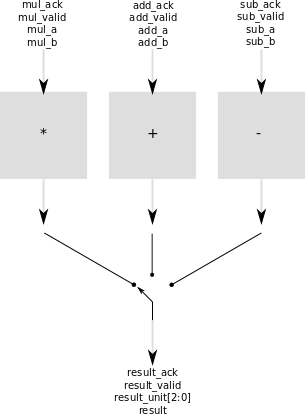
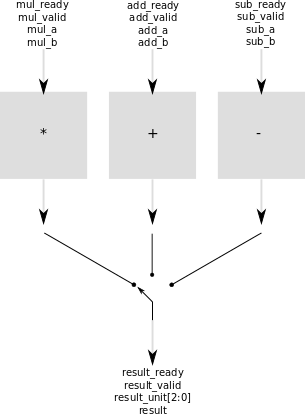
# Swarm64 Hardware Design Exercise

This is the Swarm64 exercise for hardware designers. It consists of two parts. The first part is to design and verify a digital module. The later part is to prototype the design with Altera Quartus II, optimize for performance and meet timing requirements. The target is a Cyclone V FPGA.

A set of embryo files for the Verilog design, the C++ verification code and the Quartus project are provided in the archive accompanying this exercise. The file *quartus\_howto.txt* which is also included contains instructions on how to install and configure Quartus, while *simulation\_howto.txt* describes how to build and run a simulation and examine the waveform.

## Module Description

The task is to implement a calculator capable for three mathematical operations: addition, subtraction and multiplication. Each arithmetic operation is conducted in a different arithmetic unit. An independent input interface is used for each arithmetic unit. The interface consists of a *valid/ack* handshake pair and two arithmetic operands. The arithmetic units should accept an input whenever the *valid* signal is high and they are not blocked due to congestion further down in the pipeline. An arbiter is responsible to fairly arbitrate between the output of the arithmetic units and propagate them to a common output interface. In addition to the *valid/ack* and *data* outputs, an additional output *result\_unit* is showing which arithmetic unit the result came from.

**Valid/ack interface**

This interface connects a sender (producing a datum) and a receiver (consuming a datum) in such a way that the sender and the receiver both have the ability to regulate the speed. The *valid* signal go from the sender to the receiver, while the *ack* signal goes from the receiver to the sender. Any number of additional signals can be present in order to transfer data from the sender to the receiver. *Valid* high signifies that there is *valid* data on these ports, while *ack* signifies that the receiver has accepted the data. Thus, such transfer occurs if and only if both *ack* and *valid* are asserted at a rising clock edge. The *ack* signal is combinatorial and based in part on the *valid* signal – *ack* should never be high unless *valid* is also high.

### Requirements

1. Arithmetic units should accept calculation commands as often as possible
2. Arithmetic units use have a *valid/ack* input interface. An operand set is accepted with the assertion of the *valid*(sender) and *ack*(receiver) signals within the same cycle. Any other state of the *valid* and *ack* signals should not trigger calculations or cause output changes.
3. All results should be propagated to the test-bench through a streaming interface. The test-bench should have the ability to stall the output by holding *ack* de-asserted. An output is accepted only when the *valid* and *ack* signals are asserted within the same cycle. In any other case the calculator should maintain the *valid* signal asserted and the *result* value set, until the test-bench acknowledges the acceptance of an output
4. The output of the module is **fairly** selected among the valid outputs of the three arithmetic units.
5. Under-utilizing any of the arithmetic units should not cause idle cycles for the output interface. The only case when the output *valid* signal can be de-asserted is when no arithmetic unit has completed a calculation
6. All calculations should be mathematically correct
7. The one-hot encoded output signal *result\_unit* is showing which arithmetic unit is selected for output when the *valid* signal is asserted

## Design Verification

A randomized test written in C or C++ should be provided with the above design. It should fully randomize the inputs in order to ensure that the module is driven into all the relevant states, while checking that module behaves correctly. The test code must be easily understandable, and robust (not error-prone). You may create more than one test with different types of randomized stimuli if you feel that this is required, but in our experience, one or two tests, and 100-200 lines of code (semicolons) is all it takes. Consider the requirements related to correct mathematical results, fairness and throughput, and make sure your checking logic will catch all bugs. As part of the code review, we might try to insert bugs into the Verilog code to see that the test detects them.

## Design Prototyping

When the design is complete and verified, it has to be synthesized and fitted for an Altera Cyclone V FPGA. You should only be using the synthesizable subset of the Verilog or SystemVerilog language for that reason. Once this is complete, observe the maximum operation frequency in the TimeQuest Timing Analyzer (Fmax summary).

## Design Optimization

The goal for this last part of the exercise is to beat the maximum frequency Quartus achieves when synthesizing Verilog behavioral code ( a <= c + b) .

Quartus recognizes operands such as \* and +, and synthesizes them into combinatorial logic in the most efficient way for your FPGA. However, for wide operands, the critical path may become too long. The best way to overcome this, is to combine the result of multiple narrower calculations in a pipeline.

Identify the critical path in the design. Optimize the design to reach a maximum operating frequency of 150Mhz.

To understand what pipeline is, and how to improve timing in designs you might consult the following lecture handouts:

Lecture 1. Implementing Algorithms I [(2/page pdf)](http://www.imit.kth.se/courses/IL2204/F1_2.pdf) [(6/page pdf)](http://www.imit.kth.se/courses/IL2204/F1_6.pdf)

Lecture 2. Implementing Algorithms II [(2/page pdf)](http://www.imit.kth.se/courses/IL2204/F2_2.pdf) [(6/page pdf)](http://www.imit.kth.se/courses/IL2204/F2_6.pdf)

Lecture 3. Implementing Algorithms III [(2/page pdf)](http://www.imit.kth.se/courses/IL2204/F3_2.pdf) [(6/page pdf)](http://www.imit.kth.se/courses/IL2204/F3_6.pdf)

Please note that these lecture handouts are not FPGA specific and do not take into account that FPGA devices might have dedicated hardware for efficient multiplication.

*GOOD LUCK!*