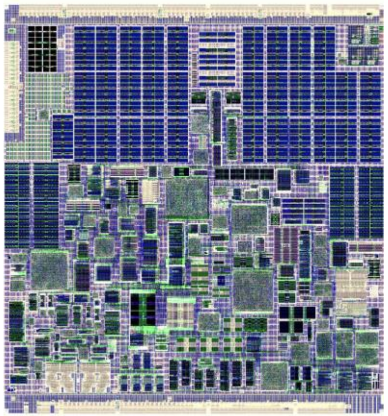
- descended from 16-bit 8086 CPU from ***1978***
- extended to 32 bits, then 64
- each version can **run most programs** from the previous version
  - you can run programs written in 1978 on a brand new CPU!

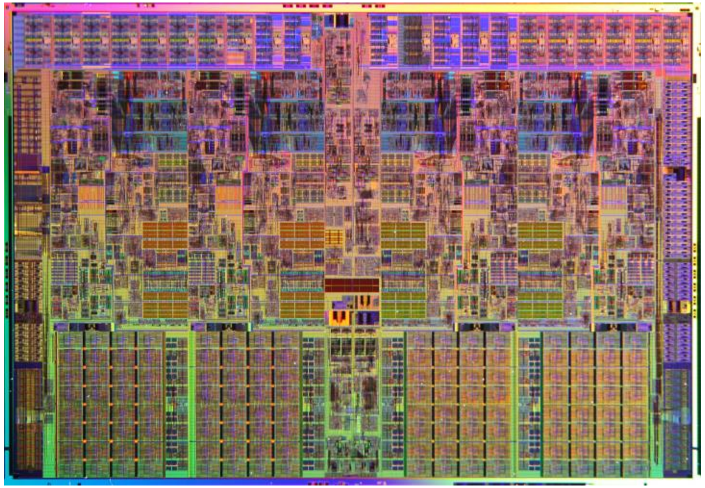
- “so why don't we learn x86 in this course?”
  - It can do *a lot* of things…
  - Its machine language is very complex! (40 years of growth!!)
  - Making an x86 CPU is… difficult.
  - Ultimately, **we would waste a ton of time.**

All three processors run the same programs...

- but they're TOTALLY different on the inside

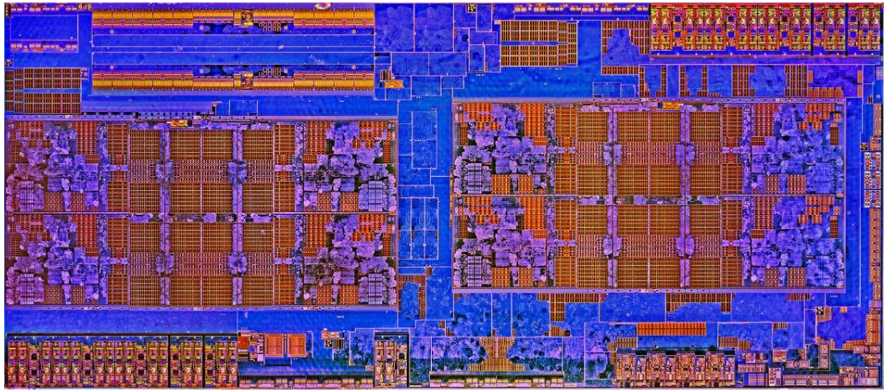
VIA Nano





Intel Core i7

AMD Zen



Types of ISAs: CISC

- **CISC:** "**C**omplex **I**nstruction **S**et **C**omputer"
- ISA designed for **humans** to write asm
  - from the days *before compilers!*
- **lots** of instructions and ways to use them
- **complex** (multi-step) instructions to shorten and simplify programs
  - "search a string for a character"
  - "copy memory blocks"
  - "check the bounds of an array access“
- Without these, you’d just write your programs to use the simpler instructions to build the complex behavior itself.

- x86 is *very* CISCy

Types of ISAs: RISC

- **RISC:** "**R**educed **I**nstruction **S**et **C**omputer"
- ISA designed to make it easy to:
  - **build the CPU hardware**
  - make that hardware **run fast**
  - **write *compilers*** that make machine code
- a **small number** of instructions
- instructions are **very simple**
- MIPS is *very* RISCy
- MIPS and RISC were the original RISC architectures developed at two universities in California
  - the research leads were… Patterson and Hennessy…

Popular ISAs Today:

- **x86** (these days, it's x86-64 or "x64")
  - most laptops/desktops/servers have one
  - (modern x86 CPUs are just RISC CPUs that can read the weird x86 instructions)
- **ARM**
  - almost *everything else* has one
  - ARMv8 (AArch64) is pretty similar to MIPS!
- **Everything else:** Alpha, Sparc, POWER/PPC, z, z80, 29K, 68K, 8051, PIC, AVR, Xtensa, SH2/3/4, 68C05, 6502, SHARC, **MIPS**…
  - microcontrollers, mainframes, some video game consoles, and historical/legacy applications
- despite its limited use today, MIPS has been incredibly influential! (Essentially all RISC chips model it)

Types of ISAs: Overview

- **CISC:** Complex Instruction Set Computer (does a whole lot)
- **RISC:** Reduced Instruction Set Computer (does enough)
- **Both: Equivalent!!** (RISC programs might be longer)

"Hackers" (1995) – Of course, they are talking about a Pentium x86 chip… which thanks to its backwards compatibility, is ***<u>CISC</u>***. Oh well!

The MIPS ISA: Registers

The Local Storage of the MIPS CPU

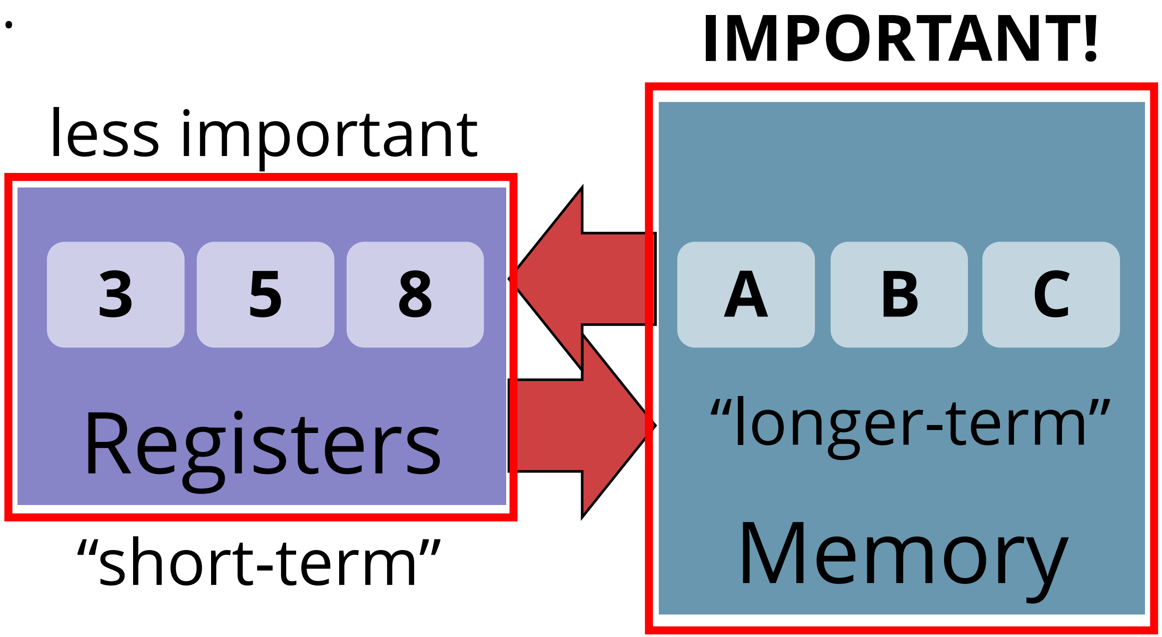
Registers

- **registers** are a kind of small, **fast,** temporary memory inside the CPU
- the CPU can **only operate on data in registers**
- MIPS has **32 registers**, and each is 32 bits (one **word**)
- the registers are numbered 0 to 31… ($0, $17, $31)
  - …but they also have nice names ($a0, $t1, $bp)
    - (I’m ambivalent about the dollar signs)
      - (Jarrett modified MARS so you don't have to use them)

```
     add $t0, $a0, $a1      (t0 = a0 + a1)
(or) add t0, a0, a1
```

Juggling Data

- Registers are... like..... hands.
- **You have a limited number and they can only hold small things.**
- Your program's variables primarily live **in memory**.
- The registers are just **a temporary stopping point** for those values.



Really… You don’t have that many…

- **You cannot write every program using only registers**
  - don't try to
    - please.
- **Every piece of your program has to share the registers.**
  - unlike high-level languages which manage this for you
  - (Compilers are nice! And complex! Assembly will teach you why!)

- **Every register has a general convention.**
  - Helps us keep track of them in our programs.
  - These social conventions help others understand our programs.

The “s” (saved) Registers

- In MIPS, there are 8 saved registers, **s0** through **s7**
- These are *kinda* like... local variables inside a function

```
s0 = 3;         li  s0, 3
s1 = 5;         li  s1, 5
s2 = s0 + s1;   add s2, s0, s1
```

- Here are your first MIPS instructions!
- **li** stands for "load immediate." what does it look like it does?
  - "immediate" means "number inside the instruction"
- **add**, uh, well, it adds... numbers.
- just like in Java, C, whatever: the **destination** is on the left

The "t" (temporary) registers

- There are ten temporary registers, **t0** through **t9**
- These are used for *temporary* values – values that are used briefly.
  - For example, say we had a longer expression:

```
s4 = (s0 + s1 - s2) * s3
```

- What does algebra say about what order we should do this in?

```
add t0, s0, s1
sub t0, t0, s2
mul s4, t0, s3
```

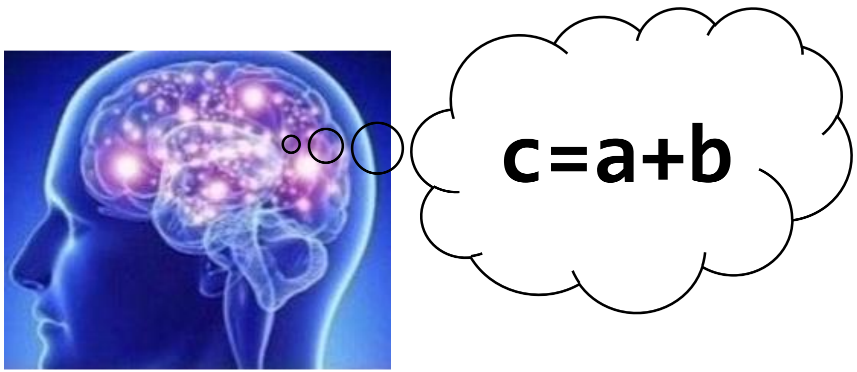
When do I use one over the other?

- We'll learn more about this in the coming weeks.
- Rule of thumb:
  - Use a “t” register first.
  - Unless you *need* the value to persist across function call (“s” register)
    - ok that's not too clear yet
      - uhhhhhhh we'll come back to this
- Basically 90% of your code will use “s” and “t” registers.

Keeping Your Humanity

- Writing asm is a different way of programming than you're used to. You have to think like a machine sometimes.
- To make the transition easier, try to **reduce your cognitive load.**
    - **cognitive load** is "the set of ideas you have to keep in your mind to perform some task."
    - **high-level languages (HLLs)** reduce cognitive load by hiding the machine code, using a compiler to write it for you.
- You can do the same thing: think about how to write a program in e.g. C, and then turn *that* into asm.
- That is, write in the higher-level language (and keep it as a comment in your code!) and then translate.

Keeping Your Humanity




```
add c, a, b
```

```
# c = a + b
add s2, s0, s1
```

There and Back Again

- going the other way is also useful

how would we write this in C/Java?

```
mul t0, s2, 33    t0 = s2 * 33
div t1, s3, s4    t1 = s3 / s4
sub s1, t0, t1    s1 = t0 - t1
```

or, if we rolled it all together,

```
s1 = (s2 * 33) - (s3 / s4)
```

that's what this asm *does*

Why?? (Reprise)

- 447 is about **building a mental model of how a computer works**.
- Understanding what is happening when you write code or run programs gives you a much deeper understanding:
  - “Why should I avoid using this programming language feature in this speed-critical part of my code?"
  - “Why wouldn't this crazy idea be very fast on current architectures?"
  - “This program is breaking in a really confusing way, I have to look at the asm to debug it"
- This stuff is specialized but hey you're majoring/minoring in it right?