5.2. Умножитель знаковых целых чисел

Устройства для выполнения арифметических операций являются одними из наиболее распространенных электронных схем и входят в большинство существующих проектов. Представленное ниже устройство для умножения знаковых целых чисел [32] может быть использовано вместе с разработанным ранее сумматором для формирования арифметико-логических устройств, а также для аппаратной реализации различных алгоритмов цифровой обработки сигналов.

Интерфейсная модель умножителя знаковых чисел *Signed_ _Multiply*, приведенная на рис. 5.2, содержит:

- *a* и *b* – знаковые множители в виде 4-разрядных входных сигналов (отрицательные числа представляются в дополнительном коде);
- *clk* – тактирующий вход;
- *reset* – входной сигнал сброса, срабатывающий по низкому уровню сигнала;
- *multiply_en* – сигнал разрешения начала вычислений;
- *product* – произведение входных сигналов *a* и *b*, представленное в виде 8-разрядного знакового целого числа;
- *valid* – сигнал, сигнализирующий об окончании работы над очередным набором множителей.

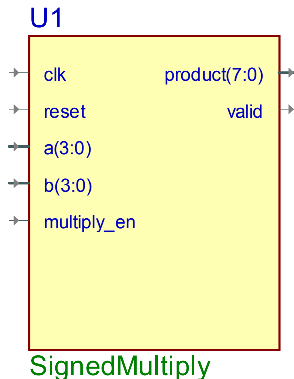

Рис. 5.2. Интерфейсная модель умножителя знаковых чисел

Поведенческая модель умножителя, характеризующаяся наиболее высоким уровнем абстракции, использует стандартную операцию умножения двоичных векторов. Кроме этого, в программу добавлен контроль знаков множителей и результирующего произведения.

5.2.1. Поведенческий код умножителя

Ниже представлен поведенческий код Verilog-программы, реализующей знаковый умножитель, разработанный без учета времени выполнения операций.

```
// ОПРЕДЕЛЕНИЯ КОНСТАНТ
`define DEL       1      // Задержка распространения сигнала в
                         // устройстве. Нулевая задержка может
                         // привести к проблемам.
`define OP_BITS 4        // Разрядность операндов

// ГЛАВНЫЙ МОДУЛЬ
module SignedMultiply(
          clk,
          reset,
          a,
          b,
          multiply_en,
          product,
          valid);

// ВХОДЫ
input                        clk;        // Тактирующий импульс
input                        reset;  // Сигнал сброса
input [`OP_BITS-1:0]     a;          // Множитель
input [`OP_BITS-1:0]     b;          // Множитель
input            multiply_en;        // Разрешение опреации

// ВЫХОДЫ
output [2*`OP_BITS-1:0]        product;        // Произведение
output           valid;  // Сигнал готовности выхода
```

```
// ОБЪЯВЛЕНИЯ СИГНАЛОВ
wire                          clk;
wire                          reset;
wire [`OP_BITS-1:0]      a;
wire [`OP_BITS-1:0]      b;
wire                          multiply_en;
reg [2*`OP_BITS-1:0]    product;
reg                           valid;
reg [`OP_BITS-1:0]       a_abs;  // Абсолютное значение a
reg [`OP_BITS-1:0]       b_abs;  // Абсолютное значение b

// ОСНОВНОЙ КОД

// Отслеживание событий на сигнале reset
always @(reset) begin
   if (reset)
         #`DEL assign valid = 0;
   else
         deassign valid;
end

// Ожидание возрастания тактирующего сигнала
always @(posedge clk) begin
   if (multiply_en) begin
         // Проверка знаков операндов, затем умножение их
         // абсолютных значений и изменение знака произведения,
         // если это необходимо
         case ({a[`OP_BITS-1],b[`OP_BITS-1]})
               2’b00: begin
                     product <= #`DEL a*b;
               end
               2’b01: begin
                     b_abs = ~b + 1;
                     product <= #`DEL ~(a*b_abs)+1;
               end
               2’b10: begin
                     a_abs = ~a + 1;
                     product <= #`DEL ~(a_abs*b)+1;
               end
```

```
                    2'b11: begin
                                a_abs = ~a + 1;
                                b_abs = ~b + 1;
                                product <= #`DEL a_abs*b_abs;
                           end
            endcase
            valid <= #`DEL 1'b1;
    end
end
endmodule                     // Конец модуля SignedMultiply
```

5.2.2. Код уровня регистровых передач

Для реализации умножителя знаковых целых чисел на уровне регистровых передач избрано три различные схемы, основанные на: комбинаторной логике [8], использовании просмотровых таблиц [14] и алгоритме частичных сумм [8].

Исходный текст модели умножителя на уровне регистровых передач, базирующейся на комбинаторной логике, приведен ниже.

```
// ОПРЕДЕЛЕНИЯ КОНСТАНТ
`define DEL      1       // Задержка распространения сигнала в
                         // устройстве. Нулевая задержка может
                         // привести к проблемам.

`define OP_BITS 4                // Разрядность операндов

// ГЛАВНЫЙ МОДУЛЬ
module SignedMultiply(
            clk,
            reset,
            a,
            b,
            multiply_en,
            product,
            valid);

// INPUTS
```

```
input                    clk;        // Тактирующий импульс
input                    reset;  // Сигнал сброса
input [`OP_BITS-1:0]   a;        // Множитель
input [`OP_BITS-1:0]   b;        // Множитель
input multiply_en;               // Разрешение операции

// ВЫХОДЫ
output [2*`OP_BITS-1:0]     product;    // Произведение
output           valid;       // Сигнал готовности выхода

// ОБЪЯВЛЕНИЯ СИГНАЛОВ
wire                          clk;
wire                          reset;
wire [`OP_BITS-1:0]   a;
wire [`OP_BITS-1:0]   b;
wire                          multiply_en;
reg [2*`OP_BITS-1:0]  product;
reg                           valid;

reg [`OP_BITS-1:0]    a_abs;  // Абсолютное значение a
reg [`OP_BITS-1:0]    b_abs;  // Абсолютное значение b

// ОСНОВНОЙ КОД

// Ожидание возрастания тактирующего сигнала clk
// и возрастания сигнала reset
always @(posedge reset or posedge clk) begin
   if (reset)
            valid <= #`DEL 1'b0;
   else if (multiply_en)
            valid <= #`DEL 1'b1;
end

// Ожидание возрастания тактирующего сигнала
always @(posedge clk) begin
   if (multiply_en) begin
            // Проверка знаков операндов, затем умножение их
            // абсолютных значений и изменение знака произведения,
```

```
            // если это необходимо
            case ({a[`OP_BITS-1],b[`OP_BITS-1]})
                2'b00: begin
                    product <= #`DEL a*b;
                end
                2'b01: begin
                    b_abs = ~b + 1;
                    product <= #`DEL ~(a*b_abs)+1;
                end
                2'b10: begin
                    a_abs = ~a + 1;
                    product <= #`DEL ~(a_abs*b)+1;
                end
                2'b11: begin
                    a_abs = ~a + 1;
                    b_abs = ~b + 1;
                    product <= #`DEL a_abs*b_abs;
                end
            endcase
    end
end
endmodule               // Конец модуля SignedMultiply
```

Исходный текст RTL-уровня модели умножителя, работающий по принципу просмотровой таблицы значений (с использованием постоянного запоминающего устройства):

```
// ОПРЕДЕЛЕНИЯ КОНСТАНТ
`define DEL     1       // Задержка распространения сигнала в
                        // устройстве. Нулевая задержка может
                        // привести к проблемам.

`define OP_BITS 4               // Разрядность операндов

// ГЛАВНЫЙ МОДУЛЬ
module SignedMultiply(
            clk,
            reset,
```

```
          a,
          b,
          multiply_en,
          product,
          valid);

// ВХОДЫ
input                        clk;          // Тактирующий импульс
input                        reset;  // Сигнал сброса
input [`OP_BITS-1:0]     a;             // Множитель
input [`OP_BITS-1:0]     b;             // Множитель
input multiply_en;        // Разрешение операции

// ВЫХОДЫ
output [2*`OP_BITS-1:0]        product;        // Произведение
output              valid;          // Сигнал готовности выхода

// ОБЪЯВЛЕНИЯ СИГНАЛОВ
wire                                 clk;
wire                                 reset;
wire [`OP_BITS-1:0]      a;
wire [`OP_BITS-1:0]      b;
wire                                 multiply_en;
reg [2*`OP_BITS-1:0]     product;
reg                                  valid;

reg [`OP_BITS-1:0]       a_abs;  // Абсолютное значение a
reg [`OP_BITS-1:0]       b_abs;  // Абсолютное значение b

// ОСНОВНОЙ КОД

// Включение просмотровой таблицы в ПЗУ
Sm_Rom Rom(
   .address({a,b}),
   .out(romout));

// Ожидание возрастания тактирующего сигнала clk
// и возрастания сигнала reset
```

```
always @(posedge reset or posedge clk) begin
   if (reset)
            valid <= #`DEL 1'b0;
   else if (multiply_en)
            valid <= #`DEL 1'b1;
end

// Ожидание возрастания тактирующего сигнала
always @(posedge clk) begin
   if (multiply_en)
            product <= #`DEL romout;
end
endmodule                // Конец модуля SignedMultiply
// ПОДЧИНЕННЫЙ МОДУЛЬ
`include "sm_rom.v"
```

В последней строке предыдущей программы используется директива включения файла sm_rom.v, моделирующего просмотровую таблицу (постоянное запоминающее устройство):

```
// ГЛАВНЫЙ МОДУЛЬ
module Sm_Rom( address, out);

// ВХОДЫ
input [7:0] address; // ROM address

// ВЫХОДЫ
output [7:0] out; // ROM output

// ОБЪЯВЛЕНИЯ СИГНАЛОВ
wire [7:0] address;
wire [7:0] out;

reg [7:0] rom[255:0];

// ОПЕРАТОРЫ ПОТОКОВ ДАННЫХ
assign out = rom[address];
```

```
// ОСНОВНОЙ КОД

initial begin
   rom[0] = 0;
   rom[1] = 0;
   rom[2] = 0;
   rom[3] = 0;
   rom[4] = 0;
   rom[5] = 0;
   rom[6] = 0;
   rom[7] = 0;
   rom[8] = 256;
   rom[9] = 256;
   rom[10] = 256;
   rom[11] = 256;
   rom[12] = 256;
   rom[13] = 256;
   rom[14] = 256;
   rom[15] = 256;
   rom[16] = 0;
   rom[17] = 1;
   rom[18] = 2;
   rom[19] = 3;
   rom[20] = 4;
   rom[21] = 5;
   rom[22] = 6;
   rom[23] = 7;
   rom[24] = 248;
   rom[25] = 249;
   rom[26] = 250;
   rom[27] = 251;
   rom[28] = 252;
   rom[29] = 253;
   rom[30] = 254;
   rom[31] = 255;
   rom[32] = 0;
   rom[33] = 2;
   rom[34] = 4;
   rom[35] = 6;
   rom[36] = 8;
   rom[37] = 10;
   rom[38] = 12;
   rom[39] = 14;
   rom[40] = 240;
   rom[41] = 242;
   rom[42] = 244;
   rom[43] = 246;
   rom[44] = 248;
   rom[45] = 250;
   rom[46] = 252;
   rom[47] = 254;
   rom[48] = 0;
   rom[49] = 3;
   rom[50] = 6;
   rom[51] = 9;
   rom[52] = 12;
   rom[53] = 15;
   rom[54] = 18;
   rom[55] = 21;
   rom[56] = 232;
   rom[57] = 235;
   rom[58] = 238;
   rom[59] = 241;
   rom[60] = 244;
   rom[61] = 247;
   rom[62] = 250;
   rom[63] = 253;
   rom[64] = 0;
   rom[65] = 4;
   rom[66] = 8;
   rom[67] = 12;
   rom[68] = 16;
   rom[69] = 20;
   rom[70] = 24;
   rom[71] = 28;
   rom[72] = 224;
   rom[73] = 228;
   rom[74] = 232;
```

```
rom[75] = 236;
rom[76] = 240;
rom[77] = 244;
rom[78] = 248;
rom[79] = 252;
rom[80] = 0;
rom[81] = 5;
rom[82] = 10;
rom[83] = 15;
rom[84] = 20;
rom[85] = 25;
rom[86] = 30;
rom[87] = 35;
rom[88] = 216;
rom[89] = 221;
rom[90] = 226;
rom[91] = 231;
rom[92] = 236;
rom[93] = 241;
rom[94] = 246;
rom[95] = 251;
rom[96] = 0;
rom[97] = 6;
rom[98] = 12;
rom[99] = 18;
rom[100] = 24;
rom[101] = 30;
rom[102] = 36;
rom[103] = 42;
rom[104] = 208;
rom[105] = 214;
rom[106] = 220;
rom[107] = 226;
rom[108] = 232;
rom[109] = 238;
rom[110] = 244;
rom[111] = 250;
rom[112] = 0;
rom[113] = 7;
```

```
rom[114] = 14;
rom[115] = 21;
rom[116] = 28;
rom[117] = 35;
rom[118] = 42;
rom[119] = 49;
rom[120] = 200;
rom[121] = 207;
rom[122] = 214;
rom[123] = 221;
rom[124] = 228;
rom[125] = 235;
rom[126] = 242;
rom[127] = 249;
rom[128] = 256;
rom[129] = 248;
rom[130] = 240;
rom[131] = 232;
rom[132] = 224;
rom[133] = 216;
rom[134] = 208;
rom[135] = 200;
rom[136] = 64;
rom[137] = 56;
rom[138] = 48;
rom[139] = 40;
rom[140] = 32;
rom[141] = 24;
rom[142] = 16;
rom[143] = 8;
rom[144] = 256;
rom[145] = 249;
rom[146] = 242;
rom[147] = 235;
rom[148] = 228;
rom[149] = 221;
rom[150] = 214;
rom[151] = 207;
rom[152] = 56;
```

```
rom[153] = 49;
rom[154] = 42;
rom[155] = 35;
rom[156] = 28;
rom[157] = 21;
rom[158] = 14;
rom[159] = 7;
rom[160] = 256;
rom[161] = 250;
rom[162] = 244;
rom[163] = 238;
rom[164] = 232;
rom[165] = 226;
rom[166] = 220;
rom[167] = 214;
rom[168] = 48;
rom[169] = 42;
rom[170] = 36;
rom[171] = 30;
rom[172] = 24;
rom[173] = 18;
rom[174] = 12;
rom[175] = 6;
rom[176] = 256;
rom[177] = 251;
rom[178] = 246;
rom[179] = 241;
rom[180] = 236;
rom[181] = 231;
rom[182] = 226;
rom[183] = 221;
rom[184] = 40;
rom[185] = 35;
rom[186] = 30;
rom[187] = 25;
rom[188] = 20;
rom[189] = 15;
rom[190] = 10;
rom[191] = 5;
rom[192] = 256;
rom[193] = 252;
rom[194] = 248;
rom[195] = 244;
rom[196] = 240;
rom[197] = 236;
rom[198] = 232;
rom[199] = 228;
rom[200] = 32;
rom[201] = 28;
rom[202] = 24;
rom[203] = 20;
rom[204] = 16;
rom[205] = 12;
rom[206] = 8;
rom[207] = 4;
rom[208] = 256;
rom[209] = 253;
rom[210] = 250;
rom[211] = 247;
rom[212] = 244;
rom[213] = 241;
rom[214] = 238;
rom[215] = 235;
rom[216] = 24;
rom[217] = 21;
rom[218] = 18;
rom[219] = 15;
rom[220] = 12;
rom[221] = 9;
rom[222] = 6;
rom[223] = 3;
rom[224] = 256;
rom[225] = 254;
rom[226] = 252;
rom[227] = 250;
rom[228] = 248;
rom[229] = 246;
rom[230] = 244;
```

```
    rom[231] = 242;
    rom[232] = 16;
    rom[233] = 14;
    rom[234] = 12;
    rom[235] = 10;
    rom[236] = 8;
    rom[237] = 6;
    rom[238] = 4;
    rom[239] = 2;
    rom[240] = 256;
    rom[241] = 255;
    rom[242] = 254;
    rom[243] = 253;
    rom[244] = 252;
    rom[245] = 251;
    rom[246] = 250;
    rom[247] = 249;
    rom[248] = 8;
    rom[249] = 7;
    rom[250] = 6;
    rom[251] = 5;
    rom[252] = 4;
    rom[253] = 3;
    rom[254] = 2;
    rom[255] = 1;
  end
  endmodule // Конец модуля Sm_Rom
```

Исходный текст RTL-уровня модели умножителя, базирующейся на алгоритме частичных сумм:

```
// ОПРЕДЕЛЕНИЯ КОНСТАНТ
`define DEL      1        // Задержка распространения сигнала в
                          // устройстве. Нулевая задержка может
                          // привести к проблемам.

`define OP_BITS 4                // Разрядность операндов

// ГЛАВНЫЙ МОДУЛЬ
```

```
module SignedMultiply(
        clk,
        reset,
        a,
        b,
        multiply_en,
        product,
        valid);

// ВХОДЫ
input                       clk;        // Тактирующий импульс
input                       reset;   // Сигнал сброса
input [`OP_BITS-1:0]    a;                  // Множитель
input [`OP_BITS-1:0]    b;                  // Множитель
input multiply_en;      // Разрешение операции

// ВЫХОДЫ
output [2*`OP_BITS-1:0]     product;        // Произведение
output          valid;          // Сигнал готовности выхода

// ОБЪЯВЛЕНИЯ СИГНАЛОВ

wire                            clk;
wire                            reset;
wire [`OP_BITS-1:0]     a;
wire [`OP_BITS-1:0]     b;
wire                            multiply_en;
reg [2*`OP_BITS-1:0]    product;
wire                            valid;

reg [`OP_SIZE-1:0]      count;  // Число битов для
                                        // сдвига
reg [2*`OP_BITS-1:0]    acount;// Знаковое
                                        // представление входа a

// ОПЕРАТОРЫ ПОТОКОВ ДАННЫХ
assign valid = (count == 0) ? 1'b1 : 1'b0;

// ОСНОВНОЙ КОД
```

```
// Ожидание возрастания тактирующего сигнала clk
// и возрастания сигнала reset
always @(posedge reset or posedge clk) begin
   if (reset) begin
            count <= #`DEL `OP_SIZE 'b0;
   end
   else begin
            if (multiply_en && valid) begin
                     // Установить все разряды счетчика в 1 –
                     // максимально допустимое значение
                     count <= #`DEL ~`OP_SIZE 'b0;
            end
            else if (count)
                     count <= #`DEL count – 1;
   end
end

// Ожидание возрастания тактирующего сигнала
always @(posedge clk) begin
   if (multiply_en & valid) begin

            // Формирование расширенного представления сигнала a
            acount[`OP_BITS-1:0] = a;
            // Установка знака в расширенном представлении сигнала a
            if (a[`OP_BITS-1])
                     acount[2*`OP_BITS-1:`OP_BITS] =
                                        ~`OP_BITS 'b0;
            else
                     acount[2*`OP_BITS-1:`OP_BITS] =
                              `OP_BITS 'b0;

            if (b[`OP_BITS-1]) begin
                     // Если наиболее значащий бит множителя равен 1, то
                     // вычесть знаковый расширенный множитель
                     // из произведения
                     product <= #`DEL 0 – acount;
            end
            else begin
```

```
                // Если наиболее значащий бит множителя равен 1, то
                // установить произведение в 0
                product <= #`DEL 0;
            end
        end
        else if (count) begin
            if (b[count-1]) begin
                // Если текущий бит множителя равен 1, то
                // сдвинуть произведение влево и прибавить к
                // произведению знаковый расширенный множитель
                product <= #`DEL (product << 1) + acount;
            end
            else begin
                // Если текущий бит множителя равен 1, то
                // только сдвинуть произведение влево
                product <= #`DEL product << 1;
            end
        end
    end
    endmodule          // Конец модуля SignedMultiply
```

5.2.3. Испытательный стенд

Для тестирования приведенных выше моделей умножителя разработан испытательный стенд, обеспечивающий формирование тестовых векторов, состоящих из случайных 4-разрядных знаковых чисел, а также осуществляющий контроль за результатами работы спроектированного устройства:

```
// ОПРЕДЕЛЕНИЯ КОНСТАНТ
`define OP_BITS 4          // Разрядность операндов

// ГЛАВНЫЙ МОДУЛЬ
module smultiply_sim();

// ОБЪЯВЛЕНИЯ СИГНАЛОВ

reg                              clock;
```

```
reg                                   reset;
reg [`OP_BITS-1:0]          a_in;
reg [`OP_BITS-1:0]          b_in;
reg                                   multiply_en;
wire [2*`OP_BITS-1:0]       product_out;
wire                                  valid;

reg [2*`OP_BITS:0] cycle_count; // Подсчет тактов до
                                      // установки сигнала valid

integer          val_count;           // Подсчет тактов
                                      // между пакетами данных

integer          a_integer;     // Знаковое представление
                                      // множителя a_in
integer          b_integer;     // Знаковое представление
                                      // множителя b_in
reg [`OP_BITS-1:0]       temp;  // Временное хранилище
reg [2*`OP_BITS-1:0] ltemp;     // Временное хранилище
integer          expect_integer;// Ожидаемое значение
                 // произведения в целочисленном представлении
reg [2*`OP_BITS-1:0]     expect;// Ожидаемое значение выхода

// ОСНОВНОЙ КОД

// Включение тестируемого объекта – умножитель
SignedMultiply smult(
                 .clk(clock),
                 .reset(reset),
                 .a(a_in),
                 .b(b_in),
                 .multiply_en(multiply_en),
                 .product(product_out),
                 .valid(valid));

// Инициализация входов
initial begin
   clock = 1;
   cycle_count = 0;
```

```
    multiply_en = 1;          // Начало операции умножения

    reset = 1;                // Формирование сигнала сброса
    #10 reset = 0;   // для инициализации выхода valid

    val_count = 0;  // Обнуление счетчика тактов
end

// Генератор тактирующего импульса
always #100 clock = ~clock;

// Моделирование
always @(negedge clock) begin
    if (valid === 1'b1) begin
     // Проверка результата на корректность
     if (product_out !== expect) begin
    $display("\nОШИБКА в момент времени %0t:", $time);
            $display("Умножитель не работает");
            if (multiply_en)
                    $display("Умножение разрешено");
            else
            $display("Умножение не разрешено");
            $display(" a_in = %d (%h)", a_integer, a_in);
            $display(" b_in = %d (%h)", b_integer, b_in);
            $display(" ожидаемый результат = %d (%h)",
                                        expect_integer, expect);
            expect_integer = long_to_int(product_out);
            $display(" полученный результат = %d (%h)\n",
                                    expect_integer, product_out);

                    // Использование $stop для отладки
            $stop;
     end

            // Формирование входных сигналов
            // из диапазона между 0 и всеми единицами
            a_in = cycle_count[2*`OP_BITS-1:`OP_BITS];
            b_in = cycle_count[`OP_BITS-1:0];
```

```
        // Конвертирование беззнаковых чисел к знаковым числам
        a_integer = short_to_int(a_in);
        b_integer = short_to_int(b_in);

// Сколько тактов проходит до установки
// соответствующего сигнала на выходе?
        val_count = 0;

        // Подсчет числа тактов
        cycle_count = cycle_count + 1;
        if (cycle_count[2*`OP_BITS]) begin
                case (cycle_count[1:0])
                0: begin
                expect_integer = a_integer * b_integer;
                expect = expect_integer[2*`OP_BITS-1:0];
                end
                1: begin
                multiply_en = 0;          // Остановка операции
// Изменяются входы, но не изменяется ожидаемый выход,
// пока выполнение операции приостановлено
                        end
                        2: begin
                        $display
                ("\nМоделирование завершено без ошибок\n");
                                $finish;
                        end
                endcase
        end
        else begin
                expect_integer = a_integer * b_integer;
                expect = expect_integer[2*`OP_BITS-1:0];
        end
end
else begin
        val_count = val_count + 1;

        if (val_count > 2*`OP_BITS+3) begin
$display("\nОШИБКА в момент времени %0t:", $time);
$display("Слишком много тактов до установки выхода\n");
```

```
                    // Использование $stop для отладки
                    $stop;
            end
    end
end

// ФУНКЦИИ

// Функции для конвертирования регистров в знаковые числа integer
function integer short_to_int;

input [`OP_BITS-1:0]      x;

begin
    if (x[`OP_BITS-1]) begin
            temp = ~x + 1;
            short_to_int = 0 – temp;
    end
    else
            short_to_int = x;
end
endfunction

function integer long_to_int;

input [2*`OP_BITS-1:0]  x;

begin
    if (x[2*`OP_BITS-1]) begin
            ltemp = ~x + 1;
            long_to_int = 0 – ltemp;
    end
    else
            long_to_int = x;
end
endfunction
endmodule                    // Конец модуля smultiply_sim
```

Для разработки реальных проектов в каждом конкретном случае следует выбирать из трех приведенных выше примеров реализации устройства умножения целых знаковых чисел ту схему, которая является более эффективной с точки зрения выбранной аппаратной платформы (серии FPGA-микросхем). Схему, основанную на использовании ПЗУ, следует выбирать в проектах, требующих высокого быстродействия. Хорошим компромиссом между быстродействием и необходимым для реализации количеством вентилей может послужить схема, использующая алгоритм частичных сумм.