RISC-V Single Cycle

Introduction to Computer Systems, Fall 2024

Instructors: Joel Ramirez Travis McGaha

Head TAs: Adam Gorka Daniel Gearhardt
Ash Fujiyama Emily Shen

TAs:

Ahmed Abdellah
Angie Cao
August Fu
Caroline Begg
Cathy Cao
Claire Lu
Eric Sungwon Lee
Ethan Weisberg
Garrett O'Malley Kirsch
Hassan Rizwan
Iain Li
Jerry Wang
Juan Lopez
Keith Mathe
Maya Huizar
Meghana Vasireddy
Perrie Quek
Sidharth Roy
Sydnie-Shea Cohen
Vivi Li
Yousef AlRabiah

pollev.com/tqm

- Any Questions?

Logistics

- Midterm: Soon[tm]
  - Edge cases need to be looked at, and a few files need to be rescanned
  - I hope tonight
- HW07 is out and due this Friday at midnight
  - Should have everything you need after this lecture
  - Autograder is out
  - Simulator is out
- HW08 comes out this Friday/weekend

Logistics Pt. 2

- Office Hours this week
  - No Office Hours on Halloween (Thursday)
  - Reduced TA presence at office hours the day after (Friday)
- Check-in06 out later this week
- Election happening soon
  - No lecture on Tuesday next week.

Lecture Outline

- **Von Neuman & Processor Start**
- RISCV Singe Cycle Processor
  - Decoder
  - Register File
  - ALU
  - Branch unit
  - The Rest
  - “Single Cycle”

This Lecture: Hardware/Software Interface

- We've looked at some hardware topics and some software topics (C & RISC-V)

- Today we are looking at the hardware/software interface for the RISC-V ISA
  - How does assembly run on hardware?
  - How do we create hardware that runs assembly code?

- Hardware details abstracted, uses a lot of the components previously talked about (Mux, Adders, Incrementors, etc.)
  - You will implement something like this in CIS 4710

HW/SW Interface

- Assembly is a middle ground between software and hardware

- Software:
  - It can still be very hard to read assembly code
  - More complex coding languages translates into assembly

- Hardware:
  - Hardware only needs to implement these “simple” instructions.
  - Hardware does not *need* to implement a custom “calculate the Fibonacci sequence” piece of hardware.
  - Instructions translate directly into binary that hardware can read.

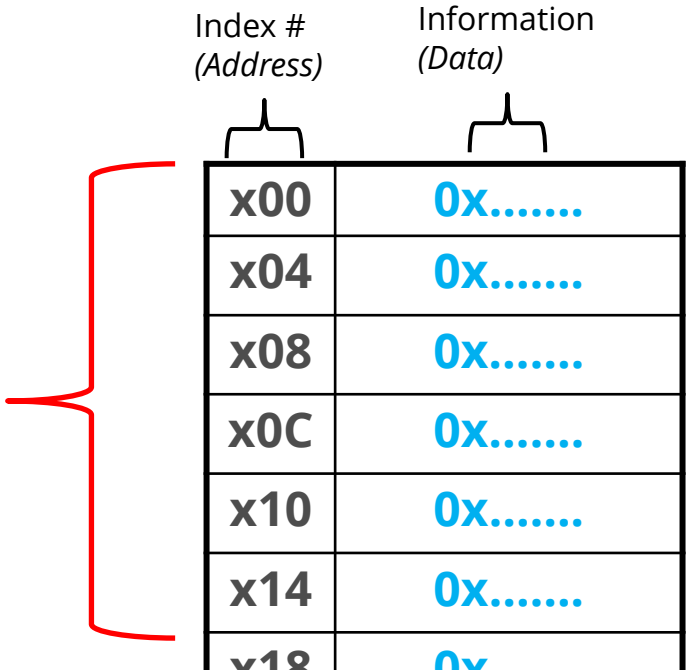
Reminder: Instructions are bits

- An instruction fits in 32-bits (4-bytes, 4 memory locations)

- These instructions are stored in memory and accessed sequentially
  - When we trace through the code, we are just accessing the next instruction in memory

```
li x0, #32
li x1, #16
li x2, #64

div x1, x2, x1
add x3, x3, x0
sub x0, x2, x3
```

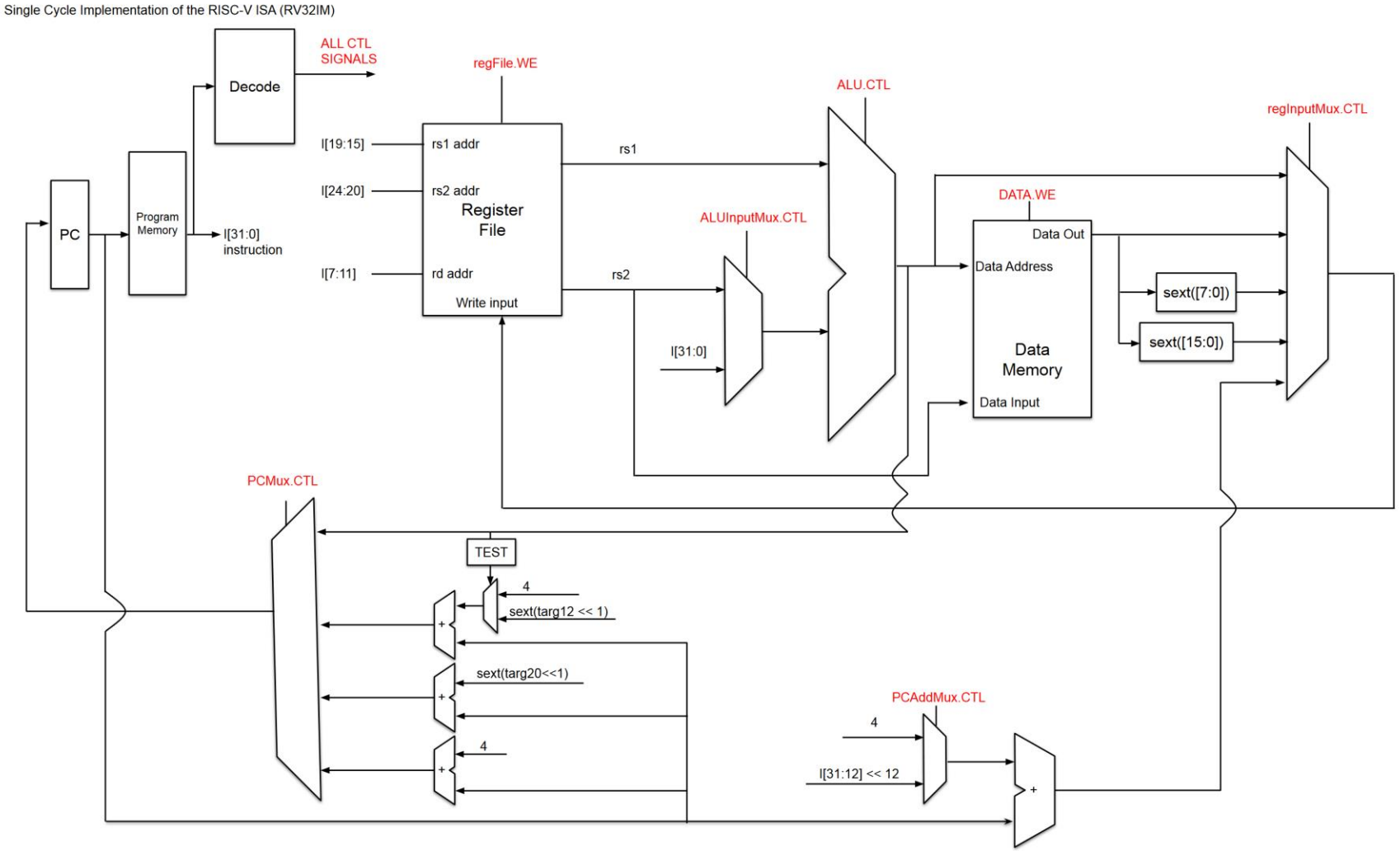


The Von Neumann Loop

- Von Neumann Processor essentially does:
  - Fetch instruction at Program Counter
  - Decode instruction
  - Execute instruction & Update PC
  - Repeat

- Critical Requirement
  - Each iteration of this loop must appear **atomic** (All or nothing)
  - Key word from programmer perspective: atomic
    - Maintains sanity
  - Key word from hardware perspective: appear
    - Enables hardware to perform various tricks for performance >:]

An Idea of what we are doing :



More RISC-V References!

- More RISC-V References added to the course website
- Highly recommend you print out a copy of the "Control Signals Description" handout

- RISC-V Single Cycle Processor is the diagram on the previous slide, may also want to print this

Lecture Outline

- Von Neuman & Processor Start
- **RISCV Singe Cycle Processor**
  - **Decoder**
  - **Register File**
  - **ALU**
  - **Branch unit**
  - **The Rest**
  - **“Single Cycle”**

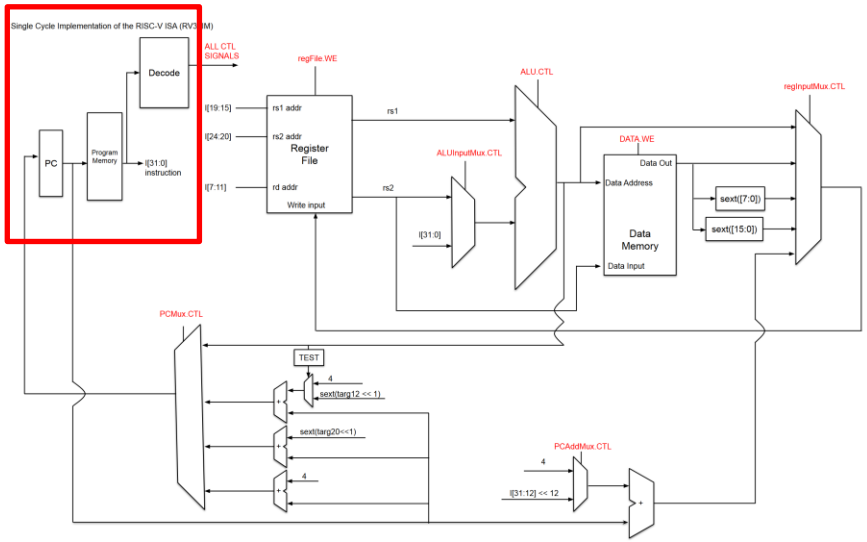
Aside: bit selecting syntax

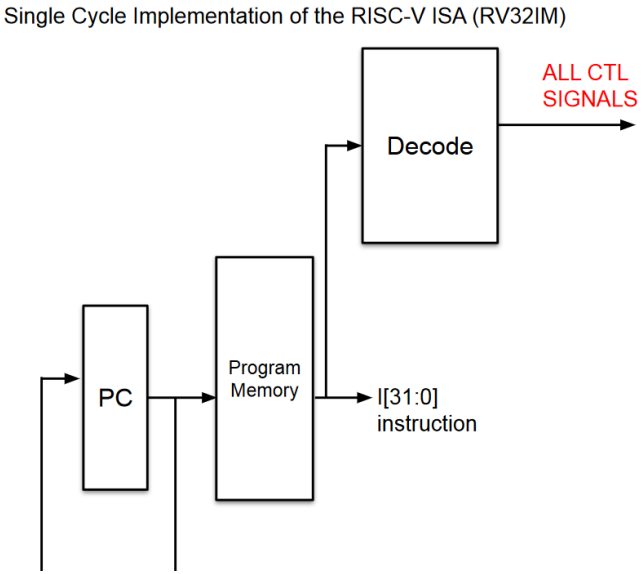
- Assume we have a 32-bit pattern called X
  - X: 00000000000101010000001010010011

- We can refer to a specific subsection of X with the syntax X[n:m]
  - X[31:0] // all 31 bits
  - X[2:0] // 3 least most significant bits
  - X[19:15] // 5 bits in the middle

Fetch & Decode

- First & second step: Fetch an instruction and decode it
  - Read instruction at PC in memory (stored as 32 bits)
  - **From those 32-bits, outputs signals to control the processor to execute the instruction.**
  - Common exam question: implement part of the decoder with logic gates.

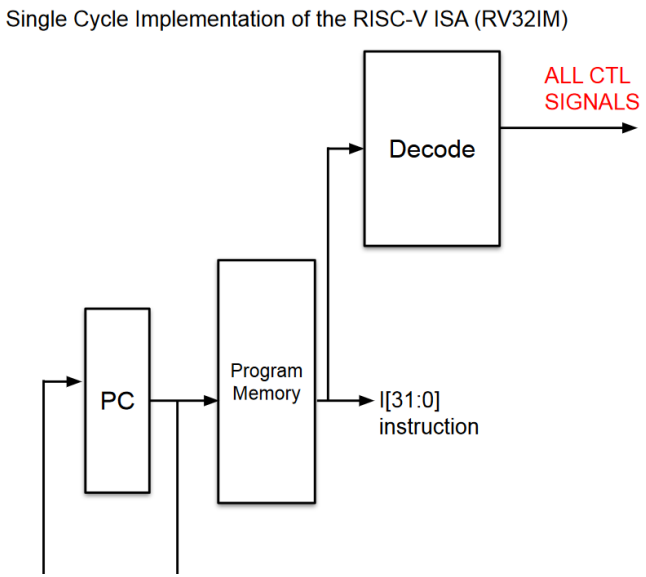




Fetch & Decode: ADDI x5, x10, 1

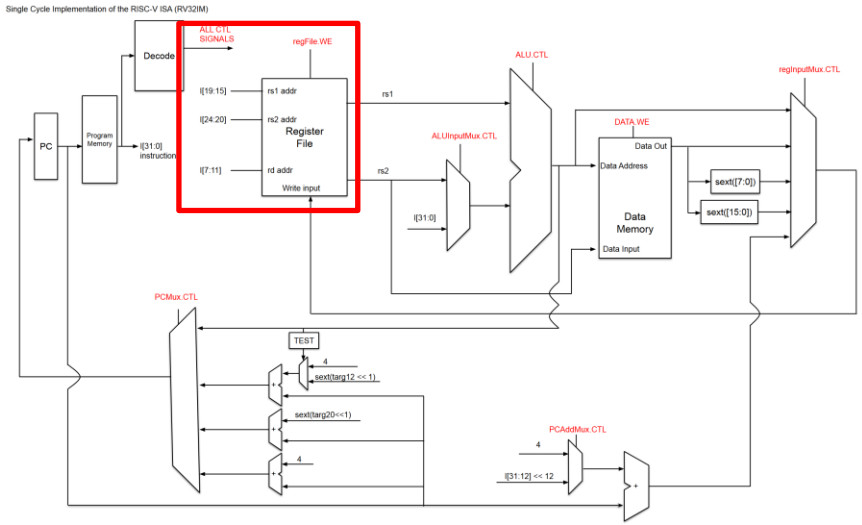
- Throughout this lecture, we will assume we just fetched the instruction `ADDI x5, x10, 1` and decide what the control signals for this should be

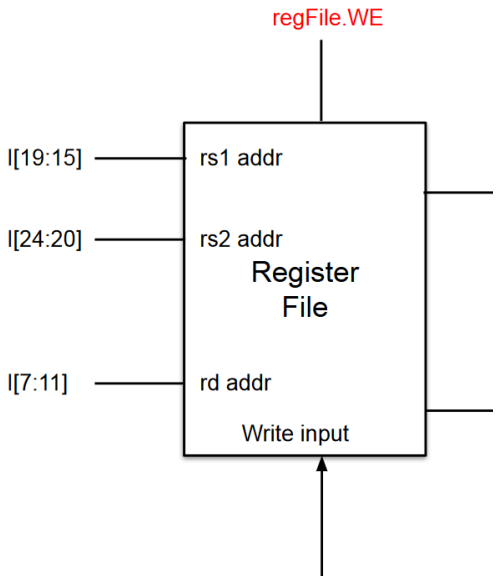
- We have fetched and decoded the instruction, now we must execute it



Register File

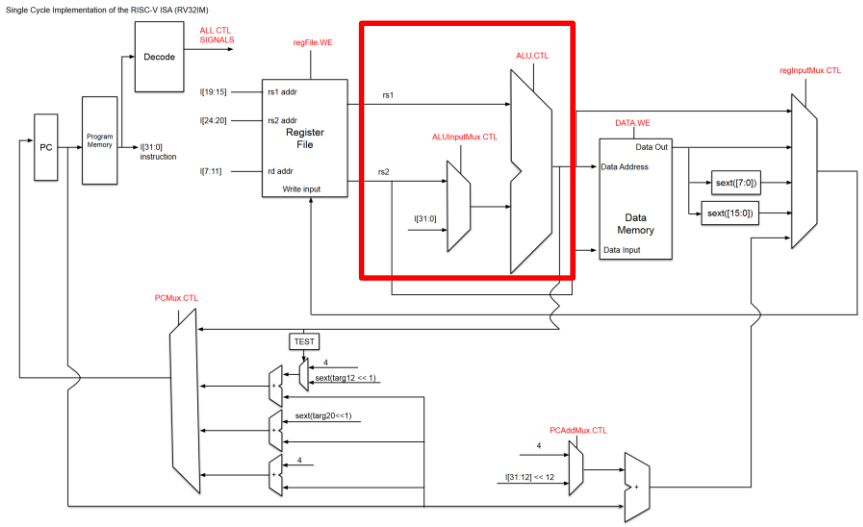
- Array of the 32 general purpose processor registers x0 - x32
- The registers used are always at the same position in the instruction encoding.
- Not a typical "File" on a computer
  - (We will talk about regFile.WE later this lecture)

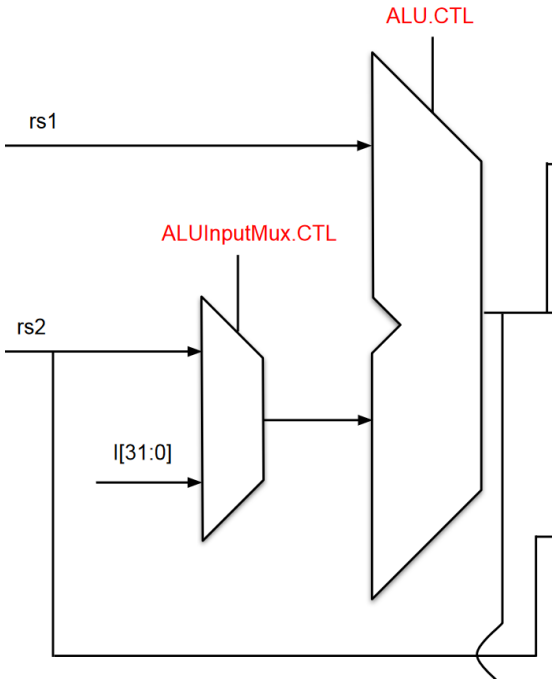




ALU: Arithmetic Logic Unit

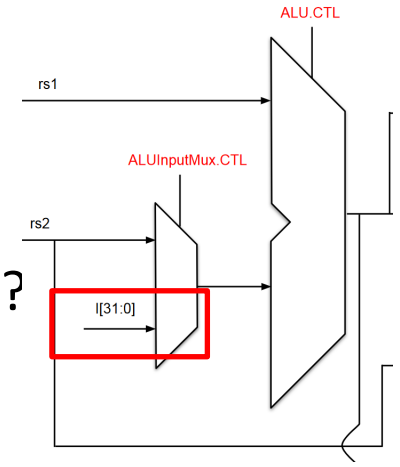
- Performs Arithmetic and Logical operations
  - Where most instructions perform their "work"
- Use Control Signals to decide
  - what operation is performed
  - what the inputs are





ALU Input Mux: ADDI x5, x10, 1

- ALUInputMux.CTL
  - This signals decide what should be used as input for the ALU (thing that does most arithmetic / logical operations)
- **How to decide signals generally**:
  - Look at the options available for this control signal (Single Cycle handout or Control Signal Description handout)
  - Determine which signal matches up for the current instruction



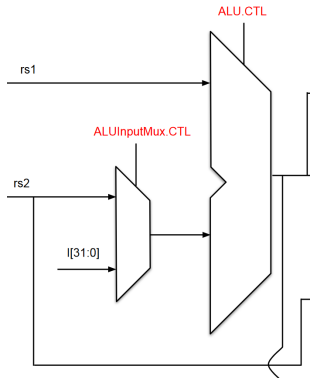
- ADDI Example:
  - Do we use an rs2 or an immediate for ADDI?

```
iiii iiii iiii ssss s000 dddd d001 0011
```

  - From the encoding directly
  - so AluInputMux.CTL is 1

Control Signals Description Handout

- Can use the Control Signals Description Handout to look up signals
- ADDI: Do we use an rs2 or do we use the bits from the instruction encoding?



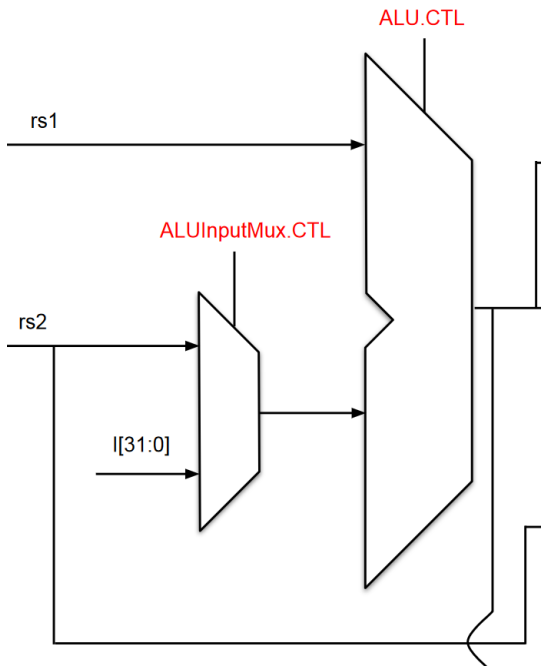
Signal Name	# of bits	value	action
ALUInputMux.CTL	2	0	Second input to ALU is rs2
		1	Second input to ALU is the 32 bits of the instruction

- Then ALUInputMux.CTL should be 1

ALU: ADDI x5, x10, 1

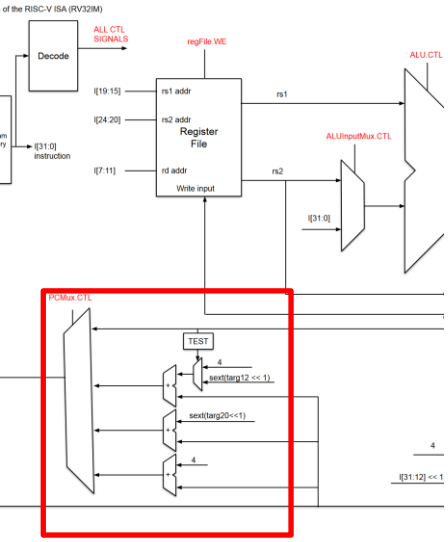
- ALU.CTL decides which arithmetic/logical operation to perform.
  - 36 different options: look at the control signals description sheet
- ADDI operation is C = A + B, but B is sign extended from the immediate, So ALU.CTL is 0

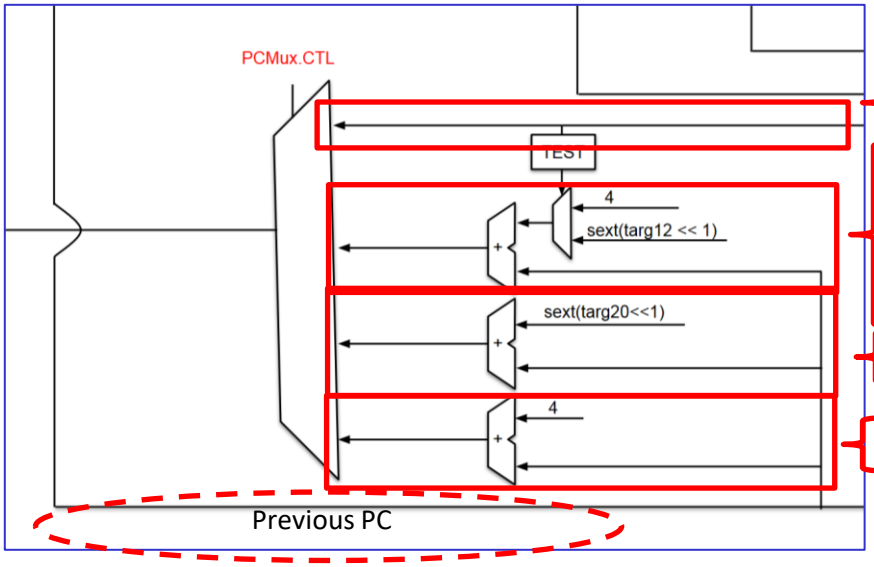
Signal Name	# of bits	value	action
ALU.CTL	6		
Arithmetic Ops (I)		0	C = A + se(B[31:20])
		1	C = A < se(B[31:20]) ? 1 : 0
		2	C = A < unsigned se(B[31:20]) ? 1 : 0
		3	C = A ^ se(B[31:20])
		4	C = A | se(B[31:20])
		5	C = A & se(B[31:20])
		6	C = A << se(B[24:20])
		7	C = A >> se(B[24:20])
		8	C = A >>> se(B[24:20])



Branch Unit

- Updates PC
- PCMux.CTL: decides how PC is updated





0: PC is output of ALU

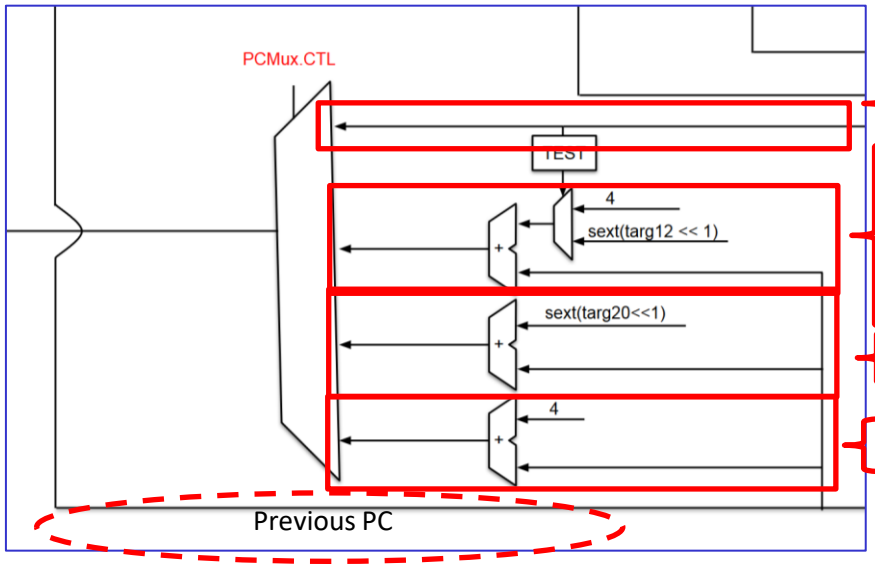
1: test then
PC = PC + 4
or
PC = PC + se(TARG12)

2: PC += se(TARG20 << 1)

3: PC = PC + 4

Branch Unit: ADDI x5, x10, 1

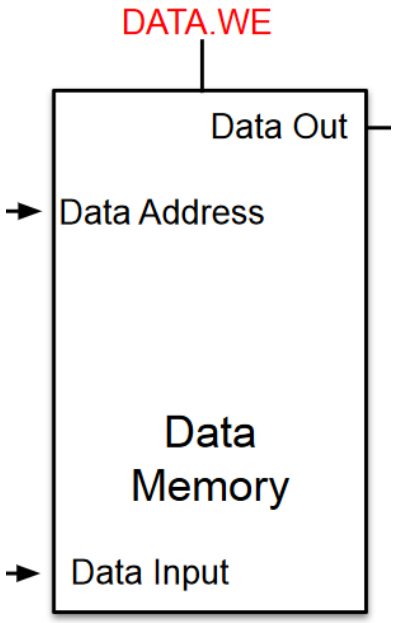
- How does ADDI x5, x10, 1 update the PC?
  - PC = PC + 4 so, PCMux.CTL is 3



0: PC is output of ALU
1: test then
PC = PC + 4
or
PC = PC + se(TARG12)
2: PC += se(TARG20 << 1)
3: PC = PC + 4

Data Memory

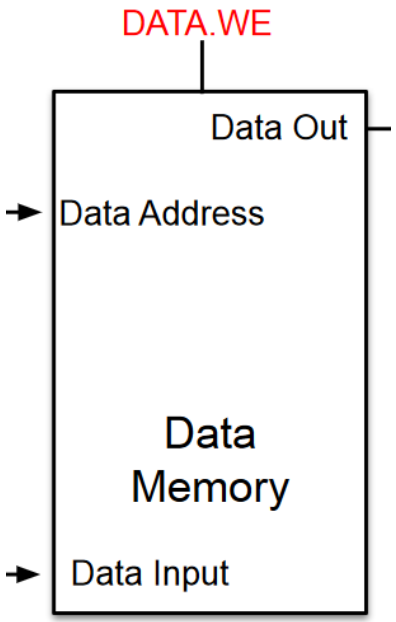
- Contains the data memory of our program
- Takes in the:
  - Address of the data to access
  - What data to write at that address
- Outputs the data at the specified address
- DATA.WE decides if we are updating any data in memory.
  - Does ADDI update any data in memory?
  - No, so DATA.WE for ADD is 0



Data Memory (cont.)

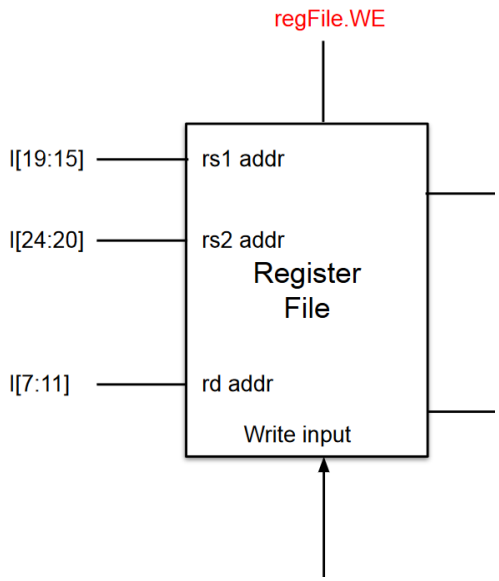
- DATA.WE decides if we are updating any data in memory.
  - Not shown with ADDI instruction: DATA.WE is 4 bits.

- Why is DATA.WE 4-bits?
  - We can write 0, 1, 2, or 4-bytes to data memory.
  - Each byte we could write will have "its" own WE bit to determine if we write that byte.
  - E.g. if we wanted to write all 4-bytes of data input, then DATA.WE would be 0b1111 (15)



Register File: regFile.WE

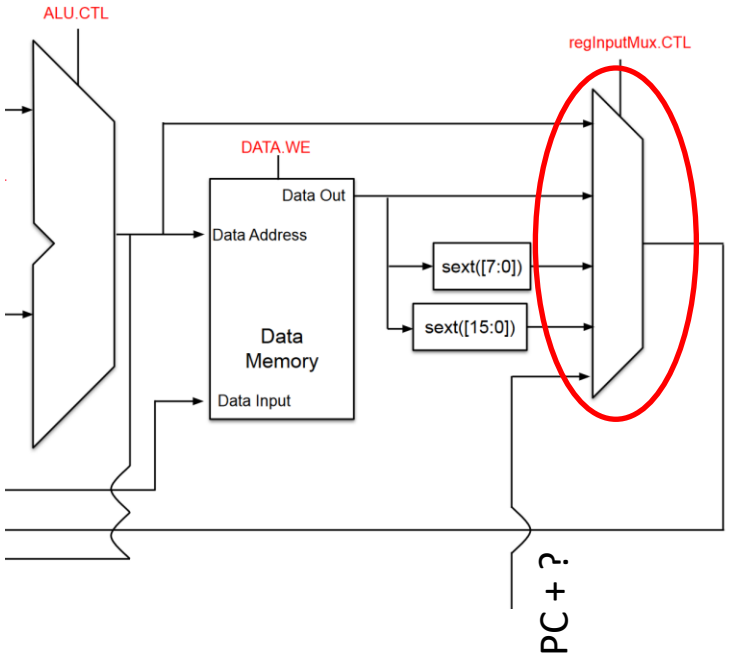
- regFile.WE: write-enable for the register file. If we are writing to a register it should be 1, 0 otherwise
  - ADD writes to a register, so regFile.WE is 1



regInputMux.CTL

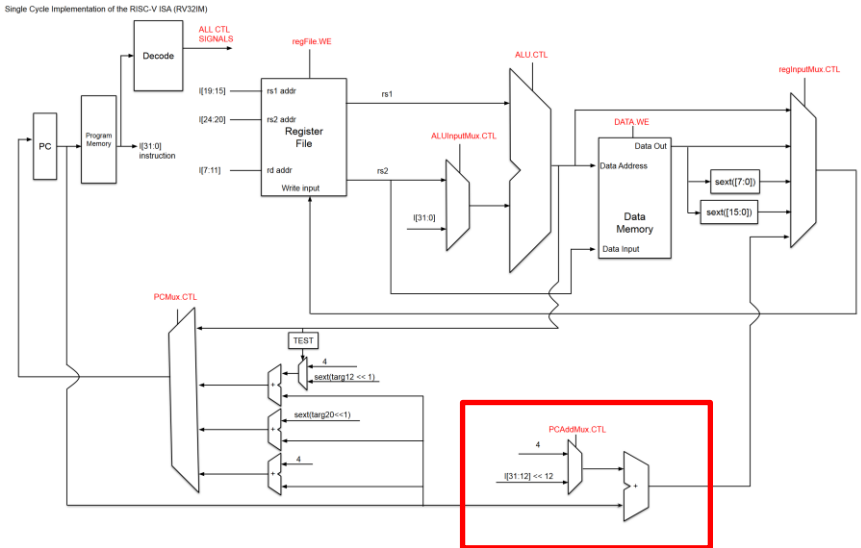
- Decides what gets written back to the register file
  - 0 = output of ALU
  - 1 = 4-bytes of data memory
  - 2 = 1-byte of data memory
  - 3 = 2-bytes of data memory
  - 4 = PC + <something>

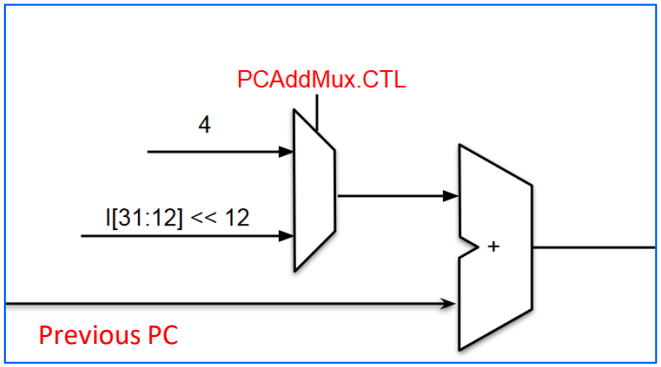
- What does ADDI store into a register?
  - Output of ALU, so for ADDI regInputMux.CTL = 0



PCAddMux.CTL

- When we store the PC to the register file, we also have to add something to it. This decides what we add to the PC value that would get written back to the register file

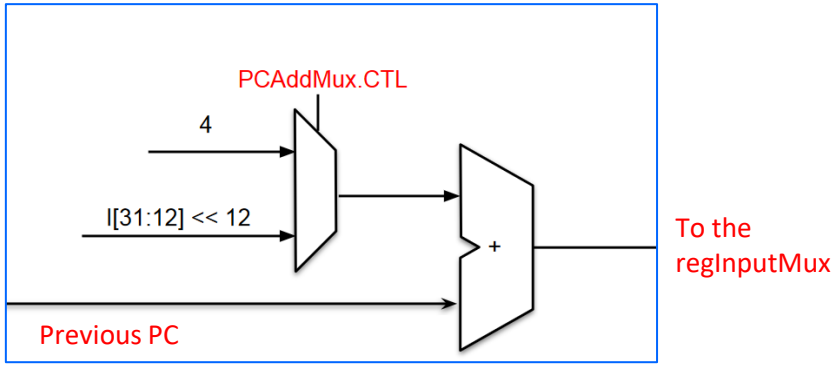




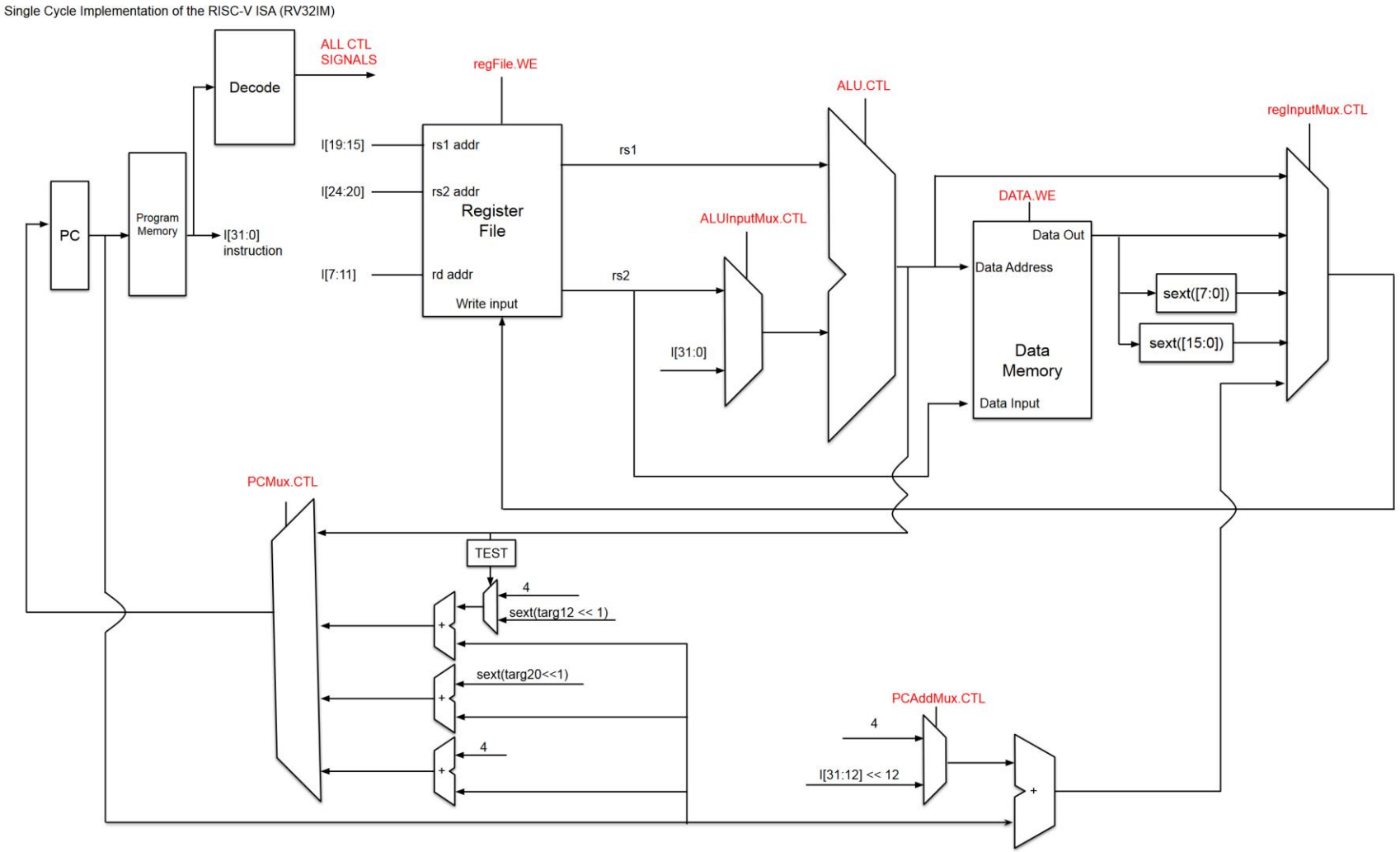
To the regInputMux

PCAddMux.CTL

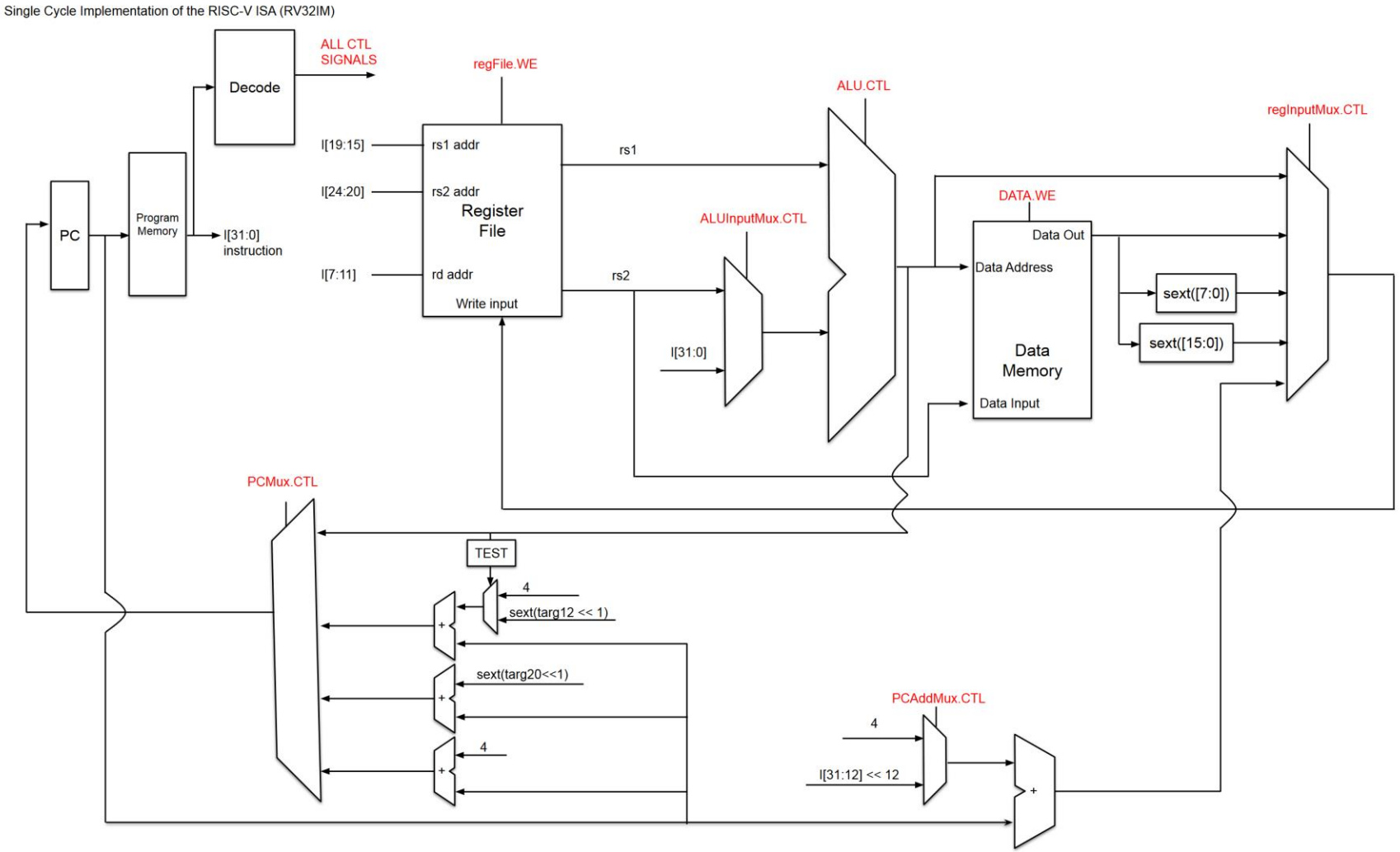
- When we store the PC to the register file, we also have to add something to it. This decides what we add to the PC value that would get written back to the register file
- What does ADDI do with the PC value stored to register file?
  - **It doesn't store the PC into the register file.**
  - **If we look at the diagram given our other signals, the output of this is unused.**
  - **We can mark this signal as X (unused) for the ADD instruction**
  - **Will not always be X, sometimes it will be 0 or 1.**
  - **Other signals can be X too**



The Complete Picture

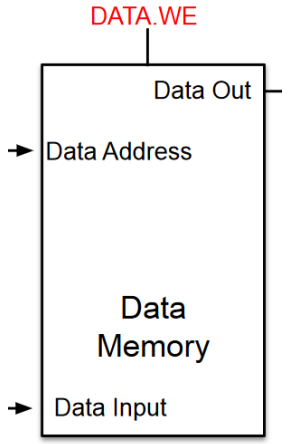


Questions?


Single Cycle Implementation of the RISC-V ISA (RV32IM)
Decode
ALL CTL SIGNALS
regFile.WE
ALU.CTL
regInputMux.CTL
PC
Program Memory
I[31:0] instruction
I[19:15]
I[24:20]
I[7:11]
rs1 addr
rs2 addr
Register File
rd addr
Write input
rs1
rs2
ALUInputMux.CTL
I[31:0]
DATA.WE
Data Out
Data Address
Data Memory
Data Input
sext([7:0])
sext([15:0])
PCMux.CTL
TEST
4
sext(targ12 << 1)
sext(targ20<<1)
4
PCAddMux.CTL
4
I[31:12] << 12
+

Reminder: Circuits are not Code

- We are dealing with circuits, not software
  - **All components are "working" all the time.**
  - **We may not be using their output all the time though.**
- WE Signals always matter, we never "don't care" about them
  - Example: ADDI and DATA.WE
    - ADDI doesn't use data memory at all, but the data address and data input will still be some value (which may be garbage)
    - We do NOT want to write garbage to memory so DATA.WE should be 0



pollev.com/tqm

- What are the control signals for the LUI instruction?
  - 7 different control signals questions on PollEv
- Probably want to pull up the Control Signals Description, RISC-V Encoding, and Single Cycle Sheet
- If you are reading the slides after lecture and want to go over this, should probably watch the lecture recording

"Single Cycle"

- This whole Lecture I've been talking about processor with the term "Single Cycle"
  - This means that one instruction is executed in one clock cycle.
  - That means the length of the program is directly proportional to the number of instructions executed

- "Single Cycle" is a convenient way for programmers to think about the processor, but most current processors are **not** like this
  - More in CIS-5710 (Or a special topic at the end of the semester)

pollev.com/tqm

- Any Questions On Registration?