2GA3 Tutorial #7

DATE: November 5th, 2021

TA: Jatin Chowdhary

Changes

- Spend more time on tutorial questions
  - *i.e. More raw calculations*
    - *Less theory, more application*
- Less time on review questions
  - You will have to do these on your own
    - Answer is provided, but don’t look at it!
- Tutorial slides may be uploaded to Avenue
  - BUT they’re already posted on Teams
- **If you have any ideas, tell me!**

Midterm #2

- Don’t slack off
- It’s going to be tough
  - Study hard
    - Prepare now!
- I can already picture everyone panicking
  - It’s gonna be like flushing…
- Mostly on chapter 3 & 4
- Marking scheme has changed
  - Best of 2
    - Regardless, be well prepared!

Midterm #2

- **Little**/Big Endian will *probably* show up again
  - Maybe not on midterm 2, but final exam
  - *i.e. Here is a number stored in memory, blah blah, what is it?*

Address	+0	+1	+2	+3
0x00000018	00	00	00	00
0x00000014	00	00	00	00
0x00000010	00	00	00	00
0x0000000c	00	00	00	00
0x00000008	13	05	32	00
0x00000004	33	05	52	00
0x00000000	93	42	43	00

***Question:** Convert the floating point number at address 0x00000008 into its corresponding decimal number.*

Show all of your work.

Anything about anything I just said? Or anything you just saw?

Homework Question #1

- **Question:** Implementation in hardware yields better *p*__________, and is more *e*_________ than software
- **Options:**
  - A) Performance, Effective
  - B) Price, Efficient
  - C) Performance, Efficient
  - D) Price, Effective
  - E) None of the above
  - **The answer is:** C

Homework Question #2

- **Question:** What does MSD stand for?
- **Options:**
  - A) Most signed digits
  - B) Most signed decimals
  - C) Most significant decimal
  - D) Most significant dollar
  - E) Most significant digit
  - F) None of the above
  - **The answer is:** E

Homework Question #3

- **Question:** For a single precision floating point number (IEEE 754 format), how big is the bias?
- **Options:**
  - A) 2^7
  - B) $2^7 - 1$
  - C) 2^8
  - D) $2^8 - 1$
  - E) 127
  - **The answer is:** B & E

Homework Question #4

- **Question:** For a double precision floating point number in IEEE 754, how big is the bias?
- **Options:**
  - A) $2^{10} + 1$
  - B) $2^{10} – 1$
  - C) 2^{10}
  - D) $2^{10} + 754$
  - E) $2^{10} – 754$
  - **The answer is:** B (which evaluates to 1023)

Homework Question #5

- **Question:** How big is the bias if the exponent is 24-bits wide? Assume we are using the IEEE 754 format.
- **Option:**
  - A) $2^{24} - 1$
  - B) $2^{24} + 1$
  - C) 2^{24}
  - D) $2^{23} - 1$
  - E) $2^{23} + 1$
  - F) 2^{23}
  - G) None of the above
  - **The answer is:** D
- *Hint: Look at the size of the bias relative to the exponent for single and double precision floating point numbers stored in IEEE 754 format!*

Review Question #1

- **Question:** How does the hardware (i.e. CPU) know the difference between *ints* and *floats* in memory?
  - *Asked By Steven*
- **Answer:** It doesn't!
  - Everything in memory is a 0 or a 1
  - The CPU does not care about the semantics of the data
    - The sole job of the CPU is to carry out instructions. Period
      - The CPU is not aware of what you are trying to do *on a high level*
  - It is up to the programmer, you, to make sense of the data by specifying the type – like int, float, char, etc.
    - Based on the type of the data, the compiler uses the appropriate register
      - i.e. Integer VS. Floating Point

Review Question #2

- **Question:** What is the difference between signed magnitude, one's complement, two's complement?
  - *Asked By Steven*
- **Answer:** All of them are ways of representing numbers – but the technique is different
  - Signed Magnitude Format
    - To calculate a negative number, flip the MSD to 1
      - *i.e.* $(+57)_{10} \rightarrow (\mathbf{0}011\ 1001)_2$
      - *i.e.* $(-57)_{10} \rightarrow (\mathbf{1}011\ 1001)_2$
  - One's Complement
    - To calculate a negative number, flip all the bits (or take the complement)
      - *i.e.* $(+57)_{10} \rightarrow (0011\ 1001)_2$
      - *i.e.* $(-57)_{10} \rightarrow (1100\ 0110)_2$

** Assume that the binary numbers are 8-bits **

Review Question #2

- **Question:** What is the difference between signed magnitude, one's complement, two's complement?
  - *Asked By Steven*
- **Answer:** All of them are ways of representing numbers – but the technique is different
  - Two's Complement
    - To calculate a negative number, flip all the bits, and then add 1
      - *i.e.* $(+57)_{10} \rightarrow (0011\ 1001)_2$
      - *i.e.* $(-57)_{10} \rightarrow (1100\ 011\mathbf{1})_2$
  - In signed magnitude and one's complement, there are 2 ways to represent 0
    - In two's complement, there's only 1 way to represent 0
      - So the ranges are different!
        - −127 VS. −128

** Assume that the binary numbers are 8-bits **

Anything About Anything?

Tutorial Question #1

- **Question:** The Hewlett-Packard 2114, 2115, and 2116 used a format with the leftmost 16 bits being the fraction stored in two's complement format, followed by another 16-bit field which had the leftmost 8 bits as an extension of the fraction (making the fraction 24 bits long), and the rightmost 8 bits representing the exponent. However, in an interesting twist, the exponent was stored in sign-magnitude format with the sign bit on the far right! Write down the bit pattern to represent -1.5625×10^{-1} assuming this format. No hidden 1 is used. Comment on how the range and accuracy of this 32-bit pattern compares to the single precision IEEE 754 standard.
- **Answer:**
  - Next slide
    - (This is kind of homework)

Tutorial Answer #1

- **Precursor:**
  - What do we know?
    - Leftmost 16-bits is the fraction
      - Stored in two's complement
    - The 16-bits after the fraction are:
      - Leftmost 8-bits are part of the fraction
        - Fraction is 24-bits long
      - Rightmost 8-bits represent the exponent
        - Stored in signed magnitude format
        - Sign bit on the far right
  - What does it look like?
    - 00000000000000000000000000000000
      - Fraction | Exponent | Exponent's Sign
        - Fraction = 24-bits
        - Exponent = 8-bits
    - The whole thing is 32-bits = *(24 + 8)*

Tutorial Answer #1

- **Answer:**
  - Using the format stated, we need to convert the following number into its bit pattern. The number is: -1.5625×10^{-1}
  - **Step 1)** Align the number:
    - -1.5625 **x** $10^{-1} \rightarrow -0.15625$ **x** 10^{0}
  - **Step 2)** Convert to binary:
    - -0.15625 **x** $10^{0} \rightarrow -0.00101$ **x** 2^{0}
  - **Step 3)** Move binary point *two* places to the right:
    - -0.00101 **x** $2^{0} \rightarrow -0.101$ **x** 2^{-2}
  - So far, we have the fraction, exponent, and the exponent's sign
    - Remember, the format looks like this: 00000000000000000000000000000000
      - Fraction = 24-bits
      - Exponent = 8-bits

Tutorial Answer #1

- **Answer:**
  - *Currently, we have: –0.101* ***x*** *2^{-2}*
    - Now, we need to convert what we have into the format listed in the question
  - **Step 4)** Fraction = –0.101
    - *There is no hidden 1*
    - –0.101 is stored in two's complement
      - So, we flip the bits and add 1
        - 0101 → 1010 → 1011
    - The fraction representation is: **1011**

Tutorial Answer #1

- **Answer:**
  - **Step 5)** Exponent = –2
    - Remember, the exponent is stored in *sign magnitude format*
      - Rightmost bit is the sign bit
        - Hence, sign = 1 (because number is negative)
    - First, let's convert (+2) into binary:
      - $(+2)_{10} \rightarrow (0000\ 010\mathbf{0})_2$
        - *The rightmost bit is the sign bit*
    - Second, we flip the sign bit to convert $(+2)_{10}$ into $(-2)_{10}$
      - $(0000\ 010\mathbf{0})_2 \rightarrow (0000\ 010\mathbf{1})_2$
        - *We flip the sign bit from 0 to 1, to get a negative number*
    - The exponent representation: **0000 0101**
      - *Sticking with our color codes, the number is:* $(0000\ 010\mathbf{1})_2$

Tutorial Answer #1

- **Answer:**
  - Recall that the format is:
    - 00000000000000000000000000000000
      - Fraction | Exponent | Exponent's Sign
  - Currently, we have:
    - Fraction = 1011
    - Exponent = 0000 0101
  - **Our final answer, in binary, is:**
    - **1011 0000 0000 0000 0000 0000 0000 0101**

Anything About Anything?

Tutorial Question #2

- **Question:** IEEE 754-2008 contains a half precision that is only *16-bits* wide. The leftmost bit is still the sign bit, the exponent is *5-bits* wide and has a bias of 15, and the mantissa is *10-bits* long. A hidden 1 is assumed.

  Calculate $((3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^{3})$ by hand, assuming each of the values is stored in the *16-bit* half precision format. Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even.

- **Answer:**
  - Next slide

Floating Point Problems

- **Precursor:**
  - First of all:
    - Recall that operations on floating point numbers cannot be represented in a given amount of precision
      - For example, how do we accurately represent ⅓ in base-2?
        - We cannot! Because ⅓ repeats. Therefore, we round results
    - There are many different ways of rounding
      - Each method has its own use and exists for different reasons
    - The goal is to round the number so the result is as "*correct*" as possible
  - Second point:
    - The IEEE standard uses 3 (extra) bits of less significance than the 24-bits in the Mantissa; this is for ALL precisions
      - The 3 (extra) bits are: Guard (g), Round (r), Sticky (s)

Guard, Round, Sticky Bit

- **Precursor:**
  - What does the guard, round, and sticky bit look like?
    - 1.XXXXXXXXXXXXXXXXXXXXXXX g r s
      - 1 = Hidden bit/one
      - XXXXXXXXXXXXXXXXXXXXXXX = 23-bit mantissa
        - *(However, for this question, the mantissa is 10-bits long)*
      - g = Guard bit
      - r = Round bit
      - s = Sticky bit
    - This is the internal format for a single precision floating point value
      - *The guard, round, and sticky bit are extra bits; they are not officially part of the mantissa*

Guard, Round, Sticky Bit

- **Precursor:**
  - What is the point of a guard, round, and sticky bit?
    - The guard and round bits are just 2 extra bits of *precision* that are used in calculations
    - The sticky bit indicates what is, or could be, in lesser significant bits, but is not present
      - If a **1** is shifted into the sticky bit, then the sticky bit stays at **1**, regardless of further shifts
      - *The point of a sticky bit is to determine if we have lost any precision. Essentially, the sticky bit says, "we use to have a 1 in the right bits, but we lost it at some point, hence we have lost precision"*
  - Example on next slide

G.R.S Example

- **Quick Example:**

Shift	Mantissa	Guard (g)	Round (r)	Sticky (s)
0	1.11000000000000000000100	0	0	0
1	0.11100000000000000000010	0	0	0
2	0.01110000000000000000001	0	0	0
3	0.00111000000000000000000	1	0	0
4	0.00011100000000000000000	0	1	0
5	0.00001110000000000000000	0	0	1
6	0.00000111000000000000000	0	0	1
7	0.00000011100000000000000	0	0	1
8	0.00000001110000000000000	0	0	1

G.R.S. Rounding Rules

- Mingzhe posted a table that summarizes rounding based on the Guard, Round, and Sticky bits
  - Here it is:

Guard bit	Round bit	Sticky bit	Rounding action
0	0	0	Truncate
0	0	1	Truncate
0	1	0	Truncate
0	1	1	Truncate
1	0	0	Round to Even**
1	0	1	Round up
1	1	0	Round up
1	1	1	Round up

** If the bit before the guard bit is 0, we do not round, and if the bit before the guard bit is 1, we add 1 to the least significant bit.

G.R.S. Rounding Rules

Guard Bit	Round Bit	Sticky Bit	Action
0	0	0	Truncate
0	0	1	Truncate
0	1	0	Truncate
0	1	1	Truncate
1	0	0	Round To Even**
1	0	1	Round Up
1	1	0	Round Up
1	1	1	Round Up

** If the bit before the guard is 0, we do not round. If the bit before the guard is 1, we add 1 to the least significant bit.

Tutorial Answer #2

- Back to the question:
  - The representation is half a precision (16-bits (in total))
  - Leftmost bit is the sign bit
  - Exponent is 5-bits wide
    - Bias of 15
  - Mantissa is 10-bits
    - Hidden 1 is assumed
- ***Calculate:*** $((3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^{3})$

Tutorial Answer #2

- *Calculate $((3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^{3})$*
  - **Step 1)** Convert the numbers to binary
    - **a)** 3.984375×10^{-1}
      - First, we align the number: $(3.984375 \times 10^{-1}) \rightarrow (0.3984375 \times 10^{0})$
      - Then, we convert to binary: $(0.3984375 \times 10^{0}) \rightarrow (0.0110011 \times 2^{0})$
        - *Note: $(0.0110011)_2 = 2^{-2} + 2^{-3} + 2^{-6} + 2^{-7} = (0.3984375)_{10}$*
      - Finally, we normalize the number: $(0.0110011 \times 2^{0}) \rightarrow (1.10011 \times 2^{-2})$
        - *Move the decimal point, **2** places to the right*
          - *Exponent calculation: $0 - 2 = -2$*
      - **So, the number is: (1.10011×2^{-2})**
    - *Note: This is the IEEE-754 2008 format for half precision*
      - *Half precision = 16-bits*

Tutorial Answer #2

- *Calculate* $((3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^{3})$
  - **Step 1)** Convert the numbers to binary
    - **b)** 3.4375×10^{-1}
      - First, we align the number: $(3.4375 \times 10^{-1}) \rightarrow (0.34375 \times 10^{0})$
      - Then, we convert to binary: $(0.34375 \times 10^{0}) \rightarrow (0.01011 \times 2^{0})$
        - Note: $(0.01011)_2 = 2^{-2} + 2^{-4} + 2^{-5} = (0.34375)_{10}$
      - Finally, we normalize the number: $(0.011011 \times 2^{0}) \rightarrow (1.10011 \times 2^{-2})$
        - Move the decimal point, **2** places to the right
          - Exponent calculation: 0 – 2 = 2
      - **So, the number is: (1.01100×2^{-2})**
    - *Note: The logic for converting (b) is the same as the logic for converting (a)*
      - *Once again we are converting the number into IEEE-754 2008 format*

Tutorial Answer #2

- *Calculate ((3.984375 × 10^{-1} + 3.4375 × 10^{-1}) + 1.771 × 10^{3})*
  - **Step 1)** Convert the numbers to binary
    - c) 1.771 × 10^{3}
      - First, align the number: (1.771 × 10^{3}) → (1771.0 × 10^{0})
      - Then, convert to binary: (1771.0 × 10^{0}) → (11011101011.0 × 2^{0})
        - $(11011101011)_2 = 2^0 + 2^1 + 2^3 + 2^5 + 2^6 + 2^7 + 2^9 + 2^{10} = (1771)_{10}$
        - Note: The number is not stored in a floating point format like IEEE-754 2008; it is stored as an integer.
      - Finally, normalize the number: (11011101011.0 × 2^{0}) → (1.1011101011 × 2^{10})
        - When we normalize a number, we want it to be in the following format: **a.b × y^{z}**
      - **So, the number is: (1.1011101011 × 2^{10})**

Tutorial Answer #2

- *Calculate* $((3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^{3})$
  - **Step 2)** Add the first two numbers, because they take precedence (due to the brackets)
    - The calculation is: $[(1.1001100000 \times 2^{-2}) + (1.0110000000 \times 2^{-2})]$

(Add)	$(1.1001100000) \times 2^{-2}$
+	$(1.0110000000) \times 2^{-2}$
=	$(10.1111100000) \times 2^{-2}$
(Normalize) =	$(1.0111110000) \times 2^{-1}$

Tutorial Answer #2

- *Calculate ((3.984375 × 10^{-1} + 3.4375 × 10^{-1}) + 1.771 × 10^{3})*
  - **Step 3)** Add the remaining numbers (the last two numbers)
    - From the previous slide we know that the first number is:
      - $1.01111100000 \times 2^{-1}$
    - The second number, *to add*, is:
      - $1.1011101011 \times 2^{10}$
    - The calculation is: $(1.01111100000 \times 2^{-1}) + (1.1011101011 \times 2^{10})$
    - Now, align the first number to look like the second number:
      - $(1.01111100000 \times 2^{-1}) \rightarrow (0.0000000000\ \mathbf{1}\ \mathbf{0}\ \mathbf{1}\ 11110000 \times 2^{10})$
        - Move the decimal point **11** places to the right
          - Exponent calculation: $-1 + 11 = 10$
        - Note:
          - Guard (g) = 1
          - Round (r) = 0
          - Sticky (s) = 1

Tutorial Answer #2

- *Calculate ((3.984375 × 10^−1^ + 3.4375 × 10^−1^) + 1.771 × 10^3^)*
  - **Step 3)** Add the remaining numbers (the last two numbers)
    - After aligning the first number, the calculation is:

      $(1.1011101011 \times 2^{10}) + (0.0000000000\ \mathbf{1}\ \mathbf{0}\ \mathbf{1}\ 11110000 \times 2^{10})$

(Add)	$1.1011101011 \times 2^{10}$
+	$0.0000000000\ \mathbf{1}\ \mathbf{0}\ \mathbf{1}\ 11110000 \times 2^{10}$
=	$1.1011101011\ \mathbf{1}\ \mathbf{0}\ \mathbf{1}\ 11110000 \times 2^{10}$
(Round Up) =	$1.1011101100 \times 2^{10}$

Tutorial Answer #2

- *Calculate* $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^{3}$
  - The answer, in binary, is: $(1.1011101100 \times 2^{10})$
    - Remember, we rounded up from:

      $(1.1011101011\ 1\ 0\ 1 \times 2^{10}) \rightarrow (1.1011101100 \times 2^{10})$
  - **Step 4)** Convert to back to decimal
    - $(1.1011101100 \times 2^{10}) = (11011101100 \times 2^{0})_2 = (1772)_{10}$
      - $(11011101100 \times 2^{0})_2 = (2^2 + 2^3 + 2^5 + 2^6 + 2^7 + 2^9 + 2^{10})$
  - **Step 5)** The answer is: $\mathbf{(1772)_{10}}$

Tutorial Answer #2 (Bonus)

- The *IEEE 754-2008* format is: **S EEEEE FFFFFFFFFF**
  - S = Sign Bit
    - 1 Bit
  - E = Exponent
    - 5 Bits
  - F = Fraction
    - 10 Bits
- To represent $(1772)_{10}$ in the *IEEE 754-2008* format, we need its binary representation
  - From the previous slide, it is: **($1.1011101100 \times 2^{10}$)**
- Now, we divide up the binary number from above to satisfy the *IEEE 754-2008* format:
  - *Next slide*

Tutorial Answer #2 (Bonus)

- Now, we divide up the binary number to satisfy the *IEEE 754-2008* format:
  - Sign (Bit) = 0
    - Since the number is positive, the sign bit is 0
      - This is because $(-1)^0 = 1$
  - $\text{Exponent}_{\text{Actual}} = 10$
    - But wait, we need to apply the bias, which is 15:
      - Biased Exponent = Actual Exponent + Bias

        Biased Exponent = 10 + 15

        Biased Exponent = 25
  - $\text{Exponent}_{\text{Biased}} = (25)_{10} = \mathbf{(11001)_2}$
    - Tip: Don't forget to convert each part to its binary form
  - Fraction = 1011101100
    - Note: We removed the initial **1.**
      - *This is the hidden one*
    - (**1.**1011101100) → (1011101100)

Tutorial Answer #2 (Bonus)

- Finally, we put it all together:
  - Sign = 0
  - Exponent = 11001
  - Fraction = 1011101100
- IEEE 754-2008 format is: S ++ EEEEE ++ FFFFFFFFFF
  - 0 11001 1011101100
- Therefore, the *IEEE 754-2008* representation of $(1772)_{10}$ is: 0110011011101100
  - But wait, how does *0110011011101100* manage to equal 1772?
    - *Next slide*

Tutorial Answer #2 (Bonus)

- To go from **0110011011101100** to **1772**, we need to do the following:
  - $(-1)^{S} \times (1 + \textbf{Fraction}) \times 2^{(\textbf{Exponent} - \text{Bias})}$
    - Note: The (1 +) is the hidden one – it's back now
    - Note: The bias is 15

$= (-1)^{0} \times (1 + (\textbf{1011101100})_2) \times 2^{((\textbf{110011}) - \text{Bias})}$

$= (1) \times (1 + (\textbf{1011101100})_2) \times 2^{(25 - 15)}$

$= (1) \times (1 + (\textbf{1011101100})_2) \times 2^{10}$

$= (1) \times (1 + \textbf{0.73046875}) \times 2^{10}$

$= (1) \times (1.73046875) \times 2^{10}$

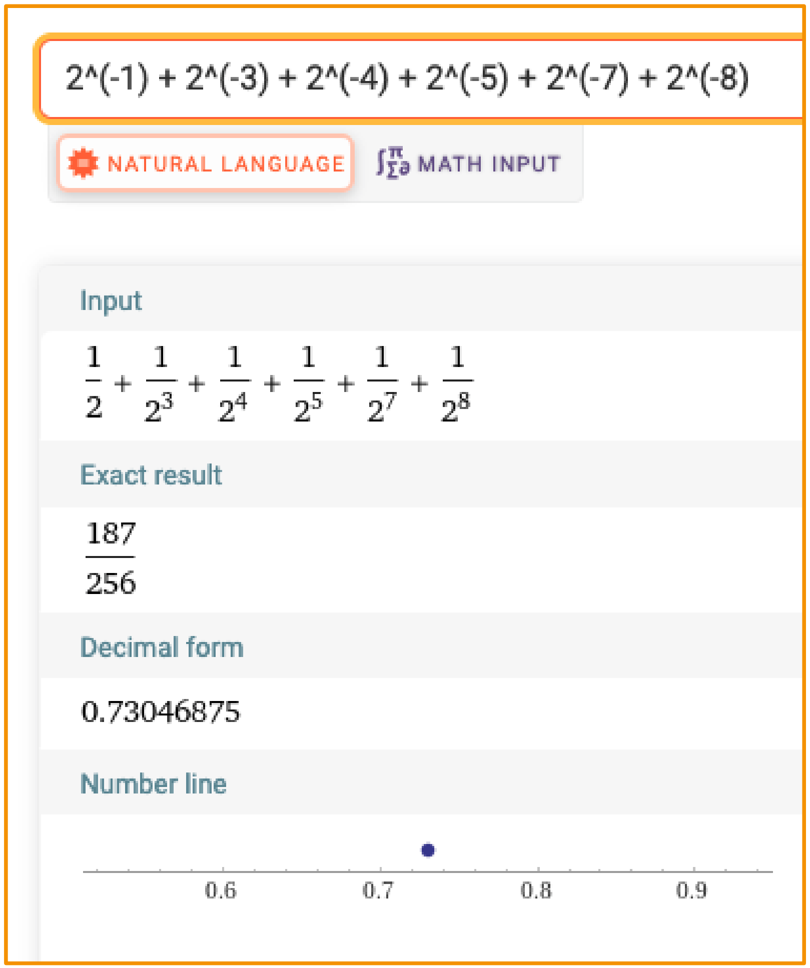
$= (1772)_{10}$

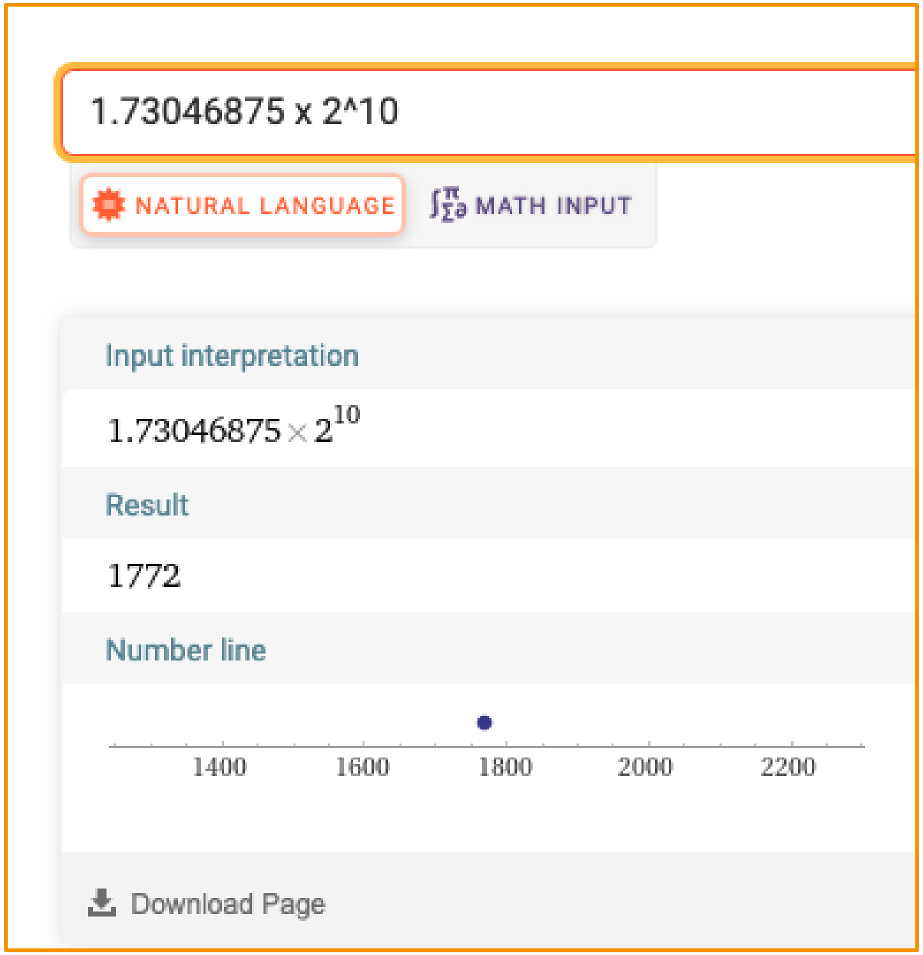
$\textbf{1011101100} = 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-7} + 2^{-8}$

$\textbf{1011101100} = \textbf{0.73046875}$

Tutorial Answer #2 (Bonus)

- Proof:

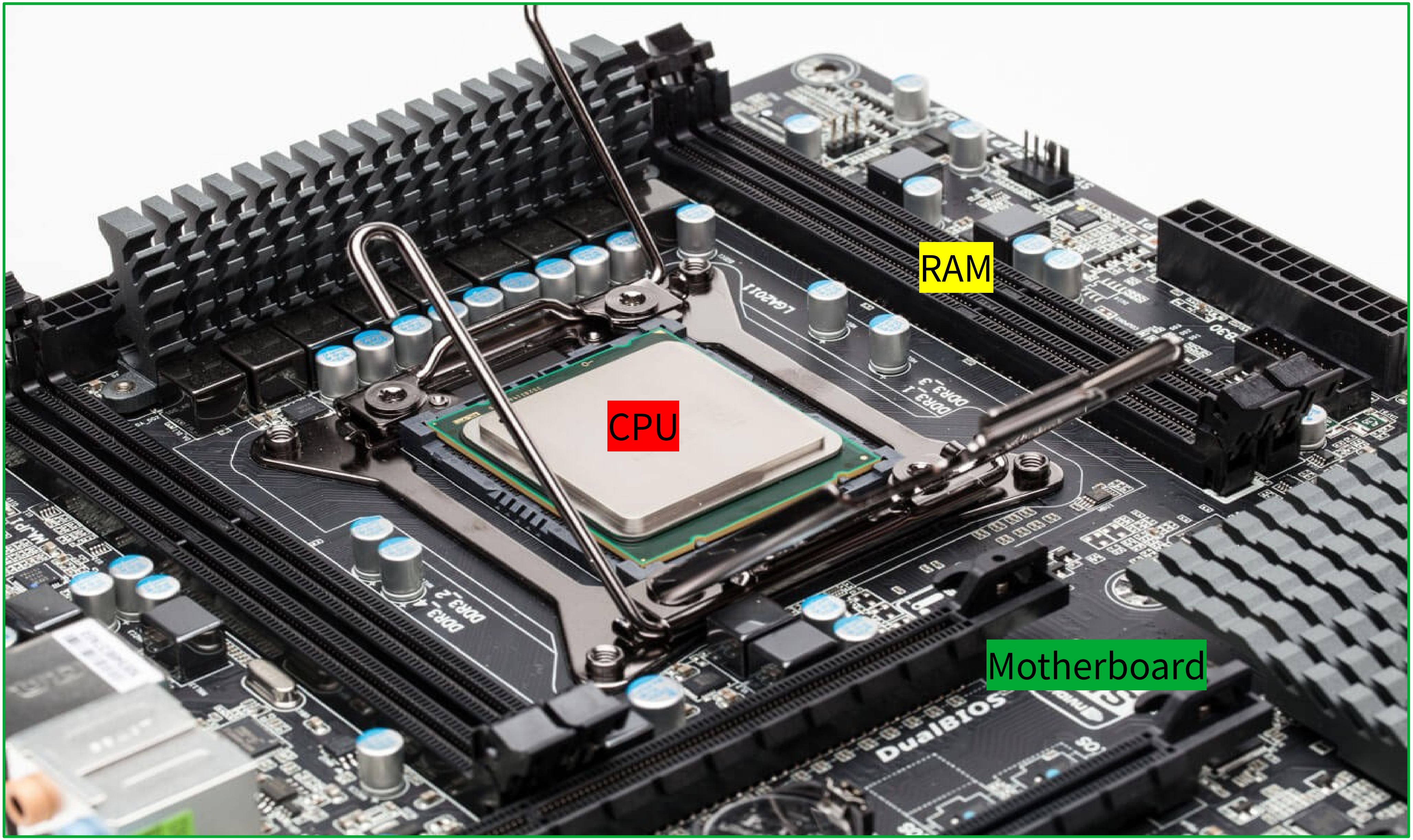




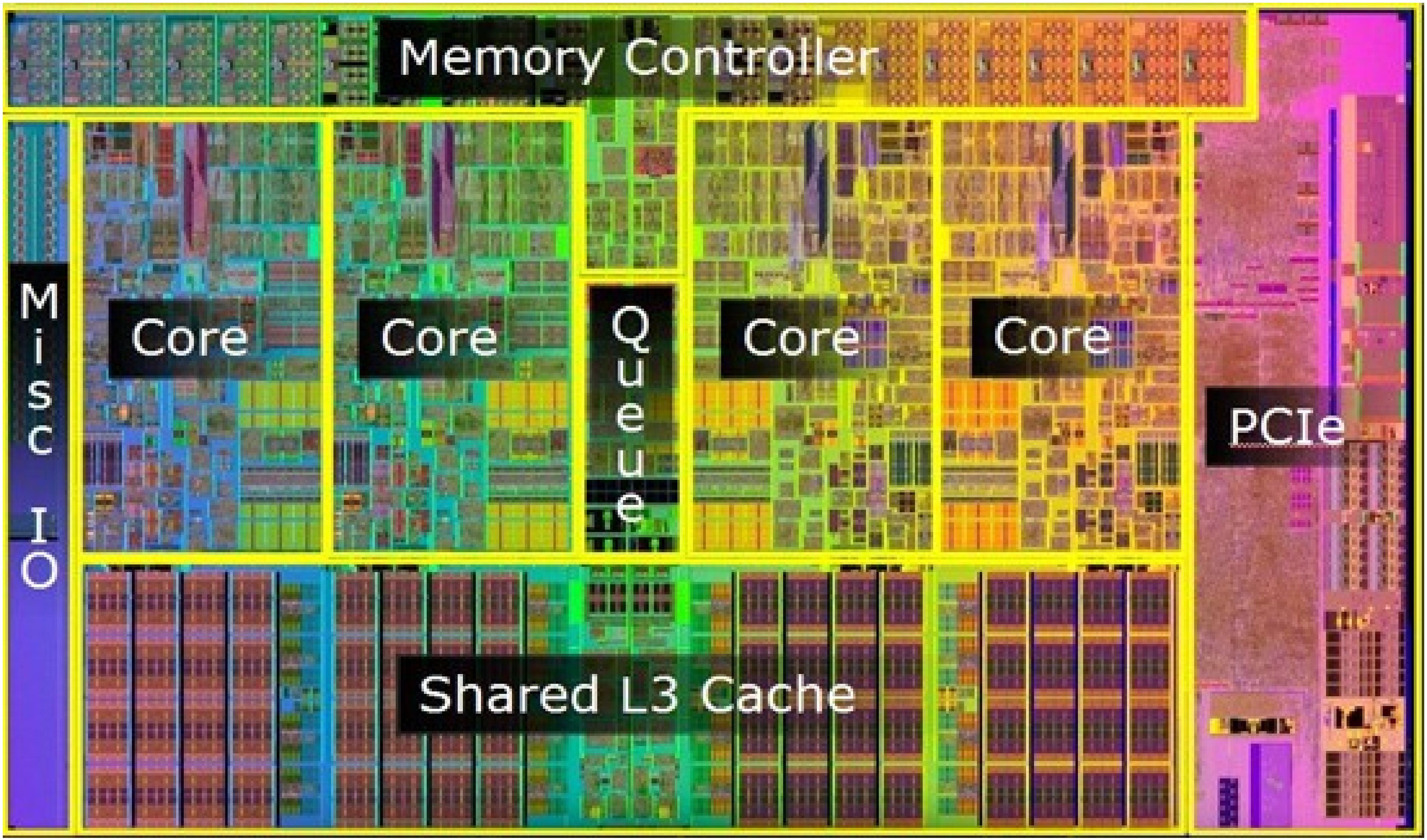
The Big Picture

- This is what the inside of a desktop looks like

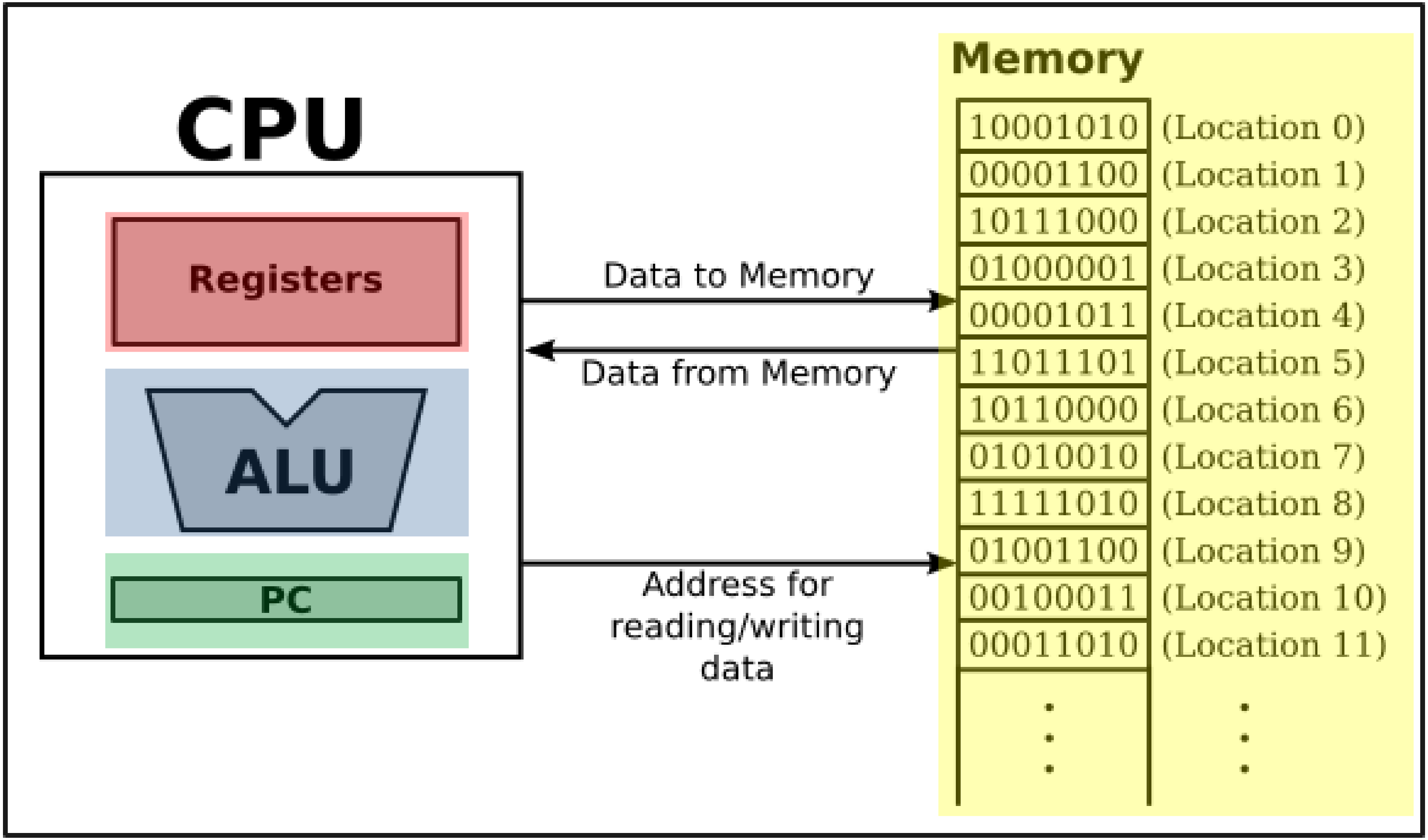
Motherboard



CPU Cross Section



Abstract View



Datapath Abstract View

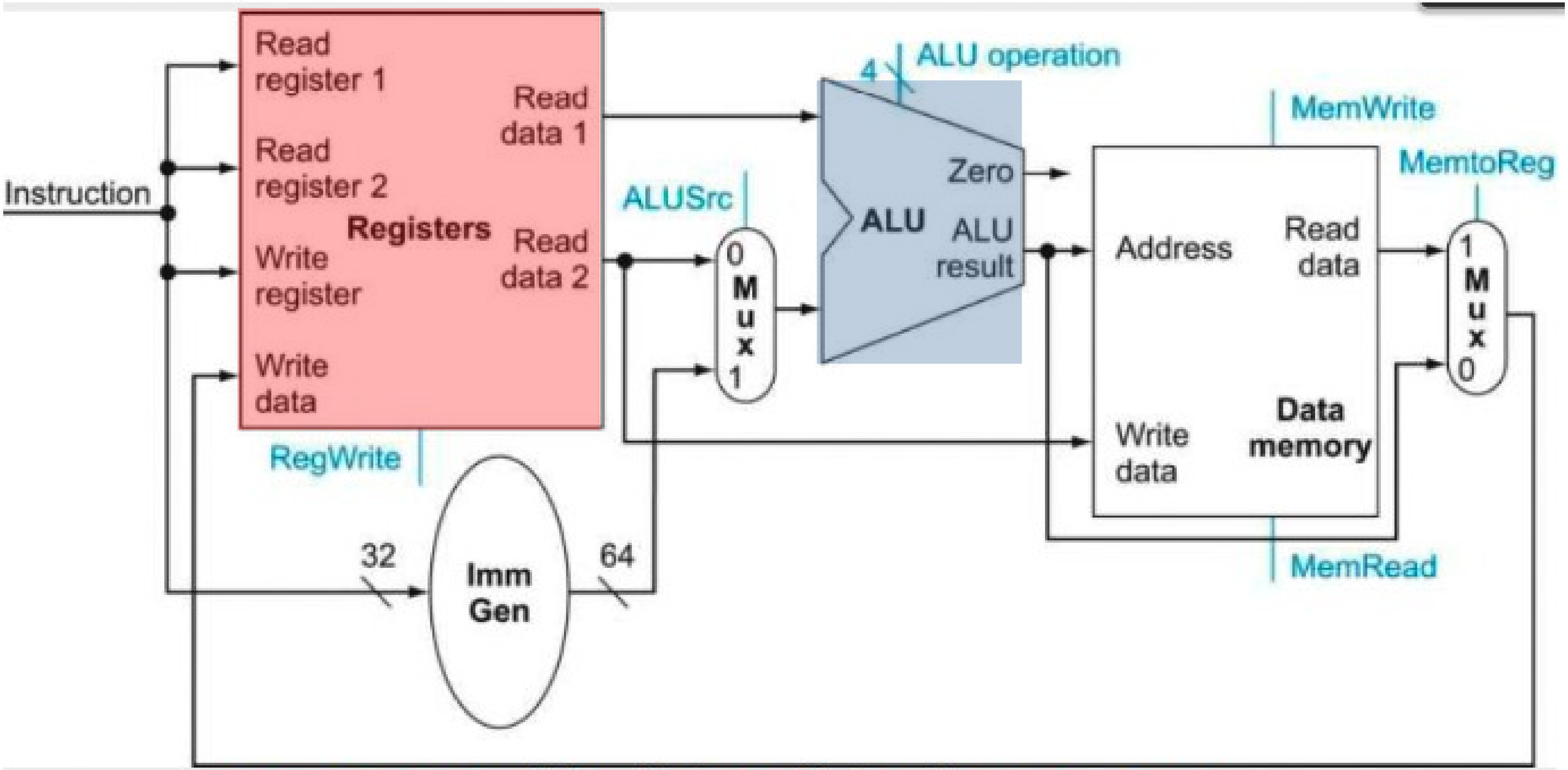
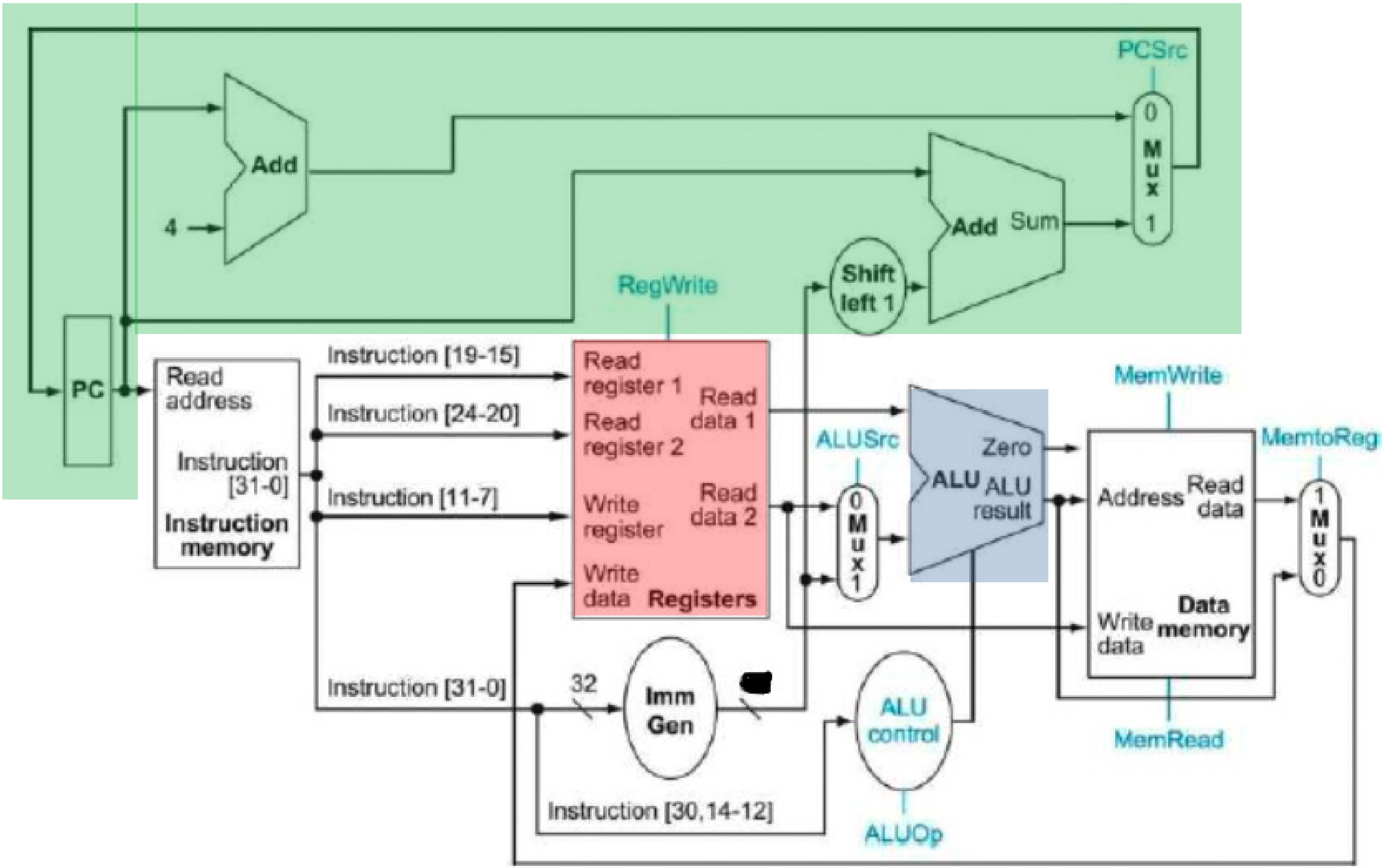


FIGURE 4.10 **The datapath for the memory instructions and the R-type instructions.**

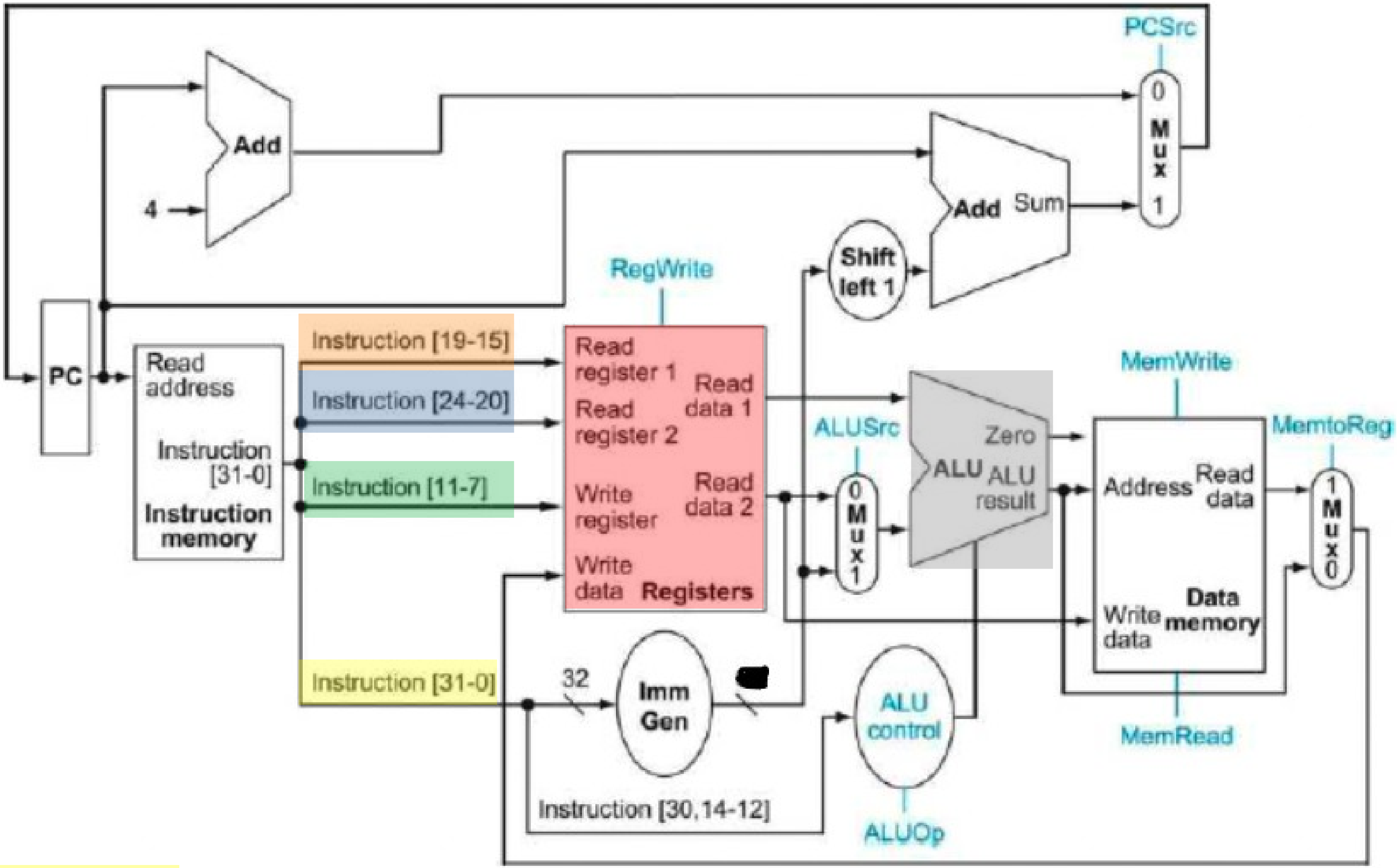
Datapath Abstract View



How Does It Work?

- A RISC-V instruction like *xori x5, x6, 4*, translates to 0000 0000 0100 0011 0100 0010 1001 0011 in machine code.
  - The 0's and 1's are *low* and *high* voltages
    - This is where the voltage difference comes from
    - The more 0's and 1's we can shove, the *faster* it is
      - In nerd terms, more *cycles* equals *faster* performance
  - <u>These 0's and 1's are sent to different parts of the Datapath for decoding/processing</u>
- So If I asked you what is a register and why do computers only understand a 0 and a 1, all of you should be able to answer this
  - But do you see the big picture?
    - Rather, the whole picture and how everything ties in together!

Instructions & Datapath



add x5, x6, x7 == 0000000 **00000** **00000** 000 **00000** **0110011**

More Details

- Going based off the previous slide:
  - Each part of the machine code (i.e. The bits) is sent along a different path
    - *Note: The colors correspond to what goes where*
  - The registers, x5, x6, and x7, get sent to the Registers block
    - The Registers block is highlighted in red
  - The calculation is performed in the ALU
    - The ALU is highlighted in gray
- *mux* stands for <u>mu</u>ltiple<u>x</u>er
  - The role of a multiplexer is to control the signals
    - If a *mux* is given a high signal (i.e. 1), then it is "on" (or active)
    - If a *mux* is given a low signal (i.e. 0), then it is "not on" (or inactive)

Tutorial Question #3

- **Question:** Consider the following instruction *and rd, rs1, rs2*. The instruction is interpreted as Reg [rd] = Reg [rs1] **AND** Reg [rs2]. Note: The **AND** is logical AND.
  - **A)** What are the values of control signals generated by the control in Figure 4.10 for this instruction?
    - *Figure on next slide*
  - **B)** Which resources (blocks) perform a useful function for this instruction?
  - **C)** Which resources (blocks) produce no output for this instruction? Which resources produce output that is not used?

Figure 4.10

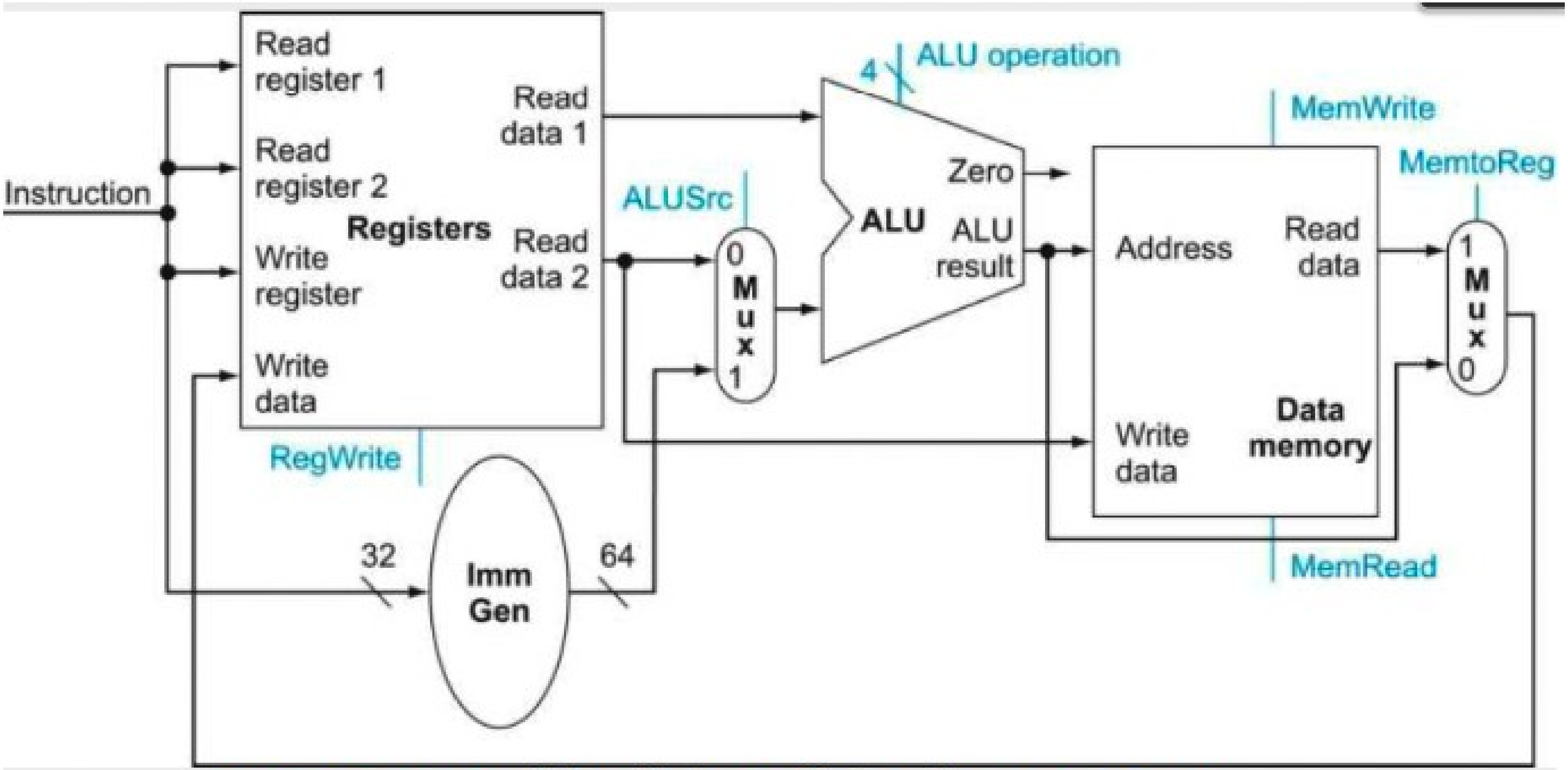
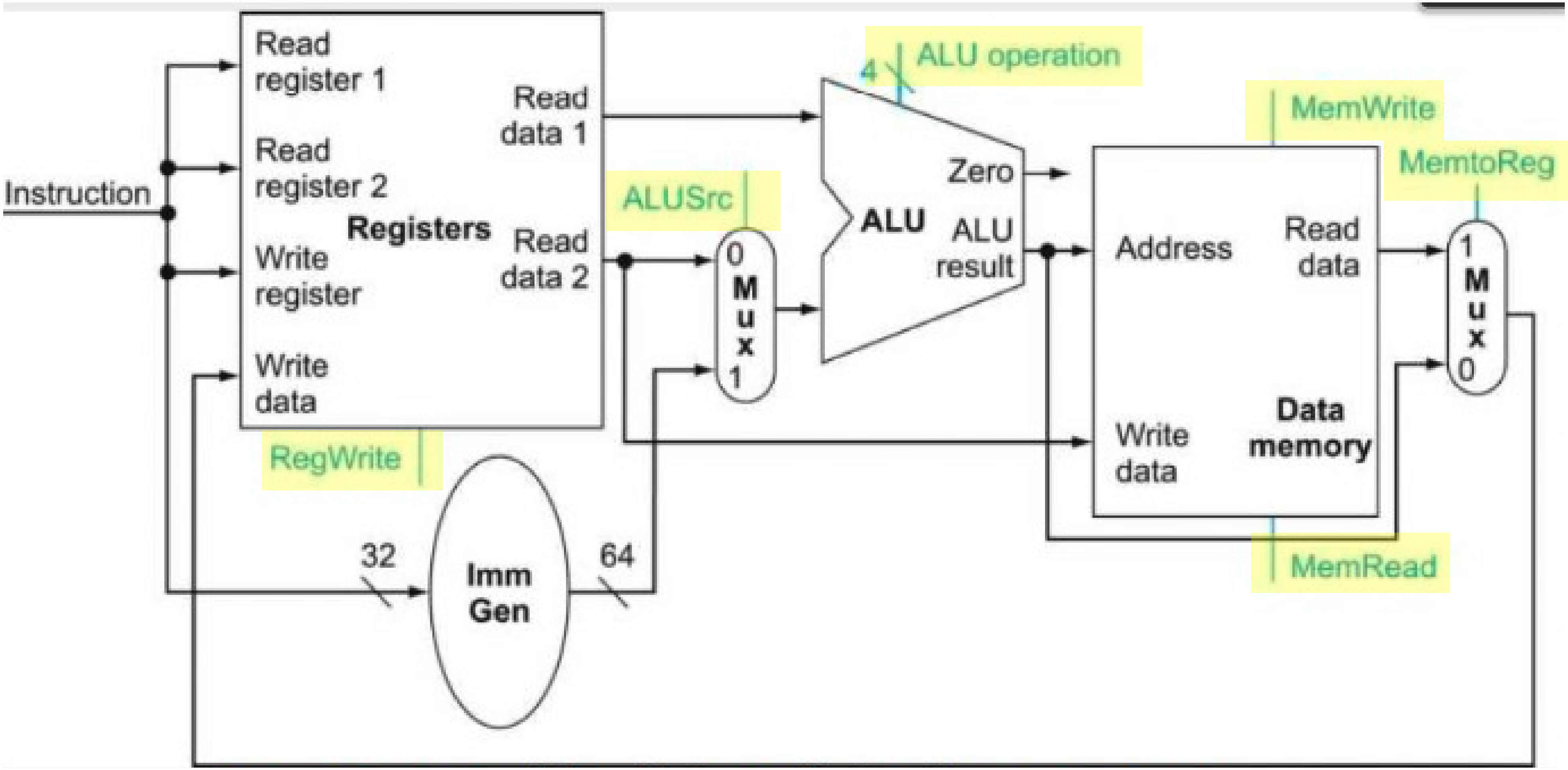


FIGURE 4.10 **The datapath for the memory instructions and the R-type instructions.**

Tutorial Answer #3

- (A) **Question:** What are the values of control signals generated by the control in Figure 4.10 for this instruction?
- **Precursor:**
  - First of all, what are control signals?
    - In short, control signals are generated by the *Control*, and these signals are used to influence the behaviour of multiplexers, *mux*.
  - So what is the question asking?
    - *Next slide*

Tutorial Answer #3

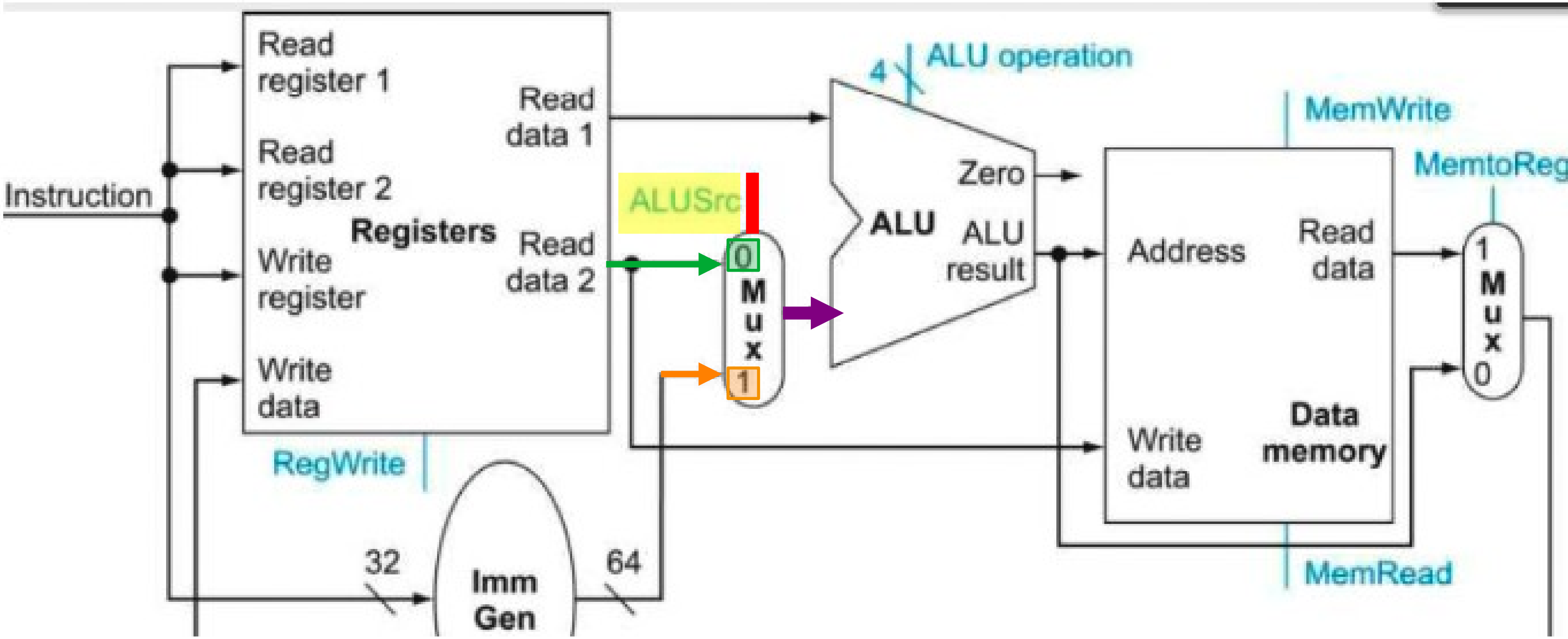


- Basically, the question is asking you to calculate/determine the values highlighted in yellow
  - These are control signals, and they are determined by the *Control*

Tutorial Answer #3

- (A) **Question:** What are the values of control signals generated by the control in Figure 4.10 for this instruction?
- **Precursor:**
  - With the the exception of *ALUoperation*, the value of the signals are either **0/1** or **true/false**
    - "Technically", you can say that:
      - **0** = **False** = Low Signal
      - **1** = **True** = High Signal
  - Remember, these values control the behaviour of the multiplexer
    - Next slide for more information

Tutorial Answer #3



- Let's focus on the ALUsrc *mux*
  - If the *ALUsrc* is "not activated", which means that a "low signal" is sent from the *Control*, then the value/data that corresponds to **0** is passed forward
  - If the *ALUsrc* is "activated", which means that a "high signal" is sent from the *Control*, then the value/data that corresponds to **1** is passed forward

Tutorial Answer #3

- **(A) Question:** What are the values of control signals generated by the control in Figure 4.10 for this instruction?
- **Precursor:**
  - We need to determine the values of:
    - RegWrite (Are we writing to a register?)
    - ALUsrc (What value is being passed forward? *Read Data 2* or *Imm Gen*?)
    - ALUoperation (What operation/instruction is being performed in the ALU?)
    - MemWrite (Are we writing to memory?)
      - In other words, *MemWrite* enables a memory write, which is used for store (i.e. sw) instructions
    - MemRead (Are we reading from memory?)
      - In other words, *MemRead* enables a memory read, which is used for load (i.e. lw) instructions)
    - MemToReg (Determines where the value to be written comes from (i.e. Is the value from the ALU or from memory?))

Tutorial Answer #3

- **(A) Question:** What are the values of control signals generated by the control in Figure 4.10 for this instruction?
- **Answer:** (Fill in the table)

	RegWrite	ALUsrc	ALU-operation	MemWrite	MemRead	MemTo-Reg
Options	*True {OR} False*	*0 {OR} 1*	*“instruction”*	*True {OR} False*	*True {OR} False*	*0 {OR} 1*
Answer	True	0	“and”	False	False	0

Tutorial Explanation #3

- **Explanation:**
  - RegWrite = True
    - Yes, we are writing to a register. The *and* operation performs a logical *AND* on two values and <u>stores</u> the result in the <u>destination register</u>. Hence, we are writing to a register
      - *If the instruction was "sw", then this would be False*
        - *If the instruction was "lw", then this would be True*
  - ALUsrc = 0
    - We want the value from <u>*Read data 2*</u> to be forwarded by the <u>*ALUsrc mux*</u>. Hence, a low signal should be passed to the <u>*ALUsrc mux*</u> by the Control
      - Thus, the signal for *ALUsrc* is **0**.
  - ALUOperation = "and"
    - The operation the ALU needs to perform is logical AND. This information is inferred from the instruction.

Tutorial Explanation #3

- **Explanation:**
  - MemWrite = False
    - Since we are only dealing with registers (i.e. rs1, rs2, rd), and we are not writing to memory, *MemWrite* is False
      - If the instruction was store *(i.e. sw)*, then *MemWrite* would be True
  - MemRead = False
    - Since we are only dealing with registers (i.e. rs1, rs2, rd), and we are not reading from memory, *MemRead* is False
      - If the instruction was load *(i.e. lw)*, then *MemRead* would be True
  - MemToReg = 0
    - Since the "*and*" instruction has nothing to do with Data memory, this value is 0
      - So the *MemToReg* multiplexer (mux) is sent a low signal, or not activated, and the result is written back to the destination register *(i.e. rd)*

What Does It Look Like?

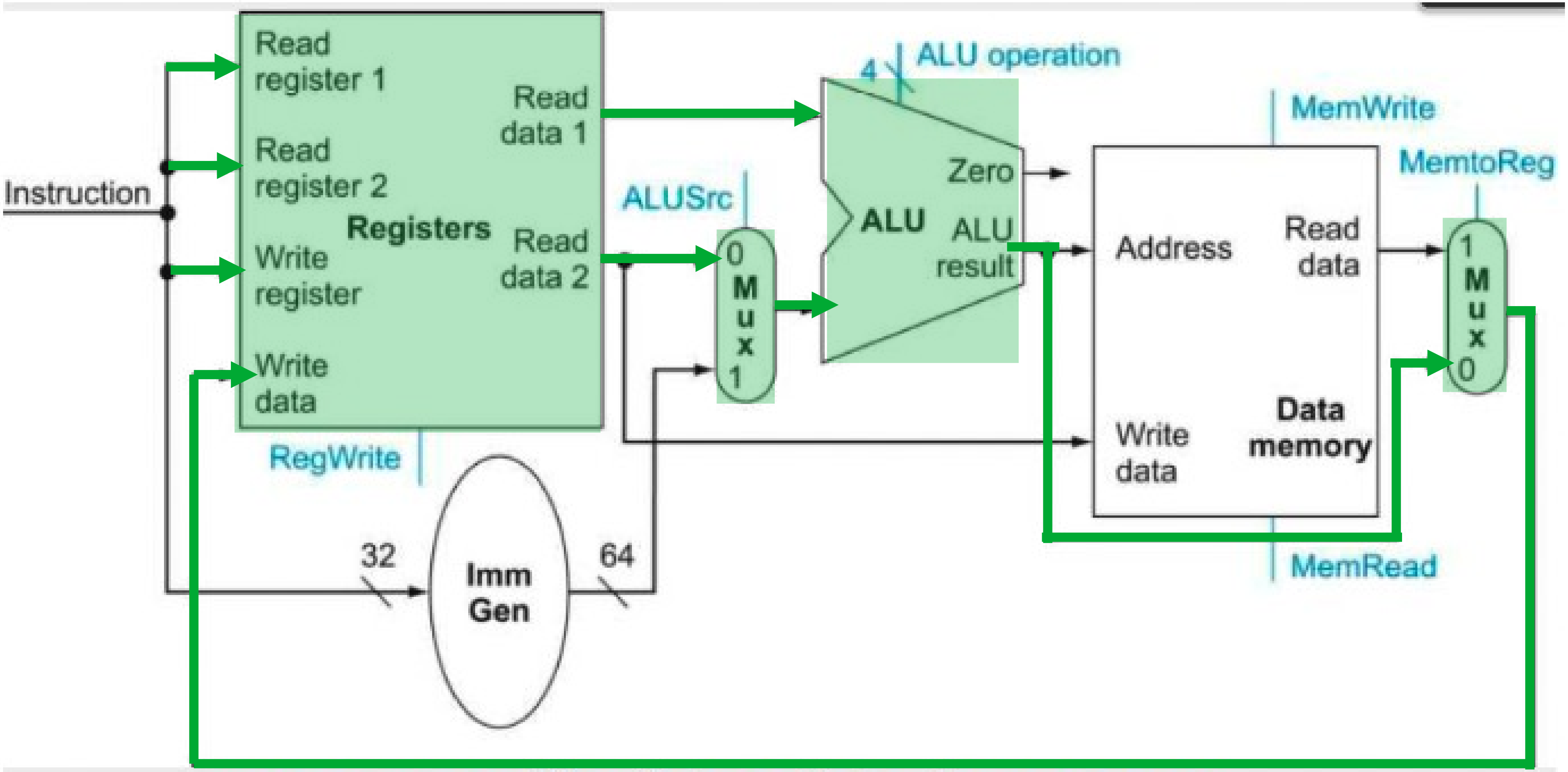


FIGURE 4.10 **The datapath for the memory instructions and the R-type instructions.**

Tutorial Answer #3

- (B) **Question:** Which resources *(or blocks/chunks)* perform a useful function for this instruction?
- **Answer:** Trace through it
  - *Next slide*

Tutorial Answer #3

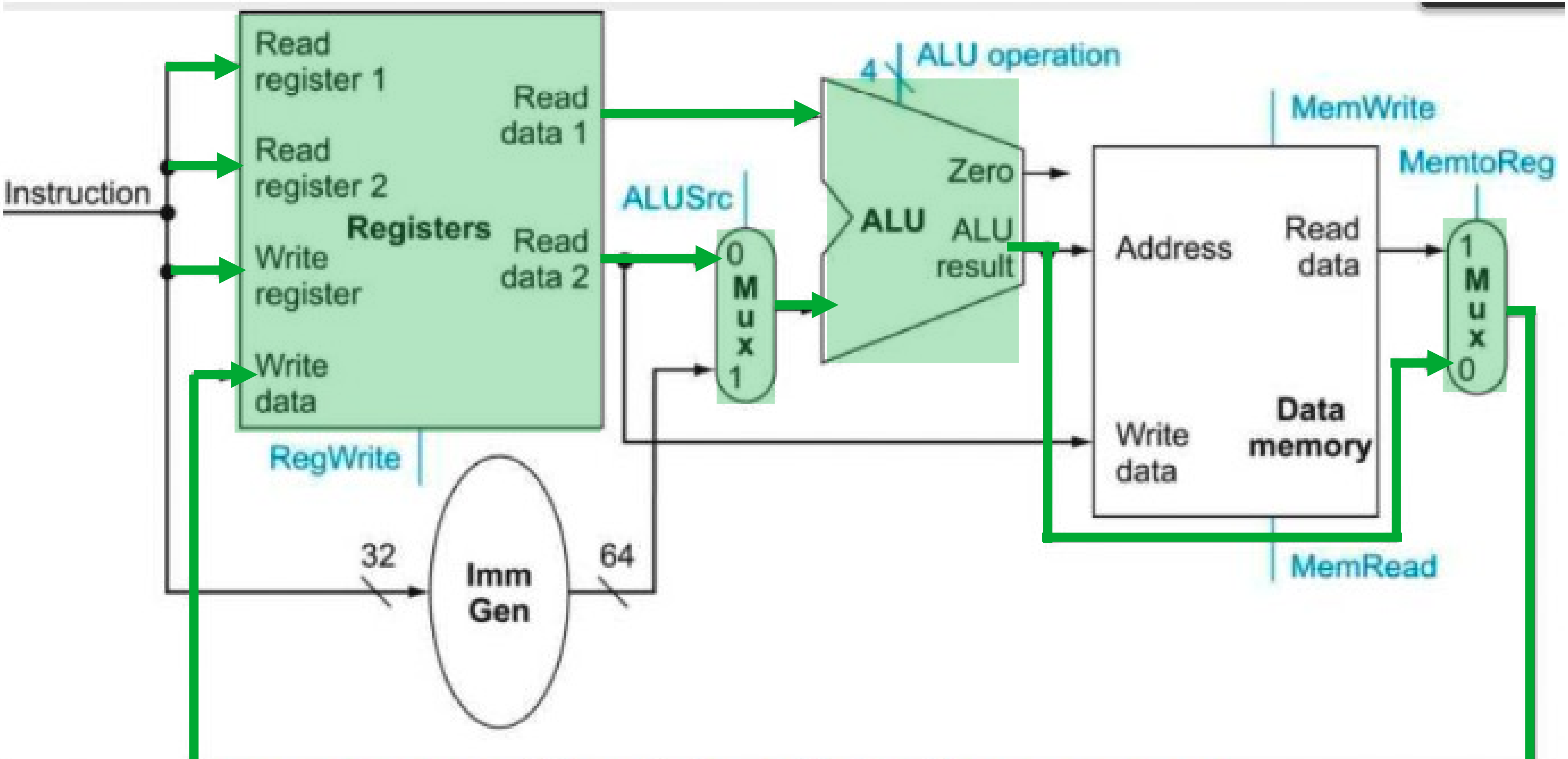


FIGURE 4.10 The datapath for the memory instructions and the R-type instructions.

Tutorial Answer #3

- **(B) Question:** Which resources *(or blocks/chunks)* perform a useful function for this instruction?
- **Answer:**
  - As per the previous slide, the blocks highlighted in green are used to perform a useful function for the instruction, "*and*". *These blocks are:*
    - Registers
    - ALUsrc mux
    - ALU
    - MemtoReg mux

Tutorial Answer #3

- (C) **Question:** Which resources *(or blocks/chunks)* produce no output for this instruction? Which resources produce output that is not used?
- **Precursor:**
    - We already know what blocks produce useful output
        - So, the remaining blocks produce no useful output
            - BUT, all blocks produce output. However, we don’t use it because it is useless
    - *Next slide*

Tutorial Answer #3

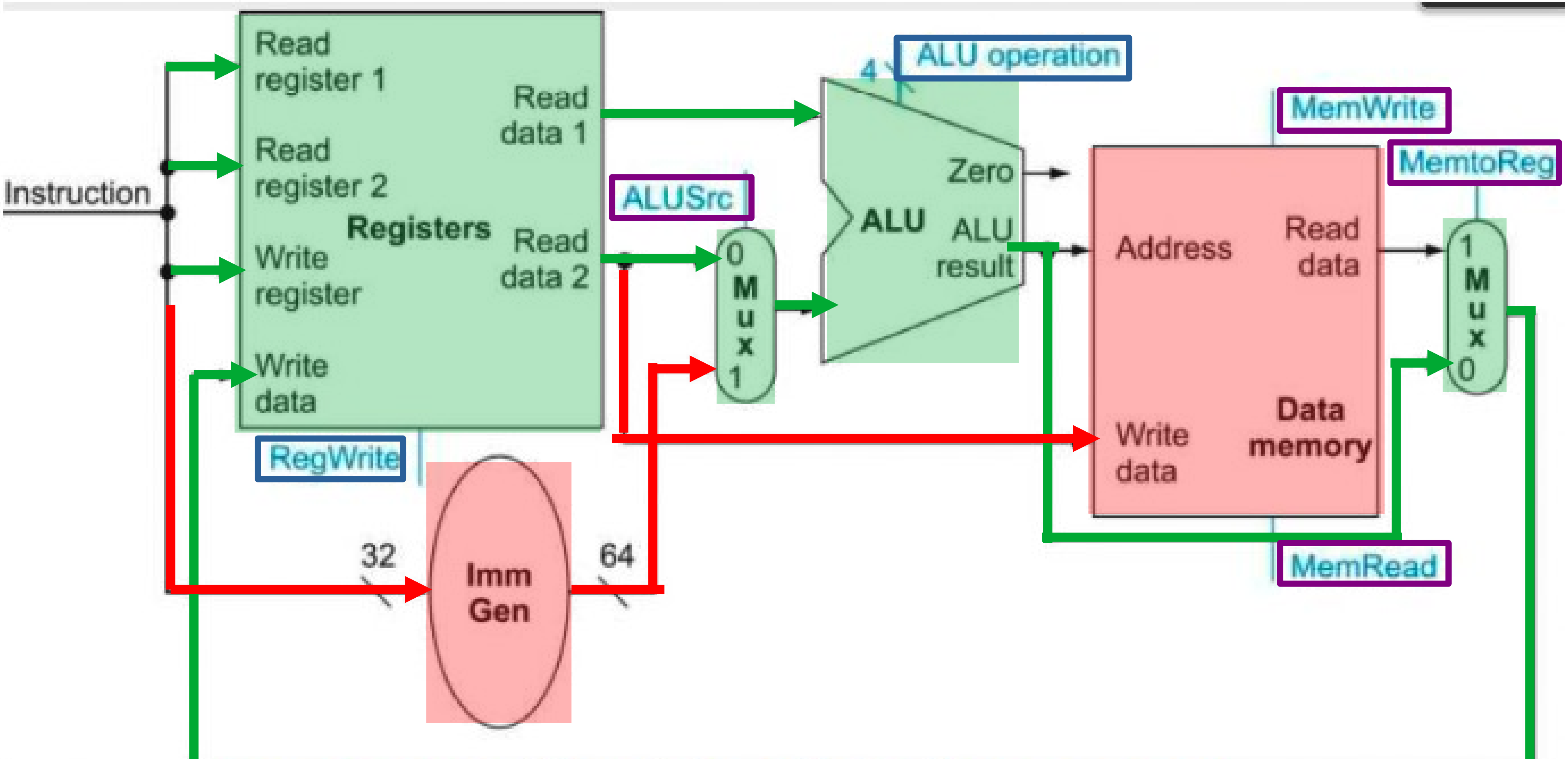


FIGURE 4.10 The datapath for the memory instructions and the R-type instructions.

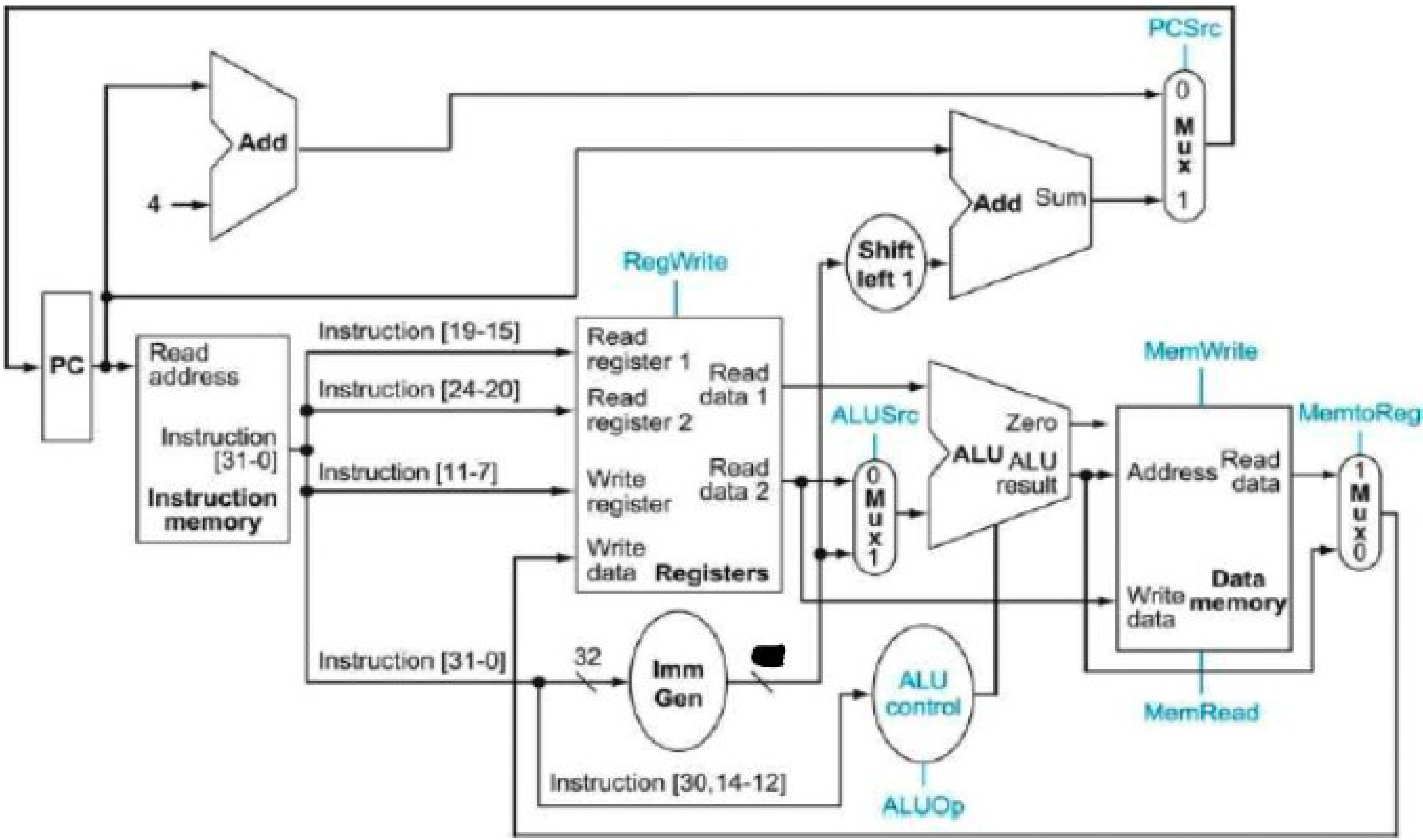
Tutorial Answer #3

- **(C) Question:** Which resources *(or blocks/chunks)* produce no output for this instruction? Which resources produce output that is not used?
- **Answer:**
  - As per the previous slide, the blocks in green produce useful output, and the blocks in red do not produce useful output
    - The control signals outlined in blue are sent a *high* signal, and the control signals outlined in purple are sent a *low* signal
      - *The signals are sent by the Control*
  - i) All blocks/chunks produce output for this instruction
  - ii) The outputs of *Imm Gen* and *Data Memory* are not used

Important Question

- **Question:** Why have *MemRead* and *MemToReg*?
  - *Asked By Emily*
- **Answer:** Why have *MemRead* and *MemToReg*?
  - *MemRead* enables a memory read for load instructions
    - *i.e. lw, ld, etc.*
  - *MemToReg* determines where the value to be written comes from *(i.e. ALU or Memory)*
    - *For the previous question, the value is a result from the ALU, hence the control signal for MemToReg is **0***

Tutorial Question #4



Tutorial Question #4

- **Question:** Using the diagram on the previous slide, answer the following questions. Problems in this exercise refer to a clock cycle in which the processor fetches the following instruction word: <u>*0x00c6aa23*</u>
  - **A)** What is the encoded instruction?
  - **B)** What are the values of the ALU control unit's inputs for this instruction?
  - **C)** What is the new PC address after this instruction is executed?
  - **D)** For each mux, show the values of its inputs and outputs during the execution of this instruction. List values that are register outputs at Reg [xn].
  - **E)** What are the input values for the ALU and the two add units?
  - **F)** What are the values of all inputs for the registers unit?

Tutorial Answer #4

- **(A) Question:** What is the encoded instruction?
- **Answer:**
  - Step 1) We are given a value in hex, and to get the instruction, we need to convert it into binary
    - 0x00c6aa23 = 0000 0000 1100 0110 1010 1010 0010 0011
  - Step 2) Once we have the corresponding binary value, we look at bits *[0:6]* to narrow down the instruction type, and then we look at bits *[12:14]* to get the exact instruction
    - 0000 0000 1100 0110 1**010** 1010 0**010 0011**
      - 010 0011 tells us that the instruction can be *sb*, *sh*, or *sw*
      - 010 tells us that the instruction is *sw*

Tutorial Answer #4

- (A) **Question:** What is the encoded instruction?
- **Answer:**
  - Step 3) Now that we know the instruction is *sw*, let's divide the bit pattern as per the instruction set
    - 0000000 01100 01101 010 10100 0100011
      - 0100011 = opcode = **sw**
      - 10100 = imm[4:0] = $2^2 + 2^4$ = **20**
      - 010 = funct3
      - 01101 = rs1 = $2^0 + 2^2 + 2^3$ = **13**
      - 01100 = rs2 = $2^2 + 2^3$ = **12**
      - 0000000 = imm[11:5] = 0
  - Step 4) We know that the format of an *sw* instruction is: ***sw rs2, imm(rs1)***
    - So, when we substitute the values, we get: ***sw x12, 20(x13)***
      - *Don't forget to add '**x**' to indicate registers*

Tutorial Answer #4

- (B) **Question:** What are the values of the ALU control unit's inputs for this instruction?
- **Answer:** (Refer to table below)

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
ld	00	load register	XXXXXX	add	0010
sd	00	store register	XXXXXX	add	0010
beq	01	branch on equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001

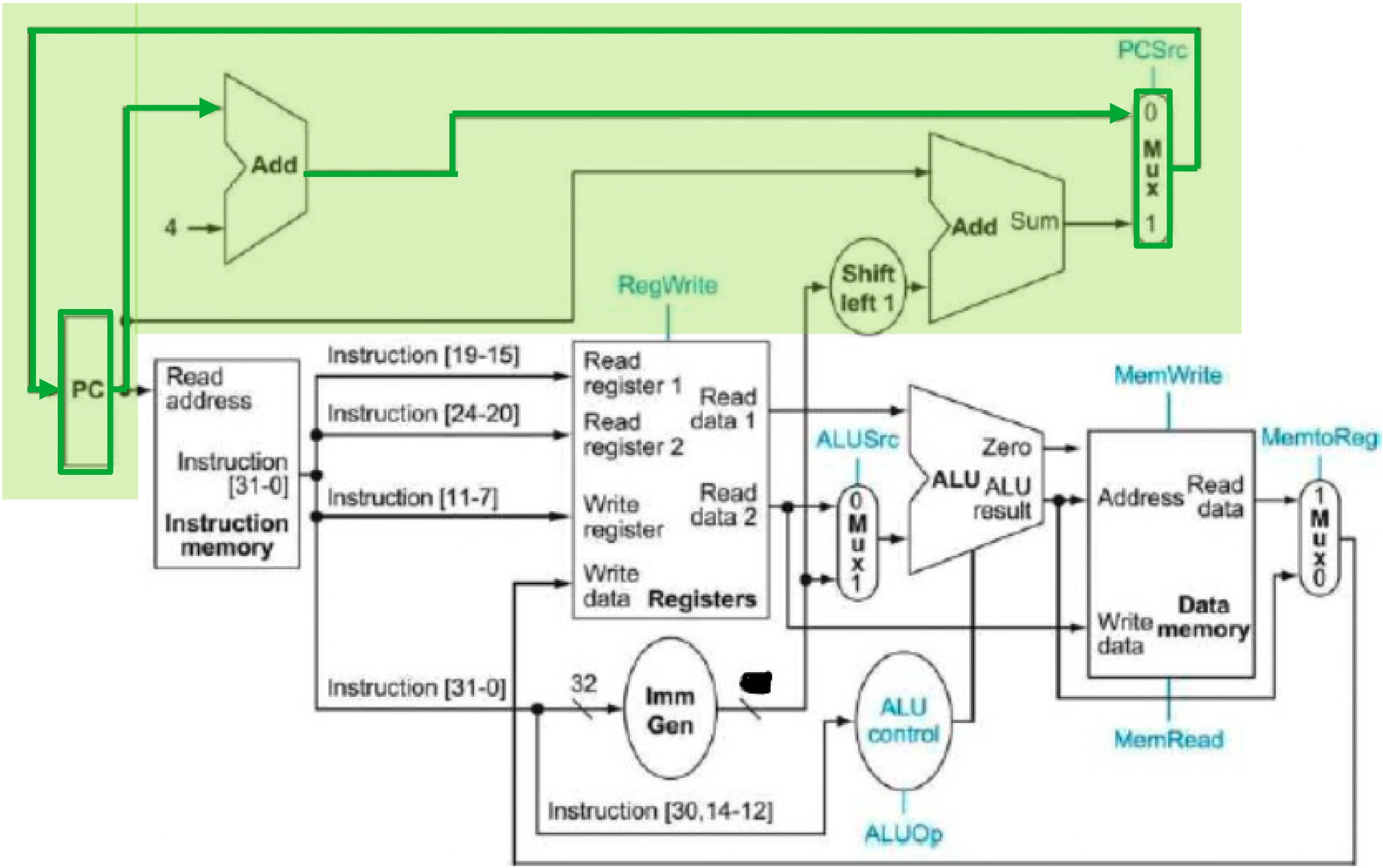
Tutorial Answer #4

- **(B) Question:** What are the values of the ALU control unit's inputs for this instruction?
- **Answer:**
  - We need the values for ALU control and ALUop
    - According to the table on the previous slide:
      - <u>ALUop = 00</u>
      - <u>ALU control = 0010</u>
  - *Note: I got the table from Dr. Nokovic's slides*
    - *i.e. Slide #34 in Chapter4.pdf*

Tutorial Answer #4

- (C) **Question:** What is the new PC address after this instruction is executed?
- **Answer:**
  - Recall that PC stands for Program Counter
    - *It indicates where a computer is in its program sequence*
  - Since we are not jumping or branching in a *store (i.e. sw)* instruction, the new PC is the old PC + 4
    - $\underline{PC_{new} = PC_{old} + 4}$
      - The signal travels through the *adder*, *PCsrc mux*, and then back to the PC block
        - The *adder* is where 4 is added to the PC
          - *See next slide*

Program Counter Trace



Tutorial Answer #4

- (D) **Question:** For each mux, show the values of its inputs and outputs during the execution of this instruction. List values that are register outputs at Reg [xn].
- **Answer:**
  - There are 3 multiplexers (mux) we need to account for
    - i. ALUsrc mux
      - The Control sends *ALUsrc* a high signal (1), because we need to send the immediate value, which is the offset, to the ALU
    - ii. MemToReg mux
      - The Control sends *MemToReg* a low signal (0), because it indicates that the value to be written to memory comes from the ALU
    - iii. PCsrc mux
      - The Control sends *PCsrc* a low signal (0), because this is a store instruction, and no branching or jumping is involved. After the store instruction is executed, we move on to the next instruction

Tutorial Answer #4

- (D) **Question:** For each mux, show the values of its inputs and outputs during the execution of this instruction. List values that are register outputs at Reg [xn].
- **Answer:** **(ALUsrc mux) [Signal = High (1)]**
  - Inputs:
    - Reg[x12] = rs2 = register 2 = read data 2
    - 0x00000014 = $(20)_{10}$ = Immediate value
  - Output
    - 0x00000014 = $(20)_{10}$ = Immediate value
      - *The output is the immediate value $(20)_{10}$, because the ALUsrc is sent a high signal (i.e. 1) from Control. Since we need to calculate the memory address, we need to send the base address and offset to the ALU. Hence, ALUsrc is turned on, so the ALU can compute the memory address where information needs to be stored. The ALUsrc forwards the immediate value to the ALU.*

Tutorial Answer #4

- (D) **Question:** For each mux, show the values of its inputs and outputs during the execution of this instruction. List values that are register outputs at Reg [xn].
- **Answer:** (MemToReg mux) [Signal = Low (0)]
  - Inputs:
    - Reg[x13] + 0x00000014
      - This is the location in memory (base address + offset) that we need to write to. Remember, we are storing data into memory
    - <undefined>
      - The value that comes from Read data is undefined
  - Output
    - <undefined>
      - The output of the MemToReg mux is undefined, and we don't care, because we are only interested in storing/writing the value to data memory. We are not writing to a register; there is no write back (WB).

Tutorial Answer #4

- (D) **Question:** For each mux, show the values of its inputs and outputs during the execution of this instruction. List values that are register outputs at Reg [xn].
- **Answer:** (PCsrc mux) [Signal = Low (0)]
  - Inputs:
    - PC + 4
      - This input is the program counter plus 4; the addition is done by the adder
    - PC + 0x00000028
      - This input is the program counter plus the immediate value shifted left, once
        - The immediate value = (0x00000014) = $(20)_{10}$
          - A left shift on 0x00000014 = 0x00000014 * 2 = 0x00000028
          - (0x00000014) = $(20)_{10}$
          - (0x00000028) = $(40)_{10}$
  - Output
    - PC + 4
      - Since the Control sends the *PCsrc mux* a low signal (i.e. 0) , the value that is forwarded is (PC + 4)
        - *(PC + 4) points to 0 on the PCsrc mux*
        - *(PC + 0x00000028) points to 1 on the PCsrc mux*

Tutorial Answer #4

- **(E) Question:** What are the input values for the ALU and the two add units?
- **Answer:**
  - Basically, you need to list the inputs for the following blocks:
    - ALU
      - *The ALU takes the base address and the offset*
    - First Adder (Left Side)
      - *Takes the PC and 4 as its inputs*
    - Second Adder (Right Side)
      - *Takes the PC and immediate value shifted once as its inputs*

Tutorial Answer #4

- **(E) Question:** What are the input values for the ALU and the two add units?
- **Answer:** (ALU)
    - Input
        - Reg[x13]
            - This is the base address
        - 0x00000014
            - This is the offset

Tutorial Answer #4

- (E) **Question:** What are the input values for the ALU and the two add units?
- **Answer:** (First Adder, Left Side)
  - Input
    - PC
      - This is the address of the current instruction
    - 4
      - This is added to the program counter to get to the next instruction

Tutorial Answer #4

- **(E) Question:** What are the input values for the ALU and the two add units?
- **Answer:** (Second Adder, Right Side)
  - Input
    - PC
      - This is the address of the current instruction
    - 0x00000028
      - This value is the immediate value multiplied by 2. It is multiplied by 2 because a logical shift is done
        - (srli (0x00000014, 1)) = (0x00000028)

Tutorial Answer #4

- **(F) Question:** What are the values of all inputs for the registers unit?
- **Answer:**
  - We need to look at it the Registers (file) to determine what the <u>inputs</u> are for the register units
    - The input units for the Registers (file) are:
      - Read register 1
        - x13 (This is the base address)
      - Read register 2
        - x12 (This is the data that needs to be stored)
      - Write Register
        - X (Don’t care; there shouldn’t be any write backs)
      - Write Data
        - X (Don’t care; there shouldn’t be any write backs)
      - RegWrite
        - False (Since we are not writing to a register, this is false)

THE END