В этой главе будем экспериментировать с одним из HDL[1] - Verilog. Почему именно он, а не VHDL? По сугубо субъективным причинам - Verilog более лаконичен и похож на C, что импонирует автору.

Цель экспериментов - пощупать руками что это такое, как с ним практически работать и на что примерно он способен. На очень простых примерах, имея в перспективе главную цель - проектирование и реализация процессора с собственной архитектурой.

Экспериментировать будем с заслуженной FPGA[2] от Altera - Cyclone-IV. Основная причина почему именно с ней - она оказалась под рукой и содержит достаточно ресурсов для наших целей. А именно:

- 6272 логических элемента
- 270 кбит встроенной SRAM памяти (триггеры)
- два блока PLL (ФАПЧ), синтезатор частоты
- 15 умножителей 18x18 или 30 штук 9x9

Для тестов используем отладочную плату Марсоход2bis потому, что она оказалась под рукой, при желании переход на любую другую аналогичную плату не составит труда. Помимо ПЛИС наша плата содержит

- Кварцевый генератор на 100 МГц
- Программатор FTDI FT2232H для связи с компьютером через USB
- 8 Мб микросхему SDRAM
- пользовательскую кнопку
- три пользовательских светодиода
- кое что еще, что нам не потребуется

В качестве средства разработки будем использовать свободную версию Quartus II 13.1 Web Edition от Altera (можно и новее, не принципиально).
А содержимое тестов так основано в том числе на тестах платы Марсоход2bis.

Исходные тексты всех тестов можно найти здесь.

### 5.1.1. Помигаем светодиодами.

Раз у нас есть кнопка и светодиоды, первой задачей выберем демонстрацию работы кнопки - пусть один из светодиодов горит когда кнопка нажата, а второй наоборот, когда не нажата.

- создаём (New Project Wizard) новый проект, назовём его test_button,
- выбираем в качестве рабочего устройства (там же) Cyclone IV E, EP4CE6E22C8L (маркировка микросхемы)
- добавляем в проект новый файл Verilog: test_button.v

[1] Hardware Definition Language
[2] Field-Programmable Gate Array

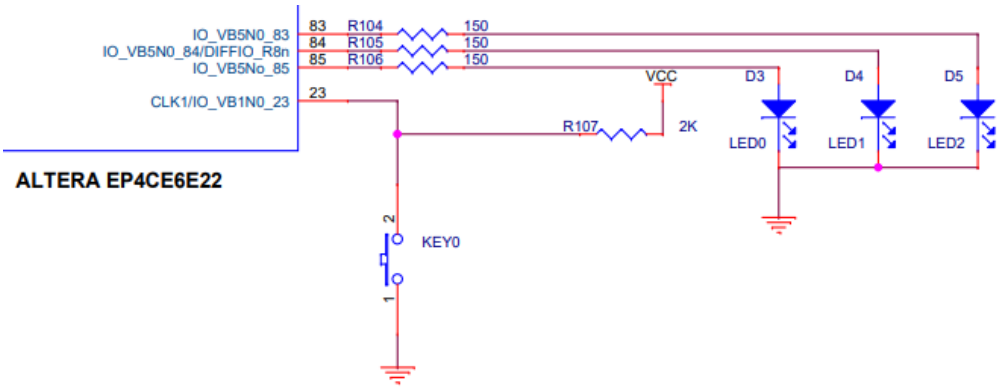
- в этом файле определяем логику

```
1   module test_button (
2         input wire key0,
3         output wire led0,
4         output wire led1
5         );
6
7   assign led0 = key0;
8   assign led1 = !key0;
9
10  endmodule
```

Фиг.5.1.1 содержимое test_button.v

Модуль имеет один вход - кнопку и два выхода - светодиода,
один из которых копирует состояние кнопки, второй инвертирует его.

- Теперь надо привязать входы и выходы модуля к реальным устройствам. Для этого придётся взглянуть на схему платы.




Фиг.5.1.2 фрагмент схемы отладочной платы (лист 2)

На втором листе видим, что вход с кнопки соответствует “ноге” №23,
светодиоды - 83..85

Обратим внимание, не нажатой кнопке соответствует высокий уровень напряжения, т.е. логическая единица.

- Теперь идём в (меню) Assignments / Assignments Editor и записываем

| | Status | From | To | Assignment Name | Value | Enabled | Entity |
|---|---|---|---|---|---|---|---|
| 1 | Ok | | key0 | Location | PIN_23 | Yes | |
| 2 | Ok | | led0 | Location | PIN_83 | Yes | |
| 3 | Ok | | led1 | Location | PIN_85 | Yes | |
| 4 | | <<new>> | <<new>> | <<new>> | | | |

Фиг.5.1.3 привязка сигналов средствами Quartus

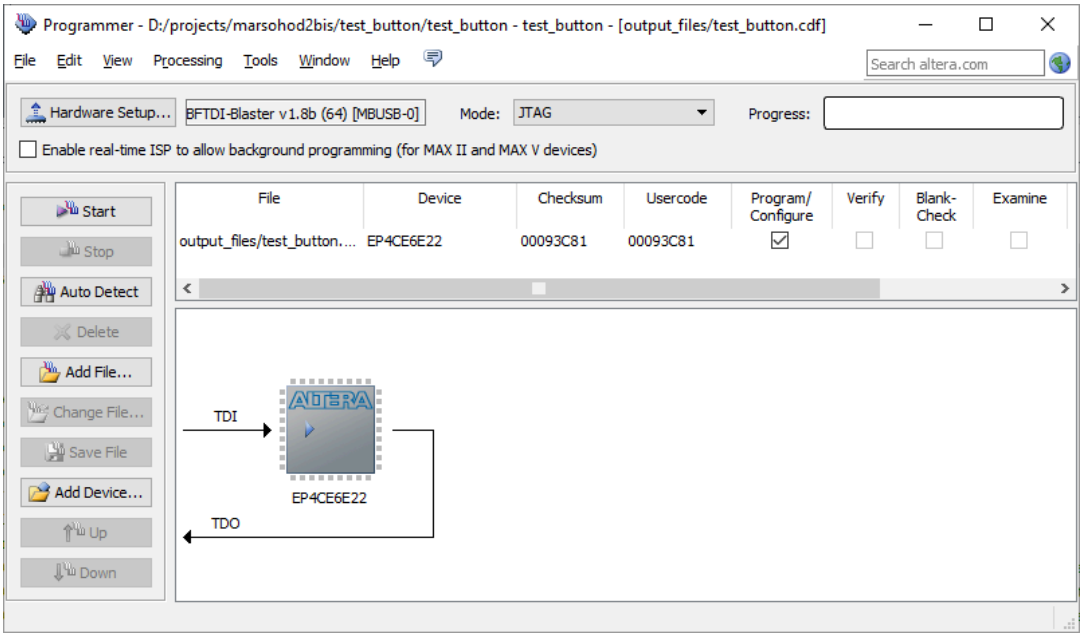
В колонке “To” имена вводов/выводов модуля.
PIN_xx - предопределенные имена “ног”.

Обратим внимание, led0 в модуле соответствует LED2 на схеме,

led1 - LED0 схемы (платы).

- Компилируем проект, убеждаемся что не возникло критических ошибок, предупреждения вида “No clocks defined in design.” не в счет, действительно, тактовый сигнал не использован.

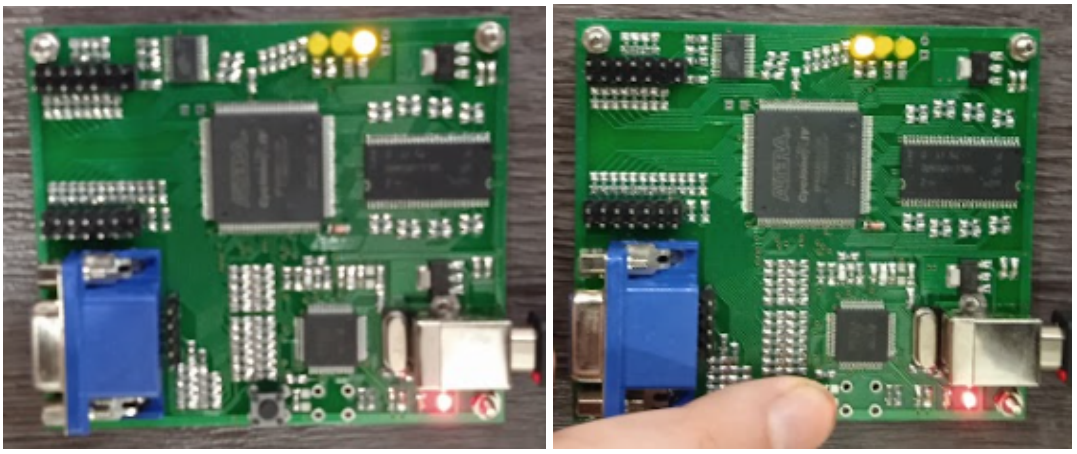
- Меню Tools / Programmer




Фиг.5.1.4 загрузка проекта в ПЛИС средствами Quartus

Раз всё определилось автоматически, нажимаем Start, в противном случае после AutoDetect придётся вписать руками скомпилированный файл “test_button/output_files/test_button.sof”

- Работает!



Фиг.5.1.5 демонстрация работы

### 5.1.2. Таймер / счетчик.

Здесь задача ненамного сложнее - заставим мигать один из светодиодов с некоторой периодичностью. На плате есть тактовый генератор с частотой 100 МГц, придётся поделить его частоту, чтобы мигание было уловимо глазом, иначе он просто не будет успевать гаснуть.

Для деления частоты воспользуемся таким трюком - заведём целочисленный счетчик, значение которого увеличивается на единицу каждый раз, когда приходит фронт тактового сигнала. Каждый N-й разряд счетчика (если считать с 0) соответствует делению частоты на $2^N$. При частоте 100 МГц выберем 27 разряд ($2^{20}$ - это грубо миллион и $2^7$ - 128) для того, чтобы светодиод мигал чуть реже, чем раз в секунду.

Итак,

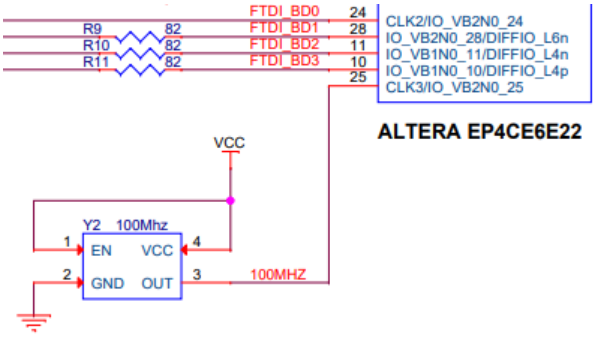
- создаём новый проект: test_timer
- добавляем новый файл, test_timer.v

```
1   module test_timer (
2       input wire  CLK100MHZ,
3       input wire  KEY0,
4       output wire [2:0]LED
5   );
6
7   reg [31:0]counter;
8   always @(posedge CLK100MHZ)
9       if( KEY0 )
10          counter <= counter+1;
11
12  assign LED[0] = counter[27];
13
14  endmodule
```

Фиг.5.1.6 содержимое test_timer.v

добавился новый вход - CLK100MHZ
а также внутренняя переменная *counter* шириной в 32 разряда
always @(posedge CLK100MHZ) означает, что следующее утверждение срабатывает когда выполняется определённое условие - приходит позитивный фронт сигнала CLK100MHZ и не нажата пользовательская кнопка

- необходимо привязать CLK100MHZ, для этого на схеме платы находим на какую “ногу” ПЛИС приходит сигнал тактового генератора




Фиг.5.1.7 фрагмент схемы платы (лист 5)

Искомый номер - 25

- Делаем привязку сигнала (Assignments / Assignments Editor)

| | Status | From | To | Assignment Name | Value | Enabled | Entity |
|---|---|---|---|---|---|---|---|
| 1 | ✓ ... | | in KEY0 | Location | PIN_23 | Yes | |
| 2 | ✓ ... | | in CLK100MHZ | Location | PIN_25 | Yes | |
| 3 | ✓ ... | | out LED[2] | Location | PIN_83 | Yes | |
| 4 | ✓ ... | | out LED[1] | Location | PIN_84 | Yes | |
| 5 | ✓ ... | | out LED[0] | Location | PIN_85 | Yes | |
| 6 | | <<new>> | <<new>> | <<new>> | | | |

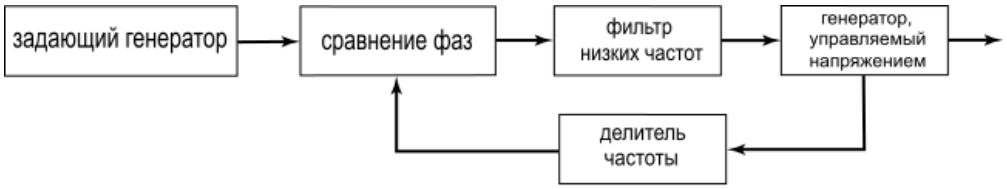
Фиг.5.1.8 привязка сигнала средствами Quartus

- Компилируем, запускаем, … работает!

### 5.1.3. Таймер / PLL[3].

Усложним задачу - пусть требуется, чтобы светодиод мигал с частотой точно один раз в секунду. Для этого воспользуемся блоком PLL

PLL - это генератор частоты с подстройкой фазы (ФАПЧ).



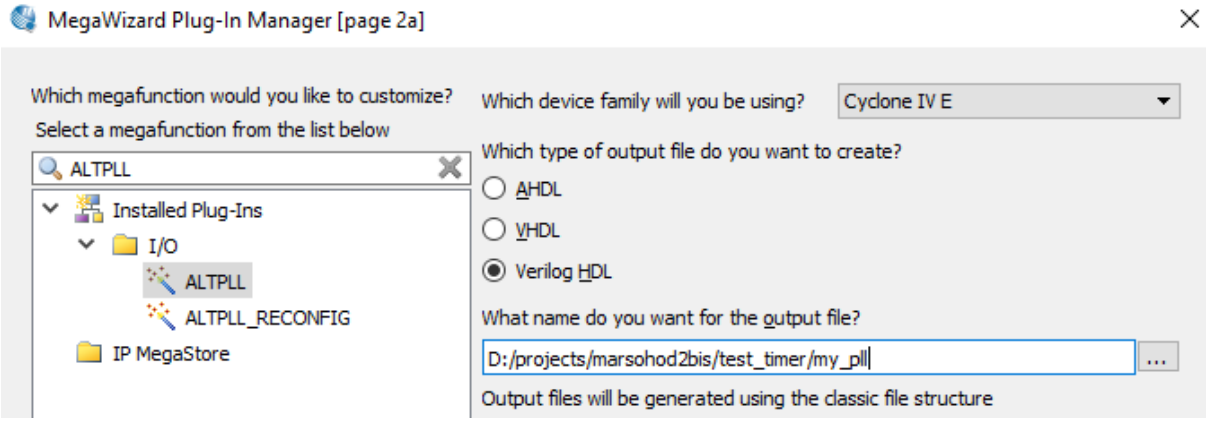

Фиг.5.1.9 устройство ФАПЧ (отсюда)

Он содержит в себе блок VCO[4] (ГУН[5]) с обратной связью, управляемой разностью фаз исходного и выходного сигналов. Полученный от ГУН сигнал проходит через делитель частоты и поступает на сравнение в Фазовый Детектор. Последний умножает мгновенные напряжения сигналов, пропускает через фильтр, который оставляет только низкие частоты и получившийся сигнал поступает как управляющий в ГУН.
В результате система с обратной связью после небольшого интервала времени (период захвата) стремится поддерживать нулевую разность фаз между исходным и вышедшим из делителя частоты сигналом. Что и требовалось.

Все блоки устройства можно настраивать, результатом этих настроек является возможность задать требуемую выходную частоту сигнала.

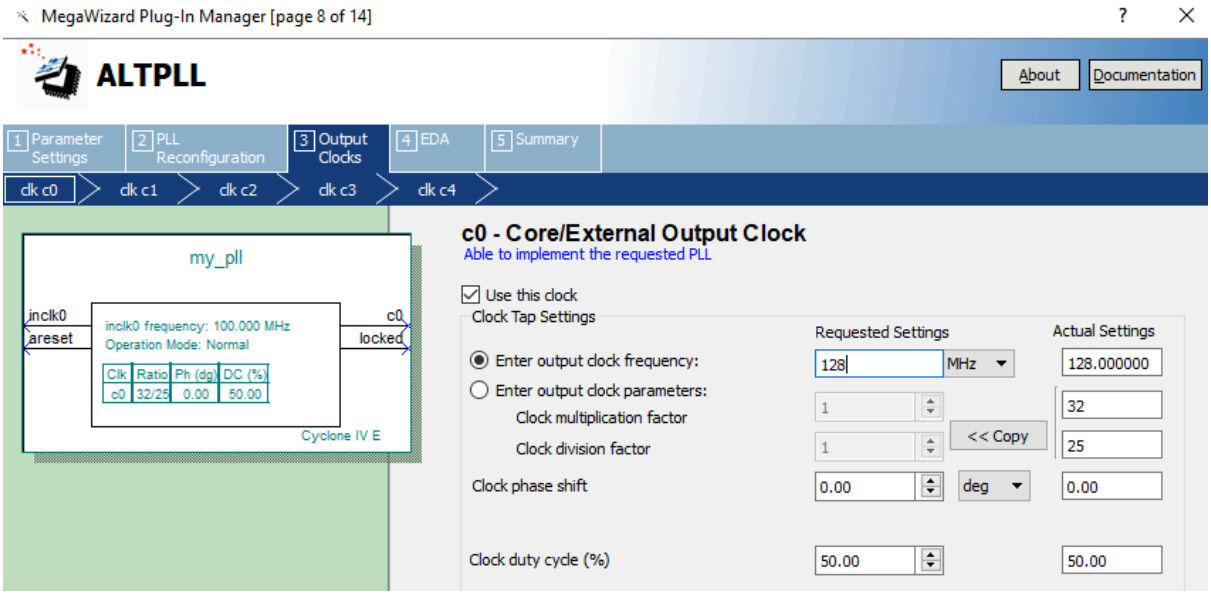
Для настройки встроенного блока PLL воспользуемся “MegaWizard Plug-In Manager”-ом Quartus. Он позволит произвести настройки блока PLL относительно понятным образом, скрыв кучу малозначащих (для нас в данный момент) деталей и создаст код Verilog для работы с блоком.

[3] Phase-Locked Loop
[4] Voltage-Controlled Oscillator
[5] Генератор, управляемый напряжением




Фиг.5.1.10 выбор I/O ALTPLL в окне MegaWizard'a

Откроется Wizard уже непосредственно блока PLL, в котором много настроек, которые, к счастью, можно оставить как есть. Самая важная для нас - страница 8, где задаются параметры выходной частоты.




Фиг.5.1.11 настройка выходной частоты

Выходная частота задаётся значениями умножителя и делителя частоты, их можно ввести вручную или просто задать выходную частоту и Wizard сам подберёт параметры. Зададим частоту в 134.218 МГц (128*1024*1024 = 134 217 728) и всё готово.

В данной PLL можно задать до пяти (c0..c4) выходных частот (не каких угодно, обычно имеющих общий делитель, например, 33, 66, 99, 132 МГц), но нам нужна только одна (c0). Можно нажать "Finish".

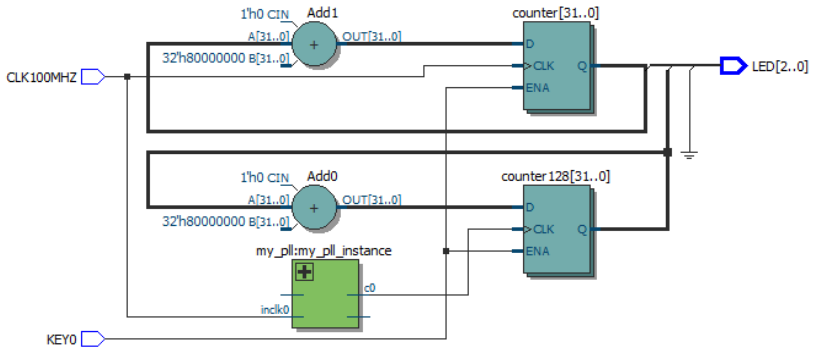
В папке проекта появились файлы с выбранным нами ранее префиксом my_pll, теперь следует подключить таймер к проекту.

```
 7    wire clk128Mhz;
 8  my_pll my_pll_instance(
 9      .inclk0( CLK100MHZ ),
10      .c0( clk128Mhz ),
11      .locked()
12      );
13  reg [31:0]counter128;
14  always @(posedge clk128Mhz)
15      if( KEY0 )
16          counter128 <= counter128+1;
17  assign LED[1] = counter128[27];
```

Фиг.5.1.12 подключение PLL средствами Verilog

- создаём внутренний сигнал clk128Mhz
- добавляем модуль типа me_pll
- входной тактовый сигнал my_pll : известный нам CLK100MHZ
- выходной сигнал my pll: clk128Mhz
- вход блокировки locked не задействован
- создаём счетчик counter128 аналогичный ранее использованному
- 27 разряд счетчика отправляем на LED[1],
  он должен мигать с частотой, близкой к 1 Гц.
- компилируем, запускаем, работает!

На работающем устройстве у нас мигают два светодиода - один с частотой 1 Гц, второй - 0.78 Гц (100 / 128)




Фиг.5.1.13 Схема получившегося устройства.

### 5.1.4 Hello world.

Отладочная плата содержит программатор  FTDI FT2232H, в котором есть два последовательных порта, один используется самим программатором для загрузки данных в микросхему FPGA, второй доступен для общения с внешним миром, скорость передачи до 12 Мбит/сек. Сейчас мы выведем что-нибудь в этот порт, чтобы увидеть это на экране компьютера.

Вообще, использовать отладочную печать - это очень полезный навык.
Он нам еще пригодится.

- создаём новый проект, он будет называться, скажем, test_strout.
- создаём, как мы это делали ранее, PLL настройки для частоты 0.1152 МГц, 115.2 КГц - одна из стандартных частот последовательного порта.
- подключаем получившийся модуль PLL

```
13    wire clkl15Khz;
14  my_pll my_pll_instance(
15      .inclk0( CLK100MHZ ),
16      .c0( clkl15Khz ),
17      .locked()
18  );
```

Фиг.5.1.14 подключаем модуль PLL средствами Verilog (test_strout.v)

- отправлять данные станем по нажатии кнопки, для этого нам потребуется поймать событие нажатия

```
21    reg [1:0]prev_key_state;
22    always @( posedge clkl15Khz )
23       prev_key_state <= { prev_key_state[0], KEY0 };
24
25    reg key_press_event;
26    always @( posedge clkl15Khz )
27       key_press_event <= (prev_key_state== 2'b10);
```

Фиг.5.1.15 отслеживаем момент нажатия на кнопку (test_strout.v).

Здесь мы слушаем генератор тактовой частоты и проверяем состояние кнопки. При этом запоминается два последних состояния кнопки. Если кнопка нажата, а в прошлый раз не была нажата, фиксируем этот факт. Вспомним, что согласно схеме отладочной платы, нажатая кнопка даёт 0, не нажатая: 1.

- Самое время заняться передачей данных. Передатчик предоставляет нам доступ к четырём “ногам”:
  - RxD - линия для приема данных (FTDI_BD0)
  - TxD - линия для передачи данных (FTDI_BD1)
  - RTS - сигнал запроса передачи (FTDI_BD2)
  - CTS - сигнал готовности приёмника (FTDI_BD3)

```
 1  module test_strout (
 2      input wire  CLK100MHZ,
 3      input wire  KEY0,
 4      output wire [2:0]LED,
 5
 6      //FTDI serial port signals
 7      input wire  FTDI_BD0,   //from FTDI, RxD
 8      output wire FTDI_BD1,   //to FTDI, TxD
 9      input wire  FTDI_BD2,   //from FTDI, RTS
10      output wire FTDI_BD3    //to FTDI, CTS
11  );
```

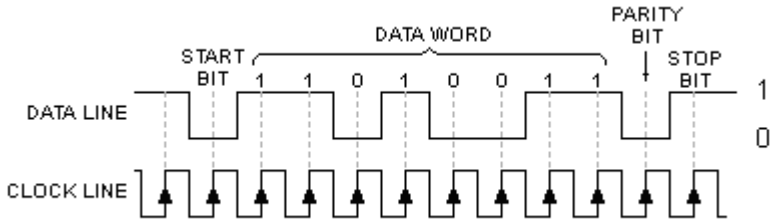
Фиг.5.1.17 описание сигнальных линий модуля

| | | | | | | |
|---|---|---|---|---|---|---|
| 6 | ✓ | ... | in | FTDI_BD0 | Location | PIN_24 | Yes |
| 7 | ✓ | ... | out | FTDI_BD1 | Location | PIN_28 | Yes |
| 8 | ✓ | ... | in | FTDI_BD2 | Location | PIN_11 | Yes |
| 9 | ✓ | ... | out | FTDI_BD3 | Location | PIN_10 | Yes |

Фиг.5.1.16 привязка выводов модуля

Однако, в нашем случае связь однонаправленная, передача ведётся на не очень высокой частоте (примерно 1% от максимальной), контроль состояния приемника можно опустить, достаточно одной линии TxD

- Последовательная передача данных имеет несколько вариантов. Мы станем использовать самый простой и самый распространёный вариант - 8-разрядные байты, по одному стартовому и стоповому разряду, без контроля чётности.




Фиг.5.1.17 передача байта через последовательный канал (отсюда)
в нашем случае parity bit не пишется

Стартовый бит всегда нулевой, стоповый - всегда единица. В результате после отправки байта может быть сколь угодно долгая задержка вплоть до получения нулевого сигнала.

- Модуль последовательной передачи назовём serial_tx
  - его интерфейс

```
1  module serial_tx(
2    input wire clk115,
3    input wire [7:0] sbyte,
4    input wire sbyte_rdy,
5    output wire tx,
6    output wire end_of_send
7  );
8
```

Фиг.5.1.18 интерфейс (serial_tx.v)

clk115 - выход с PLL с частотой 115.2 Кбит
sbyte - байт данных на отправку
sbyte_ready - команда на отправку
tx - линия последовательной отправки, TxD
end_of_send - сигнал что отправка байта закончена

- внутреннее состояние модуля

```
 9   reg [9:0] sreg;
10   assign tx = sreg[0];
11   reg [3:0] cnt = 0;
12   wire busy = (cnt != 0);
```

Фиг.5.1.19 состояние модуля

sreg - буфер для отправки - 8 бит данных + старт/стоп биты
младший разряд буфера выведен в tx, т.е. это значение на каждом такте уходит в последовательный порт
cnt - число еще не отправленных битов
busy - флаг, осталось ли еще что-нибудь на отправку

- собственно отправка

```
14   always @(posedge clk115)
15   begin
16     if(sbyte_rdy & ~busy) begin
17       sreg <= { 1'b1, sbyte, 1'b0 }; //load
18       cnt <= 9;
19     end else begin
20       sreg <= { 2'b11, sreg[9:1] }; //shift
21       if (busy)
22           cnt <= cnt - 1'b1;
23     end
24   end
```

Фиг.5.1.20 логика отправки

Если пришли новые данные, записываем их в буфер, обрамляя 0 (старт) и 1 (стоп) битами.
Если есть что отправлять, сдвигаем буфер на разряд вправо

- конец передачи

```
26   reg prev_busy;
27   always @(posedge clk115)
28     prev_busy <= busy;
29   assign end_of_send = busy==1'b0 && prev_busy==1'b1;
```

Фиг.5.1.21 проверка конца передачи

если сейчас отправлять нечего но на прошлом такте еще что-то было, значит взводим флаг (возврат значения из модуля)

- .встраиваем модуль передатчика в проект

```
37    reg [8*15-1:0] message = "*\n\r!dlroW olleH";
38    wire end_of_send;
39    wire send;
40    reg [15:0]message_bit_index;
41    wire [7:0]send_char;
42    assign send_char = message >> message_bit_index;
43
44    //use instance of serial port transmitter
45    serial_tx serial_tx_instance(
46      .clk115( clk115Khz ),
47      .sbyte( send_char ),
48      .sbyte_rdy( send ),
49      .tx( FTDI_BD1 ),
50      .end_of_send( end_of_send ),
51      .ack()
52    );
```

Фиг.5.1.22 встраиваем передатчик (test_strout.v)

message - строка, которую будем отправлять при нажатии кнопки,
в Verilog строчные литералы записываются справа налево,
заканчивается стоп-символом ‘*’
end_of_send - флаг, говорящий что байт отправлен
send - команда на отправку
message_bit_index - указатель (в разрядах) на отправляемый байт
send_char - байт на отправку

- готовимся к отправке

```
54    localparam STATE_WAIT_KEY_PRESS = 0;
55    localparam STATE_SEND_CHAR = 1;
56    localparam STATE_WAIT_CHAR_SENT = 2;
57
58    reg [1:0]state = STATE_WAIT_KEY_PRESS;
59
60    always @( posedge clk115Khz )
61    begin
62       case( state )
63       STATE_WAIT_KEY_PRESS:
64          begin
65             if( key_press_event ) state <= STATE_SEND_CHAR;
66          end
67       STATE_SEND_CHAR:
68          begin
69             state <= STATE_WAIT_CHAR_SENT;
70          end
71       STATE_WAIT_CHAR_SENT:
72          begin
73             if( end_of_send )
74                state <= ((send_char==8'h2A) ? STATE_WAIT_KEY_PRESS : STATE_SEND_CHAR);
75          end
76       endcase
77    end
```

Фиг.5.1.23 конечный автомат подготовки данных

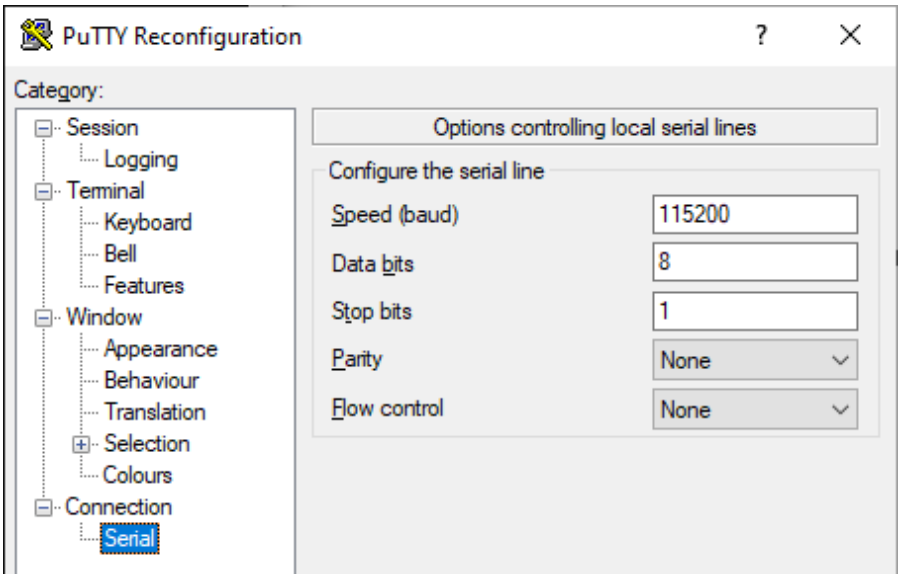
Выбирается нужный момент для отправки данных,
строка отправляется до тех пор, пока не встретится стоп-символ ’*’ (0x2A).

- Последний штрих, устанавливаем указатель в буфере и взводим флаг отправки

```
72    assign send = (state == STATE_SEND_CHAR);
73    always @( posedge clkl15Khz )
74       if( state==STATE_WAIT_KEY_PRESS )
75          message_bit_index <= 0;
76       else
77       if( state==STATE_SEND_CHAR )
78          message_bit_index <= message_bit_index+8;
```

Фиг.5.1.24 указатель в буфере и флаг отправки

- Компилируем, запускаем.
  Чтобы проверить работоспособность, необходимо прочитать виртуальный порт на стороне компьютера.
  Автор сделал это с помощью программу pytty (windows)




Фиг.5.1.25 настройки чтения порта

Список имеющихся портов можно посмотреть в реестре:
\HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\SERIALCOMM

Работает, “проверено электроникой”(C).

### 5.1.5 Работа с памятью.

Микросхема Cyclone IV содержит 270 кбит встроенной SRAM памяти, этого более чем достаточно на первых порах. Поучимся работать с этой памятью.

Речь идёт о встроенной (SRAM) памяти ПЛИС, которой в нашем случае аж 270 кбит. Средствами Verilog доступ к этой памяти очень прост - достаточно написать

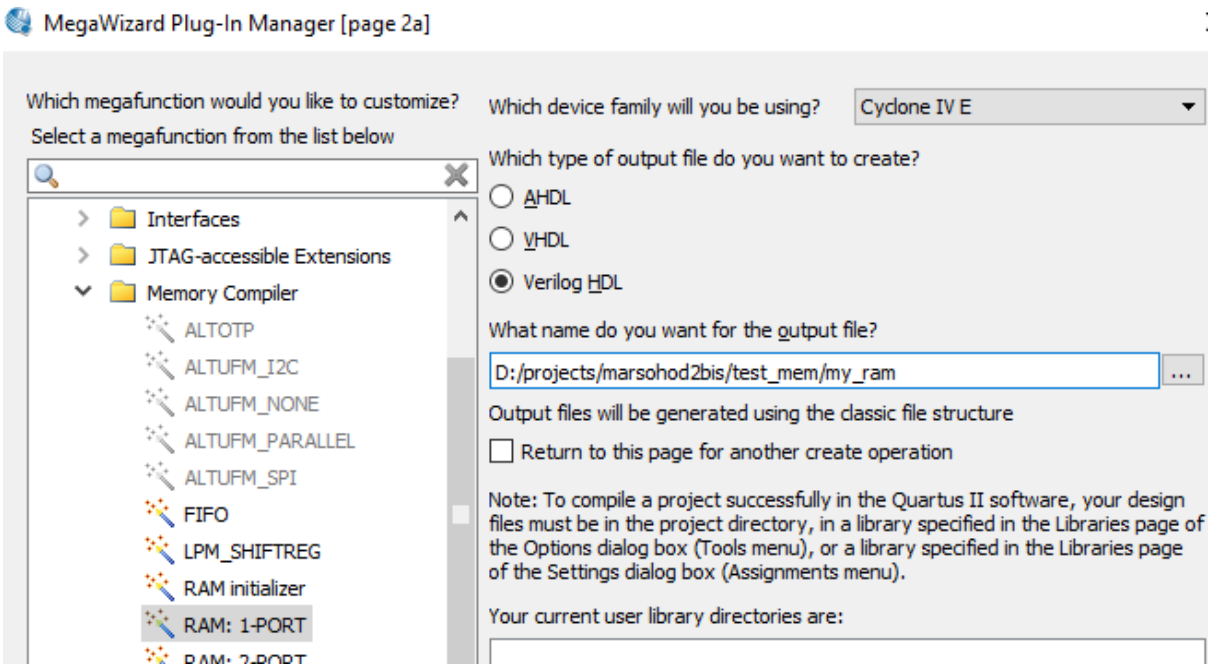
```
21      reg [DATA_WIDTH-1:0]  mem [DEPTH];
22      ...
23      mem[addr] <= tmp_data; // запись
24      tmp_data <= mem[addr]; // чтение
```

Фиг.5.1.26 объявление и работа с памятью средствами Verilog
DEPTH - размер слова памяти
DATA_WIDTH - размер массива в словах

как где-то в массиве памяти будет выделен блок и мы получим прямой доступ к его элементам просто по индексу.

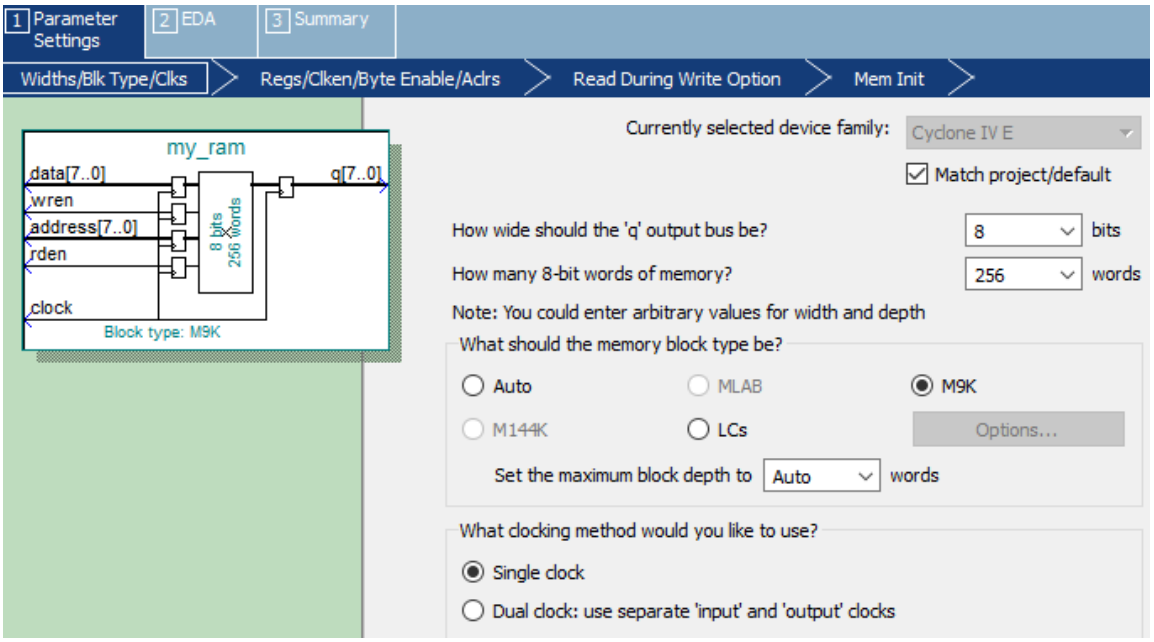
Однако же, мы не можем опираться только на память такого рода, ведь наш задуманный процессор должен иметь доступ к скомпилированому коду, который каким-то образом надо уметь до него донести. К счастью, такая задача возникла не в первый раз и давно решена. Но нам опять придётся воспользоваться услугами “MegaWizard Plug-In Manager”-а.

Но сначала потребуется создать новый проект, который будет базироваться на ранее представленном примере “Hello world”.



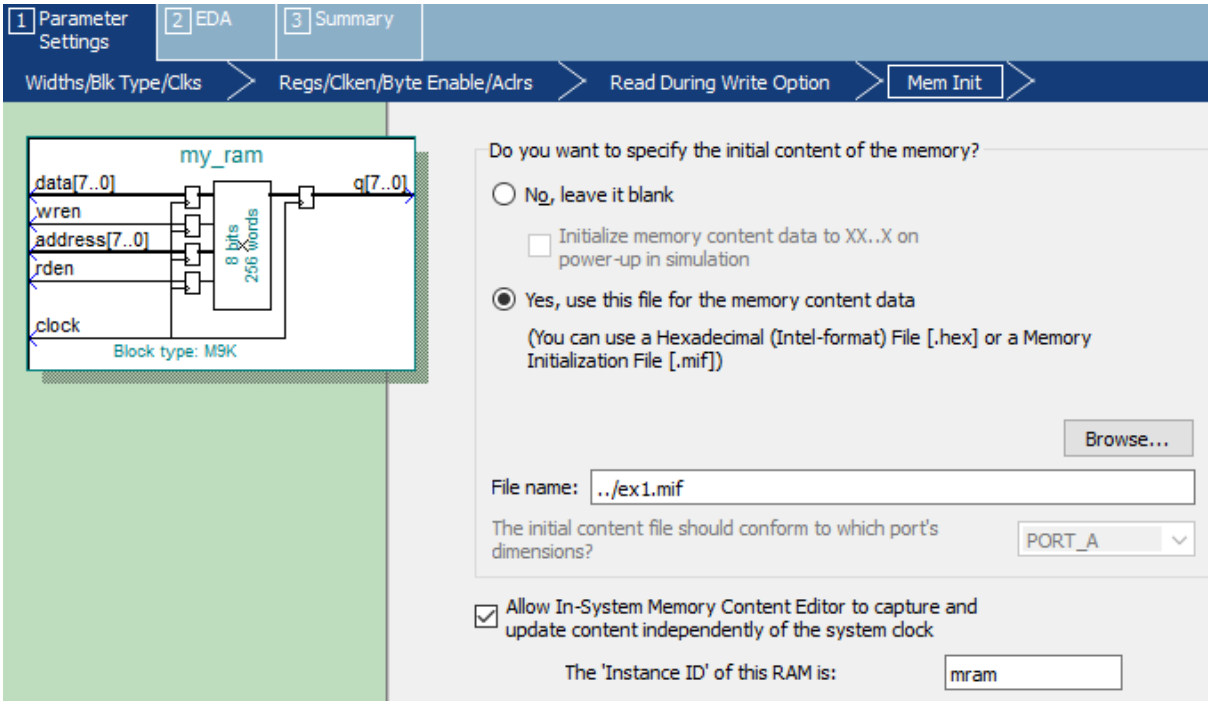

Фиг.5.1.27 запуск “мега-визарда”

На этот раз будем использовать однопортовую RAM память из раздела “Memory Compiler”.




Фиг.5.1.28 настройки размера

Задаём размер области памяти, далее можно попросить создать управляющие сигналы (или оставить всё по умолчанию). Важная настройка - инициализация памяти.




Фиг.5.1.29 инициализация памяти.

Нет, мы не хотим, чтобы они остались заполненными нулями и предпочитаем инициализацию из файла.

Кстати, галочка внизу очень важная, выбрав ее и задав Instance ID, получаем возможность пользоваться утилитой просмотра/редакции внутренней памяти “Tools / In-System Memory Content Editor”.

Файл ex1.mif текстовый, его формат довольно прост и показан на Фиг.5.1.30.

```
WIDTH = 8;              -- размер слова
DEPTH = 256;            -- число слов
ADDRESS_RADIX = HEX;    -- адрес представлен шестнадцатеричным числом
DATA_RADIX = HEX;       -- данные представлены ..
CONTENT BEGIN
0000 : 30;
0001 : 31;
0002 : 32;
0003 : 33;
0004 : 34;
0005 : 35;
...
00FA : 0d;
00FB : 0A;
00FC : 2A;
00FD : 20;
00FE : 20;
00FF : 20;
END
```

Фиг.5.1.30 MIF-файл

После завершения работы “мега-визарда”, в папке с проектом появится несколько файлов с префиксом “my_ram” (который мы выбрали в визарде) .
Теперь можно подключить модуль памяти к проекту. В основном файле (test_mem.v) добавляем

```
38    reg [7:0] rd_char;
39    reg  [7:0] addr2 = 0;
40
41  my_ram my_mem2_test (
42      .address( addr2 ),
43      .clock( clkl15Khz ),
44      .q ( rd_char ));
45
```

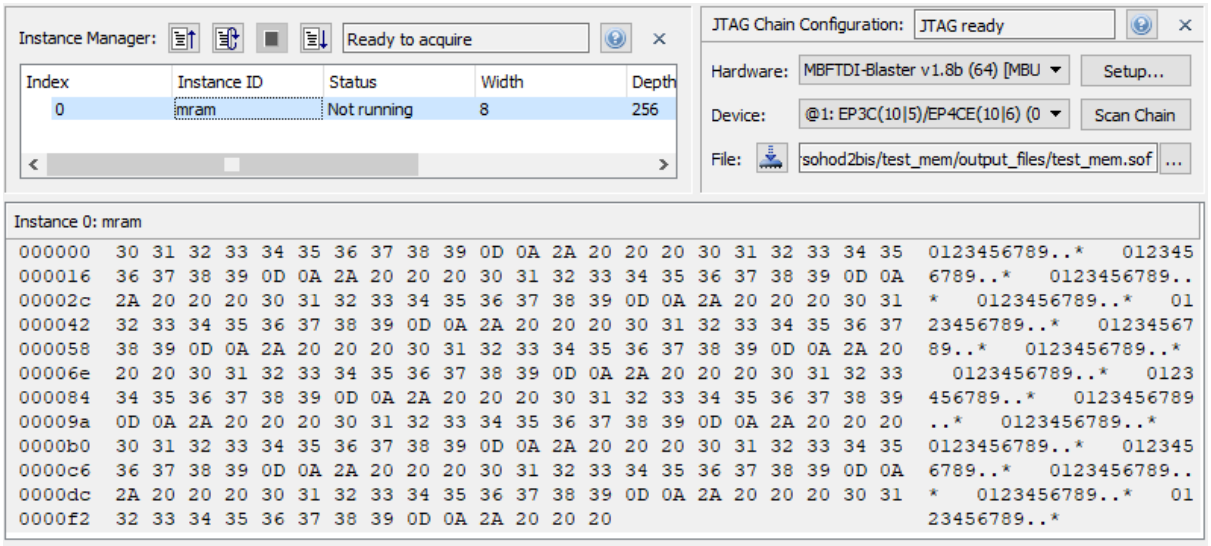
Фиг.5.1.31 встраиваем модуль памяти в проект

Поскольку наш проект основан на “Hello world”, его рабочая частота 115 200 Гц, этого более чем достаточно чтобы память работала синхронно. Каждый раз при приходе восходящего фронта сигнала clock, в переменной rd_char будет появляться содержимое байта памяти с индексом addr2.

Если раньше на каждом такте в последовательный порт отправлялся байт из внутреннего буфера, то теперь байт, прочитанный из памяти.

Компилируем, запускаем, работает.

Немного о редакторе памяти. Если мы не забыли поставить соответствую галочку в “мега-визарде”, то появляется возможность на-лету смотреть и менять содержимое блока памяти. Как это выглядит, показано на Фиг.5.1.32.




Фиг.5.1.32 содержимое блока “mram”

.