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

Bill Siever
&
Michael Hall

## This week

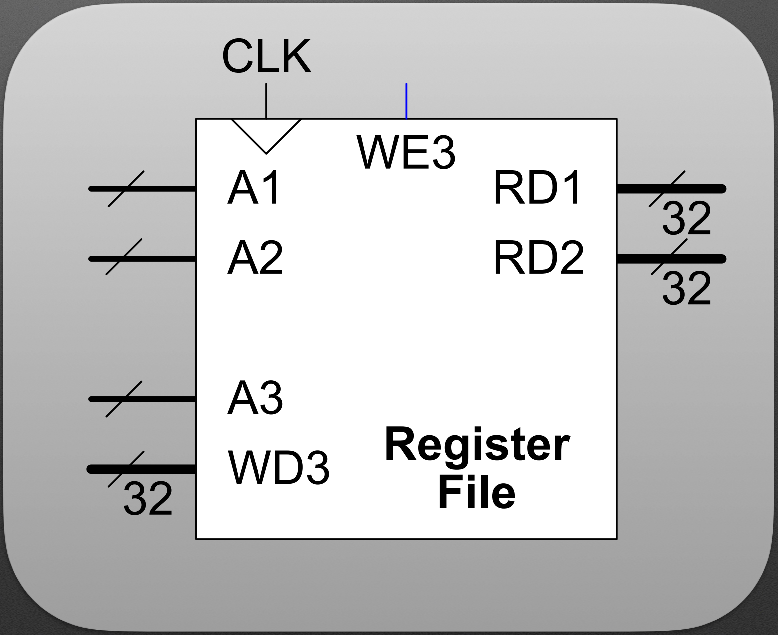
- Homework 6A posted tonight
    - Gradescope dropbox by Thursday
- Thursday:  Won’t need kits
    - Will post to Piazza when available Will span week.

## Studio 5

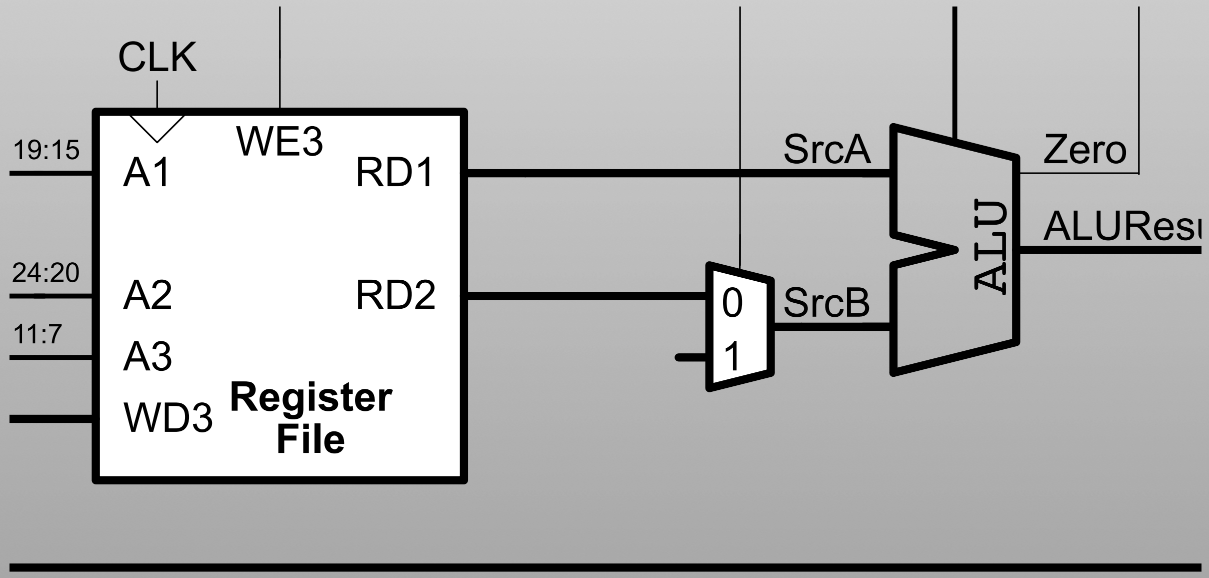
# Chapter 5

## Review: Register File

- ALU will Need TWO inputs: need a memory structure that provides two values (I.e. dual output ports)
- The "Register File"
- Also supports writing (updating)



# Big Picture: add x, y, z



## Verilog: RISC-V Register File

```
// 32 x 32 register file with 2 read, 1 write port
module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [4:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  always_ff @(posedge clk)
    if (we3) rf[wa3] <= wd3;

  assign rd1 = rf[ra1];
  assign rd2 = rf[ra2];
endmodule
```

## FPGA

- Field Programmable
- Gate Array
  - Lattice iCE40 UP5k: Architecture Overview
    - RAMs, (Dual and Single Port)
    - Look Up Tables (LUTs): 4 inputs
    - D Flip Flops
    - Lots: ~5,000

## Questions

- Why so many memory types / what are the differences?
    - Evolution over time
    - Different needs:  Capacity vs. Need — the memory hierarchy

## Questions

- PLA vs. FPGA
  - PLA: (largely) 2-level logic / simple combinational logic
  - FPGA: Array of many programmable blocks with programmable interconnects
    - Can efficiently achieve more than 2-layer logic
    - Memory/storage is inherent (can do full state machine…see hw 4b)

# Chapter 6

## Architectures

- “Architecture”: Programmer’s view of CPU
  - “Instruction Set Architecture” (ISA):
    Precise details of structure of cpu model, instructions, their semantics, and their encoding
    - Examples: RISC-V, ARM, MIPS, x86/IA64
  - Microarchitecture: How CPU is built to read/do ISA
    - Where Digital Logic becomes actual machine!

## RISC-V ISA

- “Open Source” ISA
- Book Covers / PDF: https://www.yellkey.com/impact (good for 24 hours)
    - Assembly Language
    - Machine Language

# Registers

| Name | Register 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 |

## RISC-V Design Criteria

1. Favor regularity (things that are consistent)
   a = b+c    => `add a,b,c`
   Subtract? (a=b-c)

   - => `sub a,b,c`

2. Make most used instructions fast (largest impact on performance)

3. Smaller is (usually) faster. Small, efficient memory can be key to performance. Like…the register file!

4. Can’t do everything well: Compromises are necessary

## 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”

## Basic Model

- 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
    ```

## “Stored Program” Concept

- Assembly instructions can be represented by numbers
  - A substitution code: Replace symbols with numbers using pattern
- Convert `add t0,t1,t2` to machine code (32-bit hexadecimal) (Hint: `t0 = x05`)
  - What about `sub t0,t1,t2` ?

## Assembly Language Programming Basic Data Manipulation (ALU)

- (Independent / non-cumulative) Examples: Assuming a in s0, b in s1, etc.

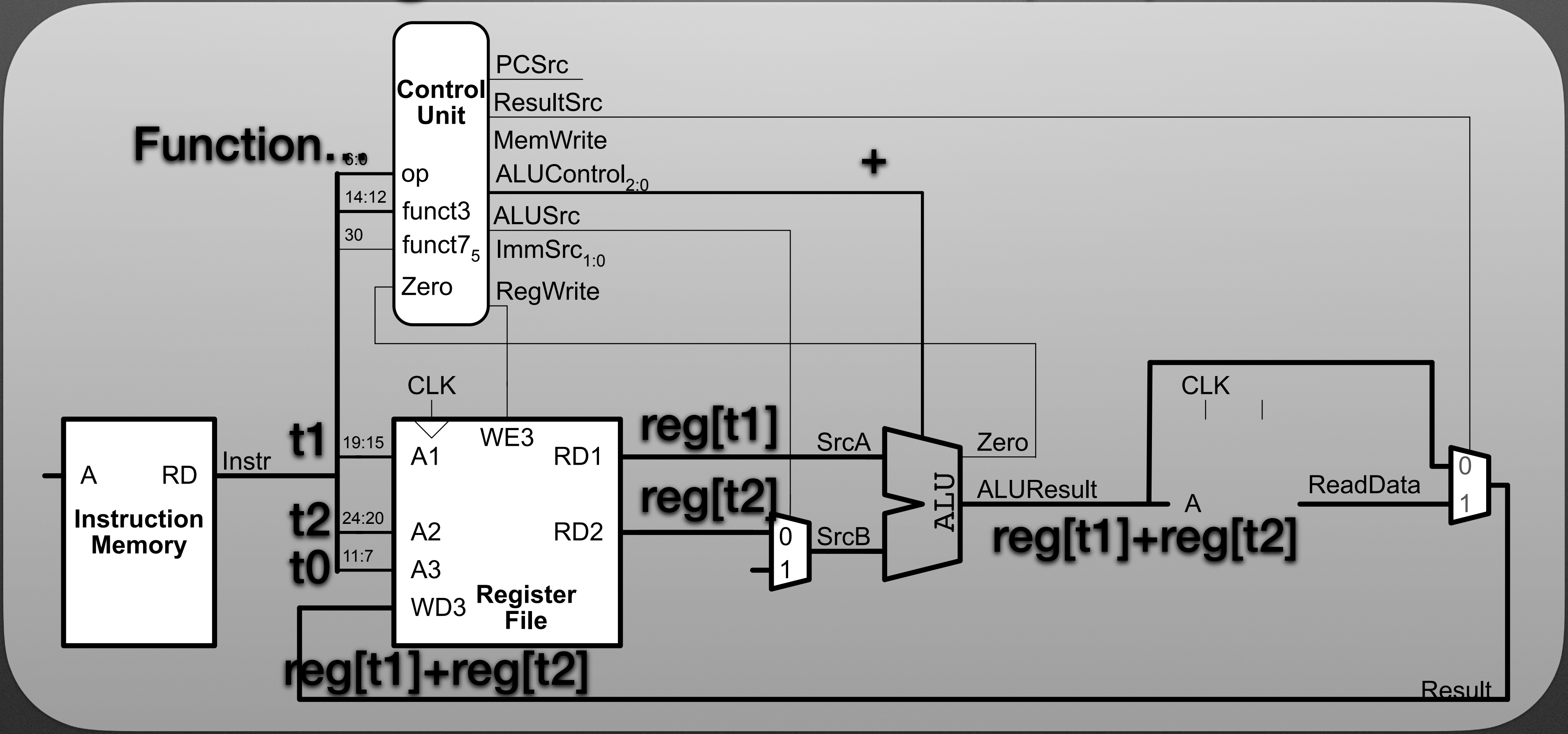
```
1.  a = b+c-d

2.  a = b+4

3.  a = 7

4.  a = b
```

# Big Picture: add t0, t1, t2



## Loops & Labels: Basic

- Label:  Used in assembly language…to label a line of code
  - Instructions are in a memory
  - They have an index
  - Labels turn into a number for that index
- Syntax:    identifier:
- Use:  Loops, if/else (decisions), functions/methods

## Loops & Labels: For-loop

- Label:  Used in assembly language…to label a line of code

```
 // add the numbers from 0 to 9
int sum = 0;    // Use s1
int i;          // Use s0
for (i = 0; i < 10; i = i + 1) {
  sum = sum + i;
}
```

## Pre-condition Loops: To ASM

- One pattern / template:  There are alternatives that sometimes are better in some sense

```
      // add the numbers from 0 to 9
     initialization …

loop_start_label:
  loop_check / jump to loop_end_label

  loop body (including increment)
  j loop_start_label

loop_end_label:
```

```
for (i = 0; i < 10; i = i + 1) {
   sum = sum + i;
 }
```

# Pre-condition Loops: To ASM

- One pattern / template:  There are alternatives that sometimes are better in some sense

```
      // add the numbers from 0 to 9
     ~~initialization~~ …

loop_start_label:
  loop_check / jump to loop_end_label

  loop body (including increment)
  j loop_start_label

loop_end_label:
```

```
while (i < 10) {
   sum = sum + i;
   i = i + 1
 }
```

## Conditionals & Labels: if-statement

```
// add the numbers from 0 to 9
int sum = 0;    // Use s1
int i;          // Use s0
for (i = 0; i < 10; i = i + 1) {
  sum = sum + i;
  if (i==4) {
     print(sum);  // ecalls
  }
}
```

## Pre-condition if: To ASM

- One pattern / template:  There are alternatives that sometimes are better in some sense

```
    check condition and branch to avoid body

  body

end_label:
```

```
if (i == 4) {
   …
 }
```

## 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

- Assume array named `scores` starts at address 100. I.e., RAM[100]
  - What is the RAM index of scores[1]

## 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