Министерство образования и науки Российской Федерации
ТОМСКИЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

КАФЕДРА ТЕЛЕКОММУНИКАЦИЙ И ОСНОВ РАДИОТЕХНИКИ (ТОР)

УТВЕРЖДАЮ
Зав. кафедрой ТОР
А.Я. Демидов

Методические указания по проведению практических занятий и организации самостоятельной работы

по дисциплине «**Программирование логических интегральных схем**»

РАЗРАБОТЧИКИ:
_______ Д.А. Покаместов,
___________ Я.В. Крюков,
_________ Ж.Т. Эрдынеев,
аспиранты каф. ТОР

ТОМСК – 2014

Оглавление

1. Введение....2

2. Цифровые логические схемы....3

3. Основные элементы и функции языка Verilog....9

4. Триггеры....20

5. Мультиплексор, демультиплексор, дешифратор, счетчик....23

6. Сдвиговые регистры, счетчик....26

7. Верификация проектов с помощью Modelsim....28

8. Логический анализатор SignalTap II....39

9. MegaWizard....45

10. Машина конечных состояний....49

11. Модули памяти....54

12. Фильтрация ПЛИС....61

13. Согласование модулей....68

1. Введение

Программируемые логические интегральные схемы - класс устройств, способных менять логику работы в соответствии с прошивкой. Для программирования используются программаторы и отладочные среды, позволяющие задать структуру устройства с помощью описания на специальных языках: Verilog, VHDL, AHDL.

Учебное пособие содержит базовые сведения для обучения проектированию цифровых систем на основе конструкций языка Verilog. Рассмотрен пакет проектирования Quartus II, его компоненты MegaWizard и SignalTap, пакет симуляции ModelSim.

Включены задания для самостоятельной работы, они начинаются со второй половины пособия, после знакомства с базовыми понятиями и конструкциями Verilog

2. Цифровые логические схемы.

Простые логические элементы.

Логические операции совсем не сложны, и мы встречаемся с ними каждый день. Можно привести пару примеров:

а) Внутреннее освещение автомобиля включено, когда открыта левая ИЛИ правая дверь ИЛИ открыты обе двери.

б) Сигнальная лампа ручного тормоза горит, когда включен ручной тормоз И при этом включено зажигание.

в) Центрифуга будет работать только тогда, когда крышка НЕ открыта.

В примере приведены три наиболее важные логические функции: ИЛИ, И и НЕ, комбинируя которые можно получить следующие простейшие логические элементы: ИЛИ-НЕ, И-НЕ, исключающее ИЛИ, исключающее ИЛИ-НЕ. Графическое отображение логических схем по российскому (ГОСТ) и зарубежному (ANSI) стандарту приведено на рис. 2.1.

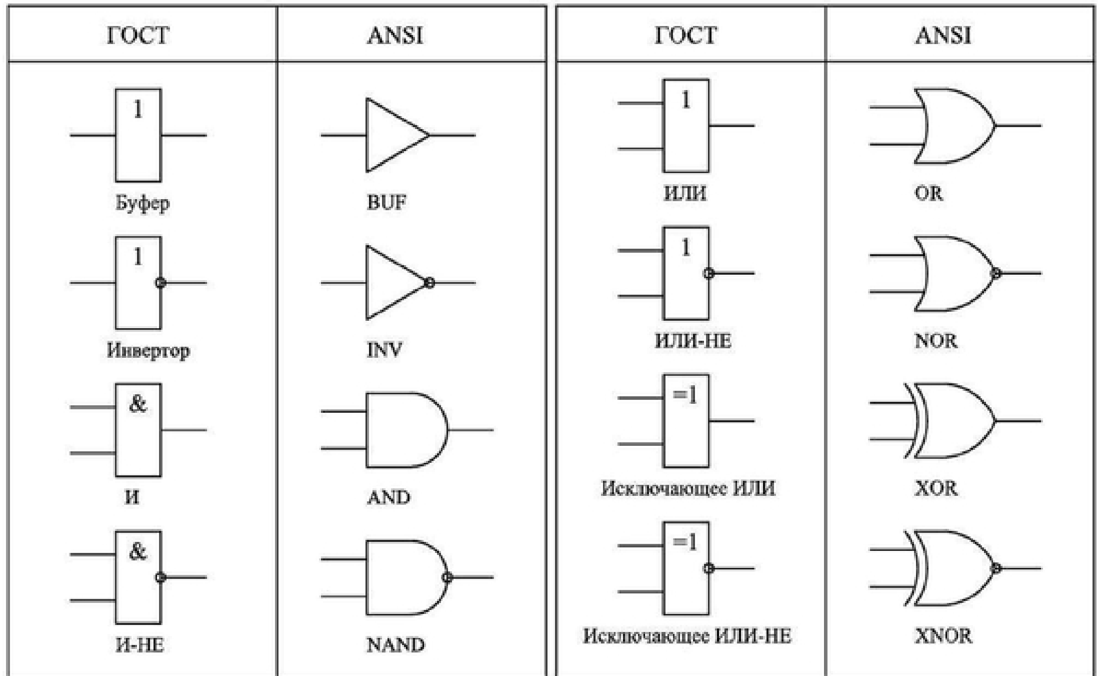
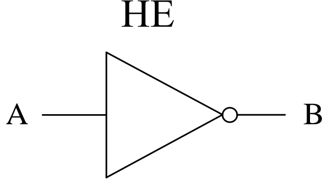


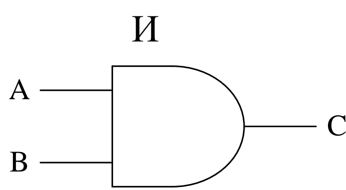
Рисунок 2.1 - Графическое отображение логических схем по российскому (ГОСТ) и зарубежному (ANSI) стандартам

Работа логических схем описывается с помощью булевой алгебры. Другой способ, значение которого при конструировании логических устройств исключительно велико, состоит в использовании таблиц истинности или табличной записи функции. Согласно этому методу для логического элемента или системы в целом просто перечисляются все

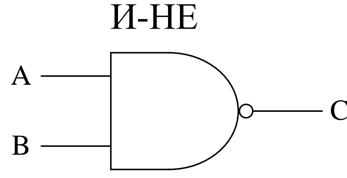
возможные комбинации значений входных и выходных сигналов. Таблицы истинности и схема образования для вышеописанных логических элементов приведены ниже.



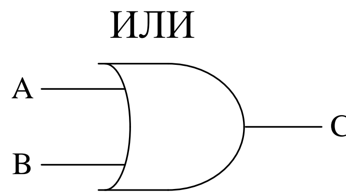
Вход	Выход
A	B
1	0
0	1



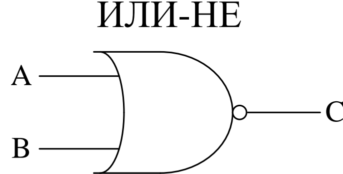
Вход		Выход
A	B	C
0	0	0
1	0	0
0	1	0
1	1	1



Вход		Выход
A	B	C
0	0	1
1	0	1
0	1	1
1	1	0



Вход		Выход
A	B	C
0	0	0
1	0	1
0	1	1
1	1	1



Вход		Выход
A	B	C
0	0	1
1	0	0
0	1	0
1	1	0

На рисунке 2.2 изображен полусумматор, на входы которого поступают одноразрядные двоичные числа, называемые, как правило, битами. Схема выдает бит суммы и необходимую цифру переноса. Если сигнал переноса не используется, то полусумматор называют схемой ИСКЛЮЧАЮЩЕЕ ИЛИ, “неравенства” или несовпадения. Происхождение этих названий обусловлено тем, что выход равен нулю всякий раз, когда оба входа имеют один и тот же логический уровень, и на выходе появляется 1, когда входные сигналы различны.

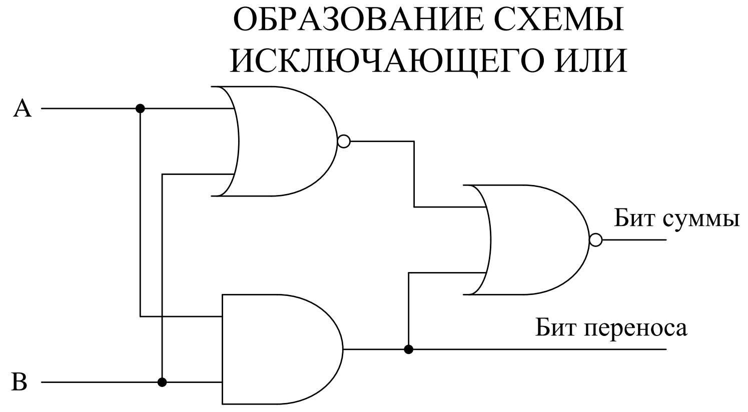
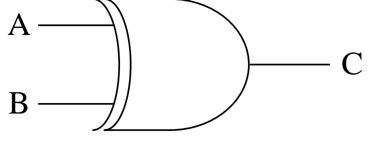


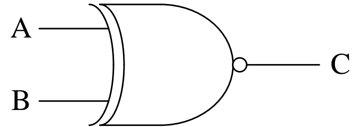
Рисунок 2.2 – Полусумматор

ИСКЛЮЧАЮЩЕЕ ИЛИ



Вход		Выход
A	B	C
0	0	0
1	0	1
0	1	1
1	1	0

ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ



Вход		Выход
A	B	C
0	0	1
1	0	0
0	1	0
1	1	1

Последовательные логические схемы – триггеры и память.

Одним из наиболее распространенных элементов памяти является триггер (мультивибратор с двумя устойчивыми состояниями). Итак, триггер - логическое устройство, обладающее способностью находиться в одном из устойчивых состояний (0 или 1). Простейшие триггеры:

RS – триггер;

Синхронный RS – триггер;

D – триггер;

JK – триггер.

RS-триггер

Для начала рассмотрим простейший RS – триггер (рис. 2.3) с входом S (Set) установки состояния 1 и входом R (Reset) установки состояния 0. Если на обоих входах S и R поддерживается логический 0, то схемы ИЛИ-НЕ работают просто как инверторы и триггер сохраняет свое состояние неограниченно долго.

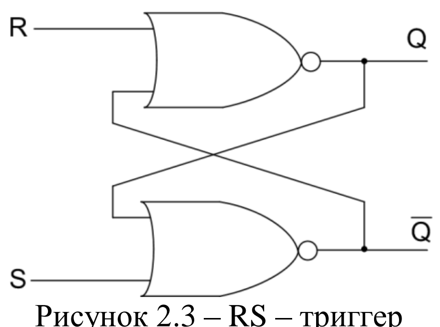


Рисунок 2.3 – RS – триггер

Если на входы R и S одновременно падать 1, то оба входа схем ИЛИ-НЕ примут значения 0. Это состояние нельзя будет запомнить, когда сигналы на входах R и S вернуться к значению 0, поскольку перекрестная связь требует, чтобы при этом выходы имели различные значения. Практически запомненное состояние будет зависеть от того, какой из входов R или S первым примет значения 0. Когда это зависит от случая, комбинация входных сигналов R=S=1 приводит к неопределенности сохраняемых данных, и эта комбинация никогда не должна сопровождаться комбинацией R=S=0. Условное обозначение RS-триггера изображено на рис.2.4. Таблица истинности (переходов) RS-триггера отображена в табл. 2.1.

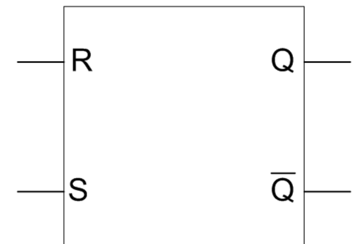


Рисунок 2.4 – Условное обозначение схемы RS-триггера

Таблица 2.1 – Таблица истинности переходов RS - триггера

Вход		Выход	
R	S	Q	НЕ Q
0	0	Qхан	НЕ Qхран
1	0	0	1
0	1	1	0
1	1	0	0
		неопред.	

Синхронный RS-триггер

Существенной чертой большинства последовательных логических систем является необходимость осуществлять переходы в определенные моменты времени. Обычно это достигается с помощью регулярной последовательности тактовых импульсов, которые управляют последовательностью событий. На рис. 2.5 показан RS-триггер, устроенный так, что он может изменять свое состояние только тогда, когда входные тактовые импульсы (Clock Pulse, CP) принимают значение логической 1.

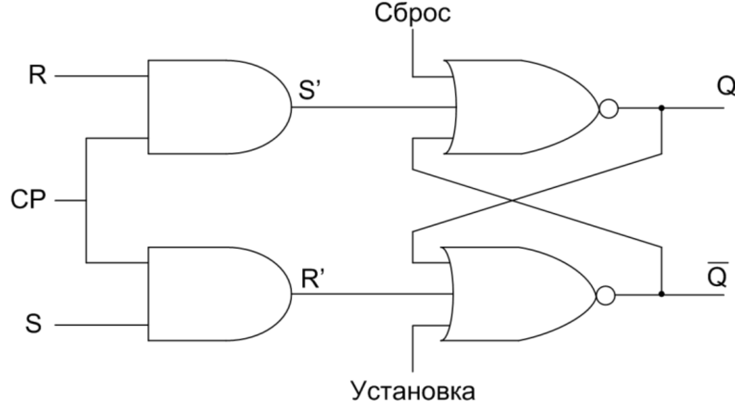


Рисунок 2.5 – Схема синхронного RS-триггер

Дополнительные входы *установка* и *сброс*, на которых нормально поддерживается значение логического 0, обеспечивают непосредственную реализацию собственной способности триггера устанавливаться в единичное состояние или сбрасываться путем

использования дополнительных входов у элементов ИЛИ-НЕ. Полезным свойством любого синхронного триггера является возможность устанавливать его состояние независимо от тактового сигнала путем кратковременного подъема напряжения на соответствующем входе до уровня логической 1. Таблица истинности для синхронного RS-триггера аналогична таблице истинности для RS-триггера.

D-триггер

D-триггер, или триггер данных является синхронным RS-триггером, управляемым только по одному входу. Его достоинство заключается в том, что входы R и S не могут одновременно принять значение логической 1 и привести к сохранению неопределенного значения сигнала на выходе. Схема триггера показана на рис. 2.6. Прямоугольником на схеме изображен синхронный RS-триггер. Состояние триггера сохраняется до тех пор, пока логический уровень на входе не изменится с 0 на 1, когда любой из логических уровней на входе D передается на выход Q. Таблица истинности для D-триггера представлена в табл. 2.2.

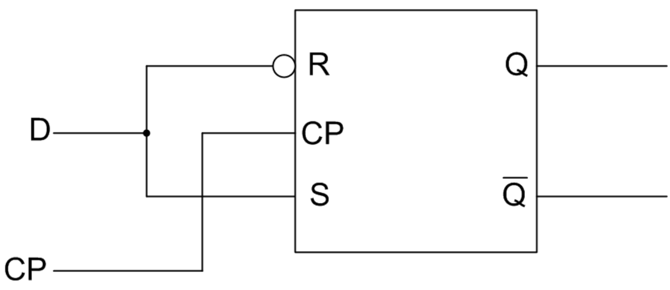


Рисунок 2.6 – Схема D-триггера

Таблица 2.2 – Таблица истинности переходов D-триггера

Вход	Выход
D	Q
1	1
0	0

JK-триггер

Самым гибким усовершенствованием RS-триггера является JK-триггер, схема которого изображена на рис. 8. В отличии от D-триггера здесь два входа, но удается избежать неопределенности путем запоминаемого состояния при S=R=1 путем стробирования каждого из входных сигналов сигналом с противоположного входа. Принято стробируемый вход установки обозначать буквой J, а стробируемый вход сброса – буквой K. Триггер чувствителен к входным сигналам только тогда, когда тактовый

сигнал (CP) принимает высокий уровень. Таблица истинности JK-триггера представлена в табл. 2.3.

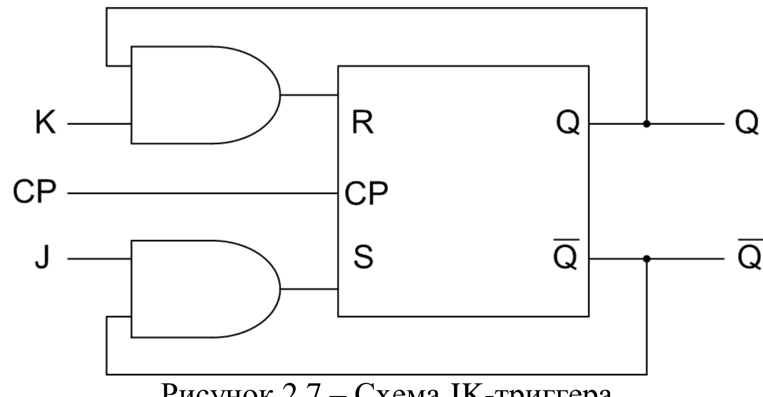


Рисунок 2.7 – Схема JK-триггера

Таблица 2.3– Таблица истинности переходов JK-триггера

Вход		Выход
J	K	Q
0	0	Q0
1	0	1
0	1	0
1	1	НЕ Q0

3. Основные элементы и функции языка Verilog.

Проводники и регистры.

Один из базовых типов источника сигнала в языке Verilog – это цепь или проводник, *wire*. Таким образом, результат арифметического или логического выражения можно ассоциировать с именованным проводником и позже использовать его в других выражениях. Значение проводника (*wire*) – это функция того, что присоединено к нему.

Пример декларации однобитного проводника в программе, написанной на языке Verilog:

```
wire a;
```

Назначение проводнику *a* сигнала “*b*”:

```
wire b;
assign a = b;
```

Тоже самое в одном выражении:

```
wire a = b;
```

Проводники могут передавать несколько бит:

wire [3:0] c; //это четыре провода

Проводники, передающие несколько бит информации называются “шиной”, или “вектором”:

wire [3:0] d;

assign c = d; //“подключение” одной шины к другой

Количество проводников в шине определяется любыми двумя целыми числами разделенными двоеточием внутри квадратных скобок.

wire [11:4] e; //восьмибитная шина

wire [0:255] f; //256-ти битная шина

Из шины можно выделить некоторые нужные биты и назначить другому проводу:

wire g;

assign g = f[2]; //назначить сигналу “g” второй бит шины “f”

Кроме того, выбираемый из шины бит может определяться переменной:

wire [7:0] h;

wire i = f[h]; // назначить сигналу “i” бит номер “h” из шины “f”

Из сигнальной шины можно выделить некоторый диапазон битов и назначить другой шине с тем же количеством битов:

wire [3:0] j = e[7:4];

Так же, в большинстве диалектов Verilog, можно определять массивы сигнальных шин:

wire [7:0] k [0:19]; //массив из двадцати 8-ми битных шин

Существует другой тип источника сигнала называемый регистр: *reg*. Регистр *reg* в языке Verilog обозначает переменную, которая может хранить значение. Тип *reg* используется при поведенческом (*behavioral)* и процедурном описании цифровой схемы. Если регистру постоянно присваивается значение комбинаторной (логической) функции, то он ведет себя точно как проводник (*wire*). Если же регистру присваивается значение в синхронной логике, например по фронту сигнала тактовой частоты, то ему, в конечном счете, будет соответствовать физический D-триггер или группа D триггеров. D-триггер – это логический элемент способный запоминать один бит информации. В англоязычных статьях D-триггер называют *flipflop*.

Регистры описываются так же, как и проводники:

reg [3:0] m;

reg [0:100] n;

Регистры могут использоваться, как и проводники - в правой части выражений, как операнды:

wire [1:0] p = m[2:1];

Можно определить массив регистров, которые обычно называют “память”:

reg [7:0] q [0:15];   //память из 16 слов, каждое по 8 бит

Еще один тип источника сигнала – это *integer*, который аналогичен регистру *reg*, но всегда является 32х битным знаковым типом данных. Например:

integer loop_count;

Verilog позволяет группировать логику в блоки. Каждый блок логики называется “модулем” (*module*). Модули имеют входы и выходы, которые ведут себя как сигналы *wire*. При описании модуля сперва перечисляют его порты (входы и выходы):

module my_module_name (port_a, port_b, w, y, z);

А затем описывают направление сигналов:

```
input port_a;
output [6:0] port_b;
input [0:4] w;
inout y; //двунаправленный сигнал, обычно используется
  //только для внешних контактов микросхем
```

Позже будет ясно, что выход модуля может быть сразу декларирован как регистр *reg*, а не как провод *wire:*

```
output [3:0] z;
reg [3:0] z;
```

Позволяется сразу в описании модуля указать тип и направление сигналов:

```
module my_module
(
input wire port_a,
output wire [6:0]port_b,
input wire [0:4]w,
inout wire y,
output reg [3:0]z
);
```

После этого можно использовать входные сигналы, как провода *wire*:

wire r = w[1];

Теперь позволяется делать постоянные назначения выходам, как функции от входов:

assign port_b = h[6:0];

В конце описания логики каждого модуля пишем слово *endmodule*.

```
module my_module_name (input wire a, input wire b, output wire c);
assign c = a & b;
endmodule
```

Последний тип источника сигнала, о котором стоит упомянуть – это постоянные сигналы или просто числа:

```
wire [12:0] s = 12;   /* 32-х битное десятичное число, которое будет
"обрезано" до 13 бит */
wire [12:0] z = 13'd12;   //13-ти битное десятичное число
wire [3:0] t = 4'b0101;   //4-х битное двоичное число
wire [7:0] q = 8'hA5;     // 8-ми битное шестнадцатеричное число A5
wire [63:0] u = 64'hdeadbeefcafebabe;   /*64-х битное
шестнадцатеричное число */
```

Если точно не определить размер числа, то оно принимается по умолчанию 32-х разрядным. Это может быть проблемой при присвоении сигналам с большей или меньшей разрядностью.

```
wire [3:0] aa;
wire [3:0] bb;
assign bb = aa + 1;
```

Арифметические и логические функции.

Сложение и вычитание.

Приведен пример модуля, который одновременно и складывает и вычитает два числа. Здесь входные операнды 8-ми битные, а результат 9-ти битный. Verilog корректно сгенерирует бит переноса *(carry bit)* и поместит его в девятый бит выходного результата. С точки зрения Verilog входные операнды беззнаковые.

```
module simple_add_sub(
operandA, operandB,
out_sum, out_dif);
//два входных 8-ми битных операнда
input [7:0] operandA, operandB;
/*Выходы для арифметических операций имеют дополнительный 9-й бит
переполнения*/
output [8:0] out_sum, out_dif;
assign out_sum = operandA + operandB; //сложение
```

```
assign out_dif = operandA - operandB; //вычитание
endmodule
```

Логический и арифметический сдвиг.

Приведем пример модуля, который выполняет сдвиги. В этом примере результат для сдвига влево 16-ти битный. При сдвиге влево или вправо не большое число разрядов,то в результате получится ноль. Часто для сдвига используются только часть бит от второго операнда, для того, чтобы упростить логику.

```
module simple_shift (
operandA, operandB,
out_shl, out_shr, out_sar);
// два входных 8-ми битных операнда
input [7:0] operandA, operandB;
// Выходы для операций сдвига
output [15:0] out_shl;
output [7:0] out_shr;
output [7:0] out_sar;
//логический сдвиг влево
assign out_shl = operandA << operandB;
/* пример: на сколько сдвигать определяется 3-мя битами второго
операнда */
assign out_shr = operandA >> operandB[2:0];
//арифметический сдвиг вправо (сохранение знака числа)
assign out_sar = operandA >>> operandB[2:0];
endmodule
```

Битовые логические операции

Битовые операции в Verilog выглядят так же, как и в языке C. Каждый бит результата вычисляется отдельно соответственно битам операндов. Пример:

```
module simple_bit_logic (
operandA, operandB,
out_bit_and, out_bit_or, out_bit_xor, out_bit_not);
//два входных 8-ми битных операнда
```

```
input [7:0] operandA, operandB;
//Выходы для битовых (bit-wise) логических операций
output [7:0] out_bit_and, out_bit_or, out_bit_xor, out_bit_not;
assign out_bit_and = operandA & operandB; //И
assign out_bit_or = operandA | operandB; //ИЛИ
assign out_bit_xor = operandA ^ operandB; //исключающее ИЛИ
assign out_bit_not = ~operandA; //НЕ
endmodule
```

Булевы логические операции.

Булевы логические операторы отличаются от битовых операций. Так же, как и в языке C, здесь значение всей шины рассматривается как ИСТИНА если хотя бы один бит в шине не 0 или ЛОЖЬ, если все биты шины – 0. Результат получается всегда однобитный (независимо от разрядности операндов) и его значение "1" (ИСТИНА) или "0" (ЛОЖЬ).

```
module simple_bool_logic (
operandA, operandB,
out_bool_and, out_bool_or, out_bool_not);
//два входных 8-ми битных операнда
input [7:0] operandA, operandB;
// Выходы для булевых (boolean) логических операций
output out_bool_and, out_bool_or, out_bool_not;
assign out_bool_and = operandA && operandB; //И
assign out_bool_or = operandA || operandB; //ИЛИ
assign out_bool_not = !operandA; //НЕ
endmodule
```

Операторы редукции.

В Verilog существуют операторы редукции. Эти операторы позволяют выполнять операции между битами внутри одной шины. Так, можно определить все ли биты в шине равны единице *(&bus),* или есть ли в шине хотя бы одна единица *(|bus)*. Приведу пример:

```
module simple_reduction_logic (
operandA,
out_reduction_and, out_reduction_or, out_redution_xor);
//входной 8-ми битный операнд
```

```
input [7:0] operandA;
// Выходы для логических операций редукции
output out_reduction_and, out_reduction_or, out_reduction_xor;
assign out_reduction_or = |operandA;
assign out_reduction_and = &operandA;
assign out_reduction_xor = ^operandA;
endmodule
```

Дополнительные операторы редукции:

~|operandA обозначает, что в шине нет единиц.

~&operandA обозначает, что некоторые биты в шине равны нулю.

Оператор условного выбора

В языке Verilog существует оператор условного выбора, который фактически реализует мультиплексор. В данном примере на выходе мультиплексора окажется значение *operandA* если сигнал *sel_in* единица. И наоборот. Если входной сигнал *sel_in* равен нулю, то на выходе мультиплексора будет значение *operandB*.

```
module simple_mux (
operandA, operandB, sel_in, out_mux);
//входные 8-ми битные операнды
input [7:0] operandA, operandB;
//входной сигнал селектора
input sel_in;
//Выход мультиплексора
output [7:0]out_mux;
assign out_mux = sel_in ? operandA : operandB;
endmodule
```

Операторы сравнения

Продемонстрируем оператор сравнения на примере:

```
module simple_compare (
operandA, operandB,
out_eq, out_ne, out_gt, out_lt, out_ge, out_le);
//входные 8-ми битные операнды
input [7:0] operandA, operandB;
```

```
//Выходы операций сравнения
output out_eq, out_ne, out_gt, out_lt, out_ge, out_le;
assign out_eq = operandA == operandB; //равно
assign out_ne = operandA != operandB; //не равно
assign out_ge = operandA >= operandB; //больше или равно
assign out_le = operandA <= operandB; //меньше или равно
assign out_gt = operandA > operandB; //больше
assign out_lt = operandA < operandB; //меньше
endmodule
```

В приведенных примерах были рассмотрены основные арифметические и логические операторы языка Verilog. Далее рассмотрим процедурные блоки.

Процедурные блоки

Ранее были рассмотрены постоянные назначения сигналов:

```
wire a,b,c;
assign c = a & b;
```

Постоянные назначения весьма полезны, но с не сколькими недостатками. Самый главный недостаток такого кода – сложность чтения написанного кода. Чтобы сделать язык Verilog более выразительным, существуют так называемые "*always*" блоки. Они используются при описании системы с помощью процедурных блоков. Оно позволяет выразить алгоритм так, чтобы он выглядел как последовательность действий. Для описания процедурного блока используется следующий синтаксис:

```
always @(<sensitivity_list>) <statements>
```

<sensitivity_list> – это список всех входных сигналов, к которым чувствителен блок.

Это список входных сигналов, изменение которых влияет выходные сигналы этого блока. "*Always*" переводится как "всегда". Иначе запись читается так: "Всегда выполнять выражения *<statements>* при изменении сигналов, описанных в списке чувствительности *<sensitivity list>*".

Если указать список чувствительности неверно, то это не должно повлиять на синтез проекта, но может повлиять на его симуляцию. В списке чувствительности имена входных сигналов разделяются ключевым словом "*or*":

```
always @(a or b or d) <statements>
```

Включение в список чувствительности всех сигналов:

```
always @* <statements>
```

Тогда исправляя выражения в *<statements>* не нужно задумываться об изменении списка чувствительности. При описании выражений внутри процедурных блоков комбинаторной логики, с правой стороны от знака равенства можно использовать типы сигналов *wire* или *reg*, а вот с левой стороны теперь используется только тип *reg*:

```
reg [3:0] c;
always @(a or b or d)
begin
c = <выражение использующее входные сигналы a,b,d>;
end
```

Обратите внимание, что регистры, которым идет присвоение в таких процедурных блоках не будут выполнены в виде D-триггеров после синтеза. В этом случае присвоение регистрам происходит с помощью оператора "=", который называется "блокирующим". Для симулятора это означает, что выражение вычисляется, его результат присваивается регистру приемнику и он тут же, немедленно, может быть использован в последующих выражениях. Блокирующие присвоения обычно используются для описания комбинаторной логики в процедурных блоках. Не блокирующие присвоения будут описаны позднее – они обычно используются для описания синхронной логики и вот уже там регистры *reg* после синтеза будут представлены с помощью D триггеров. Нельзя путать блокирующие и не блокирующие присвоения.

```
wire [3:0] a, b, c, d, e;
reg [3:0] f, g, h, j;
always @(a or b or c or d or e)
begin
f = a + b;
g = f & c;
h = g | d;
j = h - e;
end
```

То же самое можно сделать по другому, вот так:

```
always @(a or b or c or d or e)
begin
j = (((a + b) & c) | d) - e;
end
```

После того, как проект будет откомпилирован, список всех сигналов проекта *(netlist)* может сильно сократиться. Многие описанные сигналы, могут исчезнуть –

синтезатор выбросит их, создав цепи из оптимизированной комбинаторной логики. В нашем примере сигналы *f, g* и *h* могут исчезнуть из списка сигналов проекта после синтезатора, все зависит от того используются ли эти сигналы где-то еще в проекте или нет. Синтезатор даже может выдать предупреждение *(warning)* о том, что сигналу *"f"* присвоено значение, но оно нигде не используется. Теперь рассмотрим условные переходы, множественный выбор по условию и циклы.

Ранее был приведен пример описания простого мультиплексора с помощью оператора "?":

```
reg [3:0] c;
always @(a or b or d)
begin
c = d ? (a & b) : (a + b);
end
```

Приведем пример той же самой функции, написанной иначе:

```
reg [3:0] c;
always @(a or b or d) begin
if (d) begin
c = a & b;
end
else begin
c = a + b;
end
end
```

Вместо параметра *"d"* может быть любое выражение. Если значение этого выражения истина (не равно нулю), выполнится первое присвоение "*c = a & b*". Если значение выражения "*d*" ложь (равно нулю), выполнится второе присвоение "*c = a + b*". Если нужно сделать выбор из нескольких вариантов, можно использовать конструкцию *case*. Базовый синтаксис *case* конструкции:

```
case (selector)
option1: <statement>;
option2: <statement>;
default: <if nothing else statement>; //по желанию, но желательно
endcase
```

или

```
wire [1:0] option;
```

```
wire [7:0] a, b, c, d;
reg [7:0] e;
always @(a or b or c or d or option) begin
case (option)
0: e = a;
1: e = b;
2: e = c;
3: e = d;
endcase
```

Поскольку входы – 8-ми битные шины, в результате синтеза получится восемь мультиплексоров четыре-к-одному. Теперь рассмотрим циклы. На языке Verilog цикл скорее описывает сколько экземпляров логических функций должно быть реализовано аппаратно. Цикл должен иметь ограниченное число итераций, которое можно однозначно определить на этапе синтеза. Рассмотрим простой пример – нужно определить номер самого старшего ненулевого бита вектора (шины):

```
module find_high_bit(input wire [7:0]in_data, output reg [2:0]high_bit, output reg valid);
integer i;
always @(in_data)
begin
//определим, есть ли в шине единицы
valid = |in_data;
//присвоим хоть что нибудь
high_bit = 0;
for(i=0; i<8; i=i+1)
begin
if(in_data[i])
begin
// запомним номер бита с единицей в шине
high_bit = i;
end
end
end
endmodule
```

4. Триггеры

Простой триггер (flip-flop)

Это просто регистр или триггер - он запоминает входные данные со входа *d* и подает их на выход *q*. Запоминание происходит в момент, когда сигнал тактовой частоты *clk* переходит из нуля в единицу (то есть в момент фронта сигнала *clk*). На языке описания аппаратуры Verilog это выглядит вот так:

```
reg q;
always @(posedge clk)
 q <= d;
```

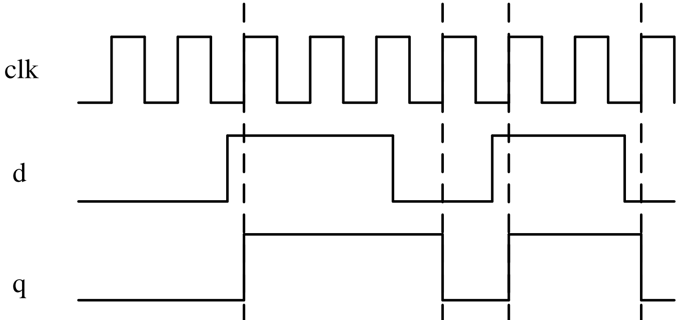


Рисунок 4.1 – Временная диаграмма работы простого триггера

Триггер чувствительный к отрицательному фронту сигнала тактовой частоты

Триггер или регистр может быть активным не только по фронту (*posedge*), но и по спаду сигнала тактовой частоты – на языке Verilog это поведение описывается командой *negedge*:

```
reg q;
always @(negedge clk)
q <= d;
```

Рисунок 4.2 – Временная диаграмма работы простого триггера, чувствительного к отрицательному фронту сигнала тактовой частоты

Триггер с асинхронным сбросом

Асинхронный, значит не связанный с тактовой частотой. Асинхронные сброс или установки применяются в основном для инициализации триггеров или регистров схемы в исходное состояние перед началом работы. Сигнал reset приводит триггер в нулевое состояние.

```
reg q;
always @(posedge clk or posodge reset)
if (reset)
q <= 1'b0;
else
q <= d;
```

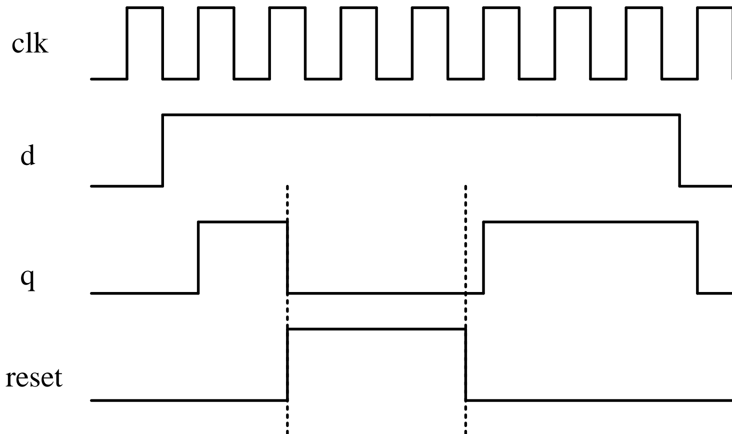


Рисунок 4.3 – Временная диаграмма работы простого триггера с асинхронным сбросом

Триггер с синхронным сбросом

Триггер с синхронном сбросом реализуется аналогично триггеру с асинхронным сбросом, отличие лишь в том, что в список чувствительности always не включен *posedge reset*. Синхронный сброс осуществляется только тогда, когда на триггер поступает положительный фронт сигнала *clk*.

```
reg q;
always @(posedge clk)
if(reset)
q <= 1'b0;
else
```

```
q <= d;
```

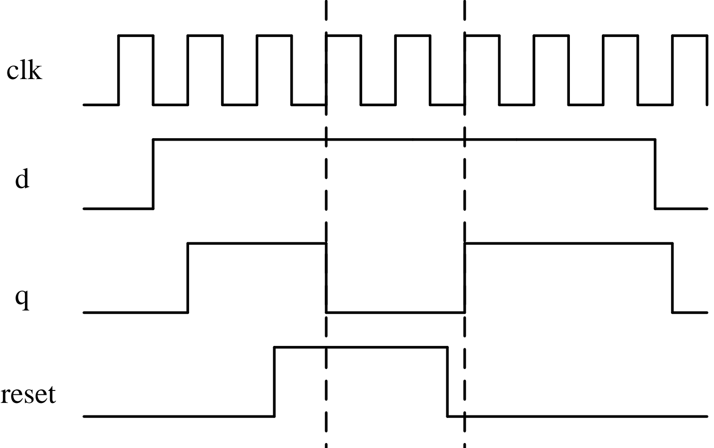


Рисунок 4.4 – Временная диаграмма работы простого триггера с синхронным сбросом

Триггер с входом разрешения на запись

Запись в регистр происходит только в те фронты сигнала тактовой частоты, во время которых сигнал enable установлен в 1.

```
reg q;
always @(posedge clk)
if (enable)
q <= d;
```

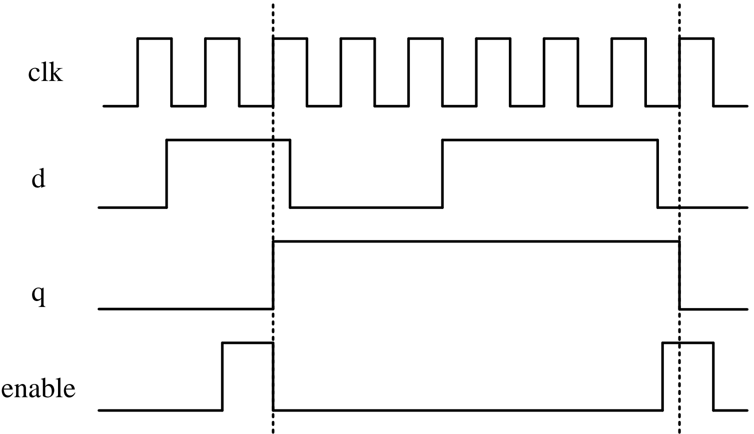


Рисунок 4.4 – Временная диаграмма работы простого триггера с входом разрешения на запись

5. Мультиплексор, демультиплексор, дешифратор, счетчик

Мультиплексор

Мультиплексор – устройство, имеющее несколько сигнальных входов, один или более управляющих входов и один выход. Мультиплексор позволяет передать сигнал с одного из входов на выход, при этом выбор желаемого входа осуществляется подачей соответствующей комбинации управляющих сигналов. Самый простой способ реализации мультиплексора на языке Verilog – использование конструкции *if-else*. Пример мультиплексора с двумя сигнальными входами, реализованного на Verilog, представлен ниже. Схематичная реализация данного кода представлена на рисунке 5.1. На временной диаграмме пунктиром выделены участки, когда *q = a* (при *sel = 1*)

```
reg q;
always @*
if (sel)
q = a;
else
q = b;
```

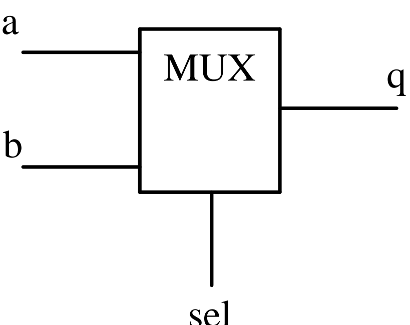


Рисунок 5.1 – Схема мультиплексора

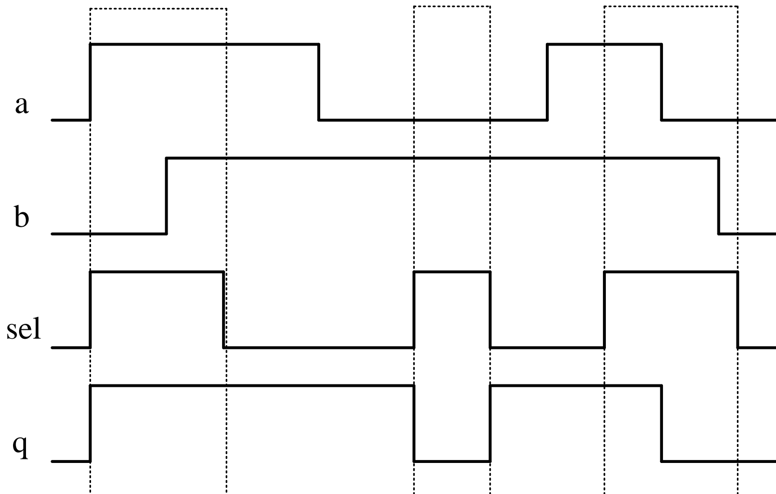


Рисунок 5.2 – Временная диаграмма, демонстрирующая работу мультиплексора

Удобнее реализовывать мультиплексор с использованием оператора *case*. Пример реализации подобного мультиплексора с помощью оператора *case* приведен ниже. Схематическое представление и временная диаграмма при этом остаются неизменными.

```
reg q;
always @*
case (sel)
1 : q = a;
0 : q = b;
endcase
```

Демультиплексор

Демультиплексор – устройство, имеющее один сигнальный вход и несколько сигнальных выходов. Демультиплексор выполняет функцию обратную мультиплексору – подключает входной сигнал к нужному выходному с помощью управляющего сигнала. Существует несколько простых способов описать мультиплексор. Приведем самый простой и понятный пример.

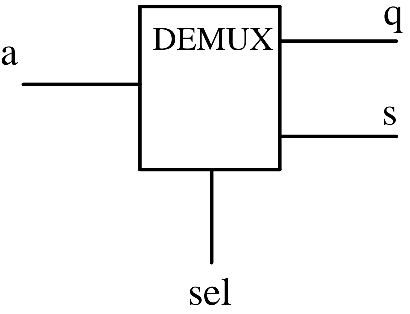


Рисунок 5.3 – Схема демультиплексора

Пример. Однобитный сигнал *signal* должен быть подан на один из нужных выходов. 2х битный сигнал управления *addr* определяет номер выхода.

```
module demux (
input wire signal,
input wire [1:0] addr,
output reg [3:0] out);
always @*
begin
	case (addr)
```

```
            2’d0: out = { 3’b000, signal       };
            2’d1: out = { 2’b00, signal, 1’b0};
            2’d2: out = { 1’b0, signal, 2’b00};
            2’d3: out = {        signal, 3’b000};
            endcase
    end
    endmodule
```

Дешифратор

Дешифратор – устройство, преобразующее двоичный код в любой другой. Самый простой пример дешифратора – управление семисегментным индикатором (рис. 5.4). На вход устройства подается 4х битное число, которое определяет отображаемый на индикаторе символ. К 7-ми входам устройства подключается по отдельности все семь сегментов индикатора.

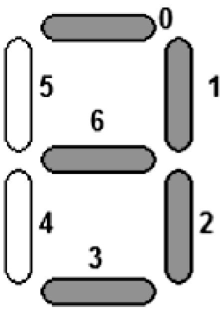


Рисунок 5.4 – Семисегментный индикатор

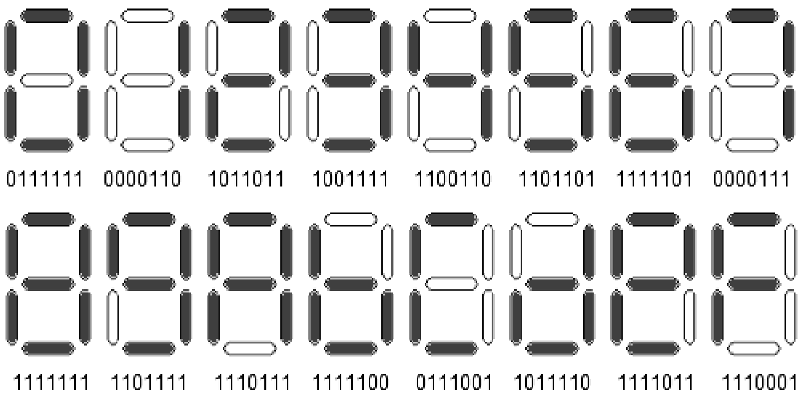


Рисунок 5.5 – Возможные отображаемые символы и их двоичные коды

```
module indicator(
input wire [3:0] code,
output reg [6:0] segments
```

```
);

always @*
begin
case (code)
4’d0: segments = 7’b0111111;
4’d1: segments = 7’b0000110;
4’d2: segments = 7’b1011011;
4’d3: segments = 7’b1001111;
4’d4: segments = 7’b1100110;
4’d5: segments = 7’b1101101;
4’d6: segments = 7’b1111101;
4’d7: segments = 7’b1111101;
4’d8: segments = 7’b1111111;
4’d9: segments = 7’b1101111;
4’d10: segments = 7’b1110111;
4’d11: segments = 7’b1111100;
4’d12: segments = 7’b0111001;
4’d13: segments = 7’b1011110;
4’d14: segments = 7’b1111011;
4’d15: segments = 7’b1110001;
endcase
end
endmodule
```

6. Сдвиговые регистры, счетчик

Сдвиговый регистр

Сдвиговый регистр представляет собой разновидность регистра, для которого по фронту тактового сигнала происходит сдвиг содержимого на один или несколько разрядов какую-либо сторону. Сдвиговый регистр можно представить следующим текстом на Verilog:

```
module shift_reg(
input clk,
```

```
input d_in,
input ce,
output [15:0] q
);
reg [15:0] data;
always @(posedge clk)
begin
    if (ce) data <= {data[15:1], d_in};
end
assign q = data;
endmodule
```

В примере производится сдвиг содержимого внутреннего регистра *data* на один разряд влево. В младший разряд при этом помещается значение, подающееся на вход *d_in*, а старший разряд теряется. Работа сдвигового регистра управляется входом *ce*, низкий уровень на котором запрещает сдвиг.

Регистр, сдвигающий свое содержимое на регулируемое число разрядов, называется регистром барабанного сдвига. Это более сложное для реализации устройство, чем обычный сдвиговый регистр, поскольку требует дополнительного мультиплексора, из-за чего занимаемый в ПЛИС объем существенно возрастает. Регистр барабанного сдвига применяется, например, для сложения и вычитания чисел с плавающей точкой. Пример описания устройства барабанного сдвига:

```
reg [3:0] q;
always @*
case (sel)
    2’b00 : q = d;
    2’b00 : q = d << 1;
    2’b00 : q = d << 2;
    default q = d << 3;
endcase
```

Счетчик

Счетчик выполняет последовательное увеличение или уменьшение своего выходного значения по каждому тактовому импульсу. Простейший вариант счетчика:

```
reg [7:0] cnt;
always @(posedge clk)
```

cnt <= cnt + 1;

Посколько для хранения значения счетчика выбрано 8 разрядов, счетчик будет осуществлять циклическое приращение своего значения от 0 до 255, после чего операция 255+1 опять приведет значение cnt в нулевое состояние.

7. Верификация проектов с помощью Modelsim

Процесс проектирования начинается с того, что уточняется задание на проект и вырабатываются проектные спецификации. Далее идет разработка файлов проекта. И одновременно с этим начинается верификация, то есть проверка проекта. Весь процесс верификации сводится к тому, что разработчик сравнивает то, что он должен получить от изделия, с тем, что он наблюдает в поведении проверяемого проекта. На основании сравнения разработчик определяет, соответствует ли ожидаемое поведение проекта полученному.

Существует несколько способов провести проверку проекта: с помощью встроенного симулятора Quartus, инструмента Signal Tap, с помощью различных сторонних симуляторов. Пакет программных средств ModelSim™ корпорации Model Technology (одного из подразделений компании Mentor Graphics) в настоящее время является самой распространенной системой HDL-моделирования

Для симуляции модуля в ModelSim необходимо написать программу – «тестбенч»(test bench – испытательный стенд), в которой описываются действия, осуществляемые с модулем, а именно подача на входные порты каких-либо воздействий (в том числе сложных сигналов, сгенерированных в сторонних программах), и просмотр выходных шин и регистров, а так же операции над ними, например экспорт в сторонние пакеты. Тестбенч может писаться на том же языке программирования, что и сам тестируемый модуль.

Testbench

Модуль тестбенч может быть написан на языке Verilog, как и основной модуль. Нам необходимо подать на входные шины какие-либо значения и посмотреть при них значения выходной шины.

Каждый тестбенч начинается с так называемой директивы шкалы времени:

`timescale 10 ns / 10 ns

Первое число означает шаг, второе-точность симуляции. Далее следует название модуля тестбенча:

```
module mult_tb;
```

Обратите внимание на отсутствие входов и выходов. Следующий шаг-объявление внутренних переменных модуля:

```
reg signed [15:0] a;
reg signed [15:0] b;
wire signed [31:0] c;
```

Их разрядность совпадает с разрядностью портов, так же они имеют знаковый тип. Входные порты исследуемого модуля можно соединять как с регистрами, так и с шинами, выходные порты должны быть соединены только с шинами!

Приступим к присвоению значений. Как и в обычных модулях, написанных на языке Verilog, все операции с регистрами производятся в блоках always или блоках initial. Но в отличии от синтезируемых модулей, always блоки в тестбенчах могут не содержать списка чувствительности, а работать в виде бесконечного цикла: когда компилятор выполняет последнюю операцию в блоке always, он возвращается к первой. Примеры:

С самого начала симуляции регистру a присваивается значение 100, регистру b значение 400, которые в дальнейшем не изменяются:

```
initial
begin
a=100;
b=400;
end
```

Обозначение # означает задержку в симуляции на количество тактов, равных числу после символа решетки. Один такт происходит в течение времени, указанного в директиве в первой строчке тестбенча. В начальный момент времени значение регистра a равно «0», спустя три такта симуляции оно переходит в «100», которое сохраняет до конца симуляции:

```
initial
begin
a=0;
#3
a=100;
end
```

На рис. 7.1 изображены временные диаграммы регистра a.

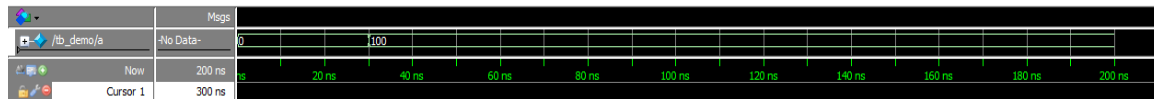


Рисунок 7.1 - Временные диаграммы регистра а

Регистр а равен «0», спустя один такт он переходит в значение «1», через такт команды в always блоке заканчиваются, компилятор возвращается назад и а снова принимает значение «0», таким образом реализован генератор тактовых импульсов:

```
always
begin
a=0;
#1;
a=1;
#1;
end
```

На рис. 7.2 изображены временные диаграммы регистра а.

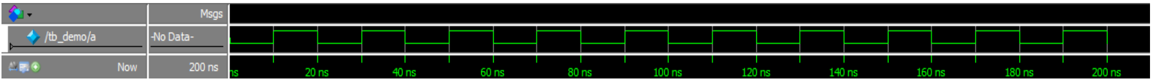


Рисунок 7.2 - Временные диаграммы регистра а

Тестбенч позволяет подавать на вход тестируемых модулей сигналы сложной формы. Разберем следующий код:

```
initial
 h_signal_in_tb = $fopen("C:/ MatlabFiles/m_test.txt","r");
```

В блоке initial открывается файл m_test.txt, расположенный на диске «С» в папке «MatlabFiles», в режиме "r"-чтение. Информацию о файле содержит структура h_signal_in_tb, которую в начале тестбенча следует объявить как переменную типа integer.

```
always @ (posedge clk)
b_data_read<=$fscanf(h_signal_in_tb,"%d",input_signal);
```

Каждый положительный фронт тактовой частоты из структуры h_signal_in_tb считывается отсчет сигнала, и записывается в регистр input_signal, "%d", означает, что отсчеты в файле записаны в десятичном виде. Структура b_data_read так же задана как переменная типа integer.

```
always @ (posedge clk)
b_data_read<=$fscanf(h_signal_in_tb,"%d",input_signal);
```

На рис. 7.3 изображена синусоида с частотой 15МГц, при частоте дискретизации 100МГц, сгенерированная в пакете Matlab, записанная в текстовый файл и загруженная в моделсим.

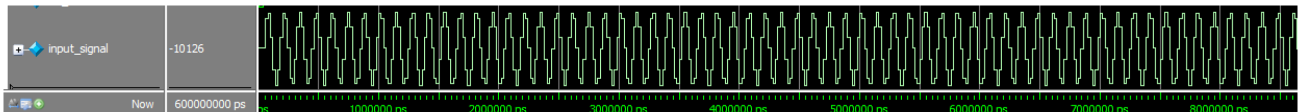


Рисунок 7.3 - Временные диаграммы сигнала, загруженного из текстового файла

Когда заданы все входные воздействия, следует подать их на исследуемый модуль:

mult mult_in_tb(

.a(a),

.b(b),

.c(c)

);

В конце тестбенча, как и в любом модуле Verilog пишется слово *endmodule*.

Modelsim

Для того чтобы начать моделирование в Modelsim, необходимо создать новый проект. Для этого в главном меню выбираем File→New→Project. Появится окно Create Project. Для создания проекта выполним следующие шаги: нам необходимо указать имя проекта и путь к файлам проекта, название рабочей библиотеки:

- В окне Project Name укажем имя проекта (My_testbench).
- В окне Project Location укажем рабочую директорию (E:/My_testbench).
- В окне Default Library Name оставим имя work.
- Нажмем OK (рис. 7.4).

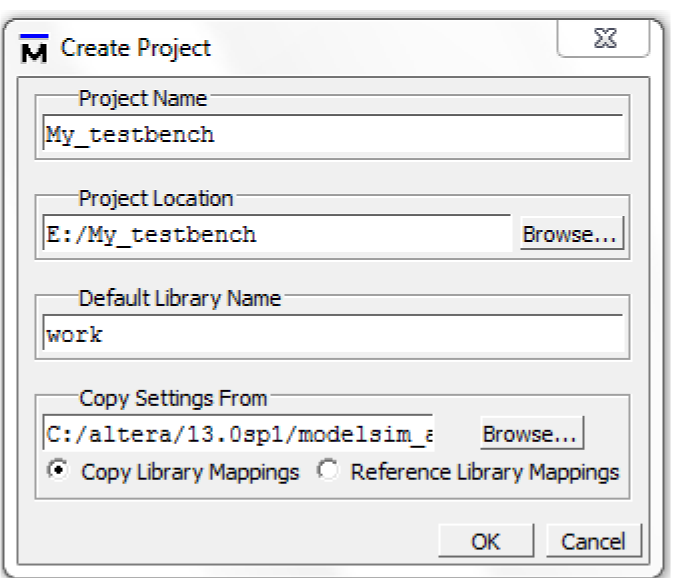


Рисунок 7.4 -Задание имени проекта, пути к файлам проекта и название рабочей библиотеки

Если пользователь программы захочет повторно создать проект с тем же именем, в той же рабочей директории, то программа выдаст предупреждение о том, что проект с

этим именем уже существует. Если мы не хотим менять имя проекта, и старый проект нам не нужен, то в таком случае в этом окне необходимо указать «Да» (рис. 7.5).

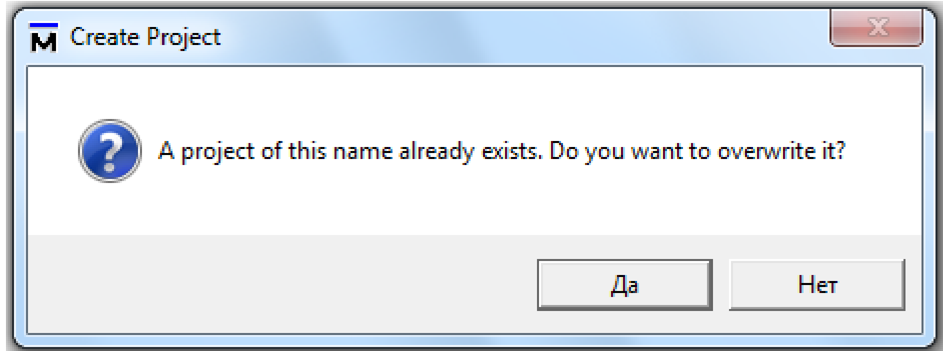


Рисунок 7.5 - Подтверждение имени проекта

Если же файлов для проекта еще нет, то в открывшемся окне выбираем пункт Create New File. Откроется окно для ввода текстовых файлов. И далее создаем файлы проекта один за другим. Если файлы для данного проекта уже имеются, то в открывшемся окне Add Items to the Project выбираем Add Existing File (рис. 7.6).

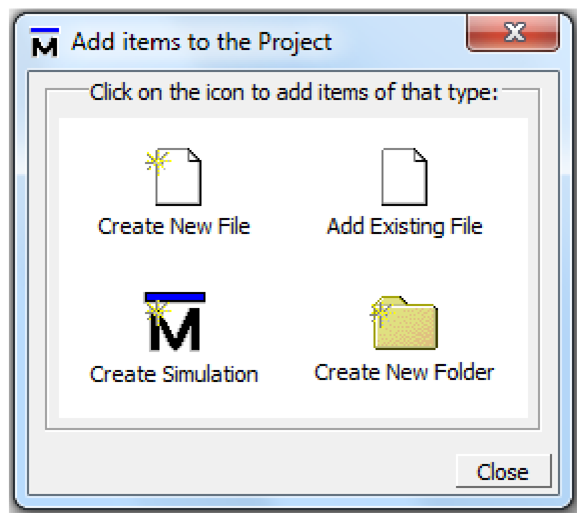


Рисунок 7.6 - Добавление файлов в проект в окне Add Items to the Project

После выбора Add Exiting Fileпоявится окно Add file to Project(рис. 7).

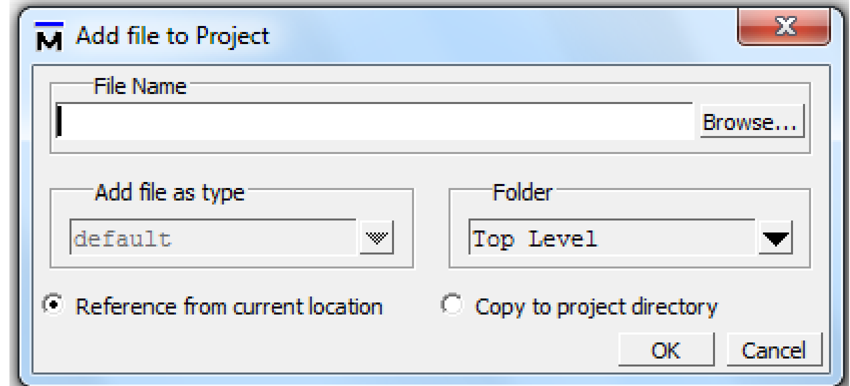


Рисунок 7.7- Окно для выбора добавляемого в проект файла

Добавляем файлы в проект либо по очереди, либо это можно сделать одновременно, если отметить сразу несколько файлов. В окне Workspace поочередно появляются имена добавляемых файлов проекта (рис. 7.8). Файлы будут расположены в том порядке, в котором они были добавлены в проект. Но для работы компилятора необходимо указать иерархию файлов. Это можно сделать либо вручную, либо автоматически.

Чтобы установить порядок компиляции файлов вручную, необходимо учесть, что компиляция осуществляется согласно иерархии описания проекта, и сначала на компиляцию должны поступать следующие файлы:

- описания модулей нижнего уровня;
- затем описания модулей верхнего уровня, содержащие установленные модули, которые описаны в модулях нижнего уровня, и т. д.;
- последний файл — это тестбенч.

Выбираем Compile→Compile Order, далее появится окно Compile Order (рис. 7.8). Далее можно выставить порядок компиляции файлов.

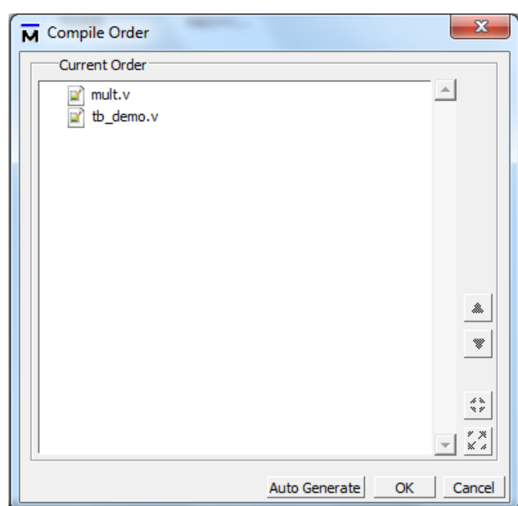


Рисунок 7.8 - окно Compile Order

В дальнейшем можно компилировать проект нажатием на кнопку Compile all (рис. 7.9).

Рисунок 7.9 – Расположение кнопок компиляции и симуляции

Если файлы не содержат ошибок, то компилятор выдаст в нижнем окне Transcript соответствующие сообщения, например:

 # Compile of FFTModule.v was successful.

 # Compile of WinFcn.v was successful.

 # Compile of SA_Assembly_1.v was successful.

 # Compile of tb_new.v was successful.

 # 4 compiles, 0 failed with no errors.

В окне Workspace напротив имен файлов знак вопроса после компиляции заменяется на «галочку».

Если программа содержит ошибку, то в нижнем окне Transcript появится сообщение об этом. Двойной щелчок на сообщении об ошибке выдает строку Verilog - текста, в которой может быть ошибка (однако на самом деле ошибка может быть совершенно в другой строке — синтаксис Verilog сложный, и так же, как и в компиляторах языка Си, ошибка может быть в строке перед указанной строкой).

Выдача текста Verilog-файла для редактирования осуществляется после двойного щелчка по имени соответствующего файла.

Следующим шагом будет установка верхнего модуля проекта для симуляции в окне Start Simulation. Выбрать в главном меню Simulate→Start Simulation, или нажать на соответствующую кнопку на панели инструментов (рис. 7.9) появится окно, изображенное на рис. 7.10.

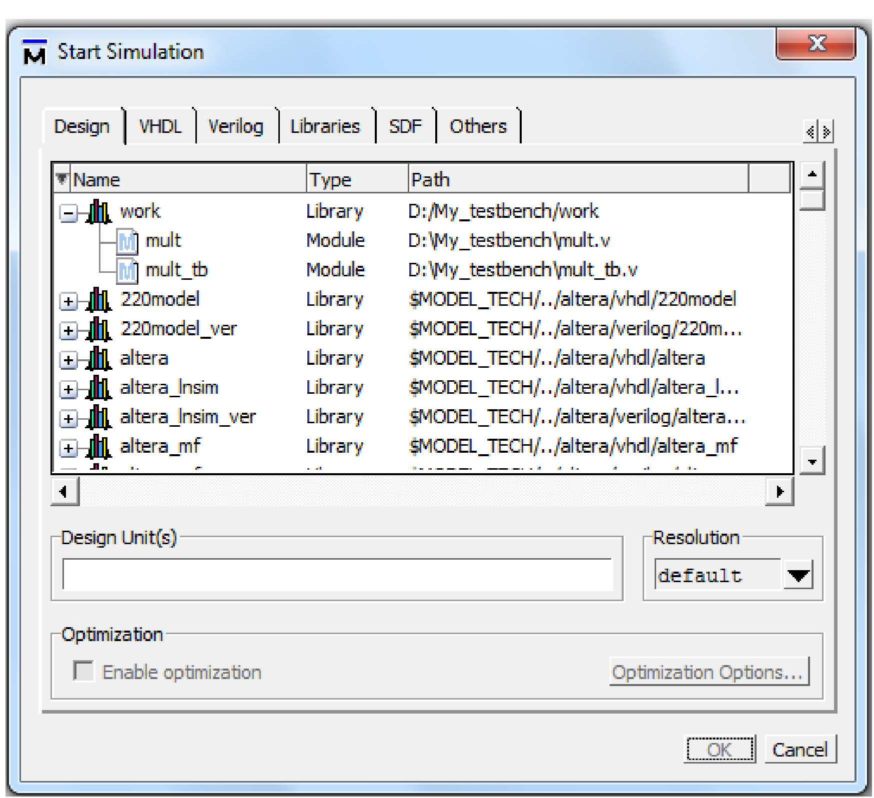


Рисунок 7.10 - Окно Start Simulation для указания верхнего модуля проекта

В окне Start Simulation необходимо выполнить следующие установки: раскрыть библиотеку work, выбрать (двумя щелчками) имя верхнего модуля проекта (mult_tb), остальное можно оставить по умолчанию. Нажать OK. После выбора верхнего модуля

проекта в окне Workspace появится закладка sim, и внешний вид программы будет такой, как на рис. 7.11.

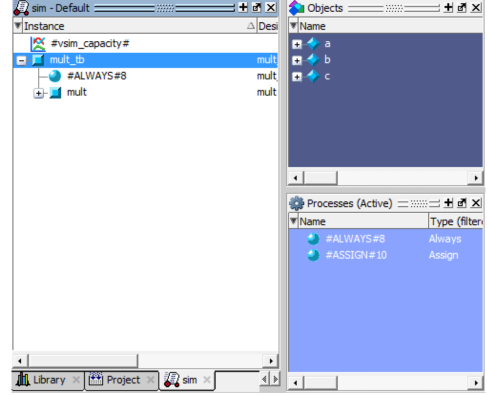


Рисунок 7.11 - Окно Start Simulation для указания верхнего модуля проекта

Следующим шагом будет открытие окна Wave. Для этого в окне Workspace, в закладке sim, выделим строку с названием модуля верхнего уровня test_statmach. Теперь кликнем правой клавишей мышки по этой строке и в открывшемся меню выберем строку Add→Add to Wave. При этом откроется окно Wave и в него будут загружены названия сигналов модуля test_statmach. Так же можно добавить временные диаграммы из модулей нижестоящего уровня. Можно добавлять каждый сигнал модуля по отдельности, из поля Object.

Далее можно начинать симуляцию, для этого следует задать время симуляции и нажать кнопку Run (рис. 7.12)

Рисунок 7.12 – Старт симуляции

Следует выбирать такое время симуляции, чтобы на его протяжении были видны все значимые изменения сигналов, но не слишком большое, чтобы симуляция не занимала много времени.

На рис. 7.13 приведены временные диаграммы умножителя, входные порты имеют размерность 16 бит, выходной-32 бита.

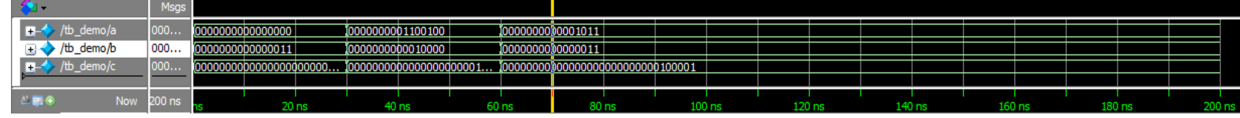


Рисунок 7.13 – Временные диаграммы умножителя

В данном примере числа представлены в двоичном виде, чтобы сделать их десятичными, необходимо правой кнопкой мыши нажать на название регистра и выбрать Radix/Decimal, рис. 7.14.

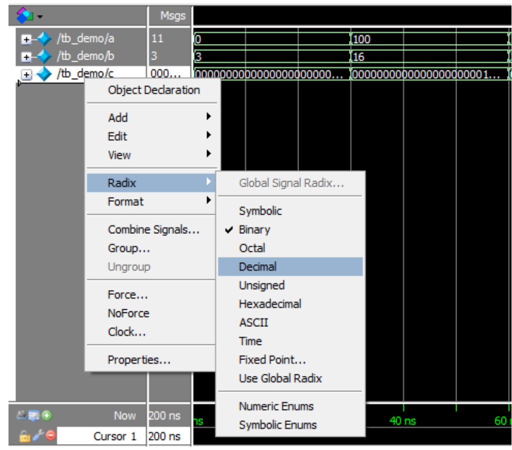


Рисунок 7.14 – Изменение системы счисления диаграмм

Диаграммы также можно представлять в аналоговом виде. Для этого необходимо щелкнуть правой кнопкой по названию диаграммы и выбрать Format/Analog (automatic) для автоматической нормировки максимальных значений, или Format/Analog (custom) для ручного выставления интервалов отображения.

Если вносятся какие-либо изменения в файл исследуемого модуля или тестбенча, необходимо сохранить этот файл, откомпилировать (рис. 7.8), нажать кнопку Restart на панели инструментов, левее времени симуляции (рис. 7.12), и нажать на кнопку Run (рис. 7.12).

Временные диаграммы могут отображаться цифровом или аналоговом виде, так же регистры могут иметь неопределенное значение, которое отображается красной прямой на диаграмме, а шины – состояние высокого импеданса (разрыв), если они ни к чему не подключены, которое отображается в виде синей прямой на диаграмме, рис. 7.15.

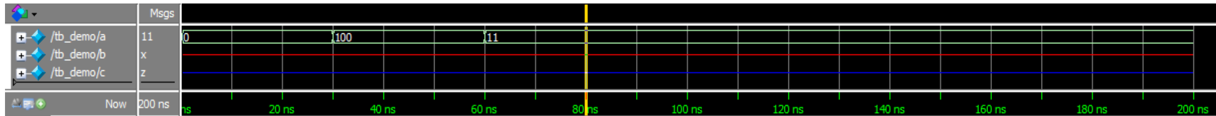


Рисунок 7.15 – Неопределенное состояние и высокий импеданс

Важной особенностью симуляции в Modelsim является обязательность присвоение значений всех регистров исследуемого модуля в начале симуляции. Ниже приведен листинг модуля, в котором реализован накопитель значений входа a, по переднему фронту тактового импульса *clk* с асинхронным сбросом.

```
module summ (clk,reset,a,c);
input clk,reset;
input [5:0] a;
output reg [10:0] c;
always @(posedge clk or posedge reset)
    if (reset)
```

```
			c<=0;
		else
			c<=c+a;
	endmodule
```

Листинг тестбенча:

```
	`timescale 10 ns / 10 ns
	module tb_summ;
	reg reset,clk;
	reg [5:0] a;
	wire [10:0] c;
	always
		begin
			clk=0;
			#1;
			clk=1;
			#1;
		end
	initial
		begin
			reset=0;
			#2;
			reset<=1;
			#1;
			reset <=0;
		end
	initial
		begin
			a=3;
		#2;
			a=8;
		#2;
			a=55;
		#2;
			a=3;
		#2;
```

```
                a=12;
        #2;
        end
        summ   summ_in_tb(
                        .clk(clk),
                        .reset(reset),
                        .a(a),
                        .c(c)
                        );
endmodule
```

Диаграммы исследуемого модуля приведены на рис. 7.16.

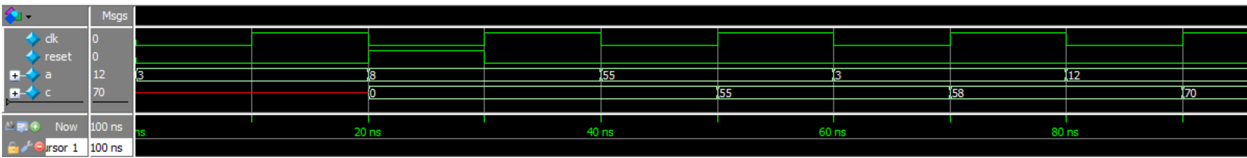


Рисунок 7.16 – Временные диаграммы накопителя значений

Регистр «*c*» в начальный момент времени не определен, по положительному фронту *clk* к нему добавляется значение регистра «*a*», равное «3», в результате неопределенное значение не изменилось. По положительному фронту сигнала «reset» регистр «c» принимает значение «0», после чего начинается накапливание значений со входа. [1]

Задания на самостоятельную работу

1) Создайте модуль восьмиразрядного счетчика тактовых импульсов (clk) с асинхронным сбросом (reset). Напишите к этому модулю тестбенч, просимулируйте проект с помощью ModelSim.
2) Создайте модуль делителя тактовой частоты на 4. Напишите к этому модулю тестбенч, просимулируйте проект с помощью ModelSim
3) Создайте модуль синхронного сумматора. Напишите к этому модулю тестбенч, просимулируйте проект с помощью ModelSim
4) Создайте модуль вычисления суммы квадратов чисел, поступаемых с двух входов. Напишите тестбенч, просимулируйте проект с помощью ModelSim
5) Создайте модуль трехканального мультиплексора. Напишите к этому модулю тестбенч, просимулируйте проект с помощью ModelSim

6) Создайте модуль в котором входной 20-разрядный сигнал будет обрезаться до 15 разрядов. Напишите к этому модулю тестбенч, на вход модуля подайте синусоиду, сгенерированную в пакете SciLab просимулируйте проект с помощью ModelSim.

8. Логический анализатор SignalTap II

В состав комплекта ПО Altera Quartus II входит логический анализатор SignalTap. Этот логический анализатор помогает отлаживать проект FPGA путём пробирования состояний внутренних сигналов без использования внешней обвязки. Все захваченные сигналы данных легко сохраняются в памяти чипа, пока вы не будете готовы для чтения и анализа данных. Обладая достаточными ресурсами внутренней памяти и логических элементов FPGA, вы достаточно просто сможете построить любую схему события (или последовательности событий), по которому данные будут защёлкиваться в памяти встроенного анализатора. SignalTap II - это наиболее экономичный способ получения многофункционального логического анализатора для быстрой и эффективной отладки проектов любой сложности.

Добавление встроенного логического анализатора SignalTap® II в ваш проект

Поскольку встроенный логический анализатор SignalTap II размещается в логике вашего чипа, он должен быть добавлен к вашему проекту FPGA в виде отдельной части проекта. Существует два способа для создания встроенного логического анализатора SignalTap II и добавления его в ваш проект для отладки [3]:

- Создайте **.stp** файл и используйте Редактор SignalTap II для конфигурирования частей логического анализатора;

  или

- Создайте и сконфигурируйте **.stp** файл с помощью менеджера плагинов MegaWizard и разместите его в вашем проекте.

Создание и разрешение файла SignalTap II

Для создания встроенного логического анализатора, используйте существующий **.stp** файл или создайте новый файл. После того, как файл создан или выбран, вам нужно разрешить его использование в проекте [3].

Создание файла SignalTap II

Файл .stp содержит настройки встроенного логического анализатора SignalTap II и полученные данные для наблюдения и анализа. Для создания нового .stp файла выполните следующие шаги:

1. В меню Файл кликните Новый.

2. В диалоговом окне Новый кликните на вкладку Другие файлы и выберите файл встроенного логического анализатора SignalTap II.

3. Кликните ОК.

Для открытия существующего .stp файла, уже ассоциированного с вашим проектом, в меню Инструменты кликните на Логический анализатор SignalTap II. Этот метод может быть использован и для создания нового .stp файла, если нет ассоциированного с проектом .stp файла.

Конфигурирование встроенного логического анализатора SignalTap® II

Файл .stp имеет некоторые опции для конфигурирования элементов логического анализатора. Большинство настроек просты, их можно встретить в обычных внешних логических анализаторах. Другие настройки уникальны для встроенного логического анализатора SignalTap II, поскольку они требуются для конфигурации встроенного логического анализатора. Все настройки дают вам возможность сконфигурировать логический анализатор по вашему желанию, чтобы помочь в отладке вашего проекта [3].

Назначение такта захвата

Назначьте тактовый сигнал для контроля за захватом данных встроенным логическим анализатором SignalTap II. Логический анализатор получает данные на каждом положительном (нарастающем) фронте такта захвата. Логический анализатор не поддерживает получение данных на негативном (спадающем) фронте такта захвата. В качестве такта захвата вы можете использовать любой сигнал в вашем проекте. Однако, для лучших результатов, Altera рекомендует вам использовать глобальную, не вентильную синхронизацию для тестовых сигналов захвата данных. Использование вентильных тактов в качестве такта захвата может дать в результате неожиданные данные, которые не передают в точности поведение вашего проекта. Инструмент статичного анализа Quartus II показывает максимальную частоту такта захвата, на которой вы можете запускать ваш проект. Обратитесь к секции временного анализа в отчѐте компилятора, чтобы найти максимальную частоту такта логического анализатора. Чтобы назначить такт захвата, выполните следующие шаги [3]:

1. В окне логического анализатора SignalTap II кликните на вкладку Установка.

2. На панели Конфигурация сигналов рядом с полем Такт кликните Посмотреть. Откроется диалоговое окно поиска узлов.

3. В списке Фильтры выберите SignalTap II: пост-компоновка или SignalTap II: пре-синтез.

4. В поле Обозначение, введите существующее имя узла, который вы хотите использовать в качестве такта захвата, или найдите узел, используя неполное имя и символы дикой карты.

5. Для начала поиска узлов кликните Список.

6. В списке найденных узлов выберите узел, который представлен глобальным тактовым сигналом проекта.

7. Добавьте выбранное имя узла в Выбранные узлы, кликнув ">" или дважды кликнув на имя узла.

8. Кликните ОК. Узел теперь определён в качестве такта захвата в Редакторе SignalTap II.

Если вы не назначаете такт захвата в редакторе SignalTap II, программа Quartus II автоматически создаѐт тактовый вывод, называемый auto_stp_external_clk. Вы должны сделать назначения для этого вывода независимо от проекта. Проследите, чтобы тактовый сигнал в вашем проекте подводился к такту захвата.

Добавление сигналов в файл SignalTap II

Во время конфигурирования логического анализатора, добавьте сигналы в список узлов в .stp файле, чтобы выбрать, какие сигналы вашего проекта вы хотите наблюдать. Выбранные сигналы используются для определения триггеров. Вы можете назначить следующие два типа сигналов в вашем .stp файле [3]:

■ Пре-синтез – этот сигнал появляется после выработки проекта, но до выполнения любой оптимизации синтеза. Этот набор сигналов может отражать сигналы на Уровне Переходов Регистра (RTL).

■ Пост-компоновка – этот сигнал появляется после оптимизации физического синтеза и размещения и разводки.

Если вы не используете инкрементную компиляцию, добавляйте только сигналы пре-синтеза в ваш .stp файл. Использование пре-синтеза особенно полезно, если вы хотите добавить новый узел после того, как сделали изменения в проекте. Изменения в исходном файле добавляются в Поиск узлов после выполнения Анализа и Выработки. В меню Процессы выберите Старт и кликните Старт анализа и выработки.

Программа Quartus II не ограничивает количество сигналов, доступных для наблюдения в окне временных диаграмм SignalTap II. Однако, количество доступных каналов прямо пропорционально количеству логических элементов (LEs) или адаптивных логических модулей (ALMs) в чипе. Поэтому, это является физическим ограничением количества каналов, доступных для наблюдения. Сигналы, показанные синим текстом – имена узлов пост-компоновки. Сигналы, показанные чѐрным текстом – имена узлов пост-синтеза.

После успешного выполнения Анализа и Выработки, сигналы, показанные красным текстом, - это некорректные сигналы. Только если вы знаете, что эти сигналы корректные, удалите их из .stp файла. Индикатор состояний SignalTap II показывает, когда существует некорректное имя узла в .stp файле.

Главное, чтобы сигналы могли быть отведены и подключены к элементам SignalTap II, если доступны ресурсы разводки (строки или столбцы внутренних соединений). Например, сигналы, которые существуют на элементе I/O (IOE) не могут быть прямо отведены, поскольку нет прямых ресурсов разводки от сигнала на IOE до логического элемента ядра. Для входных выводов, вы можете отводить сигналы, которые ведут к блоку логического массива (LAB) от IOE, или для выходных выводов, вы можете отводить сигналы, которые ведут от LAB до IOE.

Когда вы добавляете сигналы пре-синтеза, все соединения, сделанные для встроенного логического анализатора SignalTap II делаются приоритетными для синтеза. Выделяются ресурсы логики и разводки во время перекомпиляции, чтобы сделать соединения, т.о. изменяется ваш файл проекта. По существу, имена сигналов пре-синтеза, идущие к и от IOE, совпадают с именами сигналов ассоциированных с выводом.

В случае сигналов пост-компоновки, соединения, которые вы делаете для встроенного логического анализатора SignalTap II, - это имена сигналов существующих атомов в вашем списке соединений пост-компоновки. Соединение может состояться только, если сигналы являются частью существующих в списке соединений пост-компоновки, а также существуют ресурсы разводки для соединения интересующего сигнала с встроенным логическим анализатором SignalTap II. В случае выходных сигналов пост-компоновки, отводите COMBOUT или REGOUT сигналы, ведущие от IOE блока. Для входных сигналов пост-компоновки, сигналы, идущие к ядру логики, совпадают с именами сигналов, назначенных выводу.

Пример добавление анализатора SignalTap II в проект.

Рассмотрим пример использования SignalTap II в проекте. В качестве примера был взят синхронный 8-разрядный счетчик со входом разрешения на запись код которого выглядит следующим образом:

```
module counter_signaltap(
input clk, input en,
input reset,
output reg [7:0] LEDS);
always @(posedge clk)
begin
if(reset)
LEDS <= 0;
else if(en)
LEDS <= LEDS + 1'b1;
end
endmodule
```

В начале, как показано на рис. 8.1, нужно назначить сигнал захвата из списка SignalTap II: пост-компоновка. К

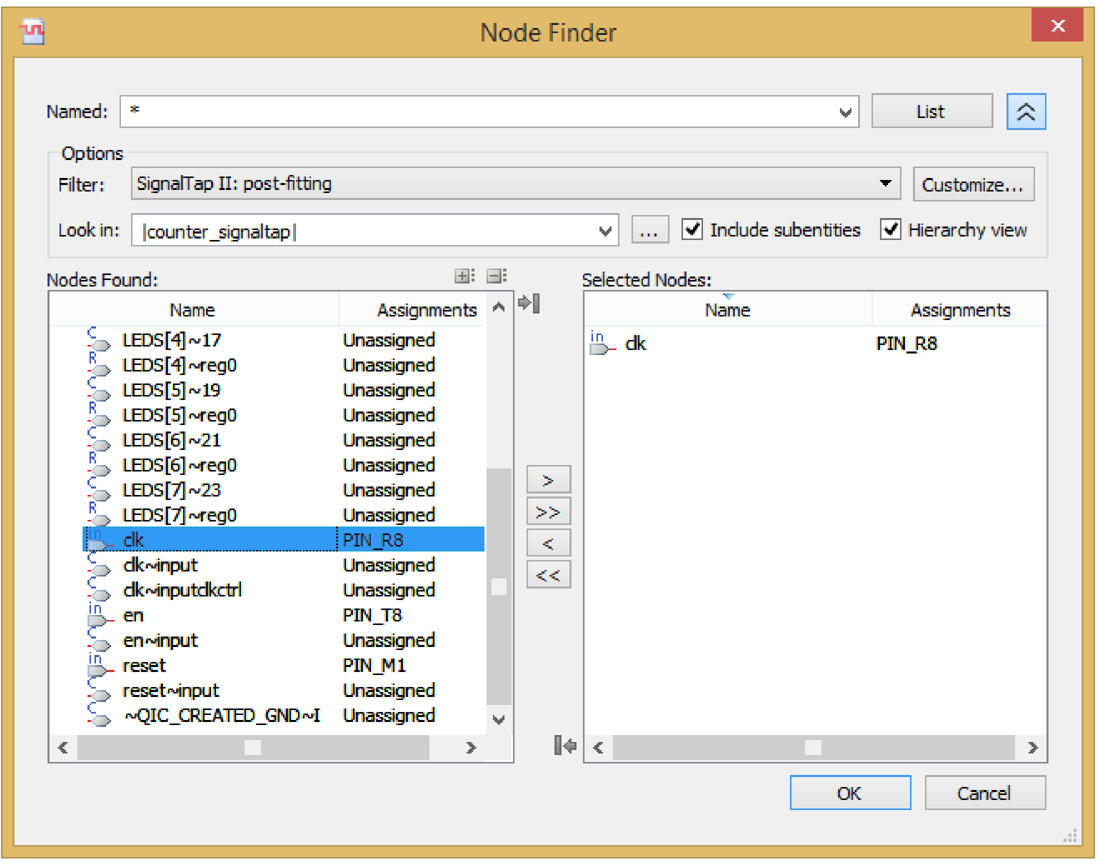


Рисунок 8.1– Выбор сигнала захвата

После этого нужно выбрать необходимое количество данных для анализа. Возьмем к качестве примера 256 значений.

Сделаем так, чтобы данные снимались после триггера на сигнале разрешения. Для этого ставим галочку на Trigger in и указываем из списка SignalTap II: пост-компоновка, как показано на рис.8.2. Теперь указываем условие (pattern) при котором триггер сработает. Так как шина «en» в исходном положении равна нулю, то добавим условие Rising edge.

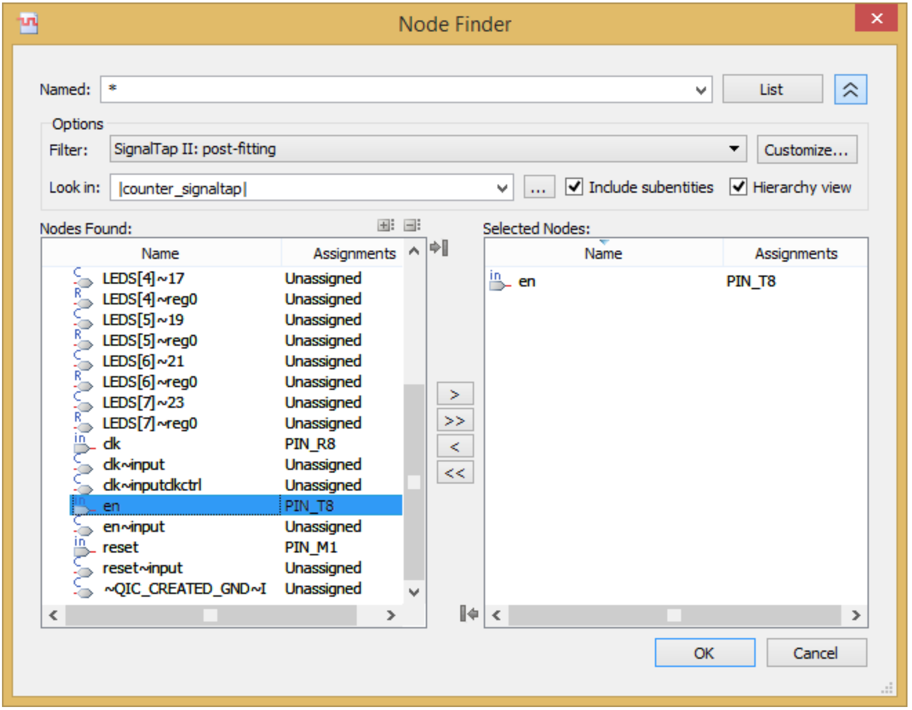


Рисунок 8.2 – Выбор триггера

Дважды щелкнув на поле уставновки добавляем сигналы, которые мы хотим увидеть из того же самого списка SignalTap II: пост-компоновка.

После завершения установки необходимых параметров необходимо сделать следующиме шаги для того, чтобы увидеть сигналы:

1) Компилируем проект в Quartus II (не забудьте назначить все пины, соответствующие входу и выходы)

2) Снова открываем тот файл .stp и прошиваем плату в правом верхнем углу.

3) Затем нажимаем Run Analysis. И если вы объявили триггер, то он будет ждать до тех пор пока вы не подадите сигнал на триггер

4) Результат показан на рис. 8.3. Если необходимо увеличить количество точек, то нужно указать соответствующее новое значение и повторить шаги 1-3.

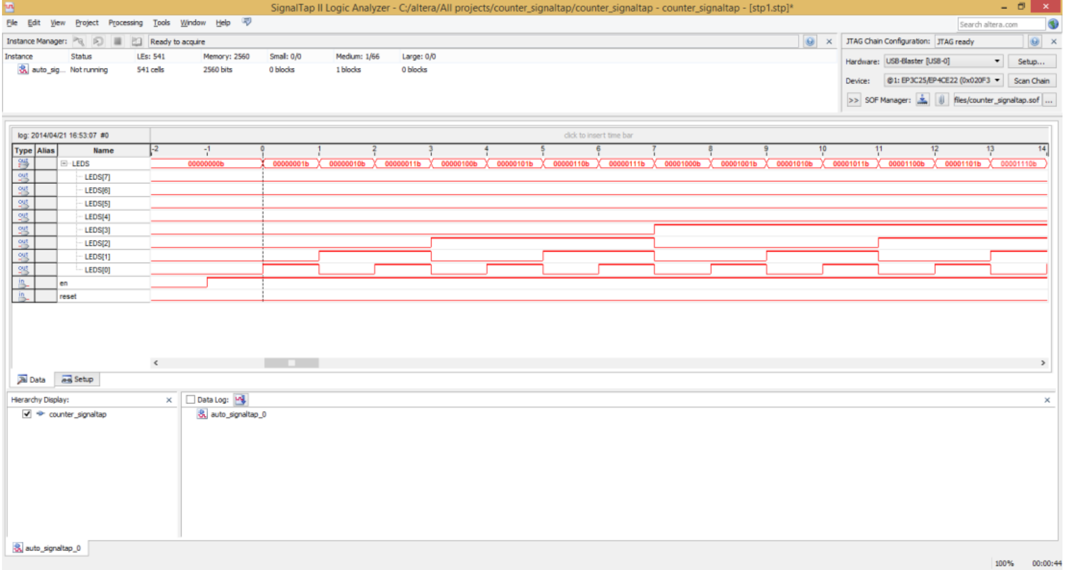


Рисунок 8.3 – Отображение сигналов в SignalTap

9. MegaWizard

Часто возникает ситуация, когда в проект необходимо добавить сложный в реализации, но типовой модуль. Для этого в Quartus присутствует библиотека готовых решений (так называемых мегафункций и IP ядер) – Altera MegaWizard Plug-In Manager. Мегафункции из стандартной библиотеки Quartus II разделяются на два типа. Первый тип - мегафункции, разработанные фирмой Altera. Их название в библиотеке начинается с “Alt”. Второй тип составляют мегафункции, созданные сторонними производителями по стандарту библиотеки параметризируемых модулей. Их название начинается с “LPM” (Library of parameterized modules). Модули MegaWizard являются легко конфигурируемыми, гибкими в использовании. Они оптимизированы для реализации в кристаллах фирмы Altera. Их применение в проекте пользователя позволит уменьшить трудоёмкость и ускорить выполнение проекта.

Чтобы вызвать менеджер мегафункций, необходимо выбрать Tools/ MegaWizard Plug-In Manager. Появляется окно, изображенное на рис. 9.1.

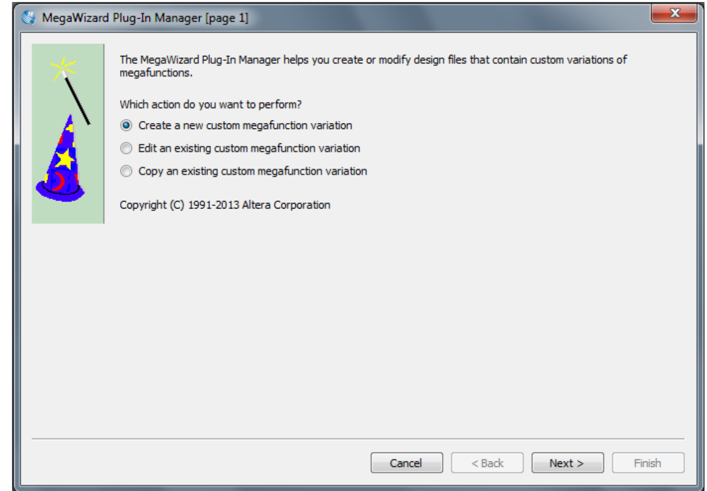


Рисунок 9.1 – MegaWizard Plug-In Manager

Нам предложено выбрать из создания новой функции, редактирования или копирования существующей. Выбираем создание новой мегафункции, появляется их список (рис. 9.2)

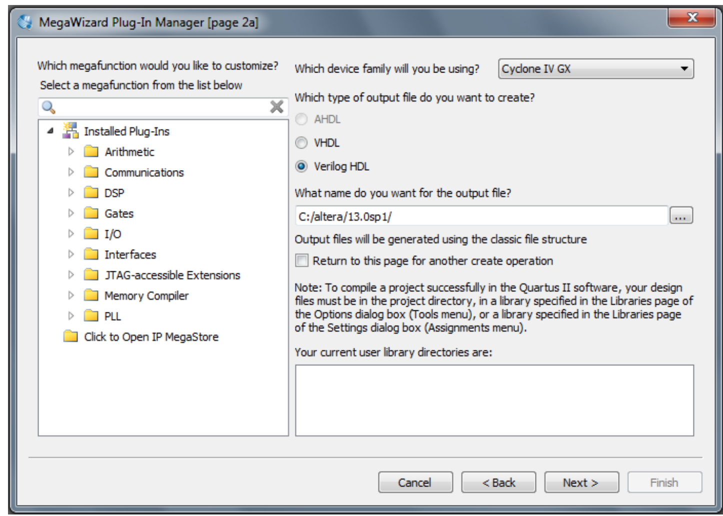


Рисунок 9.2 – Выбор мегафункции

На этом этапе выбирается семейство ПЛИС, к которому относится ваша микросхема. Следует иметь в виду, что модули не являются универсальными и работают не на всех кристаллах. Далее выбирается нужная функция, язык, на котором будет сгенерирован ее код, папка и название файла верхнего уровня. В качестве примера выберем функцию умножителя – ALTFP_MULT. Жмем Next, появляется окно с выбором параметров модуля, рис. 9.3.

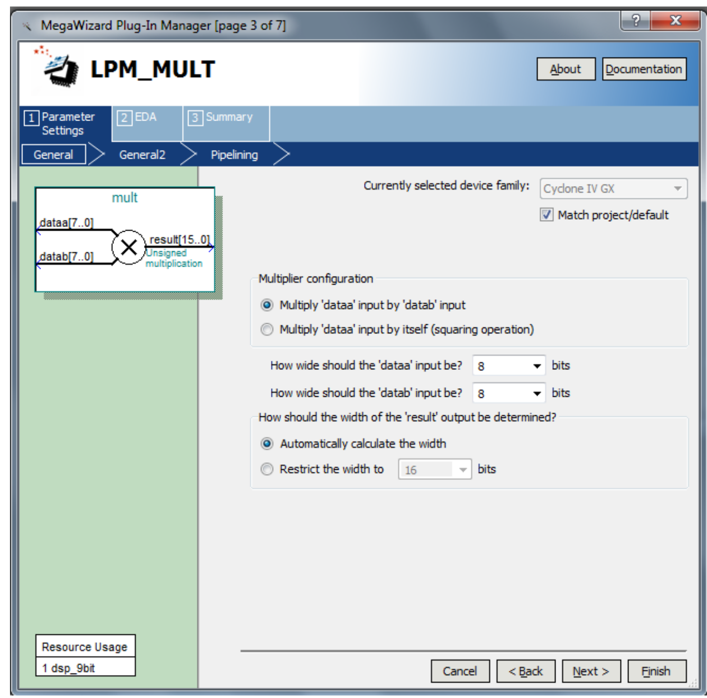


Рисунок 9.3 – Выбор параметров мегафункции, шаг 1

На первом шаге выбирается размерность перемножаемых чисел или возведения числа в квадрат, можно в ручную ввести разрядность выходного числа. Слева внизу в поле Resource Usage показаны используемые ресурсы. После выбора всех параметров жмем Next, появляется окно изображенное на рис. 9.4

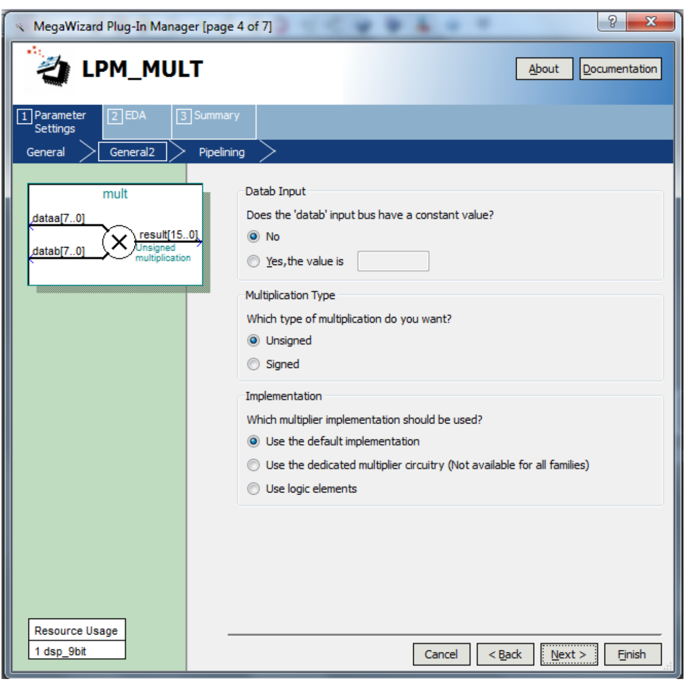


Рисунок 9.4 – Выбор параметров мегафункции, шаг 2

На этом шаге предложено подать на один из входов константу, выбрать знаковый или беззнаковый тип данных и способ реализации-на умножителях, или логических элементах.

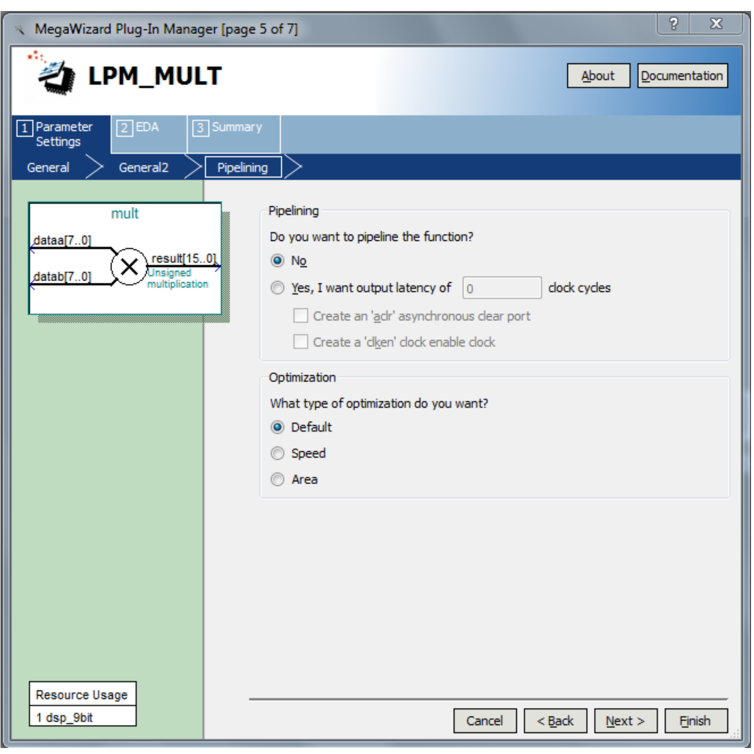


Рисунок 9.5 – Выбор параметров мегафункции, шаг 3

На третьем этапе предложено ввести задержку, ввод разрешения и асинхронного сброса. Так же предложен выбор типа оптимизации. Далее предлагается информация о симуляции и возможность выбора списка соединений, рисунок.

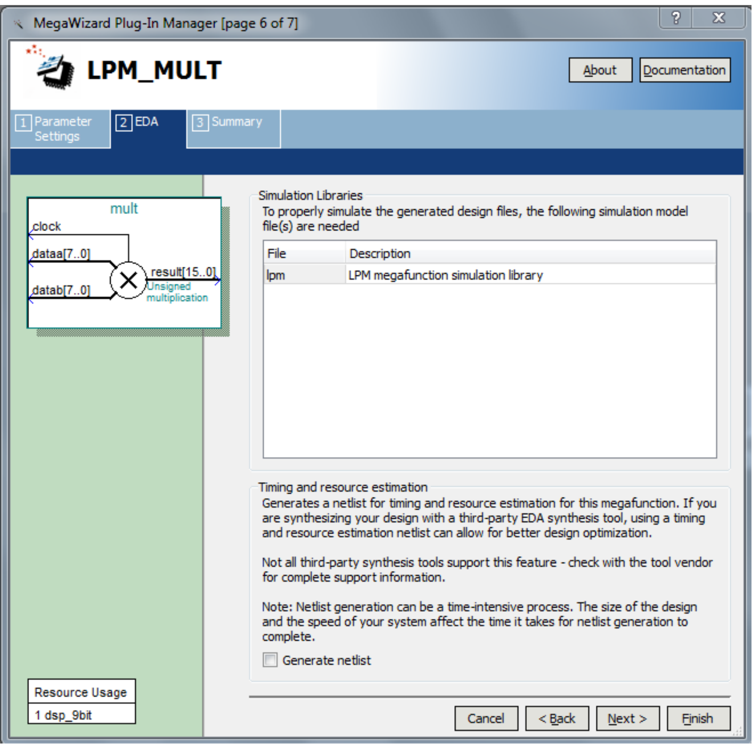


Рисунок 9.6 – Выбор параметров мегафункции, шаг 4

В конце выводится список файлов, нужные необходимо отметить галочкой. Это файлы, представляющие функцию на разных языках, если ничего из предложенного не требуется, отмечать их не надо.

В результате проделанных операций создается модуль mult, с портами clock, dataa, datab, result, который сразу предлагается добавить в проект.

Код сгенерированных модулей как правило весьма сложен и содержит много уровней, не стоит пытаться внести в него изменения с целью модифицировать или изменить параметры функции, для этого лучше открыть MegaWizard, и отредактировать программу с его помощью.

С помощью MegaWizard можно генерировать такие сложные функции, как быстрое преобразование Фурье, различные фильтры, математические операции, синтезаторы частот, интерфейсы передачи данных и др.

Задания на самостоятельную работу

1) Создайте с помощью инструмента MegaWizard сумматор с шестнадцатиразрядными входами.
2) Создайте с помощью инструмента MegaWizard блок ФАПЧ синтезирующий частоту 32 МГц, при частоте генератора 50МГц.
3) Создайте с помощью инструмента MegaWizard блок, вычисляющий квадратный корень числа.
4) Создайте с помощью инструмента MegaWizard блок, вычисляющий арктангенс числа.
5) Создайте с помощью инструмента MegaWizard блок памяти на 32Кб.
6) Создайте с помощью инструмента MegaWizard КИХ фильтр нижних частот с частотой среза 10МГц.

10. Машина конечных состояний

Машина конечных состояний или конечный автомат - это математическая абстракция, используемая при проектировании алгоритмов. Говоря простым языком, машина с конечным числом состояний умеет считывать последовательности входных данных. Когда она считывает входной сигнал, то переключается в новое состояние. Куда именно переключится, получив данный сигнал, - заложено в её текущем состоянии.

Представим устройство, которое читает длинную бумажную ленту. На каждом дюйме этой ленты напечатана буква – *a* и *b*.

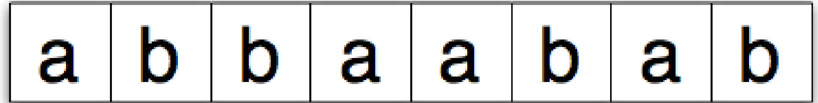


Рисунок 10.1 – Бумажная лента с символами *a* и *b*

как только устройство считывает букву, оно меняет своё состояние. Вот очень простой граф переходов для такой машины:

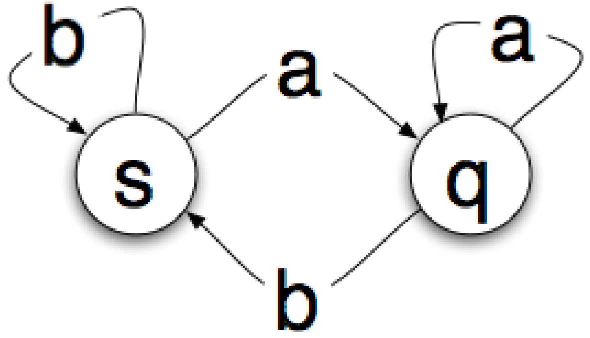


Рисунок 10.2 – Диаграмма переходов состояний.

Кружки – это состояния, в которых машина может быть. Стрелочки – переходы между ними. Так что, если вы находитесь в состоянии *s* и считываете *a*, то вам необходимо перейти в состояние *q*. А если *b*, то просто остаётесь на месте.

Итак, если изначально мы находимся с состоянии *s* и начинаем читать ленту из первого рисунка слева направо, то сначала будет прочитана *a*, и мы переместимся в состояние *q*, затем *b* — вернёмся обратно в *s*. Следующая *b* оставит нас на месте, а *a* вновь переместит в *q*.

Оказывается, если пропустить ленту с буквами через FSM, то по её итоговому состоянию можно сделать некоторые выводы о последовательности букв. Для приведённого выше простого конечного автомата финальное состояние *s* означает, что лента закончилась буквой *b*. Если же мы закончили в состоянии *q*, то последней на ленте была буква *a*.

Рассмотрим пример реализации конечного автомата на языке Verilog. Предположим, что стоит задача реализовать преобразователь уровня в импульс (Level-to-Pulse), который генерирует одиночный импульс каждый раз, когда вход из «0» переходит в «1». Другими словами, это синхронный обнаружитель нарастающего фронта (рис. 10.3).

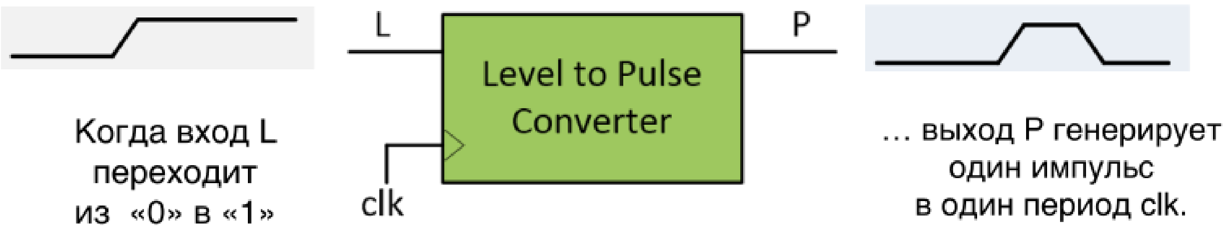


Рисунок 10.3 – Принцип работы преобразователя Level-to-Pulse

Диаграмма состояний для данного алгоритма будет выглядеть следующим образом:

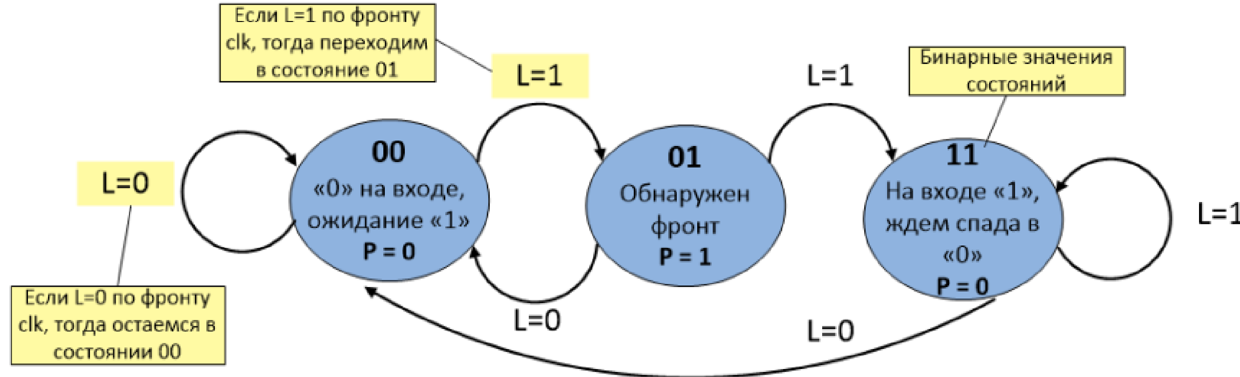


Рисунок 10.4 – Диаграмма состояний

На рис. 10.4 представлена диаграмма состояний для реализации преобразователя уровня в импульс. Как видно по состоянию «01» появляется импульс на регистре “P” после чего следующий импульс появится только после из состояния «00», когда снова произойдет перепад с 0 на 1 на входе «L».

Ниже представлен пример кода на языке Verilog, соответствующий данному алгоритму.

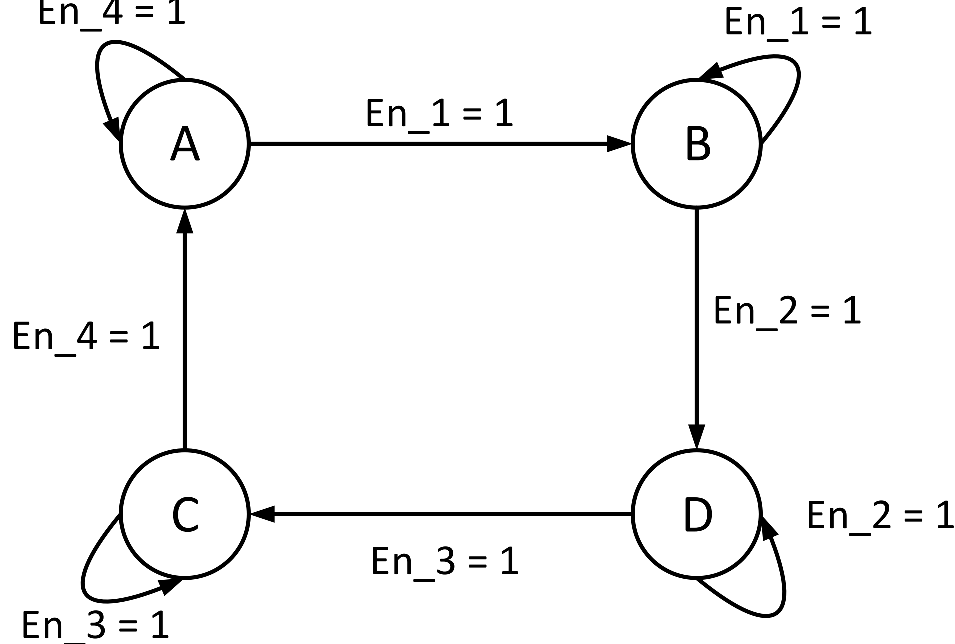
```
module FSM_lab(input reset, input clk, input L, output reg P);
reg [1:0] state;
always @(posedge clk)
if (~reset)
  state <= 0;
else
case(state)
  2'b00: if (L==1) state <= 2'b01;
  2'b01: if (L==1) state <= 2'b11;
         else state <=2'b00;
  2'b11: if (L==0) state <= 2'b00;
  default: state <=0;
endcase
```

```
always @(posedge clk)
if (~reset)
  P<=0;
else if (state==2'b01) P <= 1;
    else P <= 0;
endmodule
```

Задания для самостоятельной работы:

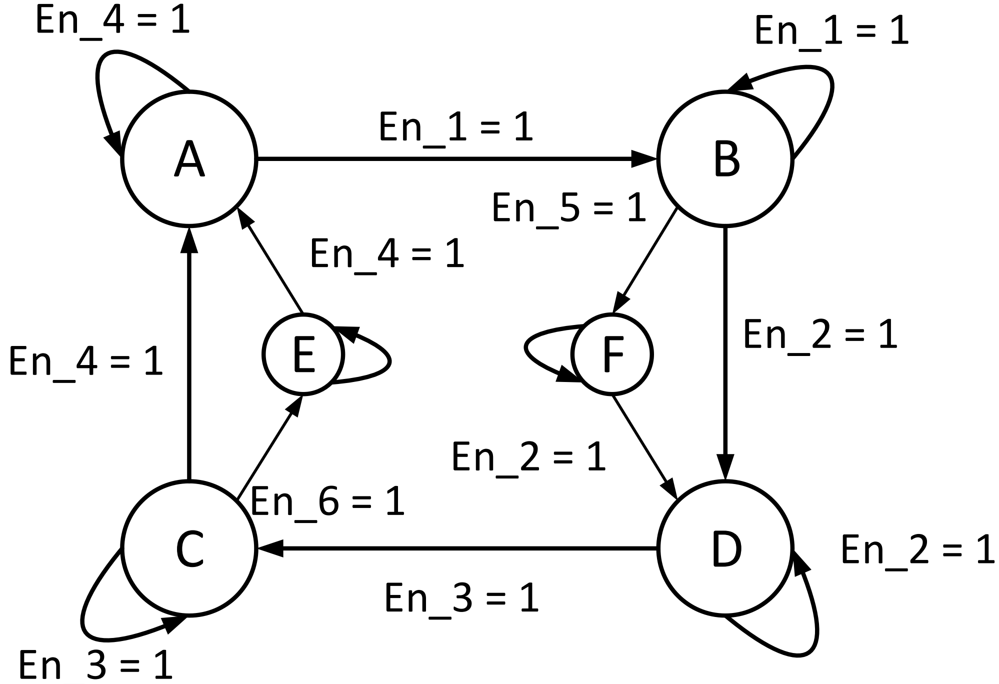
Спроектировать следующие конечные автоматы (представленные в виде диаграмм переходов):

Вариант 1. Схема конечного автомата представлена на рисунке.



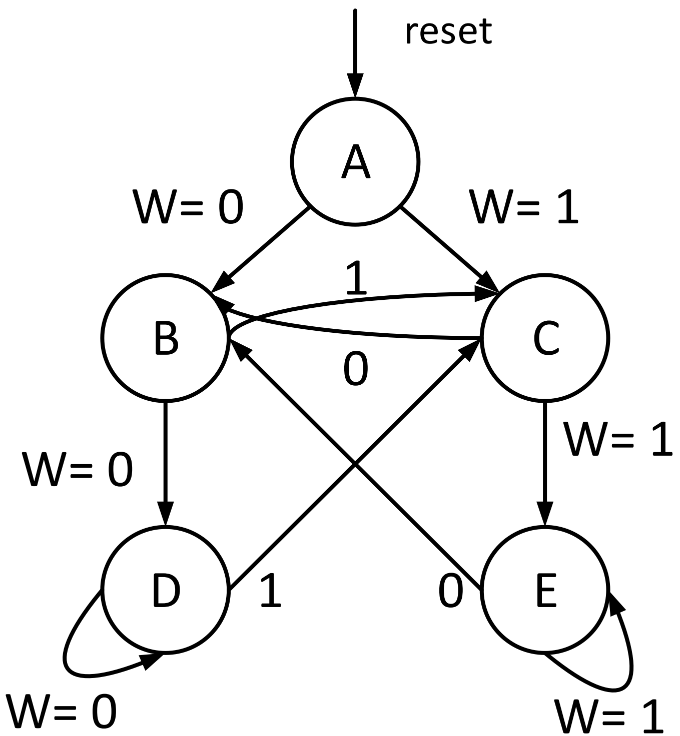
Он имеет 4 состояния – A, B, C и D. Входные сигналы EN_1, EN_2, EN_3, EN_4 и выходной сигнал MAX, зависящий от состояний: A: OUT = 2’b00; B: OUT = 2’b01; C: OUT = 2’b10; D: OUT = 2’b11.

Вариант 2. Схема конечного автомата представлена на рисунке.



Он имеет 4 состояния – A, B, C, D, E и F. Входные сигналы EN_1, EN_2, EN_3, EN_4, EN_5, EN_6 и выходной сигнал OUT, зависящий от состояний: A: OUT =3’b000; B: OUT = 3’b001; C: OUT = 3’b010; D: OUT = 3’b011; E: OUT = 3'b100; F: OUT = 3'b101.

Вариант 3. Схема конечного автомата представлена на рисунке.



Он имеет 8 состояний – A, B, C, D, E. Входные сигналы reset и w, выходной сигнал OUT, зависящий от состояний: A: OUT =3’b000; B: OUT = 3’b001; C: OUT = 3’b010; D: OUT = 3’b011; E: OUT = 3'b100.

11. Модули памяти

Память является важным и часто используемым элементом современных цифровых устройств. Она представляет собой массив однотипных ячеек, хранящих число фиксированной разрядности. По характеру выполняемых с ней операций память подразделяется на:

- постоянные запоминающие устройства (ПЗУ, также ROM – Read-Only Memory), которые хранят фиксированные данные без возможности их изменения;
- оперативные запоминающие устройства (ОЗУ, также RAM – Random Access Memory), допускающие изменение записанных данных.

Типы памяти также разделяют на энергозависимую (сохраняющую данные только при поданном питании) и энергонезависимую. Для ранних вариантов исполнения энергонезависимая память являлась практически синонимом ПЗУ, поскольку техническая реализация таких микросхем подразумевала хранение данных в ячейках с пережигаемыми перемычками, ультрафиолетовым стиранием и т.п. Все эти принципы исполнения обеспечивали сохранность данных при отключении питания, но и не позволяли изменять содержимое памяти без специального оборудования – например, память с ультрафиолетовым стиранием требовала, как следует из ее названия, источника УФ излучения для стирания данных и специального программатора. В настоящее время существуют устройства энергонезависимой памяти, которые допускают изменение содержимого без специального программатора. Например, flash-память, память с электрическим стиранием, память FRAM, другие перспективные типы памяти. Часть из них требует отдельного цикла стирания, а часть позволяет произвольно перезаписывать данные, как для микросхем ОЗУ [2].

По способу организации интерфейса модули памяти подразделяются на модули с асинхронным или синхронным интерфейсом, а также с параллельным или последовательным доступом. Наиболее простой вариант – память с параллельным асинхронным доступом. Графическое изображение такого модуля показано на рис. 11.1.

Рисунок 11.1 - Графическое изображение модуля памяти с асинхронным интерфейсом.

Постоянное запоминающее устройство с асинхронным интерфейсом может быть описано с помощью оператора *case*:

```
module rom(     input [3:0] a,     output [7:0] d     );
        reg [7:0] data;
always @(a)
case (a)
  0: data <= 1;
  1: data <= 3;
  2: data <= 4;
  default : data <= 0;
endcase
assign d = data;
endmodule
```

При описании асинхронного ПЗУ с помощью оператора case используются строки вида <addr>: <data> - для каждого варианта адреса записывается то значение, которое хранится по этому адресу. Можно заметить, что при таком подходе сложно описать массивы памяти большого объема без применения средств автоматизации.

Синхронный интерфейс памяти обеспечивает более высокую производительность, поэтому память такого вида используется чаще. Блочная память, размещаемая в FPGA, является памятью с синхронным интерфейсом. Отличием такого типа интерфейса является выполнение всех действий по фронту тактового сигнала. Для памяти с возможностью чтения и записи (ОЗУ) используются следующие дополнительные сигналы:

din – данные для записи; *we* – разрешение записи (Write Enable).

Память такого типа работает следующим образом: если по фронту тактового сигнала активен сигнал *we*, то производится запись данных *din* в ячейку памяти с адресом *addr*. Иначе производится чтение из памяти, и на выходе *dout* появляется содержимое

ячейки памяти с адресом *addr*. Пример описания на *Verilog* памяти с произвольным доступом с синхронным интерфейсом [2]:

```
module ram(     input clk,
input [7:0] addr,
input [15:0] din,
input we,
output reg [15:0] dout );
reg [15:0] ram_array [7:0];
// Инициализация данных с файла
// initial
//   $readmemh("file_name",   ram_array,   <begin_addr>,
//<end_addr>);
always @(posedge clk)
begin
   if (we)
        begin
           ram_array[addr] <= din;
        end
        dout = ram_array[addr];
end
endmodule
```

Графическое изображение модуля показано на рис. 11.2.

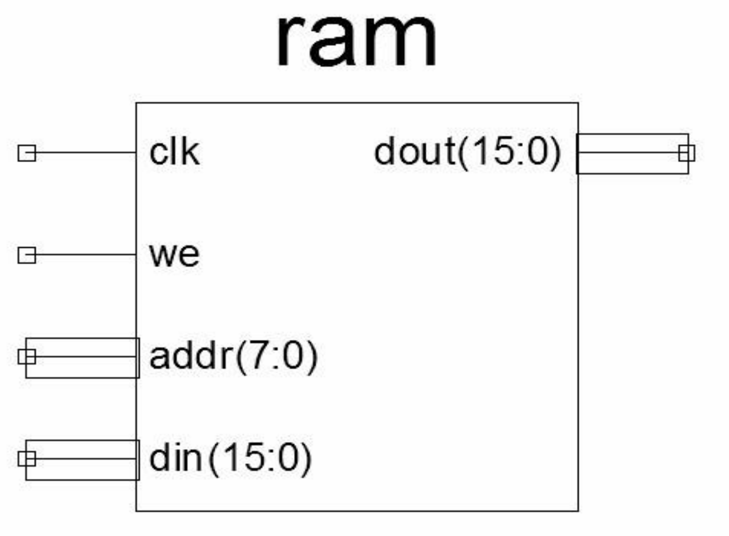


Рисунок 11.2 - Графическое изображение модуля ОЗУ с синхронным интерфейсом.

Из описания порядка работы модуля ОЗУ виден его недостаток, проявляющийся в том, что при записи в память невозможно одновременно читать ее содержимое. Например, при использовании блока памяти для хранения таблицы значений, непрерывно выдаваемых на цифро-аналоговый преобразователь, устройство отображения информации или на иное устройство, требующее непрерывного потока данных, возникнет проблема, связанная с тем, что для обновления содержимого памяти придется прервать процесс чтения записанных в нее данных. От этого недостатка свободны многопортовые модули памяти, которые позволяют обращаться к одному и тому же массиву ячеек с помощью нескольких наборов линий addr, din, we, и имеют соответствующее количество выходных шин dout. Простейшим вариантом многопортовой памяти является двупортовая (dual-port memory), которая имеет достаточно много разновидностей. По функциональным возможностям второго порта двупортовая память подразделяется на simple dual-port, или pseudo dual-port («простая двупортовая», или «псевдо-двупортовая» память), и true dual-port («истинно двупортовая память»). Их отличием является то, что память true dualport имеет два независимых и равноправных порта, по каждому из которых возможно проведение операций чтения и записи. У памяти simple dual-port один порт является универсальным (чтение и запись), а второй – только для чтения. Память такого типа вполне может быть использована в проектах, где требуется обеспечение непрерывного потока читаемых данных. В этом случае при необходимости перезаписи используется универсальный порт, а второй и используется для постоянного считывания. Графическое изображение simple dual-port памяти показано на рис. 11.3 [2].

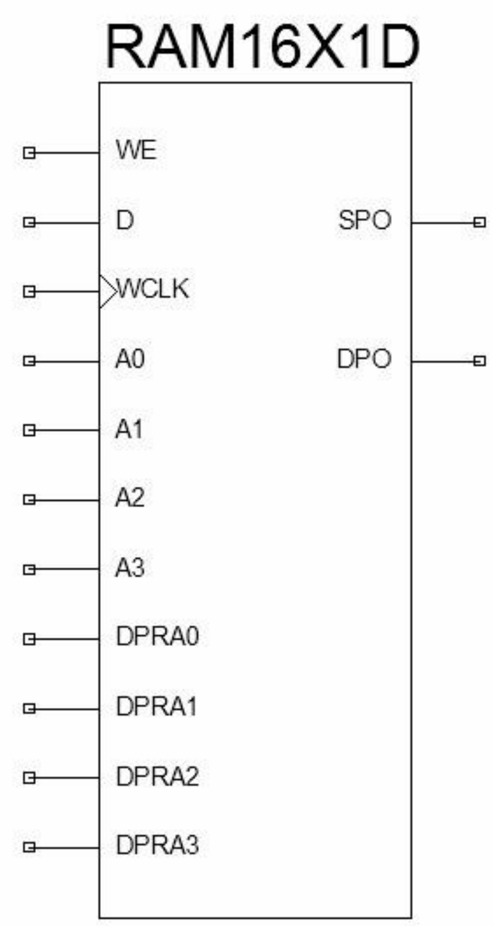


Рисунок 11.3 - Двупортовая память в конфигурации simple dual-port, построенная на базе логической ячейки FPGA

Логические генераторы программируемых ячеек FPGA представляют собой массивы статической памяти, хранящие таблицы истинности. Поэтому они могут быть использованы и в качестве модулей памяти, являясь при этом simple dual-port памятью. Такая память в терминологии FPGA называется также *распределенной* (*distributed*), поскольку распределена по программируемым ячейкам [2].

Задания для самостоятельной работы:

Задание 1:

1. Создайте новый проект и укажите тип используемой микросхемы – EP4CE22F17C6 семейства Cyclone IV E.

2. С помощью генератора мегафункций Mega Wizard Plug-in Manager создайте блок памяти (см. рис. 11.4). Для этого воспользуйтесь функцией RAM: 1-PORT (папка Memory Compiler). В качестве выходного файла укажите Verilog HDL, и задайте имя файлу memory.v (в рабочей директории проекта).

3. На следующих страницах генератора мегафункций укажите формат блока памяти (32 8-разрядных слова) и используемые ресурсы (блок M4K). Оставьте заданную по умолчанию настройку – использовать одну тактовую частоту для синхронизации всех регистров блока памяти. Отключите опцию применения выходных регистров в блоке памяти (как показано на рис. 11.4) – установка Read output port(s) в разделе Which ports should be registered?.

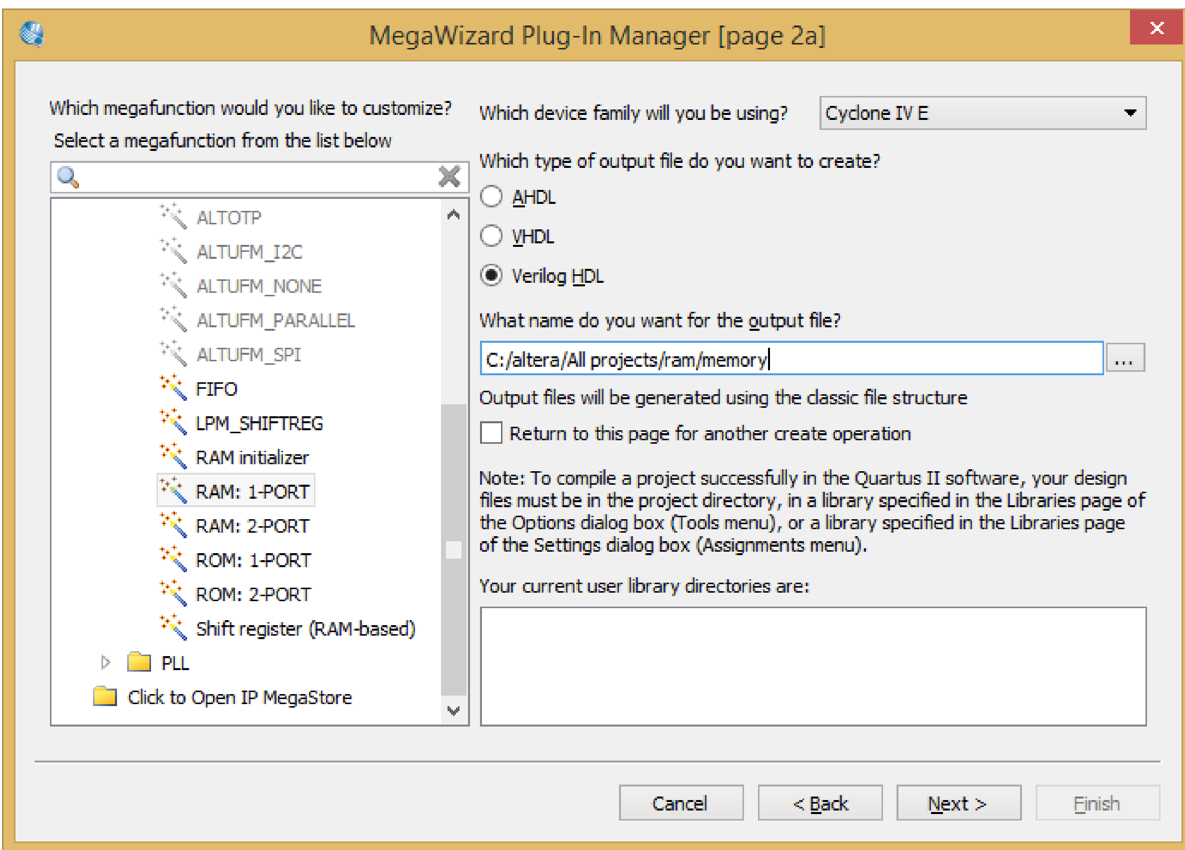


Рисунок 11.4 – Внешний вид генератора мегафункций

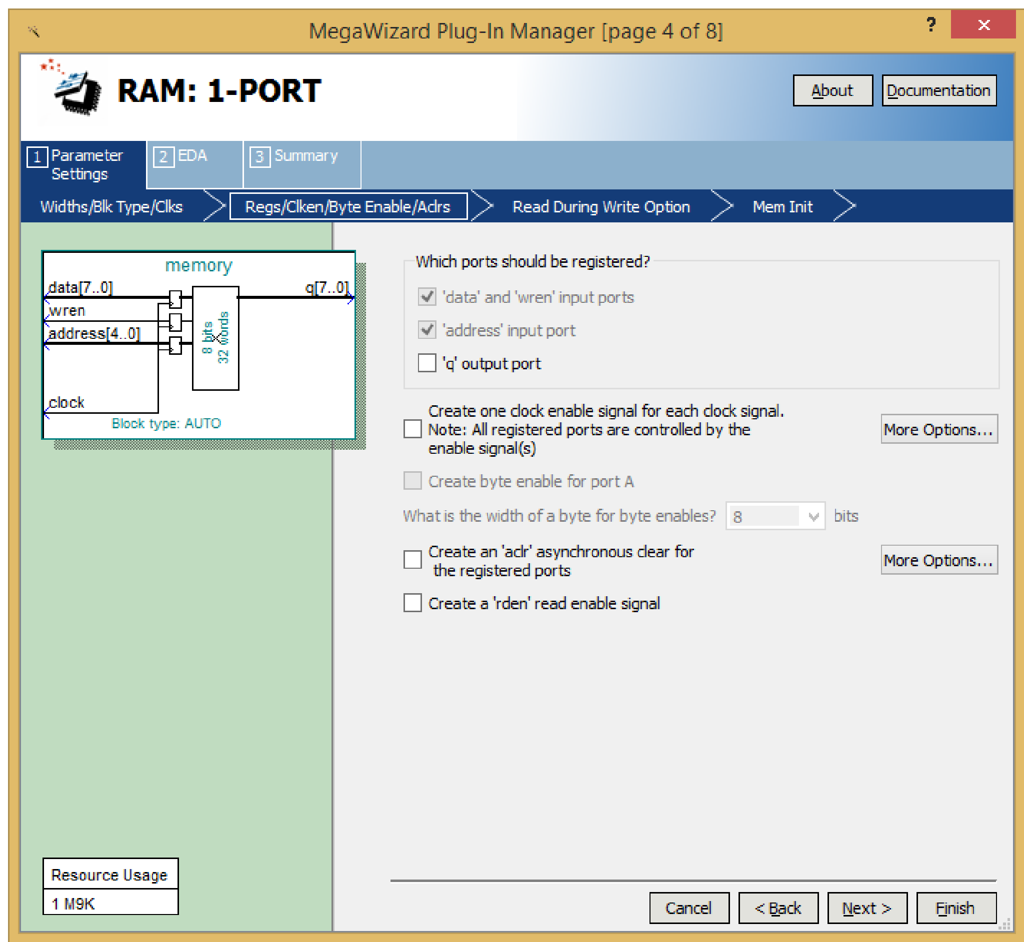


Рисунок 11.5 – Отключение выходных регистров блока памяти

4. Все остальные настройки оставьте по умолчанию.

5. Создайте файл верхнего уровня иерархии. Для подачи входных данных создайте 8-разрядную шину *data[7:0]*, адресация 5-х разрядной шиной *address [4:0]* и т.д. для всех остальных входов и выходов.

6. Проверить работоспособность проекта с помощью программы Modelsim, подав на вход два байта информации произвольной формы.

Задание 2:

1. Создайте блок памяти, используя оператор объявление массива на языке Verilog: *reg [7:0] memory_array [31:0].*

2. Все назначения для входных и выходных сигналов – такие же, как и в предыдущем задании.

3. Проверить работоспособность проекта с помощью программы Modelsim, подав на вход два байта информации произвольной формы.

12. Фильтрация ПЛИС

Цифровая обработка сигналов является одной из основных сфер применения ПЛИС из-за высоких требований к скорости. Рассмотрим реализацию цифровых фильтров на примере КИХ фильтра нижних частот.

Фильтр с конечной импульсной характеристикой (КИХ, FIR- finite impulse response, нерекурсивный фильтр) – один из видов цифровых фильтров с ограниченной по времени импульсной характеристикой.

Фильтр может быть задан разностным уравнением:

$$y(n) = a_0x(n) + a_1x(n-1) + a_2x(n-2) + \cdots + a_kx(n-k) \ , \quad (1)$$

где y(n)-выходной сигнал, a_i-коэффициенты фильтра, x(n-i)-входной сигнал, задержанный на i отсчетов, k-порядок КИХ фильтра. Можно сказать, что значение выхода фильтра есть значение отклика на входное значение фильтра и сумма постепенно затухающий откликов k предыдущих отсчетов. На рисунке 12.1 изображена реализация КИХ - фильтра четвертого порядка.

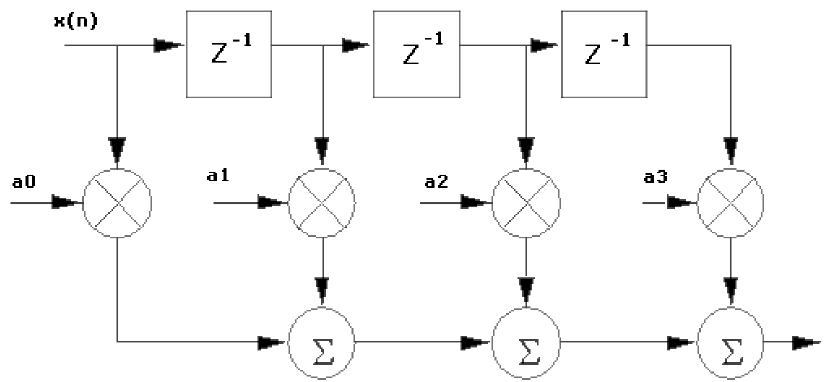


Рисунок 12.1- Реализация КИХ фильтра четвертого порядка

На рис. 12.1 x(n)-входной сигнал, Z^{-1}- задержка, a_0-коэффициенты фильтра, ×-умножитель, ∑-сумматор.

Реализация КИХ фильтра сводится к трем этапам:

- Рассчет коэффициентов фильтра
- Реализация фильтра в Quartus
- Верификация результатов с помощью симулятора

Рассчет коэффициентов может быть произведен в специализированных пакетах программ, таких как Matlab (приложение fdatool), SciLab, и др. Рассмотрим рассчет на примере fdatool MatLab.

Необходимо открыть MatLab, в командной строке ввести «fdatool», откроется окно приложения, рис. 12.2.

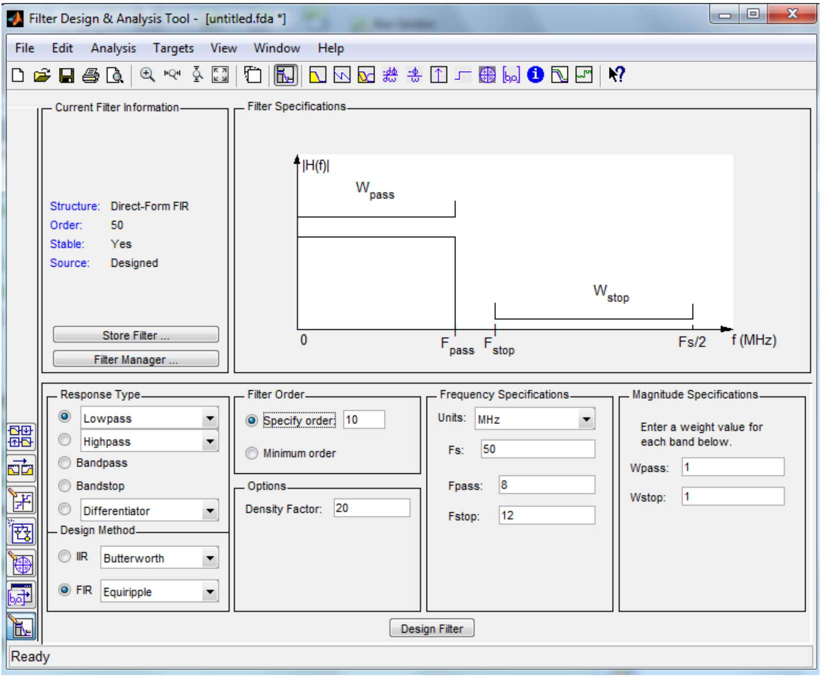


Рисунок 12.2 – Окно приложения Filter Design & Analysis Tool

В окне стоят опции выбора фильтруемых частот (ФНЧ, ФВЧ, ПФ), типа фильтра(FIR, IR), формы АЧХ, порядка фильтра, частот среда, дискретизации. После установки всех необходимых параметров жмем Design Filter, и получаем сгенерированный фильтр и его АЧХ, рис. 12.3.

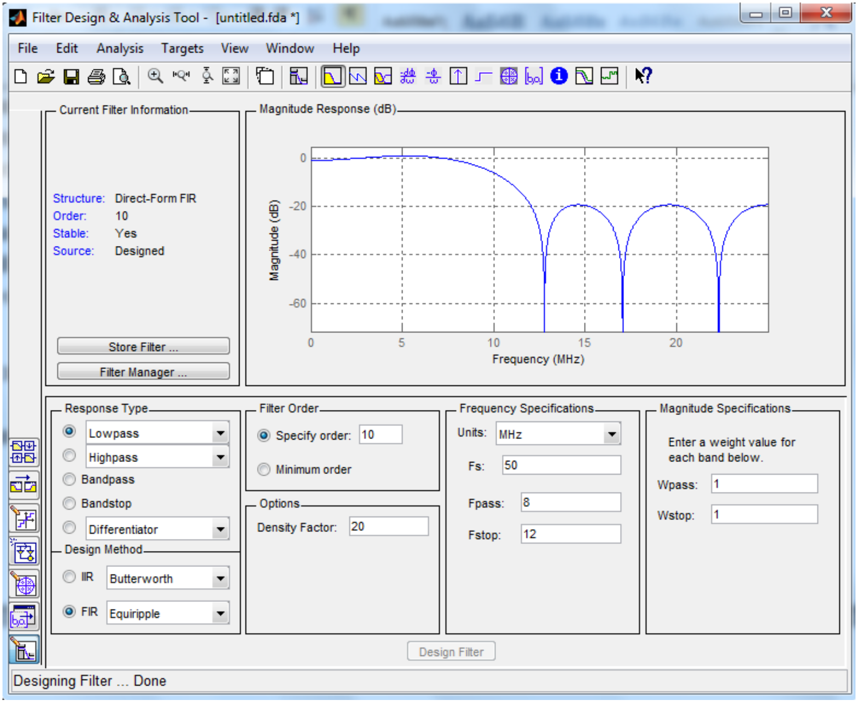


Рисунок 12.3 – Рассчитанный КИХ фильтр

Нам нужно экспортировать коэффициенты фильтра в рабочее пространство MatLab, для этого необходимо нажать File/Export, выбрать название массива коэффициентов, рис. 12.4.

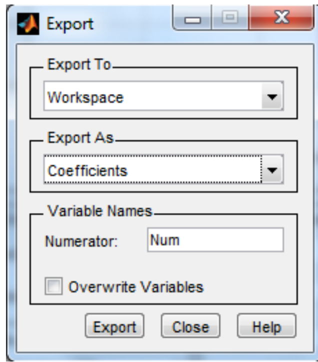


Рисунок 12.4 – Экспорт коэффициентов в рабочую область

При реализации фильтра необходимо умножить входные отсчеты сигнала на коэффициенты фильтра. Отсчеты сигнала имеют размерность 12 бит. Необходимо обеспечить такую же разрядность коэффициентов. Для этого следует отнормируйте полученные коэффициенты относительно модуля максимального. После этой операции максимальный коэффициент станет равный единице, модуль остальных будет лежать в интервале [0:1Теперь нужно привести коэффициенты к разрядности 12, в двоичном виде. Для этого умножим все значения на 2^{n-1} -1 (n=12), при этом максимальный коэффициент будет занимать все n разрядов. Показатель степени 2 равен n-1 в силу того, что старший разряд числа знаковый. В итоге округлим числа до целого значения

Эти операции описывает формула:

$$koeff_{\text{норм}} = \frac{koeff}{max(abs(koeff))} * (2^{n-1} - 1) \ , \qquad (3)$$

которой соответствует код MatLab:

```
coeff_norm=Num./(max(abs(Num)));
coeff_razr=coeff_norm*(2^11-1);
coeff_razr=round(coeff_razr);
```

Полученные коэффициенты:

55 -479 -308 456 1532 2047 1532 456 -308 -479 55

Реализация КИХ-фильтра на ПЛИС

Напишем программу на языке Verilog, описывающую работу КИХ-фильтра. Разрядность входных данных - 12 бит. Коэффициенты фильтра (a) и его задержки (delay_pipe) задаются в виде массива знаковых регистров длиной n (порядок фильтра), с

размерностью в 12 бит. Значения регистров коэффициентов фильтра задаются в блоке Initial:

```
module fir (
        clk,
        reset,
        filter_in,
        filter_out
        );
input wire clk, reset;
input  wire signed  [11:0]  filter_in;
output  wire signed [23:0] filter_out;
reg signed [11:0] delay_pipe [9:0];
reg signed [33:0] summ;
reg signed [11:0] a [10:0];
initial
begin
a[0]=55;
a[1]=479;
a[2]=308;
a[3]=456;
a[4]=1532;
a[5]=2047;
a[6]=1532;
a[7]=456;
a[8]=308;
a[9]=479;
a[10]=55;
        end
```

Задержка отсчетов сигнала реализуется с помощью сдвигового регистра. Такую конструкцию удобнее всего реализовать через цикл for, каждая задержка умножается на соответствующий коэффициент:

```
integer index;
always @(posedge clk)
        begin
                if (reset)
```

```
begin
    summ<=0;
    for (index=0; index<11; index=index+1)
        delay_pipe [index] <=12'b0;
end

else
begin
    delay_pipe[0] <= filter_in;
    for (index=0; index<9; index=index+1)
        begin
            delay_pipe[index+1] <= delay_pipe
[index];
        end
        summ <= filter*a[0] + delay_pipe[0]*a[1]
+ delay_pipe[1]*a[2] + delay_pipe[2]*a[3] + delay_pipe[3]*a[4] +
delay_pipe[4]*a[5] + delay_pipe[5]*a[6] + delay_pipe[6]*a[7] + delay_pipe[7]*a[8]
+ delay_pipe[8]*a[9] + delay_pipe[9]*a[10];
end
end
assign filter_out = summ[33:10];
endmodule
```

Операции в регистрах задержки происходят синхронно с тактовой частотой(always блок). Младшему регистру массива delay_pipe (массив регистров задержки) присваивается входной сигнал, остальные регистры массива передают свои значения друг другу по цепочке. Каждая задержка умножается на свой коэффициент, результат n произведений суммируется. Таким образом система использует n умножителей 12*12. Обратите внимание, что входной сигнал без задержки также умножается на свой коэффициент (a_0), и суммируется с остальными.

Результат произведения и суммирования будет иметь максимальную потенциальную разрядность, равную 34, на выходе она обрезается до 24 бит.

Верификация

Для верификации фильтра в ModelSim подадим на него в тестбенче частотно модулированный сигнал, сгенерированный в Matlab, код Matlab:

```
N=2^18;      % количество точек:
Fs=50e6;    % частота дискретизации
t=(0:N)/Fs;  % массив времени для сигнала
df=25e6;     % перестройка частоты - полоса сигнала
f0=1e6;        % начальная частота, Гц
t1=max(t)/4; % время окончания сигнала
koeff=(2^11-1); % коэффициент для приведения сигнала к нужной разрядности
s=chirp(t,f0,t1,df);
s=s./max(abs(s));
s=s.*koeff;
```

В итоге получаем ЛЧМ сигнал, показанный на рис. 12.5

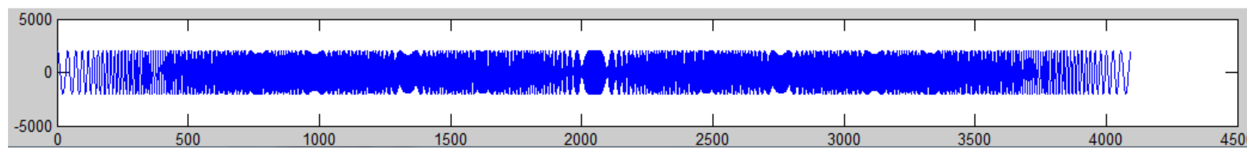


Рисунок 12.5 – Сигнал, сгенерированный в MatLab

Код модуля тестбенч написанный для верификации фильтра:

```
`timescale 10 ns / 10 ns
module fir_tb;
reg clk;
reg reset;
wire signed [13:0] filter_out;
integer signal_in_tb,data_read;
reg signed [11:0] input_signal;

// clock generation
always
begin
clk <= 1'b1;
 # 1;
clk <= 1'b0;
# 1;
end

initial
```

```
begin
reset <=1'b1;
        # 30;
reset <=1'b0;
end

initial
    begin
        signal_in_tb =
$fopen("C:/Users/Dmitriy/Desktop/tb_fir/m_test.txt","r");
    end
always  @(posedge clk)
    if(!reset)
        begin
            data_read<=$fscanf(signal_in_tb,"%d",input_signal);
        end
    else
        input_signal <=1'b0;

fir fir (
      .clk(clk),
    .reset(reset),
    .filter_in(input_signal),
    .filter_out(filter_out)
    );
endmodule
```

На рисунке 12.6 изображены результаты симуляции:

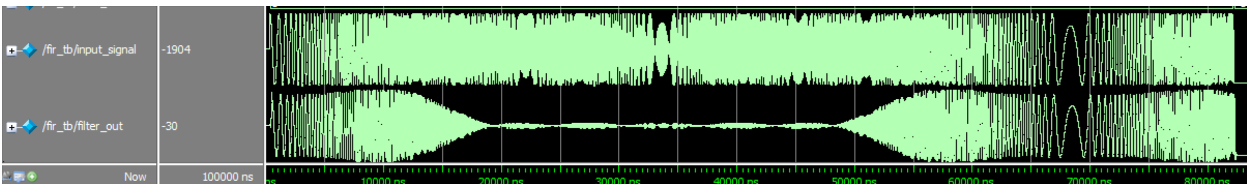


Рисунок 12.6 результаты симуляции фильтра в Modelsim

Огибающая сигнала на выходе-АЧХ фильтра.

Для более точной симуляции и дальнейшей обработки сигнал нужно экспортировать в MatLab, умножить на оконную функцию и взять быстрое

преобразование Фурье, сравнить прохождение сигнала через фильтр в MatLab и модуле Verilog.

Задания на самостоятельную работу

1) Создайте и просимулируйте модуль КИХ фильтра 12-го порядка, ФВЧ, частота среза 15МГц, частота дискретизации 50МГц.
2) Создайте и просимулируйте модуль КИХ фильтра 11-го порядка, ПФ, частоты среза 8МГц, 18МГц, частота дискретизации 50МГц.
3) Создайте и просимулируйте модуль БИХ фильтра 11-го порядка, ФНЧ, частота среза 8МГц, частота дискретизации 50МГц.
4) Создайте и просимулируйте модуль БИХ фильтра 12-го порядка, ФВЧ, частота среза 15МГц, частота дискретизации 50МГц.
5) Создайте и просимулируйте модуль CIC фильтра 12-го порядка, ФВЧ, частота среза 15МГц, частота дискретизации 50МГц.
6) Создайте и просимулируйте модуль CIC фильтра 12-го порядка, ФНЧ, частота среза 8МГц, частота дискретизации 50МГц.
7) Экспортируйте данные с выхода фильтра, приведенного в примере, в MatLab, постройте АЧХ фильтра.

13. Согласование модулей

Как правило, проект состоит из большого количества модулей разного уровня, составляющих иерархическую структуру. Для подключения модуля используется оператор включения: Синтаксис оператора включения модуля:

Имя_Включаемого_Модуля Имя_Включения(Интерфейс);

Здесь *Имя_Включения* – обязательный параметр, представляющий собой уникальный идентификатор для данного включения; *Интерфейс* – список сигналов модуля верхнего уровня, присоединяемых к входам и выходам включаемого подчиненного модуля. При этом подключение осуществляется в порядке следования сигналов, например:

```
//Включаемый модуль:
module Delay ( X1, X2 );
 input X1; // X1 – входной порт модуля
 output X2; // X2 – выходной порт модуля
 wire X1, X2;
```

```
// . . . Текст модуля
endmodule

//Модуль верхнего уровня:
module Main ( a, b );
 input a; // a – входной порт модуля
 output b; // b – выходной порт модуля
 wire a, b;
 // Включение модуля нижнего уровня Delay под именем INST00:
 Delay INST00 ( a, b ); //*

// . . . Текст модуля
endmodule
```

Существует также другая форма записи интерфейса в операторе включения, которая предполагает явное указание соответствия между подключаемыми сигналами и портами включаемого модуля. В этом случае синтаксис оператора включения модуля следует записать таким образом:

Модуль Включение (.порт1(сигнал1), .порт2(сигнал2)); ,

где *порт1, порт2* – идентификаторы портов включаемого модуля; *сигнал1*, *сигнал2*– идентификаторы подключаемых сигналов модуля верхнего уровня.

Предыдущая программа, записанная с использованием явного указания соответствия сигналов в операторе включения модуля, приведена ниже:

```
//Включаемый модуль:
module Delay ( X1, X2 );
 input X1; // X1 – входной порт модуля
 output X2; // X2 – выходной порт модуля
 wire X1, X2;
// . . . Текст модуля
endmodule

//Модуль верхнего уровня:
module Main ( a, b );
 input a; // a – входной порт модуля
 output b; // b – выходной порт модуля
```

```
wire a, b;
// Включение модуля нижнего уровня Delay под именем INST00:
Delay INST00 (
.X1(a),
.X2(b)
);
endmodule
```

При использовании операторов включения модуля необходимо следить за возможным образованием циклических включений. Пример такого включения приведен ниже:

```
// Циклическое включение модулей
module XMod (a,b) ;
 input a;
 output b;
 wire a,b;
 YMod mod01 (a,b); //Включение модуля YMod
endmodule

module YMod (c,d);
 input c;
 output d;
 wire c,d;
 XMod mod02(c,d); //Включение модуля XMod
endmodule
```

Как видно из вышеприведенного примера, модуль *XMod* являетсячастью модуля *YMod*, а, в свою очередь, модуль *YMod* является частью модуля *XMod*. Представить такую схему в виде конечной структуры для реализации определенной задачи невозможно, однако средства компиляции популярных программных пакетов, реализующих Verilog, не распознают подобную ошибку. Обычно эта ошибка выявляется только перед началом моделирования, когда производится компоновка отдельных структурных частей устройства.

Для корректности согласования типов и направлений портов включаемого модуля с внутренними сигналами модуля верхнего уровня следует руководствоваться следующими общими правилами:

- для выходных портов подчиненного модуля: внутренний сигнал включаемого модуля может иметь тип «цепь» или «регистр», а внешний сигнал обязательно должен иметь тип «цепь»;
- для входных портов подчиненного модуля: внутренний сигнал включаемого модуля обязательно должен иметь тип «цепь», а внешний сигнал может иметь тип «цепь» или «регистр»;
- для двунаправленных портов подчиненного модуля и внешний, и внутренний сигналы должны соответствовать типу «цепь».

Рис. 13.1 иллюстрирует применение данных правил.

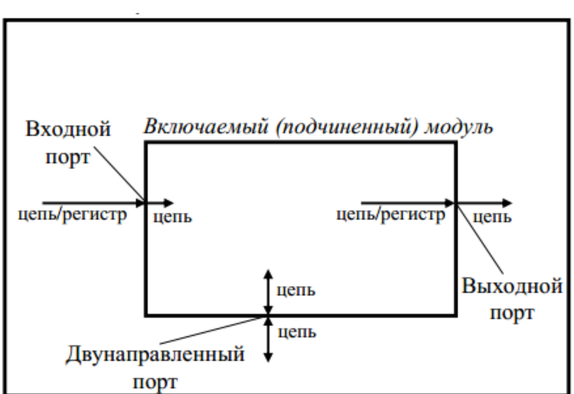


Рис. 13.1. Правила согласования типов и направлений внутренних и внешних сигналов при включении модулей

Создание настраиваемых модулей

Часто при написании программ возникает необходимость в использовании в различных частях проекта аналогичных по функциям модулей, отличающихся, однако, временем задержки, разрядностью входных и выходных сигналов или другими подобными характеристиками. Создание отдельных модулей для каждого такого случая приведет к неоправданному многократному повторению фрагментов кода. Так, например, модуль «8-битный регистр» будет отличаться от модуля «16-битный регистр» только операторами, описывающими типы портов. Избежать неэффективных решений такого рода позволяет применение определяемых внутри модуля констант – параметров (parameter), дающих возможность изменения различных аспектов функционирования при использовании модулей в различных включениях как структурного элемента проекта.

В качестве примера рассмотрим разработку универсального параллельного регистра с входами синхронизации и разрешения на запись [25]. Разрядность регистра и время его срабатывания являются настраиваемыми параметрами:

```
module Reg (X_In, CLK, WE, X_Out);
 input X_In;
 input CLK;
 input WE;
 output X_Out;

 // Настраиваемый параметр – разрядность регистра
 parameter n = 5;
  reg [n:0] X_Out; // Разрядность задана с помощью параметра n
 wire CLK;
 wire WE;

 //Задержка задана с помощью параметра
 //Setting_Time (см. разделы 4.1.2, 4.2.7)
 always @(posedge CLK) //Поведенческие операторы
 #Setting_Time X_Out = WE? X_In : X_Out;
endmodule

 // Настраиваемый параметр – время срабатывания регистра
 parameter Setting_Time = 9; // = 9 ´ 10 ns = 90 ns
 wire [n:0] X_In; // Разрядность задана с помощью параметра n
```

Далее создадим модуль верхнего уровня, дважды включающий в себя модуль Reg с различными значениями параметров. Включение Reg16 интерпретируется как 16-разрядный регистр с временем задержки 40 ns, а включение Reg4 – как 4-разрядный регистр с временем задержки 70 ns:

```
module Main ();
  reg [15:0] Bus01;
 wire [15:0] Bus02;
 reg [3:0] Bus03;
 wire [3:0] Bus04;
  wire clk, we;
  // Определение параметров включения Reg16
 defparam Reg16.n = 15;
```

```
defparam Reg16.Setting_Time = 4;
Reg     Reg16 (
.X_In(Bus01),
.CLK(clk),
                .WE(we),
.X_Out(Bus02)
);
 // Определение параметров включения Reg4
defparam Reg4.n = 3;
defparam Reg4.Setting_Time = 7;
Reg     Reg4 (
.X_In(Bus03),
.CLK(clk),
.WE(we),
.X_Out(Bus04)
 );
endmodule
```

В вышеприведенной программе для установки конкретных значений настраиваемых параметров включаемого модуля Reg использован оператор defparam, синтаксис которого приводится ниже:

defparam Имя_Включения.Параметр = Значение_Параметра;

Таким образом, использование настраиваемых параметров (например, n и Setting_Time) представляет собой удобный механизм для создания гибких универсальных настраиваемых структурных компонентов.[1]

Задания на самостоятельную работу

1) Создайте проект на основе схемы, приведенной на рисунке 13.2. Sum-сумматор, Mult-умножитель, реализуйте их в отдельных модулях. Разрядность выхода сумматора и выхода умножителя поставьте самостоятельно

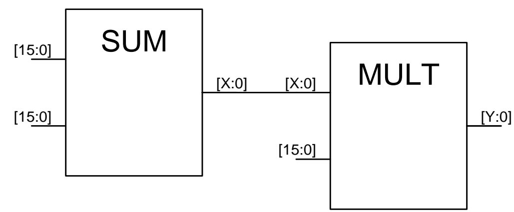


Рисунок 13.2 - Задание 1

2) Создайте проект на основе схемы, приведенной на рисунке 13.3. Sum-сумматор, Mult-умножитель, реализуйте их в отдельных модулях. Разрядность выхода сумматора и выхода умножителя поставьте самостоятельно, задайте их с помощью . *defparam.*

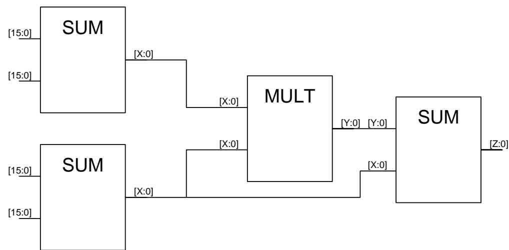


Рисунок 13.3 - Задание 2

3) Создайте проект на основе схемы, приведенной на рисунке 13.4. Sum-сумматор, Mult-умножитель, реализуйте их в отдельных модулях. Разрядность выхода сумматора и выхода умножителя поставьте самостоятельно, задайте их с помощью. *defparam.*

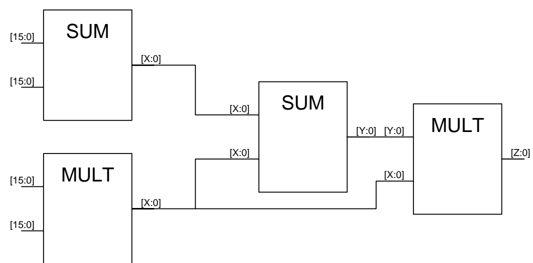


Рисунок 13.4 - Задание 3

4) Создайте проект на основе схемы, приведенной на рисунке 13.5 Sum-сумматор, Mult-умножитель, реализуйте их в отдельных модулях. Разрядность выхода сумматора и выхода умножителя поставьте самостоятельно, задайте их с помощью. defparam.

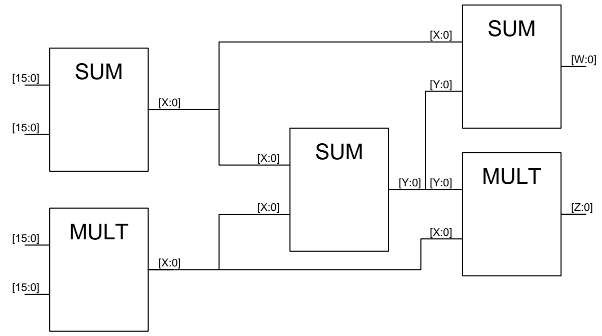


Рисунок 13.5 - Задание 4

5) Создайте проект на основе схемы, приведенной на рисунке 13.6. Sum-сумматор, Mult-умножитель, реализуйте их в отдельных модулях. Разрядность выхода сумматора и выхода умножителя поставьте самостоятельно, задайте их с помощью. *defparam.*

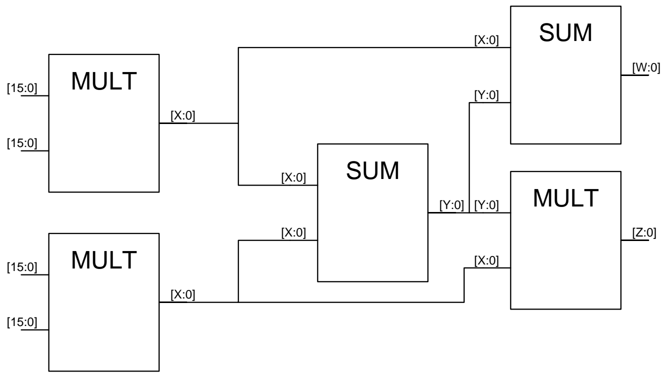


Рисунок 13.6 - Задание 4

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. И.Г. Каршенбойм Краткий курс HDL, Компоненты и Технологии №1, 2008 - М: Компоненты и Технологии, 2008

2. Тарасов И.Е., Певцов Е.Ф. Программируемые логические схемы и их применение в схемотехнических решениях: Учебное пособие / Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования "Московский государственный технический университет радиотехники, электроники и автоматики" – М., 2012. – 184 с.

3. Quartus II Handbook, Volume 3. Verification [Electronic Resource] //Mode of access: http://www. altera. com/literature/hb/qts/quartusii_handbook. pdf.

4. Ю.П.Кондратенко, В.В.Мохор, С.А.Сидоренко. Verilog-HDL для моделирования и синтеза цифровых электронных схем – Н: НГТУ, 2002