LC4 Single Cycle Processor

Introduction to Computer Systems, Fall 2022

Instructor: Travis McGaha

TAs:

Ali Krema	Andrew Rigas	Anisha Bhatia
Audrey Yang	Craig Lee	Daniel Duan
David LuoZhang	Eddy Yang	Ernest Ng
Heyi Liu	Janavi Chadha	Jason Hom
Katherine Wang	Kyrie Dowling	Mohamed Abaker
Noam Elul	Patricia Agnes	Patrick Kehinde Jr.
Ria Sharma	Sarah Luthra	Sofia Mouchtaris

How was fall break?

Logistics

- Check-in04: **Due before lecture on Wednesday**

- HW04 LC4 Programming: **This Friday** 10/14 @ 11:59 pm
  - Should have everything you need
  - Practice in Recitations this week
  - Normal programming assignment ☺

- HW05 Control Signals: to be released this Friday
  - Programming assignment

Lecture Outline

- **Von Neuman & Processor Start**
- LC4 Singe Cycle Processor
  - Decoder
  - Register File
  - ALU
  - Branch unit
  - The Rest
  - “Single Cycle”

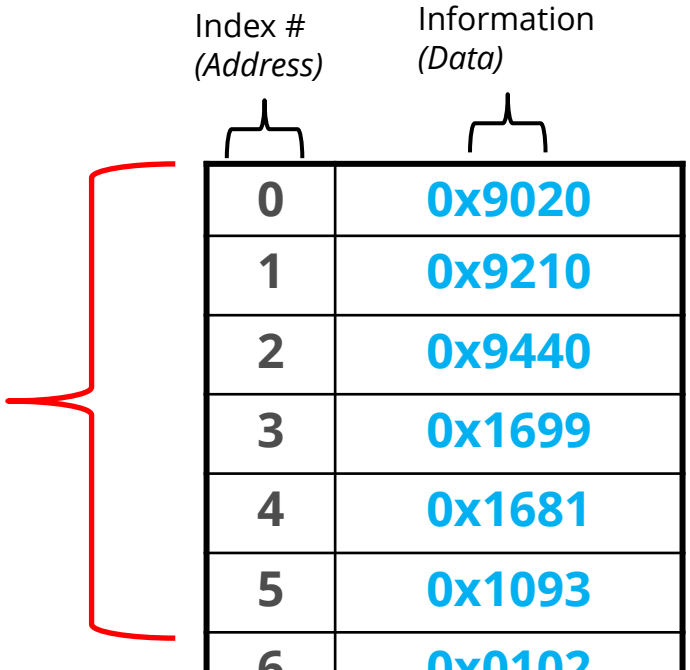
Reminder: Instructions are bits

- An instruction fits in 1 memory location (16 bits)

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

```
CONST R0, #32
CONST R1, #16
CONST R2, #64

DIV R3, R2, R1
ADD R3, R3, R0
SUB R0, R2, R3
```



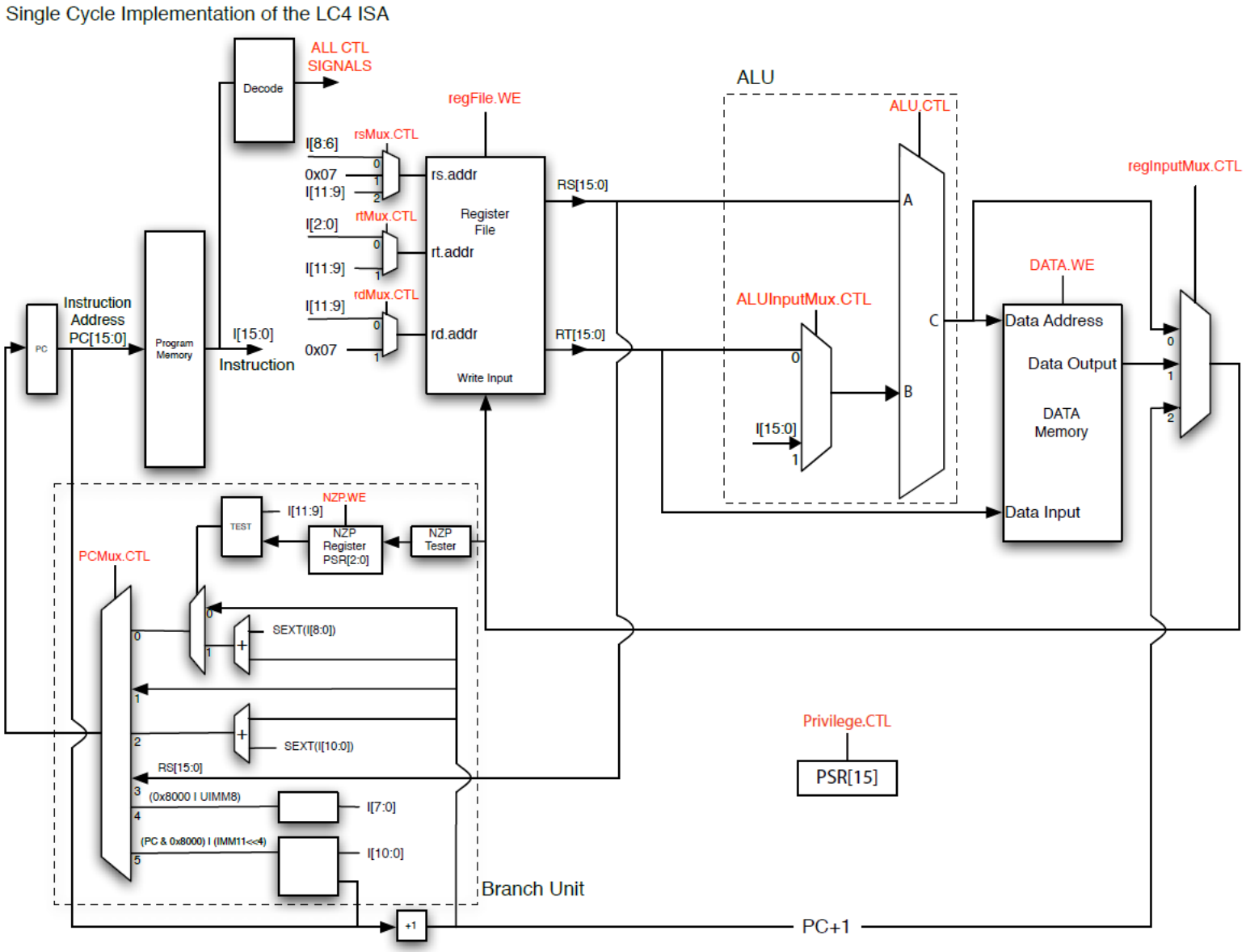
This Lecture: Hardware/Software Interface

- We've looked at some hardware topics and some software topics (LC4 assembly)

- Today we are looking at the hardware/software interface for the LC4 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

An Idea of what we are doing this lecture:



More LC4 References!

- More LC4 References added to the course website
- Highly recommend you print out a copy of the "Control Signals Description" handout

- LC4 Single Cycle Processor is the diagram on the previous slide

- ALU Internals explains slightly more detail about the ALU than I will cover

Aside: bit selecting syntax

- Assume we have a 16-bit pattern called X
  - X: 0001000001000101

- We can refer to a specific subsection of X with the syntax X[n:m]
  - X[15:0] // all 16 bits
  - X[2:0] // 3 least most significant bits
  - X[5:4] // 2 bits in the middle

The Von Neumann Loop

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

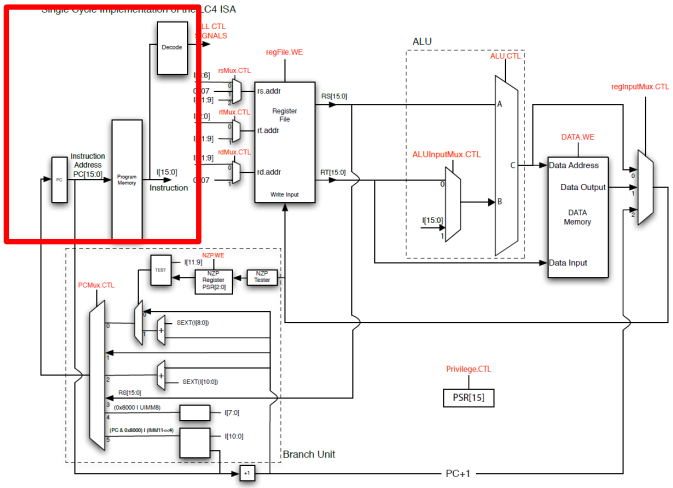
- Critical Requirement
  - Each iteration of this loop must appear **atomic** (All or nothing)
  - Key word from programmer perspective: atomic
    - Maintains sanity
  - Key word from hardware perspective: appear
    - Enables hardware to perform various tricks for performance >:]

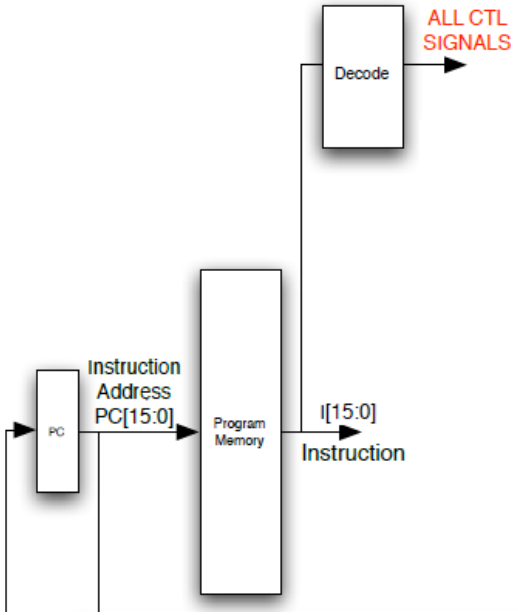
Lecture Outline

- Von Neuman & Processor Start
- **LC4 Singe Cycle Processor**
  - **Decoder**
  - **Register File**
  - **ALU**
  - **Branch unit**
  - **The Rest**
  - **“Single Cycle”**

Fetch & Decode

- First & second step: Fetch an instruction and decode it
  - Read instruction at PC in memory (stored as 16 bits)
  - **From those 16-bits, outputs signals to control the processor to execute the instruction.**
  - Common exam question: implement part of the decoder with logic gates.

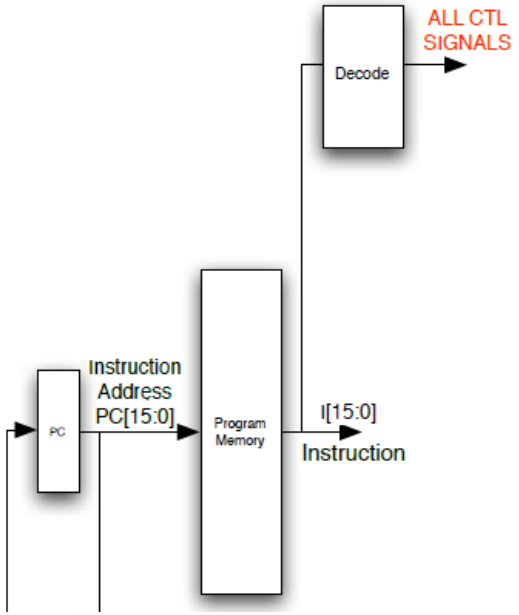




Fetch & Decode: ADD R0, R1, R5

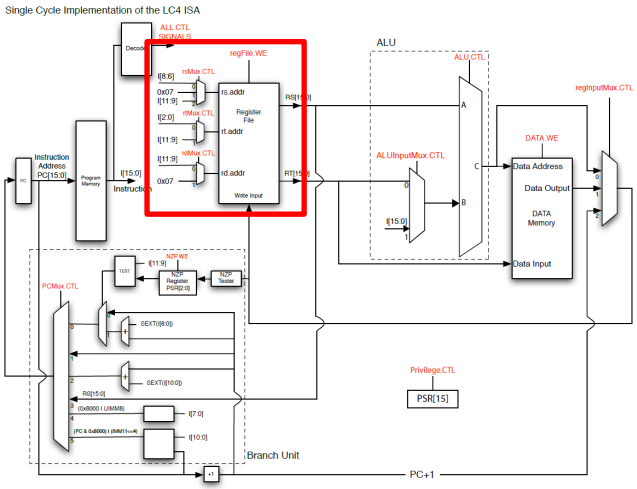
- Throughout this lecture, we will assume we just fetched the instruction ADD R0, R1, R5 and decide what the control signals for this should be

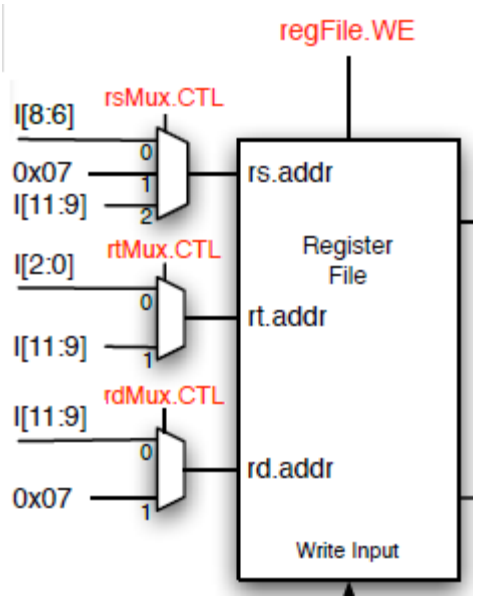
- We have fetched and decoded the instruction, now we must execute it



Register File

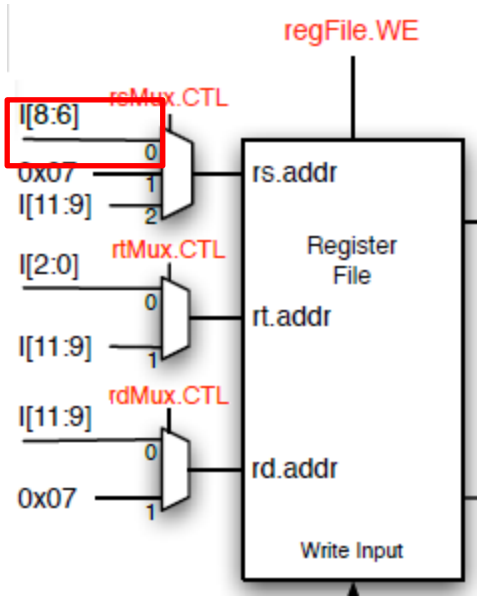
- Array of the 8 general purpose processor registers R0 - R7
- We use control signals to decide what registers we are using and if we are writing to the register file
- Note the usage of a MUX to select register addresses
- Not a typical “File” on a computer





Register File: ADD R0, R1, R5

- rsMux.CTL, rtMux.CTL, rdMux.CTL
  - These signals decide which register should be Rs, Rt, and Rd respectively.
- **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



- ADD Example:
  - What is used for Rs in ADD?

    0001 ddds ss00 0ttt
  - I[8:6], so rsMux.CTL is 0

Control Signals Description Handout

- Can use the Control Signals Description Handout to look up signals
- If ADD has Rs chosen by I[8:6]

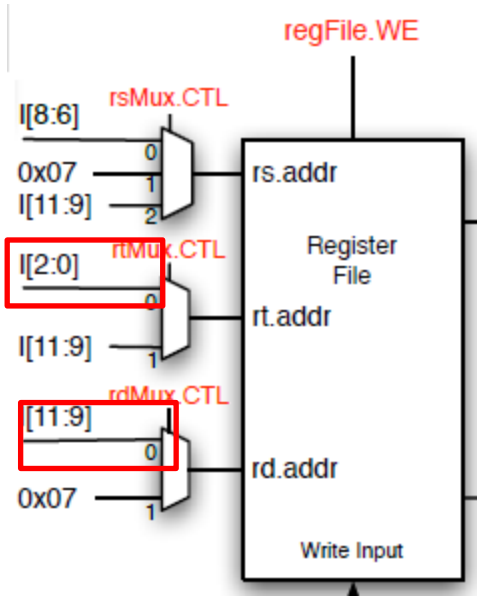
Signal Name	# of bits	Value	Action
PCMux.CTL	3	0	Value of NZP register compared to bits I[11:9] of the current instruction if the test is satisfied then the output of TEST is 1 and NextPC = BRANCH Target, (PC+1) + SEXT(IMM9); otherwise the output of TEST is 0 and NextPC = PC + 1
		1	Next PC = PC+1
		2	Next PC = (PC+1) + SEXT(IMM11)
		3	Next PC = RS
		4	Next PC = (0x8000 | UIMM8)
		5	Next PC = (PC & 0x8000) | (IMM11 << 4)
rsMux.CTL	2	0	rs.addr = I[8:6]
		1	rs.addr = 0x07
		2	rs.addr = I[11:9]

- Then rsMux.CTL should be 0

Register File: ADD R0, R1, R5

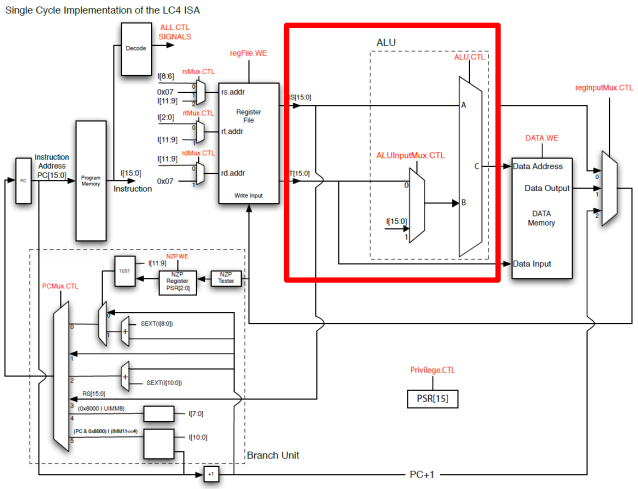
- rsMux.CTL, rtMux.CTL, rdMux.CTL
  - These signals decide which register should be Rs, Rt, and Rd respectively.
- ADD Example:
  - What is used for Rd and Rt in ADD?

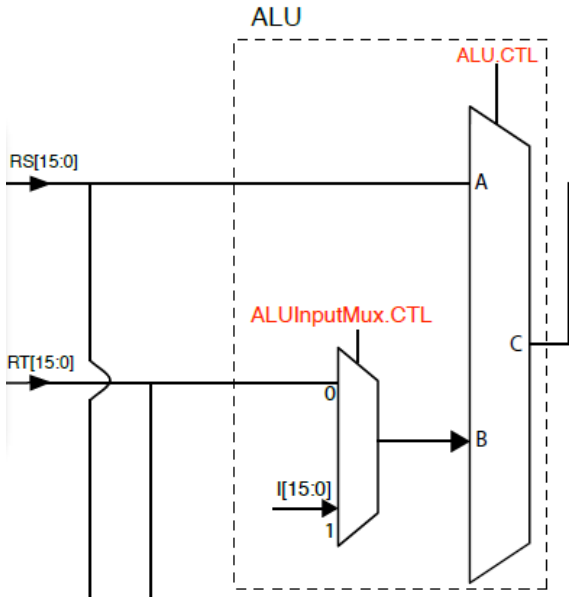
    0001 ddds ss00 0ttt
  - I[11:9] for Rd, so rdMux.CTL is 0
  - I[2:0] for Rt, so rtMux.CTL is 0
- regFile.WE: write-enable for the register file. If we are writing to a register it should be 1, 0 otherwise
  - ADD writes to a register, so regFile.WE is 1



ALU: Arithmetic Logic Unit

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

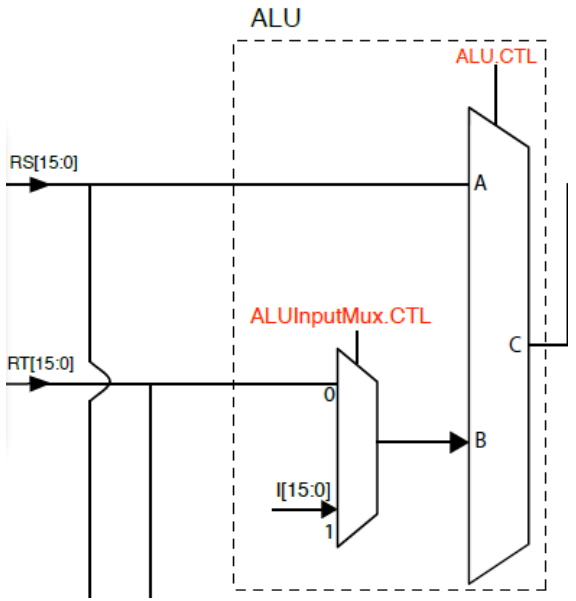




ALU: ADD R0, R1, R5

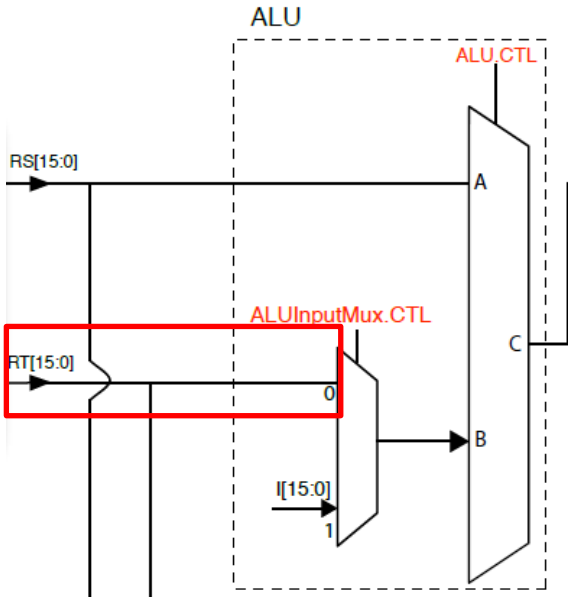
- ALU.CTL decides which arithmetic/logical operation to perform.
  - 21 different options: look at the control signals description sheet
- ADD operation is C = A + B with all 16 bits form each, no Sign EXTension, etc. So ALU.CTL is 0

Signal Name	# of bits	Value	Action
ALU.CTL	6		
Arithmetic Ops		0	C = A + B : Addition
		1	C = A * B : Multiplication
		2	C = A - B : Subtraction
		3	C = A / B : Division
		4	C = A % B : Modulus
		5	C = A + SEXT(B[4:0])
		6	C = A + SEXT(B[5:0])
Logical Ops		8	C = A AND B : Bitwise Logical Product
		9	C = NOT A: Bitwise Negation
		10	C = A OR B: Bitwise Logical Sum
		11	C = A XOR B: Bitwise Exclusive OR
		12	C = A AND SEXT(B[4:0])
Comparator Ops		16	C = signed-CC(A-B) [-1, 0, +1]
		17	C = unsigned-CC(A-B) [-1, 0, +1]
		18	C = signed-CC(A-SEXT(B[6:0])) [-1, 0, +1]
		19	C = unsigned-CC(A-SEXT(B[6:0])) [-1, 0, +1]
Shifter Ops		24	C = A << B[3:0] : Shift Left Logical
		25	C = A >>> B[3:0] : Shift Right Arithmetic
		26	C = A >> B[3:0] : Shift Right Logical
Constant Ops		32	C = SEXT(B[8:0])
		33	C = (A & OxFF) | (B[7:0] << 8)



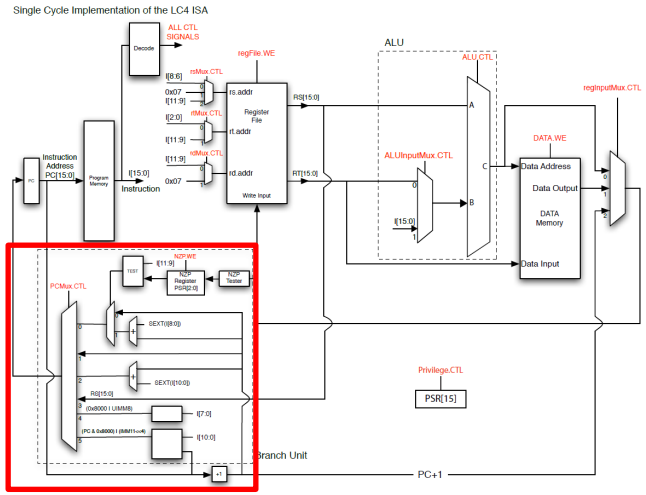
ALU: ADD R0, R1, R5

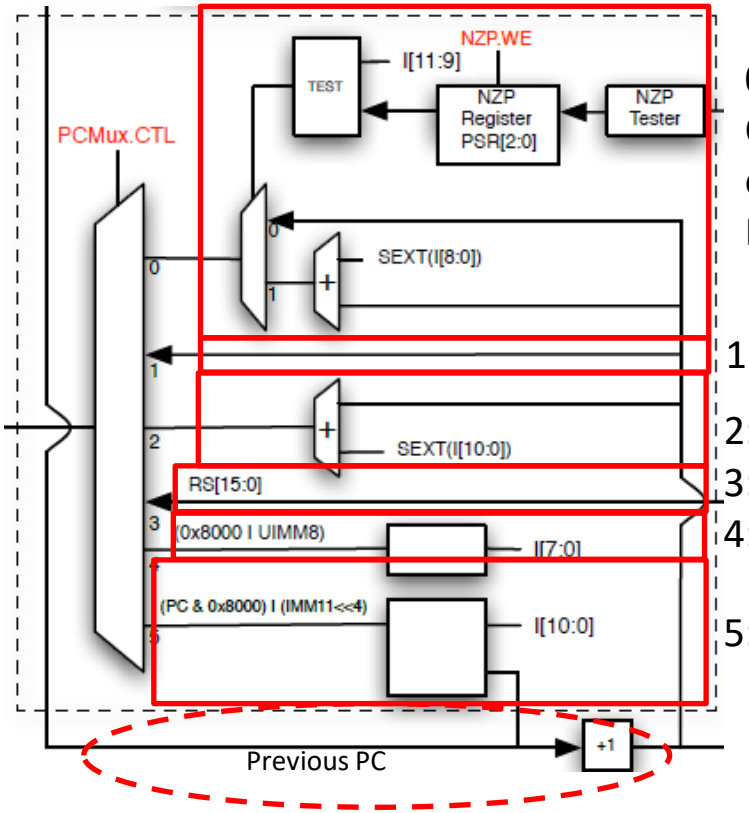
- ALUInputMux.CTL decides what the second input to the ALU will be.
  - The first input is always the 16 bit value stored in Rs
  - The second value can either be:
    - the value in Rt
    - the 16-bits of the instruction encoding
- ADD R0, R1, R5 uses Rt so ALUInputMux.CTL is 0
- If we executed an instruction that had an integer immediate, then we use the bits from the instruction



Branch Unit

- Updates PC and the NZP bits
- PCMux.CTL: decides how PC is updated





0: NZP test then
OC = PC +1
or
PC = PC + 1 + IMM9

1: PC = PC +1

2: PC = PC +1 + IMM11

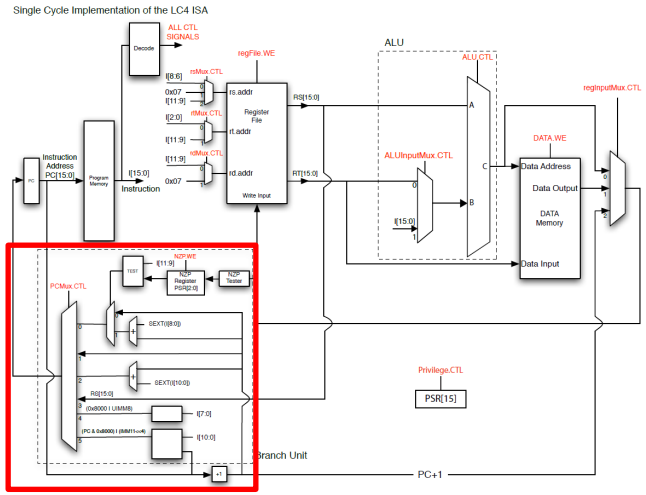
3: PC = RS

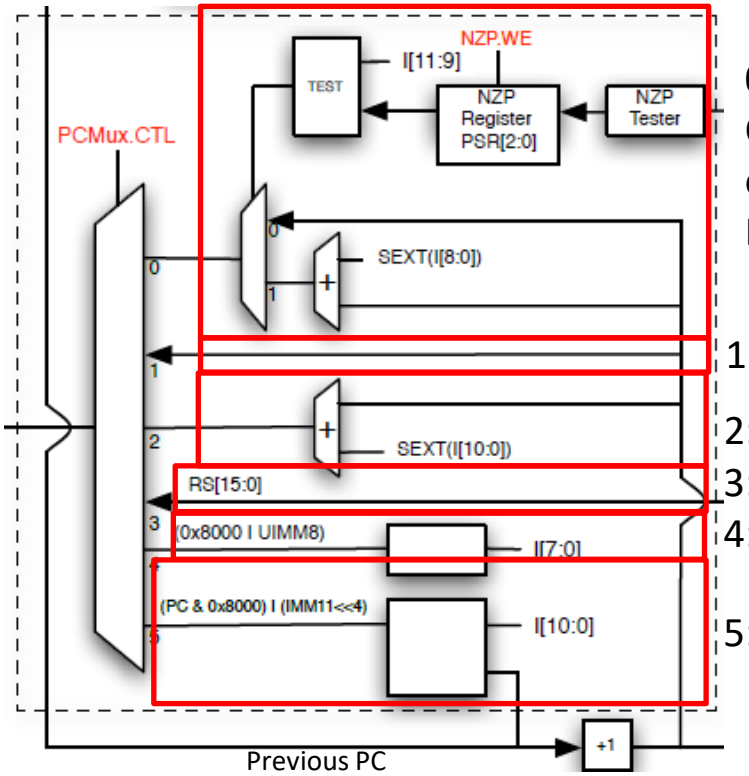
4: PC = 0x8000 | IMM8

5: PC = PC & 0x8000 |
(IMM11 << 4)

Branch Unit: ADD R0, R1, R5

- How does ADD R0, R1, R5 update the PC?
  - PC = PC + 1 so, PCMux.CTL is 1





0: NZP test then
OC = PC +1
or
PC = PC + 1 + IMM9

1: PC = PC +1

2: PC = PC +1 + IMM11

3: PC = RS

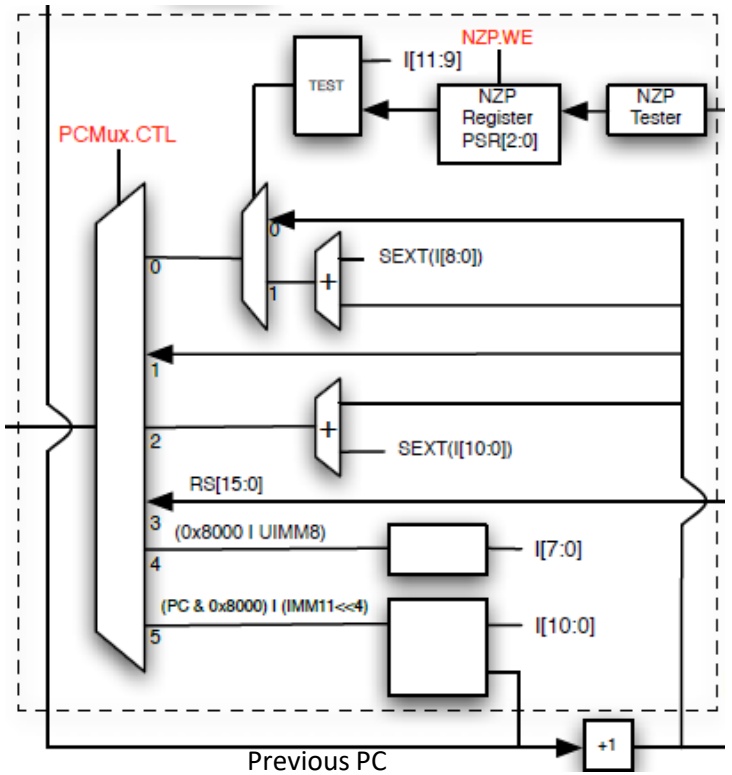
4: PC = 0x8000 | IMM8

5: PC = PC & 0x8000 |
(IMM11 << 4)

Branch Unit: ADD R0, R1, R5

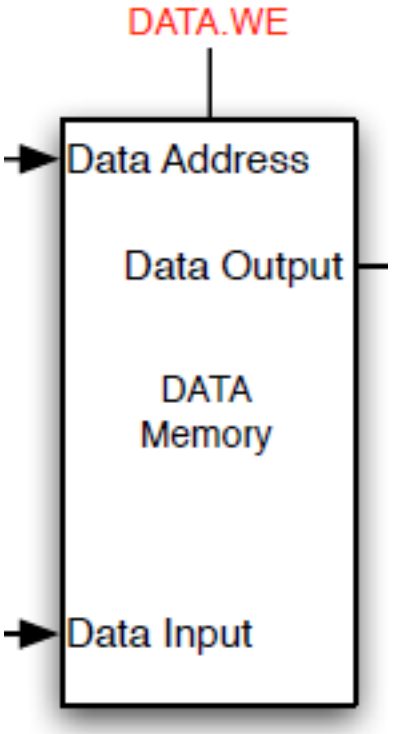
- NZP.WE: decides if NZP is updated for this instruction.

- Is NZP updated for ADD?
  - Yes, so NZP.WE is 1



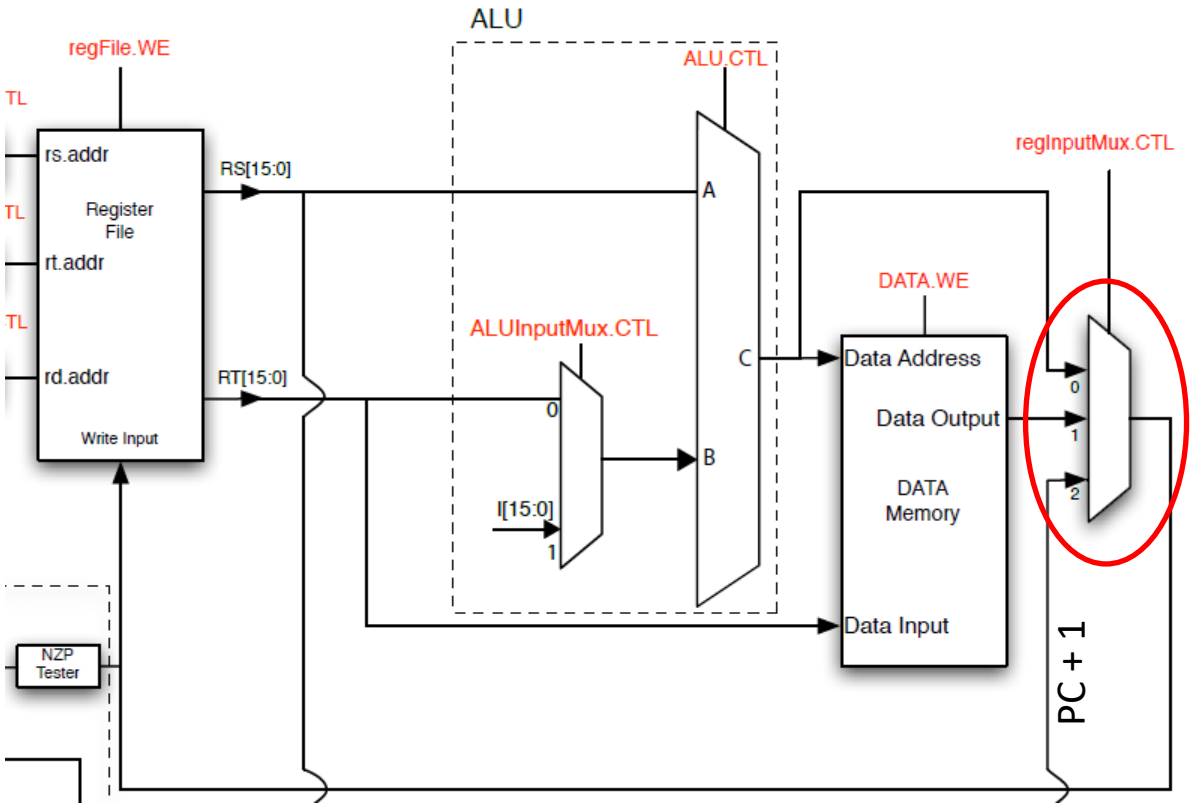
Data Memory

- 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 ADD update any data in memory?
  - No, so DATA.WE for ADD is 0

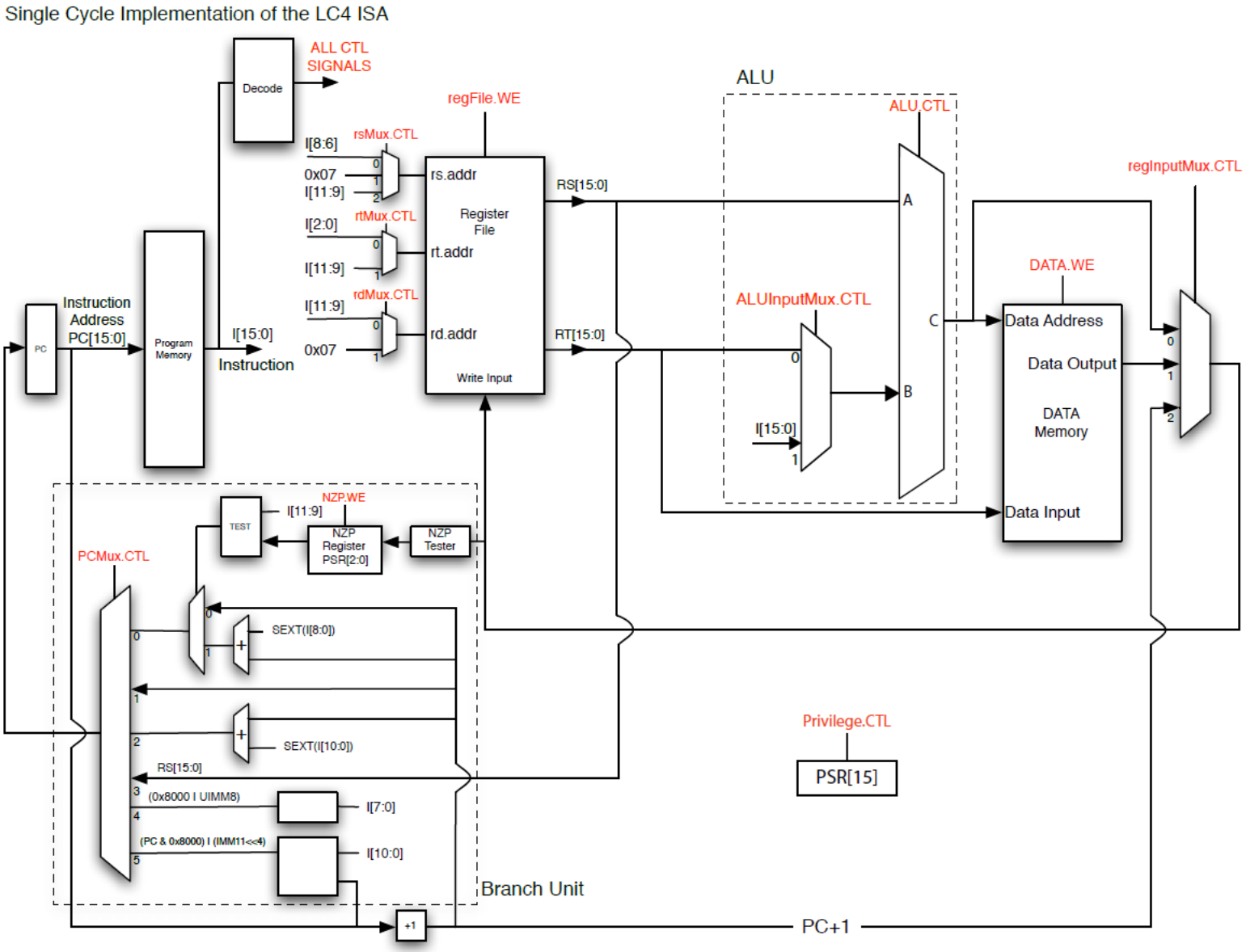


regInputMux.CTL

- Decides what gets written back to the register file
  - 0 = output of ALU
  - 1 = output of data memory
  - 2 = PC + 1



The Complete Picture



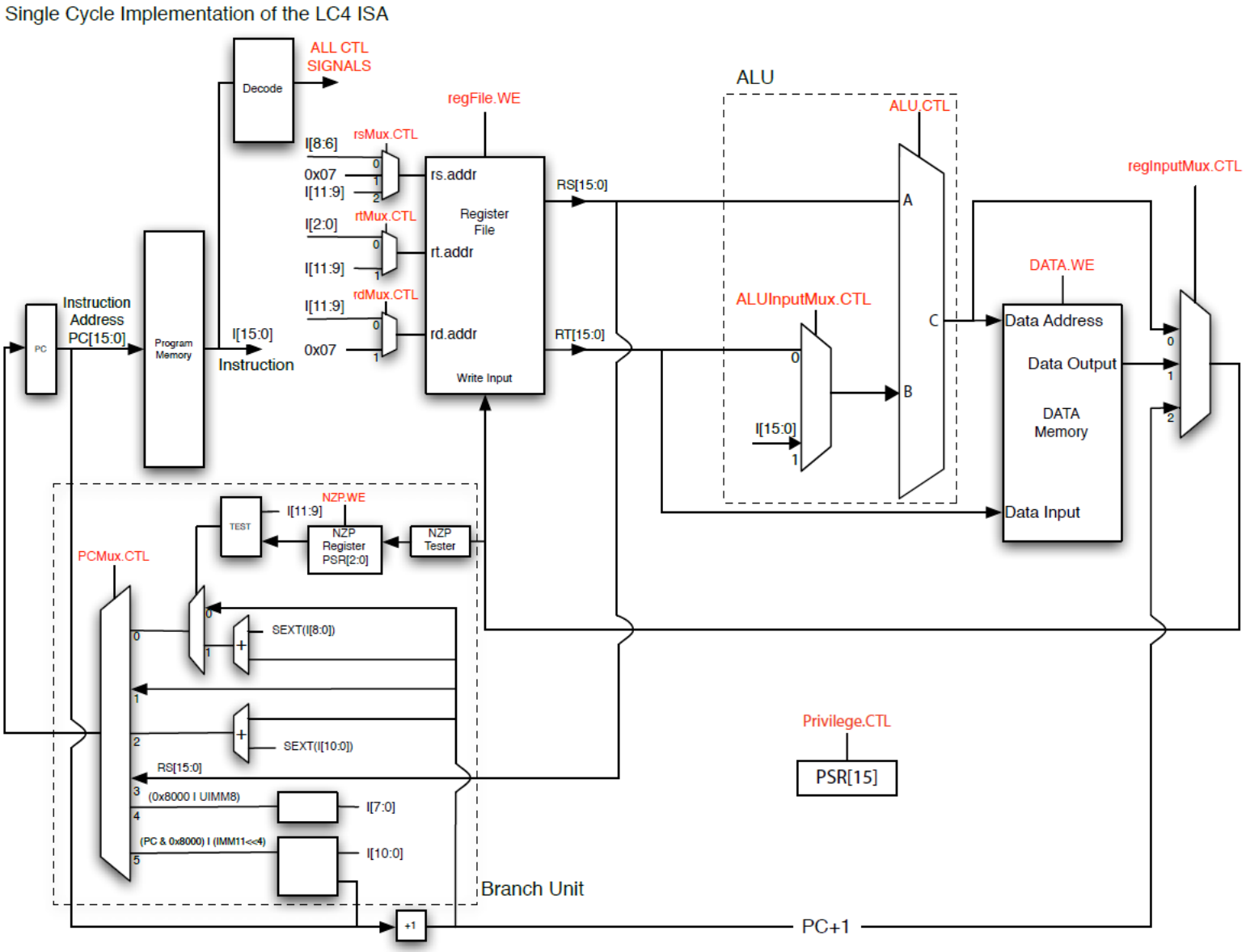
Privilege.CTL

- More on Privilege in future lectures
- Short version: TRAP and RTI modify privilege, all other instructions leave it alone.

Privilege.CTL	2	0	PSR[15] = 0 – Clear privilege bit
		1	PSR[15] = 1 – Set privilege bit
		2	PSR[15] unchanged – no change to privilege bit

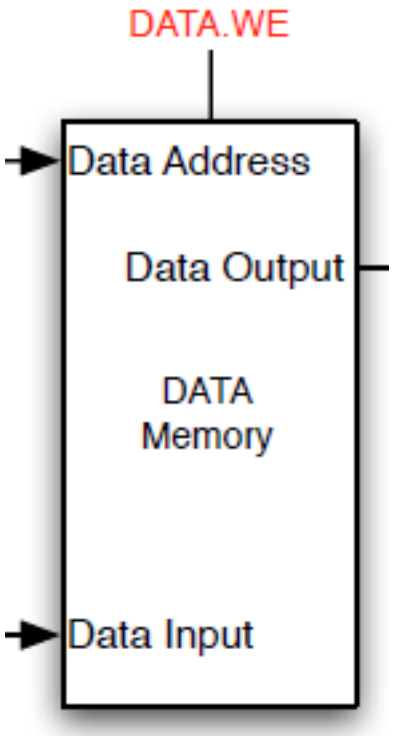
- All instructions except TRAP and RTI have Privilege.CTL set to 2

Questions?


Single Cycle Implementation of the LC4 ISA
Decode
ALL CTL SIGNALS
regFile.WE
rsMux.CTL
rtMux.CTL
rdMux.CTL
I[8:6]
0x07
I[11:9]
I[2:0]
rs.addr
rt.addr
rd.addr
Register File
Write Input
RS[15:0]
RT[15:0]
ALU
ALU.CTL
A
B
C
ALUInputMux.CTL
I[15:0]
regInputMux.CTL
DATA.WE
Data Address
Data Output
DATA Memory
Data Input
PC
Instruction Address PC[15:0]
Program Memory
Instruction
NZP.WE
TEST
NZP Register PSR[2:0]
NZP Tester
PCMux.CTL
SEXT(I[8:0])
SEXT(I[10:0])
(0x8000 | UIMM8)
I[7:0]
(PC & 0x8000) | (IMM11<<4)
I[10:0]
Branch Unit
Privilege.CTL
PSR[15]
+1
PC+1

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: ADD and DATA.WE
    - ADD 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 JMP instruction?
  - 11 different control signals questions on PollEv
- Probably want to pull up the Control Signals Description, LC4 Instruction, 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 on Wednesday