The Design of an Asynchronous VHDL Synthesizer

Sun-Yen Tan, Stephen B. Furber

Department of Computer Science
University of Manchester
Manchester, M13 9PL, UK
tansy@cs.man.ac.uk, sfurber@cs.man.ac.uk

Wen-Fang Yen

Department of Electronic Engineering
National Taipei University of Technology
Taipei, Taiwan, R.O.C.
yen@ntut.edu.tw

Abstract

This paper presents a straightforward approach for synthesizing a standard VHDL description of an asynchronous circuit from a behavioural VHDL description. The asynchronous circuit style is based on 'micropipelines', a style currently used to develop asynchronous microprocessors at Manchester University. The rules of partition and conversion which are used to implement the synthesizer are also described. The synthesizer greatly reduces the design time of a complex micropipeline circuit.

1 Introduction

The design of asynchronous circuits generally follows a modular approach, where a system is designed as an interconnection of modules. In the 1988 Turing Award Lecture, Sutherland expounded a modular approach to building hardware systems based on data-driven asynchronous self-timed logic elements called micropipelines [1]. This micropipeline design methodology has attracted the attention of many researchers, including those in the Department of Computer Science at Manchester University. The modular approach reduces design time and cost. It is being used to develop high performance and low power microprocessing systems by the AMULET group at Manchester University [2, 3].

Asynchronous circuits are attracting renewed interest because they avoid the problems caused by the clock in synchronous circuits. The design of asynchronous circuits is in general more difficult than that of synchronous circuits. The control circuits of micropipelines are composed from event-driven logic modules and their interconnections. To implement the control circuits the designers only need to construct the interconnections between the modules. The event-driven logic modules may be designed by specialists in advance, and replaced by new modules whenever better designs become available. It is not necessary to change the interconnections when such replacements happen. Thus, the design of asynchronous circuits is simplified by the application of micropipeline design techniques. Most micropipeline designs are presently done by hand. An automatic synthesis tool can reduce the design time required to obtain high performance circuits. VHDL is an IEEE standard introduced in 1987 [4] which supports an environment within which designs may be described using VHDL statements and then simulated. It is easy to check the correctness of the specifications and requirements of the designs at an early stage. In this paper we introduce a straightforward approach in converting top level behavioural VHDL models into structural VHDL models and Verilog models based on micropipelined asynchronous circuits.

Section 2 introduces Event-driven Logic Modules and micropipelines. Section 3 describes the synthesis method which is used by the synthesizer. A brief introduction to VHDL models will be presented in Section 4. Section 5 will present some rules and steps for implementing the synthesizer. A test example used to show how the synthesizer works is also included in section 5. Finally, Section 6 will give a short conclusion and suggestions for further research.

2 Micropipelines

Micropipelines are members of a class of asynchronous pipelines whose operation is based on a two-phase bundled data convention and are self-timed, event-driven systems [1]. A "two-phase bundled data convention" is a communication system where a two-phase handshaking protocol is used and an arbitrary number of data wires must be treated as a bundle together with the request signal wire. In the two-phase handshaking protocol, rising transitions and falling transitions of either control wire have the same meaning; they represent request events or acknowledge events. In this signalling scheme, the operating cycle is (1) data available (2) request event, and (3) acknowledge event.

Various circuits have been devised for controlling transition signals. These are called event-driven logic modules [1]. Figure 1 shows the circuit symbols of these event modules.

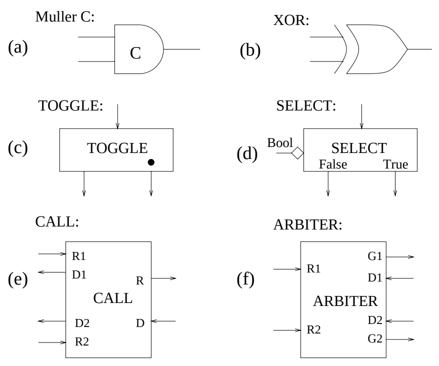

Figure 1: Circuit symbols of event-driven logic modules.

The basic modules are :

Muller C-element: A C-element [5] performs the rendezvous function for events.

XOR circuit: An XOR circuit performs the merge function for events.

TOGGLE circuit: A TOGGLE circuit sends events from an incoming stream alternately to one of two outputs.

SELECT module: A SELECT module can be used to deliver events to one of two outputs under the control of a boolean signal.

CALL module: A CALL module can be used to give shared access to a sub-process from two independent (but mutually exclusive) higher-level processes.

ARBITER module: When two events occur simultaneously or nearly simultaneously, an arbitration element must be used to impose an ordering on the events. Only one event is granted at a time. An ARBITER element can be connected directly to the CALL element to enforce mutual exclusively. Arbiter trees can be used to implement arbiters which have more than two inputs [6].

Figure 2 is an example of a micropipeline stage. A micropipeline stage consists of control circuits, combinational logic circuits and storage elements. When

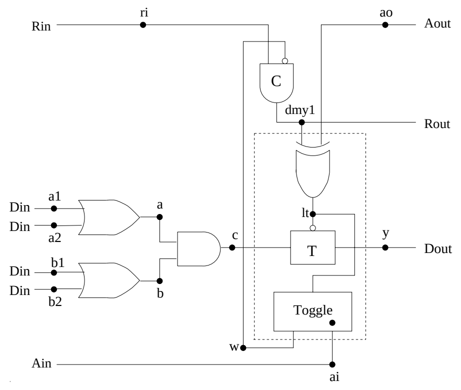

Figure 2: A micropipeline stage.

data is available on the inputs, there is an event called ‘request’ on ‘Rin’ to enable the storage elements to capture this valid data. When the storage elements have latched the input data, there is an event called ‘acknowledge’ on ‘Ain’ to inform the preceding stage that the data has been accepted and an event called ‘request’ on ‘Rout’ to enable the successor stage to capture the data. When the successor stage holds data, it will place an event on ‘Aout’ to clear the storage elements of the current stage and enable the current stage to capture subsequent data.

The control circuits are composed from event-driven logic modules, such as the Muller C, TOGGLE, SELECT, CALL ... etc. In this example, the combinational logic circuit consists of two OR gates and one AND gate. The storage element consists of an XOR, an active-low transparent latch and a TOGGLE. The control circuit here is only a Muller C. It is used to implement the rule “if predecessor and successor differ in state then copy predecessor’s state else hold present state” [1]. This rule allows data to flow through the micropipeline stages. Such event-driven logic modules can also be used to connect micropipeline stages in different configurations. For example, Figure 3 shows a micropipeline stage which selectively connects to one of two micropipeline stages which call the same output micropipeline stage.

The communication between stages or storage elements in micropipelines is not fully delay-insensitive if control and data signals are considered separately. The delays in data transmission must be less than the delays in transmitting the request signal. The combinational logic circuits for computations within the micropipeline must be carefully designed. High speed electronic techniques, such as dynamic CMOS tech-

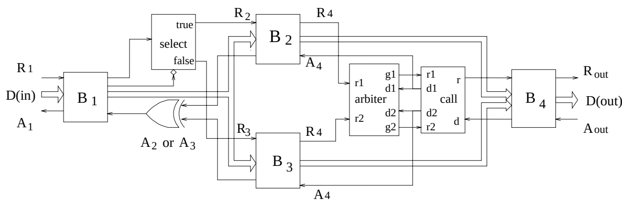

Figure 3: One micropipeline stage(B_1) selectively connects to one of two micropipeline stages(B_2 or B_3) which call the same output micropipeline stage(B_4).

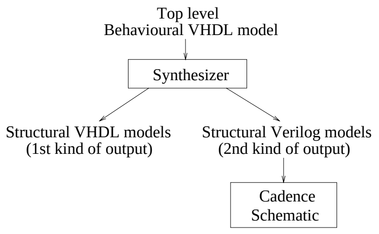

Figure 4: Structure of the synthesizer.

```
ENTITY ex1 IS
  PORT ( RIN, AOUT : IN  MVL;
         AIN, ROUT : OUT  MVL;
         A, B, C, D: IN MVL;
         SUM : OUT MVL_VECTOR( 1 DOWNTO 0 );
         CARRY : OUT MVL;
         RESET : IN MVL );
END ex1;
architecture BEHAVIOR of ex1 is
begin
  U1: process
    variable QA, QB : MVL_VECTOR( 1 DOWNTO 0 );
    variable QC : MVL_VECTOR( 2 DOWNTO 0 );
    begin
        QA := A + B;
        QB := C + D;
        QC := QA + QB;
        SUM <= QC( 1 DOWNTO 0 );
        CARRY <= QC( 2 );
    end process U1;
end BEHAVIOR;
```

Figure 5: A top level behavioural VHDL model.

niques, may be applied when designing this part. The data signals must propagate through a micropipeline faster than the control event through its control circuits. Therefore, special delays are sometimes required in the control path when significant processing logic is put between storage elements in the data path.

Previous work has been carried out in translation from programs to circuits [7, 8, 9, 10, 11]. A synthesis method of transforming concurrent programs based on a variant of CSP [13] into self-timed circuits has been developed by S. Burns in 1988 [7]. A subset of OCCAM [12] based on CSP is used to describe computations and a compiler has been constructed that automatically performs the translation and transformation [8, 9]. The translation of Tangram programs based on CSP and guarded-command language [14] into handshake circuits and then VLSI circuits has also been developed by Philips Research Laboratories [10]. A silicon compiler which converts a high-level programming language, OCCAM(async), to asynchronous CMOS circuits has been described [11]. However, previous work has not used VHDL which is an IEEE standard introduced in 1987 [4]; this supports an environment within which designs may be described using VHDL statements and then simulated easily. The authors are interested in developing a novel method for translating VHDL behavioural descriptions into micropipeline circuits. The research is ongoing and progress so far is presented in this paper.

3 The method of synthesis

Figure 4 shows the structure of the synthesizer. The input language of the synthesizer is a **top level behavioural VHDL model**. The VHDL statements are partitioned into several blocks. Then the statements within each block are converted into actual circuits which are denoted by objects of C++ classes. The synthesizer has two kinds of output. The first kind of output are files of **structural VHDL models** of the components. These components are pre-defined behavioural VHDL models of event-driven logic modules and VHDL models of standard logic elements. The output may be simulated to check the correctness of the synthesized circuits. The second kind of output of the synthesizer are files of **structural Verilog models** of the components. This output may be imported into a Design Framework II database using Verilog In [21]. This is convenient for subsequent design work in the Cadence environment.

The synthesis steps are summarized as follows:

1. Partition the statements into several stages. The partition rules are described in Section 5. For example, the statements of the behavioural VHDL model shown in Fugure 5 are partitioned into two stages. Statements $QA := A + B$; and $QB := C + D$; are within the first stage. Statement QC := QA + QB; is within the second stage.

2. Analyse the statements within each stage to obtain synthesis information. The synthesis information is used to recognize whether the stage contains IF, FOR, and WHILE statements or only as-

signments. The two stages which are partitioned from the statements shown in Figure 5 contain only assignments.

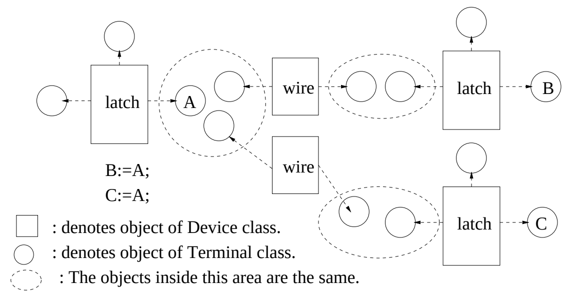
latch
A
wire
latch
B
wire
latch
C
B:=A;
C:=A;
: denotes object of Device class.
: denotes object of Terminal class.
: The objects inside this area are the same.

Figure 6: The denotation of components using objects of Device and Terminal classes.

3. Convert all statements into internal circuit models which are denoted by objects of Device and Terminal classes, as shown in Figure 6, for each stage. These classes have been used to implement a micropipeline simulator [15, 16] based on Petri net modelling techniques [17]. The conversion rules are mentioned in Section 5. The internal circuit models of the data path for the above two stages are shown in Figure 7.

4. Using the synthesis information from each stage produce the relevant control circuits. Because the above two stages only contain assignments, the control circuits for these two stages are the same as the control circuit shown in Figure 2. The rules for producing control circuits are described in Section 5.

5. Using information from the predecessor and successor of each stage produce the interconnections between stages. For the above two stages, the predecessor of the first stage is the top level, i.e. the external data inputs and RIN and AIN. The successor of the first stage is the second stage, i.e. the data inputs and RIN and AIN of the second stage. Similarly, the predeccessor of the second stage is the first stage, i.e. the data outputs and ROUT and AOUT. The successor of the second stage is the top level, i.e. the external data outputs and ROUT and AOUT.

6. Produce the structural VHDL model for each stage and for the whole network. The steps for producing structural VHDL output files are presented in Section 5. The corresponding structural VHDL model is shown in Figure 8.

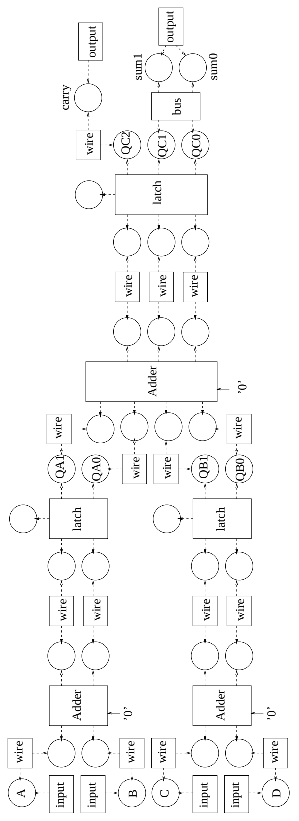

Figure 7: The internal circuit model of the data path for the VHDL model of Figure 5.

4 VHDL

The reasons for choosing VHDL as the input and output models are that :

- VHDL is IEEE standard 1076-1987 [4]. It is an important industrial design tool for digital systems.
- The event-driven logic modules shown in Figure 1 have been modelled individually using VHDL. These VHDL models have also been compiled and entered into the library. Therefore, they can be used as components in the synthesized circuits.
- Input or output models written using VHDL are easily simulated and their functions verified. The benefits of VHDL are described elsewhere [18] and VHDL synthesis examples are also given.

The first kind of output of the synthesizer is a set of **structural VHDL models** of the components. An entity and a structural architecture pair for each

```
architecture STRUCTURAL of EX1 is
  component stg1
     PORT ( reset_stg1 : IN MVL;
                rin_stg1 : IN MVL;    aout_stg1 : IN MVL;
                ain_stg1 : OUT MVL;   rout_stg1 : OUT MVL;
                U11a0: IN MVL;         U11b0: IN MVL;
                U12a0: IN MVL;         U12b0: IN MVL;
                QA0: OUT MVL;          QA1: OUT MVL;
                QB0: OUT MVL;          QB1: OUT MVL  );
  end component;
  for Ustg1: stg1 use entity work.stg1(behavioral);

  component stg2
     PORT ( reset_stg2 : IN MVL;
                rin_stg2 : IN MVL;     aout_stg2 : IN MVL;
                ain_stg2 : OUT MVL;   rout_stg2 : OUT MVL;
                U11a0: IN MVL;         U11a1: IN MVL;
                U11b0: IN MVL;         U11b1: IN MVL;
                QC0: OUT MVL;          QC1: OUT MVL;
                QC2: OUT MVL  );
   end component;
   for Ustg2: stg2 use entity work.stg2(behavioral);

   signal bus21, bus22, bus23 : MVL_VECTOR( 0 to 1);
   signal wire11, wire12, wire13, wire14, wire15, wire16, wire17 : MVL;
   signal wire18, wire19, wire110, wire111, wire112 : MVL;

 begin
   wire17 <= RESET;    wire11 <= RIN;    wire12 <= AOUT;
   ROUT <= wire112;  AIN <= wire19;
   wire13 <= A;      wire14 <= B;    wire15 <= C;     wire16 <= D;
   SUM( 1 ) <= bus23( 1 );    SUM( 0 ) <= bus23( 0 );    Carry <= wire18;

   Ustg1: stg1 PORT map ( reset_stg1 => wire17,
                           rin_stg1 => wire11,      aout_stg1 => wire111,
                           ain_stg1 => wire19,      rout_stg1 => wire110,
                           U11a0 => wire13,         U11b0 => wire14,
                           U12a0 => wire15,         U12b0 => wire16,
                           QA0 => bus21( 0 ),       QA1 => bus21( 1 ),
                           QB0 => bus22( 0 ),       QB1 => bus22( 1 )  );
   Ustg2: stg2 PORT map ( reset_stg2 => wire17,
                           rin_stg2 => wire110,     aout_stg2 => wire12,
                           ain_stg2 => wire111,     rout_stg2 => wire112,
                           U11a0 => bus21( 0 ),     U11a1 => bus21( 1 ),
                           U11b0 => bus22( 0 ),     U11b1 => bus22( 1 ),
                           QC0 => bus23( 0 ),       QC1 => bus23( 1 ),
                           QC2 => wire18  );
 end STRUCTURAL;
```

Figure 8: The structural VHDL model for the behavioural VHDL model shown in Figure 5.

stage is written into separate output files. A structural architecture for the whole system is also written to an output file. Only some of the VHDL statements are used for describing the behaviour of the synthesized system. They are the assignments of variables and signals, the IF statement, the WHILE statement and the FOR statement. The operators which can be used to perform comparisons, form boolean equations and perform arithmetic at the right hand side of assignments and the conditions of IF and WHILE statements are as follows:

Boolean	not, and, or, nand, nor, xor
Comparison	$=, /=, <, <=, >, >=$
Arithmetic	$+, -, *, /$

The Leapfrog VHDL simulator was used to simulate the input and output files as well as VHDL models of each event-driven logic module shown in Figure 1.

The library contains standard gates and user-defined gates, such as the event-driven logic modules shown in Figure 1. The VHDL models for each of the event-driven logic modules are implemented individually. They have been simulated and verified independently. The library also contains the components which are generated during synthesizing.

5 The implementation

The synthesizer is implemented using C++. Classes and their member functions are defined to handle the input descriptions, such as String and StringList. To represent the internal circuit models, classes and their members are also defined, such as Terminal, Device, DeviceList, TerminalList, Stage, StageList. Before implementing the synthesizer, rules for partition, conversion, and the production of control circuits must be defined and followed. They are introduced in the following subsections.

Partition rules

When the synthesizer reads the input file, which contains the behavioural descriptions written in VHDL, the corresponding registers for the variables are created. For example, as shown in Figure 7, the registers for the variables QA, QB, and QC declared in Figure 5 are created. They are denoted by the rectangles labelled 'latch'. Then the partition rules are applied to divide statements into different blocks. Such blocks are called stages. Figure 7 contains two stages. The two latches at the left hand side, and the computations at the front of these two latches, are the first stage. The right hand side latch and the computation between the left hand side latches and the right hand side latch are the second stage. The partition rules are summarized as follows:

- If the output variables or the output signals from a set of non-LOOP statements are not input variables or input signals to the other statements, these statements can be evaluated concurrently. This means they can be put into the same stage. For example, the statements $QA := A+B;$ and $QB := C + D;$ shown in Figure 5 are put into the same stage. The input variables QA and QB of the statement $QC := QA+QB;$ are the output variables of the statements $QA := A + B;$ and $QB := C+D;$ respectively. Therefore, the statement $QC := QA + QB;$ is not put into the same stage as the statements $QA := A + B;$ and $QB := C + D;$.

- If a statement contains shift operations it will be put into a stage on it own. The reason is that a special control circuit is required for shift operations. This control circuit is not able to handle other functional statements at the same time.

- If a statement is a LOOP statement, such as FOR or WHILE, it will be put into a stage on its own. The reason is the same as for the shift operations.

- If an IF statement contains unbalanced assignments on its THEN and ELSE branches it will be put into its own stage. An IF statement with balanced assignments on its THEN and ELSE branches can easily be implemented by multiplexers. However, an IF statement with unbalanced assignments must be constructed with feedback and the control circuits are more complex. Therefore, an IF statement with unbalanced assignments will be put into its own stage.

Conversion rules

After partitioning the statements into several blocks the synthesizer starts one of main missions presented in this paper, the conversion of statements to internal circuit models. There are also some conversion rules which help to deal with the conversions. They are illustrated as follows:

- The expression at the right hand side of an assignment may be a value, a mathematical expression, or a logical expression. To convert it just create objects of Device class to denote each of the values, mathematical or logical computations and connections.

- An IF statement with balanced THEN and ELSE branches can be converted into objects of Device class to denote multiplexers and connections. To convert multi-level IF statements into their internal circuit models a FILO method is used to calculate the position of the multiplexer.

- In micropipeline circuits the input to a latch may be from several different sources at different times. This means one signal or variable may appear at the left hand side of several different assignments in the same description. A selector is applied to connect different sources to the input of the corresponding latch if the number of assignments with the same left hand side is greater than 1.

Rules for producing control circuits

Control circuits are required within each stage to deliver the events to related latches for the capture of data as well as to the predecessor and successor stages for handshaking. They are as follows:

- If a stage does not contain any SHIFT operation, LOOP statement or unbalanced 'IF' statements its control circuits are the same as that shown in Figure 2. Otherwise, a control circuit similar to that used in a previous 4-bit asynchronous multiplier circuit [19] will be used. Figures 9 and 10 show the control circuits which are used for WHILE statements and SHIFT operations.

- Following the configuration of a stage, it is necessary to connect its control signals to its predecessor and successor stages. Some event-driven logic modules are required for these interconnections between stages.

After the internal circuit models are constructed the synthesizer produces the structural VHDL output files for each stage and for the whole system.

```
fra_h := fra_f;          exp_h := exp_n;
while (fra_h(2) = '0') and (not exp_h = "0") loop
      fra_r := sh_left( fra_h, 1 );     exp_r := exp_h - '1';
      fra_h := fra_r;                   exp_h := exp_r;
end loop;
fra_x := fra_h;
```

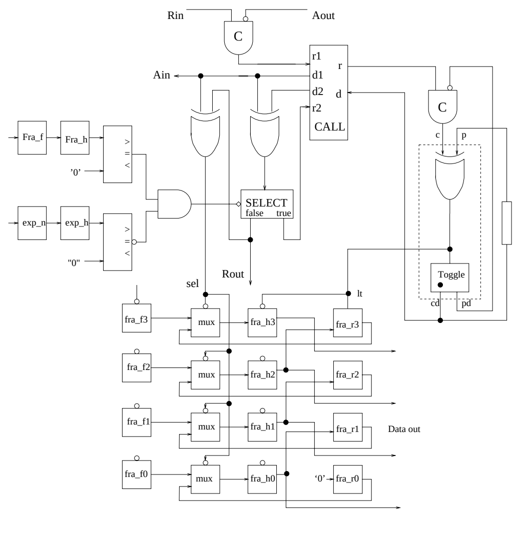

Figure 9: The control circuit for a WHILE loop.

Experimental results

All the rules and steps mentioned in previous subsections were implemented. A test example of the behavioural description shown in Figure 11 was used to test the synthesizer. This example is a small floating point processor with a 4-bit exponent and a 5-bit

```
fra_h := fra_f;
fra_h := sh_left( fra_h, 1 );
```

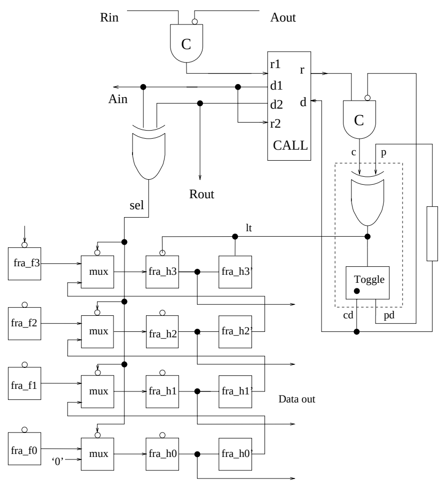

Figure 10: The control circuit for a SHIFT loop.

```
ENTITY F_P_Processor1 IS
  PORT ( RIN, AOUT :IN  MVL;       AIN, ROUT :OUT  MVL;
             EA, EB :IN MVL_VECTOR ( 0 TO 3 );
             MA, MB :IN MVL_VECTOR ( 0 TO 4 );
             OP :IN MVL_VECTOR ( 1 DOWNTO 0 );
             EO :OUT MVL_VECTOR ( 0 TO 3 );
             MO1, MO2 :OUT MVL_VECTOR ( 0 TO 4 );
             RESET :IN MVL );
END F_P_Processor1;
architecture BEHAVIOR of F_P_Processor1 is  begin
 U1: process ( RIN, AOUT, EA, EB, MA, MB, OP, RESET )
    variable comp : MVL;
    variable QEA, QEB : MVL_VECTOR ( 0 TO 3 );
    variable EXP_DIFF1, EXP_DIFF2, EXP_DIFF, EXP_C : MVL_VECTOR ( 0 TO 3 );
    variable QMA, QMB : MVL_VECTOR ( 0 TO 5 );
    variable FRA_C, FRA_DD : MVL_VECTOR ( 0 TO 8 );
    begin
     QEA := EA;    QEB := EB;    QMA(0) := MA(0);    QMB(0) := MB(0);
     QMA(2 to 5) := MA(1 to 4);    QMB(2 to 5) := MB(1 to 4);
     if EA = "0000" then QMA(1) := '0';   else QMA(1) := '1';   end if;
     if EB = "0000" then QMB(1) := '0';   else QMB(1) := '1';   end if;
     exp_diff1 := QEA - QEB;    exp_diff2 := QEB - QEA;
     if QEA >= QEB then comp := '1';  else comp := '0';   end if;
     if comp = '1' then exp_diff := exp_diff1;  else exp_diff := exp_diff2; end if;
     if comp = '1' then exp_c := QEA; fra_c(0 to 5) := QMA; fra_dd(0 to 5) := QMB;
                    else  exp_c := QEB; fra_c(0 to 5) := QMB; fra_dd(0 to 5) := QMA;
     end if;
     fra_c(6 to 8) := "000";  fra_dd(6 to 8) := "000";
     MO1(0) <= fra_c(0);      MO1(1 to 4) <= fra_c(2 to 5);
     MO2(0) <= fra_dd(0);     MO2(1 to 4) <= fra_dd(2 to 5);     EO <= exp_c;
  end process U1;
end BEHAVIOR;
```

Figure 11: A test example.

mantissa. The external port definitions are RIN, AIN, ROUT, AOUT, RESET, EA, EB, MA, MB, OP, EO, MO1, and MO2. RIN, AIN, ROUT, and AOUT are two-phase handshaking control signals. EA, EB, MA, and MB are the exponents and mantissaes of two number repectively. OP is the operation, i.e. $+, -$. RESET is the master clear signal. EO, MO1, and MO2 are the outputs. EO is the bigger of EA and EB. One of the goals of designing the synthesizer is to synthesize a micropipelined coprocessor. Therefore, this example was chosen to test whether the basic operations of the synthesizer are correct. When the synthesis was complete three structural VHDL files and their entity files were automatically generated for the three stages which are partitioned by the synthesizer itself. A structural VHDL file for the whole system is also produced automatically. All the output files were executed on a leapfrog VHDL simulator without error. The expected output values and waveforms were produced.

The structural architecture VHDL files of the three stages contain 286, 288, and 405 lines respectively. A structural architecture VHDL file for the whole system contains 404 lines. As the files are long it is not possible to list all the output files here. We only show the number of the components and the internal signals which were used inside each stage and the whole system. These components are event-driven logic modules, active-low transparent latches, multiplexers, subtractors, comparators, and some basic logic gates. The numbers of components and internal signals which were used inside each synthesized structural VHDL file are shown in Figure 12. The three stages, stg1, stg2, and stg3, were treated as components inside the structural VHDL file for the whole system.

The structural Verilog files of the three stages and the whole system were successfully imported into the Design Framework II database.

6 Conclusions

A synthesizer has been developed which converts behavioural VHDL into asynchronous structural VHDL and Verilog following the micropipeline design style. The synthesizer can support addition, subtraction, multiplication, division, and some logical operations. However, components which execute these mathematical and logical operations must exist in the library. The internal circuit model is similar to a DFG [20]. To optimize synthesized circuits the method used in DFG is proposed to eliminate the redundant circuits and balance the computations within the datapaths.

The research presented in this paper demonstrates that VHDL can be used to describe the behaviour of micropipelined systems. It is also suitable for representing the circuits output from the synthesizer. At the high level the computations of a whole system can be treated as one computation block with micropipelined latches. This simple structure helps the designer to estimate the expected behaviour of an abstract system as early as possible using a VHDL simulator. The simulated results may be stored for comparison with the simulated results from the final synthesized circuits to verify the correctness of the design.

Structural VHDL files	Stg1	Stg2	Stg3	Network
Components & signals				
c2	1	1	1	2
comparator1			4	
comparator4	2	1		
ltlatch1	2	1		
ltlatch4		2	2	
ltlatch6	2			
ltlatch9			2	
mux:2x1	2	1		
mux:2x4			2	
mux:2x6			2	
inverter	1	1	1	
or2		1		
subtractor4		2		
toggle	1	1	1	
xor2	1	1	1	
bus3			2	
bus4	2	2	2	11
bus6			2	2
wire	16	16	16	16
Stg1				1
Stg2				1
Stg3				1

Figure 12: The numbers of components and internal signals inside each synthesized VHDL file.

Synthesis enables the design time to be greatly reduced. With careful partitioning of the system logic functions into several stages, it is possible to improve the system to achieve optimum performance.

References

[1] Sutherland, I.E., "Micropipelines", Communications of the ACM, Vol. 32, No. 6, Jun. 1989, pp. 720-738.

[2] Furber, S. B., Day, P., Garside, J.D., Paver, N.C. and Woods, J.V., "A Micropipelined ARM", 1993 International Conference on VLSI, Grenoble, France, Sep. 6-10, 1993.

[3] Furber, S.B., Day, P., Garside, J.D., Paver, N.C. and Temple, S., "AMULET2e", EMSYS'96 - OMI Sixth Annual Conference, Berlin, 23-25 Sep. 1993, IOS Press ISBN 90 5199 300 5.

[4] IEEE Std, IEEE Standard VHDL Language Reference Manual, Standard 1076-1987, IEEE Inc., 1988.

[5] Miller, R. E., "Switching Theory Vol. II: Sequential Circuits and Machines", John Wiley, 1965.

[6] Plummer, W. W., "Asynchronous Arbiters", IEEE Transactions on Computers, Vol. c-21, Jan. 1972, pp. 37-42.

[7] Burns, S.M. and Martin, A.J., "Syntax-directed Translation of Concurrent Programs into Self-timed Circuits", Proceedings of the 5th MIT Conference on Advanced Research in VLSI, edited by J. Allen and F. Leighton, 1988, pp. 35-50.

[8] Brunvand, E.L. and Sproull, R.F., "Translating Concurrent Communicating Programs into Delay-Insensitive Circuits", Carnegie Mellon, CMU-CS-89-126, Apr. 1989.

[9] Brunvand, E.L., "Translating Concurrent Communicating Programs into Asynchronous Circuits", PhD Thesis, Carnegie Mellon, Sep. 1991.

[10] van Berkel, K., "Handshake Circuits – An Asynchronous Architecture for VLSI programming, 1993, Cambridge University Press.

[11] Nedelchev, I. M., "Asynchronous VLSI Design", Ph.D. Thesis, University of Surrey, UK, Jun. 1995.

[12] Inmos, Occam Programming Manual, 1983.

[13] Hoare, C. A. R., "Communicating Sequential Processes", Prentice-Hall, 1985.

[14] Dijkstra, E.W., "Guarded commands, nondeterminacy and the formal derivation of programs", Communications of the ACM, Vol.18, 1975, pp. 453-457.

[15] Tan, S.-Y., "High Level Modelling of Micropipelines", Master Thesis, University of Manchester, UK, Oct. 1992.

[16] Tan, S.-Y., "A Simulation Model of Micropipelines", Journel of National Taipei Institute of Technology, Vol. 27-2, Taiwan, R.O.C, Jul. 1994, pp. 115-147.

[17] Peterson, J. L., "Petri Net Theory and the Modeling of Systems", Prentice-Hall, 1981.

[18] Rushton, A., "VHDL For Logic Synthesis – An Introductory Guide For Achieving Design Requirements", McGraw-Haill, 1995.

[19] Day, P., "A Micropipelined Multiplier Design", ACiD-WG/EXACT Workshop on Asynchronous Data Processing, Eindhoven University of Technology, Veldhoven, The Netherlands, Dec. 1992.

[20] Gajski, D., Dutt, N., Wu, A. and Lin, S., "High-Level Synthesis – Introduction to Chip and System Design", Kluwer Academic Publishers, 1992.

[21] Cadence, "Design Data Translators Reference", Sep. 1994.