Реализация CORDIC-алгоритма на ПЛИС

Дмитрий ДАЙНЕКО
dde29@sibmail.com

Введение

При прохождении практики в университете автор начал разрабатывать устройства на ПЛИС. Первым делом нужно было найти литературу по выбранной теме. Не только теоретические материалы, но и практические, в которых были бы представлены примеры готовых проектов, что способствовало бы лучшему пониманию поставленной задачи. Имея четкое представление о языках описания аппаратуры, автор ощущал нехватку информации о том, в каком ПО удобнее писать HDL-код, где эффективнее моделировать и отлаживать логику работы. Другими словами, нигде в литературе не был представлен реальный пример проекта от момента создания — математического описания определенного алгоритма, основ создания проекта в САПР — до моделирования с помощью временных диаграмм. Именно это побудило автора написать статью, в которой им была предпринята попытка систематизировать сведения и показать полный цикл создания системы, которая генерирует синусоидальный сигнал алгоритмом CORDIC.

CORDIC — это аббревиатура от Coordinate Rotation Digital Computer: цифровое вычисление поворота системы координат.

Содержание статьи:

- Обоснованность использования и математическое описание CORDIC-алгоритма.
- Создание HDL-кода проекта.
- Создание проекта в ModelSim.
- Моделирование в ModelSim.

Представьте, что в вашей системе необходимо сгенерировать квадратурный цифровой сигнал, который затем нужно перевести в аналоговый вид с помощью цифро-аналогового преобразователя (ЦАП) (рис. 1).

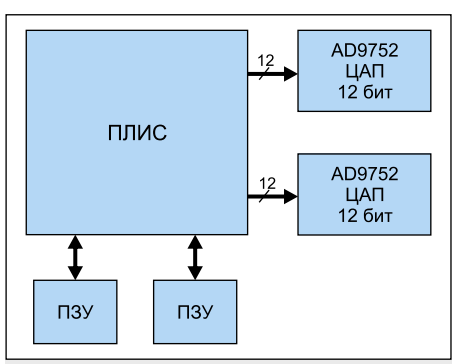

Рис. 1. Простейшая схема

Конечно, можно этот проект реализовать следующим образом. Поставить два постоянных запоминающих устройства (ПЗУ), в которых будут храниться готовые отсчеты обеих синусоид, и внутри ПЛИС реализовать простейший конечный автомат, который циклически считывал бы отсчеты и отправлял бы их на каждый из ЦАП.

Это решение, безусловно, легко реализовать (конечный автомат — простейшее, что можно сделать в ПЛИС), но оно имеет несколько минусов. Это два дополнительных компонента на схеме, а значит, увеличиваются габариты, потребление и стоимость окончательного изделия. Но если мы будем использовать CORDIC-алгоритм, который сам рассчитает отсчеты для ЦАП с нужной нам точностью, то избавимся от «лишних» компонентов на плате.

Читатель может заявить, что внутри ПЛИС имеются встроенные блоки памяти, которые после подачи питания будут переписывать готовые отсчеты из того же ПЗУ или конфигурационной flash-памяти, содержащей прошивку для ПЛИС. Но следует подчеркнуть, что, помимо генератора синусоидального сигнала, в проект на ПЛИС могут входить различные интерфейсы для остальной периферии на плате и какие-то дополнительные вычисления, которые требуют обязательного наличия памяти в ПЛИС. А значит, занимать встроенную память еще и для генератора синусоидального сигнала было бы излишне.

Так как автор решил «с нуля» преподнести разработку проекта, начнем с определения CORDIC-алгоритма и его математического описания. Это поможет читателю тогда, когда мы начнем писать HDL-код нашего проекта.

Математическая часть

Первоначально CORDIC-алгоритм был придуман для поворота вектора на плоскости с помощью операций «сдвиг регистра вправо» и «сложение регистров». Другими словами — для реализации поворота вектора аппаратно (при помощи цифровой схемотехники).

Рассмотрим суть этого алгоритма. Например, нам необходимо повернуть некий вектор (рис. 2) с координатами (x_0, y_0) на угол φ, то есть нужно вычислить его новые координаты.

Координаты x_1 и y_1 вычисляются по формулам:

$$x_1 = x_0 \times \cos(\varphi) - y_0 \times \sin(\varphi),$$

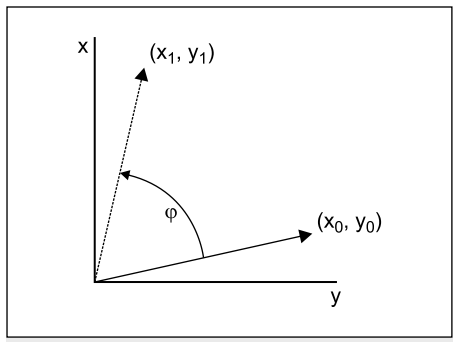

Рис. 2. Поворот вектора

$$y_1 = x_0 \times \sin(\varphi) + y_0 \times \cos(\varphi).$$

Проделав простейшие тригонометрические преобразования, эти формулы можно переписать в виде:

$$x_1 = \cos(\varphi) \times (x_0 - y_0 \times \tan(\varphi)),$$

$$y_1 = \cos(\varphi) \times (y_0 + x_0 \times \tan(\varphi)).$$

Если выбирать такой угол поворота, что $\tan(\varphi) = \pm 2^{-i}$, где i — целое число, то умножение значений x_0 и y_0 на $\tan(\varphi)$ превращается в простую операцию сдвига значений x_0 и y_0 на i разрядов (если представить их в двоичном счислении) вправо.

Читатель, наверное, уже догадался, о чем идет речь. Если некий произвольный угол представить в виде суммы углов:

$$\varphi_i = \pm \operatorname{atan}(2^{-i}), \text{ где } i = 0, 1, 2 \text{ и т.д.},$$

то операция поворота вектора будет состоять из последовательных элементарных поворотов. Также необходимо отметить, что направление поворота не влияет на множитель $\cos(\varphi)$, так как функция cos — четная. В формулах $\cos(\varphi)$ можно представить как $\cos(\operatorname{atan}(2^{-i}))$. Так как i = 0, 1, 2…, то данная функция является сходящейся, результат обычно обозначается как K_i, равен ≈0,607 и называется коэффициентом деформации. Значит, помимо операций «сдвига» и «суммирование/вычитание» векторов, необходимо полученные координаты умножить на этот коэффициент деформации.

Рассмотрим несколько итераций поворота вектора. Допустим, необходимо рассчитать координаты вектора, показанного на рис. 3.

Дабы не увлекаться однообразными математическими расчетами, проведем только

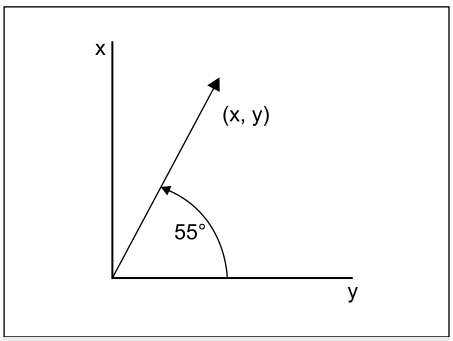

Рис. 3. Рассчитываемый вектор

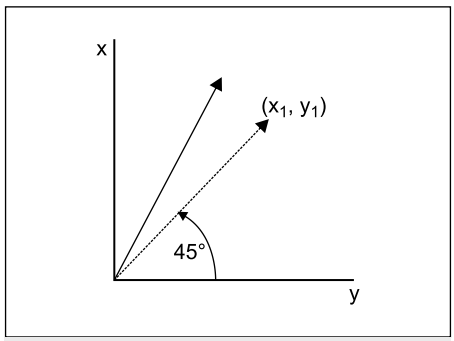

Рис. 4. Первая итерация

три первые и одну последнюю итерации поворота. Каждая итерация содержит следующие вычисления:

$$\sigma_i = \text{sign}(z_i),$$

$$x_{i+1} = x_i - \sigma_i \times y_i \times 2^{-i},$$

$$y_{i+1} = y_i + \sigma_i \times x_i \times 2^{-i},$$

$$z_{i+1} = z_i - \sigma_i \times \text{atan}(z^{-i}).$$

Здесь σ_i определяет направление поворота (принимает значение –1 или +1). Множитель 2^{-i}, как догадывается читатель, и есть этот пресловутый tan(φ).

Представим, что исходный вектор длиной 10 полностью лежит на оси X. Вектор необходимо повернуть на угол 55° (рис. 3). Координаты нового вектора:

$$X_{calc} = \cos(55) \times 10 = 5{,}73,$$

$$Y_{calc} = \sin(55) \times 10 = 8{,}19.$$

Напомним, что окончательные результаты, рассчитанные с помощью CORDIC-алгоритма, необходимо умножить на коэффициент деформации K_i, который равен 0,607.

Расчет первой итерации:

$$\sigma_0 = \text{sign}(z_0) = \text{sign}(55) = +1,$$

$$x_1 = x_0 - \sigma_0 \times y_0 \times 2^0 = 10 - 1 \times 0 \times 1 = 10,$$

$$y_1 = y_0 + \sigma_0 \times x_0 \times 2^0 = 0 + 1 \times 10 \times 1 = 10,$$

$$z_1 = z_0 - \sigma_0 \times \text{atan}(z^0) = 55 - 45 = 10°.$$

В первой итерации мы повернули вектор на 45°, что в итоге составило разницу с нашим углом 10°. Заметим, что направление поворота определяется именно этой разницей (рис. 4).

Расчет второй итерации:

$$\sigma_1 = \text{sign}(z_1) = \text{sign}(10) = +1,$$

$$x_2 = x_1 - \sigma_1 \times y_1 \times 2^{-1} = 10 - 1 \times 10 \times 0{,}5 = 5,$$

$$y_2 = y_1 + \sigma_1 \times x_1 \times 2^{-1} = 10 + 1 \times 10 \times 0{,}5 = 15,$$

$$z_2 = z_1 - \sigma_1 \times \text{atan}(z^{-1}) = 10 - 26{,}6 = -16{,}6°.$$

Так как разница углов в первой итерации получилась положительной, направление поворота не изменилось. Полученный в первой итерации вектор мы повернули еще, на меньший угол — 26,6° (рис. 5).

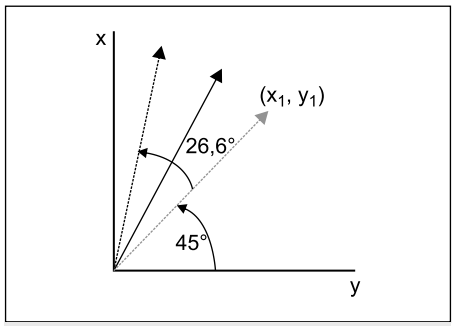

Рис. 5. Вторая итерация

Здесь разница в углах получилась отрицательной, значит, в третьей итерации (рис. 6) мы должны изменить направление расчета.

Расчет третьей итерации:

$$\sigma_2 = \text{sign}(z_2) = \text{sign}(-16{,}6) = -1,$$

$$x_3 = x_2 - \sigma_2 \times y_2 \times 2^{-2} = 5 + 1 \times 15 \times 0{,}25 = 8{,}75,$$

$$y_3 = y_2 + \sigma_2 \times x_2 \times 2^{-2} = 15 - 1 \times 5 \times 0{,}25 = 13{,}75,$$

$$z_3 = z_2 - \sigma_2 \times \text{atan}(z^{-2}) = -16{,}6 + 14{,}03 = -2{,}57°.$$

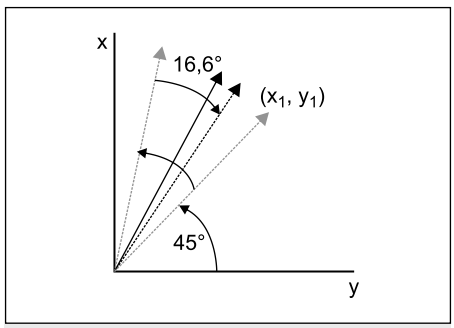

Рис. 6. Третья итерация

Мы заметили, что с каждой итерацией разница между углами уменьшается. Поэтому мы можем сказать, что чем больше итераций проделаем, тем ближе подберемся к нужному вектору, а значит, меньше будет погрешность результата.

Давайте пропустим последующие итерации, а рассмотрим только 8-ю итерацию, чтобы убедиться в верности расчета координат угла CORDIC-алгоритмом:

$$\sigma_7 = \text{sign}(z_7) = \text{sign}(0{,}085) = +1,$$

$$x_8 = x_7 - \sigma_7 \times y_7 \times 2^{-7} = 9{,}47 - 1 \times 13{,}47 \times 2^{-7} = 9{,}46,$$

$$y_8 = y_7 + \sigma_7 \times x_7 \times 2^{-7} = 13{,}47 + 1 \times 9{,}47 \times 2^{-7} = 13{,}48,$$

$$z_8 = z_7 - \sigma_7 \times \text{atan}(z^{-7}) = 0{,}085 - 0{,}45 = -0{,}365°.$$

Здесь мы видим, что ошибка в вычислениях при восьми итерациях составила 0,365°.

Теперь, чтобы сверить координаты вектора, полученного CORDIC-алгоритмом, с ранее рассчитанными, если так можно выразиться, на калькуляторе, нужно полученные значения координат умножить на коэффициент деформации K_i:

$$X_{cordic} \times K_i = 9{,}46 \times 0{,}607 = 5{,}74,$$

$$Y_{cordic} \times K_i = 13{,}48 \times 0{,}607 = 8{,}18.$$

Как видим, отличаются они всего на 0,01. Таким образом, нужная точность вычислений достигается увеличением количества итераций.

Надеемся, математическая часть статьи не утомила читателя, но напомним: мы стремимся практически «с нуля» реализовать на «железе» цифровую синусоиду.

Создание HDL-кода проекта

Чтобы лучше представлять себе систему, а также четко понимать взаимодействие модулей проекта между собой, составим структурную схему (рис. 7).

Итак, в нашей системе должен быть один вход — тактовый (например, 96 МГц), два выхода данных для обоих ЦАП (разрядностью 12 бит) и тактовые частоты для обоих ЦАП.

Чтобы получить нужную частоту синуса на выходе, необходимо на входе конвейера CORDIC каждый такт входной частоты **clk** изменять фазу с определенным алгоритмом. Это будет выполнять модуль **step_control.v** — модуль управления шагом. Чтобы каждый новый такт **clk** на выходе системы появлялся новый отчет синуса, реализуем CORDIC конвейером (последовательная цепочка поворачивающих модулей **rotator.v**). Необходимо сказать, что CORDIC

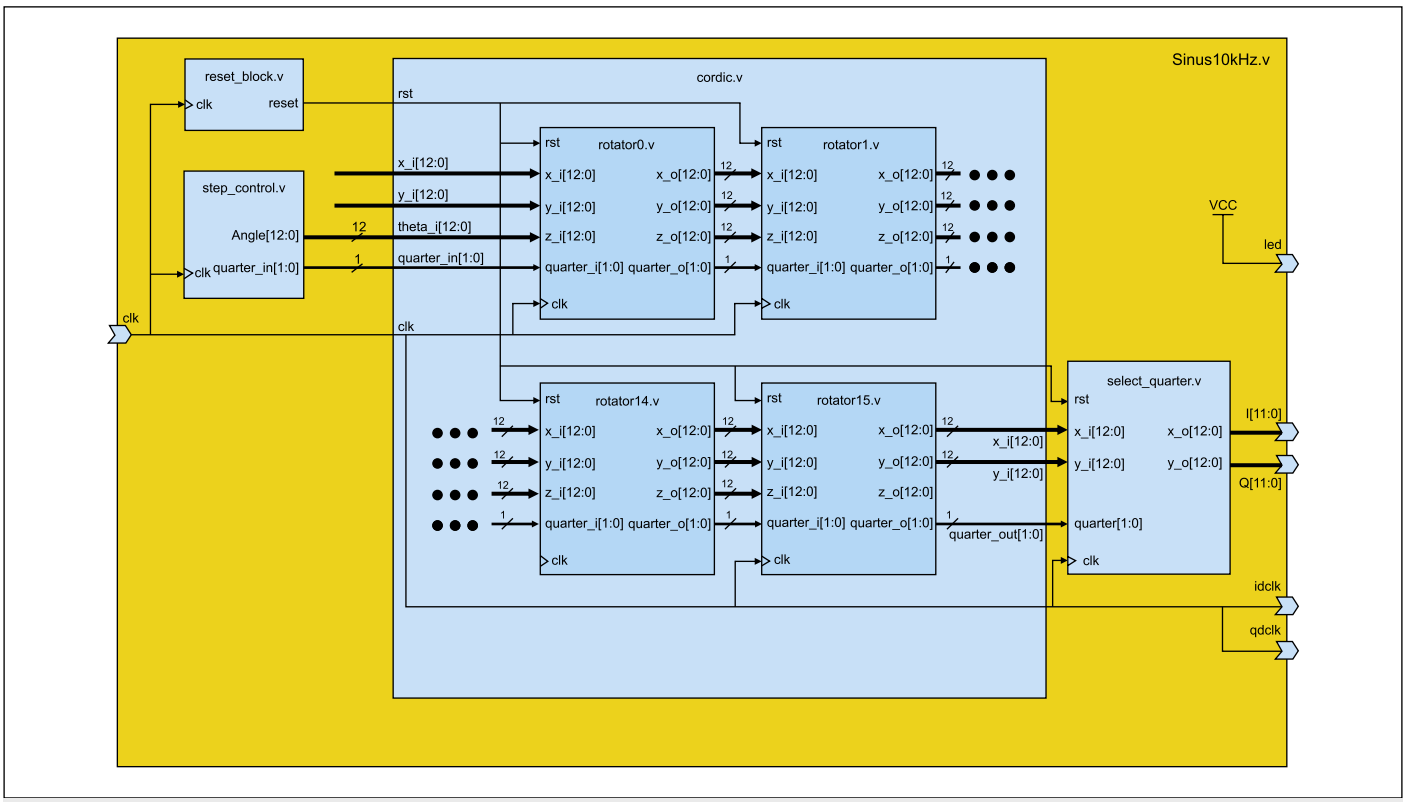

Рис. 7. Структурная схема

вычисляет только первую четверть периода синусоиды. Остальную часть сигнала поможет «дорисовать» модуль **select_quarter.v**.

Рассмотрим иерархию модулей проекта, который формирует синус и косинус для двух 12-разрядных ЦАП.

- **Sinus10kHz.v** — головной модуль, или, как говорят FPGA-дизайнеры, **top-level** модуль. Он содержит в себе все остальные модули, связывает их между собой и имеет следующие входные и выходные сигналы:
  - **input** clk — тактовая частота от PLL выбранной ПЛИС. В нашем проекте — 96 МГц.
  - **output** led1 — выход на светодиод. Показывает наличие питания на ПЛИС.
  - **output** idclk, qdclk — тактовые выходы для каждой ЦАП.
  - **output** [11:0] I, Q — отсчеты для двух ЦАП (синус и косинус соответственно), для формирования квадратурного сигнала.
- **step_control.v** — модуль, формирующий угол (шаг фазы) для вычисления синуса и косинуса, а также четверть, в которой находится выходной сигнал. Напомним, что частота выходных синуса и косинуса должна составлять 10 кГц. Список входных и выходных сигналов:
  - **input** clk — тактовая частота.
  - **output** Angle — шаг фазы.
  - **output** quarter_in — четверть.
- **Cordic.v** — модуль, формирующий конвейер из блоков, вычисляющих промежуточные значения синуса и косинуса (содержит все итерации вычислений). Входные и выходные сигналы модуля:
  - **input** clk — тактовая частота.
  - **input** rst — аппаратный сброс при включении питания.
  - **input** theta_i — фаза, для которой необходимо вычислить синус и косинус.
  - **input** quarter_in — входная четверть сигнала.
  - **output** quarter_out — выходная четверть сигнала.
  - **input** x_i — начальное значение синуса (равно коэффициенту деформации).
  - **input** y_i — начальное значение косинуса (равно нулю).
  - **output** x_o — вычисленное значение синуса.
  - **output** y_o — вычисленное значение косинуса.
  - **output** theta_o — остаточное значение угла (погрешность фазы).
- **select_quarter.v** — модуль, подводящий вычисленное значение синуса и косинуса под весь диапазон ЦАП. Входы и выходы аналогичны рассмотренным выше.
- **reset_block.v** — организует аппаратный сброс и установку всех регистров в начальное значение.
- **rotator.v** — модуль, поворачивающий вектор на заданный угол (реализует каждую из итераций, рассмотренных в математической части статьи).

Теперь начнем анализировать код этих модулей (вспомним каждую из итераций в математической части статьи).

Модуль Sinus10kHz.v

```
module Sinus10kHz (clk, led1, idclk, qdclk, I, Q);

parameter Width_Data = 12;
parameter Width_Angle = 16;
parameter Koef_Mash = 13'h4DB;
parameter Freq_Step = 16'd3400;

input clk; // 96 МГц
output led1; // светодиод
output idclk, qdclk; // тактовые частоты для двух ЦАП
output [Width_Data-1:0] I; // отсчет синуса
output [Width_Data-1:0] Q; // отсчет косинуса
```

В описанной части кода помимо шапки модуля и входных/выходных сигналов приведены также параметры. Поясним их значения:

- **Width_Data** — разрядность ЦАП.
- **Width_Angle** — разрядность вычисляемого угла. Разрядность фазы должна быть больше разрядности выходных отсчетов, чтобы увеличить точность вычислений.
- **Koef_Mash** — коэффициент деформации (см. математическую часть статьи).
- **Freq_Factor** — фактор выходной частоты (если можно так выразиться). Будет подробно пояснен при рассмотрении модуля **step_control.v**.

Так как коэффициент деформации представляет собой вещественное число меньше единицы (0,607), нам необходимо представить его числом размерностью 12 бит:

$$\text{Koef\_Mash} = \\ = 0{,}607(2^{\text{Width\_Data}})/2 = 12\text{'h4DB}.$$

Почему при расчете коэффициента деформации мы взяли половину диапазона, будет пояснено при рассмотрении модуля **cordic.v**.

Продолжим рассмотрение модуля:

```
wire led1, clk;
wire reset;
wire [Width_Data:0] Xi_cordic, Yi_cordic;
wire [Width_Data:0] Xo_cordic, Yo_cordic;
wire [Width_Data:0] Xq, Yq;
wire [Width_Data:0] Angle_i, Angle_o;
wire [1:0] quarter_in, quarter_out;
wire idclk, qdclk;
```

Шины **Xi_cordic** и **Yi_cordic** — это начальное значение X и Y составляющих (x_0 и y_0 в первой итерации математической части статьи). **Xo_cordic** и **Yo_cordic** — это вычисленные значения X и Y составляющих (x_1 и y_1 в последней итерации). Разрядность шин **Xi_cordic**, **Yi_cordic**, **Xo_cordic**, **Yo_cordic**, **Angle_i** и **Angle_o** увеличилась на 1 бит потому, что при вычислениях нам потребуется еще и знаковый разряд (12 бит данных + 1 знаковый бит = 13 бит). Сигнал **quarter_in** — значение четверти выходного сигнала — на входе конвейера CORDIC. Сигнал **quarter_out**, соответственно, — значение четверти выходного сигнала на выходе конвейера:

- 2'b00 — первая четверть (0…π/2);
- 2'b01 — вторая четверть (π/2…π);
- 2'b10 — третья четверть (π…3π/2);
- 2'b11 — четвертая четверть (3π/2…2π).

Сигналы Xq, Yq — отсчеты выходных синуса и косинуса, адаптированные к диапазону значений ЦАП.

Далее:

```
assign led1 = 1'b1;
```

На выход **led1** подведена «лог. 1», чтобы убедиться, что на ПЛИС подано питание.

```
assign I[Width_Data-1:0] = Xq[Width_Data-1:0];
assign Q[Width_Data-1:0] = Yq[Width_Data-1:0];
```

Вычисленные значения синуса и косинуса — на выход ПЛИС, а далее — на логические входы данных ЦАП.

```
assign idclk = clk;
assign qdclk = clk;
```

Входной тактовой частотой тактируются и наши два ЦАП.

```
assign Xi_cordic = Koef_Mash;
assign Yi_cordic = 13'h000;
```

В начале вычислений наш вектор лежит на оси X, и длина его будет равна коэффициенту деформации: это избавит нас от операции умножения в конце вычислений. Итак, на начальное значение синуса подан коэффициент деформации **Koef_Mash**, а на начальное значение косинуса — ноль.

Далее подключаем модули более низкого уровня:

```
reset_block reset_block_user ( .clk(clk), .reset(reset) );
```

Этот блок после подачи питания на ПЛИС удерживает выход **reset** в низком уровне несколько тактов **clk**, что устанавливает в начальное значение модули **rotator.v** и **select_quarter.v**.

Ниже подключен модуль **step_control.v**.

```
    defparam step_control_user.freq_factor = Freq_Factor;
    step_control step_control_user ( .clk(clk), .Angle(Angle_i),
.quarter_in(quarter_in) );
```

На выходе этого модуля каждый такт **clk** появляется новое значение угла (или фазы), для которого необходимо вычислить значение синуса и косинуса. Также на выходе появляется четверть выходного сигнала. Внутрь модуля **step_control.v** передано значение параметра **Freq_Factor**.

Идем далее:

```
defparam cordic_user.width_data = Width_Data;
defparam cordic_user.width_angle = Width_Angle;
cordic cordic_user ( .clk(clk), .rst(reset),
                     .x_i(Xi_cordic), .y_i(Yi_cordic), .theta_i(Angle_i),
                     .x_o(Xo_cordic), .y_o(Yo_cordic), .theta_o(Angle_o),
                     .quarter_in(quarter_in), quarter_out(quarter_out));
```

Здесь подключен модуль **cordic.v**. К его входам и выходам подключены вышеописанные сигналы. Также ему сообщены значения двух параметров.

Затем подключен модуль **select_quarter.v**.

```
select_quarter quarter_user (
        .clk(clk), .rst(reset),
        .Xi(Xo_cordic), .Yi(Yo_cordic),
        .Xo(Xq), .Yo(Yq),
        .quarter (quarter_out)
        );
```

На входы **Xi** и **Yi** этого модуля подключены вычисленные CORDIC-алгоритмом значения синуса и косинуса. На выходах **Xo** и **Yo** с каждым тактом **clk** появляются, в зависимости от входной четверти **quarter**, отсчеты синуса и косинуса для ЦАП.

Так, с **Sinus10kHz.v** закончили, переходим к модулям более низкого уровня.

Модуль reset_block.v

Шапка, входы и выходы модуля аппаратного сброса **reset_block.v**:

```
module reset_block (
                    clk, reset
                    );
input clk;
output reset;
```

Нет смысла объяснять назначение одного тактового входа и одного выхода: это понятно по названию. Перейдем сразу к логике:

```
reg [3:0] count_reset = 4'h0;
reg reset;

always @ (posedge clk)
begin
    if (count_reset<=4'd10) begin
        reset<=1'b1;
        count_reset<=count_reset+1'b1; end
    else begin
        reset<=1'b0;
        count_reset<=4'd15; end
end
```

Здесь видно, что после подачи питания на ПЛИС в данном модуле производится инкремент счетчика **count_reset**. Пока значение этого счетчика меньше 10, сигнал **reset** находится в высоком состоянии. Соответственно, после 10 тактов **clk** значение сигнала сброса **reset** переходит в состояние логической единицы, а счетчик **count_reset** фиксируется в значении 15. Таким образом, можно сказать, что в течение первых 10 периодов тактового сигнала **clk** остальные модули проекта должны при значении сигнала **reset**, равном единице, установить свои регистры в начальное состояние.

Модуль step_control.v

Шапка, входы/выходы и параметры модуля:

```
module step_control ( clk, Angle, quarter_in );

parameter first = 0;
parameter second = 1;
parameter third = 2;
parameter fourth = 3;
parameter freq_factor = 16'd3400;
parameter acc_decr = 16'd10000 - freq_step;

input clk;
output reg [12:0] Angle = 13'd0;
output reg [1:0] quarter_in = 2'b00;
```

Параметры **first**, **second**, **third** и **fourth** — это состояния конечного автомата, который мы рассмотрим ниже.

Параметр **freq_factor** мы уже упоминали, а вот с назначением параметра **acc_decr** необходимо разобраться.

Назначением данного модуля является изменение значения фазы **Angle** каждый такт **clk** на такое значение, при котором при заданной тактовой частоте в 96 МГц на выходе ПЛИС мы получаем цифровой синус с частотой 10 кГц.

Значение **Angle** в 3216 соответствует фазе в 90° (π/2). Допустим, если каждый такт **clk** мы будем увеличивать **Angle** на 1, то на выходе мы получим синус с частотой $96\times10^6/3216/4 = 7462{,}68$ Гц (на 4 поделено, чтобы получить полный период, а не только до 90°). Назовем эту частоту типовой.

Далее, если мы хотим получить на выходе частоту, которая больше типовой в 1,2 раза, то есть 8955 Гц, то нужно **Angle** также увеличивать каждый такт **clk** на 1, но каждый пятый такт — на 2 (потому что частота больше типовой на 1/5 часть). А как же быть, если нужная нам частота (10 кГц) больше типовой в 1,34 раза? Опять вернемся к частоте 8,955 кГц. Возьмем некий счетчик Acc

разрядностью 16 бит и будем каждый такт прибавлять к нему значение 16'd2000. Когда значение счетчика перевалит за 10 000 (а это будет происходить раз в 5 тактов), значение **Angle** нужно будет увеличить не на 1, а на 2. Следовательно, чтобы получить на выходе частоту в 10 кГц, следует каждый такт **clk** увеличивать Acc на 3400 (так как нужная нам частота больше типовой в 1,34 раза). Вот откуда взялось значение фактора выходной частоты **Freq_Factor**, о котором мы упоминали выше. В связи с этим давайте напишем код, который соответствует изложенному алгоритму:

```
reg [15:0] M = freq_factor; // 16'd3400
reg ena_incr = 1'b0;
reg [15:0] Acc = 16'd0;

always @ (posedge clk)
begin
if (reset_acc == 1'b1) Acc <= 16'd0;
   else
      if (Acc >= 16'd10000)
      begin
         ena_incr <= 1'b1;
      Acc <= Acc - acc_decr;
      end
   else
      begin
         ena_incr <= 1'b0;
         Acc <= Acc + M;
      end
end
```

Следует добавить только три момента:

- Сигнал **ena_incr** при '1' позволяет увеличить **Angle** на 2 (как мы увидим позже), а при '0' — увеличивает на 1.
- Условие **Acc >= 16'd10000** выполняется так часто, как указано в выше описанном алгоритме для частоты 10 кГц.
- По сигналу **reset_acc = 1'b1** счетчик-аккумулятор **Acc** обнуляется (это будет происходить на рубеже переключения состояний автомата).

А теперь рассмотрим конечный автомат, реализующий инкремент/декремент фазы, а также переключение четвертей.

```
always @ (posedge clk)
begin
   case (state)
      first:    //состояние первой четверти периода синусоиды
         <.....>
      second: //состояние второй четверти периода синусоиды
         <.....>
      third:    //состояние третьей четверти периода синусоиды
         <.....>
      fourth:  //состояние четвертой четверти периода синусоиды
         <.....>
   endcase
end
```

Процесс, описывающий конечный автомат, довольно громоздкий, чтобы приводить его полностью. Поэтому приведем код лишь первого (**first**) состояния автомата:

```
first:
   if (count_ang >= 12'd3216) begin
      state <= second;
      count_ang <= 12'd0;
      quarter <= 2'b01;
   end
   else begin
      state <= first;
      quarter_in <= 2'b00;
      if (ena_incr) begin
         Angle <= Angle + 12'd2;
         count_ang <= count_ang+12'd2; end
   else begin
      Angle <= Angle + 12'd1;
      count_ang <= count_ang+12'd1; end
end
```

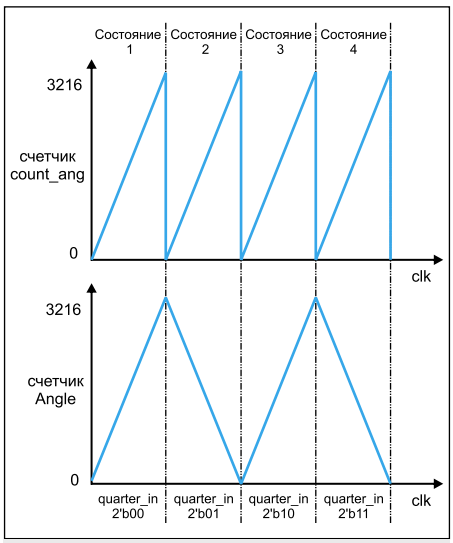

Рис. 8. Принцип работы конечного автомата — инкремент/декремент фазы

Здесь 12-разрядный счетчик **count_ang** с каждым тактом **clk** ведет счет от 0 до 3216 в каждом состоянии (каждой четверти). 13-разрядный счетчик **Angle** содержит именно ту фазу, для которой необходимо рассчитать значение синуса и косинуса. **Angle** подключен на вход конвейера, рассчитывающего синус и косинус данной фазы. **count_ang** и **Angle** практически идентичны, за исключением того, что первый во всех состояниях только увеличивается, а второй увеличивается/уменьшается в зависимости от состояния автомата. Если сигнал **ena_incr** находится в низком состоянии, то **Angle** и **count_ang** увеличивают свои состояния на 1, а если в высоком — то на 2. Таким образом обеспечивается нужный период выходного синуса в 10 кГц. В каждом состоянии контролируется момент перехода на следующее состояние. Когда счетчик **count_ang** достигает значения 3216 (соответствует фазе 90°), на следующий такт **clk** автомат перейдет во второе состояние (*state <= second*).

Далее рассмотрим отличие остальных состояний автомата от первого. Во втором и четвертом состоянии осуществляется не инкремент счетчика **Angle** (как в первом состоянии), а декремент. Третье состояние идентично первому.

Думаем, читателю не нужно объяснять, какое значение сигнала **quarter** и **state** нужно передавать в каждом состоянии.

Ниже описан процесс, который разрешает сброс счетчика Acc (reset_acc <= 1'b1).

```
always @ (posedge clk)
begin
      if (count_ang == 12'd3216) reset_acc <= 1'b1;
      else reset_acc <= 1'b0;
end
```

Далее рассмотрим так называемое «ядро» проекта — конвейер, реализующий все итерации расчета значений синуса и косинуса для какого-то угла.

Модуль cordic.v

Рассмотрим шапку модуля, входы/выходы и параметры:

```
module cordic (
   clk, rst,
   x_i, y_i, theta_i,
   x_o, y_o, theta_o,
   quarter_in, quarter_out
);

parameter width_data = 12;
parameter width_angle = 16;

   input wire clk;
   input wire rst;

   input wire signed [width_data:0] x_i;
   input wire signed [width_data:0] y_i;
   input wire signed [width_data:0] theta_i; //вычисляемая фаза

   output wire signed [width_data:0] x_o;
   output wire signed [width_data:0] y_o;
   output wire signed [width_data:0] theta_o; //остаточное значение фазы (погрешность)
```

В параметрах упомянуты размерности регистров для синуса и косинуса (width_data), а также размерность регистров для фазы (width_angle). Напомним, что размерность фазы больше размерности синуса и косинуса, для более точного вычисления. Назначение входов и выходов модуля мы объяснили ранее.

Рассмотрим функцию, которая выдает для каждого элемента конвейера свое значение арктангенса.

```
function [width_angle:0] tanangle;
      input [3:0] i;
      begin
         case (i)
         4'b0000: tanangle = 17'd25736;      // 1/1
         4'b0001: tanangle = 17'd15192;      // 1/2
         4'b0010: tanangle = 17'd8028;       // 1/4
         4'b0011: tanangle = 17'd4074;       // 1/8
         4'b0100: tanangle = 17'd2045;       // 1/16
         4'b0101: tanangle = 17'd1024;       // 1/32
         4'b0110: tanangle = 17'd512;        // 1/64
         4'b0111: tanangle = 17'd256;        // 1/128
         4'b1000: tanangle = 17'd128;        // 1/256
         4'b1001: tanangle = 17'd64;         // 1/512
         4'b1010: tanangle = 17'd32;         // 1/1024
         4'b1011: tanangle = 17'd16;         // 1/2048
         4'b1100: tanangle = 17'd8;          // 1/4096
         4'b1101: tanangle = 17'd4;          // 1/8192
         4'b1110: tanangle = 17'd2;          // 1/16384
         4'b1111: tanangle = 17'd1;          // 1/32768
         endcase
      end
endfunction
```

Вещественное значение арктангенса необходимо хранить в 17-разрядных регистрах. Читатель может вполне справедливо задать вопрос: «А почему такие странные значения арктангенсов?» Вернемся опять ненадолго к математике. Приведем значения арктангенсов для всех 16 блоков конвейера:

- $\mathrm{arctg}(1) = 45° \approx 0{,}785$ рад;
- $\mathrm{arctg}(2^{-1}) = 26{,}6° \approx 0{,}464$ рад;
- $\mathrm{arctg}(2^{-2}) = 14{,}04° \approx 0{,}245$ рад;
- $\mathrm{arctg}(2^{-3}) = 7{,}92° \approx 0{,}1244$ рад;
- $\mathrm{arctg}(2^{-4}) = 3{,}97° \approx 0{,}0624$ рад;
- $\mathrm{arctg}(2^{-5}) = 1{,}99° \approx 0{,}0312$ рад;

- arctg(2^{-6}) = 0,99° ≈ 0,0156 рад;
- arctg(2^{-7}) = 0,49° ≈ 7,8×10^{-3} рад;
- arctg(2^{-8}) = 0,22° ≈ 3,9×10^{-3} рад;
- arctg(2^{-9}) = 0,11° ≈ 1,95×10^{-3} рад;
- arctg(2^{-10}) = 0,06° ≈ 9,8×10^{-4} рад;
- arctg(2^{-11}) = 0,03° ≈ 4,9×10^{-4} рад;
- arctg(2^{-12}) = 0,013° ≈ 2,4×10^{-4} рад;
- arctg(2^{-13}) = 0,007° ≈ 1,2×10^{-4} рад;
- arctg(2^{-14}) = 0,003° ≈ 6×10^{-5} рад;
- arctg(2^{-15}) = 0,001° ≈ 3×10^{-5} рад.

Теперь уже можно сказать, что вычисленное значение синуса и косинуса не будет превышать ошибки в 0,001°. Но нам нужно эти нецелые значения перевести в цифровой вид, чтобы можно было их хранить в регистрах. Вспомним, что разрядность угла мы выбрали на 4 разряда больше, чем размерность регистров данных (X и Y). Значит, диапазон принимаемых значений 16-разрядного регистра — 0…216 = 0…65 535. Но мы будем использовать только половину этого диапазона (0…32 767), чтобы подогнать вычисленные отсчеты под весь диапазон ЦАП. (Подробнее об этом будет рассказано при рассмотрении модуля **select_quarter.v**.) Переведем значение 0,785 рад (arctg(1)) в значение для 16-разрядного регистра: 32 768×0,785 = 25 736. Таким же образом значения остальных арктангенсов читатель может посчитать сам.

Чтобы связать все элементы конвейера, создадим массивы из цепей:

```
wire signed [width_data:0] x [width_angle:0];
wire signed [width_data:0] y [width_angle:0];
wire signed [width_angle:0] z [width_angle:0];
wire [1:0] q [width_angle:0];
```

Подключим входные и выходные данные к крайним регистрам этих массивов:

```
assign x[0] = x_i;
assign y[0] = y_i;
assign q[0] = quarter_in;
assign x_o = x[width_angle];
assign y_o = y[width_angle];
assign quarter_out = q[width_angle];
```

А о подключении значений углов расскажем подробнее:

```
wire [width_angle:0] inbuf_ang;
wire [width_angle:0] outbuf_ang;

assign inbuf_ang[width_angle] = 1'b0;
assign inbuf_ang[width_angle-1:width_angle-width_data] =
theta_i[width_data-1:0];
assign inbuf_ang[3:0] = 4'h0;
assign z[0] = inbuf_ang;

assign outbuf_ang = z[width_angle];
assign theta_o = outbuf_ang[width_data-1];
```

Так как разрядность угла нам нужно увеличить с 12 до 16, мы использовали буфер **inbuf_ang[16:0]**, в котором старший бит — знаковый, следующие 12 бит — угол, для которого необходимо в итоге рассчитать синус и косинус, а младшие 4 бита заполняем нулями. Значение угла (**z[width_angle]**) на выходе конвейера является остаточным значением угла (характеризует погрешность вычислений) и подключается на выход модуля **cordic.v**. Перейдем к подключению модулей **rotator.v** (вспомним: именно этот модуль обеспечивает элементарный поворот вектора — каждую из итераций). Как было видно на структурной схеме, количество модулей (итераций) **rotator.v** должно быть равно 16. Это можно сделать обычным способом: банально перечислить все модули, но в статье мы приведем только первый (0-й) и последний (15-й):

```
defparam U0.iteration = 0;
defparam U0.tangle = tanangle(0);
defparam U0.width_data = width_data;
defparam U0.width_angle = width_angle;
rotator U0 (
      .clk(clk), .rst(rst),
      .x_i(x[0]), .y_i(y[0]), .z_i(z[0]),
      .x_o(x[1]), .y_o(y[1]), .z_o(z[1]),
      .quarter_i(q[0]), .quarter_out(q[1])
            );

defparam U15.iteration = 15;
defparam U15.tangle = tanangle(15);
defparam U15.width_data = width_data;
defparam U15.width_angle = width_angle;
rotator U15 (
      .clk(clk), .rst(rst),
      .x_i(x[15]), .y_i(y[15]), .z_i(z[15]),
      .x_o(x[16]), .y_o(y[16]), .z_o(z[16]),
      .quarter_i(q[15]), .quarter_out(q[16])
            );
```

Давайте не будем пока обращать внимание на подключение к модулям **rotator.v** таких параметров, как **iteration** и **tangle**, а согласимся, что подключать таким образом одни и те же модули, да еще и в таком количестве, как минимум не практично. А если в процессе разработки вы будете варьировать количество итераций? Гораздо красивее и компактнее выглядит такая конструкция:

```
genvar i;
generate for(i=0; i<width_angle; i=i+1) begin : rot
rotator U (
            .clk(clk), .rst(rst),
            .x_i(x[0]), .y_i(y[0]), .z_i(z[0]),
            .x_o(x[1]), .y_o(y[1]), .z_o(z[1]),
            .quarter_i(q[0]), .quarter_out(q[1])
);
      defparam U.iteration = i;
      defparam U.tangle = tanangle(i);
      defparam U.width_data = width_data;
      defparam U.width_angle = width_angle;
end
endgenerate
```

Здесь с помощью переменной i и конструкции **generate for** автоматически создаются все 16 экземпляров модуля **rotator.v**. Эти экземпляры соединены друг с другом массивами из цепей, которые были упомянуты выше. Каждому экземпляру переданы значения известных нам параметров (**width_data**, **width_angle**) и доселе неизвестных, таких как:

- **iteration** — количество разрядов (для каждой итерации свой), на которые нужно сдвинуть некий регистр данных. Вспомним математическую часть статьи, в которой упоминалось, что выражение «x×2^{-i}» можно реализовать путем сдвига регистра x на i разрядов вправо.
- **tangle** — значение арктангенса угла (для каждой итерации свой).

Далее рассмотрим модуль, который реализует каждую из итераций.

Модуль rotator.v

Шапка модуля, параметры, входные и выходные порты:

```
module rotator (
            clk, rst,
            x_i, y_i, z_i,
            x_o, y_o, z_o,
            quarter_i, quarter_o
      );

      parameter width_data = 12;
      parameter width_angle = 16;
      parameter integer iteration = 0;
      parameter signed [width_angle:0] tangle = 0;

      input wire clk;
      input wire rst;

      input wire signed [width_data:0] x_i;
      input wire signed [width_data:0] y_i;
      input wire signed [width_angle:0] z_i;
      output wire signed [width_data:0] x_o;
      output wire signed [width_data:0] y_o;
      output wire signed [width_angle:0] z_o;

      input wire [1:0] quarter_i;
      output reg [1:0] quarter_o;
```

Назначение входов и выходов модуля, думаем, понятно (видно на структурной схеме проекта), а параметры были описаны выше, в модуле **cordic.v**.

Теперь давайте напишем функцию, которая бы сдвигала регистр на заданное количество разрядов вправо. При этом не будем принимать во внимание знаковый, старший бит числа (знаковый бит не должен сдвигаться):

```
function signed [width_data:0] Delta;
   input signed [width_data:0] Arg;
   input integer cnt;
   integer k;
   begin
      Delta = Arg;
      for (k=0;k<cnt;k=k+1)
         begin
         Delta[width_data-1:0]=Delta[width_data:1];
         Delta[width_data]=Arg[width_data];
         end
   end
endfunction
```

Здесь **Delta** — имя функции, которая возвращает 13-разрядное число ([width_data:0]). **Arg** и **cnt** — аргументы функции, первый из которых передает регистр, а второй — число, на которое этот регистр необходимо сдвинуть.

Вот и добрались мы до самого ядра CORDIC-алгоритма. Вкупе с уже рассмотренными функциями арктангенса и сдвига нам осталось реализовать только операции сдвига и сложения/вычитания:

```
wire signed [width_data:0] Xd, Yd;
assign Xd = Delta(x_i,iteration);
assign Yd = Delta(y_i,iteration);

   always @ (posedge clk)
      if (rst) begin
         x_1 <= 0;
         y_1 <= 0;
         z_1 <= 0;
         end
      else begin
         if (z_i < 0) begin
            x_1 <= x_i + Yd;
            y_1 <= y_i – Xd;
            z_1 <= z_i + tangle; end
         else begin
            x_1 <= x_i – Yd;
            y_1 <= y_i + Xd;
            z_1 <= z_i – tangle; end
         end
```

Здесь сигналы **Xd** и **Yd** содержат результат сдвига входных сигналов **x_i** и **y_i** на количество разрядов **i**. После фронта тактового сигнала **clk** проводится проверка знака угла **z_i**, что определяет направление поворота вектора. Нет необходимости разбирать этот процесс более подробно, стоит лишь напомнить, что он реализует вычисления, приведенные в математической части статьи. Приведем их снова:

$$\sigma_i = \text{sign}(z_i),$$

$$x_{i+1} = x_i - \sigma_i \times y_i \times 2^{-i},$$

$$y_{i+1} = y_i + \sigma_i \times x_i \times 2^{-i},$$

$$z_{i+1} = z_i - \sigma_i \times \text{atan}(z^{-i}).$$

Вычисленные значения **x_1**, **y_1** и **z_1** передаем на выход модуля — для следующей итерации:

```
assign x_o = x_1;
assign y_o = y_1;
assign z_o = z_1;
```

Также, параллельно с углом и данными, необходимо передавать по итерациям четверть сигнала: для последнего модуля нашего проекта — **select_quarter.v**.

```
always @ (posedge clk)
  if (rst) quarter_o <= 1'b0;
  else quarter_o[1:0] <= quarter_i[1:0];
```

Модуль select_quarter.v

Напомним, что этот модуль предназначен для адекватного представления данных на ЦАП. На выходе модуля **cordic.v** X и Y изменяют свои значения от 0 до 2048 и по форме представляют собой модуль синуса (рис. 9). Диапазон входных данных ЦАП может принимать значения 0–4096.

На рис. 9 видно: для того чтобы получить полноценный синус для выдачи на ЦАП, в первые две четверти к входному сигналу x_i нужно прибавлять значение 2048 (середина диапазона ЦАП), а в последние 3-ю и 4-ю четверть — отнимать значение 2048.

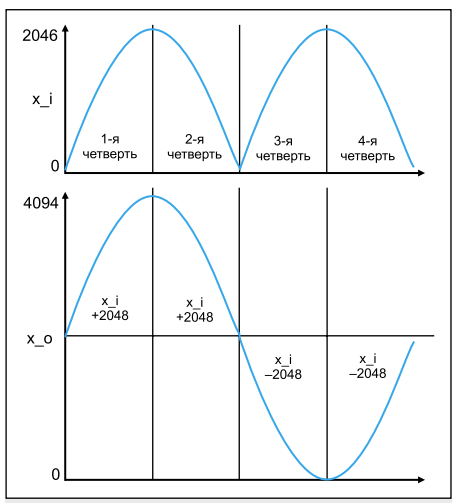

Рис. 9. Назначение модуля select_quarter.v

Рассмотрим шапку модуля, входные и выходные сигналы:

```
module select_quarter (
    input wire clk,
    input wire rst,
    input wire [12:0] Xi,
    input wire [12:0] Yi,
    output wire [12:0] Xo,
    output wire [12:0] Yo,
    input [1:0] quarter
);
```

Далее реализуем то, что показано на рис. 9:

```
wire [12:0] Xq1, Xq2;
wire [12:0] Yq1, Yq2;

assign Xq1 = Xi+13'h800;
assign Yq1 = Yi+13'h800;
assign Xq2 = 13'h800-Xi;
assign Yq2 = 13'h800-Yi;
```

Здесь значение 2046 представлено в шестнадцатеричном виде — 13'h800. Далее реализуем процесс, который по тактовому сигналу **clk**, в зависимости от четверти, присваивает сигналу **Xresult** значение **Xq1** либо **Xq2**, а сигналу **Yresult** — значение **Yq1** либо **Yq2**:

```
reg [12:0] Xresult, Yresult;

always @ (posedge clk)
begin
  if (rst) begin
      Xresult <= 13'h0; Yresult <= 13'h0;
      end
  else begin
      case (quarter)
          2'b00:begin
              Yresult <= Yq1; Xresult <= Xq1;
          end
          2'b01: begin
              Yresult <= Yq1; Xresult <= Xq2;
          end
          2'b10: begin
              Yresult <= Yq2; Xresult <= Xq2;
          end
          2'b11: begin
              Yresult <= Yq2; Xresult <= Xq1;
          end
      endcase
  end
end
```

В заключение подключим результаты на выход модуля **select_quarter.v**:

```
assign Xo = Xresult;
assign Yo = Yresult;
```

На этом мы закончили рассмотрение всех модулей. Теперь перейдем к созданию проекта в ModelSim, а также к моделированию.

Моделирование в ModelSim

Создавать HDL-проект мы будем в популярной среде для отладки и симулирования ПЛИС — ModelSim от Mentor Graphics. В пользу выбора данного ПО можно, помимо прочего, добавить тот факт, что среда проектирования FPGA-систем Quartus II от Altera Corporation начиная с версии 10.0 не содержит классический симулятор. Вместо классического симулятора в Quartus II необходимо интегрировать либо ModelSim AE (Altera Edition), либо ModelSim ASE (Altera Starter Edition). Выберем второй вариант, так как для использования первого необходима платная лицензия. Будем использовать ModelSim-Altera 6.6c (Quartus II 10.1) Starter Edition.

Запускаем ПО: **Пуск → Программы → Altera → ModelSim-Altera 6.6c (Quartus II 10.1) Starter Edition**. Создаем проект (рис. 10): в ниспадающем меню **File → New → Project**.

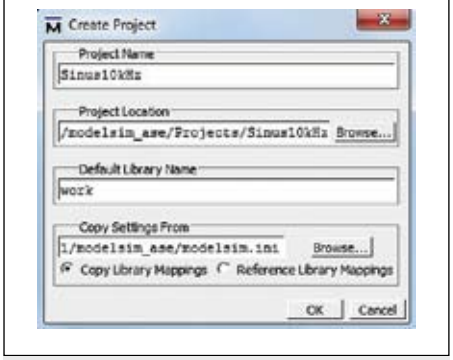

Рис. 10. Окно создания проекта

В поле **Project Name** (рис. 10) прописываем имя нашего проекта: назовем его Sinus10kHz, так как наша система будет генерировать синусоидальный сигнал частотой 10 кГц. В поле **Project Location** указываем место расположения проекта. Автор свои проекты располагает в папке *Projects*, как правило, в том же каталоге, что и программа, в которой создается проект. Поэтому путь к этому каталогу пропишем: *C:\Altera\10.1\modelsim_ase\Projects\Sinus10kHz*. Поле **Default Library Name** рекомендуется оставить по умолчанию.

После нажатия **OK** появится следующее окно: **Add items to the Projects** (рис. 11).

Здесь мы выбираем пункт **Create New File**, который позволит добавить к проекту файлы модулей, написанных на языке описания аппаратуры Verilog (при этом в пункте **Add file as type** выбираем соответствующий язык).

Рис. 11. Окно добавления файлов

Помимо головного модуля **Sinus10kHz.v** добавляем также следующие пустые файлы:

- *reset_block.v*;
- *step_control.v*;
- *cordic.v*;
- *select_quarter.v*;
- *rotator.v*.

Назначение этих модулей мы рассмотрели ранее, при составлении структурной схемы проекта.

Теперь для моделирования нашего проекта необходимо создать модуль testbench. Вспомним, что у головного модуля **Sinus10kHz.v** один вход — тактовый, а значит, чтобы проверить работоспособность системы, необходимо в testbench-модуле сгенерировать лишь этот тактовый сигнал нужной нам частоты — 96 МГц.

Добавим в наш проект еще один модуль: в ниспадающем меню **Project → Add to Project → New File**. Дадим имя модулю — test_bench.

Шапка модуля:

```
`timescale 1ns/1ps;
module test_bench ();
```

Этот модуль не содержит входных и выходных портов. Директивой ***timescale*** задаем размерность временных задержек, необходимых при моделировании. По умолчанию значение директивы ***timescale*** равно 1ns/1ns. Но в тестбенче нам необходимо будет задавать длительность полупериода тактовой частоты 96 МГц, равную 5,208 нс. Первое значение директивы ***timescale*** — это единица времени, в которой будут указываться временные задержки в тестбенче, а второе — разрешение по времени. Указав значение директивы ***timescale*** равным 1ns/1ps, мы сможем прописывать временные задержки, имеющие не целое значение.

Теперь подключим головной модуль нашего проекта:

```
reg clk;
wire led1;
   wire [11:0] I, Q;
   wire idclk, qdclk;

   Sinus10kHz Sinus10kHz_user (
                                .clk(clk), .led1(led1),
                                .idclk(idclk), .qdclk(qdclk),
                                .I(I), .Q(Q)
                                );
```

Наш ***testbench*** будет наипростейшим: нам необходимо лишь подать тактовую частоту 96 МГц на тактовый вход модуля **Sinus10kHz.v**:

```
initial
begin
    clk = 0;
    forever #5.208 clk = !clk;
end
```

Здесь мы постоянно генерируем тактовый меандр с полупериодом 5208 пс, что соответствует частоте 96 МГц. Напомним, что оператор forever — не синтезируемый, полезен только при моделировании. Нет смысла использовать его, например, в САПР Quartus II от Altera.

Теперь наш проект необходимо скомпилировать (рис. 12): **Compile → Compile Order**.

Нам нужно скомпилировать файлы проекта именно согласно иерархии — сверху вниз: начиная с файла верхнего уровня *test_bench.v* и заканчивая *rotator.v*. Доверимся ModelSim и в появившемся окне нажимаем на **Auto Generate**. При удачной компиляции в панели **Transcript** должно появиться сообщение:

```
# Compile of <файл модуля> was successful.
```

Далее необходимо запустить симулятор: **Simulate → Start Simulation**.

Рис. 12. Компиляция проекта

В появившемся окне **Start Simulation** (рис. 13) выбираем рабочую библиотеку (при создании проекта имя библиотеки оставили по умолчанию — work), указываем модуль верхнего уровня — **test_bench.v** и нажимаем OK. При удачной загрузке симулятора на панели **Instance** появится еще одна вкладка — **Sim**. Далее открываем окно временных диаграмм: **View → Wave**. В это окно нам нужно добавить сигналы из проекта, которые мы хотим проанализировать: панель **Instance**, вкладка **Sim**, правая кнопка мыши по **test_bench → Add → To Wave → All items in region**. При этом в окно **Wave** добавятся только те сигналы, которые были указаны в модуле **test_bench.v**. Если мы хотим добавить все сигналы, существующие в проекте, то выбираем **All items design** (рис. 14).

Теперь осталось лишь запустить симуляцию на нужное нам время. Для этого в консоли (панель **Transcript**) пропишем команду:

```
run 200 us
```

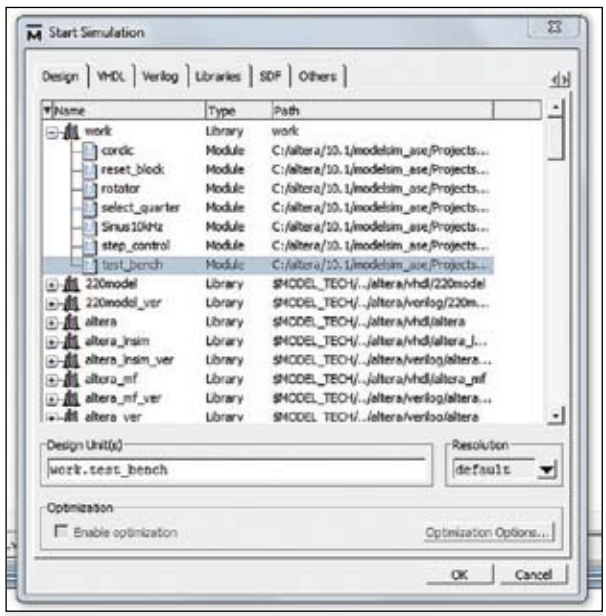

Рис. 13. Запуск симулятора

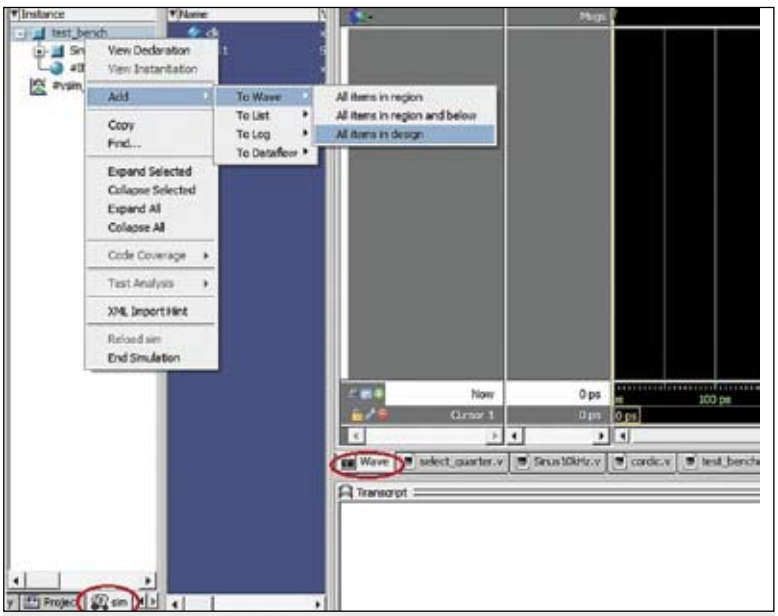

Рис. 14. Добавление сигналов в окно Wave

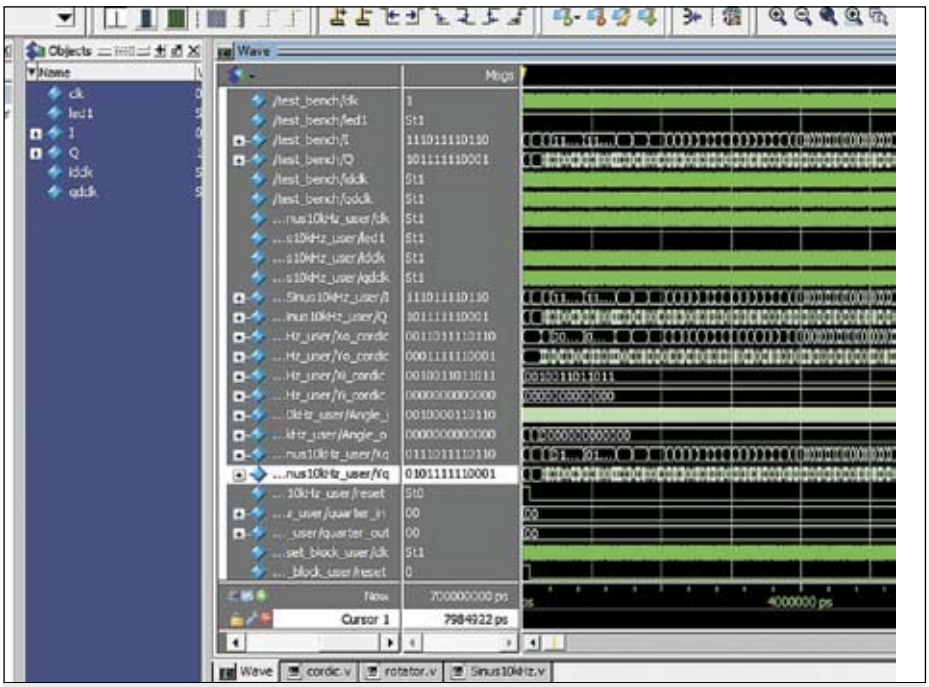

Рис. 15. Результат симуляции

После этого в окне **Wave** мы увидим результат моделирования системы продолжительностью 200 мкс.

Как может заметить читатель, последовательность действий, начиная с компилирования проекта и заканчивая запуском симуляции,— процесс долгий. Всю эту последовательность действий можно прописать в do-файле. Создадим его: **File → New → Source → Do**. Прописываем в do-файле следующий текст:

```
vlog test_bench.v Sinus10kHz.v cordic.v step_control.v reset_block.v
select_quarter.v rotator.v
vsim work.test_bench
add wave -r /*
run 200 us
```

В первой строке командой vlog и списком модулей мы компилируем наш проект. Во второй строке командой vsim мы загружаем симулятор. В третье строке мы открываем окно **Wave** и добавляем туда все существующие в нашем проекте сигналы. В последней строке мы запускаем симулятор на 200 мкс.

Сохраняем наш do-файл под именем, например, «*make.do*» в директорию проекта. Теперь, чтобы скомпилировать и просимулировать наш проект, достаточно в консоли набрать следующее:

```
do make.do
```

и нажать **Enter**. В консоли можно будет проследить исполнение всех команд, прописанных в do-файле. В появившемся окне **Wave** мы увидим примерно такую картину (рис. 15).

В этом окне будут представлены все сигналы, присутствующие в нашем проекте. Это все хорошо, только отметим пару минусов:

- Сигналы раскиданы как попало, и придется пролистывать все окно **Wave**, чтобы найти нужный нам сигнал.
- Такие сигналы, как, например, x_i, y_i из модуля **rotator.v**, лучше воспринимаются в аналоговой форме, а не в виде цифр.

Но можно окно **Wave** оформить должным образом: так, чтобы отображались только нужные нам сигналы и в удобной для нас форме. Вся эта информация прописывается все в том же do-файле. Конечно, придется потратить время на оформление окна **Wave**, но зато с красиво оформленными временными диаграммами удобнее отлаживать код проекта, да и просто комфортнее воспринимается информация.

Теперь начнем дописывать имеющийся у нас файл *make.do*. Оформление окна **Wave** нужно описывать после команды запуска симулятора — vsim work.test_bench, до команды run 200 us и, само собой, вместо команды добавления всех сигналов — add wave -r/*.

Добавляем тактовый сигнал и сигнал сброса:

```
add wave -noupdate -format Logic /test_bench/clk
add wave -noupdate -format Logic \
        /test_bench/Sinus10kHz_user/reset_block_user/reset
```

Тут мы указываем, с какого модуля и какой конкретно сигнал необходимо добавить в окно отладки **Wave**. Команды ***add wave*** можно писать одной строкой, но часто они получаются слишком длинными, в таких случаях применяют знак «\»: он позволяет перенести часть команды на следующую строку do-файла. Далее:

```
add wave -noupdate -group Top_Module_Sinus10kHz -color Cyan
-format Logic \
    {/test_bench/Sinus10kHz_user/idclk}

add wave -noupdate -group Top_Module_Sinus10kHz -color Cyan
-format Analog-Step \
    -height 150 -max 4092.0 -radix unsigned /test_bench/Sinus10kHz_
user/I

add wave -noupdate -group Top_Module_Sinus10kHz -color {Steel
Blue} -format Logic \
    {/test_bench/Sinus10kHz_user/qdclk}

add wave -noupdate -group Top_Module_Sinus10kHz -color {Steel
Blue} -format Analog-Step \
    -height 150 -max 4092.0 -radix unsigned /test_bench/Sinus10kHz_
user/Q

add wave -noupdate -group Top_Module_Sinus10kHz -color Green
-format Analog-Step \
    -height 50 -max 3216.0 -radix unsigned /test_bench/Sinus10kHz_
user/step_user/Angle

add wave -noupdate -group Top_Module_Sinus10kHz -color Green
-format Logic \
    {/test_bench/Sinus10kHz_user/step_user/quarter_in}
```

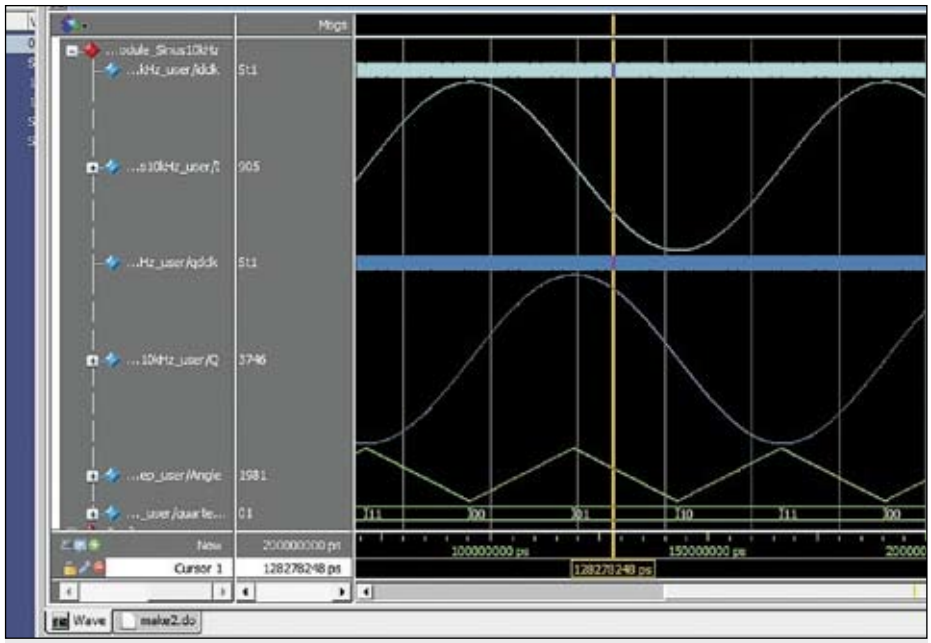

Рис. 16. Аналоговая синусоида

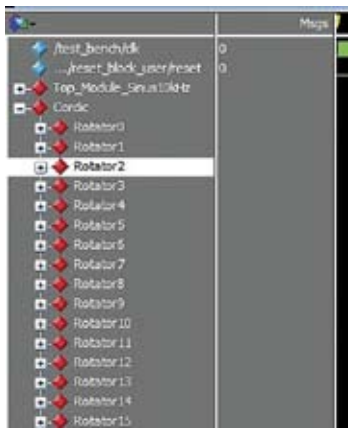

Рис. 17. Добавление всех звеньев конвейера

В выше написанном коде мы добавляем в отладочное окно сигналы тактирования обоих ЦАП — idclk, qdclk. Помимо этого мы добавляем сигналы I и Q — отсчеты квадратурного сигнала этих же ЦАП, а также Angle. Причем отображаться в окне эти сигналы будут в аналоговой форме (-**format** Analog-**Step**). Задаем высоту отображения сигнала (-**height** 150) и диапазон принимаемых значений сигнала (-**max** 4092). Заметим, что добавленные сигналы раскрашены в разные цвета (-**color** Cyan). Эти сигналы объединены в группу Top_Module_Sinus10kHz.

Забегая вперед, покажем читателю, как будет выглядеть окно **Wave** (рис. 16) после запуска do-файла, содержащего команды, которые мы только что рассмотрели.

Так как CORDIC-алгоритм реализован в виде конвейера, при симуляции нам интересно было бы видеть, как изменяются значения X и Y после каждого звена конвейера (после каждого модуля **rotator.v**).

Приведем команды добавления сигналов в окно **Wave** для первого звена конвейера:

```
#Rotator 0
add wave -noupdate -expand -group Cordic -group Rotator0 -color
Green -format Logic \
{/test_bench/Sinus10kHz_user/cordic_user/rot[0]/U/z_i}

add wave -noupdate -expand -group Cordic -group Rotator0 -color
Green -format Analog-Step \
-height 50 -max 2048.0 {/test_bench/Sinus10kHz_user/cordic_user/
rot[0]/U/x_i}

add wave -noupdate -expand -group Cordic -group Rotator0 -color
Green -format Logic \
{/test_bench/Sinus10kHz_user/cordic_user/rot[0]/U/y_i}

add wave -noupdate -expand -group Cordic -group Rotator0 -color
Green -format Logic \
{/test_bench/Sinus10kHz_user/cordic_user/rot[0]/U/z_o}

add wave -noupdate -expand -group Cordic -group Rotator0 -color
Green -format Analog-Step \
-height 50 -max 2048.0 {/test_bench/Sinus10kHz_user/cordic_user/
rot[0]/U/x_o}

add wave -noupdate -expand -group Cordic -group Rotator0 -color
Green -format Logic \
{/test_bench/Sinus10kHz_user/cordic_user/rot[0]/U/y_o}

add wave -noupdate -expand -group Cordic -group Rotator0 -color
Green -format Logic \
{/test_bench/Sinus10kHz_user/cordic_user/rot[0]/U/quarter_o}
```

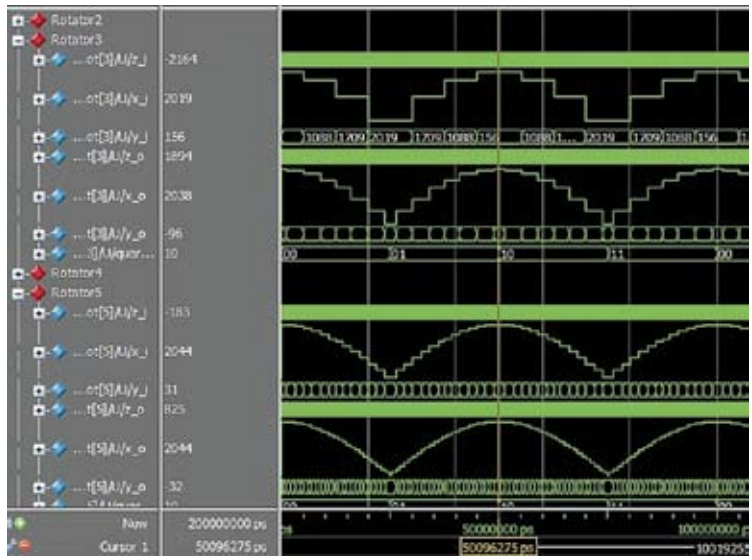

Рис. 18. Два звена конвейера

Здесь мы добавили все интересующие нас сигналы модуля **rotator.v**: входные сигналы x_i, y_i, z_i; выходные сигналы x_o, y_o, z_o; выходную четверть. В аналоговой форме мы представили только два сигнала — x_i, x_o.

Конструкция *–expand –group Cordic –group Rotator0* создает группу Cordic и еще одну вложенную группу Rotator0. Мы не будем приводить действия по добавлению в окно **Wave** всех остальных звеньев конвейера (Rotator1–Rotator15), так как это однообразное занятие.

На рис. 17 показана панель с именами сигналов окна **Wave** после добавления всех остальных звеньев конвейера.

В заключение можно добавить сигналы модуля **select_quarter.v**. Так как при добавлении этих сигналов команды ***add wave*** будут иметь одинаковые атрибуты, объединим эти сигналы в шину — с использованием всего одной команды ***add wave***:

```
#Add quarter module
add wave -noupdate -group {Select Quarter} -color Green -format Logic \
/test_bench/Sinus10kHz_user/quarter_user/Xi \
/test_bench/Sinus10kHz_user/quarter_user/Yi \
/test_bench/Sinus10kHz_user/quarter_user/Xo \
/test_bench/Sinus10kHz_user/quarter_user/Yo \
/test_bench/Sinus10kHz_user/quarter_user/quarter
```

На рис. 18 приведено окно **Wave**, в котором развернуты два звена конвейера — Rotator3 и Rotator5.

На рис. 18 отчетливо видно, как по ходу продвижения сигнала из одного звена конвейера в другой все точнее прорисовывается модуль синусоиды.

Итак, чтобы убедиться в том, что мы сгенерировали синусоиду с частотой именно

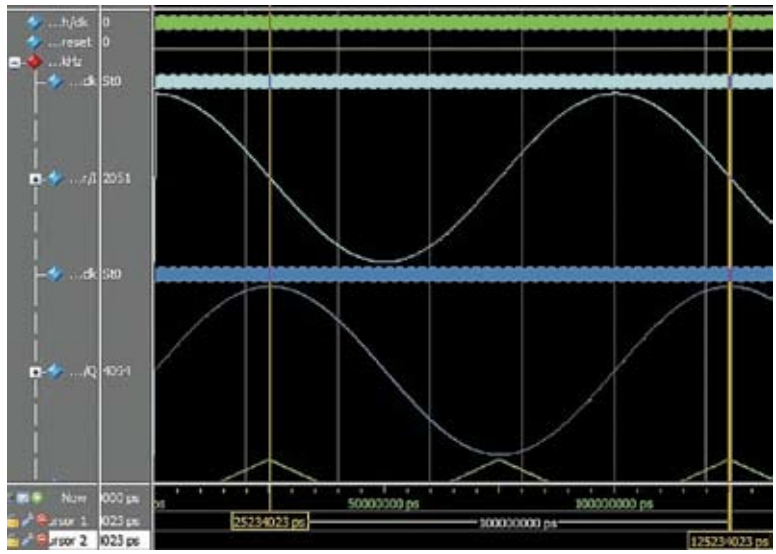

Рис. 19. Два курсора

10 кГц, добавим в наш do-файл следующую строку:

```
WaveRestoreCursors {{Cursor 1} {25234023 ps} 0} {{Cursor 2} {125234023 ps} 0}
```

Здесь мы добавили два курсора, которые располагаются на расстоянии 100 мкс друг от друга, а это и есть период синусоиды с частотой 10 кГц (рис. 19).

Давайте также убедимся в том, что мы достигли предельной точности вычислений. Для этого служит показатель выходного значения фазы — остаточный угол (рис. 20).

Здесь раскрыто последнее, 15-е звено конвейера. Можно заметить, что значение фазы на входе звена (z_i) варьируется от –1 до +1. На выходе же (z_o) эта погрешность упала до 0.

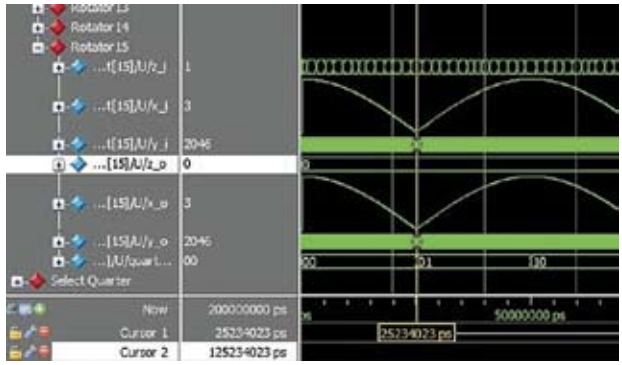

Рис. 20. Нулевой остаток значения фазы

Реализация проекта

Описанный проект автор «претворил в жизнь», используя ПЛИС от Altera Corporation семейства Cyclone EP1C6T144I7. Временной анализатор Quartus II 10.1 оценил максимальную частоту, при которой система будет работать без сбоев, в 202,55 МГц. Фиттер разместил проект на 1072 логических элементах (LE), что составляет 18% от общего размера кристалла. Из этого количества LE само ядро CORDIC-алгоритма, а именно модуль **cordic.v**, занимает 894 LE.

Автор не рекомендует использовать весь диапазон ЦАП полностью, так как ЦАП, «наевшись», может поднять уровень паразитных составляющих второстепенных гармоник нашего сигнала. Но если мы уменьшим значение коэффициента деформации Koef_Mash, мы уменьшим и амплитуду выходной синусоиды, а значит, не достигнем максимального значения 12-разрядного числа.

Заключение

Применив данный алгоритм генерации синуса, разработчик добавит своей системе гибкости. Ведь изменяя в процессе работы такие параметры системы, как фактор частоты (Freq_Step) и коэффициент деформации (Koef_Mash), мы можем получать разное значение выходной частоты и амплитуды сигнала соответственно. Для этого необходимо Freq_Step и Koef_Mash реализовать не в виде констант, а входными регистрами данных, которые будут защелкиваться по какому-то событию.

Литература

1. Захаров А.В., Хачумов В.М. Алгоритмы CORDIC. Современное состояние и перспективы. М.: Физматлит. 2004.
2. IEEE Standart Verilog Hardware Description Language 1364–2001.
3. Mentor Graphics. ModelSim Tutorial. May 2008.
4. Поляков А.К. Языки VHDL и Verilog в проектировании цифровой аппаратуры. М.: СОЛОН-Пресс, 2003.