Применение FPGA и алгоритмов Брезенхема для повышения быстродействия в системах позиционирования

Алексей ДЕНИСОВ
maildenisov@gmail.com

Для управления движением головки различных устройств с системами позиционирования используются специализированные микропроцессоры. От их производительности зависит скорость работы всего устройства в целом. В качестве альтернативы микропроцессору для повышения производительности предлагается использовать FPGA. Тогда появляется возможность осуществлять параллельно процессы управления и контроля работы станков с ЧПУ, плоттеров, графопостроителей и т. д.

Цель и задачи исследования

Что делает разработчик, если необходимо повысить производительность устройства в целом? Во-первых, это можно сделать, повысив быстродействие на существующем микропроцессоре, то есть написать оптимальную программу и алгоритм, которые учитывали бы особенности микропроцессора и его архитектуру. Во-вторых, иногда для достижения максимального быстродействия можно сделать ассемблерные вставки в программе на C/C++, но при этом программу невозможно будет перенести на другие микропроцессоры, а также это затруднит понимание самой программы. В случае если микропроцессор все равно не удовлетворяет требованиям по быстродействию, то обычно рассматривают вариант перехода на более производительный процессор.

Для достижения более высокого быстродействия предлагается использовать FPGA с ее возможностями параллельно выполнять расчеты и управление. В качестве примера приведем описание на VHDL для алгоритмов Брезенхема.

Принцип организации программы

Для начала рассмотрим принцип организации программы, которую пишет оператор для станка с числовым управлением (ЧПУ). На сегодня ввод управляющей программы для станков с ЧПУ осуществляется в стандарте ISO 6983 (ISO 7bit) или в стандарте ISO 14649 (Step-NC). Стандарт ISO 6983 использует G-коды. Программа, написанная на основе G-кодов, имеет жесткую структуру. Все команды управления объединяются в кадры — группы, состоящие из одной или более команд. Кадр завершается символом перевода строки (ПС/LF) и имеет номер, за исключением первого кадра программы и комментариев. Первый кадр содержит только один символ «%». Завершается программа командой M02 («Конец программы») или M30 («Конец информации»). Комментарии к программе размещаются в круглых скобках, занимая отдельный кадр. Рассмотрим G-коды (табл. 1), для которых можно применить алгоритмы Брезенхема.

Таблица 1. G-коды

Код	Описание	Пример
G00	Ускоренное перемещение инструмента (холостой ход)	G0 X0 Y0 Z100;
G01	Линейная интерполяция	G01 X0 Y0 Z100 F200;
G02	Круговая интерполяция по часовой стрелке	G02 X15 Y15 R5 F200;
G03	Круговая интерполяция против часовой стрелки	G03 X15 Y15 R5 F200;

Примечание.
X — координата точки траектории по оси X;
Y — координата точки траектории по оси Y;
Z — координата точки траектории по оси Z;
F — скорость рабочей подачи; R — радиус.

Ускоренное перемещение (G00) используется для начального позиционирования или холостого хода инструмента. В процессе ускоренного перемещения запрограммированное перемещение интерполируется, а движение осуществляется по прямой линии с максимальной скоростью подачи. Скорость и ускорение подачи, по крайней мере для одной оси, максимальны.

Линейная интерполяция (G01) предполагает движение по прямой линии в трехкоординатном пространстве. Перед началом интерполяционных расчетов система ЧПУ определяет длину пути, исходя из запрограммированных координат. В процессе движения осуществляется контроль контурной подачи: ее величина не должна превышать допустимых значений. Движение по всем координатам должно завершиться одновременно.

При круговой интерполяции (G02, G03) движение осуществляется по окружности в заданной рабочей плоскости. Параметры окружности (например, координаты конечной точки и ее центра или радиуса) определяются до начала движения, исходя из запрограммированных координат. В процессе движения осуществляется контроль контурной подачи: ее величина также не должна превышать допустимых значений. Движение по всем координатам должно завершиться одновременно.

Для интерполяции будем использовать алгоритм Брезенхема. К достоинствам алгоритмов Брезенхема относится их целочисленная реализация, а также отсутствие тригонометрических операций, операций деления и вычисления квадратного корня. Сразу отметим, что алгоритм для прямой не рисует горизонтальные и вертикальные линии. Это связано с тем, что рисование таких линий можно реализовать более простым способом. Эти алгоритмы хорошо описаны в Wikipedia [3].

Приведем краткое описание модернизации алгоритмов для повышения быстродействия. Для ускорения построения отрезка используют свойство центральной симметрии любого отрезка, то есть построение отрезка осуществляется из исходной точки до середины, а вторая половина отображается симметрично. Модернизация алгоритма построения окружности основана на аналогичном приеме, то есть алгоритм проходит только 1/8 часть окружности, остальные части отражаются симметрично. Также можно получить

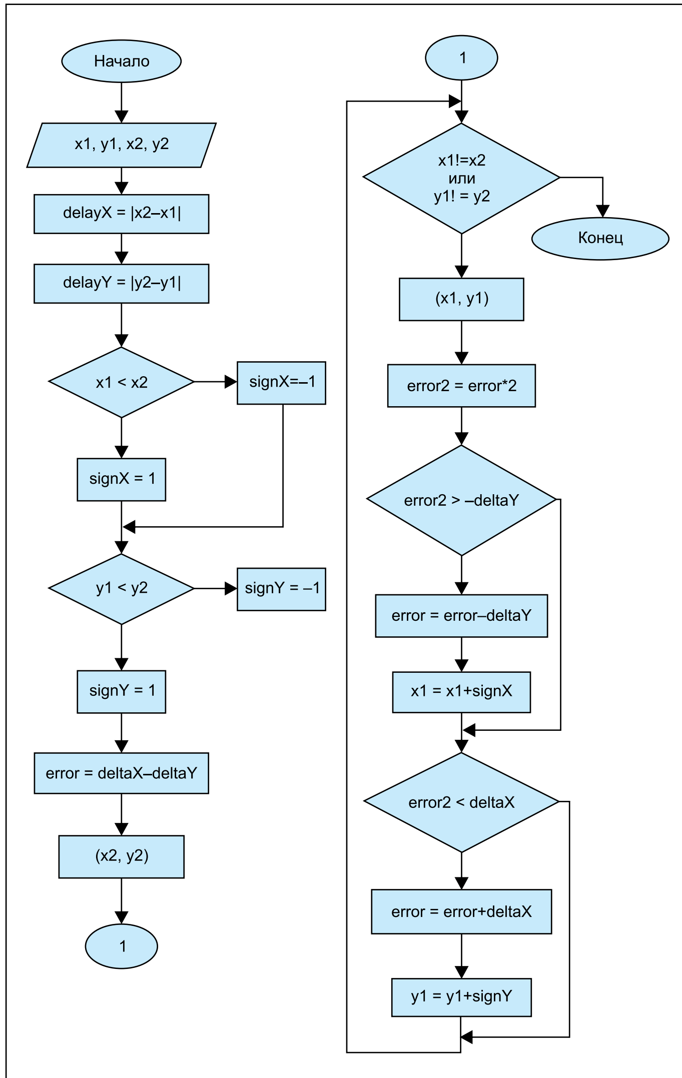

Рис. 1. Блок-схема алгоритма Брезенхема для построения прямой

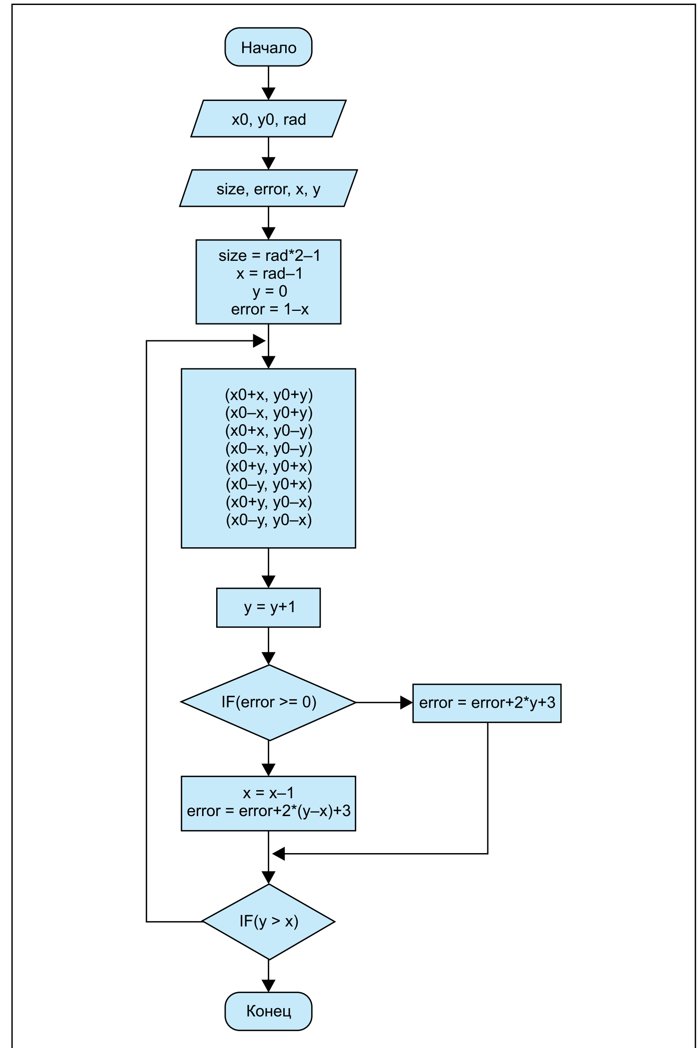

Рис. 2. Блок-схема с учетом модернизации алгоритма Брезенхема для построения окружности

критерий окончания построения 1/8 окружности, который позволит ускорить построение окружности. В модернизированных алгоритмах применяются быстрые целочисленные операции. (Вычисление середины отрезка выполняется с помощью команды сдвига.)

На рис. 1 приведена блок-схема алгоритма для построения прямой (без модернизации). Для примера на рис. 2 приводится блок-схема с учетом модернизации алгоритма Брезенхема для построения окружности.

В блок-схеме (рис. 1) функция ***Sign*** возвращает –1, 0, 1 для отрицательного, нулевого и положительного аргумента соответственно.

Тестирование алгоритмов Брезенхема на микропроцессоре

В качестве микропроцессора использовался STM32F407VGT6 [6] фирмы STMicroelectronics. Программа на C/C++ была заимствована из Wikipedia [3]. Измерение количества тактов реализовано с помощью внутреннего таймера микропроцессора. Запуск таймера происходит непосредственно перед выполнением программы для измерения. После завершения ее выполнения таймер сразу останавливается. Значение счетчика в таймере — количество тактов, которое понадобилось на выполнение операций. Программа скомпилирована с опцией оптимизации — O3 (максимальная оптимизация). Результаты работы алгоритмов Брезенхема для построения линии и окружности на микропроцессоре представлены в таблицах 2 и 3 соответственно.

Таблица 2. Результаты работы алгоритмов Брезенхема для построения линии

Начальные данные	x0(0, 0); x1(3, –3)	x0(–1, 0); x1(3, –3)
Один цикл интерполяции	13 тактов	13 тактов
Вся функция	66 такт	77 такта

Таблица 3. Результаты работы алгоритмов Брезенхема для построения окружности

Начальные данные	Центр (0, 0); R = 3	Центр (1, 1); R = 5
Один цикл интерполяции	46 тактов	67 тактов
Вся функция	132 такта	191 такт

Пример реализации на FPGA

Для проверки схемотехнической реализации на FPGA была использована плата на основе Spartan 6 [2] фирмы Xilinx. Модули разработаны на языке описания аппаратуры VHDL с использованием FSM. Основой для разработки были блок-схемы алгоритмов. Для пояснения VHDL-описания приведем назначение входов и выходов, а также дадим краткие пояснения по реализации алгоритмов [4, 5]. Описание портов модуля построения прямой (Entity line) и окружности (Entity circle) приведено в таблицах 4 и 5 соответственно.

Таблица 4. Описание портов модуля построения прямой

clk	Вход	Тактовая частота
rst	Вход	Асинхронный сброс
x0, y0	Вход	Координаты одного конца отрезка
x1, y1	Вход	Координаты другого конца отрезка
x, y	Выход	Текущее значение координаты точки отрезка при интерполяции
done	Выход	Сигнал о завершении интерполяции
start	Вход	Сигнал разрешения выполнения интерполяции

Таблица 5. Описание портов модуля построения окружности

clk	Вход	Тактовая частота
rst	Вход	Асинхронный сброс
radius	Вход	Значение радиуса
xc, yc	Вход	Координаты центра окружности
xPos, yPos	Выход	Текущее значение координаты точки на окружности
x, y	Выход	Текущее значение координаты точки окружности при интерполяции
x0, x1, x2, x3	Выход	Значения X для симметричного отображения*
y0, y1 ,y2, y3	Выход	Значения Y для симметричного отображения*
done	Выход	Сигнал о завершении интерполяции
start	Вход	Сигнал разрешения выполнения интерполяции

Примечание. * См. информацию во врезке.

В модернизированном алгоритме построения окружности необходимо сгенерировать только одну восьмую часть окружности. Остальные ее части могут быть получены последовательными отражениями. Если сгенерирован первый октант (от 0 до 45° против часовой стрелки, рисунок), то второй октант можно получить зеркальным отражением относительно прямой y = x, что дает в совокупности первый квадрант. Первый квадрант отражается относительно прямой x = 0 для получения соответствующей части окружности во втором квадранте. Верхняя полуокружность отражается относительно прямой y = 0 для завершения построения.

Описание VHDL-расчета:

Координата X	Координата Y
xc+x	yc+y
xc−x	yc−y
xc+y	yc+x
xc−y	yc−x

Далее, комбинируя/мультиплексируя полученные результаты, можно получить координаты всех восьми точек:

1.	(xc+x; yc+y)	Точка в 1-м октанте
2.	(xc−x; yc+y)	Точка в 4-м октанте
3.	(xc−x; yc−y)	Точка в 5-м октанте
4.	(xc+x; yc−y)	Точка в 8-м октанте
5.	(xc+y; yc+x)	Точка в 2-м октанте
6.	(xc−y; yc+x)	Точка в 3-м октанте
7.	(xc−y; yc−x)	Точка в 6-м октанте
8.	(xc+y; yc−x)	Точка в 7-м октанте

Расположение точек на окружности представлено на рисунке.

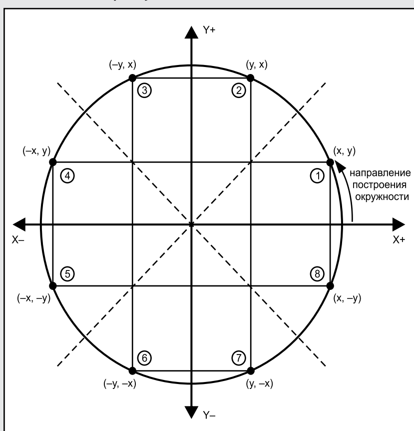

Рисунок. Восемь симметричных октантов круга и расположение восьми точек

Пример реализации алгоритма с помощью FSM для построения прямой:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

Entity line is
port (
clk, rst : in std_logic ;
x0, x1 : in std_logic_vector (9 downto 0);
y0, y1 : in std_logic_vector (9 downto 0);
x, y : out std_logic_vector (9 downto 0);
done : out std_logic;
start : in std_logic);
end line ;

ARCHITECTURE aa OF line IS
type state is (s0, s1, s2, s3, s4, s5, s6, s7);
SIGNAL s_present, s_next : state;
SIGNAL reg_x, reg_y : signed (9 downto 0);
SIGNAL err, err2 : signed (10 downto 0);
SIGNAL sx, sy, dx, dy : signed (9 downto 0);
SIGNAL reg_x0, reg_x1 : signed (9 downto 0);
SIGNAL reg_y0, reg_y1 : signed (9 downto 0);


BEGIN

PROCESS (clk, rst)
BEGIN
IF (rst = '1') THEN
s_present <= s0;
reg_x <= (others => '0');
reg_y <= (others => '0');
reg_x0 <= (others => '0');
reg_x1 <= (others => '0');
reg_y0 <= (others => '0');
reg_y1 <= (others => '0');
dx <= (others => '0');
dy <= (others => '0');
err <= (others => '0');
err2 <= (others => '0');
sx <= (others => '0');
sy <= (others => '0');

elsif clk'EVENT AND clk = '1' THEN
s_present <= s_next;

case s_present is
when s0 =>
done <= '0';
if start = '1' then
reg_x0 <= signed(x0);
reg_y0 <= signed(y0);
reg_x1 <= signed(x1);
reg_y1 <= signed(y1);
end if;

when s1 =>
reg_x <= reg_x0;
reg_y <= reg_y0;
if reg_x0 < reg_x1 then
sx <= to_signed (1, 10);
dx <= reg_x1 - reg_x0;
else
sx <= to_signed (-1, 10);
dx <= reg_x0 - reg_x1;
end if;
if reg_y0 < reg_y1 then
sy <= to_signed (1, 10);
dy <= reg_y1 - reg_y0;
else
sy <= to_signed (-1, 10);
dy <= reg_y0 - reg_y1;
end if;

when s2 =>
err <= ("0" & dx) - ("0" & dy);

when s3 =>
err2 <= err sll 1;

when s4 =>
if err2 > - ("0" & dy) then
err <= err - ("0" & dy);
reg_x <= reg_x + sx;
end if;

when s7 =>
if err2 < ("0" & dx) then
err <= err + dx;
reg_y <= reg_y + sy;
end if;

when s5 =>
x <= std_logic_vector (reg_x);
y <= std_logic_vector (reg_y);
when s6 =>
 done <= '1';

end case;

END IF;
END PROCESS;


PROCESS (s_present, start, reg_x, reg_y, reg_x1, reg_y1)
BEGIN
case s_present is

when s0 =>
if start = '1' then
   s_next <= s1;
else
   s_next <= s0;
   end if;

when s1 =>
s_next <= s2;

when s2 =>
s_next <= s3;

when s3 =>
   if reg_x = reg_x1 and reg_y = reg_y1 then
   s_next <= s6;
else
   s_next <= s4;
end if;

when s4 =>
   s_next <= s7;

when s7 =>
   s_next <= s5;

when s5 =>
   s_next <= s3;

when s6 =>
   s_next <= s0;

end case;
END PROCESS;

END aa;
```

Пример реализации алгоритма с помощью FSM для построения окружности:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

Entity circle is
port (
clk, rst : in std_logic ;
radius : in std_logic_vector (9 downto 0);
xc, yc : in std_logic_vector (9 downto 0);
xPos, yPos : in std_logic_vector (9 downto 0);
x, y : out std_logic_vector (9 downto 0);
x0, x1, x2, x3 : out std_logic_vector (9 downto 0);
y0, y1, y2, y3 : out std_logic_vector (9 downto 0);
done : out std_logic;
start : in std_logic);
end circle ;

ARCHITECTURE aa OF circle IS
type state is (s0, s1, s2, s3, s4, s5);
SIGNAL s_present, s_next : state;
SIGNAL reg_x, reg_y : signed (9 downto 0);
SIGNAL radiusError : signed (9 downto 0);
SIGNAL reg_tmp1, reg_tmp2, reg_tmp3, reg_tmp4, reg_tmp5 : signed
(9 downto 0);
SIGNAL reg_x0, reg_x1, reg_x2, reg_x3 : signed (9 downto 0);
SIGNAL reg_y0, reg_y1, reg_y2, reg_y3 : signed (9 downto 0);

BEGIN

PROCESS (clk, rst)
BEGIN
IF (rst = '1') THEN
s_present <= s0;
reg_x <= (others => '0');
reg_y <= (others => '0');
reg_tmp1 <= (others => '0');
reg_tmp2 <= (others => '0');
reg_tmp3 <= (others => '0');
reg_tmp4 <= (others => '0');
reg_tmp5 <= (others => '0');
reg_x0 <= (others => '0');
reg_x1 <= (others => '0');
reg_x2 <= (others => '0');
```

```
reg_x3 <= (others => '0');
reg_y0 <= (others => '0');
reg_y1 <= (others => '0');
reg_y2 <= (others => '0');
reg_y3 <= (others => '0');

elsif clk'EVENT AND clk = '1' THEN
s_present <= s_next;

case s_present is
when s0 =>
done <= '0';
if start = '1' then
reg_x <= signed(xPos);
reg_y <= signed(yPos);
radiusError <= 1 - signed (radius);
end if;

when s1 =>
reg_tmp1 <= reg_x - reg_y;
reg_tmp2 <= radiusError + 3;
reg_tmp3 <= radiusError + 5;
reg_tmp4 <= reg_x sll 1;

when s2 =>
reg_tmp5 <= reg_tmp1 sll 1;

when s3 =>
if radiusError < 0 then
radiusError <= reg_tmp2 + reg_tmp4;
else
radiusError <= reg_tmp3 + reg_tmp5;
reg_y <= reg_y - 1;
end if;
reg_x <= reg_x + 1;

reg_x0 <= signed (xc) + reg_x;
reg_x1 <= signed (xc) - reg_x;
reg_x2 <= signed (xc) + reg_y;
reg_x3 <= signed (xc) - reg_y;

reg_y0 <= signed (yc) + reg_y;
reg_y1 <= signed (yc) - reg_y;
reg_y2 <= signed (yc) + reg_x;
reg_y3 <= signed (yc) - reg_x;

when s4 =>
x <= std_logic_vector (reg_x);
y <= std_logic_vector (reg_y);
x0 <= std_logic_vector (reg_x0);
x1 <= std_logic_vector (reg_x1);
x2 <= std_logic_vector (reg_x2);
x3 <= std_logic_vector (reg_x3);
y0 <= std_logic_vector (reg_y0);
y1 <= std_logic_vector (reg_y1);
y2 <= std_logic_vector (reg_y2);
y3 <= std_logic_vector (reg_y3);

when s5 =>
   done <= '1';

end case;

END IF;
END PROCESS;


PROCESS (s_present, start, reg_x, reg_y)
BEGIN
case s_present is

when s0 =>
if start = '1' then
   s_next <= s1;
else
   s_next <= s0;
   end if;

when s1 =>
   s_next <= s2;

when s2 =>
   s_next <= s3;

when s3 =>
   s_next <= s4;

when s4 =>
if reg_x <= reg_y then
   s_next <= s1;
else
   s_next <= s5;
end if;

when s5 =>
   s_next <= s0;

end case;
END PROCESS;

END aa;
```

Верификация разработанных модулей

Итак, мы разработали модули на языке описания аппаратуры VHDL. Теперь нужно провести верификацию, то есть проверку правильности работы разработанных модулей. Весь процесс верификации сводится к сравнению работы заданного алгоритма с поведением разработанного модуля в ISim фирмы Xilinx.

Так как полученный FSM небольшой, то для проверки приведем:

- временные диаграммы работы модуля построения окружности (рис. 3а);
- временные диаграммы работы модуля построения прямой (рис. 3б).

На основании сравнения делаем вывод, что ожидаемое поведение модулей соответствует работе алгоритмов, представленных блок-схемами. Таким образом, разработанные модули функционируют правильно.

Выводы

По результатам Post Place & Rout для Spartan 6 фирмы Xilinx (6slx16csg324-3) объем описания на VHDL таков:

- Алгоритм построения прямой занимает около 2% логической емкости FPGA.
- Алгоритм построения окружности занимает около 5% логической емкости FPGA.

Реализация для FPGA также показала, что вычисление следующей координаты интерполяции может производиться за четыре такта для алгоритма построения окружности и линии. Тактовая частота разработанных модулей на VHDL для алгоритмов получилась выше 150 МГц. Читатель сам может сравнить результаты с результатами работы микропроцессора (табл. 2, 3) и сделать соответствующие выводы.

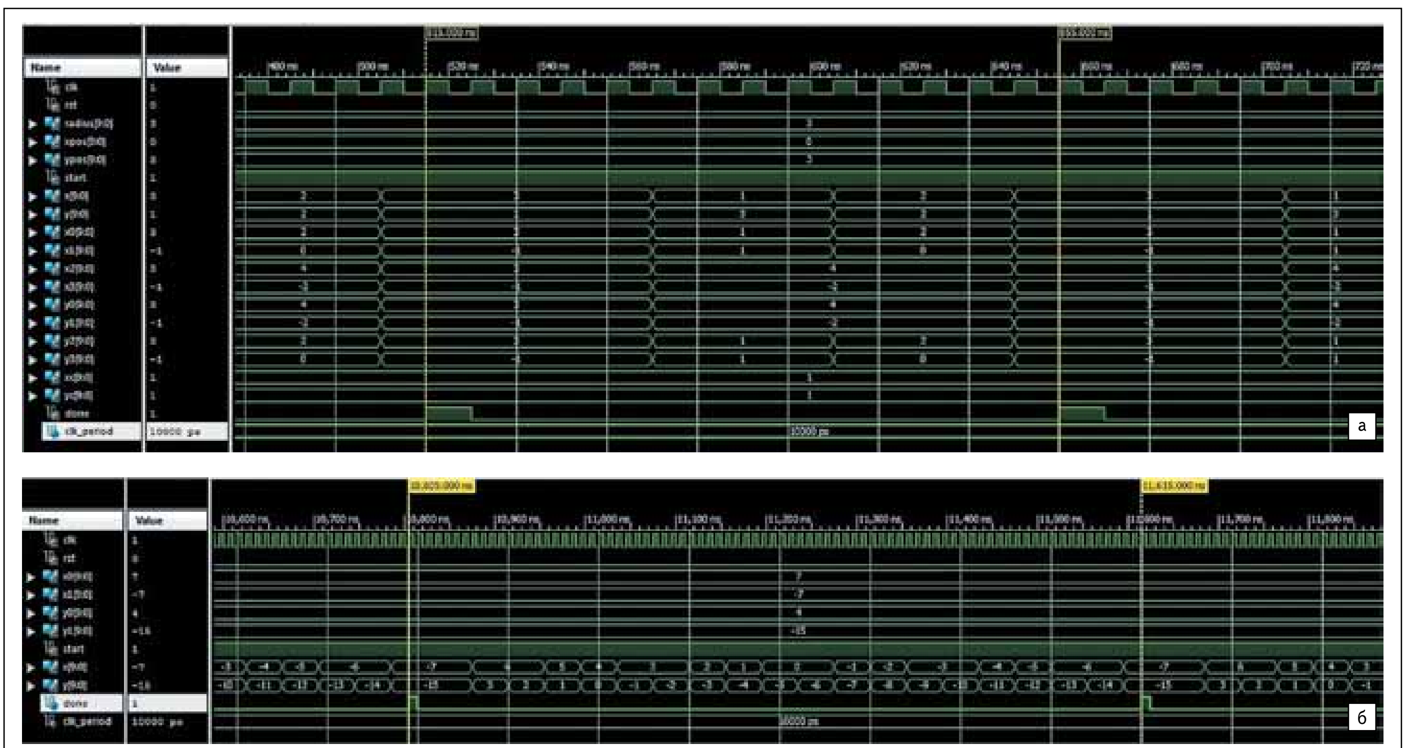

Рис. 3. Результаты моделирования: а) модуля построения окружности; б) модуля построения прямой

Также хотелось бы обратить внимание на то, что модули для построения различных прямых и окружностей занимают сравнительно мало ресурсов в FPGA, и таких независимых модулей можно создать N количество в одной FPGA, которые могут параллельно строить различные прямые и окружности. В итоге можно получить альтернативный подход к использованию FPGA в микропроцессорных системах — создание аппаратного ускорителя для процессора. В этом случае исходная программа на языке C/C++ выполняется преимущественно процессором, а наиболее затратные операции для процессора реализуются аппаратно на базе ресурсов FPGA. Этот подход в большей степени программно-ориентированный, тем не менее он часто оказывается более удобным для разработчиков микропроцессорных систем. Реализованный подобным образом аппаратный ускоритель может быть подключен к микропроцессору, например, по последовательным интерфейсам (SPI, UART, I^2C) или по параллельному интерфейсу, когда FPGA выступает для микропроцессора внешним ОЗУ (для этих целей в FPGA имеется двухпортовая память).

Заключение

Алгоритмы Брезенхема применяются во многих областях, однако в рамках этой статьи они рассматриваются на примере использования для станков с ЧПУ. Но это не значит, что приведенное решение на FPGA нельзя применить для других целей.

Продолжение статьи возможно по мере поступления вопросов от читателей. Автор готов к диалогу с разработчиками.

Литература

1. http://cadobzor.ru/G-code
2. http://www.xilinx.com/support/documentation/data_sheets/ds162.pdf
3. http://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%91%D1%80%D0%B5%D0%B7%D0%B5%D0%BD%D1%85%D1%8D%D0%BC%D0%B0
4. http://www.kit-e.ru/articles/plis/2010_9_70.php
5. http://www.kit-e.ru/articles/plis/2009_12_31.php
6. http://www.st.com/web/catalog/mmc/FM141/SC1169/SS1577/LN11/PF252140