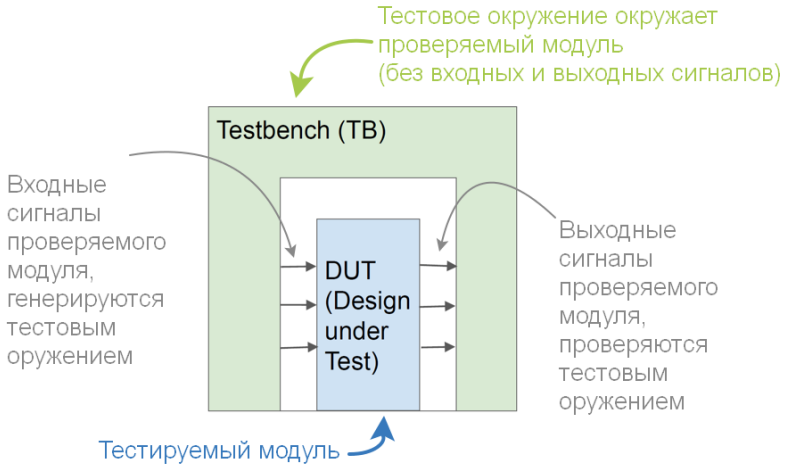
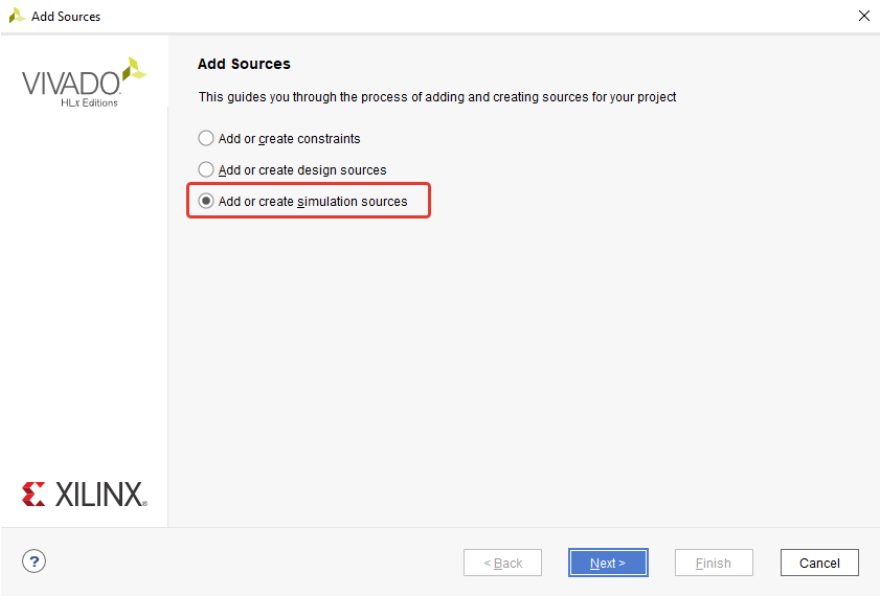
Тестовое окружение (testbench)

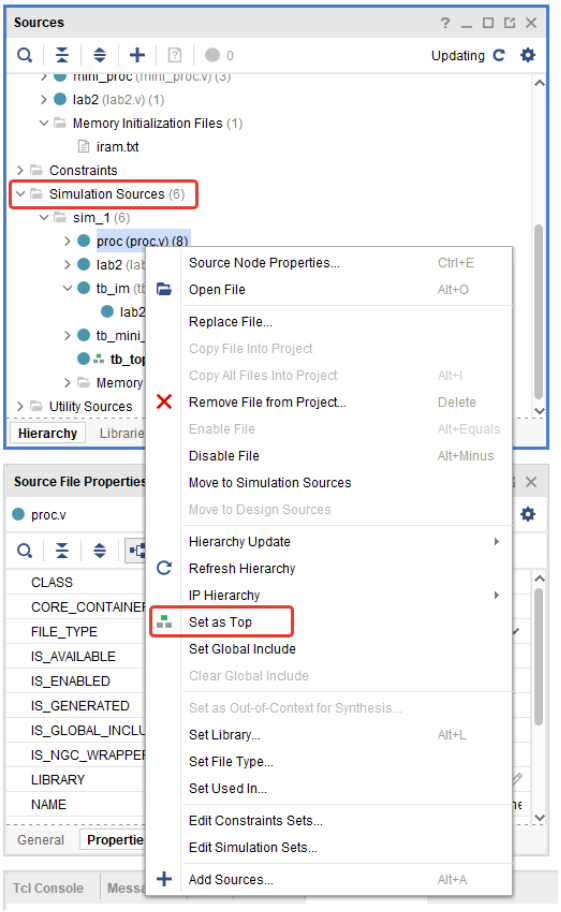
Для проверки правильного функционирования цифровых устройств необходимо разработать тестовое окружение. Тестовое окружение (**testbench**) – это блок, который окружает проверяемое устройство, формирует для него тестовые сигналы и автоматически проверяет, что сигналы на выходе проверяемого устройства соответствуют заложенным функциям. Тестовое окружение не является реальным аппаратным блоком (он только симулируется), поэтому в нем возможно использовать традиционные конструкции программирования. Например, то, что описывается в блоках **initial** в тестовом окружении, выполняется как программа в классическом программировании – строчка за строчкой.



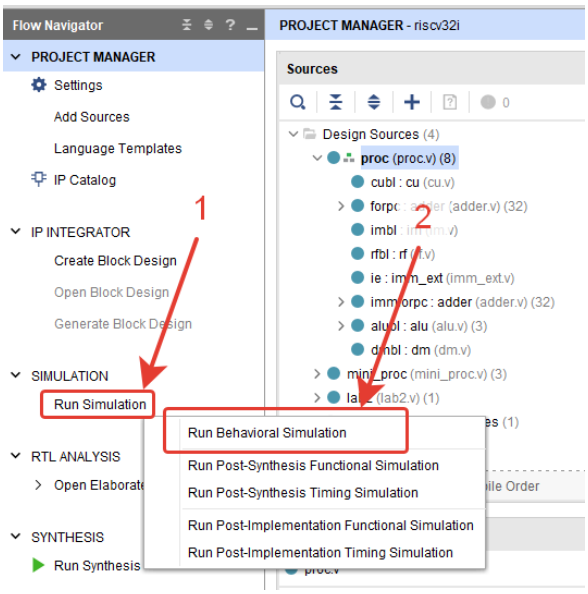
Для того, чтобы создать тестовое окружение (**testbench**) необходимо создать новый файл симуляции в проекте. Для этого нажмите на **Add source**, после чего нужно выбрать **Add or create simulation sources** (на картинке ниже) → **Create File...** и так далее.



Так как для проверки разных модулей придется создавать различные тестовые окружения, то **Vivado** придется сообщать в явном виде какой из файлов симуляции вы сейчас хотите запустить. Для этого надо щелкнуть на нужном файле правой кнопкой и выбрать пункт **Set as Top** (продемонстрировано на картинке далее).



Название файла, для которого будет запускаться симуляция, отображается жирным шрифтом в окне **Sources** в папке **Simulation Sources**. После того, как выбран нужный файл симуляции, его можно запустить через панель **PROJECT MANAGER**, нажав на **Run Simulation**, а затем, в самом простом случае, можно запустить поведенческое моделирование (**Run Behavioral Simulation**), оно позволяет увидеть поведение устройства в ответ на воздействия testbench'а. Временные задержки на прохождение сигналов через цифровые блоки при этом не учитываются. Так же можно посмотреть на реакцию устройства после синтеза (**Post-Synthesis**) или после имплементации (**Post-Implementation**). Функциональная симуляция не учитывает временные задержки, временная – учитывает. Наиболее приближенная к реальности симуляция **Post-Implementation Timing Simulation**. Она учитывает временные задержки конкретных компонентов, в конкретной ПЛИС, с конкретными задержками распространения сигнала по каждому из проводов.



Пример тестового окружение для сумматора

Ниже приводится пример тестового окружения, в котором:
❶ – создаются провода и регистры для подключения к тестируемому модулю,
❷ – подключается проверяемый модуль,
❸ – описывается задача (**task**), которую, подобно функции или подпрограмме, можно вызывать с различными параметрами,
❹ – в блоке **initial** последовательно два раза вызывается задача **add_op**, после чего симуляция останавливается **$stop**,
❺ – в данном тестовом окружении ненужная вещь и не используется (можно удалять), нужен только для демонстрации, так как это понадобится при проверках памяти, когда придется генерировать сигнал тактирования.

[не забудь убрать цифры ❶ – ❺, когда будешь копировать код]

```
`timescale 1ns / 1ps    // Первое число указывает в каких величинах задержка
                        // например, использование #10 это 10 наносекунд
                        // если бы параметр был 10ns, то #10 означало бы 100ns
                        // Второе число указывает точность симуляции
                        // тут симуляция происходит для каждой пикосекунды

module my_testbench (); // объявляем модуль тестового окружения
                        // внешних сигналов нет, поэтому скобки пустые
❶ reg [31:0] A, B;  // объявляем регистры для управления входами сумматора
  wire [31:0] S;    // объявляем провод для подключения к выходу суммы
  reg Cin;          // объявляем регистр для управления входом Cin
  wire Cout;        // объявляем провод для подключения к выходу Cout

❷ adder dut (A, B, Cin, S, Cout);      // подключаем тестируемый модуль
            // такая запись без точек допускается при полном соответствии имен
            // если внутренние имена отличаются от внешних, то надо
            // писать через точку, например, .a(A), .b(B), .res(S) и тому подобное
            // dut (device under test) - классическое название тестируемого модуля,
            // при желании можно использовать любое другое имя
  initial begin   // блок последовательного исполнения, начинает работу с момента времени 0
❹    add_op(6, 3); // запустить задачу task add_op с параметрами 6 и 3
     add_op(2, 7); // когда закончиться предыдущая задача запустить новую
     $stop;        // остановить симуляцию
  end

❸ task add_op;                          // объявляем задачу add_op
   input [31:0] a_op, b_op;             // task получает на вход два параметра
   begin
     A = a_op;                          // подать на вход A сумматора новое значение a_op
     B = b_op;                          // подать на вход B сумматора новое значение b_op
     Cin = 0;                           // подать на вход Cin ноль
     #100;   // выждать 100 ns чтобы сигнальчики разбежались и сумматор успел посчитать
     if (S == (a_op + b_op))    // если реальность (S) и ожидание (a_op+b_op) совпадают, то
        $display("GOOD %d + %d = %d", A, B, S);     // вывести в терминал сообщение good
     else                                           // в противном случае
        $display("BAD %d + %d = %d", A, B, S);      // вывести в терминал другое сообщение
   end
  endtask

  endmodule
```

```
❺ reg clk;                // это вообще не относится к описанному выше testbench'у
  always #10 clk = ~clk;  // каждые 10ns менять clk на противоположное значение
```

Блоков **initial** в тестовом окружении может быть сколько угодно. Все эти блоки запускаются на исполнение параллельно. Блоков **task** так же может быть сколько угодно много и каждый из них может выполнять разные проверки. Так же поддерживаются множество стандартных языковых конструкции, например цикл **for**. Параметры для **$display** передаются так же, как у **printf** в языке C (на википедии есть вся информация).

Данный пример проверяет две суммы (6+3) и (2+7). Тест необходимо дополнить большим количеством проверок: несколько с очень большими числами, несколько с отрицательными, несколько с отрицательными и положительными, несколько со входным переносом **Cin = 1**, несколько операций с числами вызывающим переполнение (чтобы проверить формирование **Cout**).

По аналогии с этим примером необходимо реализовать модули проверки для:

- **АЛУ** (по несколько проверок на каждую операцию)
- **Регистрового файла** (последовательно записать какие-нибудь данные в разные адреса, а потом считать, убедившись, что все правильно, при этом считывание по адресу 0 должно всегда показывать 0)
- **Памяти инструкций** (считать содержимое из нескольких ячеек, чтобы убедиться, что память проинициализирована)