CSE 260M / ESE 260
Intro. To Digital Logic & Computer Design

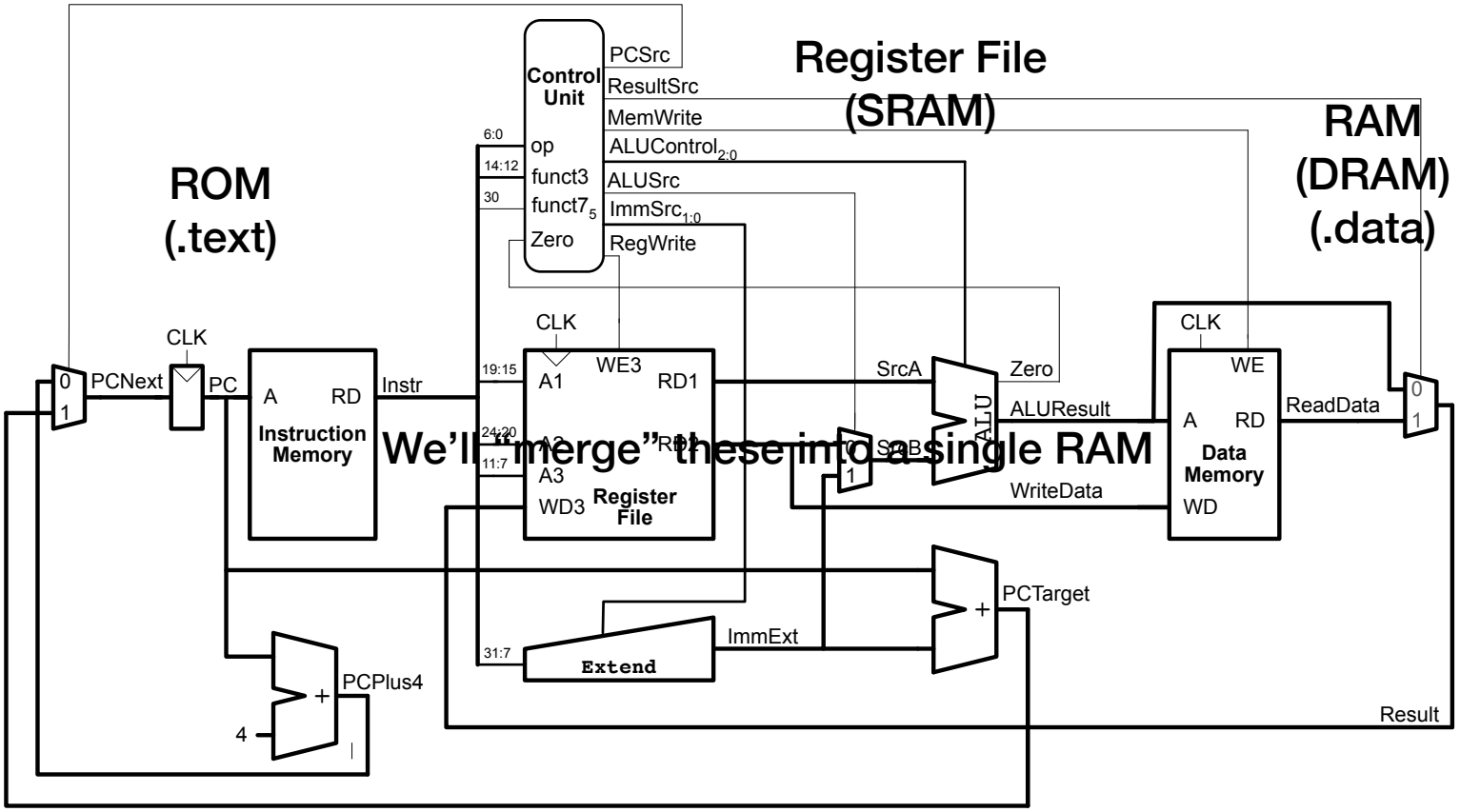
Bill Siever
&
Michael Hall

This week

- Homework 6B posted - Due Tuesday, April 8th by 11:59pm
  - Gradescope dropbox on Thursday
- Thursday: Won’t need kits

Studio 6A

Foreshadowing: Simple RISC-V Computer



Basic Model

- Machine is basically 2-3 memories + CPU
  - Registers (small, easy to use; temporary/ephemeral)
    - Ex: You have 31, 32-bit data registers = 124 ***Bytes***
  - RAM: Place for most data (Gigabytes!)
  - Program Memory: Possible in RAM or some additional “program memory”

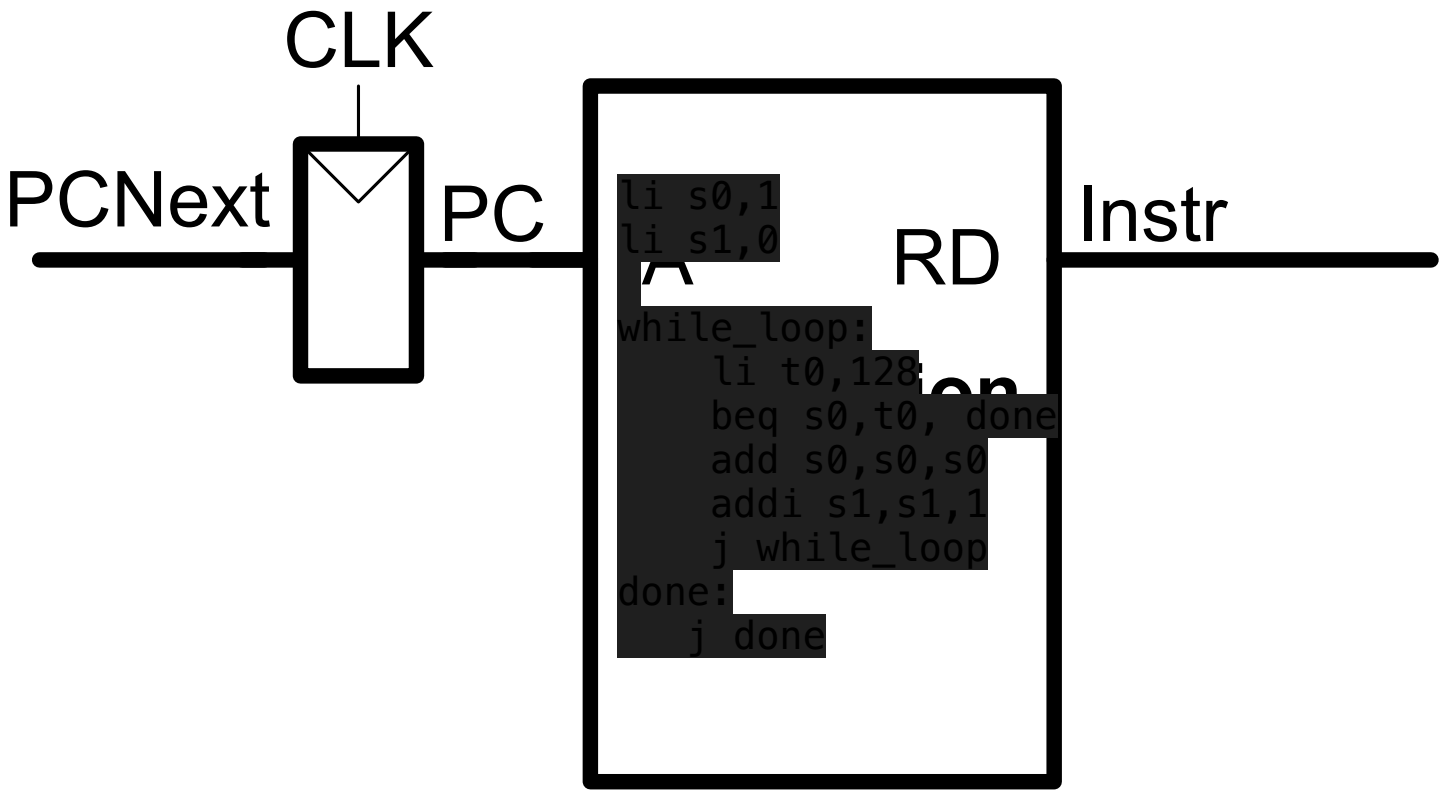
Program -> Assembly Language -> Machine Code -> Memory

Problem: Find x such that $2^x = 128$

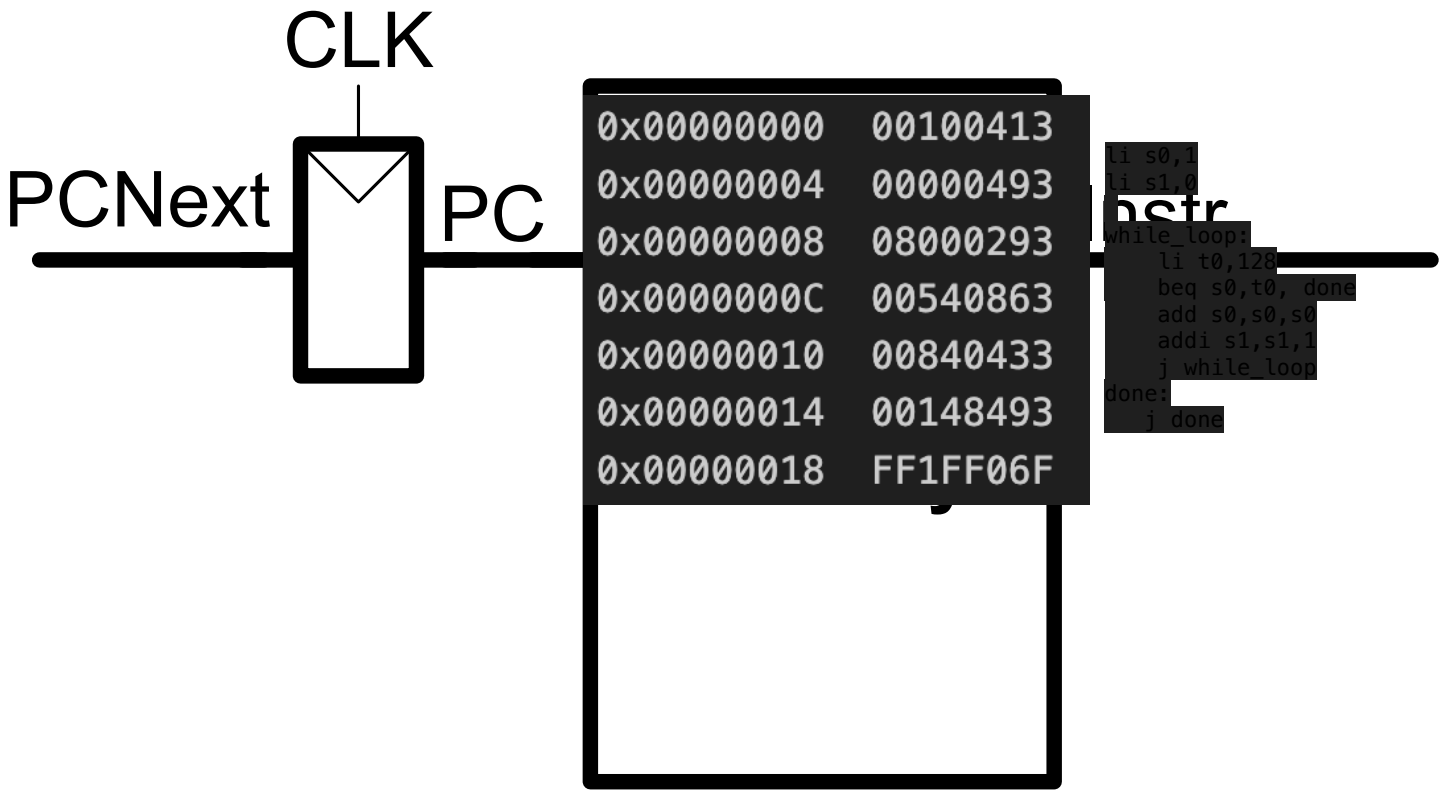
```
// determines the power
// of x such that 2x = 128
int pow = 1;
int x = 0;

while (pow != 128) {
   pow = pow * 2;
   x = x + 1;
}
```

Behavior: Parts of CPU Model



Behavior: Parts of CPU Model



Chapter 6

Instruction Sets: Basic Categories

- Machine has small primitive set of “commands” in a few rough categories:
  - Data Manipulation: “Computation” (typically uses an ALU)

```
add t0,t1,t2
```

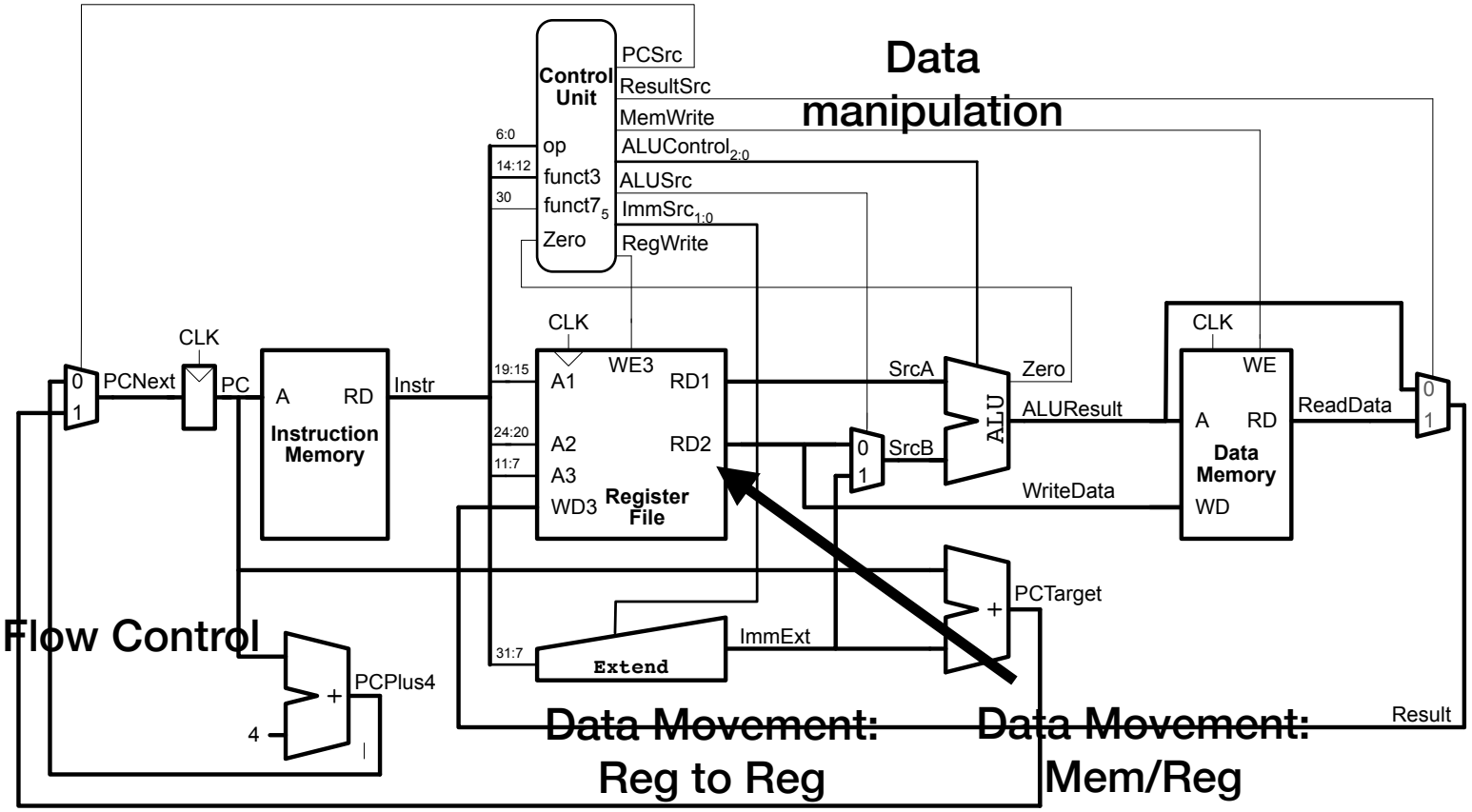
  - Data Movement: Move data between registers and RAM or initializing values

```
lw t0, 8(sp)
li t1,5
```

  - Flow Control: Controlling what instruction happens next (loops, if/else, functions)

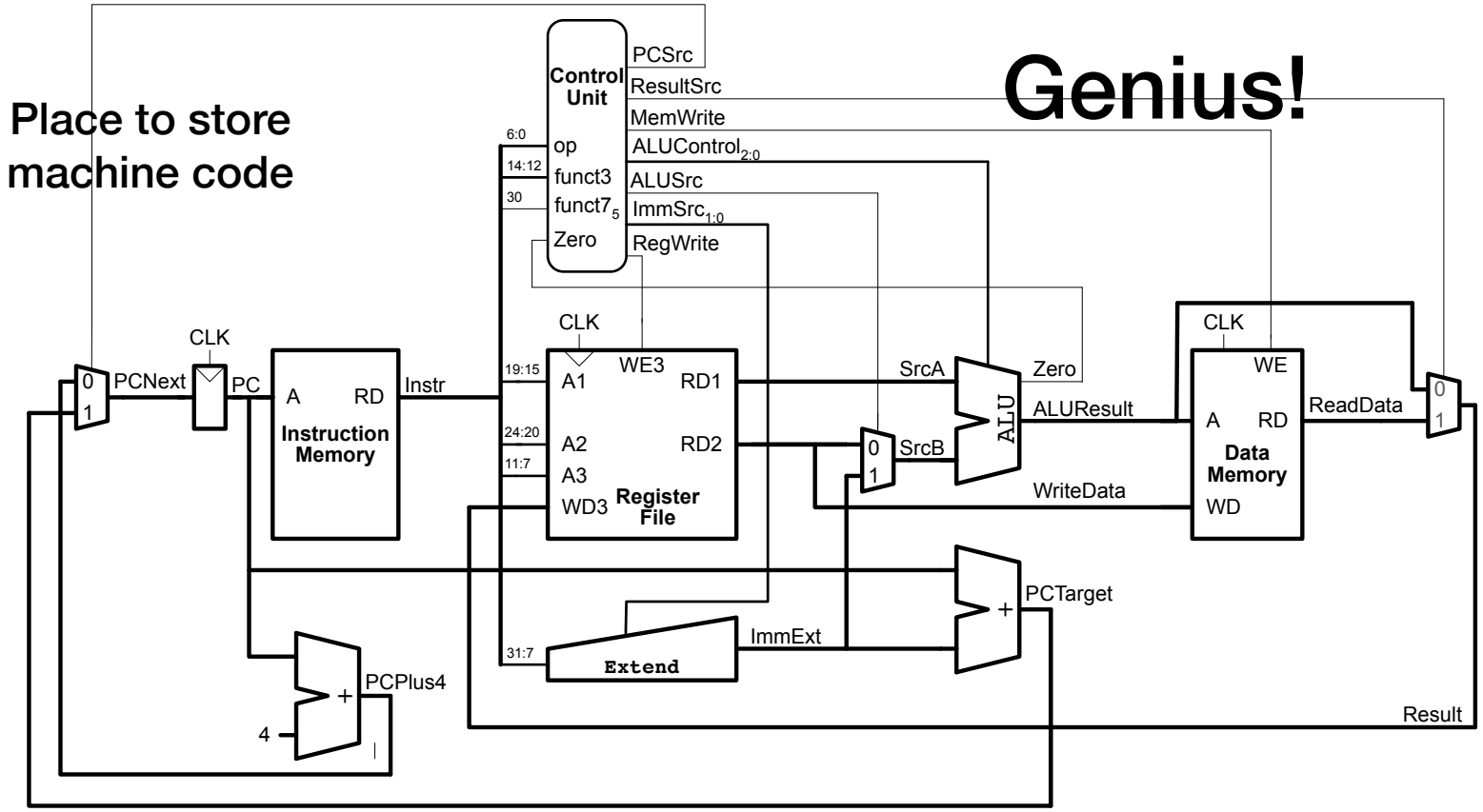
```
beq t0,t1, done
```

Foreshadowing: Simple RISC-V Computer



“Stored Program” Concept

Genius!



260 Prereq: Intro to Programming

Programming Languages

- Prereq: Intro to Programming
  - Consider large programs and how you manage info.
  - Ex: A program/function that computes an average of three integers
    - Java vs. Python vs. C
  - Examples
  - “Types”

Programming Langs: History & Motivation

1. Efficiency: Allow more people to create programs

   Compilers “compile” a high-level language (HLL) representation to a list of simpler assembly language instructions

   (Compilers are used obviously/explicitly in many languages, like Java & C; Often behind-the-scenes in others, like Python)

2. Manage complexity / avoid problems (increasing focus)

   Variables and Data Type rules are a large part of that

Registers

- Just 31, 32-bit registers (124 bytes)
  - Used for all data operations!
  - Very very different than HLL
- **No types**
- **Meaning may change with time**

Name	Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

```c
#include <stdio.h>

float mean(int a, int b, int c) {
    int sum = a+b+c;
    return sum/3.0;
}

void main(void) {
    float average;
    average = mean(1, 3, 3);
    printf("Average: %f\n", average);
}
```

a0: int (a)

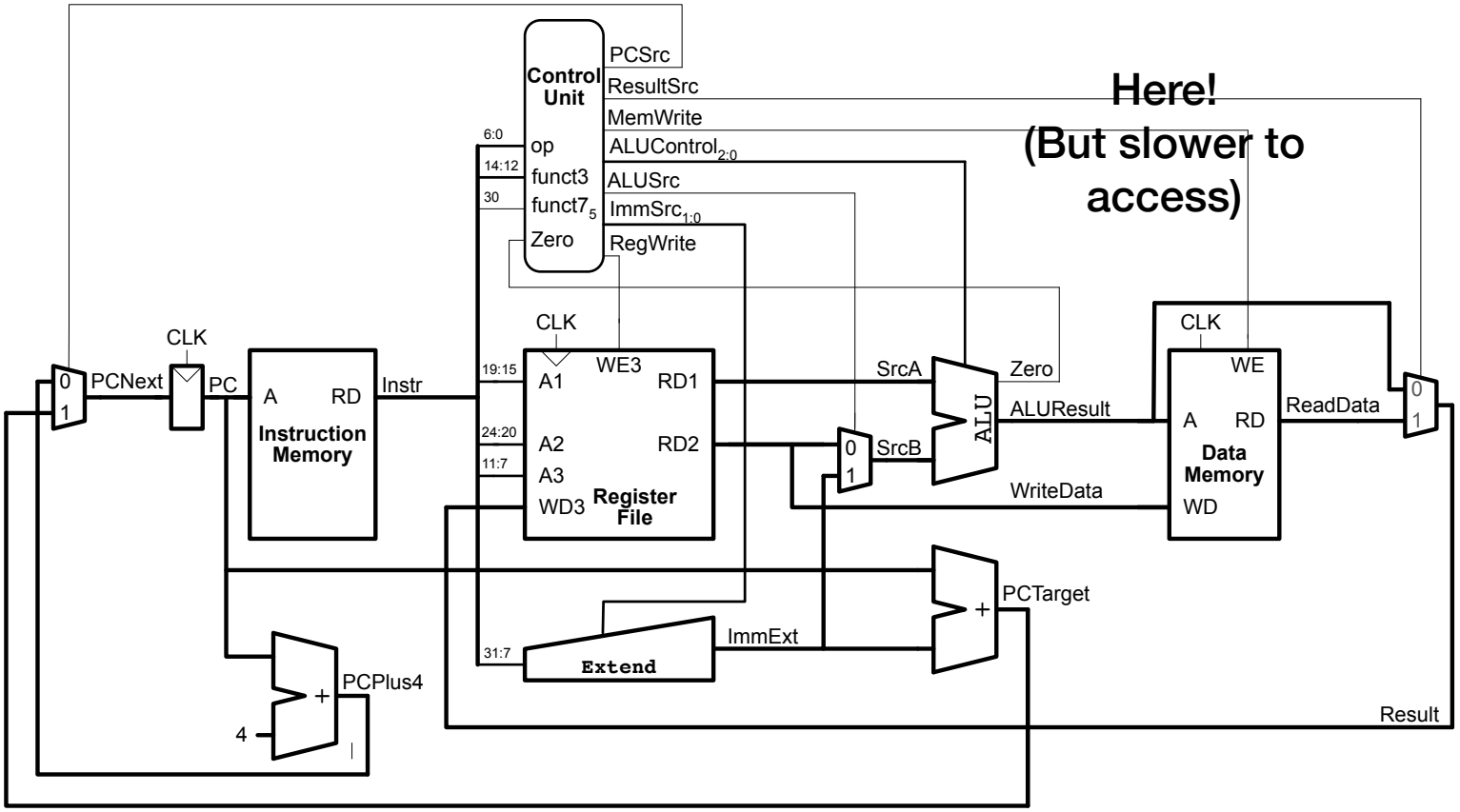
a1: int (b)

a0: float (result)

a0: string (char *)

a1: float (copy of average's value)

More Data!


Here!
(But slower to access)
Control Unit
PCSrc
ResultSrc
MemWrite
ALUControl2:0
ALUSrc
ImmSrc1:0
RegWrite
op
funct3
funct75
Zero
6:0
14:12
30
CLK
PCNext
PC
Instruction Memory
A
RD
Instr
19:15
24:20
11:7
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
SrcA
SrcB
ALU
Zero
ALUResult
WriteData
Data Memory
WE
WD
ReadData
PCTarget
ImmExt
Extend
31:7
PCPlus4
4
Result

The Rules

- Life Lesson
  - I have an older brother…
    - Who went to college
    - Left stuff at home…
    - That I wasn't supposed to touch!

The Rules

- Fun…
  - And failure…
- Discovery: Just don’t be discovered!
  - As long as things are put back *exactly* as I found them…
- Sharing Registers / Memory: You can do almost anything as long as you put things back how you found them before finishing (but how?)

The Rules

- Register Conventions:
  - “Convention” (agreement about use) for how registers will be shared
- & Beyond — Memory & the “Stack Discipline”
  - Rules for how to use memory for additional info
  - And to use as the “copy” to restore to original condition

The Rules

- Memory
  - Use part for the “run time stack”
  - Create a “stack”: Like stack data structure (CSE 247 / 2407)
    - Last-in-first-out

LIFO

Why a stack

- `main()`
  - `run_game()`
    - `update()`
      - `update_character()`
        - `update_life()`

Why a stack

- `main()`
  - `run_game()`
    - `update()`
      - `update_character()`
        -

Why a stack

- `main()`
  - `run_game()`
    - `update()`
      -

Why a stack

- `main()`
  - `run_game()`
    -

Why a stack

- `main()`
    -

LIFO with an array

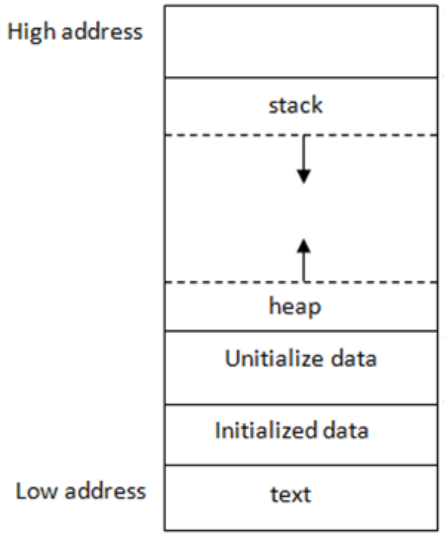
- Use an array
  - Keep track of index of “last item”
  - Add / (push)
  - Remove (pop)

Data / RAM

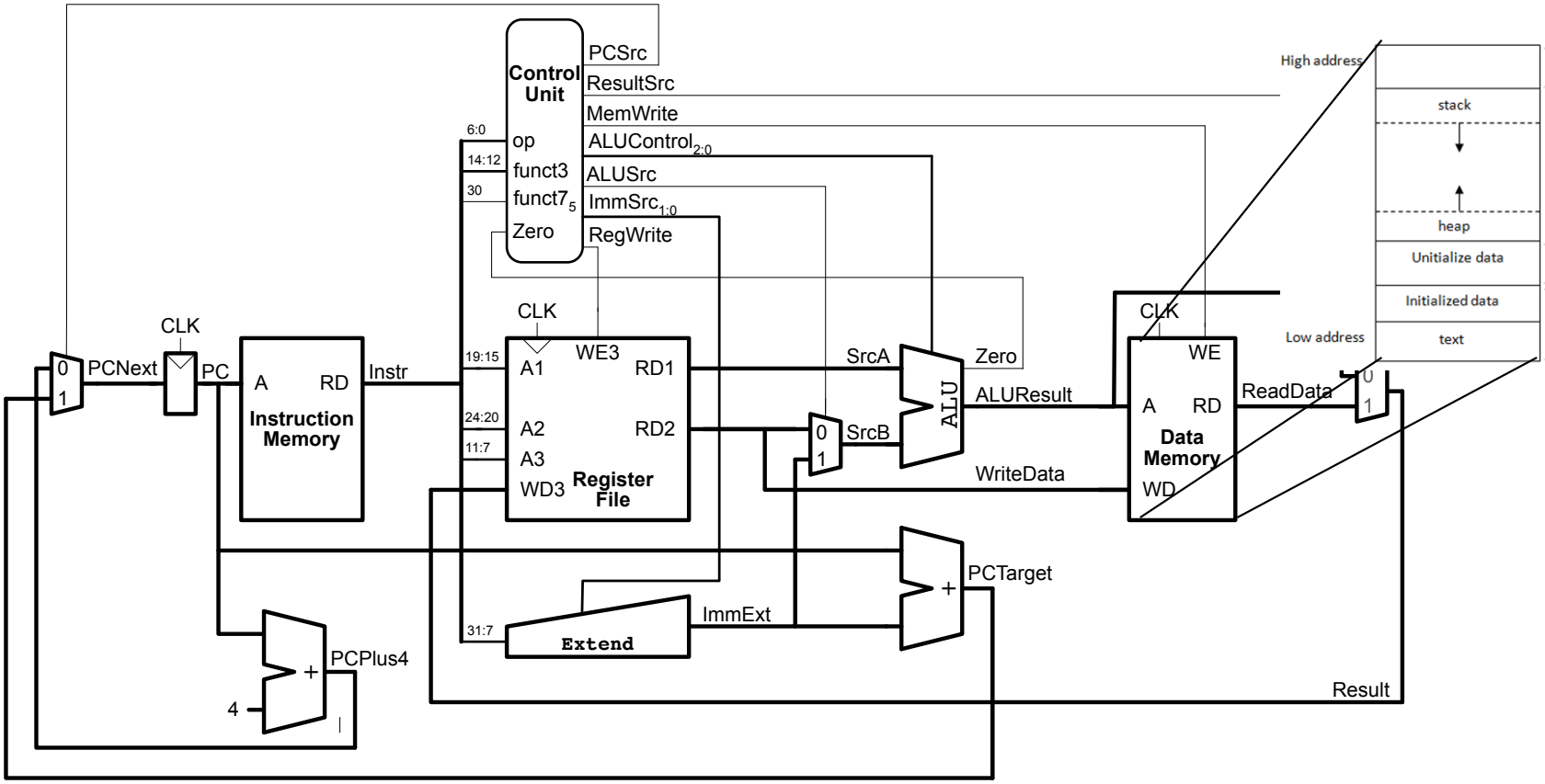
- Arrays (in programming languages) are just a representation of a segment of RAM
  - So, RAM works like arrays — index based
  - There's a "base": The index that it starts at
  - However, RAM is an array of BYTES
    - Data types like an `int` are 4 bytes

Data / RAM

- Split it into regions to serve different purposes
  - Stack
    - Of records for all the currently running functions
  - Heap: A giant heap ‘o memory
    - For *dynamic* memory (`new` / `malloc`).
      like in Java: `Thing newThing = new Thing()`



More Data!


Control Unit
PCSrc
ResultSrc
MemWrite
ALUControl2:0
ALUSrc
ImmSrc1:0
RegWrite
op
funct3
funct7 5
Zero
6:0
14:12
30
CLK
PCNext
PC
Instr
A
RD
Instruction Memory
19:15
24:20
11:7
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
SrcA
SrcB
ALU
Zero
ALUResult
WriteData
CLK
WE
A
RD
WD
Data Memory
ReadData
High address
Low address
stack
heap
Unitialize data
Initialized data
text
PCTarget
ImmExt
31:7
Extend
PCPlus4
4
Result

Studio 6A, Function Fun, & RA

Compiler Basics & Function fun (sum.c)

Questions

- Why RVC / reduced instruction sizes?
  - Smaller inst memory -> Faster / cheaper;
    Don’t waste $ on unneeded resources!
- Why isn’t RISC-V more popular? Still very new.
- Why not Intel? (Or ARM?) Isn’t this messier? ….

Arrays

```
int i;              // use s1
int scores[200];  // use s0 for the base of scores
for (i = 0; i < 200; i = i + 1)
   scores[i] = scores[i] + 10;
```

Next Time

- Studio