

CSE 2600

Intro. To Digital Logic & Computer Design

Bill Siever
&
Michael Hall

This week

- Homework 6A posted later today (hopefully)
- Studio 6A

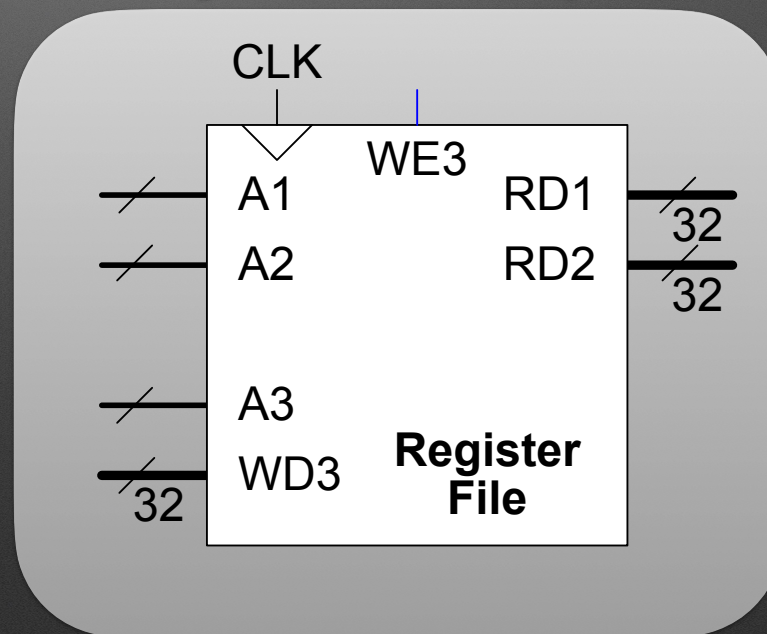
Studio 5

Chapter 5

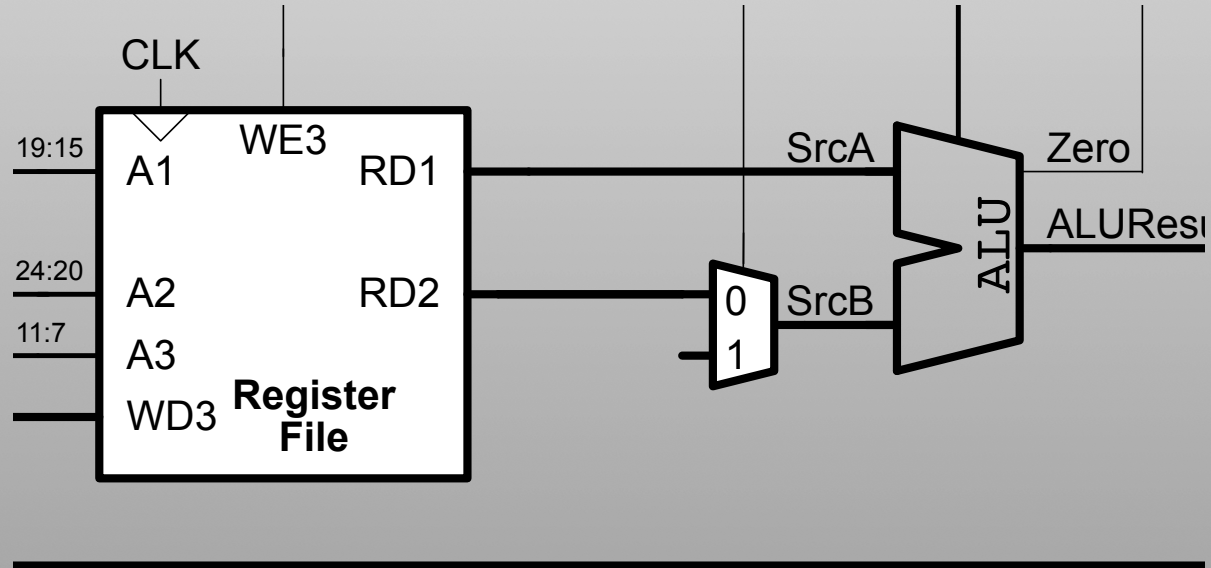
(& Beyond: It's all coming together)

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
```

Questions

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

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
 - Chapter 7: Microarchitecture: How CPU is built to read/do ISA
 - Where Digital Logic becomes actual machine!

RISC-V ISA

- “Open Source” ISA
- Book Covers / PDF: www.yellkey.com/short (good for 24 hours)
 - Get comfortable with this: needed for assignments / provided on exam
 - Assembly Language
 - Machine Language

Registers: An array of numbers

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)

$a = b+c \Rightarrow \text{add } a, b, c$

Subtract? ($a=b-c$)

- $\Rightarrow \text{sub } a, b, c$

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 + ALU
 - 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 / Machine Code

- 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.

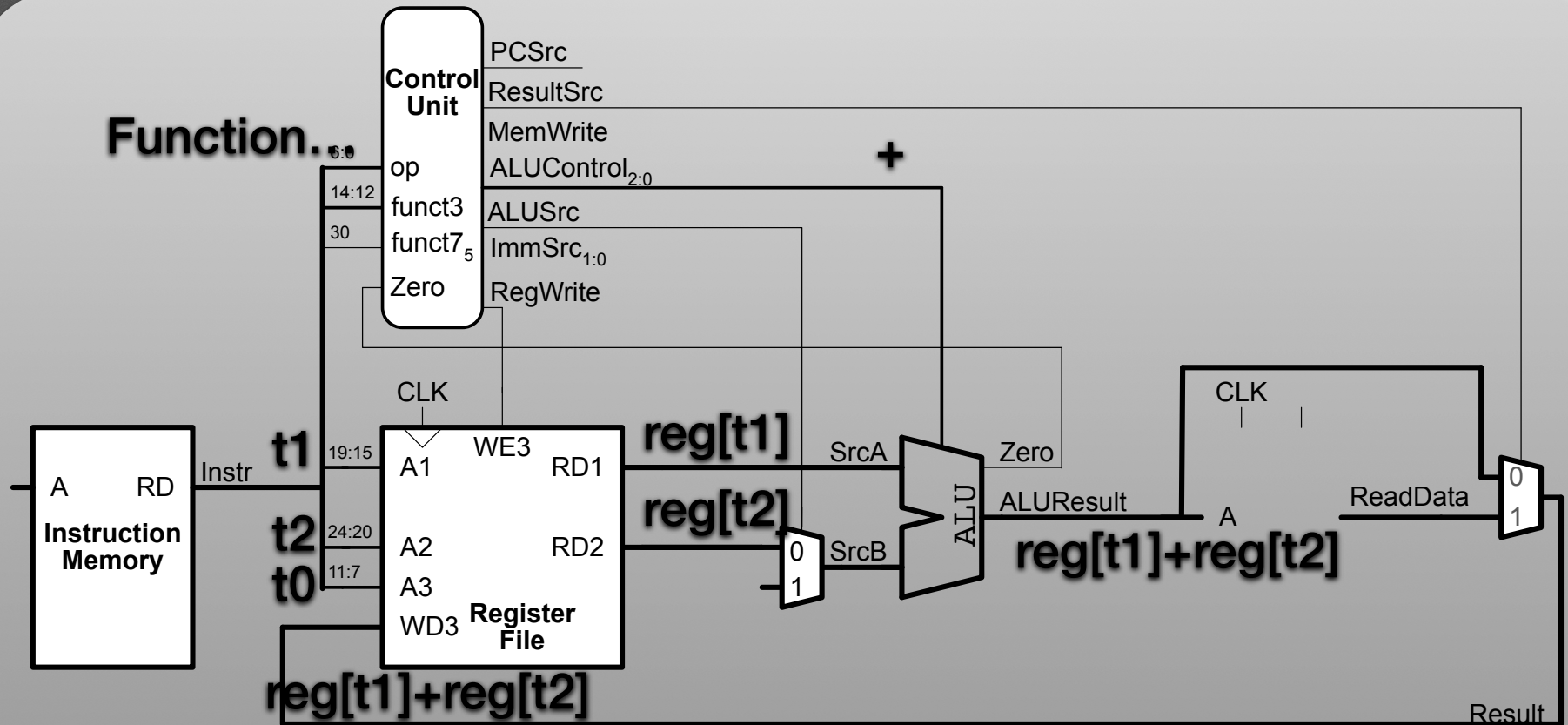
1. $a = b + c - d$

2. $a = b + 4$

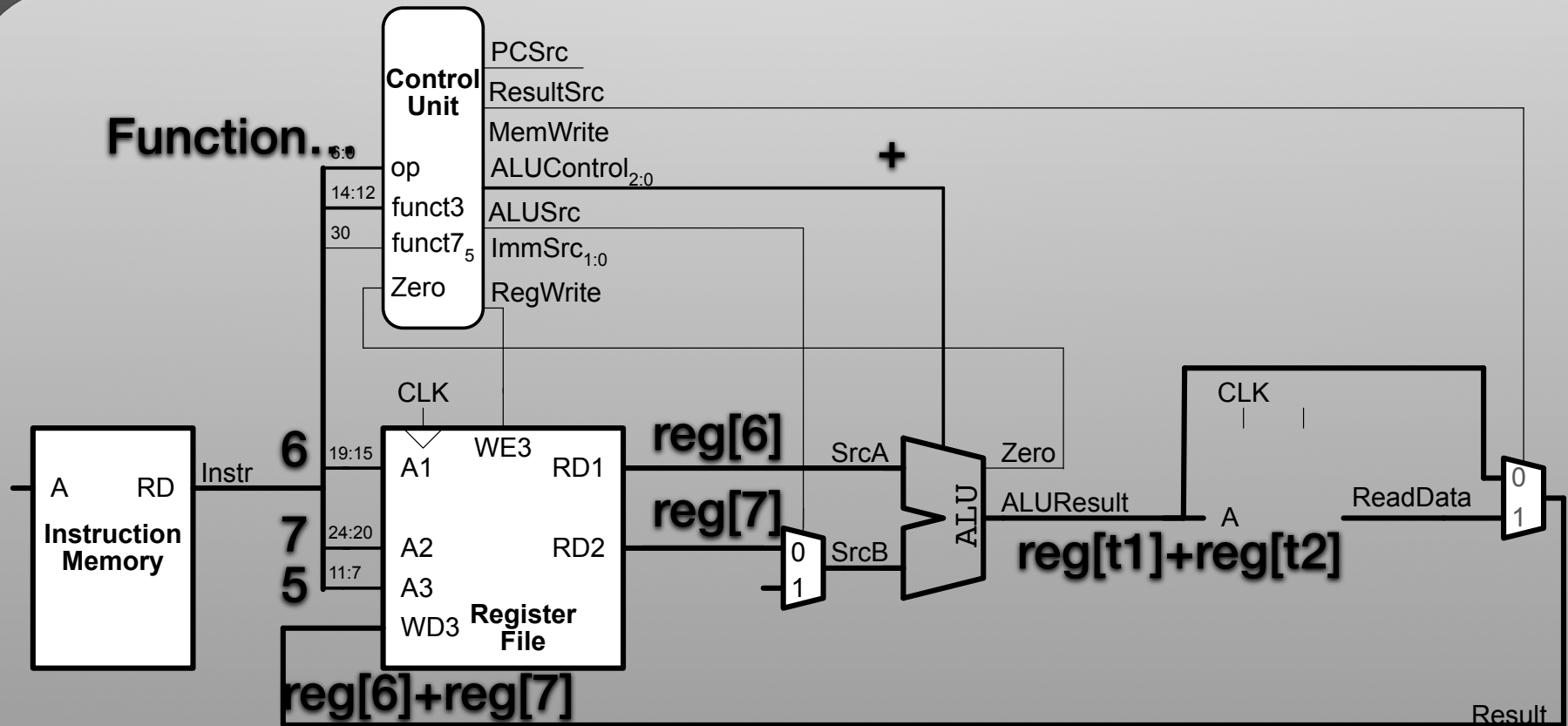
3. $a = 7$

4. $a = b$

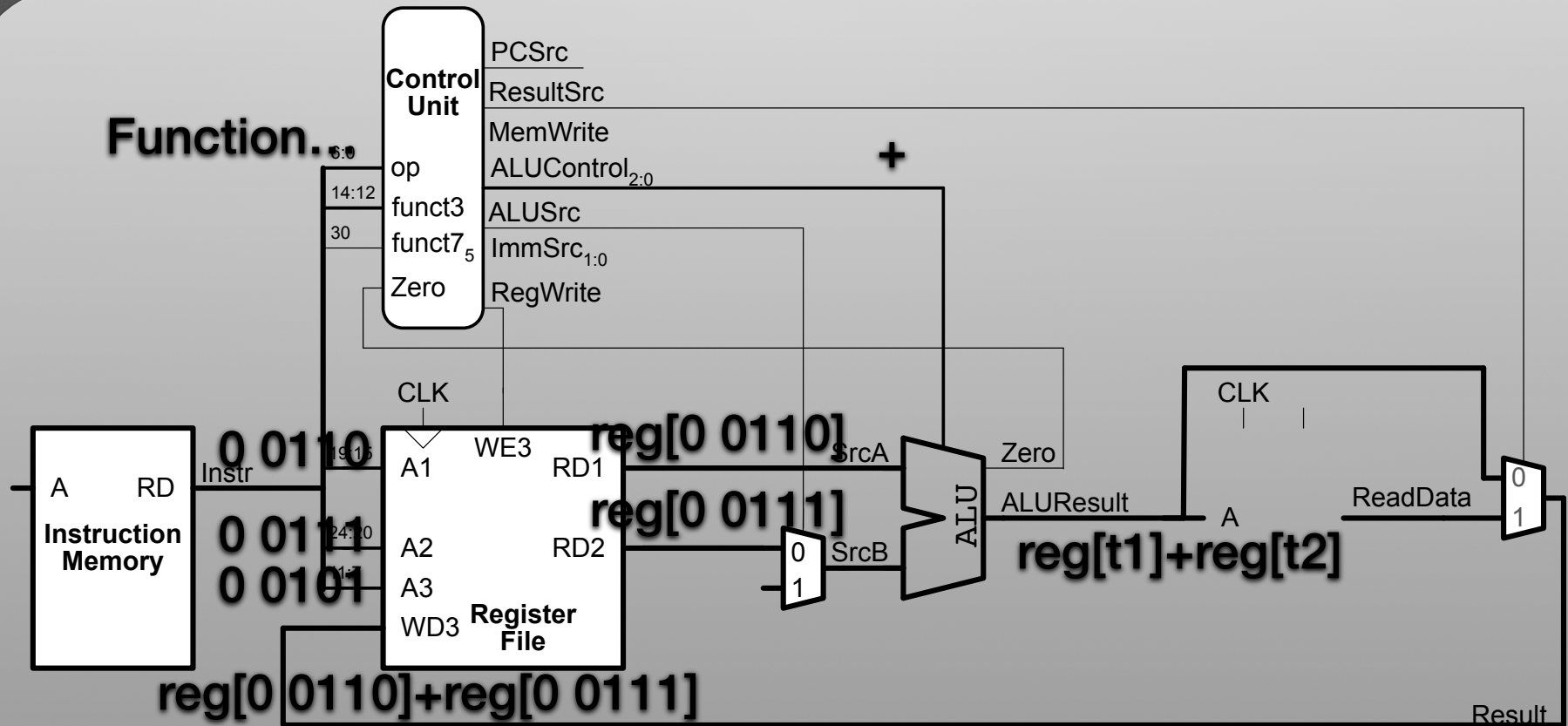
Big Picture: add t0, t1, t2 # (t0=t1+t2)



Big Picture: add t0, t1, t2

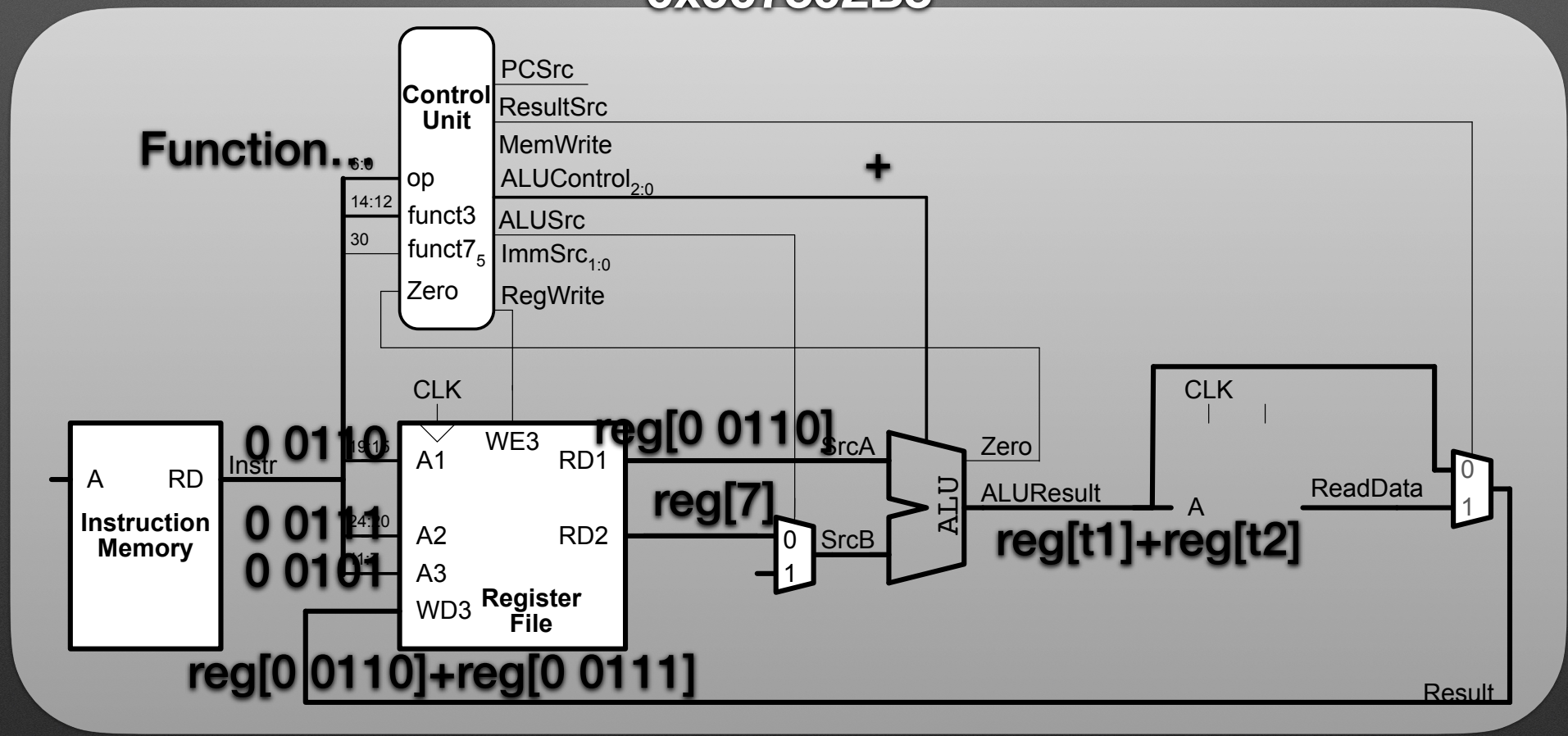


Big Picture: add t0, t1, t2



Big Picture: add t0, t1, t2

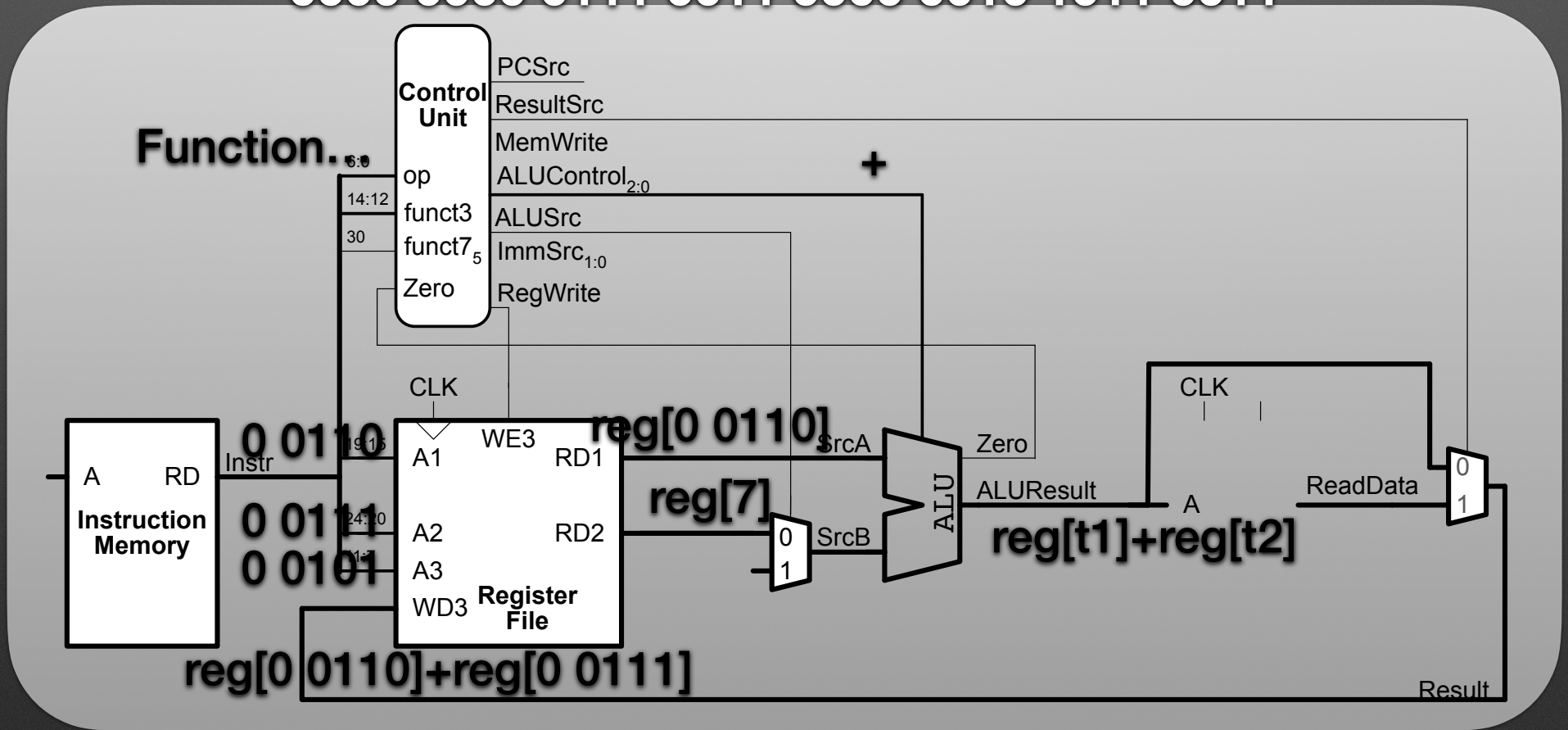
add x5 x6 x7
0x007302B3



Big Picture: add t0, t1, t2

add x5 x6 x7

0000 0000 0111 0011 0000 0010 1011 0011



Big Picture: add t0, t1, t2

add x5 x6 x7

0000 0000 0111 0011 0000 0010 1011 0011

Function. 6:0

Control Unit

PCSrc
ResultSrc
MemWrite
ALUControl_{2:0} +
ALUSrc
ImmSrc_{1:0}
RegWrite

op

14:12

func3

30

func7

Zero

CLK

A1

WE3

RD1

A2

RD2

A3

WD3

Register File

reg[0 0110]

reg[7]

0
1

ALU

ALUResult

reg[t1]+reg[t2]

CLK

A

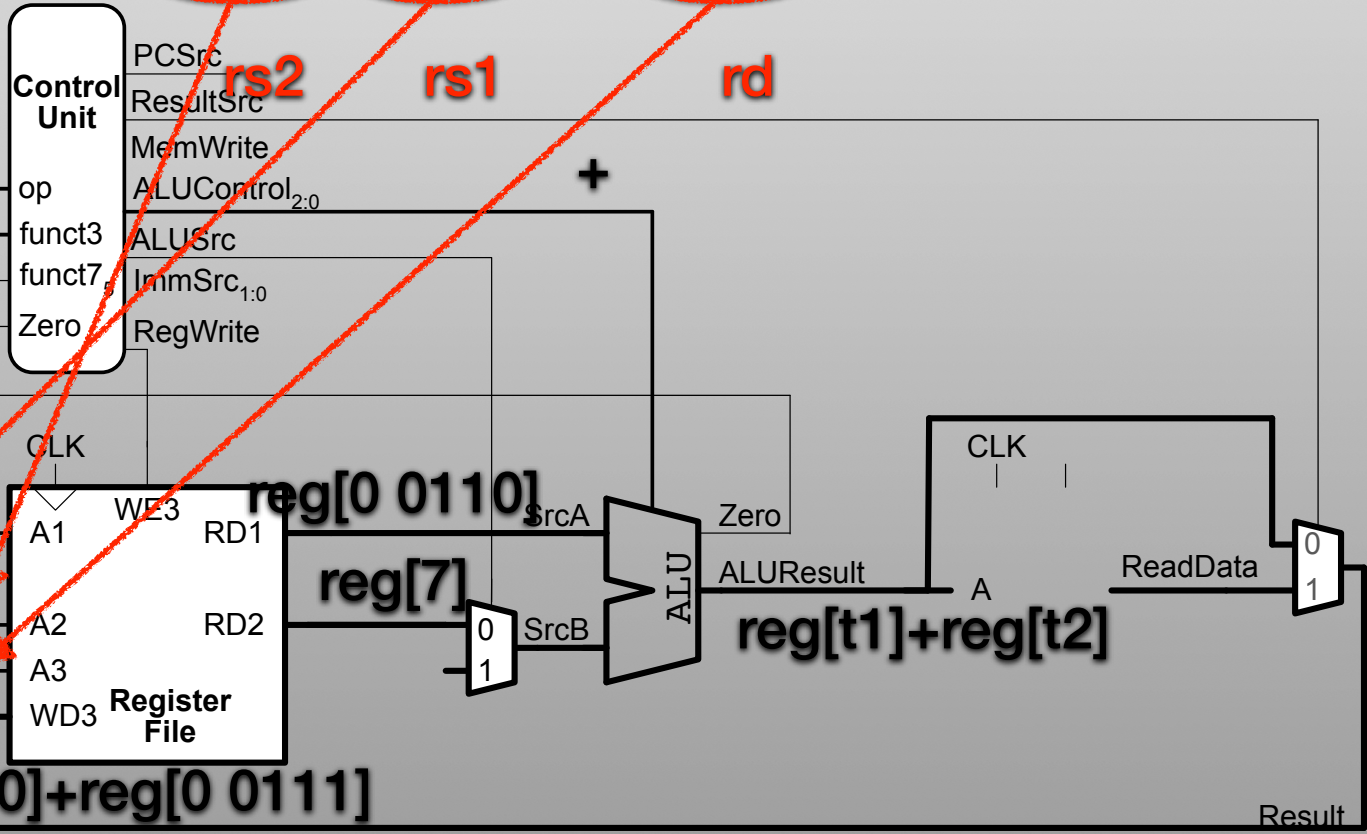
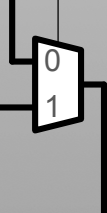
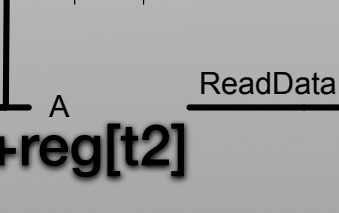
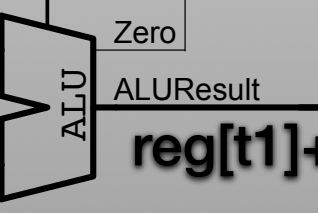
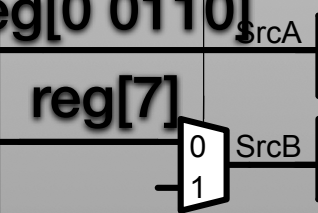
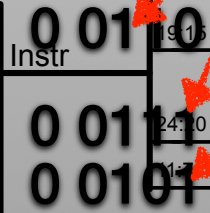
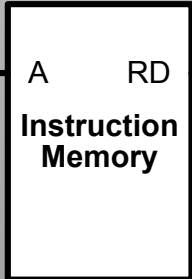
ReadData

0

1

Result

reg[0 0110]+reg[0 0111]



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 ...
```

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

```
loop_start_label:
    loop_check / jump to loop_end_label
```

```
    loop body (including increment)
    j loop_start_label
```

```
loop_end_label:
```

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 and here
    }
}
```

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 (Emory)
 - 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;
```

Pico RV-32 (& CSE 2600)



Next Time

- Studio

Questions

- Why Assembly Language?
 - Evolution of programming
- Formats? I vs. R? “Immediate”?
- Will we need to program assembly? Yes
- Will we need other memorize stuff? No exactly
- What’ll be on the exam?