Out of order Execution Divider

Objective:

- Understand how an IoD-**OoE**_IoC CPU manages to put the instruction in program order using a ROB by implementing an OoE (Out of order Execution) divider
- Understand the benefit of using OoE compared with IoE in using resources such as multiple dividers here.

Description:

Problem Statement: Implement a design that uses four single dividers in parallel. The order in which the output is committed should be the same as the order in which the input is issued.

The problem has two parts:

1. In part 1, the input is dispatched "In Order", the execution takes place "In Order" and the output is committed "In Order" (IoD-**IoE**-IoC).
2. In part 2, the input is dispatched "In Order", the execution takes place "Out of Order", however the output is committed "In Order" (IoD-**OoE**-IoC). ROB-tag technique is used to work out such a design.

Both designs consist of two memories to hold (store) the input and the output data. The memories are implemented as arrays of registers. The input data is dividend-divisor pairs which are initially stored in a memory called "mem_dividend_divisor" by the testbench. These pairs are dispatched to a free single divider in order (one dividend-divisor pair per clock). The single divider produces quotient-remainder pair as output. In case of part 1, the output is stored into the output data memory called "mem_quotient_remainder" directly. In part2, the output is stored into the ROB and once the top-most ROB entry becomes ready, the quotient-remainder pair of the corresponding ROB entry is stored into the output data memory called "mem_quotient_remainder".

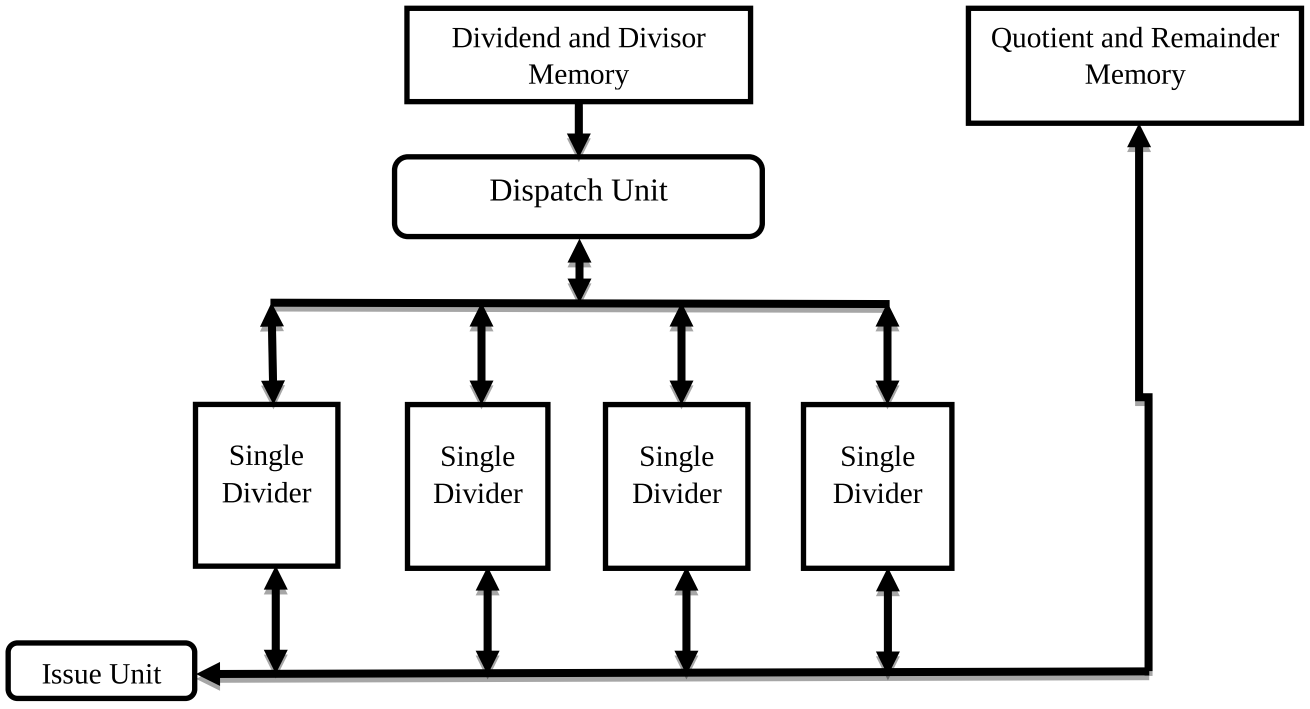
Part 1

In this part we implement an "IoD-**IoE**-IoC" divider. This design is used as the baseline design

to understand the benefits of using an "Out-of-Order" execution divider. We instantiate 4 single dividers to work together. The important components and their responsibilities are listed below:

Dispatch Unit: It reads a dividend-divisor pair from the input memory if the next single divider to be assigned is available (and of course all entries of the input memory are not yet read). The next single divider is selected in round robin manner: So if divider i was selected, the next divider will be $(i + 1)\ mod\ n$ where n is the number of dividers. The dividend-divisor pair is dispatched to the single divider and a start signal is sent to it.

Issue Unit: It collects quotient-remainder pair from the next single divider (next to be collected from) and stores it into the output memory called "mem_quotient_remainder". Similar to the dispatch unit, if divider output from divider j was collected, the next divider will be $(j + 1)\ mod\ n$ where n is the number of dividers. (**Note:** The next single dividers for dispatch unit and issue units are different. In the Verilog code, assign_pointer is used to identify the next single divider for the dispatch unit and collect_pointer is used to identify the next single divider for the issue unit). In a single clock a maximum of one quotient-remainder pair can be collected.



Part 2

In this part we implement an "IoD-OoE-IoC" divider . ROB-tag technique is used to work out such a design. Here also we instantiate 4 single dividers to work together. The important components and their responsibilities are listed below:

Dispatch Unit: It reads a dividend-divisor pair from mem_dividend_divisor in order if there is at least one single divider available, ROB is not full (and of course all entries of the input memory are not yet read). *The divider being assigned can be any of the four dividers and there is no order required. This is the important difference between the two parts and Part 2 utilizes the divider resource much better.* If multiple single dividers are available, the one with the lowest index is given the highest priority for being assigned. A ROB entry is allocated and the dividend-divisor pair and the tag of the ROB entry (the **WP** value) is dispatched to the single divider and a start signal is sent to it.

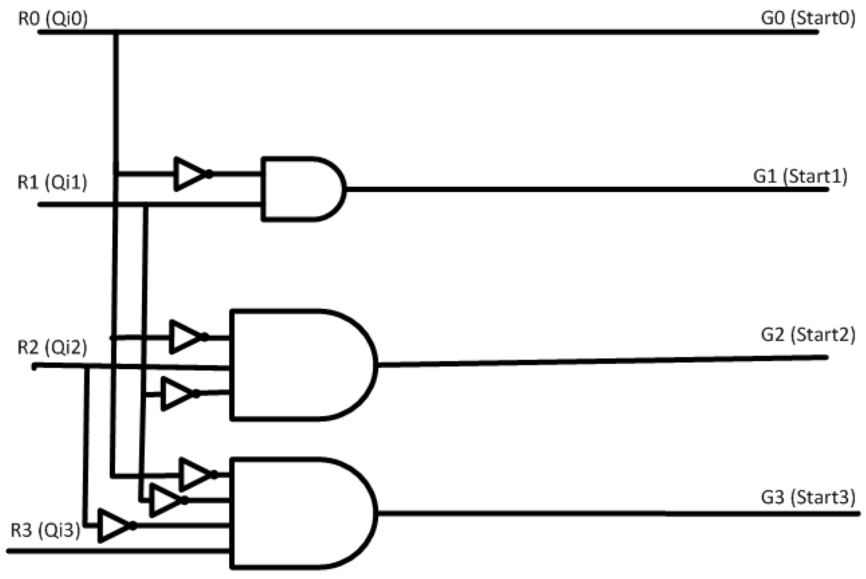
Issue Unit: If any single divider has completed its execution, the issue unit collects the quotient-remainder pair and stores it into the ROB at an entry *as per the ROB-tag assigned to* the divider. If multiple single dividers have finished their assigned division tasks, a lower indexed single divider is given higher priority and its quotient-remainder pair is collected. In a single clock a maximum of one quotient-remainder pair can be collected.

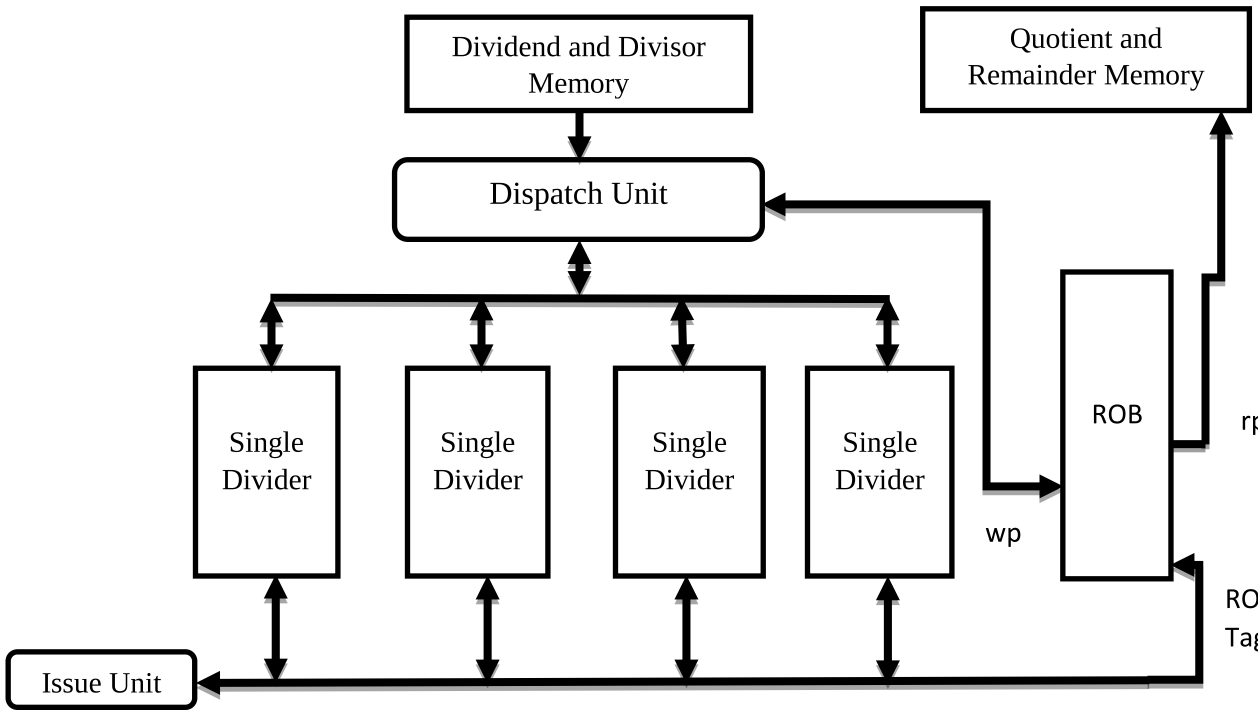
ROB: This is a crucial part to make out of order execution (**OoE**) to work. ROB is an 8-entry FIFO here. Graduation of tasks follows the same order with dispatch of the tasks, even though completion of tasks by single dividers could be out of order.

- When dispatching, ROB entry pointed by the WP (the write pointer) is allocated and write pointer is incremented.
- The top-most entry (the senior-most entry) is pointed to by the RP (the read pointer). When the top-most entry is ready, it (the the remainder and quotient pair) is read from the ROB and is written to the memory "mem_quotient_remainder". The RP is incremented.

- Although ROB is a FIFO, the issue unit can write into a location pointed by the ROB tag of the completed single divider (which can be any entry in the populated portion of the FIFO).
- When the ROB is full and no more free entries are available in the ROB, dispatch unit stalls dispatching even if dividers are available.

Prioritized Request-Grant Circuit (Priority Arbiter): A priority request-grant circuit, also known as priority arbiter allocates shared resources to the requesters in priority order. In this lab, the mem_dividend_divisor array and the ROB can be treated as shared resources. The single dividers request access to these resources. If a single divider is in Qi (initial) state, it is ready to accept a dividend-divisor pair and hence an implicit request is assumed. Similarly, if a single divider is in Qd (done) state, the quotient-remainder pair is ready to be collected and stored into ROB. Among the multiple single dividers ready, we give higher priority to the single divider with lower index. One possible gate-level implementation for the priority arbiter is as shown below:





Procedure:

Part 1:

1. Understand the design, complete the missing parts in `IoE_Divider.v`

2. Start Modelsim, build a new project, and name it.

3. Add files to the directory of the project you built. The project should include all the following files:

```
A) IoE_Divider.v (to be completed)
B) IoE_Divider_tb.v
C) Single_div.v
D) IoE_Divider_wave.do
E) IoE_Divider.do
```

4. Add `Single_Div.v, IoE_Divider_tb.v, IoE_Divider.v` files to the project and compile them.

5. Type "do `IoE_Divider.do`" in the Transcript panel to run it.

6. Compare the generated output file with golden results. If they match, then congrats!

7. If not, then use waveforms and output file to debug your design. It feels great to find a flaw and fix it.

8. Submit your completed design "`IoE_Divider.v`".

```
submit -user ee457lab -tag puvvada_ROB_P1 IoE_Divider.v names.txt
```

Part 2:

1. Understand the design, complete the missing parts in `OoE_Divider.v`

2. Start Modelsim, build a new project and name it.

3. Add files to the directory of the project you built. The project should include all the following files:

```
A) OoE_Divider.v (to be completed)
B) OoE_Divider_tb.v
C) Single_div.v
D) OoE_Divider_wave.do
E) OoE_Divider.do
```

4. Add `Single_Div.v, OoE_Divider_tb.v, OoE_Divider.v` files to the project and compile them.

5. Type "do `OoE_Divider.do`" in the Transcript panel to run it.

6. Compare the generated output file with golden results. If they match, then congrats!

7. If not, then use waveforms and output file to debug your design. It feels great to find a flaw and fix it.

8. Submit your completed design "`OoE_Divider.v`".

```
submit -user ee457lab -tag puvvada_ROB_P2 OoE_Divider.v names.txt
```

Assignment:

1. Note that we used two always @ blocks in the file `OoE_Divider.v`. Can we combine these two blocks into one clock-edge-triggered block? If yes, explain how to implement. If no, explain why.

2. Note that we instantiated 4 single dividers in the OoE divider. Let's take the signals "`Ack`" (`ack_from_OoE_div`) and "`Start`" (`start_from_OoE_div`) of single dividers as examples. Explain how we ensured that these two signals are active for exactly one clock when they should be (they are both set in an always @ block implementing a **combinational** logic in `OoE_Divider.v`).

3. In this lab, we used 4-bit ROB read/write pointers (RP and WP) for the 8-entry ROB, yet we only used the lower 3-bits to index the ROB. Do you think ignoring the MSB would cause any error in some case, explain why. Where did we use the MSB of WP and RP?