Computer Organization
5-stage RISCV32I Processor
Phase 4: Block Diagram
Fall 2022

Objective: An important part of any design is a well made block diagram. Block diagrams give us a better understanding of a system's functions; they also help with visualizing and designing the interconnections between components. A well made block diagram will allow for quicker development of the code and will allow for a better understanding of the pipeline as well.

In this phase we will be implementing the block diagram for a 5 stage pipeline. The phase will include adding the components, the signal lines, the signal names, and the pipeline registers. The block diagram we are creating in this phase will be functional for the i-type, r-type, and r-type immediate instructions. These are the types of instructions that phase 5 will start out implementing.

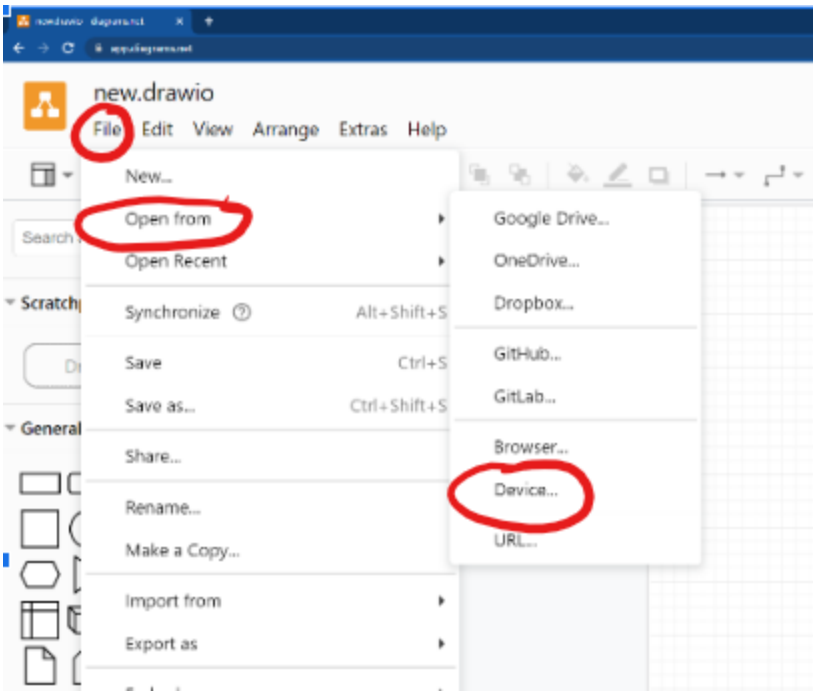
Key Learning Outcomes of this phase:
Block Diagram Creation: Through the creation of this block diagram, the phase will focus on how signals move through our processor. It will specifically show which signals need to move between stages and what they are used for within each stage. The purpose of this block diagram is to fully encapsulate the flow of information and instructions from the Instruction Fetch (IF) stage down to the Write Back (WB) stage.

Due Dates:

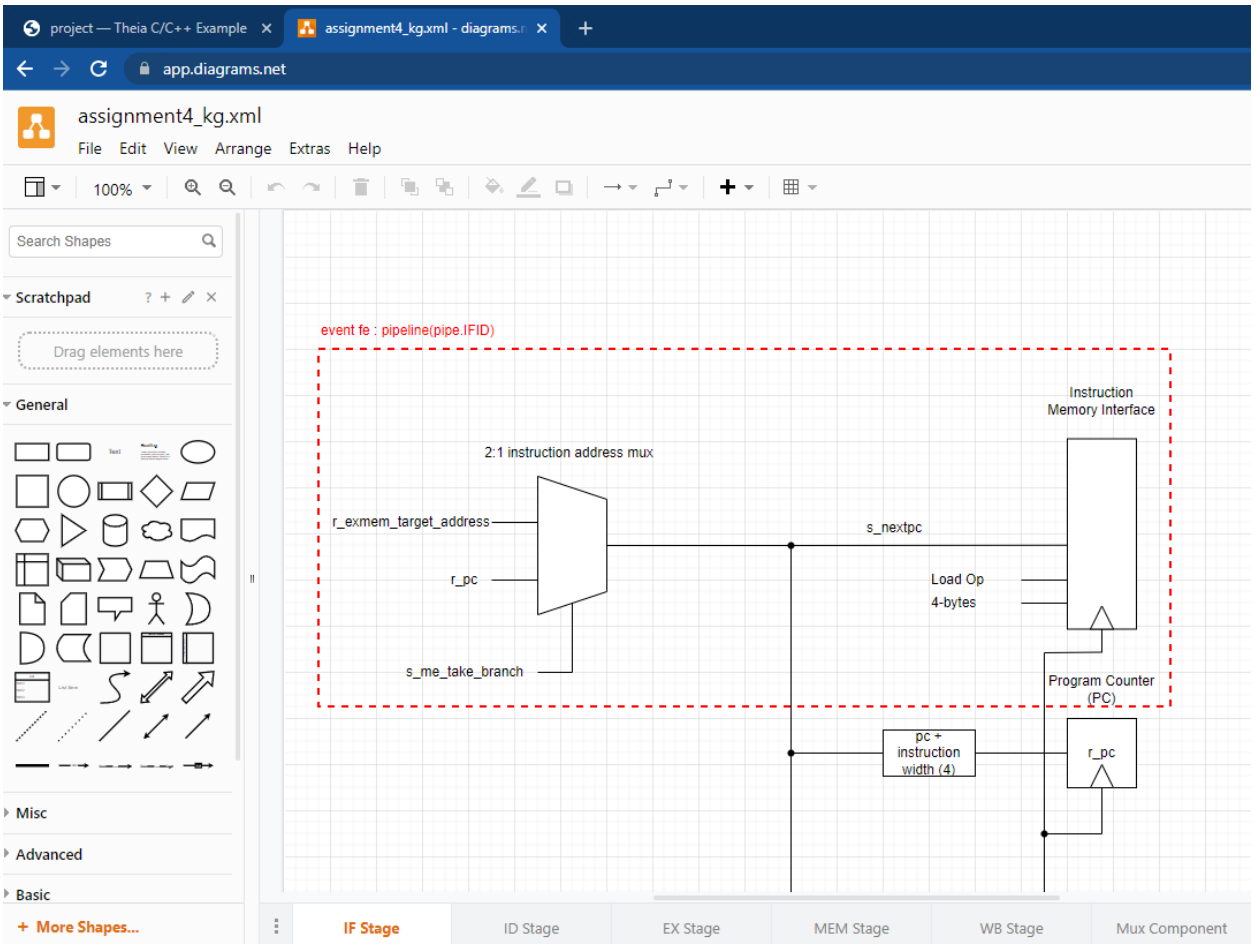
- This is a 1-week phase
- Due date: Monday October 10th 2022 11:59pm.

Instructions:

- Download phase4.xml provided by the instructing team.
- Rename your project by adding your first and last initials to the end project name such as:
    - Phase4_fl (f = your first name initial, l = your last name initial)
- You will be using the program **draw.io** as your block diagram editor. Please open **draw.io** by typing **draw.io** in your browser address bar and then return
- If upon opening **draw.io**, it does not ask which file to open, you can use the next set of instructions.
    - When requested, you should select from **Device** and open from **Existing file**
- Open your phase by selecting File from the menu bar and then selecting "**Open from**" and "**Device…**"



- “**Device…**” will open a file browser that you will then transverse to your “phase4_fl” file to select to open.
- Once open, it should look like the following:



IF Stage

The IF stage is known as the Instruction Fetch stage. Here, the PC address is calculated and incremented accordingly. Any changes in program flow are executed in this stage.

ID Stage

The ID stage is known as the Instruction Decode stage; there are many things that occur here. First, the instruction is parsed, meaning that the rs1, rs2, rd, the opcode, and the immediate values are pulled from the 32 bit instruction. Next, from the opcode, the control signals for the other stages of the pipeline are set and sent to the IDEX (Instruct Decode/Execute) pipeline register. Finally, the rs1 and rs2 values are used by the register file to find the values of the given rs1 and rs2 operands.

EX Stage

The EX stage is known as the Execute stage. This is where the ALU is located and where the actual computation takes place.

MEM Stage

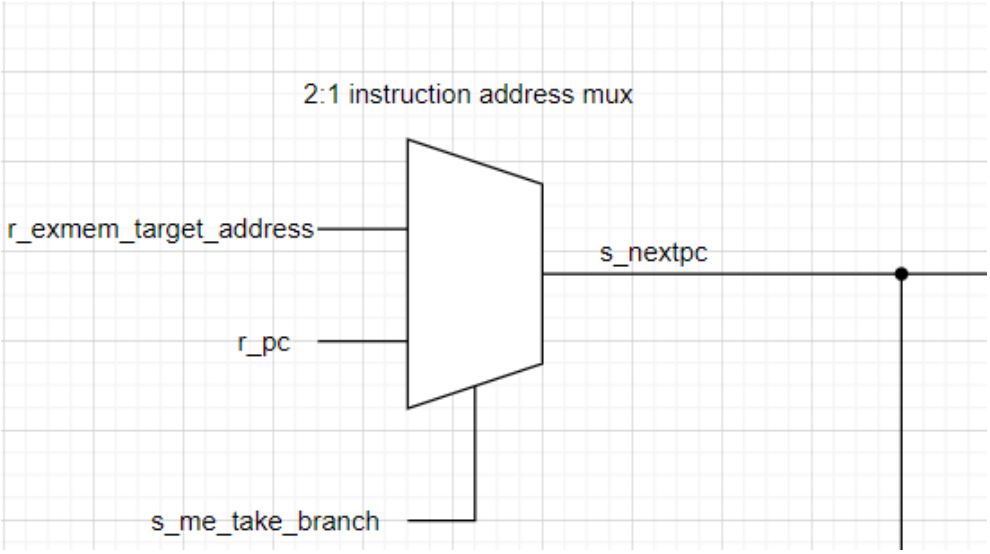
The MEM stage is known as the Memory stage. If there is a memory operation such as a load or a store, this is where it will either load the value from memory or store a value into memory.

WB Stage

The WB stage is known as the Write Backstage. Within this stage the value that was calculated is written into the register file.

IF Stage

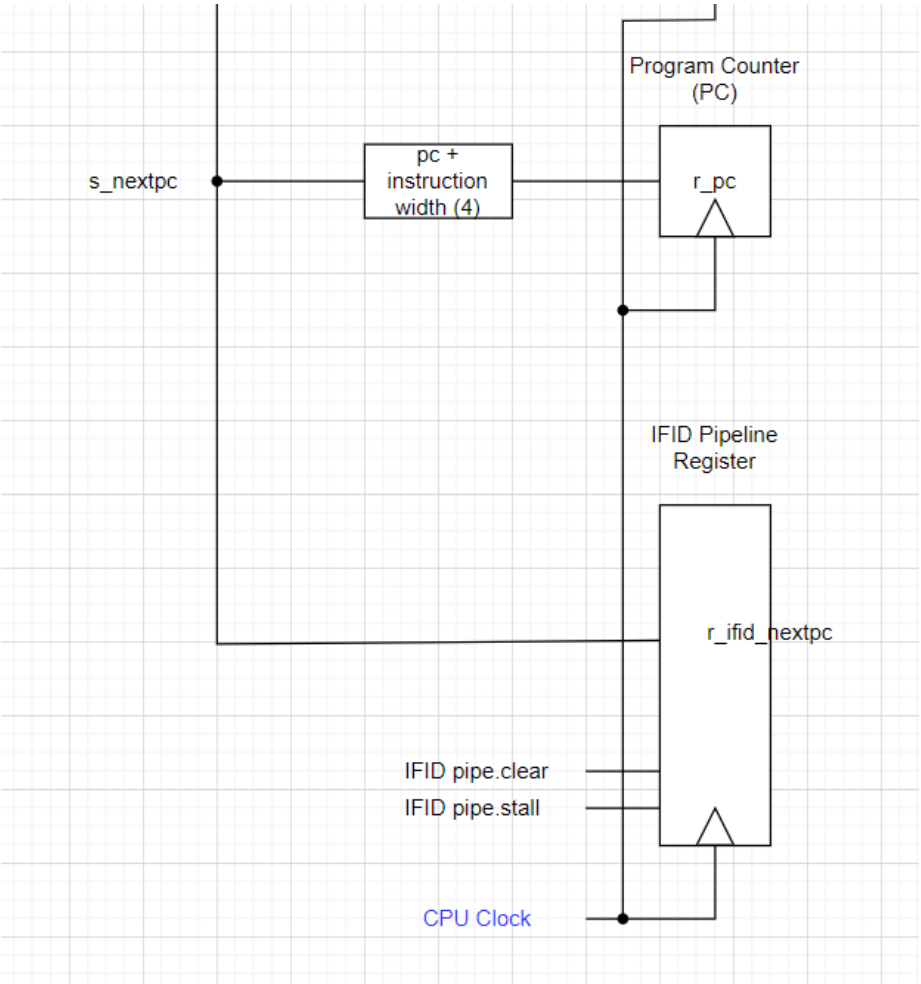
- Switch to the IF stage tab/
- You will **not** be changing the block diagram of this tab as it is already implemented for you.
- This section contains the logic for later phases and the upcoming phase 5. Looking at the diagram, the first component is a 2:1 mux as shown below.



- There are 2 register values and a control signal fed into the mux. The s_me_take_branch is a signal from the mem stage that will tell us if we are taking a branch or not. If we are taking a branch, the mux will set s_nextpc = r_exmem_target_address, otherwise s_nexpc = r_pc. This mux focuses on material in later phases, so don't worry too much about these signal names not appearing in the later sections of the document.

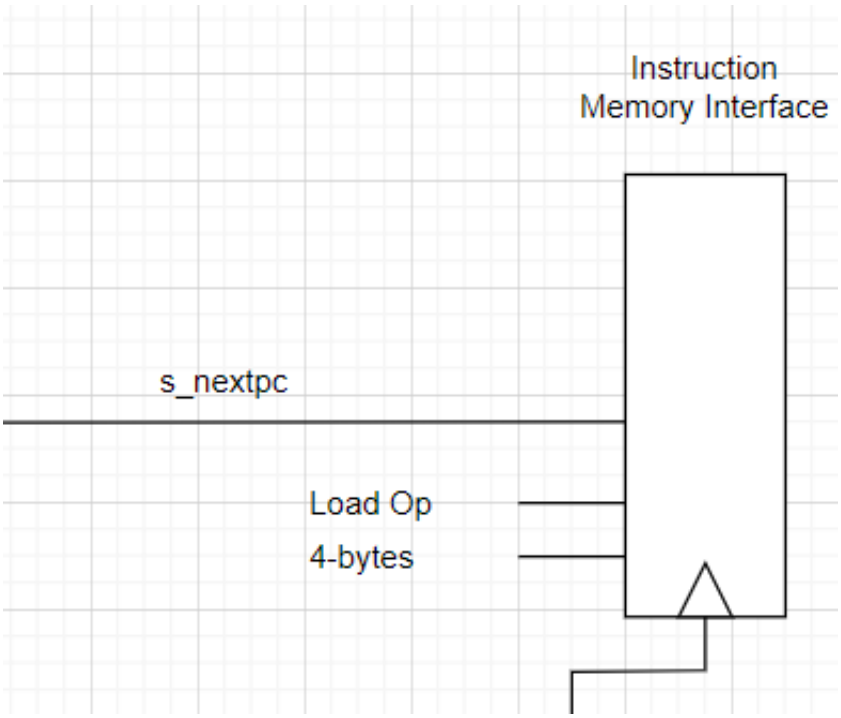
- Signal and Register Nomenclature
  - In this course, all global signals, combinatorial logic that is derived from logic and not a register or flip-flop will begin with s_ and all register output values will begin with r_. To easily identify what pipeline stage a signal/register belongs to, the next part of the name will either be the pipeline stage such as ID or the pipeline register, IDEX signifying the pipeline register between the ID and EX stages. The next term after the signal/register location is the actual name such as pc for program counter.
    - s_ signals will always be identified with the pipeline stage that the combinatorial signal is created such created in the IF, ID, EX, MEM, or WB stage
    - There are several registers that are independent of a stage or pipeline register that have no location designation such as the Processor Counter, r_pc
  - ex: r_idex_pc: IDEX pipeline register for the Program Counter (PC)

- The next section of the IF stage is the IFID pipeline register and the addition of the r_pc. That section is shown in the screenshot below.



- The s_nextpc is sent to the IFID pipeline register where it will be set to r_ifid_nextpc in the next stage of the pipeline. It will also be used to calculate r_pc for the next cycle. To do that, the r_pc is set to s_nextpc + 4 as shown at the top of the screenshot. Finally, the IFID pipe.clear and the IFID pipe.stall are signals that can be set in codasip to either stall or clear the pipeline. However, for now, we do not have to worry about their application as they will be discussed in later phases.

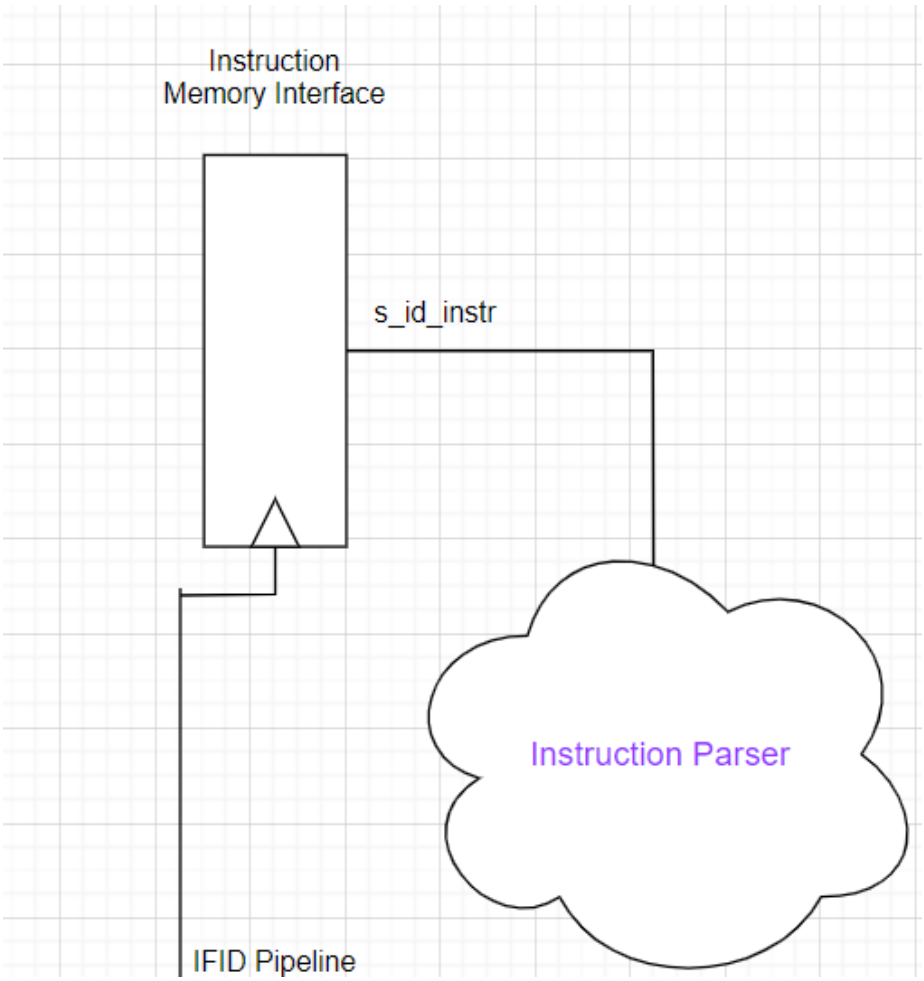
- The final part of the IF stage is the Instruction Memory Interface as shown in the screenshot below.



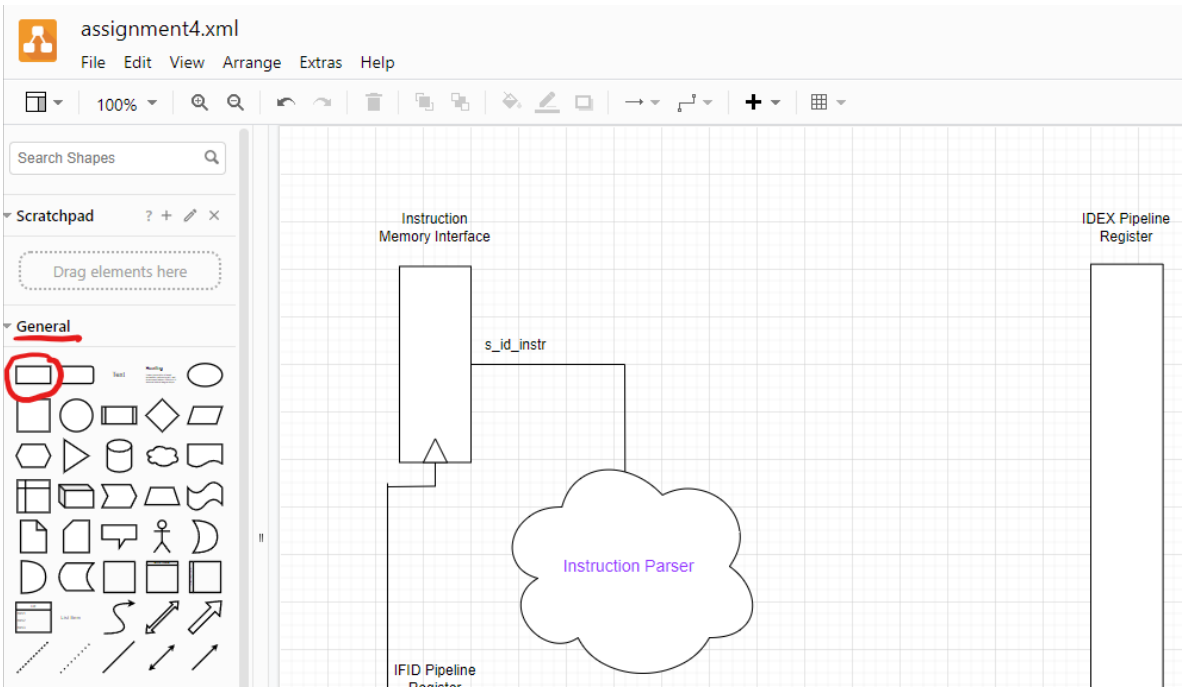
- This component will fetch the instruction that is at the address of s_nextpc. The instruction at that address will be used in the next stage of the pipeline, Instruction Decode.

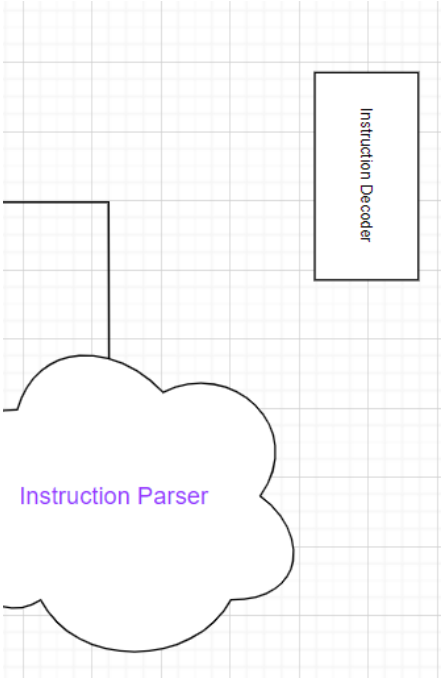
ID Stage

- After reviewing the contents of the IF stage, **switch the tab to the ID stage**.
- Within this section, there are signals and components that need to be added. We will go into each component separately and add the signals as required.
- The first component to look at in this section is the instruction parser as shown in the screenshot below.

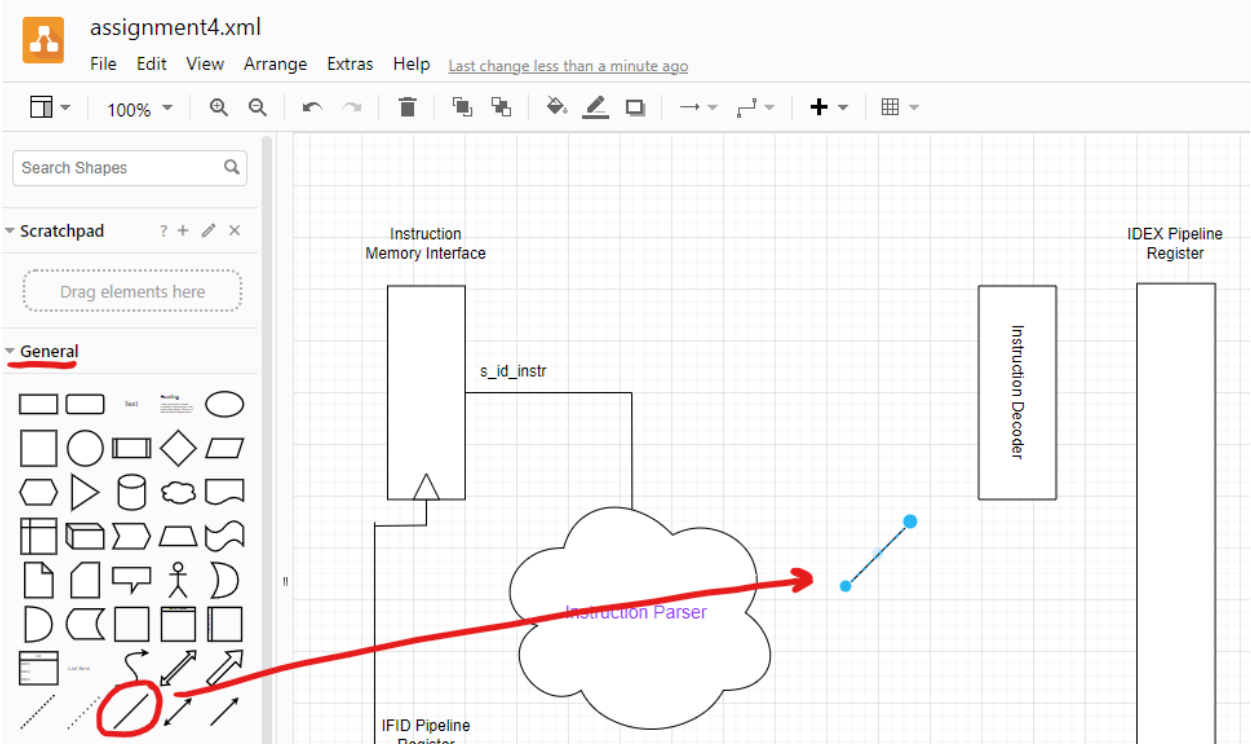


- As discussed earlier, the Instruction Memory Interface will output the 32 bit instruction. That instruction is sent to the instruction parser which gets the rs1, rs2, rd, the opcode, and the immediate value. Those values will be used by other components in the ID stage.
- Moving onto the next component, we need to add the decoder. To do that, add a rectangle, change the outline to black, and put the label “Instruction Decoder” on it as shown below.
    - You can find the rectangle to add in **draw.io** to the left of the workspace
    - Click on the rectangle and it will bring up a rectangle in your workspace. You will then drag the rectangle to where you would like to locate it in your block diagram and rotate or resize it so that it is vertical
    - To add the text inside the rectangle, with the rectangle selected, double click the inside of the rectangle and a dialogue box should appear for you to enter the text for the rectangle

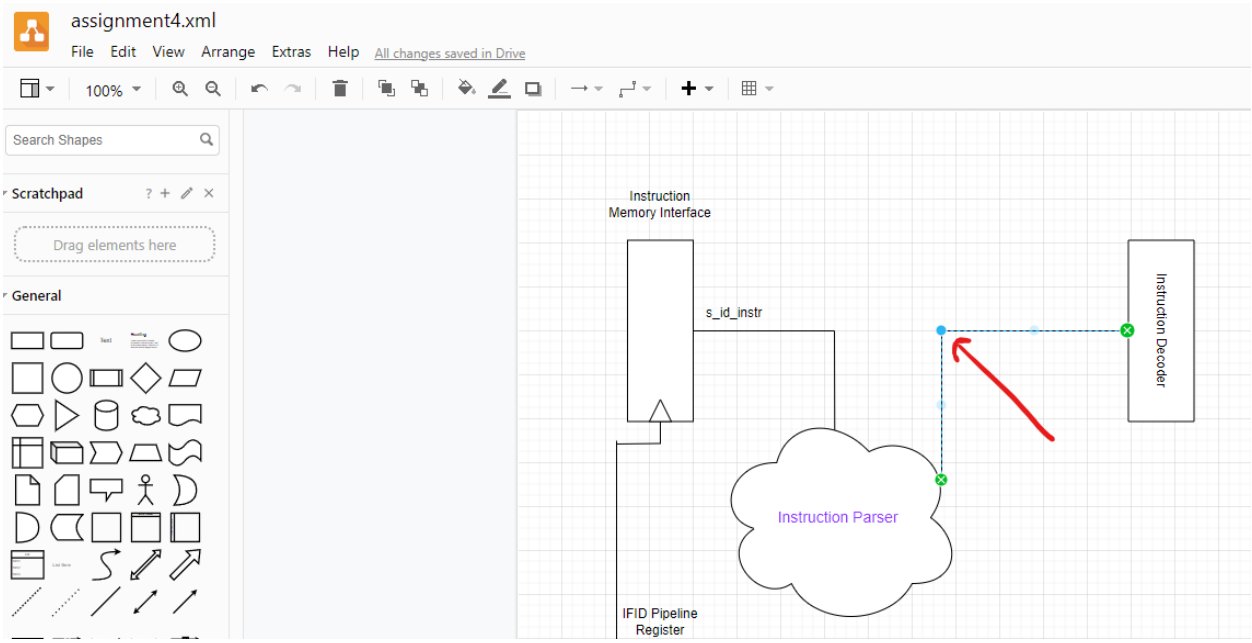




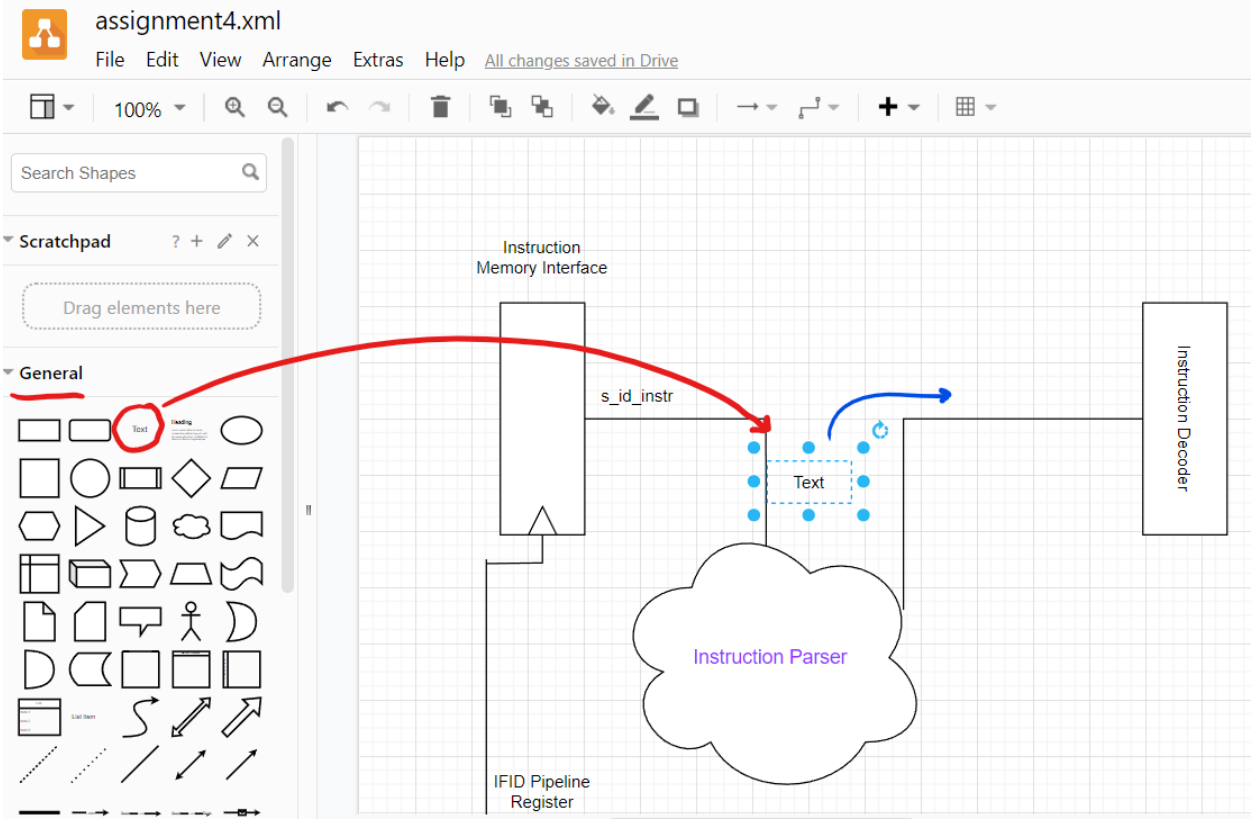
- Next, add a signal starting at the “Instruction Parser” cloud ending at the left side of the instruction decoder rectangle. The name of that signal is s_id_opcode. Change the color of the text for s_id_opcode to purple since it came from the instruction parser.
    - You can add a signal to your workspace by clicking the line in the left hand set of images and dragging one end point to the “Instruction Parser” cloud and the other to the Instruction Decoder.



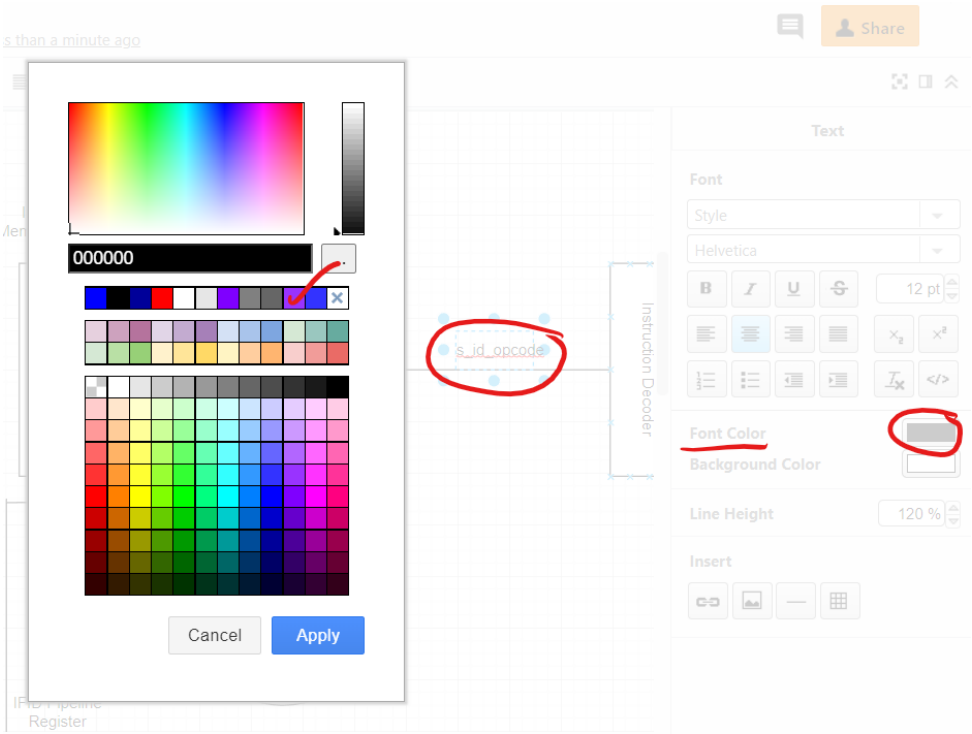
- To make the line into a right angle, click somewhere in the middle of the line and drag the middle point until you get the shape that you desire



- To add the signal name, click on “text” on the left hand side, drag the text box to the desired location, and then double click “text” while this box is selected to add the signal name, s_id_opcode



- To change the text to purple, while the text is highlighted, select the text color in the right toolbar



- From the opcode, the decoder can extract what type of instruction is being executed and will set the control signals for later stages of the pipeline.

- Now, we are going to add the control signals that are set according to the opcode.
- There are a total of 9 control signals that need to be added, so make sure there is enough room for all the control lines on the right side of the instruction decoder. These signals will be going from the Instruction Decoder to the IDEX pipeline register. Add the Control Lines and text label for the below signals:
    - s_id_regwrite - Let's the system know whether we will write to a register or not.
    - s_id_branch_inst - Tell's later stages whether or not the instruction is a branch instruction.
    - s_id_jump_inst - Tell's later stages whether or not the instruction is a jump instruction.
    - s_id_memops - Tell's later stages whether or not the instruction will be a memory operation.
    - s_id_memread - Tell's later stages whether or not the instruction will be reading from memory.
    - s_id_alusrc1 - Selects the alu source 1 input
    - s_id_alusrc2 - Selects the alu source 2 input
    - s_id_aluop - Selects the operation the alu will be performing on the two inputs
    - s_id_imm_gen_sel - Selects the immediate value for a given type of instruction

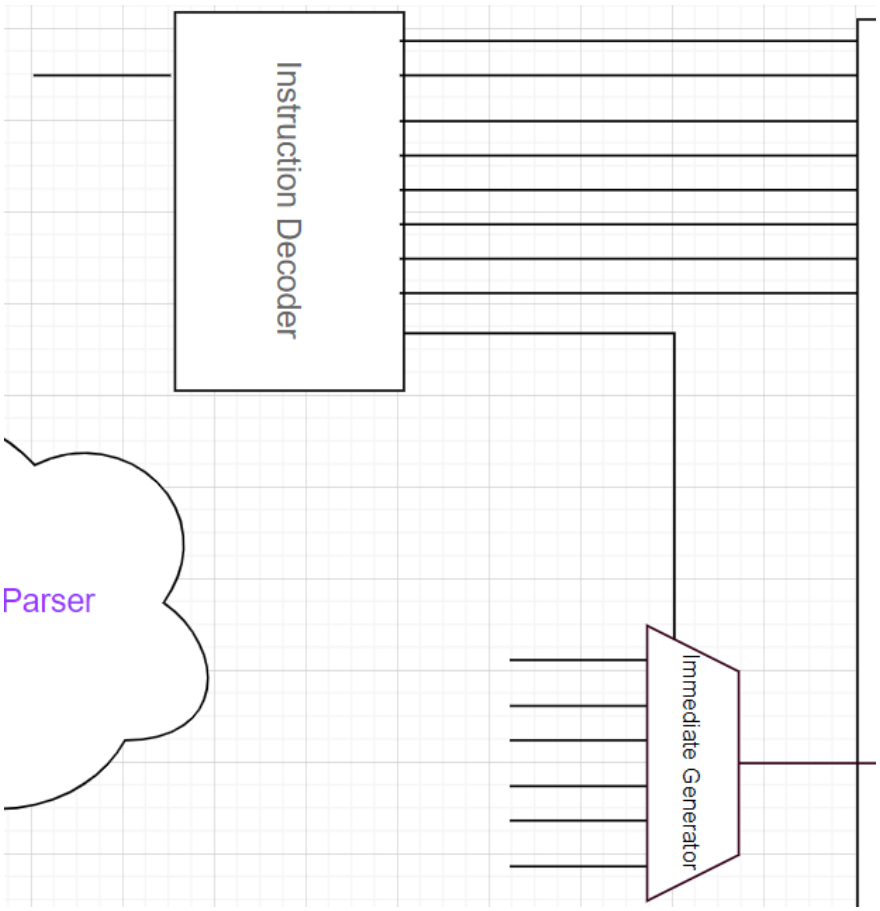
- After these control signals have been added, we are going to add the immediate generator. This is a 6 to 1 multiplexor that will take in a control signal from the instruction decoder and select the proper immediate value.

- The symbol used for a multiplexor is not a defined shape in draw.io. Because of that, the shape for the mux has been provided in the last tab of the project, Mux Component. Copy and paste that shape wherever you will need a mux.

- Add 6 lines going into the left side of the mux and one coming from the right. This will be a 6:1 mux.

- Add a label to the mux and call it Immediate Generator

- Shown below is what the immediate generator and instruction decoder should look like thus far (excluding signal names).

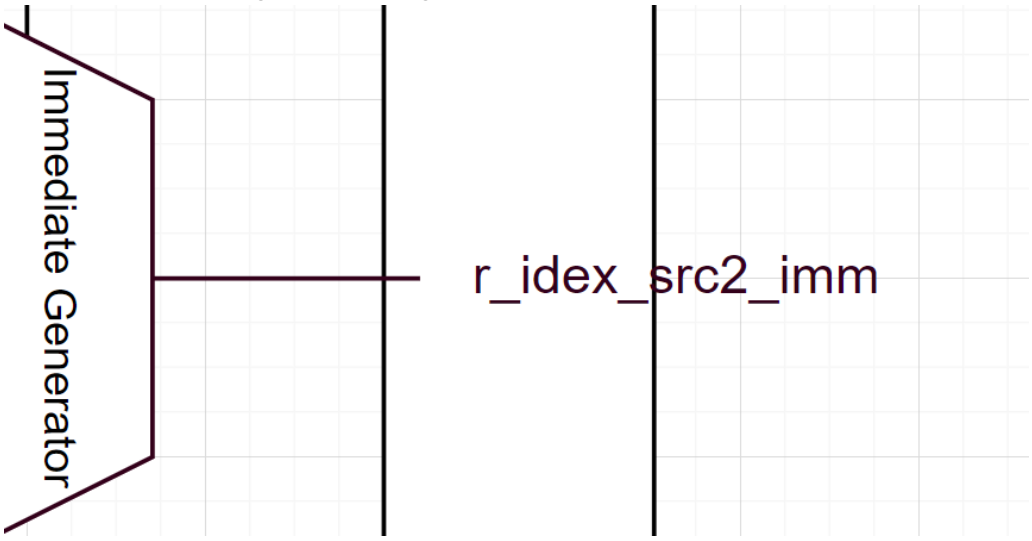


- Which signal from the instruction decoder should be used to select the immediate value? Make sure that signal is the one that is being used as the control signal for the Immediate Generator.
    - Add this control line and its text label from the Instruction Decoder to the Immediate Generator Mux

- Next, add the inputs to the Immediate generator. The inputs are,
    - s_id_imm_itype
    - s_id_imm_rtype
    - s_id_imm_stype
    - s_id_imm_btype
    - s_id_imm_utype
    - s_id_imm_jtype
- All of these signals should be purple since they come (are generated) from the instruction parser

- As for the Mux output, it is going to be sent through the IDEX pipeline. So, name the output r_idex_src2_imm as shown in the screenshot below indicating that it will be stored as a pipeline register

- Recall the naming convention for this register. It starts with r_ because it is a register and not a signal. Then it contains idex because the register output is from the IDEX pipeline register. Then finally, it is followed by src2_imm which tells us what this register is being used for.



- Moving on to the Register File, currently you should see a component currently with 2 inputs being r_memwb_data_result and r_memwb_regwrite. These register outputs are values for data forwarding used in phase 6 so don't worry too much about them being in the ID stage.

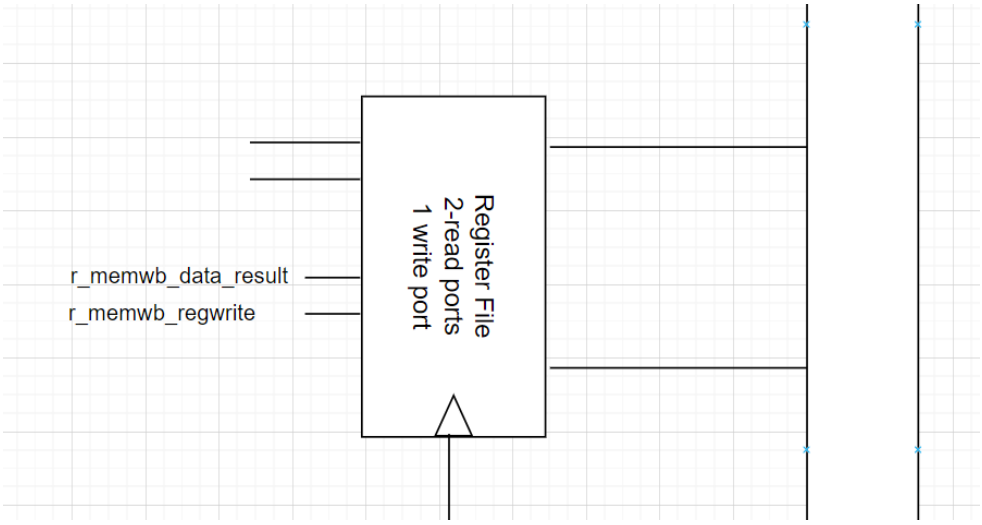
- **Register File**: A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed. (Computer Organization and Design, RISCV Edition by David Patterson and John Hennesey)
    - The register file will have multiple read ports, a minimum of one read port for each possible RISCV instruction input operand and one write port to store the result of an instruction. For your RISCV 5-stage pipeline register, there will be two read ports and one write port. The multiple ports enable two reads and one write to occur every clock cycle.
    - These registers are designed to be very fast to support the high frequency of the processor clock. To implement these very fast memories with multiple ports, they consume a relatively large area which makes them expensive to implement as well as high power/energy per bit. It would not be cost effective or possible to implement all the memory required by a processor as a register file.
    - To make large memories appear to be fast, you will be learning later in this course one of the eight great ideas of Computer Architecture: **Hierarchy of Memories**

- There are still additions that need to be made to the register file. So, add 2 more signals as inputs. These signals should be called s_id_src1 and s_id_src2. These are the register file values defined by the registers indexed by rs1 and rs2 values that have come from the instruction parser so they should be purple.

- Next, add 2 outputs from the register file, these should be called s_id_rf_src1 and s_id_rf_src2.

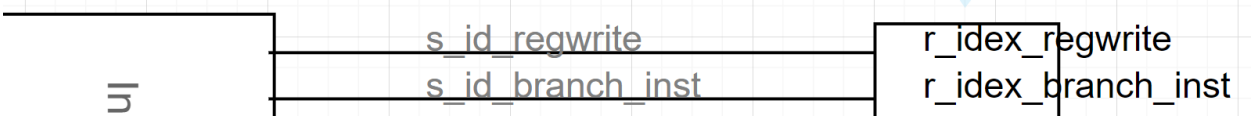
- s_id_src1 and s_id_src2 are the actual register locations, defined by the rs1 and rs2 bit locations in the instruction word, while s_id_rf_src1 and s_id_rf_src2 are the values stored within the register file. For example, let's say we have the instruction
  add x1, x2, x3
  where x2=20 and x3 = 30. In this case s_id_src1 = 2 and s_id_src2 = 3, while s_id_rf_src1 = 20 and s_id_rf_src2 = 30.

- With all the changes made, the register file should look like the screenshot below. Like before, the signal names have been omitted from the image.



- We are almost done with adding all the signals in the ID stage. To finish this section off, we need to add the names of the registers to be passed to the next stage.

- As an example, look at the screenshot below. Following the same naming convention shown below, add the rest of the register names for the signals that are being passed through the IDEX pipeline.
- Any signal that goes to the IDEX pipeline register should have a corresponding r_idex_ pipeline register name

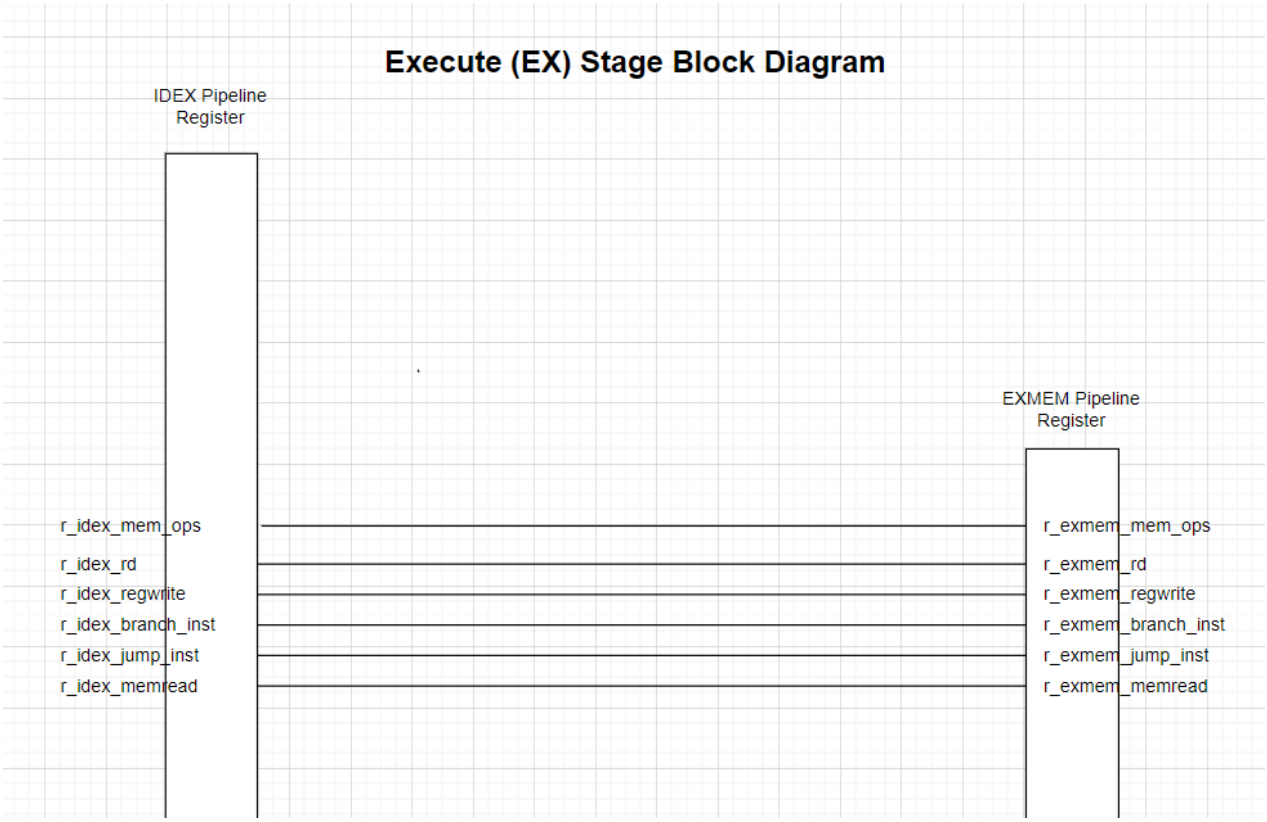


- Once all of the pipeline register names have been added, you have completed the ID stage for the block diagram.

EX Stage

- Switch the tab to the EX stage.

- As of now, within this stage, you should see the signals in the screenshot below being passed directly into the EXMEM pipeline.
- These are being sent to the next pipeline since the EX stage will not be performing any logic on them in this stage right now. They will still be used in later stages of the pipeline.



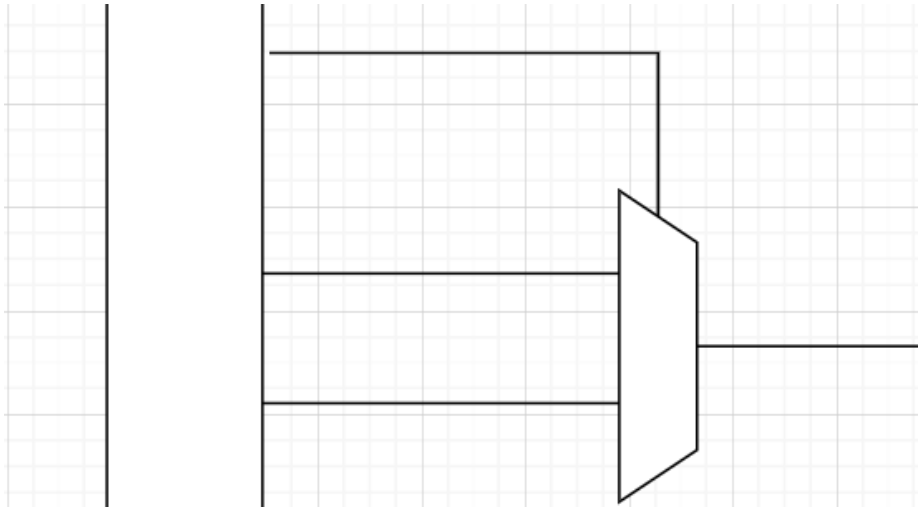
- To start this section, we want to add a 2:1 multiplexor for the source 2 operand of the ALU.

- The inputs for this 2:1 multiplexor are r_idex_rf_src2 and r_idex_src2_imm. The control signal for this multiplexor is r_idex_src2_sl.

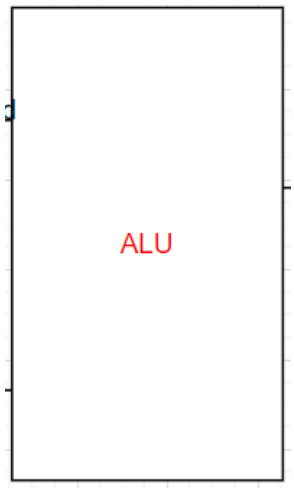
- This multiplexor will select the src2 according to r_idex_src2_sl. The select signal will be set by the decoder for the given instruction. For example, with an addi instruction we will be using an immediate value. However, with an add instruction we will instead be using the source 2 operand register value.

- The final thing to do with this multiplexor will be to add an output. The name of this output signal should be s_ex_src2_operand. This signal will be an input to the alu later, so simply leave it floating for now.

- Below is a screenshot of what the multiplexor should look like with the signal names omitted.



- The next step is to add another multiplexor above the src 2 multiplexor. This multiplexor will only have one control signal and one input signal. The input to this multiplexor is r_idex_rf_src1 and the control signal is r_idex_src1_sl.
- This multiplexor is added because there are other src1 signals that will be added in later phases. However, for now, there is only one input signal.
- Similar to the last multiplexor, add an output signal from the multiplexor and have the name of the signal be s_ex_src1_operand. Leave this signal floating for now as it will be used as the other input to the alu.
- Now that we have both operands added, it's time to add in the alu. Input a rectangle and give it the name ALU. A screenshot is provided below of what the component looks like.



- **ALU** (Arithmetic Logic Unit): The combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers. (Wikipedia)
    - The ALU performs all the logic functions defined by each assembly instruction

- With the ALU added, connect the two output signals of the source 1 and source 2 multiplexers to the left side of the ALU as its inputs.
- These are the two signals that the ALU will use as inputs on operations such as adding, shifting, etc.
- Add an output signal from the ALU to the EXMEM pipeline register and name that signal s_ex_alu_result
- This signal is the result of the operation. There are a few ways this output will be used depending on the instruction. The result could be written to the register file, it could be used to load/store values to memory, or it could be used in a branch/jump operation.
- To finish off the EX stage, we are going to pass the alu result to the next stage. To do this, add the pipeline register name of the signal on the right side of the EXMEM pipeline register. The name should be r_exmem_alu_result.

MEM Stage

- For this phase there is nothing that happens in the MEM stage. However, we still need to pass 3 signals to WB stage to be written into the register file

- Switch the tab to MEM stage and notice how all the signals that have been passed using the EXMEM pipeline are on the left side of the diagram.

- There are 3 signals that need to be passed to the WB stage. Those signals are r_exmem_alu_result, r_exmem_rd, and r_exmem_regwrite.

- At this stage of the project, the other signal names in the EXMEM pipeline register require no action

- Pass those given signals to the MEMWB pipeline and name the signals accordingly for the next stage.
    - r_exmem_alu_result will be called r_memwb_data_result in the memwb pipeline
    - r_exmem_rd will be called r_memwb_rd
    - r_exmem_regwrite will be called r_memwb_regwrite

- That is all that needs to be done here, so it's time to move on to the WB stage

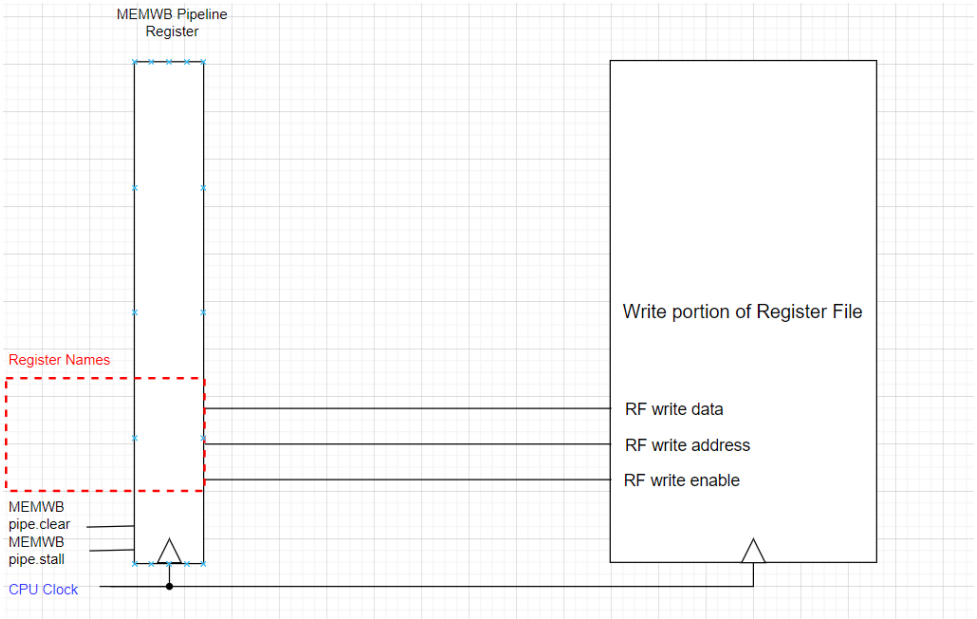
---

WB Stage

- Switch the tab to the WB stage

- Notice how the signal names of the pipeline are just floating in the middle. This is done intentionally because we will want to pair the given signals with the inputs to the register file.

- There are 3 inputs to the register file
    - RF write data - This is the data that we will be writing to the register file.
    - RF Write Address - This is the destination address we are writing to.
    - RF Write Enable - This is the signal that tells us if we are actually planning to write to the register file.
- Move the register names back to the left side across from their given input to the register file. The screenshot below shows what the WB stage should look like with the signal names omitted.
    - Use the proper pipeline register name using the standard set by this class
    - What MEMWB pipeline registers from the WB Stage will be used for each of these sigals?



- With the registers going to the proper inputs of the register file, you have successfully completed phase 4.

- With the completed phase
    - Save and Export your phase as an .xml file
    - Submit your exported phase for grading