A VHDL-BASED DESIGN METHODOLOGY FOR ASYNCHRONOUS CIRCUITS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

By
Sun-Yen Tan
Department of Computer Science
2001

Contents

List of Figures

List of tables

Abstract

This thesis presents a straightforward approach for synthesizing a structural 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 work presented in this thesis demonstrates that VHDL can be used to describe the behaviour of micropipelined systems. It is also suitable for representing the circuits output by 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 synthesizer can produce 2-phase and 4-phase structural VHDL and Verilog micropipelines. A comparison of 4-phase and 2-phase micropipelines using VHDL is also presented. From the simulations of the various configurations some insight is obtained. By carefully partitioning the computations into several stages of a 2-phase micropipeline, the circuit will speed up. Carefully partitioning the computations into several stages of a 4-phase micropipeline will not give as much speed-up.

The synthesized circuits are taken through to schematics and layouts in the Cadence design environment. Some test examples are presented to show how the synthesizer works. Postlayout simulation using TimeMill is also presented in this thesis. In summary, a design environment for asynchronous circuits has been established based upon the micropipeline style and VHDL, a standard hardware description language.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright & Ownership

Acknowledgements

I wish to thank my supervisor, Professor Steve Furber, for his continuing guidance and encouragement during this work. The help and advice offered by other members of the AMULET group is also greatly appreciated. Thanks are also to my adviser, Dr Doug Edwards, for his encouragement and support. I also wish to thank my co-supervisor, Professor Wen-Fang Yen of National Taipei University of Technology in Taipei, for his encouragement and support.

Particularly I would like to say thanks to Jeff Pepper and David Gilbert for their kind help with Cadence and VHDL problems, Steve Temple for his help with TimeMill and simulation problems, Andrew Bardsley, Craig Farnsworth and Oleg Petlin for the many discussions of VerilogIn to Cadence, Suck-heui Chung, Siamak Mohammadi, Phil Endecott, John Bainbridge, Viv Woods and Nigel Paver for their encouragement and advice, Jianwei Liu for his patience and help with many Verilog, Cadence and other questions, Aristeidis Efthymiou and Zhongchuan Yu for his help with PowerMill and simulation problems.

The research presented in this thesis was funded by the National Science Council of Taiwan. I gratefully acknowledge this support.

I would also like to thank my family and friends for their understanding and patience. Thanks particularly to my wife for her support in so many ways and keeping me going at the end. I do appreciate it.

About the author

The author graduated from the National Taipei Institute of Technology of Taiwan in 1982. Between 1984 and 1991 he worked for the same institute as a teaching assistant. He obtained the degree of Master of Science from the University of Manchester, UK in December 1992. Between 1993 and 1995 he worked for the National Taipei Institute of Technology of Taiwan as a lecturer. In October 1992, the author commenced research work for the degree of Doctor of Philosophy. From June 1995, the author was funded by the National Science Council of Taiwan. The research done during this period is presented in this thesis. Mr. Sun-Yen Tan is also an IEEE member as well as an IEE Associate.

Dedication

To the memory of

My father - Ke-Chang Tan who died in July 1999

To

My mother - Tan Fan, Mei-Chiao

My wife - Hui-Nan Liu

My son - Pi-Chung Tan

My daughter - Pi-Yuan Tan

And my new daughter born in September 2000

Chapter 1

Introduction

1.1 Introduction

Asynchronous design has potential advantages [AsynBib, AsynWeb] over synchronous design, such as no clock skew problem, low power, average case performance and good Electro-Magnetic Compatibility (EMC). The benefits may be most apparent in mobile communication applications and other portable systems which use advanced VLSI technologies. 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 [Suth89]. This micropipeline design methodology has attracted the attention of many researchers, including those in the Department of Computer Science at Manchester University. It is being used to develop high-performance, low-power microprocessing systems by the AMULET (Asynchronous Microprocessor Using Low Energy Techniques) group. Two asynchronous ARM chips, AMULET1 [Furb93, Furb93a, Furb94a, Pave94, Wood97] and AMULET2e [Furb96, Furb97, Gars96a] have been designed. AMULET3i [Furb98], a commercial asynchronous embedded system chip for communication applications, is under development.

1.2 Motivation

Asynchronous circuits are attracting 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. Hazards must be removed from the circuits to ensure that there are no unexpected transitions. However, well structured asynchronous design styles such as micropipelines reduce the difficulty. Event-driven logic modules may be designed by electronic experts. Then designers with less experience can easily build micropipelined circuits using such modules. Most micropipeline designs are presently done by hand. If an automatic synthesis tool is available the design time and cost may be reduced. VHDL is an IEEE standard introduced in 1987. VHDL supports an environment to run simulations of designs which are described using VHDL statements. In this thesis we introduce a method for converting top level behavioural VHDL models into structural VHDL models of micropipelined asynchronous circuits.

1.3 Goals of this work

The principal goals of the work described in this thesis are to:

- Develop modelling techniques which can support the use of the micropipeline design methodology to build correct and optimum asynchronous circuits;
- Find some good solutions for the synthesis of micropipeline asynchronous circuits;
- Create design methodologies for the construction of micropipeline circuits.

1.4 Thesis overview

Only the key ideas and relevant information are given here. More detailed information on aspects of the work can be found in the appendices. Chapter 2 introduces background information on asynchronous systems. The basics of asynchronous systems, classification of asynchronous circuits, handshake protocols, advantages and disadvantages, micropipelines, event-driven logic modules, properties of micropipelines and previous work are presented. Chapter 3 presents the basic idea of the synthesis method, a brief introduction to VHDL models, the library which is used during the synthesis, the procedure for synthesising micropipelined asynchronous circuits and the implementation of a synthesizer. Chapter 4 gives an introduction to 4-phase control circuits and a comparison of 2- and 4-phase circuits. Chapter 5 gives some examples to demonstrate how the synthesizer works. A postlayout simulation using TimeMill is also presented. Different loop operations are discussed. Verification, evaluation and discussion are given in Chapter 6. A comparison with existing techniques, test results and implementation issues are also discussed. Finally, Chapter 7 gives a short conclusion and proposals for further research.

1.5 Contributions

The main contributions made in this thesis are:

- In chapter 3, a synthesizer which converts the behavioural VHDL into structural VHDL and Verilog based on micropipelines is presented. VHDL models of event-drive logic modules and standard logic function elements were created. The partition rules, conversion rules and rules of generating control circuits were derived. A design flow for the construction of micropipeline circuits was proposed.

- In chapter 4, VHDL models of 4-phase control circuits are described. A comparison of 4-phase and 2-phase micropipelines with different delays using VHDL is presented. The synthesizer produces 4-phase structural VHDL and Verilog descriptions of micropipelines.
- In chapter 5, successful layout work and postlayout simulation for the synthesized circuits is presented.

1.6 Publication

The following conference paper was based on aspects of the work presented in this thesis:

Tan, S.-Y. and Furber, S. and Yen, W.-F.,

“The Design of an Asynchronous VHDL Synthesizer”, Proceedings of the Design, Automation and Test in Europe Conference 1998 (DATE 98), Paris, February 1998, pp. 44-51.

Chapter 2

Asynchronous systems

2.1 Asynchronous logic

Asynchronous design does not have a clock to govern the timing of state changes. Storage components for holding internal states and data within asynchronous systems work at different times depending on the preceding and successor circuits. For reliable operation, an asynchronous circuit must be free from critical races and unstable states and liveness checking must be undertaken. If a system contains n storage elements there are 2^n possible states. One approach is to arrange that all storage elements work together to capture data at the same time. Such a system could have a central clock and storage elements that capture internal state signals and data on the rising edge of the clock, the new states being derived from combinational logical circuits that read the old state and inputs. Furber gave a more clear definition and relation between asynchronous and synchronous designs as follows:

> It is a common misconception to view asynchronous design as a single alternative to synchronous design. It is more accurate to view synchronous design as a special case representing a single point in a multi-dimensional asynchronous design space [Furb95a].

Traditional sequential circuits have a global clock to control all the functional units operating in lock-step. Recently asynchronous design has been the subject of increasing interest because the system clock causes serious difficulties in the design of high-performance chips. One limiting factor on the maximum clock rate is clock skew. A fully asynchronous design does not need a global clock and there is no problem of clock skew. Although it is more complicated and difficult to design asynchronous circuits than synchronous circuits [Hauc95], some researchers have presented modelling techniques [Pete81], new design methodologies [Suth89], and synthesis techniques [Meng89] for analysing asynchronous behaviour which have encouraged other researchers to develop asynchronous circuits. The first design of an asynchronous processor was published by Martin [Mart89].

Asynchronous logic circuits have several important advantages over their counterparts in clocked logic. An asynchronous logic function is potentially faster because it works at the average-case delay rather than the worst-case delay. There is no global clock on asynchronous circuits so they will not unnecessarily dissipate power when there is no useful work to do. Asynchronous logic has the potential for low power [Furb95]. Asynchronous logic may be used to implement systems with lower power dissipation.

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 [Suth89].

In this chapter, the basics of asynchronous systems will first be introduced. An overview of the current state of asynchronous design methodologies will then be given, paying particularly attention to the micropipeline approach.

2.2 Some basics of asynchronous systems

Digital circuits may be split into two classifications: synchronous circuits and asynchronous circuits. Circuits are called asynchronous if they do not contain a global clock. The operation of a synchronous system is based on a global clock. Each storage element and subcircuit within a synchronous circuit is triggered by this global clock, i.e., the timing and sequence of synchronous circuits is controlled by the global clock. The clock period must be greater than the delay of the slowest combinational path in an entire system when the computation progresses. Some problems are encountered when using synchronous circuit design with a global clock:

1. A synchronous circuit is divided into several subcircuits. Each subcircuit is triggered by the global clock to perform a subcomputation. The clock period must be larger than the worst-case delay of any subcomputation. Therefore, synchronous circuits always operate at the worst-case conditions.

2. A clock pulse usually triggers a lot of subcircuits. It is necessary to consider the driver current and load on the clock driver circuitry. For example, Greenstreet has pointed out that the clock pulse may completely disappear within a very long pipeline [Gree88].

3. It may be necessary to re-analyse the timing and delays of the entire system, or to re-design completely the system if a subcircuit is replaced by a functionally equivalent but better subcircuit in a system.

Asynchronous circuits are not clock driven but event driven, where an event is the completion of a subcomputation or operation. Subcomputations in asynchronous circuits are triggered by the completion signals of other subcomputations rather than by a central clock. Therefore, asynchronous circuits have the potential for overcoming many of the problems of synchronous circuits.

Each subcircuit of an asynchronous circuit can operate at its own rate. The factors which limit their rate of operation are only the abilities of their neighboring subcircuits which produce their input data and consume their output data. Therefore, a local communications protocol is required when a subcircuit transmits its information to neighbouring subcircuits or receives information from them. With such a local communication protocol, a transmitter subcircuit must indicate that the data is available (i.e. a request signal) and a receiver subcircuit must reply that the data has been accepted (i.e. an acknowledge signal). Asynchronous computation starts the next task cycle once the current task is completed. Since the completion time of each asynchronous operation is task-dependent, an "average" processing speed can be achieved as opposed to the worst-case speed in synchronous designs [Meng89]. A modular design approach, average-case delays and no global clocks can result in an asynchronous system design with low cost, faster speed, and low power consumption. A more detailed definition of an asynchronous system is given elsewhere [Gopa90].

2.2.1 Asynchronous operating modes

Asynchronous circuits can operate in one of two modes. The first is called **fundamental** mode and assumes no further input changes can be applied until all outputs have settled in response to a previous input. The second, **input/output** mode, allows changes to the inputs while the asynchronous circuit is still generating the outputs.

2.2.2 Hazards

One of the most challenging aspects of asynchronous circuit design is the removal of hazards from circuits.

- Hazards are unwanted switching transients that may appear at the output of a circuit because different paths exhibit different propagation delays. Hazards occur in combinational circuits, where they may cause a temporary false-output value. When this condition occurs in asynchronous sequential circuits, it may result in a transition to a wrong stable state. It is therefore necessary to check for possible hazards and determine whether they cause improper operations. Steps must then be taken to eliminate their effect.

- A static hazard is a condition where a single variable change produces a momentary output change when no output should occur. A static 1-hazard causes the output to go to 0 when it should remain at 1. A static 0-hazard causes the output to go to 1 when it should remain at 0. A dynamic hazard causes the output to change three or more times when it should change from 1 to 0 or from 0 to 1. When a circuit is implemented in sum of products form with AND-OR gates or with NAND gates, the removal of static 1-hazards guarantees that no static 0-hazards or dynamic hazards will occur [Mano91].

- An essential hazard is another type of hazard that may occur in asynchronous sequential circuits. An essential hazard is caused by unequal delays along two or more paths that originate from the same input. Essential hazards cannot be corrected by adding redundant gates. To avoid essential hazards each feedback loop must be handled with individual care to ensure that the delay in the feedback path is long enough compared to the delays of other signals that originate from the same input [Mano91].

2.3 Classification of asynchronous circuits

Asynchronous circuits can be categorised into several classes by their features and architectures. A brief description of such a classification scheme is given in the following subsections.

2.3.1 Circuit models

2.3.1.1 Self-timed systems

A self-timed system [Seit80] is either a self-timed element or a 'legal' interconnection of self-timed elements. Self-timed elements must be contained within an equipotential region, i.e., a region in which wire delays are negligible. Within such regions, self-timed data communication is implemented by means of a data-valid wire. For self-timed communication between equipotential regions, where wires may introduce arbitrary delays, two or four-phase handshaking protocols, which will be discussed in a subsequent subsection, are used. Therefore, it is assumed that a circuit can be decomposed into equipotential regions inside which delays in wires are negligible if self-timed techniques are considered [Seit80]. Self-timed circuits are delay insensitive, i.e., their behaviour does not depend on any relative delays among physical elements [Unge69] [Seit80] [Meng89].

2.3.1.2 Delay-insensitive circuits

Delay-insensitive circuits are a special type of asynchronous circuit composed of protocol-based modules, i.e., a circuit is delay-insensitive if the correct operation of the circuit does not depend on any assumptions about the delays in the wires or the operators of the circuit. Such circuits do not use a clock signal or knowledge about delays [Mart89], which means that there is no assumptions about delays for delay-insensitive circuits within the system. Any gate or interconnection may take an arbitrary time to propagate a signal. This property guarantees that any correctly functioning subcircuits may be composed together and continue to operate correctly [Brun89]. Therefore, delay-insensitive techniques are particularly attractive for VLSI synthesis [Mart91].

The use of delay-insensitive circuits offers several advantages over synchronous circuits. Delay-insensitive circuits are more robust and potentially faster than their clocked counterparts, since their correct operation does not depend on worst-case delay assumptions. A delay-insensitive circuit can be formally derived by program-transformation from a high-level program description [Mart92].

The purest form of circuit is delay-insensitive and uses dual-rail encoding with transition signalling. A transition on one wire indicates the arrival of a zero, a transition on the other the arrival of a one. The levels on the wires are of no significance. Such an approach enables the design of fully delay-insensitive circuits, and is therefore ideal for automatic transformation into silicon layout as the rather arbitrary delays introduced by the layout compiler can't affect the functionality (though they will affect the performance) [Furb95a].

2.3.1.3 Speed-independent circuits

A speed independent design is tolerant to variations in gate speeds but not to propagation delays in wires [Mart89]. Therefore, speed-independent circuits adhere to a less strict model, i.e., gates may have an arbitrary propagation delay but transmission along wires is instantaneous. Some issues of the modelling, specification, and verification of speed-independent circuits have been illustrated through the consideration of self-timed queues [Dill92]. There are no race conditions and hazards within speed-independent circuits and such circuits operate at maximum speed (i.e. they allow two or more signals to be excited simultaneously.). More detail about the advantages and disadvantages of speed independent circuits is given elsewhere [Mill65].

2.3.1.4 Bounded-delay circuits

If the circuit only functions when the delays are below some predefined limit the circuit is known as bounded-delay.

2.3.2 Handshake protocols

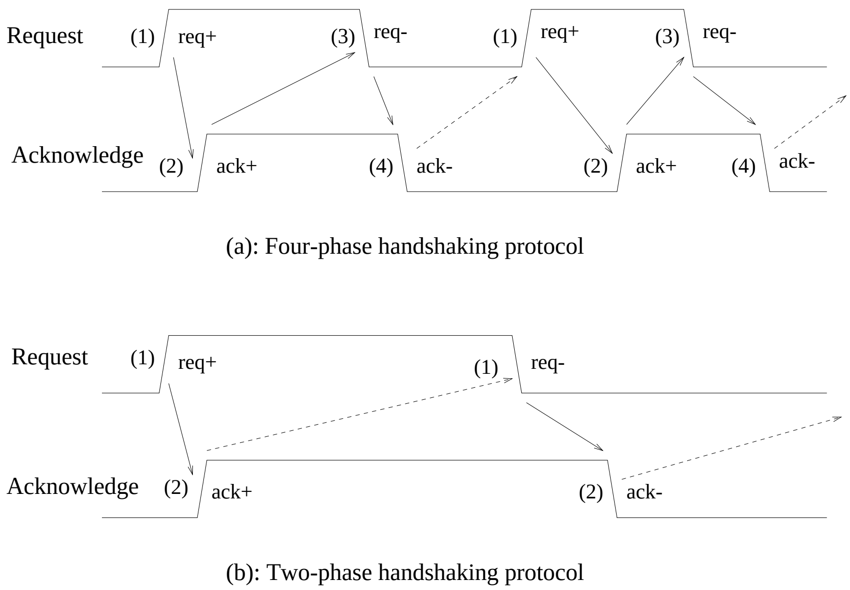


Figure 2.1: Four-phase and two-phase handshaking protocols

2.3.2.1 Four-phase handshaking protocol

Two systems may communicate using a four-phase handshaking protocol [Seit80] as shown in Figure 2.1(a). The two control wires which are used to indicate that the data is available (the request signal, req) and the data has been accepted (the acknowledge signal, ack) will return to their initial state after they have activated. This handshaking protocol frequently uses the rising transitions to initiate operation and the falling transitions to reset signals. The operating cycle is (1) request active, (2) acknowledge active, (3) request inactive and (4) acknowledge inactive, i.e., the cycle is $req+$; $ack+$; $req-$; $ack-$. The request signal adopting its active level indicates that data is ready for transfer. The acknowledge signal adopting its active level indicates that the transfer is complete. When the sender detects that the transmission has finished, it will inactivate its request signal. Then the receiver will inactivate its acknowledge signal. Therefore, the four-phase handshaking protocol is also called return-to-zero signalling. Most asynchronous systems use this protocol to synchronize operations between neighbouring subsystems.

2.3.2.2 Two-phase handshaking protocol

In a two-phase handshaking protocol [Seit80] which is shown in Figure 2.1(b), there are also two control wires. One is used to indicate the data is available (the request signal, req). The other is used to indicate the data has been accepted (the acknowledge signal, ack). In this signalling scheme, rising transitions and falling transitions on the control wires have the same meaning. They are called the request event or the acknowledge event. The phases of the two-phase handshaking protocol are: (1) request event and (2) acknowledge event. For example, $req+$; $ack+$ is an active two-phase cycle. The next cycle for this interface would be $req-$; $ack-$. The signals do not need to return to their initial state after they have activated. The opposite transition has the same meaning and may be used to represent the subsequent event. If the request wire goes to a 'logic high' level for representing a current request event, the next request event will be represented by the wire going to a 'logic low'. In this mechanism, it is the occurrence of a transition rather than a level which carries the information. Therefore, two-phase signalling is also called non-return-to-zero, transition signalling or event signalling. Transition signalling is potentially useful as it allows the transmission of information without requiring that signals return to their initial state after each event. This removal of redundant signal transitions improves the efficiency of the communication system with consequential higher performance [Will90].

The circuits working on this protocol have low power dissipation. Some event-driven logic modules have been presented [Suth89] which can be used to compose control circuits easily. Also there is a four-phase to two-phase converter with "quick return" which has been published [Gopa90].

2.3.3 Data encoding

2.3.3.1 Single-rail encoding

Single-rail encoding communicates information in a conventional level encoded manner. One wire is required for each bit of information. If the information is a data value, then a typical encoding would use a High (Vdd) level to correspond to logical '1' and a Low level (Vss) to represent a logical '0'.

2.3.3.2 Dual-rail encoding

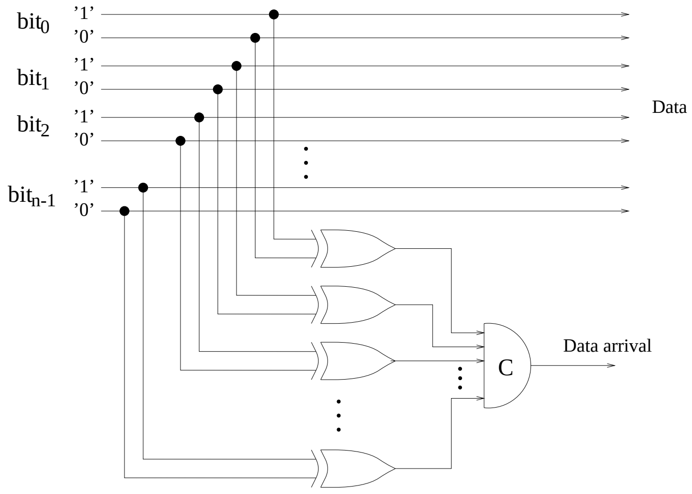


Figure 2.2: The circuit for detecting the data arrival in dual-rail encoding

In this encoding system, each boolean is implemented as two wires. An *n-bit* data value is denoted by $2^n + 1$ code combinations on $2n$ wires. It is used to indicate that every bit of a data word has arrived at a defined logic stage before the evaluation can commence within a truly delay-insensitive circuit. This means that the timing information must be encoded with the data. The usual way of achieving this is to use two separate wires to represent each bit of data, one wire for transmitting the logic '0', the other for transmitting the logic '1'. The arrival of a bit is then denoted by a transition on the relevant wire. Figure 2.2 shows the circuit which can be used for detecting the data arrival in dual-rail encoding. The arrival of each bit can be detected using an XOR gate. The arrival of the entire data can be detected by the arrival of all the data bits. This detection can be implemented by a Muller C-element. Because this encoding technique requires two wires for each data bit, designs using the technique require about double the chip area needed by an equivalent synchronous circuit.

2.4 Advantages and disadvantages

Asynchronous circuits have the potential to overcome some fundamental problems encountered in synchronous circuit design. The possible advantages are as follows:

1. *No clock skew.* There is no global clock for synchronizing the operation of each subcircuit. Consequently, asynchronous circuits have better composability. They more readily allow concurrency due to the absence of global clocks.

2. *Potential low power.* Asynchronous circuits can have low power dissipation. Clock drivers may dissipate most of the power of a synchronous circuit. Power consumption may be a significant cost factor due to the costs of packaging for high power dissipation.

3. *Design programmablity.* Some basic modules which can easily be used to compose asynchronous circuits have been published [Suth89]. The high-level design of asynchronous systems is then like constructing the flow diagram of a program, which is relatively straightforward.

4. *Easy component upgrade.* Improving the performance of an asynchronous system can be achieved by replacing a critical subsystem. Re-designing and re-analysing the entire system is not necessary.

5. *Average-case performance.* Asynchronous circuits tend to reflect the average-case delay rather than the worst-case delay. However, the design style may slow down the operating speed of the circuit. For example, the speed of a circuit which uses a four-phase handshaking protocol is usually slower than one using a two-phase handshaking protocol, because the return-to-zero of the four-phase handshaking protocol increases the operating cycle.

6. *Low electro-magnetic interference.* Electro-magnetic compatibility problems including excessive electro-magnetic emission and susceptibility to interference from electro-magnetic fields have been increasingly important issues since hand-held wireless electronic products like cellular phones are required to meet rigorous electro-magnetic compatibility specifications [Chun98]. Asynchronous design may offer reduced noise emission since there is no interference generated by regular clocking and asynchronous signals spread their energy over a broader and lower frequency range and do not generate high-energy levels at any one frequency. In synchronous design, clock speeds have already reached 500 Mhz, and gigahertz processors will probably be available within the next few years. At those clock rates, even short transmission lines will act as antennas, producing unwelcome amounts of electro-magnetic interference and cross-talk [Miro97, Liu97].

There are also some disadvantages as follows:

1. *More area for interconnections.* Asynchronous circuits need more wiring for handshaking between neighbouring subcircuits. Therefore, asynchronous circuits usually need a greater silicon area.

2. *Difficult design.* Low-level asynchronous design is harder and more constrained than synchronous design, due to the hazard problem and race conditions.

Sutherland in his 1988 Turing Award lecture [Suth89] discusses a class of asynchronous circuits called "micropipelines". Here, circuits are designed using a set of functional modules. This greatly simplifies the design process. Micropipelines will be described in more detail in the next section.

2.5 Micropipelines

Pipelining [Henn90] is an implementation technique whereby multiple instructions or data are overlapped in execution. A pipeline is like a production line, where each step in the pipeline completes a part of the instruction. Each such step is called a pipe stage or a pipe segment. The throughput of the pipeline is a measure of how many instructions or data items can pass through it per unit time. The latency of a pipeline is a measure of how long it takes a single instruction or data item to pass through it. When designing pipelines, both the latency and throughput are important. The time taken for an individual instruction or data item to traverse the pipeline, i.e. the latency, must be minimised. The number of instructions or data items processed in a given time i.e., the throughput, must be maximised. A synchronous pipeline consists of several stages which operate in lockstep. There is a global clock which is used to define when all stages must have completed their operations and data is ready to be transferred to the subsequent stages. The minimum clock period of the pipeline is determined by the time taken for the slowest pipeline stage to complete its operation. The latency of the pipeline is the clock period multiplied by the number of pipeline stages. Therefore, the performance is limited by the slowest stage.

Some research into asynchronous pipelines has shown that there is the potential for high performance and concurrency in asynchronous pipelines [Komo88] [Gino90] [Will90]. There is no central clock in an asynchronous pipeline. Each stage operates at its own rate. Each stage only needs to wait for the operation of the neighboring stages to be complete. The operation of most asynchronous pipelines is based on a four-phase handshaking protocol.

Sutherland [Suth89] describes a methodology called Micropipelines for the design of asynchronous systems using a two-phase bounded-delay protocol. 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 [Suth89]. Before introducing micropipelines, a brief description of the two-phase bundled data convention will be given.

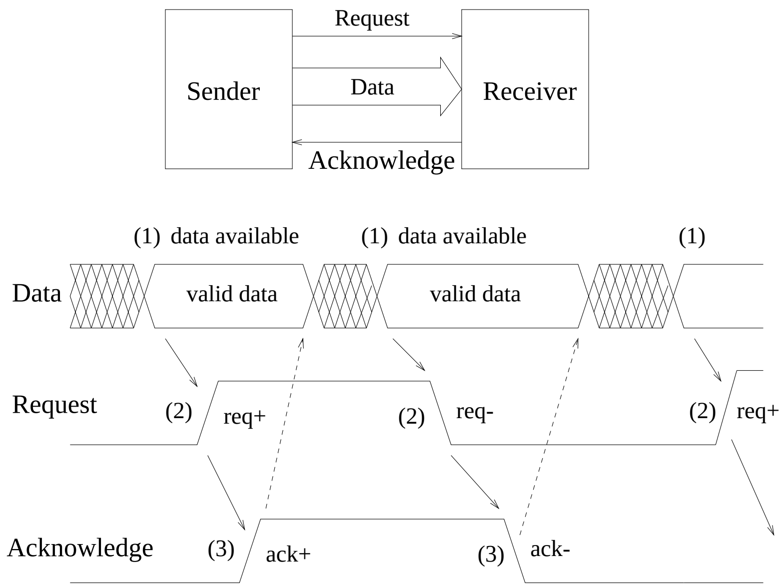


Figure 2.3: Two-phase bundled data convention

2.5.1 The two-phase bundled data convention

Bundled data has one wire for each data bit and a separate wire to indicate the timing. A "two-phase bundled data convention" is a communications 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. The data signals use a traditional data representation which is similar to that used in synchronous circuits, such as the *8-4-2-1 code* ... etc. Figure 2.3 illustrates this signalling. When using a two-phase bundled data convention for transmitting data between two systems, it must be noted that the data signal delays must be no longer than the request signal delays to the point where the data signals and the request signal arrive at the receiver.

2.5.2 Event-driven logic modules

Various circuits have been devised for controlling transition signals. These have been called event-driven logic modules [Suth89]. They are used in the next section to construct the control circuits of micropipelines. Figures 2.4, 2.5, 2.6, 2.7, 2.8 and 2.9 show the circuit symbols of these event modules.

The basic modules are :

Muller C-element

> A C-element [Mill65] performs the rendezvous function for events. A Muller C-element is shown in Figure 2.4. It is a very important element in asynchronous circuit designs. When both of its inputs are

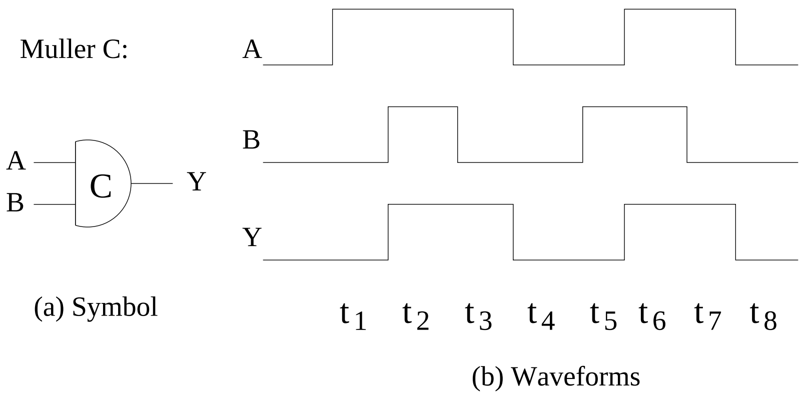
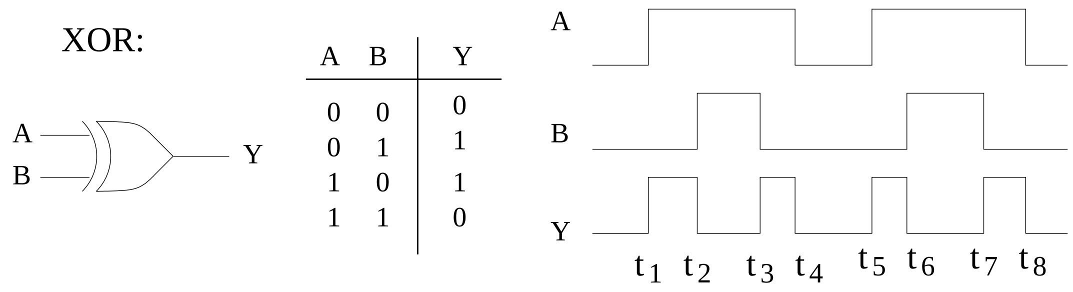


Figure 2.4: The circuit symbol and waveforms of a Muller C-element

in the same logical state, i.e. it has received an event on both of its inputs, its output will have the the same state as the inputs, i.e. there is an event on its output. When the two inputs are different, i.e. only one event has arrived on either of its two inputs, it retains its previous state and does not change its output, i.e. there is no event on its output. Thus only when both of its inputs have received an event, will an event appear at its output. For example, the input signal **A** of Figure 2.4 is changed from logic '0' to logic '1' at time t_1. Because the signals **A** and **B** have different logical states the output signal **Y** does not change at time t_1. The output signal **Y** will go high when the input signal **B** becomes logic '1' at time t_2. Subsequently, the logical state of the output signal Y will also be changed at times t_4, t_6 and t_8. For three or more inputs, it is required that all of them receive an event before producing an event at its output. Therefore, such elements are **AND** elements for events. These elements can be used to treat the forking and joining connection problem of circuits.

XOR circuit

An XOR circuit performs the merge function for events. Its symbol is shown in Figure 2.5(a). Such circuits are also called the OR element for events. The truth table of Figure 2.5(b) shows that the output of

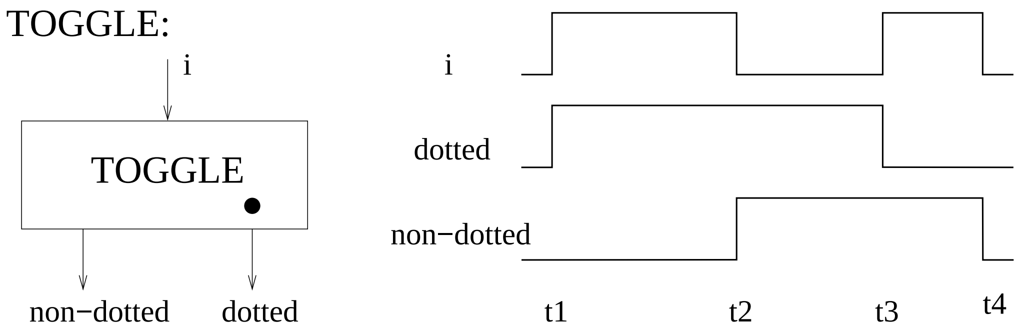


(a) Symbol (b) Truth Table (c) Waveforms

Figure 2.5: The circuit symbol, truth table and waveforms of an XOR

the XOR is logic '1' if its two inputs have different logical states and logic '0' if its two inputs have the same logical states. As shown in Figure 2.5(C) the signals **A** and **B** have different states between times t_1 and t_2. therefore, the output signal **Y** is logic '1' between times t_1 and t_2. The occurrence of an event is denoted by changing the logical state of the input signals. When one of its input changes state, its output also changes state. Therefore, when it receives an event on either of its two inputs, it will produce an event on its output. For more than two inputs, XOR performs the multiple-input **OR** function for events. Such elements can be used to treat the uncertain source connection problem of circuits.

Toggle circuit



(a) Symbol (b) Waveforms

Figure 2.6: The circuit symbol and waveforms of a Toggle

A Toggle circuit sends events from an incoming stream alternately to one of two outputs. A Toggle circuit is shown in Figure 2.6(a). After the master clear signal activates, the odd events which arrive on its input will be sent to its dotted output and the even events which arrive on its input will be sent to its non-dotted output . Such elements can be used to treat the alternate connection problem of circuits. For example, as shown in the Figure 2.6(b) the logical state changes of the input signal **i** at the times t_1 and t_3 will be sent to the dotted output. The logical state changes at the times t_2 and t_4 will be sent to the non-dotted output.

Select module

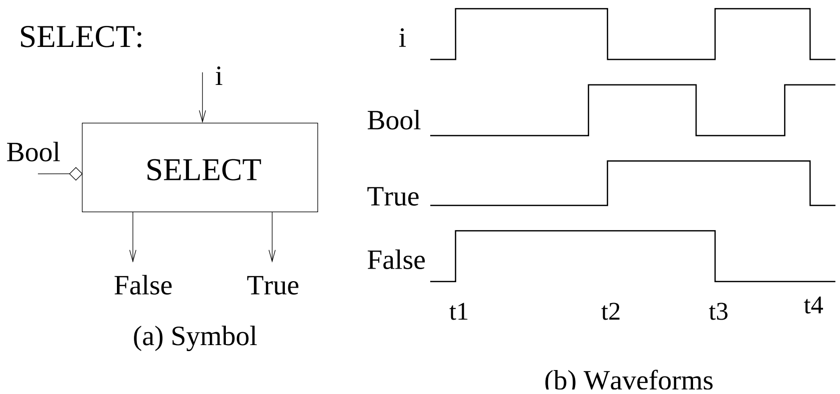


Figure 2.7: The circuit symbol and waveforms of a Select

A Select module can be used to deliver events to one of two outputs under the control of a boolean signal. A Select module is shown in Figure 2.7(a). An incoming event which arrives on its input will be sent to the output labelled "True" or the output labelled "False", depending on the value of a data input. The data input is a Boolean value. If the data is true, the incoming event is sent to the the output labelled "True". If the data is false, the incoming event is sent to the the output labelled "False". The Boolean value must arrive before the incoming event. Therefore the Boolean data and the incoming event must be treated as a bundle. Such elements can be used to treat the selective connection problem of circuits. For example, Figure 2.7(b) shows that the value of the signal **Bool** is logic '0' at the time t_1 and the logical state change of the input signal **i** is sent to the output labelled "False". At the time t_4 the value of the signal **Bool** is logic '1' and the logical state change of the input signal **i** is sent to the output labelled "True".

Call module

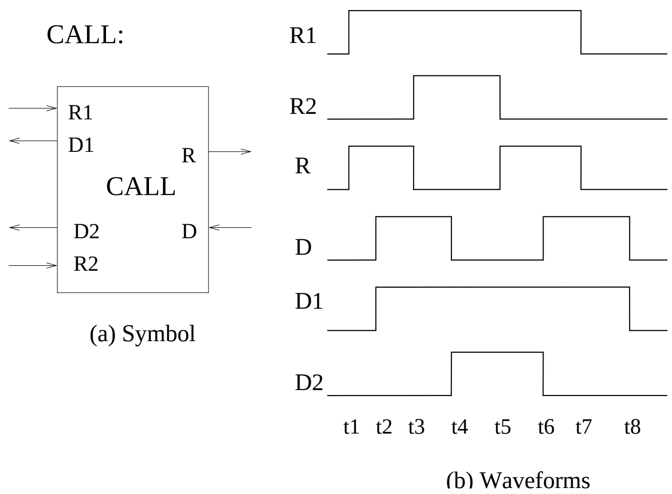


Figure 2.8: The circuit symbol and waveforms of a Call

A Call module can be used to give shared access to a sub-process from two independent (but mutually exclusive) higher-level processes. A Call module is shown in Figure 2.8(a). It remembers the event which is most recently received on its input and correctly returns an event on the output corresponding to the incoming event when the called procedure has finished. For example, as shown in Figure 2.8(b) there is a transition on the input R_1 at the time t_1 and the corresponding output event is shown on the signal R which is connected to a common shared circuit. When the process is finished there is a transition on the the input D. This transition is then sent to the output D_1.

The occurrence order of the events is R_1 R D D_1 or R_2 R D D_2. At the time t_5 there is a transition on the input R_2 and the corresponding event is sent to the output R. After the job is finished there is a transition on the the input D. This transition is then sent to the output D_2. We need to note that the Call module will operate properly only if each call completes before a subsequent call occurs. For this reason, we need an element to treat the concurrency problem and to ensure that the subsequent event does not happen before the previous call has finished. In general, such elements can be used to treat the procedure connection problem of circuits.

Arbiter module

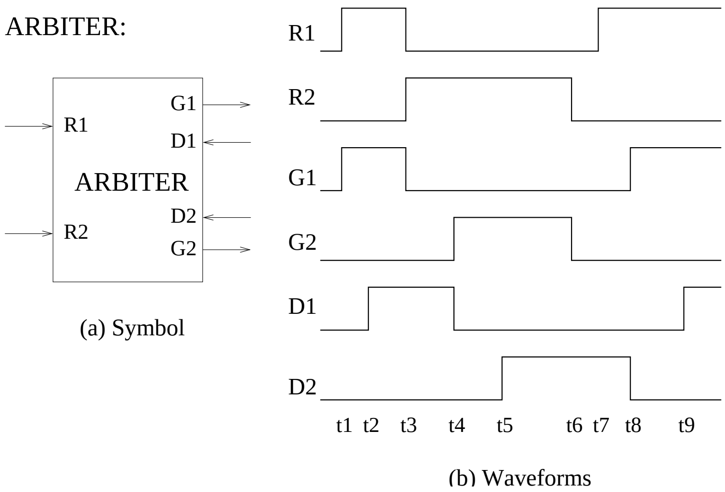


Figure 2.9: The circuit symbol and waveforms of an Arbiter

In asynchronous pipelines, when two events from two source pipes need to join into a single pipe without strict ordering, i.e. such 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. It is an important element in asynchronous pipeline circuit designs and there has been much research into the design of such modules [Plum72] [Pear75] [Calv86]. An Arbiter module is shown in Figure 2.9(a). The sequence of two events without strict ordering can be clearly decided, and then a grant event for only one of them will be produced on the corresponding output. It delays subsequent grants until the earlier grant has been finished. Therefore, only one grant at a time is sent. For example, Figure 2.9(b) shows that there is a logical state change on the input R_1 at time t_1. The corresponding output transition is on the signal G_1 at time t_1. When the process connected to the signal G_1 is finished there is a completion event on input D_1 at time t_2 to inform the Arbiter that it may send a subsequent grant if another request is pending. At time t_3 transitions arrive on inputs R_1 and R_2 simultaneously. The Arbiter sends the grant of R_1 on output G_1. The grant of R_2 must wait for the arrival of the completion event on input D_1. The grant of R_2 is sent out on G_2 when the transition of D_1 arrives at time t_4. The completion event of G_2 occurs at ime t_5. An Arbiter element can be connected directly to the Call element to enforce mutual exclusivity. Arbiter trees can be used to implement arbiters which have more than two inputs [Plum72].

2.5.3 A micropipeline stage

Figure 2.10 is an example of a micropipeline stage. A micropipeline stage consists of control circuits, combinational logic circuits and storage elements. When 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 the data, it will place an event on ‘Aout’ to clear the storage elements

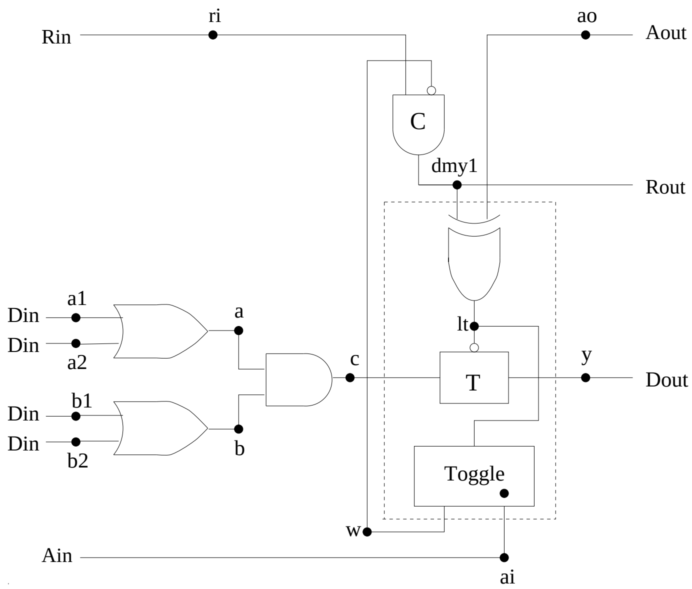


Figure 2.10: A micropipeline stage

of the current stage and enable the current stage to capture subsequent data.

The operation cycle of a micropipeline is (1) data available, (2) request event, and (3) acknowledge event. The return to inactive state action is not necessary. Therefore, the operating speed of micropipelines is faster than that of four-phase handshaking protocol pipelines. 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 with an inverter. It is used to implement the rule "if predecessor and successor differ in state then copy predecessor's state else hold present state" [Suth89]. 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 2.11 shows how five micropipeline stages are connected in series. Figure 2.12 shows a micropipeline stage that forks into two micropipeline stages. Figure 2.13 shows a micropipeline stage that forks into two micropipeline stages which subsequently join into one micropipeline stage. Figure 2.14 shows a micropipeline stage which selectively connects to one of two micropipeline stages which call the same output micropipeline stage.

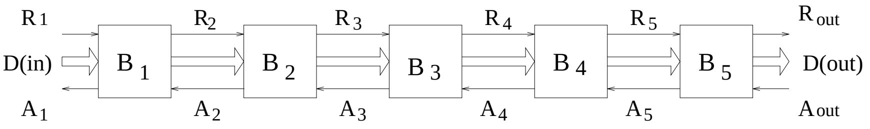


Figure 2.11: Five serially connected micropipeline stages

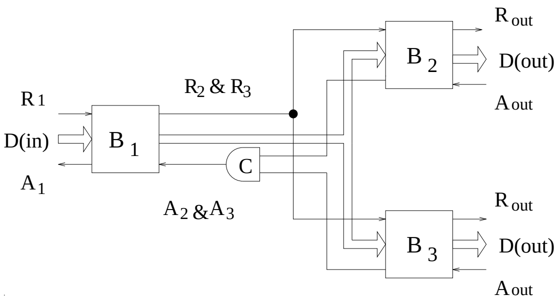


Figure 2.12: A micropipeline stage forks into two micropipeline stages

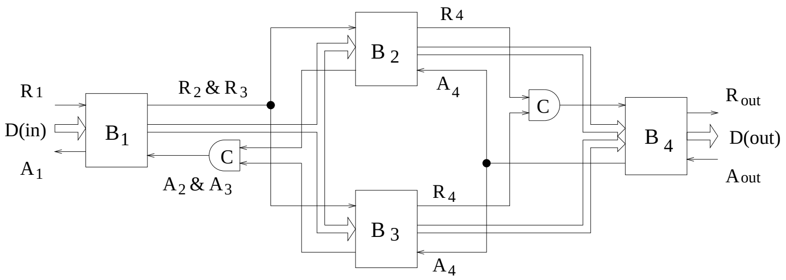


Figure 2.13: A micropipeline stage forks into two micropipeline stages which join into one micropipeline stage

2.5.4 Properties of micropipelines

Micropipelines are one kind of particularly simple and elastic pipeline in which computations can be implemented by combinational logic functions. If a micropipeline has no computation in it, it is a FIFO (Figure 2.15). Inserting some FIFOs in between the stages to accommodate waiting instructions can improve the performance of pipelines.

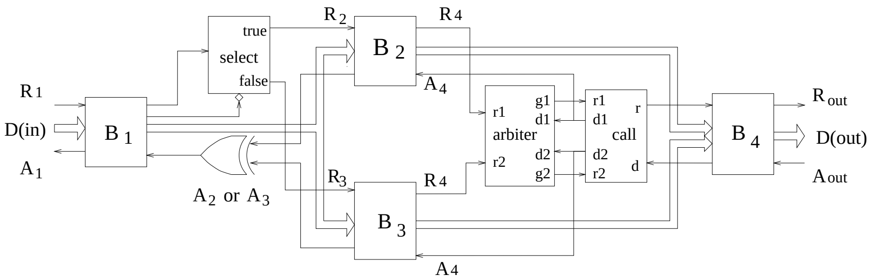


Figure 2.14: One micropipeline stage(B_1) selectively connects to one of two micropipeline stages(B_2 or B_3) which call the same output micropipeline stage(B_4).

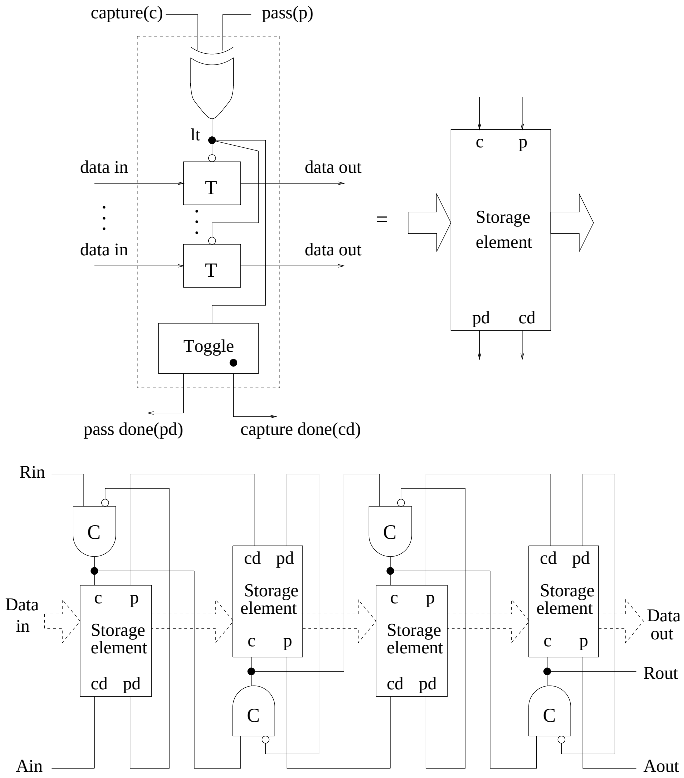


Figure 2.15: A micropipeline FIFO

Within such a FIFO, if the states of all of the Muller C-element which are used to control the storage elements have both their inputs in different states, this situation indicates that the FIFO is empty. On the other hand, if the states of all the Muller C-element are such that both their inputs have the same state, this situation indicates that the FIFO is full. To initialize a FIFO or a micropipeline into the empty state, all inputs of the Muller C-elements within such a FIFO or micropipeline can be set to the appropriate state by a master clear signal.

When a micropipeline with or without computation is full, the occurrence of the acknowledge event must wait until the second stage has consumed the data held in the first stage. After it issues an acknowledge, the subsequent data can enter into it. This means that the input device connected to the micropipeline cannot change its input data until it has received an acknowledge event for the present data. A similar handshake will occur at the output end of the micropipeline. Therefore, correct data with average-case speed can be propagated through the micropipeline.

2.5.5 The advantages of micropipelines

In addition to the general advantages of asynchronous circuits, micropipelines have several further advantages as follows:

1. In micropipelines, the use of a two-phase bundled data convention not only simplifies the initial system design but also permits the rapid mid-life upgrade of systems as new technology becomes available. Components are easily replaced by new ones. The system can more readily maintain the characteristic of high performance and low cost.
2. Because event-driven logic modules provide conditional operators, procedure calls, and other elements familiar to programmers, this makes such modules easy to compose into loops and other structures similar to those

found in programs. The design of control circuits is rather like making block diagrams for programs. Complex functions are easy to compose from simple modules which provide basic functions already familiar in programming. More complex systems can be built by composing a hierarchy of the basic modules and previously designed compositions. It is very easy to understand and to realize.

3. The micropipeline design methodology reduces the design time and cost of complex asynchronous systems. It provides an exciting alternative to conventional hardware design. The micropipeline design methodology simplifies system design because simple modules and their compositions can be further used to construct large systems.

4. Returning to the inactivate state is not necessary. Such micropipelines save time and energy costs on the return transition. Therefore, when micropipeline techniques and event-driven modules are used to design complex asynchronous systems, the systems have the property of high performance and low power dissipation. These techniques and modules are being used to develop high performance and low power microprocessing systems.

2.5.6 Micropipeline designs

Gopalakrishnan investigates some unusual Micropipeline circuits [Gopa93b] and the dynamic recordering of instruction sequences using a modified Micropipeline [Gopa92]. The AMULET group at Manchester University have investigated various aspects of Micropipeline designs. Furber and others give an overview of the design of the asynchronous ARM processor [Furb93, Furb93a, Furb93c, Furb94, Wood97, Gars93, Pave94, Day95]. Kelly constructed a bipolar implementation of the asynchronous ARM processor [Kell93]. Mehra and Garside investigated cache structures suitable for an asynchronous processor [Mehr92, Mehr95]. Tan investigated the high-level modelling of Micropipelines [Tan92]. Endecott investigated the architectural features desirable for low power asynchronous microprocessors [Ende93]. Theodoropoulos investigated the modelling and simulation of asynchronous computer architectures using Occam as a description language [Theo95a].

2.6 Synthesis techniques

There are three predominant styles of automatic synthesis of asynchronous circuits [Pave94]. They are compilations from a high level language to a VLSI circuit, using graphical descriptions and using asynchronous finite state machines. Compilation schemes derive asynchronous VLSI circuits from high-level languages based upon Communicating Sequential Processes (CSP) [Hoar78, Hoar85] and Occam [Inmo83] derivatives. Brunvand [Brun89, Brun91] presents a technique for compiling a subset of Occam into delay-insensitive control circuits with bounded-delay datapath elements. Martin [Mart86, Mart89d, Mart90d] and Burns [Burn87, Burn88a] describe a technique for translating from a “program notation” based on CSP and Dijkstra’s guarded-commands [Dijk75] to a four-phase delay-insensitive circuit. Van Berkel at Philips Research also describes a compilation system [Berk88a, Berk88b, Niel88] based upon CSP and Dijkstra’s guarded-command language. Gopalakrishnan and Akella present a design environment for the specification, simulation and synthesis of asynchronous circuits [Gopa93b, Akel90, Akel92, Akel92a, Gopa93a], their specification language is also based upon CSP.

Asynchronous systems can be specified using Petri nets [Pete77]. A methodology for synthesizing speed-independent circuits from State Transition Graphs (STGs) was proposed by Chu [Chu85, Chu86, Chu86b, Chu87]. STGs are similar to Petri Nets [Pete77, Pete81] whose transitions are labelled with signal names and whose places form the arcs of the graph. When a transition fires in an STG the associated signal in the circuit changes. Several researchers also use the STG format to described asynchronous circuits and they have developed algorithms for STG transformations and synthesis [Lin91, Lin92, Meng89, Vanb90, Vanb93, Yako92].

Davis, Coates and Stevens describe a collection of synthesis tools (MEAT) for generating hazard free asynchronous finite state machines [Coat93]. Nowick and Dill have presented a technique for the automatic synthesis of asynchronous state machines using a local clock [Nowi91a, Nowi91, Nowi92c]. Together with Yun they have also proposed a technique for the synthesis of 3D state machines [Yun92a] and an extension to the specification of burst-mode to allow more concurrency [Yun92]. Much of the work of Davis, Dill and Nowick is being integrated into a single tool called Stetson [Davi93c].

Some asynchronous design tools are available on the Internet [AsynTool].

2.7 Issues and previous work

2.7.1 Micropipelines issues

Micropipelines have the following disadvantage.

> The communication between stages or storage elements in micropipelines is not fully delay-insensitive if control and data signals are considered separately. The data signals must propagate through a micropipeline faster than the control event through its control circuits. i.e., The delays in data transmission must be less than the delays in transmitting the request signal. Special delays are sometimes required in the control path when significant processing logic is put between storage

> elements in the data path. The combinational logic circuits for computations within the micropipeline must be carefully designed. High speed electronic techniques, such as dynamic CMOS techniques, may be applied when designing this part of the circuit.

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. In this thesis we introduce a straightforward approach in converting top level behavioural VHDL models into structural VHDL models and Verilog models based on micropipelined asynchronous circuits (See chapter 3).

VHDL is an IEEE standard introduced in 1987 [Ieee87] 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.

2.7.2 Previous work

Asynchronous design techniques have been explored since the early 1950's. Most of the theory is based on the early work done by a number of researchers including Huffman, Muller, Bartky, Miller, Unger, Keller, and Seitz [Huff54, Huff57, Mull57, Mill65, Unge69, Kell74, Seit70, Seit80]. Work continues to focus on implementing asynchronous state machines [Moln85].

The Macromodules project [Clar67, Orns67, Clar74] in the early 1960's demonstrated the modular design of asynchronous circuits and the ease with which designs could be put together. The early work of Miller [Mill65], Molnar [Clar74, Moln83, Moln85], Seitz [Seit70, Seit80], Unger [Unge69], Huffman [Huff54] and Keller [Kell74] laid the foundations on which most of the recent research in asynchronous design is based. The asynchronous phenomenon of metastability was discovered during the early work [Chan73, Kinn76]. As the general aspects of asynchronous design methodologies are beyond the scope of this thesis a brief description of current asynchronous design styles is presented here; a more in-depth survey and bibliography are presented elsewhere [Gopa90, Hauc95, Davi95, Brzo95a, AsynBib].

Previous work has been carried out in translation from programs to asynchronous circuits [Burn88, Brun89, Brun91, Brun91a, Berk93, Nede95]. A synthesis method of transforming concurrent programs based on a variant of CSP [Hoar85] into self-timed circuits has been developed by S. Burns in 1988 [Burn88]. A subset of OCCAM [Inmo83] based on CSP is used to describe computations and a compiler has been constructed that automatically performs the translation and transformation [Brun89, Brun91, Brun91a]. The translation of Tangram [Scha95] programs based on CSP and guarded-command language [Dijk75] into handshake circuits and then VLSI circuits has also been developed by Philips Research Laboratories [Niel88, Berk88b, Berk93]. A silicon compiler which converts a high-level programming language, OCCAM(async), to asynchronous CMOS circuits has been described [Nede95]. However, previous work has not used VHDL which is an IEEE standard introduced in 1987 [Ieee87]; this supports an environment within which designs may be described using VHDL statements and then simulated easily. Sutherland's micropipeline methodology is the most efficient and successful asynchronous architectural methodology [Suth89]. The author is interested in developing a novel method for translating VHDL behavioural descriptions into micropipeline circuits.

Chapter 3

The synthesis method

Figure 3.1 shows a block diagram of a synthesiser. A synthesizer performs translations or compilations from specifications into the implementations of circuits. The specification describes the computation function of synthesized circuits. The libraries contain the standard gates and modules which are used to construct the synthesized circuits by the synthesizer. The specification may be denoted by languages, graphics or mathematics. Language based CSP [Hoar78, Hoar85] and Occam [Inmo83], graphic based Petri nets [Pete77] and STGs [Chu85, Chu86, Chu86b, Chu87], and mathematic based FSMs and Algebras [Jose90b, Jose91] have been used previously for synthesizing asynchronous circuits. An alternative, IEEE standard language, VHDL, is used for the specifications in this work.

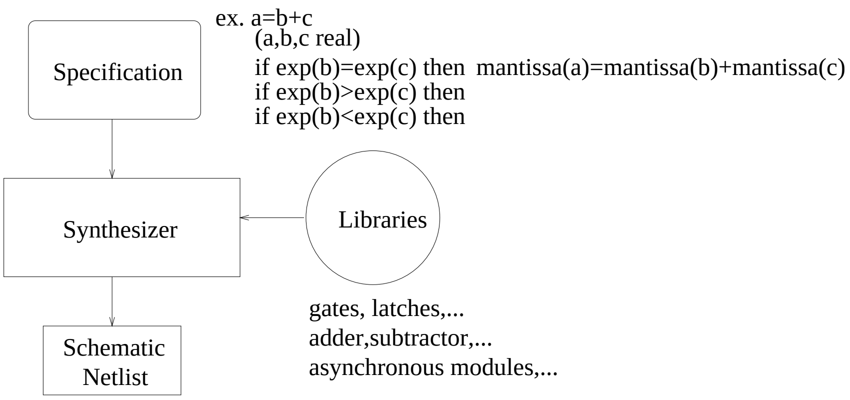


Figure 3.1: A block diagram of a synthesizer

The creation of component libraries is another key issue for synthesis. The libraries used here contain VHDL models and various views of two-phase and four-phase modules and standard gates which were created either for this research or were used to develop AMULET1 [Furb93, Furb93a] and AMULET2e [Furb96].

The output of a synthesizer may be the netlist or the schematic. The choice is subject to how the output interfaces to other design tools to realize chips. In this research structural VHDL and Verilog are chosen to be the output of the synthesizer. The optimization problems for area (cost), power, and speed will be described and discussed in Chapter 6. In order to reduce the design time to implement micropipelines with high performance and without the problems mentioned in Chapter 2, we use a straightforward approach for synthesizing a standard VHDL description of an asynchronous circuit from a behavioural VHDL description. This approach is introduced in this chapter and a design flow is also presented.

3.1 Synthesis procedure

The synthesis procedure begins by partitioning descriptions into several pipeline stages, the number depending on the concurrency and other properties of the description. Then we produce the circuits for each stage. Appropriate control circuits are automatically added into the stages. Finally, we produce the interconnections between the stages.

Figure 3.2 shows the structure of the synthesizer. The input language is a **top level behavioural VHDL model**. The VHDL statements are partitioned into several blocks. Then the statements within each block are converted into circuits which are denoted by objects of C++ classes. Only assignments will be translated to those internal circuit models as shown in Figure 3.6, which denote that the latches of the left hand side variables receive data from the latches of the right

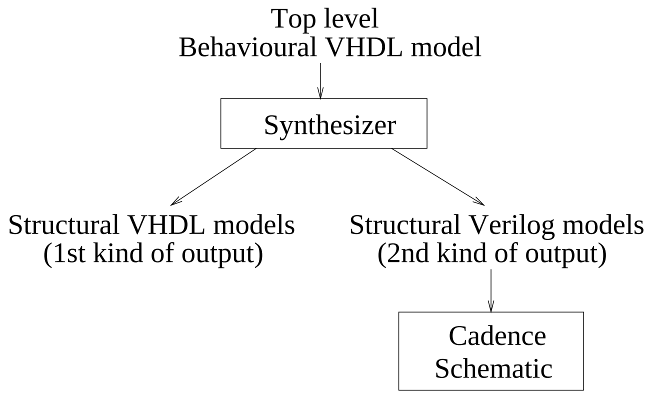


Figure 3.2: Structure of the synthesizer

hand side variables. If there are logical functions or mathematical computations within the right hand side expression, some Device objects will be put between the latches. For normal assignments the control circuit shown in Figure 2.10 will be added, which is also denoted by the internal circuit models. For WHILE and IF statements containing req_in, req_out, ack_in and ack_out, the statements within these WHILE and IF statements will be re-partitioned into several stages again. The corresponding control circuits will be added to connect these stages after converting them into internal circuit models. More detail about the synthesis steps are presented in the subsequent subsection. The synthesizer has two kinds of output. The first kind of output is **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 outputs may be simulated to check the correctness of the synthesized circuits. The second kind of output of the synthesizer is **structural Verilog models** of the components. These outputs may be imported into a Design Framework II database using Verilog In [Cade]. This is convenient for subsequent design work in the Cadence environment.

```
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 3.3: A top level behavioural VHDL model

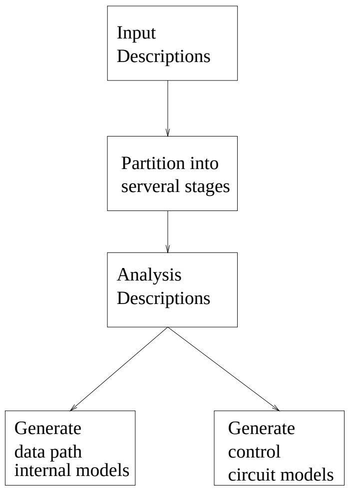


Figure 3.4: Synthesis steps

Synthesis steps

The synthesis steps are summarized as follows:

1. Create latches which are denoted by the internal circuit models as shown in Figure 3.7 for each variable. The example shown in Figure 3.3 has three variables. Two 2-bit active-low transparent latches (ltlatch2) are created for variables QA and QB. One 3-bit active-low transparent latch (ltlatch3) is created for variable QC.

2. Count the occurrence number of each left hand side variable. This is used to decide whether a selector is required at the input of the latch that holds the variable. If the occurrence number is greater than one a selector is required. For the example of Figure 3.3 the occurrence numbers of the left hand side variables QA, QB and QC as well as external signals SUM and CARRY are one. No selector is required here.

3. Partition the statements into several stages. The partition rules are described in Section 3.3. For example, the statements of the behavioural VHDL model shown in Figure 3.3 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.

4. 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 assignments. The synthesis information includes $info\_assign$, $info\_for$, $info\_if$, $info\_while$ and $info\_contain$. They work as follows:

   - $info\_assign = info\_assign + 1$ if an assignment or a balanced IF statement is met.

   - $info\_while = info\_while + 1$ if a WHILE statement is met.

- $info\_for = info\_for + 1$ if a FOR statement is met.
- $info\_if = info\_if + 1$ if an unbalanced IF statement is met.
- $info\_contain = info\_contain + 1$ if **REQ_IN**, **REQ_OUT**, **ACK_IN** or **ACK_OUT** is met within a WHILE, an IF or a FOR statement. **REQ_IN**, **REQ_OUT**, **ACK_IN** and **ACK_OUT** are procedures which are used for the behavioural simulations to produce the hand-shaking signals with the test bench program.

The two stages which are partitioned from the statements shown in Figure 3.3 contain only assignments. The $info\_assign$ for the first stage is 2 and the $info\_assign$ for the second stage is 1. The statements $SUM <= QC(1 DOWNTO 0)$; and $CARRY <= QC(2)$; have been moved to the top level (top_stg stage) before the partitioning starts. As shown in Figure 3.5 the top level (top_stg stage) connects to the first stage ($stg1$), the first stage connects to the second stage ($stg2$) and the second stage connects to the top level (top_stg stage).

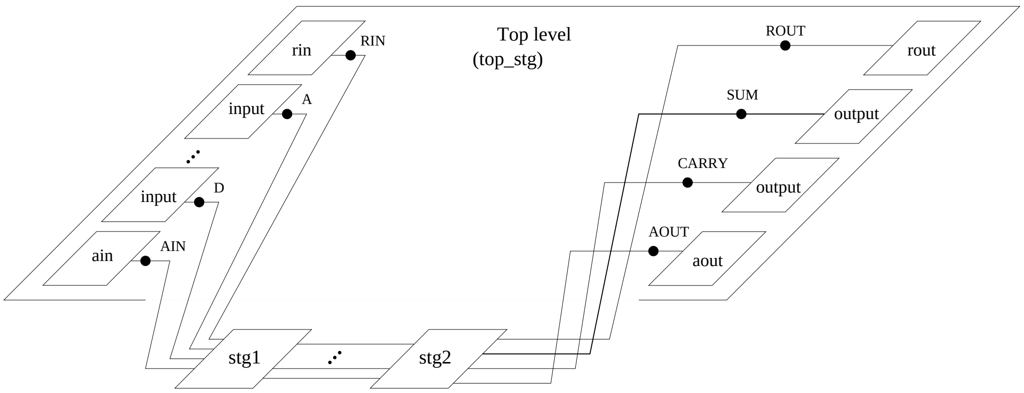


Figure 3.5: The top level(top_stg) connects to the partitioned stages

5. Convert all statements into internal circuit models which are denoted by objects of Device and Terminal classes, as shown in Figure 3.6, for each stage. These classes have been used to implement a micropipeline simulator [Tan92, Tan94] based on Petri net modelling techniques [Pete81]. The

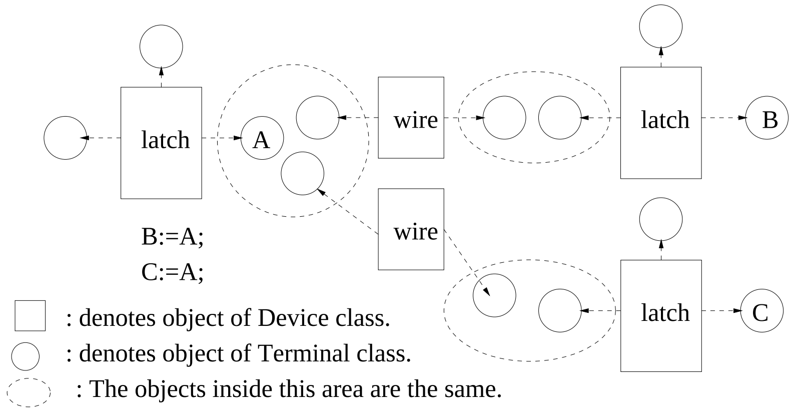


Figure 3.6: The denotation of components using objects of Device and Terminal classes

conversion rules are mentioned in Section 3.3. The internal circuit models of the data path for the above two stages are shown in Figure 3.7. The rectangles denote the objects of Device class. The circles denote the objects of Terminal class.

6. 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 2.10. The rules for producing control circuits are described in Section 3.3.

7. 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 predecessor 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.

8. 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 3.3. The corresponding structural VHDL model is shown in Figure 3.8.

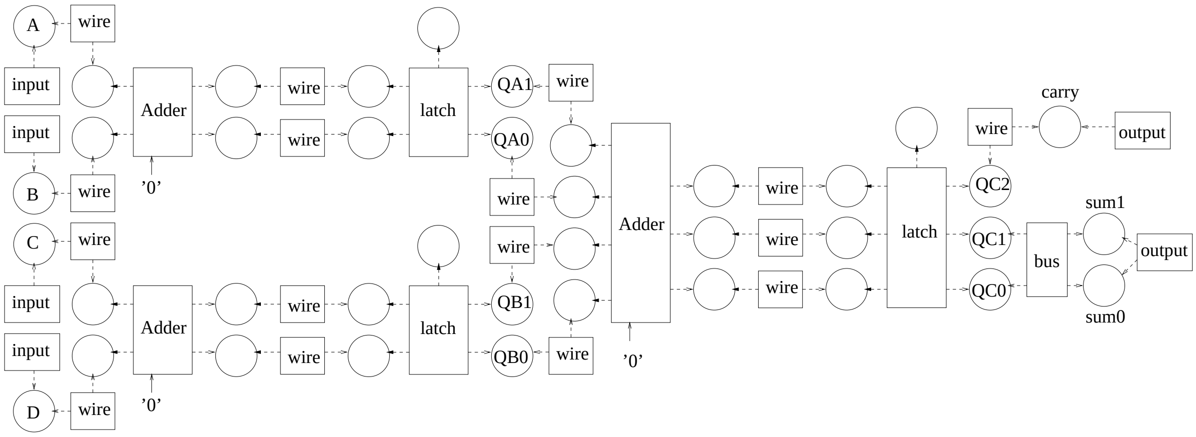


Figure 3.7: The internal circuit model of the data path for the VHDL model of Figure 3.3

3.2 VHDL

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

- VHDL is an IEEE standard 1076-1987 [Ieee87]. It is an important industrial design tool for digital systems.
- The event-driven logic modules shown in Figures 2.4 2.9 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 [Rush95] and VHDL synthesis examples are also given.

```
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 3.8: The structural VHDL model for the behavioural VHDL model shown in Figure 3.3

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 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 (see Appendix C) 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 Figures 2.4 2.9. The **WAIT FOR** statements are only used for the behavioural simulations. They will be ignored by the synthesizer.

The library contains standard gates and user-defined gates, such as the event-driven logic modules shown in Figures 2.4 2.9. 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.

VHDL models

The behavioural VHDL models of some components which are used to test the output of the synthesizer, such as c2, toggle2, not1, xor2, or2, comparator1, comparator4, subtractor4, mux2x4, mux2x1, mux2x6, ltlatch1, ltlatch4, ltlatch6, and ltlatch9 are shown in Appendix A.

3.3 Synthesis rules

The rules for partition, conversion, and the production of control circuits are introduced in the following subsections.

3.3.1 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 3.7, the registers for the variables QA, QB, and QC declared in Figure 3.3 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 3.7 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 3.3 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 which input data and output data on the same variable it will be put into a stage on it own. The reason is that a special control circuit is required for such shift operations. This control circuit is not able to handle other functional statements at the same time. For example, the control circuits shown in Figures 3.22 and 3.24 can not be used for normal assignments. Complement latches are required to store data during the shift operation. The **lt** signal of Figure 3.22 will go Hi for $a + 1$ times, where a is the value denoted by $A_7A_6A_5A_4A_3A_2A_1A_0$. The data $B_7B_6B_5B_4B_3B_2B_1B_0$ is shifted left a bits. The **lt** signal of Figure 3.24 will go Hi twice. The data in the latches fra_h3, fra_h2, fra_h1 and fra_h0 is shifted left one bit. The circuits shown in Figure 3.23 read the data in the latches fra_h2, fra_h1 and fra_h0 and assign them to the latches fra_r3, fra_r2 and fra_r1. Because the input data and output data are not on the same variable the control circuit for normal assignments is applied.

- 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.

3.3.2 Conversion rules

After partitioning the statements into several blocks the synthesizer starts one of the main missions presented in this chapter, the conversion of statements into internal circuit models. There are also some 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. They are shown as below.

```
C := ‘0’;
C := "00000";
C := B;
C := A ( 0 );
C := A ( 0 TO 5 );
C := A - B;
C := A - B ( 0 );
C := A ( 0 ) - B;
C := A ( 0 ) - B ( 5 );
C := A ( 0 TO 5 ) - B;
C := B - A ( 0 TO 5 );
 C := A ( 0 TO 5 ) - B ( 0 TO 5 );
C := A ( 0 ) or B ( 0 );
C := A and B;
A ( 0 ) := C;
A ( 0 ) := B ( 0 );
C ( 0 ) := A ( 0 ) - B ( 5 );
C ( 0 ) := A ( 0 ) - B;
C ( 0 ) := B - A ( 0 );
A ( 0 TO 5 ) := C;
A ( 0 TO 5 ) := B ( 0 TO 5 );
C ( 0 TO 5 ) := A ( 0 TO 5 ) - B;
C ( 0 TO 5 ) := B - A ( 0 TO 5 );
C ( 0 TO 5 ) := A ( 0 TO 5 ) - B ( 0 TO 5 );
C := A - B - D;
C := A ( 0 ) - B ( 5 ) - D ( 3 );
C := A ( 0 TO 5 ) - B ( 0 TO 5 ) - D ( 0 TO 5 );
                .
                .
                .
```

To convert such a statement 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. Figure 3.9 shows the implementation of the statement **"if A='0' then C:=B; else C:=D; end if;"**. The terminals of the internal circuit models of each component are denoted by the objects of Terminal class. Because Figure 3.9 is used to show how a multiplexer implements an IF statement with balanced THEN and ELSE branches the objects of Terminal class are only denoted by black dots. More clear pictures about objects of Terminal class can be found in Figures 3.6 and 3.7.

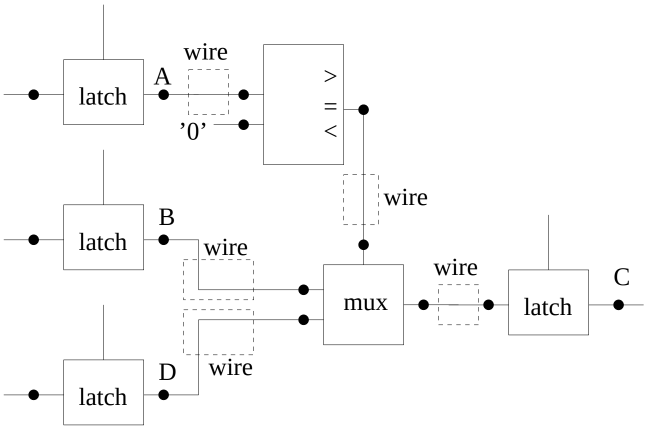


Figure 3.9: The implementation of the statement "if A='0' then C:=B; else C:=D; end if;"

To convert multi-level IF statements into their internal circuit models a **FILO** method is used to calculate the position of the multiplexer. The **FILO** method is described as follows:

1. Initially assign a variable **POSITION = 0** to denote the level of assign statements inside an "if" statement and then
   - Push **POSITION** onto the stack when we meet an "if", and then **POSITION = POSITION * 2 + 1**;
   - **POSITION++** when we meet an "else";
   - Pop **POSITION** from the stack when we meet an "end if;".

```
ENTITY ex2 IS
  PORT ( RIN, AOUT : IN  MVL;
         AIN, ROUT : OUT  MVL;
         A, B : IN MVL;
         C, D : IN MVL;
         W1, W2 : OUT MVL;
         RESET : IN MVL );
END ex2;

architecture BEHAVIORAL of ex2 is
begin
 U1: process
    variable Q1, Q2, M, N : MVL;
     begin
           if A = '1' then
               M := A;
               if B = '1' then
                   N := B; Q1 := A and B; Q2 := C or D;
               else
                   N := A; Q1 := D; Q2 := C;
               end if;
           else
               M := C; N := D; Q1 := B; Q2 := A;
           end if;
           W1 <= Q1 and Q2;  W2 <= M or N;
     end process U1;
end BEHAVIORAL;
```

Figure 3.10: A multi-level IF statement example

For example, when working on the example shown in Figure 3.10 **POSITION** is set to **0** at the beginning. When the first "if" is met the original value **0** of **POSITION** is pushed into the stack and **POSITION** becomes **1**. Therefore, the **POSITION** of the assignment **M := A;** is **1**. Then the second "if" is met. The value **1** of **POSITION** is pushed into the stack again and **POSITION** changes to **3**. The **POSITION**s of assignments **N := B; Q1 := A and B; Q2 := C or D;** are **3**. When the first "else" is met **POSITION** adds **1** and becomes **4**. The **POSITION**s of assignments **N := A; Q1 := D; Q2 := C;** are **4**. When the first "end if;" is met **POSITION** is restored a value popped from the stack. Here **POSITION** becomes **1**. When the second "else" is met **POSITION** adds **1** and becomes **2**. The **POSITION**s of assignments **M := C; N := D; Q1 := B; Q2 := A;** are **2**. When the final "end if;" is met **POSITION** is restored a value popped from the stack. **POSITION** is **0**. The process is finished. The related information can be found in the if_assign table shown in Figure 3.11.

Variable Name	Left side Expression	Level Position	Left side Expression	Logic Expression
M	M	1	A	A = ’1’
Q1	Q1	3	A AND B	B = ’1’
Q2	Q2	3	C OR D	B = ’1’
N	N	3	B	B = ’1’
Q1	Q1	4	D	B = ’1’
Q2	Q2	4	C	B = ’1’
N	N	4	A	B = ’1’
M	M	2	C	A = ’1’
N	N	2	D	A = ’1’
Q1	Q1	2	B	A = ’1’
Q2	Q2	2	A	A = ’1’

Figure 3.11: The if_assign table of the example shown in Figure 2.5

2. Create an IF_ASSIGN_TABLE in which the content is as follows:
   - Variable name of the left side of “:=” or “<=”,
   - Left side expression of “:=” or “<=”,
   - Right side expression of “:=” or “<=”,
   - Level position,
   - Whether the right side expression and the logic expression contain the variable at the left side within the same “if” statement,
   - Logic expression.

   The if_assign table of the example of Figure 3.10 is shown in Figure 3.11. Before implementing the circuits the if_assign table will be sorted first. All rows in the table with the same left expression are converted into circuits together. The sorted if_assign table is shown in Figure 3.12. The value of **POSITION** is used for getting the corresponding multiplexer and the pin number of the multiplexers.
   A device number can be obtained using

   device no. is position - 1 if position is an even number
   device no. is position if position is an odd number

   A pin number can be obtained using

   $$\textbf{pin} = \textbf{position} \ \% \ \textbf{2}$$

   Figure 3.13 shows the corresponding circuits of the if_assign table shown in Figure 3.12.

3. Create a REGISTER_TABLE in which the content is as follows:
   - Variable name,

Variable Name	Left side Expression	Level Position	Left side Expression	Logic Expression
M	M	1	A	A = '1'
M	M	2	C	A = '1'
N	N	2	D	A = '1'
N	N	3	B	B = '1'
N	N	4	A	B = '1'
Q1	Q1	2	B	A = '1'
Q1	Q1	3	A AND B	B = '1'
Q1	Q1	4	D	B = '1'
Q2	Q2	2	A	A = '1'
Q2	Q2	3	C OR D	B = '1'
Q2	Q2	4	C	B = '1'

Figure 3.12: The if_assign table of Figure 3.11 after sorting

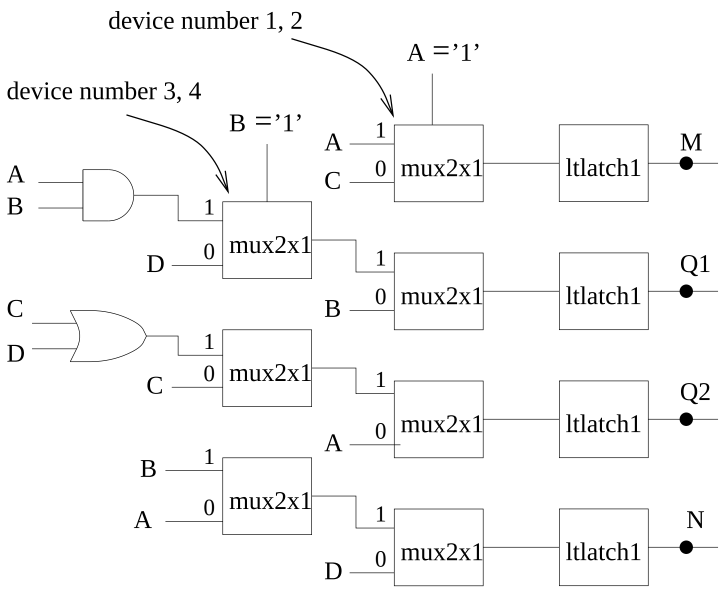


Figure 3.13: The corresponding circuits of the if_assign table shown in Figure 3.12

Variable Name	Register		Register'
A	0x137e70	input1	0x0
B	0x138380	input1	0x0
C	0x138488	input1	0x0
D	0x138590	input1	0x0
W1	0x138aa0	output1	0x0
W2	0x138ba8	output1	0x0
Q1	0x1393c0	ltlatch1	0x0
Q2	0x139568	ltlatch1	0x0
M	0x139710	ltlatch1	0x0
N	0x1398b8	ltlatch1	0x0

Figure 3.14: The register table of the example shown in Figure 3.10

Variable Name	Left side Expression	Occurence number of Left side Expression	Current occurence no. of Left side Expression	Selector
M	M	1	0	0x0
Q1	Q1	1	0	0x0
Q2	Q2	1	0	0x0
N	N	1	0	0x0
W1	W1	1	0	0x0
W2	W2	1	0	0x0

Figure 3.15: The assign table of the example shown in Figure 3.10

   - Register,
   - Register'.

   Figure 3.14 shows the register table of the example shown in Figure 3.10. The hexadecimal numbers inside each column of the register table are the addresses of internal latch models. If the right hand side expression field contains the left hand side variables complement registers are required. They will be shown in the columns labelled **Register'**. If a low-active transparent latch is used to be the register a high-active transparent latch will be used to be the register'. **0x0** denotes that the register' is not used.

4. Create a ASSIGN_TABLE in which the content is as follows:
   - Variable name of the left side of ":=" or "<=",
   - Left side expression of ":=" or "<=",
   - Number of the left side expression of ":=" or "<=" within the whole description,
   - Current number of the left side expression of ":=" or "<=" within an "if" or "while" or "for" statement,
   - Selector.

5. Count the present number of the left side expression of ":=" or "<=" within the whole description. If the present number is greater than one a selector will be applied. Figure 3.15 shows the assign table of the example shown in Figure 3.10. The occurrence number of the left hand side variable records the number of the sources which are generated the results of the left hand side variable. If the occurrence number of the left hand side variable is greater than one a selector will

be used. Then the current occurrence number of the left hand side variable records the present source count. It is used to decide the pin number of the selector. **0x0** in the column labelled Selector denotes that the selector is not used.

6. Generate actual circuits using the above information. The generated circuit is shown in Figure 3.13.

- 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.

3.3.3 Rules for producing control circuits

- As shown in Figure 3.16 we can use a two-input C gate to AND the two events and a C gate with more than two inputs to AND more than two events. If some events must happen before an other event, those events will be connected to an inverter and then input to the C gate.

- Figure 3.18 shows the basic control circuit for the event register shown in Figure 3.17. Figures 3.19 and 3.20 show the control circuits of a WHILE statement and a nested WHILE statement respectively.

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

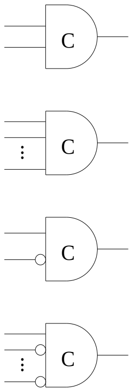


Figure 3.16: A C gate to AND events

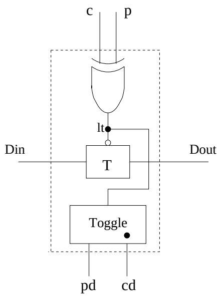


Figure 3.17: An event register

Figure 2.10. Otherwise, a control circuit similar to that used in a previous 4-bit asynchronous multiplier circuit [Day91] will be used. Figures 3.21 shows the control circuit which is used for a WHILE statement. Figures 3.22, 3.23 and 3.24 show the control circuits which are used for SHIFT operations. Figures 3.25 shows the control circuit which is used for a FOR statement.

- 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.
- The procedure to generate the interconnections of control circuits between stages. For example,

  STG1's predecessor stage is TOP_STG and
  STG1's successor stages are STG2, STG3;

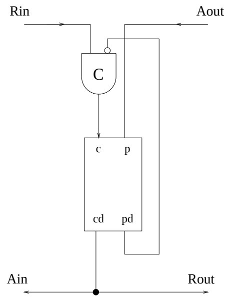


Figure 3.18: The basic control circuit for an event register

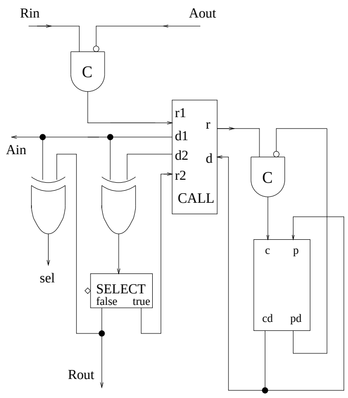


Figure 3.19: The control circuit for a WHILE loop

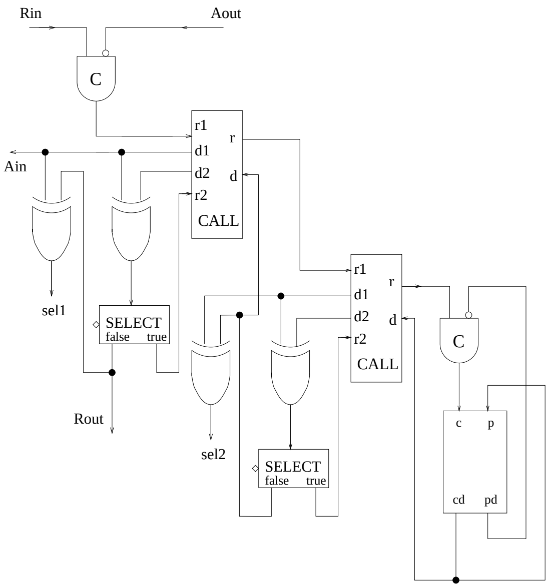


Figure 3.20: The control circuit for a nested WHILE loop

```
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;
```

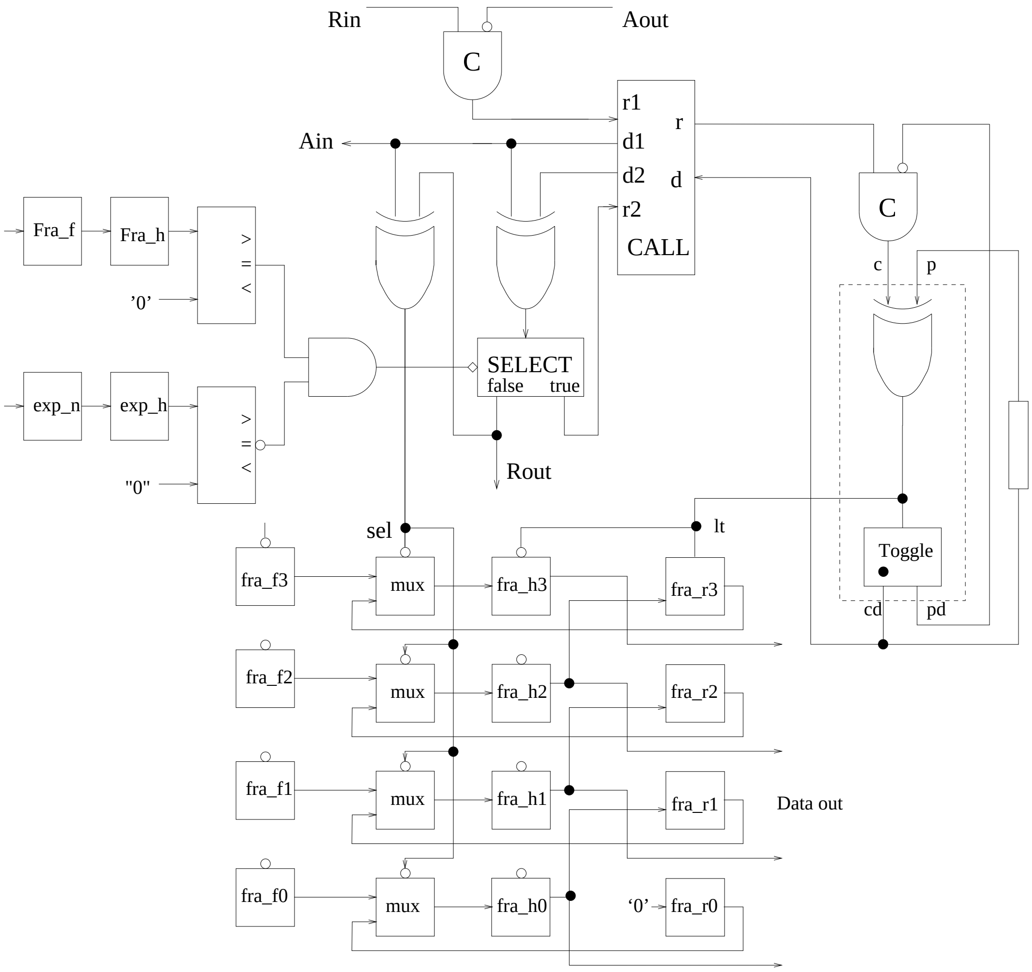


Figure 3.21: The control circuit for a WHILE loop

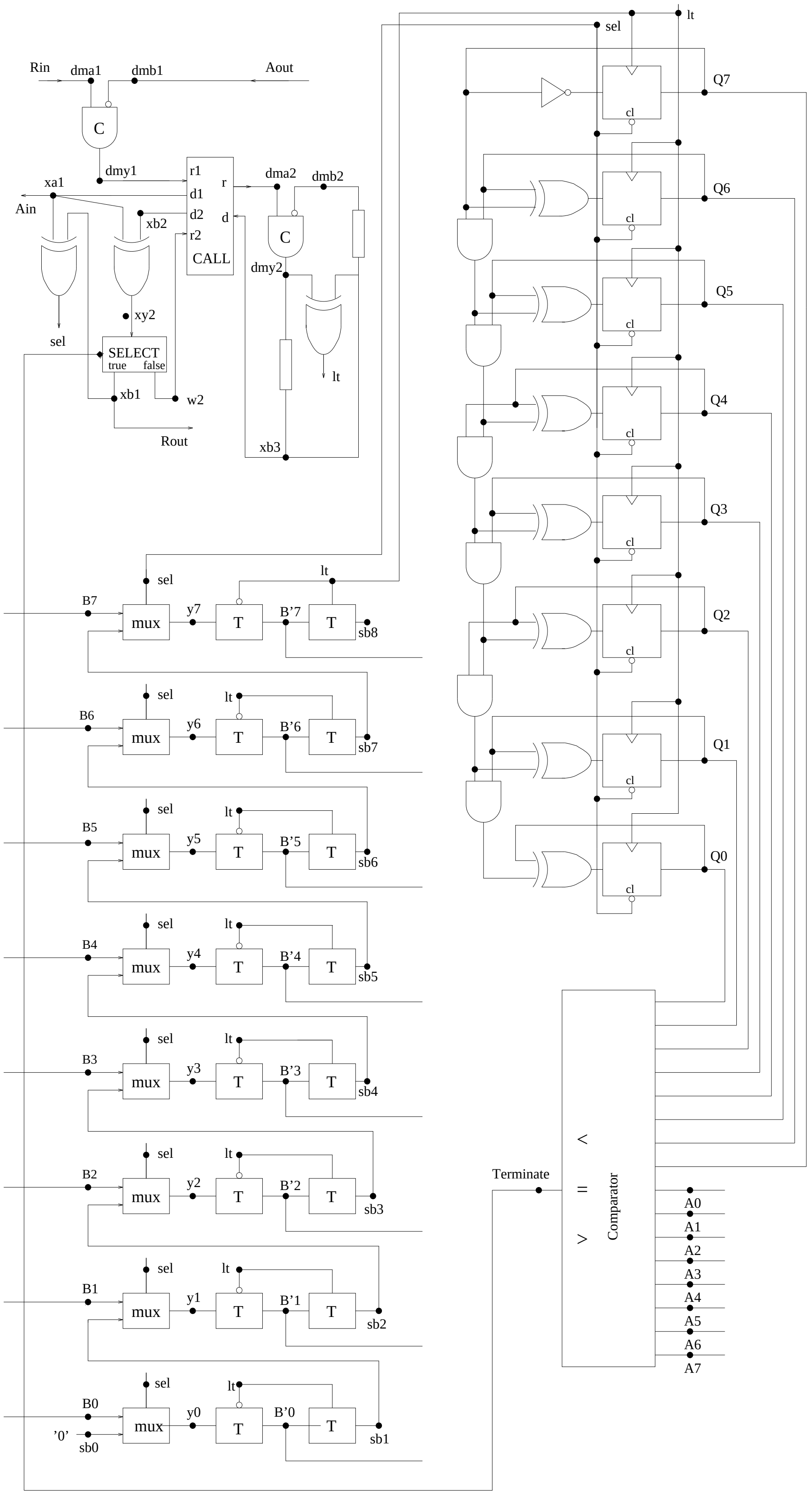


Figure 3.22: A shift right stage

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

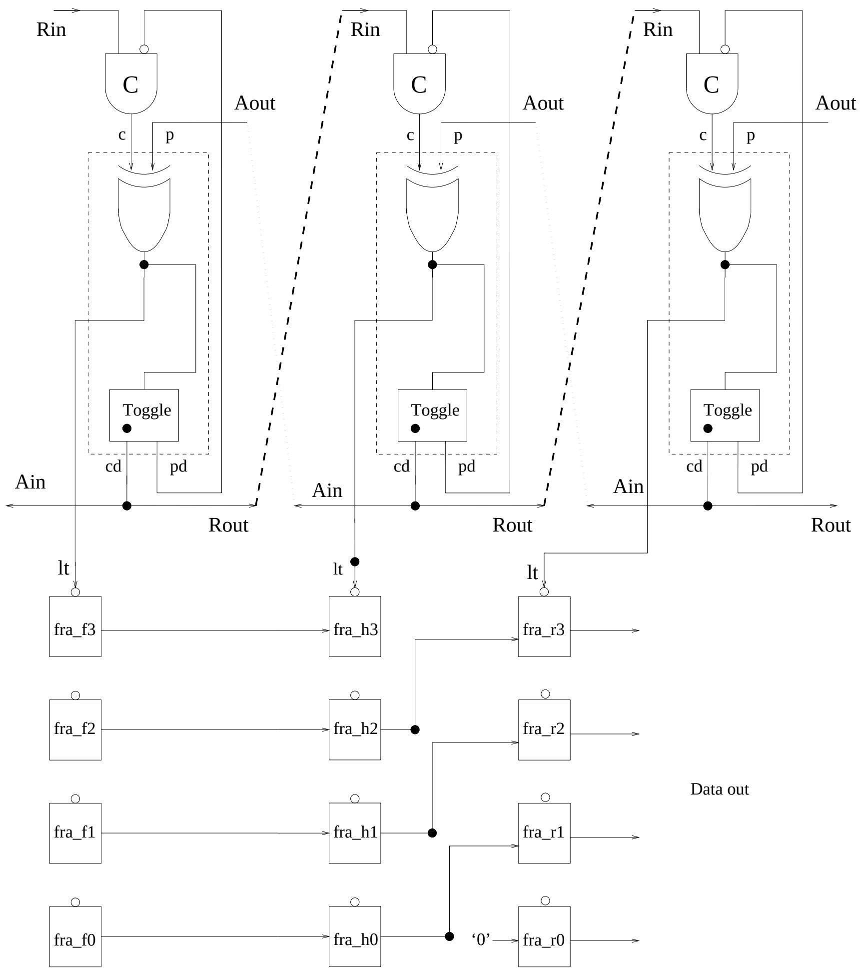


Figure 3.23: The control circuit for a SHIFT operation

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

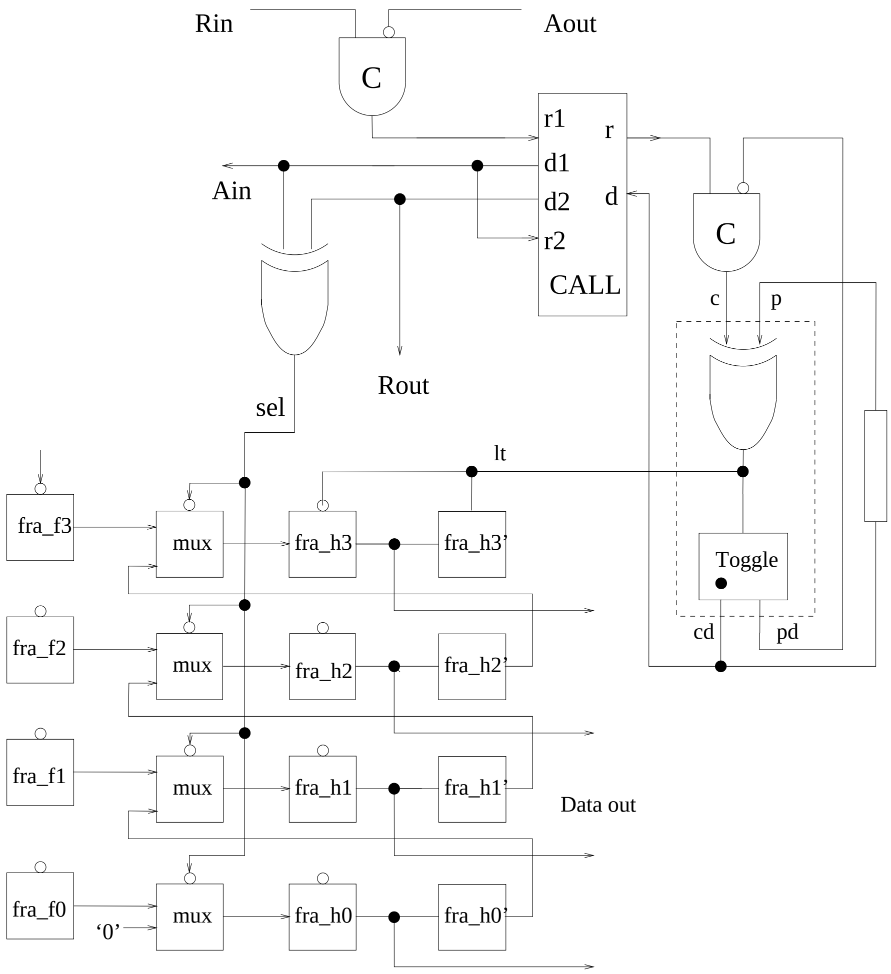


Figure 3.24: The control circuit for a SHIFT operation

```
fra_d1(0 to 54) := fra_dd(0 to 54);
fra_d1(55) := fra_dd(55) or fra_dd(56); --get sticky bit
fra_d1(56) := fra_dd(56);

for G in 1 to TO_INTEGER( exp_diff ) loop
    fra_d := sh_right( fra_d1, 1 );       -- shift
    fra_d1(0 to 54) := fra_d(0 to 54);
    fra_d1(55) := fra_d(55) or fra_d(56); --get sticky bit
end loop;
```

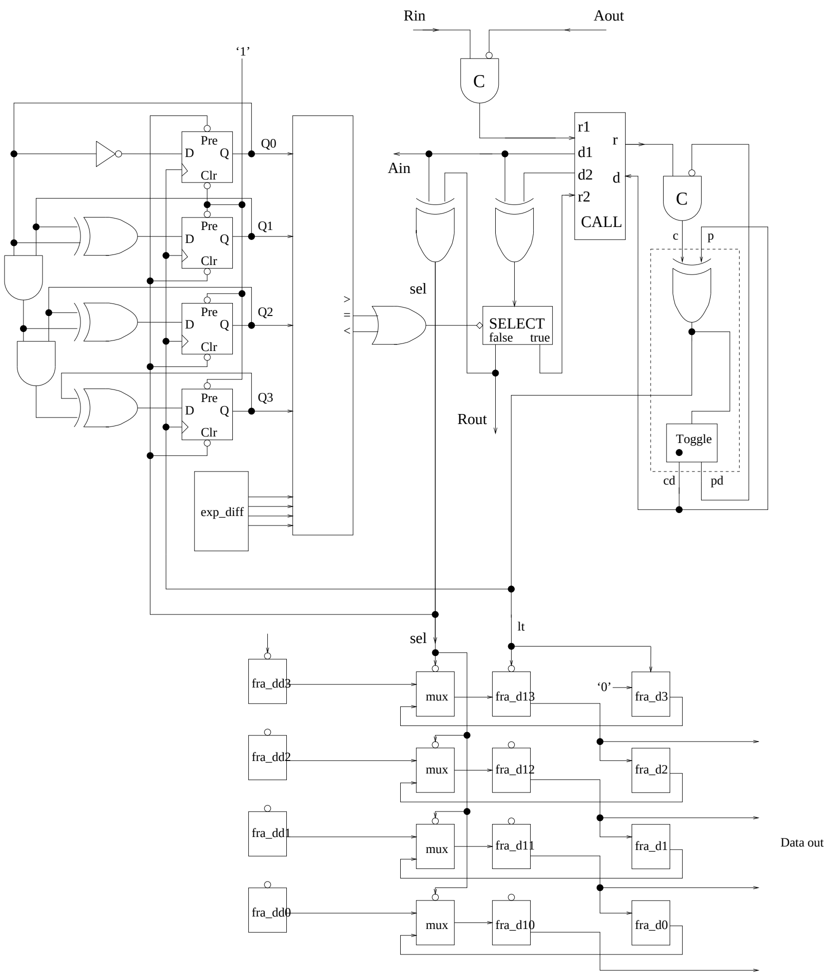


Figure 3.25: The control circuit for a FOR loop

STG2's predecessor stage is STG1 and
STG2's successor stage is STG3;

STG3's predecessor stages are STG1, STG2 and
STG3's successor stage is TOP_STG;

The correct procedures are as follows:

For each stage:

1. Check the number of the predecessor stages of the current stage.
2. If the number = 1 then connect the ROUT of the predecessor stage to the RIN of the current stage.
3. If the number > 1 then using a C gate to connect the ROUTs of the predecessor stages to the RIN of the current stage.
4. Check the number of the successor stages of the current stage.
5. If the number = 1 then connect the AIN of the successor stage to the AOUT of the current stage.
6. If the number > 1 then using a C gate to connect the AINs of the successor stages to the AOUT of the current stage.

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

Steps for producing structural VHDL output files

After the internal circuit models are constructed the synthesizer produces the structural VHDL output files for each stage and for the whole system. The procedure is described as follows:

- Check the input, output and control signals to produce the entity VHDL file for each stage. The component port declaration for this stage may be generated and written to the system VHDL file at the same time.
- Follow the objects of Device class within each stage to produce the component port declarations for each device. The configuration specifications

which are used to close the gap between entities and their component declarations are also produced at this time.

- Follow the wires and buses to produce the internal signals within each stage and between stages.

- Follow the connections of wires and buses to created the component map for each device within each stage and the stage component map between stage.

- Produce the END statement for each structural VHDL file.

```
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 3.26: A test example

3.3.4 Experimental results

All the rules and steps mentioned in previous subsections were implemented. A test example of the behavioural description shown in Figure 3.26 was used to test the synthesizer. This example is a small floating point Adder with a 4-bit exponent and a 5-bit 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 respectively. 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. Part of the internal circuit model of this example is shown in Figure 3.27; it is created during the synthesis. 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 3.28. The three stages, stg1, stg2, and stg3, were treated as

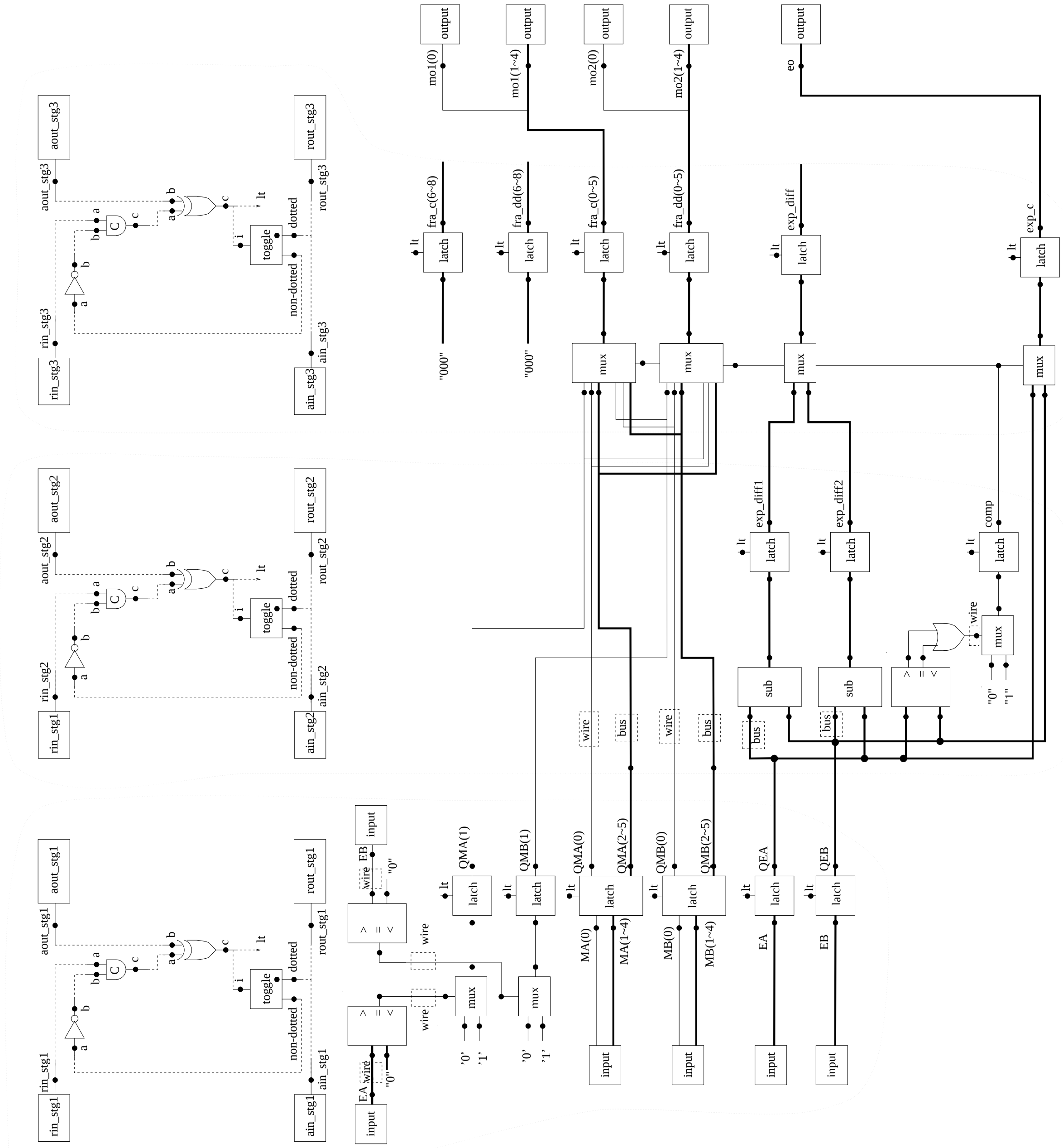


Figure 3.27: Part of the internal circuit model of the example shown in Figure 3.26

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 3.28: The numbers of components and internal signals inside each synthesized VHDL file

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.

3.4 Design flow

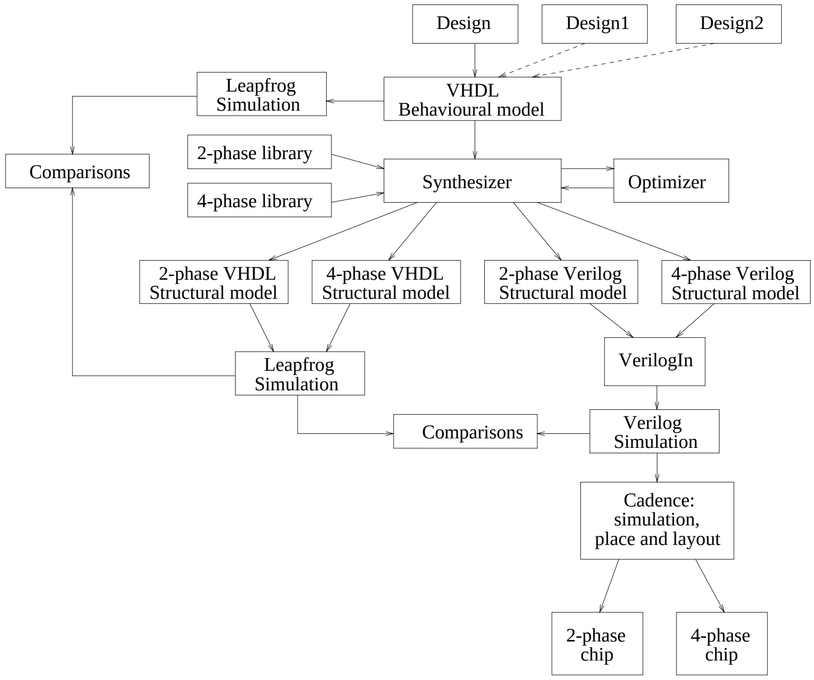


Figure 3.29: Design flow

Figure 3.29 shows a potential design flow. The designs are described in behavioural VHDL descriptions. This VHDL descriptions may be simulated using a VHDL simulator (Leapfrog). Then the descriptions can be synthesized into two-phase and four-phase VHDL structural models and Verilog structural models respectively. The structural VHDL models may also be simulated using a VHDL simulator. The simulation results may be compared with the previous behavioural simulation results to verify the implementation against its specification. For example, Figure 3.30 shows the data collected from the simulation of the 2-phase behavioural VHDL models of the Stump processor shown in Appendix G.2 and Figure 3.31 shows the data collected from the VHDL simulation of the synthesized 2-phase Stump processor of Figure 5.29. The values within double or single quotes represent the logical values of the variables or signals at specified times. There is no relationship between the times in the two figures. However, they represent the order of the data changes within each figure individually. To check that the synthesized circuit meets the specification of the behavioural model comparison steps are undertaken as follows:

1. Check whether the output data values of both figures are equal at the first rising edge of **ROUT** (i.e. ROUT changes to logical **1** from logical **0**) as well as at subsequent positive transitions on **ROUT**.

2. Check whether the corresponding acknowledges are present; i.e. **AOUT** becomes the same logical value shortly after the transitions on **ROUT**.

3. Check whether there are data changes between the transitions on **ROUT** and **AOUT**. "NO" is the expected answer. If the answer is "YES" the synthesized circuits may contain errors.

The Verilog structural models can be imported into the Design Framework II database (Cadence) using VerilogIn. Then a Verilog simulation can be undertaken. The Verilog simulation results may be compared with the VHDL simulation results to check the correctness before going down to layout work. Placement and routing can be done in the Cadence environment. The Hspice netlist of the layout can be generated in the system. TimeMill [Epic95] and PowerMill [Epic96] can be used to do post-layout simulation, timing analysis and power estimation.

Figure 3.32 shows the interface between the synthesizer and other tools, LARD [Ende98, Ende98a], Yellow [Barr96, Barr97], and Balsa [Bard96, Bard97, Bard98]. The design may be also described and simulated in LARD. The simulation results may be compared with the VHDL results to check the correctness of the input descriptions and the output implementation.

```
ROUT  3 NS '0'
ADDRESS  121 NS "0000000000000000"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

ROUT  126 NS '1'
ADDRESS  121 NS "0000000000000000"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

ROUT  185 NS '0'
ADDRESS  180 NS "0000000000000001"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

ROUT  244 NS '1'
ADDRESS 239 NS "0000000000000010"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

ROUT  303 NS '0'
ADDRESS 298 NS "0000000000000011"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

ROUT  362 NS '1'
ADDRESS 357 NS "0000000000000100"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

ROUT  421 NS '0'
ADDRESS 416 NS "0000000000000101"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

ROUT  480 NS '1'
ADDRESS 475 NS "0000000000100001"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

ROUT  539 NS '0'
ADDRESS 534 NS "0000000000000110"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

ROUT  598 NS '1'
ADDRESS 593 NS "0000000000100010"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'
        ⋮
```

```
AOUT  1 NS '0'
ADDRESS  121 NS "0000000000000000"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

AOUT 147 NS '1'
ADDRESS 121 NS "0000000000000000"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

AOUT  206 NS '0'
ADDRESS 180 NS "0000000000000001"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

AOUT  265 NS '1'
ADDRESS 239 NS "0000000000000010"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

AOUT  324 NS '0'
ADDRESS  298 NS "0000000000000011"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

AOUT  383 NS '1'
ADDRESS 357 NS "0000000000000100"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

AOUT  442 NS '0'
ADDRESS 416 NS "0000000000000101"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

AOUT  501 NS '1'
ADDRESS 475 NS "0000000000100001"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

AOUT  560 NS '0'
ADDRESS  534 NS "0000000000000110"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'

AOUT  619 NS '1'
ADDRESS 593 NS "0000000000100010"
DATAOUT 121 NS "0000000000000000"
WR 121 NS '1'
OE 121 NS '0'
EN 121 NS '0'
        ⋮
```

Figure 3.30: Part of the simulation result of the 2-phase behavioural VHDL models of the Stump processor shown in Appendix G.2

```
ROUT 0 NS '0'
ADDRESS 10 NS "0000000000000000"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

ROUT 200 NS '1'
ADDRESS 10 NS "0000000000000000"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

ROUT 521 NS '0'
ADDRESS 451 NS "0000000000000001"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

ROUT 842 NS '1'
ADDRESS 772 NS "0000000000000010"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

ROUT 1163 NS '0'
ADDRESS 1093 NS "0000000000000011"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

ROUT 1484 NS '1'
ADDRESS 1414 NS "0000000000000100"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

ROUT 1805 NS '0'
ADDRESS 1735 NS "0000000000000101"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

ROUT 2126 NS '1'
ADDRESS 2056 NS "0000000000100001"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

ROUT 2447 NS '0'
ADDRESS 2207 NS "0000000000000110"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

ROUT 2768 NS '1'
ADDRESS 2698 NS "0000000000100010"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'
        ⋮
```

```
AOUT 1 NS '0'
ADDRESS 10 NS "0000000000000000"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

AOUT 221 NS '1'
ADDRESS 10 NS "0000000000000000"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

AOUT 542 NS '0'
ADDRESS 451 NS "0000000000000001"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

AOUT 863 NS '1'
ADDRESS 772 NS "0000000000000010"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

AOUT 1184 NS '0'
ADDRESS 1093 NS "0000000000000011"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

AOUT 1505 NS '1'
ADDRESS 1414 NS "0000000000000100"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

AOUT 1826 NS '0'
ADDRESS 1735 NS "0000000000000101"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

AOUT 2147 NS '1'
ADDRESS 2056 NS "0000000000100001"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

AOUT 2468 NS '0'
ADDRESS 2207 NS "0000000000000110"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'

AOUT 2789 NS '1'
ADDRESS 2698 NS "0000000000100010"
DATAOUT 10 NS "0000000000000000"
WR 10 NS '1'
OE 10 NS '0'
EN 10 NS '0'
        ⋮
```

Figure 3.31: Part of the VHDL simulation result of the 2-phase Stump processor shown in Figure 5.29

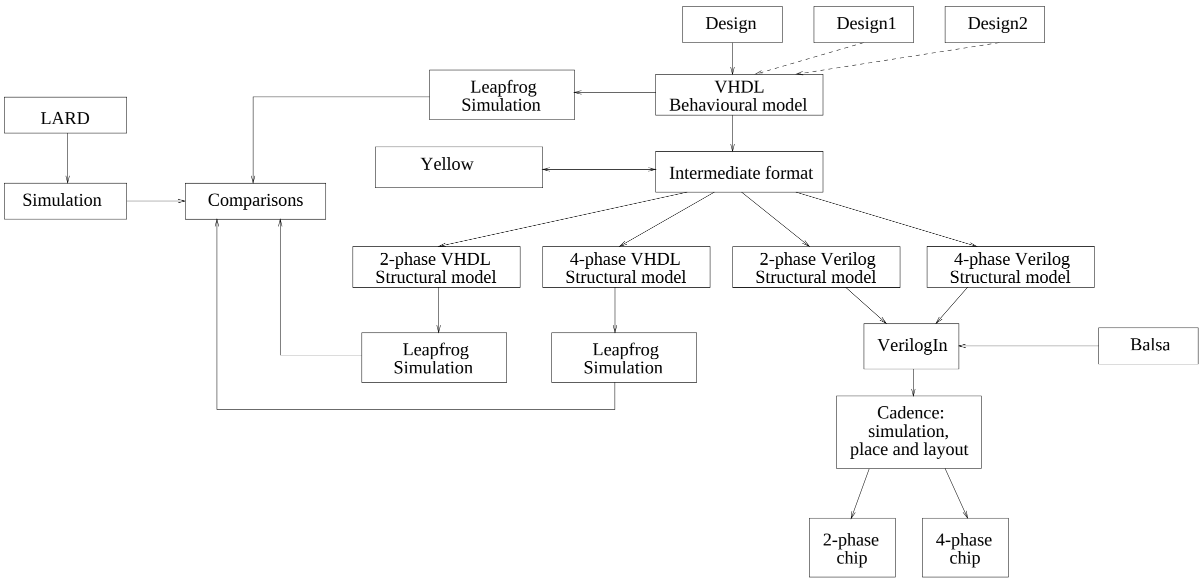


Figure 3.32: Interfacing to other asynchronous description languages

3.5 Summary

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 [Gajs92]. 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 chapter demonstrates that VHDL can be used to describe the behaviour of micropipelined systems. It is also suitable for representing the circuits output by 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.

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.

In order to synthesize correct circuits the input description must be correct first. The simulation mechanism of VHDL is able to assist in the discovery of potential design errors at an early stage.

Chapter 4

A comparison of two-phase and four-phase micropipelines

In chapter 3 the synthesis procedure and synthesis rules were described, a small 2-phase example was given to show how the synthesis works, and a design flow was proposed. In this chapter 4-phase micropipelines will be introduced. Various 4-phase protocols and various latch control circuits are introduced. A comparison of 2- and 4-phase circuits will also be made. More detail describing the design and implementation of 2- and 4-phase circuit synthesizers is presented. Further examples are given to check the correctness of the synthesizers.

4.1 Four-phase micropipelines

Figure 4.1 shows a two-phase stage using an event-controlled storage element which was described by Sutherland [Suth89]. As shown in the figure the positions of three switches enable the input and output to connect directly. This situation is called the transparent state of the storage element. Rising or falling transitions on the wire labelled "lt" change the storage element to hold the data. Rising or falling transitions on the wire labelled "Aout" change the storage element to be transparent again. A "return to inactive state" is not required. This

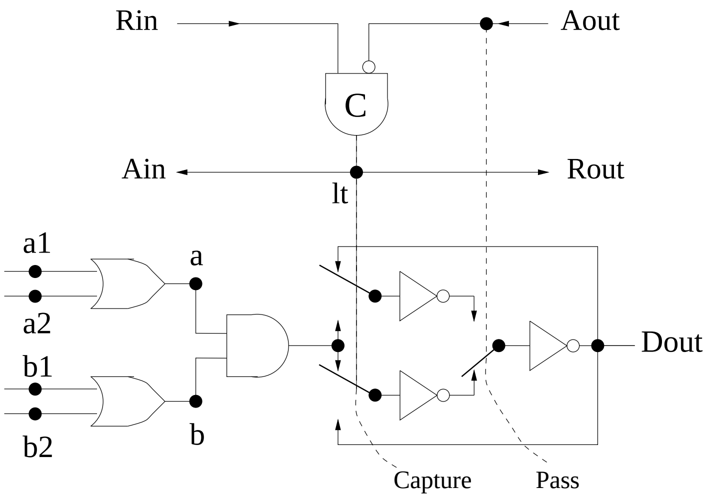


Figure 4.1: A 2-phase stage using event-controlled storage element

shows that two-phase transition signalling should be faster than four-phase transition signalling. However, the storage element is usually implemented using a Toggle, an XOR, and a transparent latch as shown in Figure 2.10. In Paver's comparison between event-controlled storage elements and transparent latches [Pave94], micropipelines using event-controlled storage elements are faster than those using transparent latches as shown in Figure 2.10. However, the size of micropipelines using event-controlled storage elements is greater than those using transparent latches when more than a certain number of data bits are used. Because the transparent latch is level-sensitive, the Toggle and XOR are used to convert 2-phase signalling to 4-phase signalling. The two-phase Toggle is a complex component [Pave94]. To reduce the cost of these conversion circuits a four-phase micropipeline design style is potentially advantageous.

4.1.1 Four-phase bundled data convention

A "four-phase bundled data convention" is a communications system where a four-phase handshaking protocol is used and an arbitrary number of data wires are treated as a bundle together with the request signal wire. In the four-phase

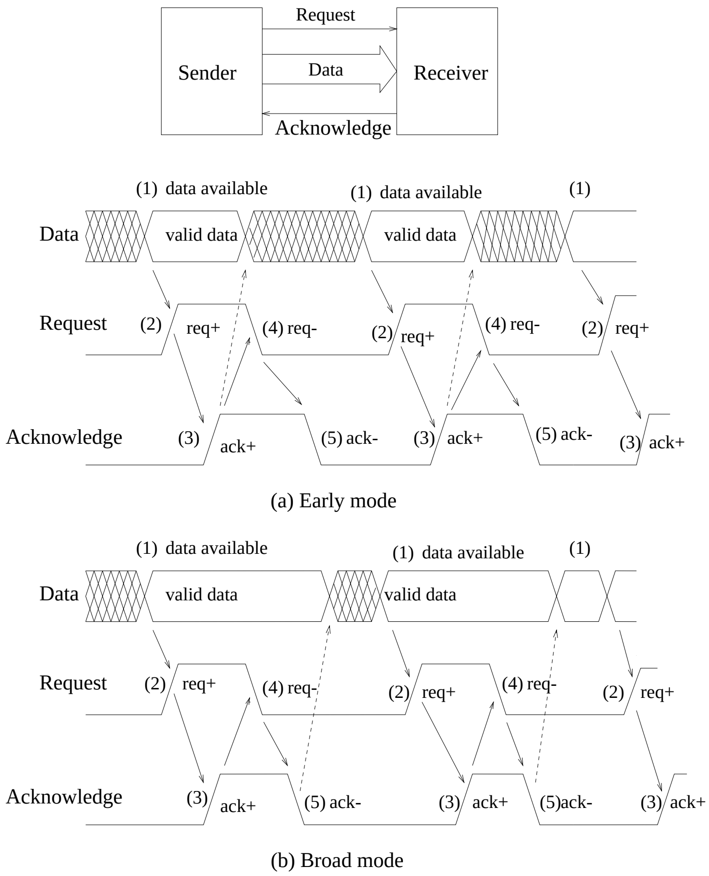


Figure 4.2: Four-phase bundled data convention

handshaking protocol, only rising transitions or only falling transitions of either control wire have the meaning; they represent request events or acknowledge events. In this signalling scheme, the operating cycle is (1) data available (2) change request to active state, (3) change acknowledge to active state, (4) return request to inactive state, and (5) return acknowledge to inactive state. If the active state is logic "1" the the operating cycle is (1) data available (2) **request**+, (3) **acknowledge**+, (4) **request**−, and (5) **acknowledge**−. The data signals can use a traditional data representation which is similar to that used in synchronous circuits, such as the *8-4-2-1 code* ... etc. Figure 4.2 illustrates two kinds of four phase signalling, the 'early' mode and the 'broad' mode [Furb96c]. The 'early' mode (Figure 4.2(a)) uses the rising edge of the Request line to indicate 'data available' and the rising edge of the Acknowledge line to indicate 'data latched'. The falling edges are return to zero actions that carry no meaning. The 'broad' mode (Figure 4.2(b)) uses the rising edge of the Request line to indicate 'data available' and the falling edge of the Acknowledge line to indicate 'data latched'. Another possible protocol is 'late' mode which uses the falling edges as active.

4.1.2 Four-phase event-driven logic modules

In chapter 2 the various event-driven logic modules shown in Figures 2.4, 2.5, 2.6, 2.7, 2.8 and 2.9 for controlling transition signals were introduced. These were devised for composing to two-phase control circuits. Muller C-elements and XOR gates are the same whether they are used in two- or four-phase designs. However, 4-phase Toggle, Select, Call and Arbiter modules are different from their 2-phase counterparts. A Toggle is used to alternately deliver events on its input to one of two outputs. Figure 4.3 shows the behaviour of a Toggle module. In the two-phase protocol each transition denotes an event. Therefore, the odd number transitions on the input of a Toggle will be sent to the dotted output and the even number transitions on the input of a Toggle will be sent to the non-dotted output. This is shown in Figure 4.3(b). In the four-phase protocol each event consists of a rising transition and a falling transition. A rising transition and the following falling transition must be sent to the same output. Therefore, the odd number rising and falling transitions on the input of a Toggle will be sent to the dotted output and the even number rising and falling transitions on the input of a Toggle will be sent to the non-dotted output. This is shown in Figure 4.3(C).

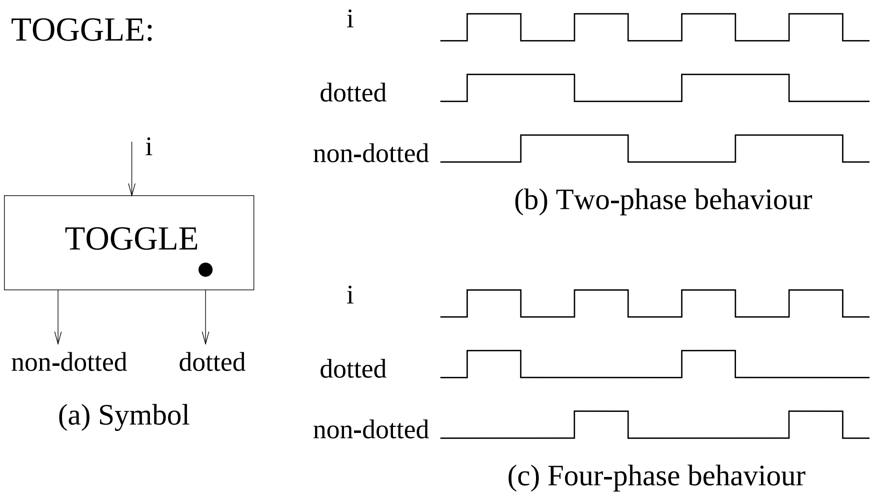


Figure 4.3: Behaviour of the Toggle module

Figure 4.4 shows the behaviour of a Select module. In the two-phase protocol each transition on the input of the Select will be sent to one of its two output depending on the boolean value of the **"Bool"** input. In Figure 4.4(b) the boolean value of the "Bool" input is logic '0' when the first transition on **i** input arrives. Thus, the first transition of the **i** input is sent to the output labelled **"False"**. Obviously the 2nd, 4th, 5th and 8th transitions are sent to the output labelled **"False"**. In the four-phase protocol the rising transition and the falling transition of an event must be sent to the same output. As shown in Figure 4.4(c) the boolean value of the "Bool" input is initially at logic '0'. Therefore, the first rising transition and falling transition are sent to the output labelled **"False"**. The boolean value then changes to logic '1' so the third event is sent to the output labelled **"True"**.

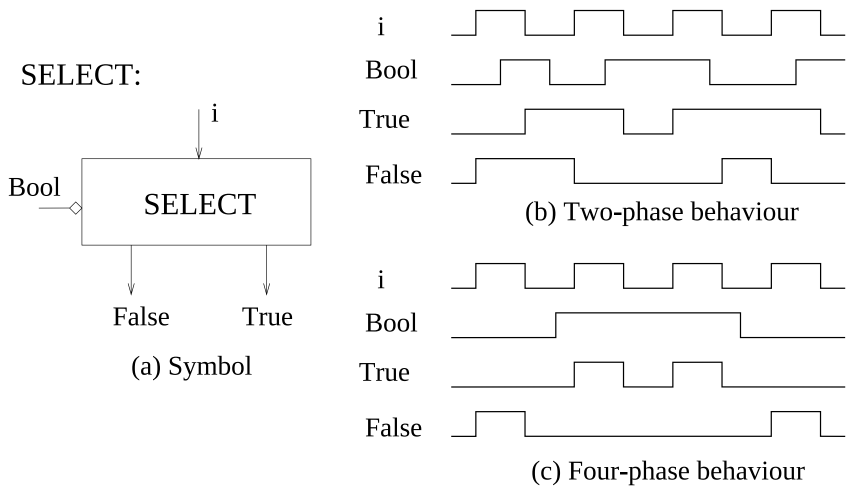


Figure 4.4: Behaviour of the Select module

Figure 4.5 shows the behaviour of a Call module. In the two-phase protocol the Call module follows one of these two sequences:

(1) a transition arrives on the R_1 input,
(2) a transition is sent out from the R output,
(3) a transition is received on the D input,
(4) a transition is sent out from the D_1 output;

(1) a transition arrives on the R_2 input,
(2) a transition is sent out from the R output,
(3) a transition is received on the D input,
(4) a transition is sent out from the D_2 output.

A new request transition must not arrive on the R_1 or R_2 input before the Call module was sent out the final transition from the D_1 or D_2 output. Figure 4.5(b) shows that there are four requests from R_1 and six requests from R_2. In total ten transitions are shown on the R output and ten corresponding complete transitions are shown on the D input. These ten corresponding complete transitions are correctly sent to D_1 or D_2. In the four-phase protocol an event consists of a rising transition and a falling transition. The Call module follows one of these two sequences:

(1) a rising transition arrives on the R_1 input,
(2) a rising transition is sent out from the R output,
(3) a rising transition is received on the D input,
(4) a rising transition is sent out from the D_1 output,
(5) a falling transition arrives on the R_1 input,
(6) a falling transition is sent out from the R output,
(7) a falling transition is received on the D input,
(8) a falling transition is sent out from the D_1 output;

(1) a rising transition arrives on the R_2 input,
(2) a rising transition is sent out from the R output,
(3) a rising transition is received on the D input,
(4) a rising transition is sent out from the D_2 output,
(5) a falling transition arrives on the R_2 input,
(6) a falling transition is sent out from the R output,
(7) a falling transition is received on the D input,
(8) a falling transition is sent out from the D_2 output.

A new rising transition must not arrive on the R_1 or R_2 input before the Call module has sent out the final falling transition from the D_1 or D_2 output. Figure 4.5(c) shows that there are three requests from R_1 and two requests from R_2. In total five requests are shown on the R output and five corresponding complete events are shown on the D input. These ten corresponding events are correctly sent to D_1 or D_2 respectively.

Figure 4.6 shows the behaviour of an Arbiter module. An Arbiter is used to order the two request signals if R_1 and R_2 arrive at the same time. Only one request is sent out and the other must wait. In the two-phase protocol Figure 4.6(b) shows that the first request arrives on R_1 and the corresponding grant is sent out from the G_1 output. No further grant will be sent out before a transition is received on the D_1 input. When the first transition is received on the D_1 input a second transition on R_1 and a first transition on R_2 arrive at the same time.

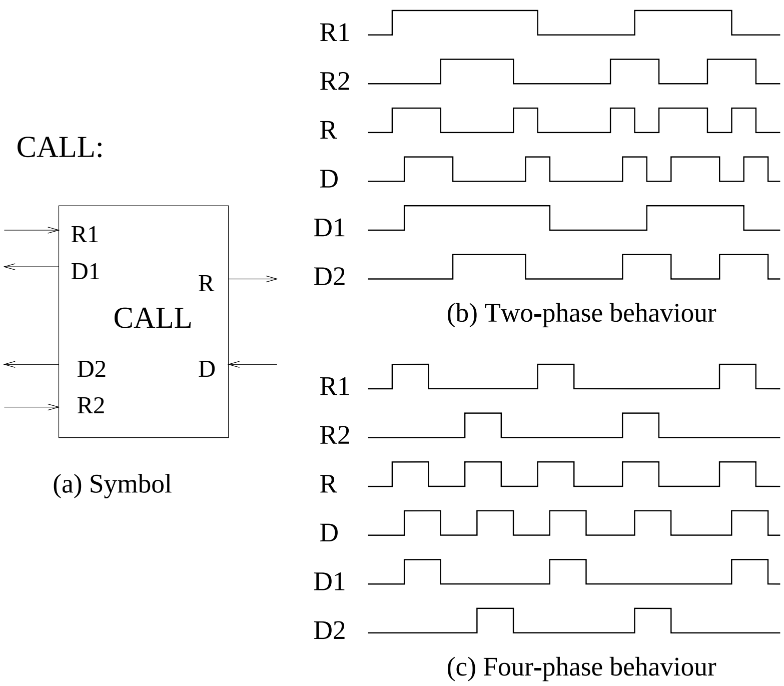


Figure 4.5: Behaviour of the Call module

The correspond grant of R_1 is sent out though this could equally have been a G_2 – in the case of coincident requests the arbiter's behaviour is non-deterministic.

The grant of R_2 is in waiting. When the second transition on the D_1 input is received the corresponding grant of R_2 is sent out. After the corresponding transition on the D_2 input is received the subsequent grant can be sent out. In the four-phase protocol an event consists of a rising transition and a falling transition. Figure 4.6(c) shows that when the first event arrives the corresponding grant is sent out from the G_1 output. It is necessary to receive the event from the D_1 input before the subsequent grant is send out. The working sequence is as follows:

(1) a rising transition arrives on the R_1 input,
(2) a rising transition is sent out from the G_1 output,
(3) a rising transition is received on the D_1 input,
(4) a falling transition arrives on the R_1 input,
(5) a falling transition is sent out from the G_1 output,
(6) a falling transition is received on the D_1 input.

When the second event on R_1 and the first event on R_2 arrive at the same time here the corresponding grant of R_1 is sent out from G_1 output. The working sequence is as follows:

(1) a 2nd rising transition on R_1 and the 1st rising transition on R_2 arrive,

(2) a rising transition is sent out from the G_1 output,

(3) a rising transition is received on the D_1 input,

(4) a falling transition arrives on the R_1 input,

(5) a falling transition is sent out from the G_1 output,

(6) a falling transition is received on the D_1 input;

(7) a rising transition is sent out from the G_2 output,

(8) a rising transition is received on the D_2 input,

(9) a falling transition arrives on the R_2 input,

(10) a falling transition is sent out from the G_2 output,

(11) the falling transition is received on the D_2 input.

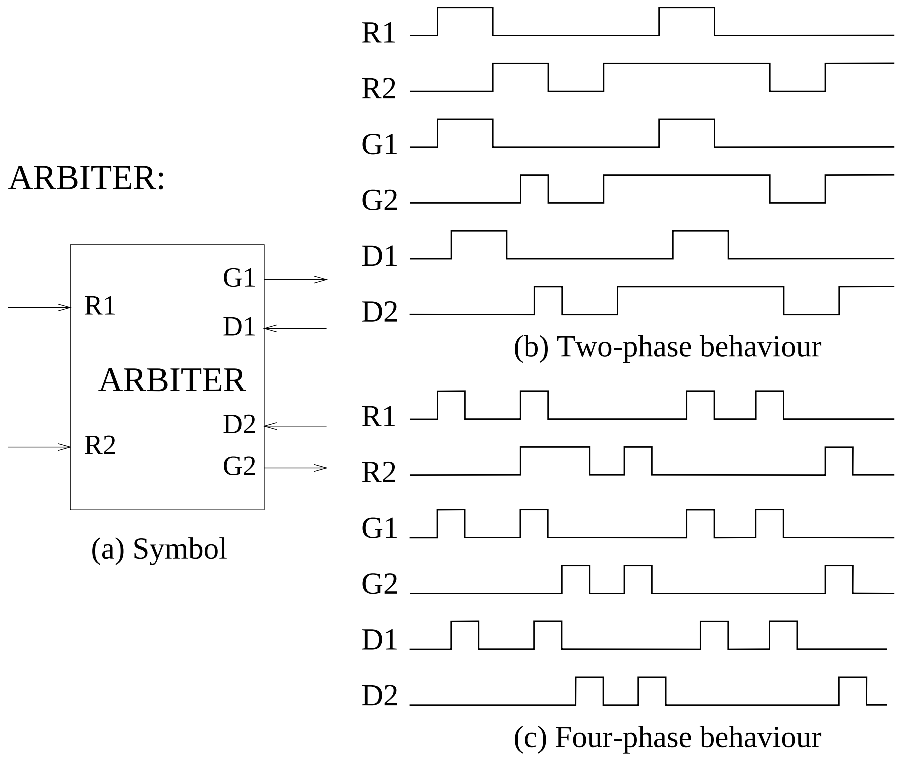


Figure 4.6: Behaviour of the Arbiter module

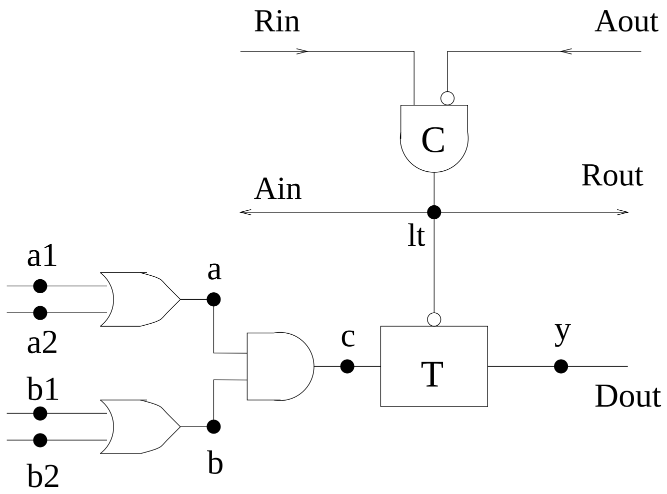


Figure 4.7: A simple 4-phase latch control circuit

4.1.3 Four-phase latch control circuits

AMULET1 was implemented using a two-phase micropipeline design style. However, AMULET2 and AMULET2e [Furb96, Furb97, Gars96a] use a four-phase micropipeline design style to improve their performance. Furber and Day developed four kinds of 4-phase latch control circuits. They are the simple, semi-decoupled, fully decoupled and long hold 4-phase latch control circuits [Furb96d] and are shown in Figures 4.7, 4.9, 4.10, and 4.11 respectively. They use the four-phase bundled data convention mentioned in the previous subsection.

The asymmetric C-gate notation shown in Figure 4.8 indicates that an input controls both edges of the output when it is connected to the main body of the gate; it controls only the rising edge when connected to the extension marked '+', and it controls only the falling edge when connected to the extension marked '−'.

As shown in Figure 4.7 **Rin**+ must wait for **Aout**− and **Rin**− must wait for **Aout**+ to proceed to **lt**+ and **lt**− when the simple 4-phase latch control circuit is used, this may lead to poor performance. However, the cost is very low.

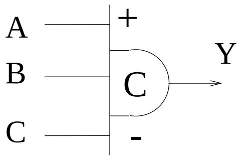


Figure 4.8: Asymmetric C-gate notation

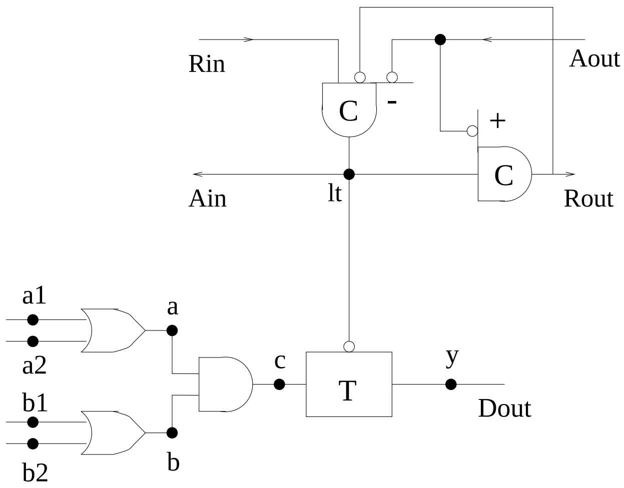


Figure 4.9: A semi-decoupled 4-phase control circuit

The semi-decoupled and fully decoupled latch control circuits may be employed to solve this problem. In the semi-decoupled latch control circuit (see Figure 4.9) **Rin**+ does not need to wait for **Aout**− and can proceed to **lt**+ after **Rout**−. But **Rin**− still needs to wait for **Aout**+ to proceed to **lt**−. The fully decoupled latch control circuit (see Figure 4.10) is much faster than simple and semi-decoupled control circuits. When it is used **Rin**+ can proceed to **lt**+ after **Rout**− and **Rin**− can through to **Ain**− after **Ain**+ shortly.

For some applications, it may be necessary that the latch holds the data stable until **Aout** goes low. The long hold 4-phase control circuit shown in Figure 4.11 can serve this purpose.

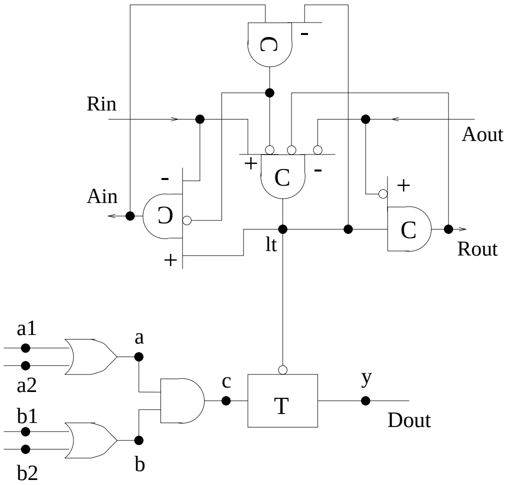


Figure 4.10: A fully decoupled 4-phase control circuit

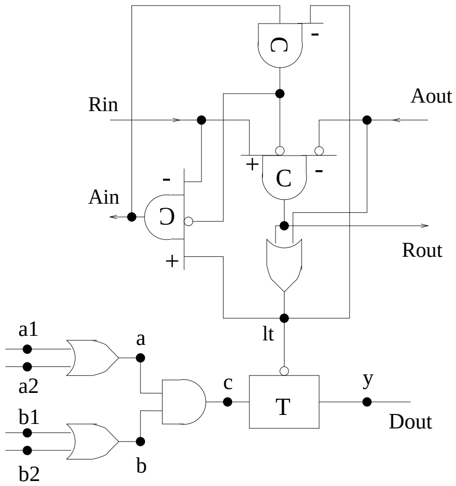


Figure 4.11: A long hold 4-phase control circuit

4.1.4 A comparison of two- and four-phase control

To demonstrate the difference between two- and four-phase control, various micropipeline stages, such as a 2-phase stage (Figure 2.10), a 4-phase simple stage (Figure 4.7), a 4-phase semi-decoupled stage (Figure 4.9), a 4-phase fully decoupled stage (Figure 4.10), and a 4-phase long hold stage(Figure 4.11) were implemented in VHDL. Then the various stages, modelled in structural VHDL, were connected into the pipeline structures shown in Figure 4.12. Each Sender is a behavioural VHDL model which makes data available on the inputs and sends a request to the pipeline 200 times. Each Receiver is also a behavioural VHDL model which receives the request from the pipeline and generates the corresponding acknowledge back to the pipeline 200 times. The simulation results were as shown in Figure 4.13. Because the entire waveforms are very large, only the final parts of the waveforms are shown in the figure. Signals, Srin1, Sain1, Srout1, Saout1, Srout2, Saout2, Srout3, and Saout3 belong to the pipeline composed of 4-phase simple latch control circuits. Signals, Lsrin1, Lsain1, Lsrout1, Lsaout1, Lsrout2, Lsaout2, Lsrout3, and Lsaout3 belong to the pipeline composed of long hold 4-phase latch control circuits. Signals, Fsrin1, Fsain1, Fsrout1, Fsaout1, Fsrout2, Fsaout2, Fsrout3, and Fsaout3 belong to the pipeline composed of fully decoupled 4-phase latch control circuits. Signals, Ssrin1, Ssain1, Ssrout1, Ssaout1, Ssrout2, Ssaout2, Ssrout3, and Ssaout3 belong to the pipeline composed of semi-decoupled 4-phase latch control circuits. Signals, Srin21, Sain21, Srout21, Saout21, Srout22, Saout22, Srout23, and Saout23 belong to the pipeline composed of 2-phase transparent latch control circuits. The simulation output shows that the 2-phase pipeline finishes the 200 operations first. The 4-phase simple pipeline is the second one to finish the operations. The 4-phase fully decoupled and semi-decoupled pipelines finish the operations at almost the same time. The 4-phase long hold pipeline is the last one to finish.

Delays are added within each Sender and Receiver. To reduce the effect of delays

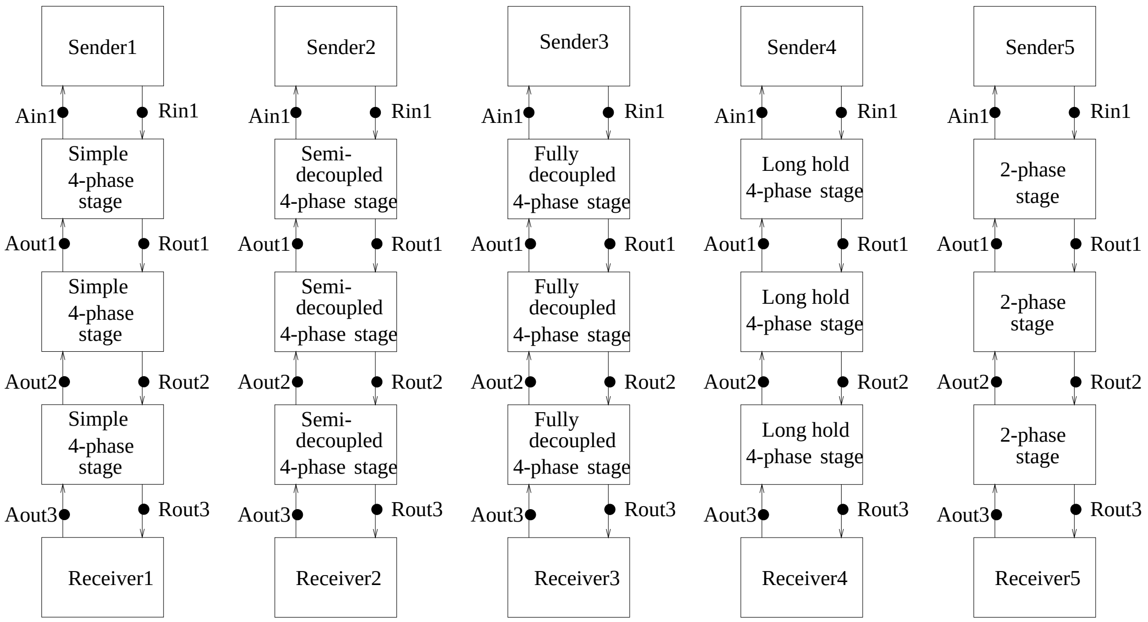


Figure 4.12: Pipeline structure used for simulations(1)

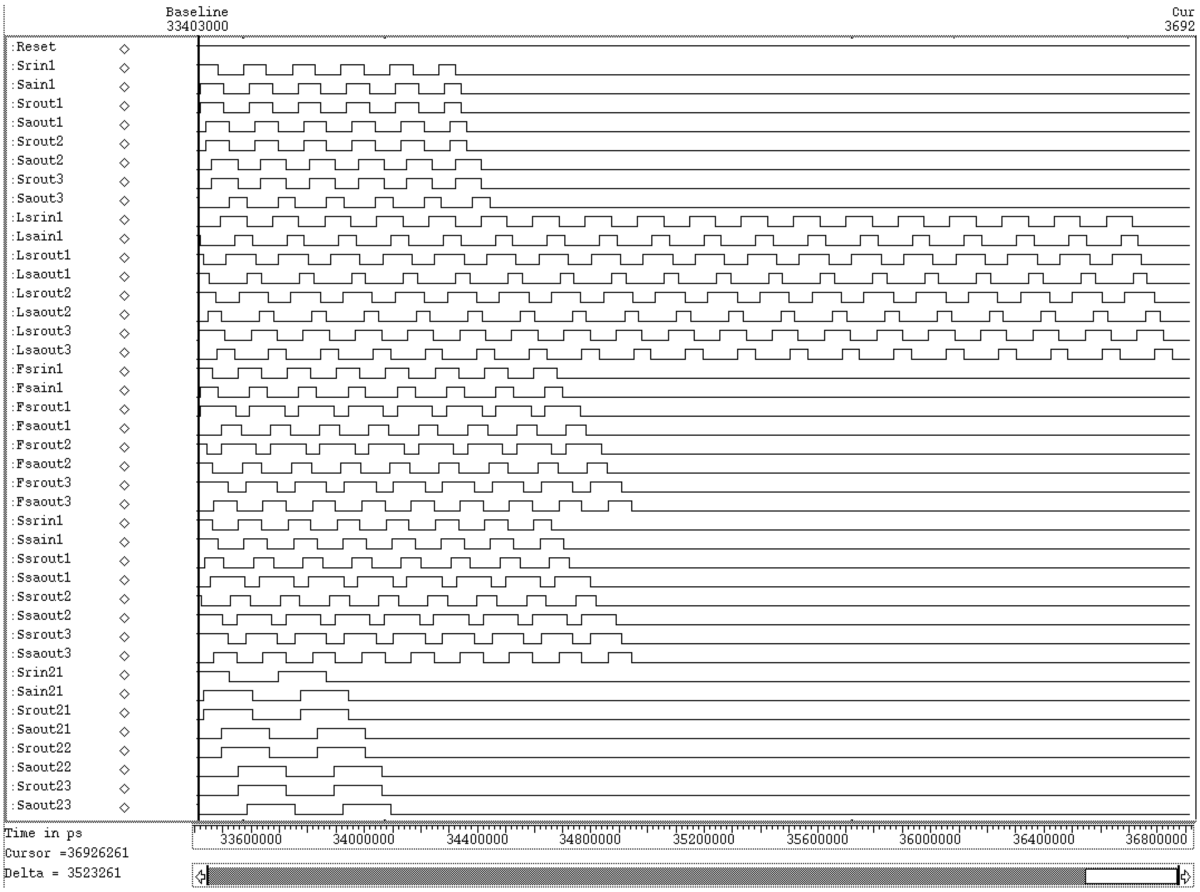


Figure 4.13: The simulation result(1)

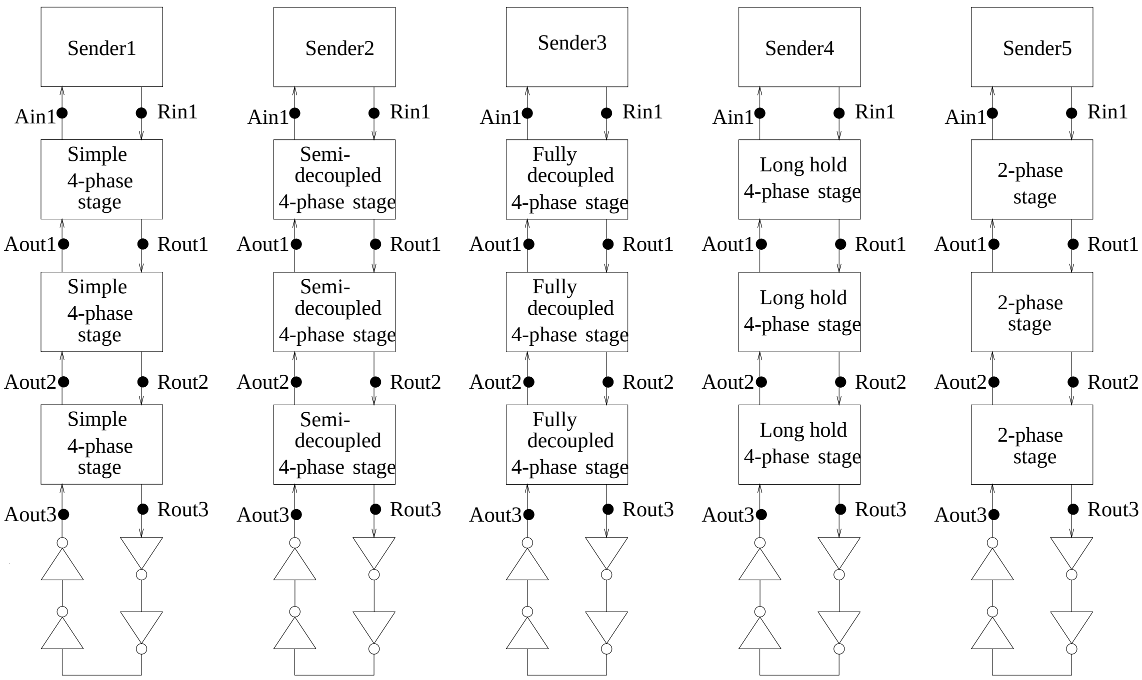


Figure 4.14: Pipeline structure used for simulations(2)

and to get more accurate simulation results, each Receiver was replaced by four inverters. The configurations are as shown in Figure 4.14. The simulation result is shown in Figure 4.15. It shows that the 4-phase simple pipeline finishes the 200 operations early. The 4-phase fully decoupled and semi-decoupled pipelines finish the operations at almost the same time. Then the 2-phase pipeline finishes the operations. The 4-phase long hold pipeline is still the last one to finish.

Next, a delay composed of eight inverters is added at the front of each stage. For each stage the request signal first flows through the eight inverters and then reachs the next stage. This configuration is shown in Figure 4.16 and the simulation results are shown in Figure 4.17. These show that the 2-phase pipeline finishes the 200 operations first. The 4-phase simple pipeline is the second one to finish the operations. The 4-phase fully decoupled and semi-decoupled pipelines finish the operations at almost the same time. The 4-phase long hold pipeline is still the last one to finish.

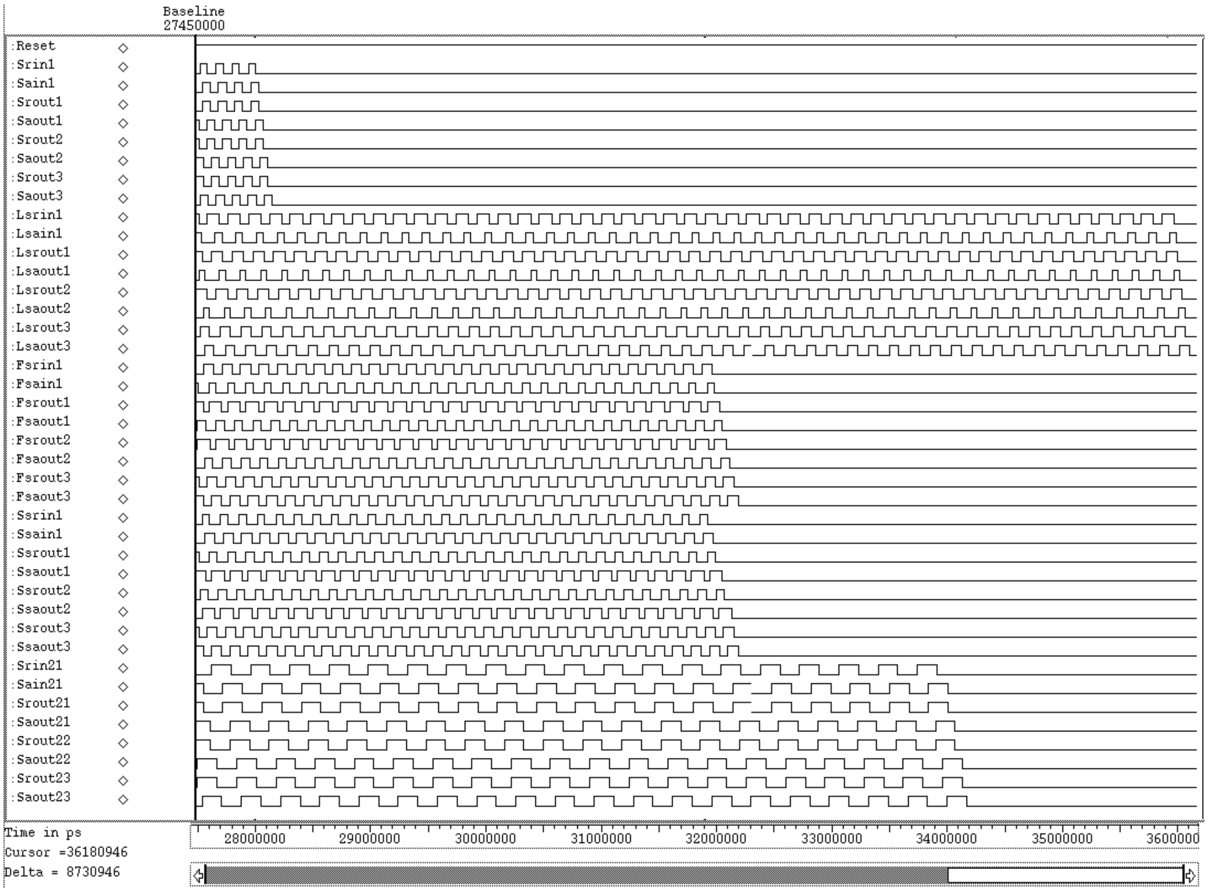


Figure 4.15: The simulation result(2)

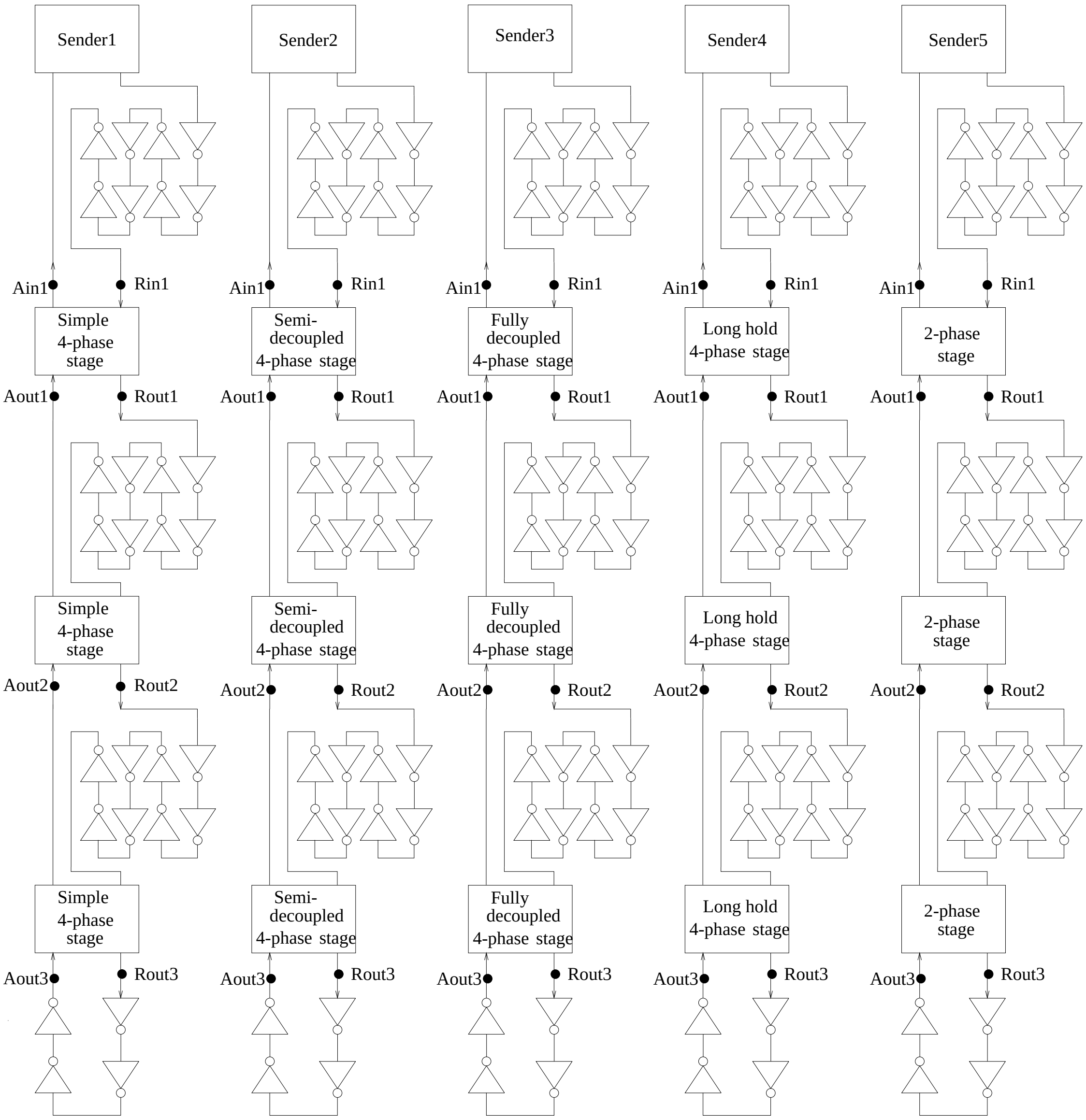


Figure 4.16: Pipeline structure used for simulations(3)

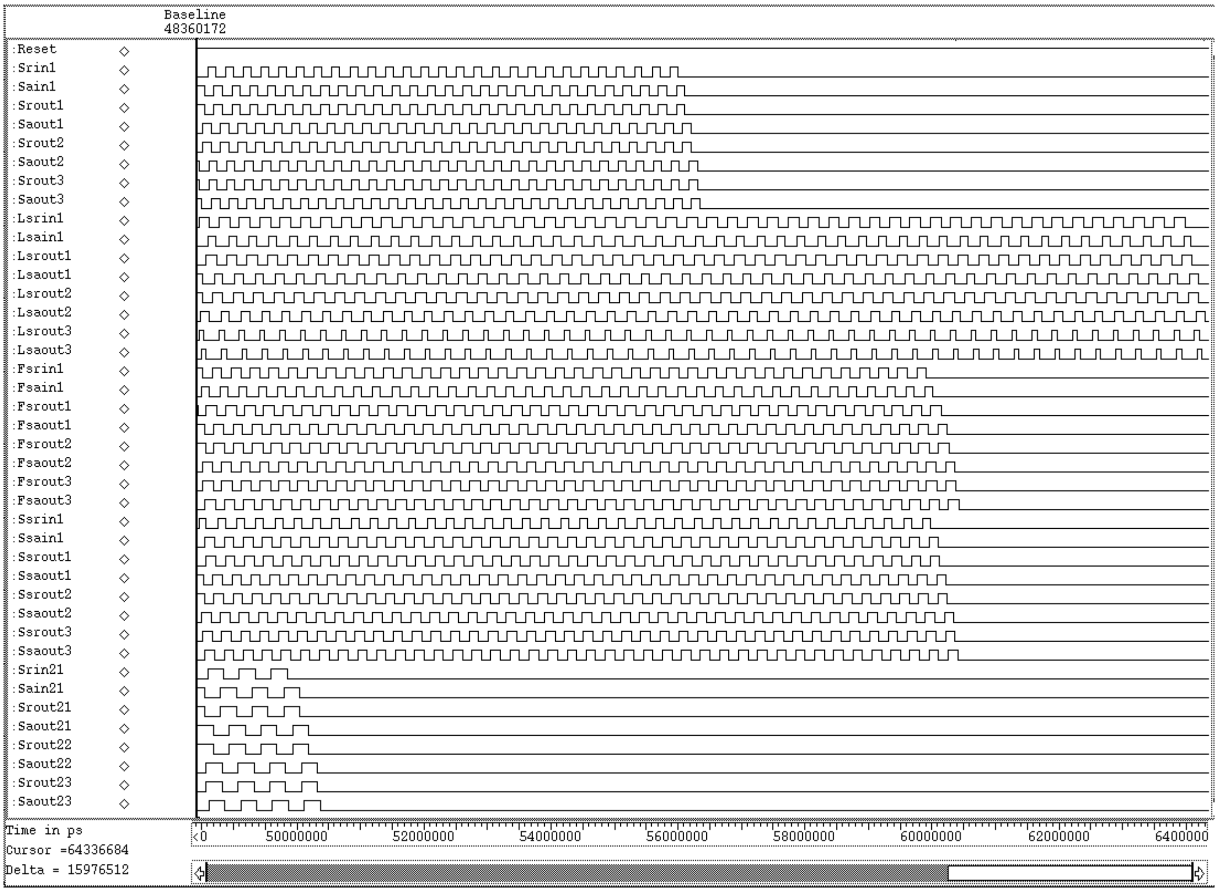


Figure 4.17: The simulation result(3-1)

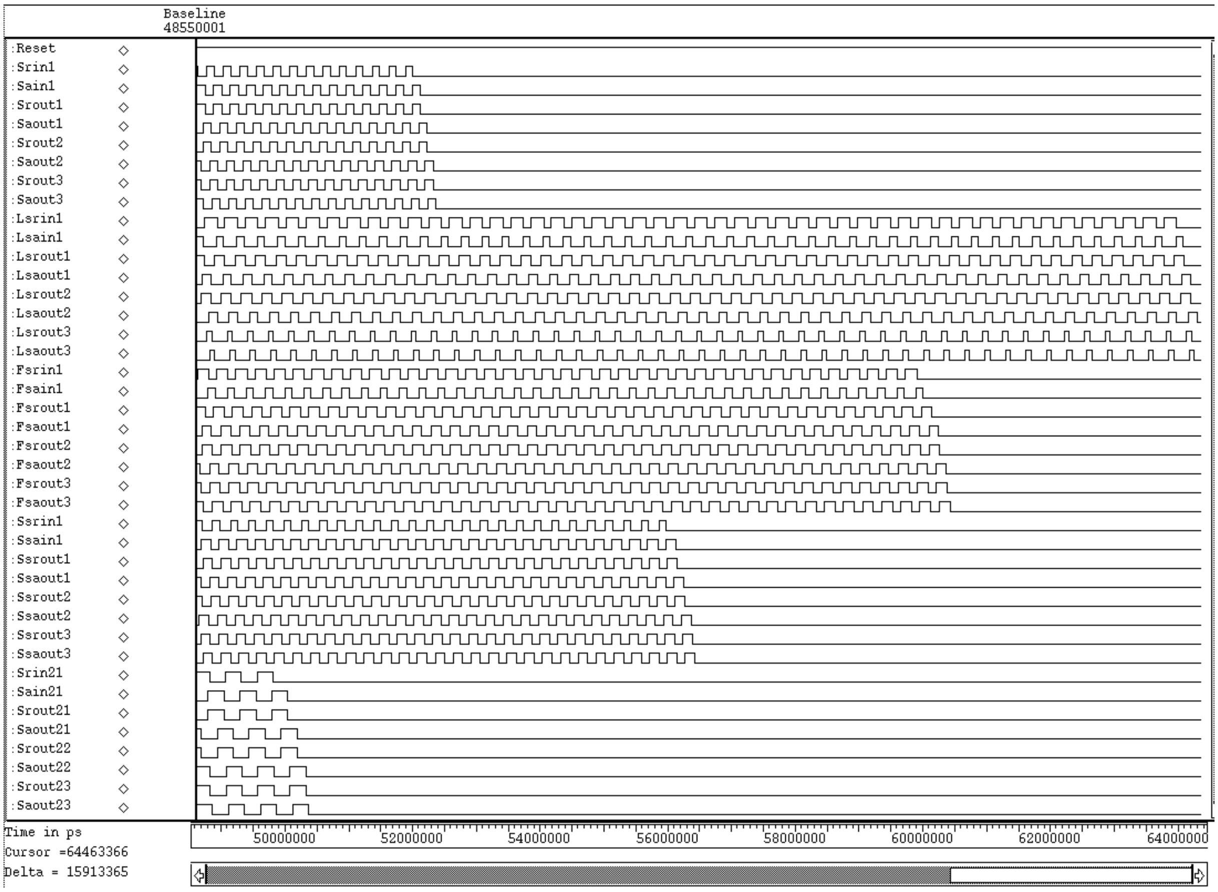


Figure 4.18: The simulation result(3-2)

To show how fast these pipelines can pass data, the Sender may be changed so that it inverts the signal of the acknowledge line for inputs and drives the request line after 10 nano-seconds. Then the simulation result is as shown in Figure 4.18. This shows that the 2-phase pipeline finishes the 200 operations early. The 4-phase simple pipeline is the second one to finish the operations. The 4-phase semi-decoupled pipeline is the third to finish the operations. The 4-phase fully decoupled pipeline is the fourth to finish the operations, and the 4-phase long hold pipeline is still the last one to finish the operations.

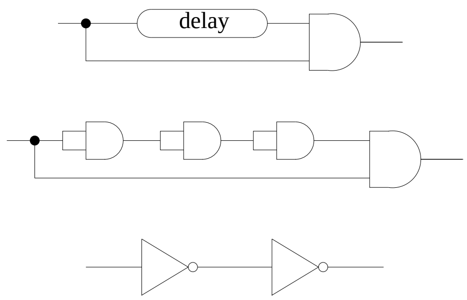


Figure 4.19: Delay models

It is not only the active transitions that flow through the eight inverters delay, but also the return transitions. The symmetric delay reduces the performance. Another test was to replace the delays for the 4-phase pipelines by asymmetric delays [Seit80]. The asymmetric delay can be implemented as shown in Figure 4.19 using a 4-gate delay for the active transitions and a 1-gate delay for the return transitions.

In Figure 4.20 the asymmetric delays are shown implemented by eight 2-input AND gates. This gives an 8-gate delay for the active transitions and a 1-gate delay for the return transitions. However, the delay for the 2-phase pipeline is composed of eight 2-input AND gates simply connected in series as there is no return transition in 2-phase signalling. The simulation result is shown in Figure 4.21. This shows that the 4-phase fully decoupled pipeline finishes the 200 operations first. The 2-phase and 4-phase long hold pipelines are second to finish the operations. The 4-phase simple pipeline is the fourth to finish and the 4-phase semi-decoupled pipeline is the last one to finish.

In Figure 4.22 there are a 4-phase fully decoupled 3-stage pipeline with 8:1 (8-gate delay for active transitions and 1-gate delay for return transitions) asymmetric delay for each stage, a 2-phase 3-stage pipeline with 8-gates delay for each stage, a 4-phase fully decoupled 6-stage pipeline with 4:1 asymmetric delay for each stage, and a 2-phase 6-stage pipeline with 4-gates delay for each stage. The simulation result of this configuration is shown in Figure 4.23. This shows that the 2-phase

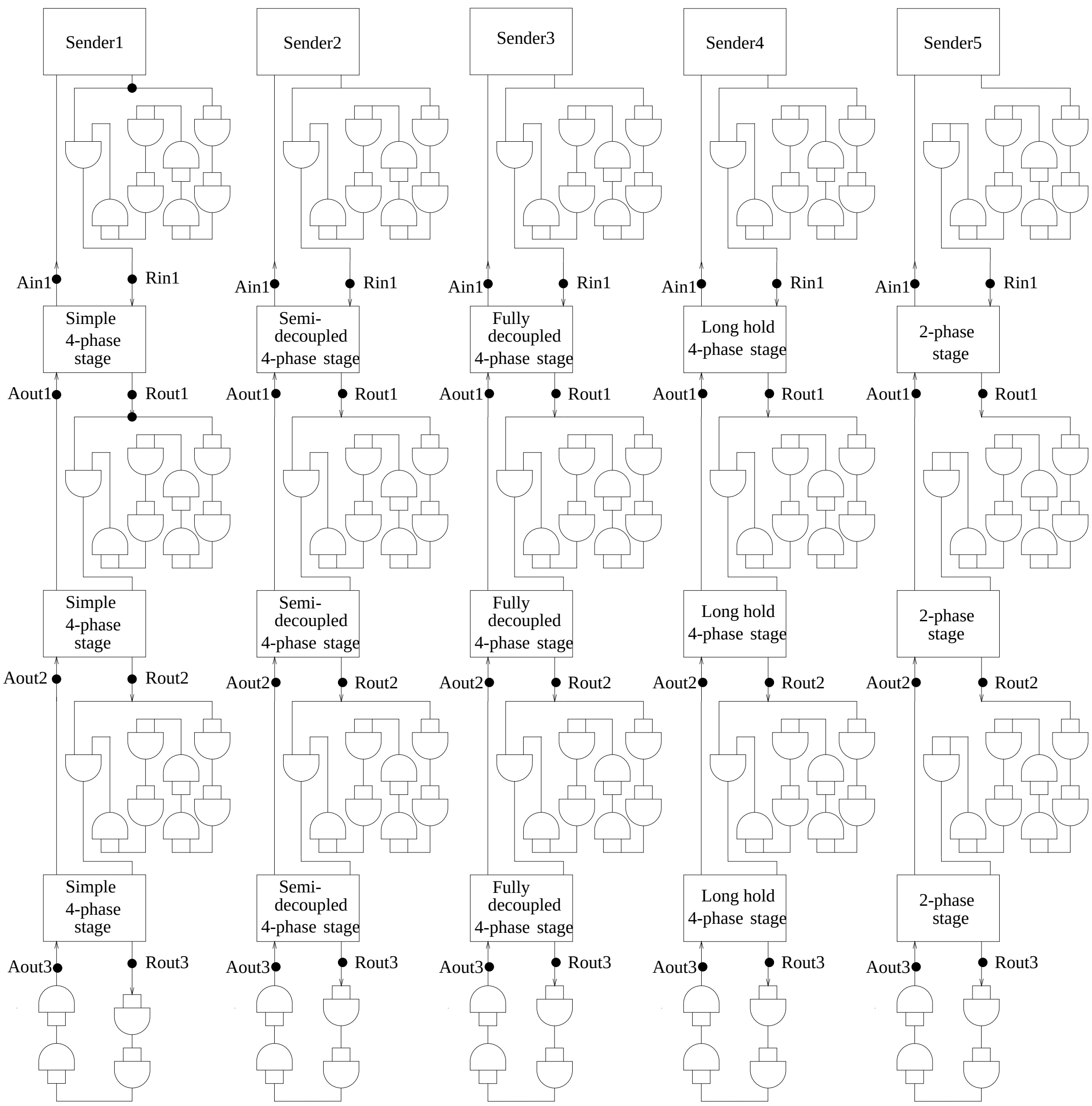


Figure 4.20: Pipeline structure used for simulations(4)

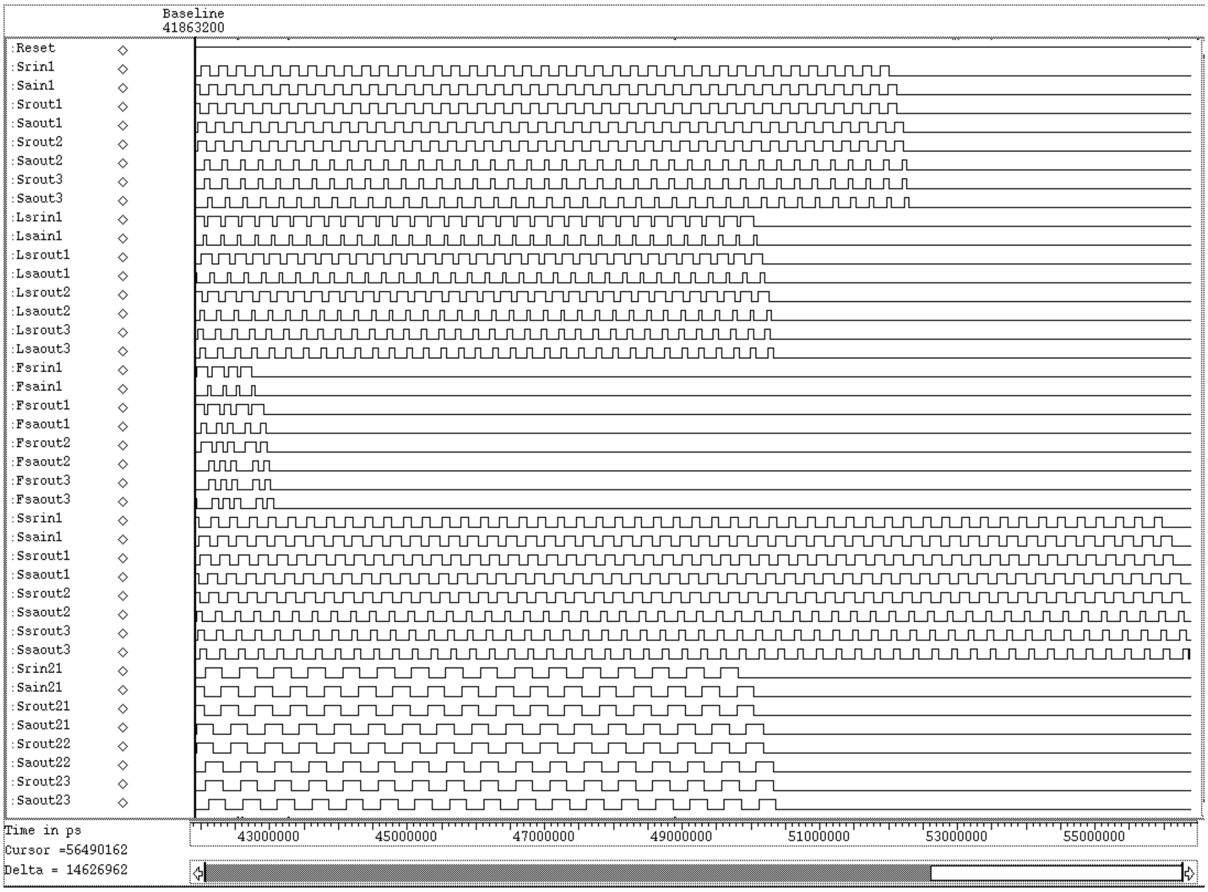


Figure 4.21: The simulation result(4)

6-stage pipeline with 4-gates delay for each stage finished the operations first. The 4-phase fully decoupled 3-stage pipeline with 8:1 asymmetric delay for each stage finishes the operations a little earlier than the 2-phase 3-stage pipeline with 8-gates delay for each stage which is the last one to finish.

From the simulations of the various configurations some insight is obtained. It is summarized as follows:

1. By carefully partitioning the computations into several stages of a 2-phase micropipeline, the circuit will speedup. However, the cost and area will grow and it will also use more power.

2. Carefully partitioning the computations into stages of 4-phase micropipelines gives less speedup. The cost and area will grow and it will also use more power.

3. The reason is that the return to zero causes some delay, even if asymmetric delays are used for active and return transitions. The minimum delay for the return transitions still exists.

4. If the delay is equal to the delay of 2-gates two-phase may be faster than four-phase. The reason is that it is only 2:1 if asymmetric delay is applied to four-phase. If the delay is equal to one gate delay two-phase is faster than four-phase.

5. If the two-phase event-controlled storage element is used two-phase may be faster than four-phase. However, the switch circuits for each data bit require more area and cost.

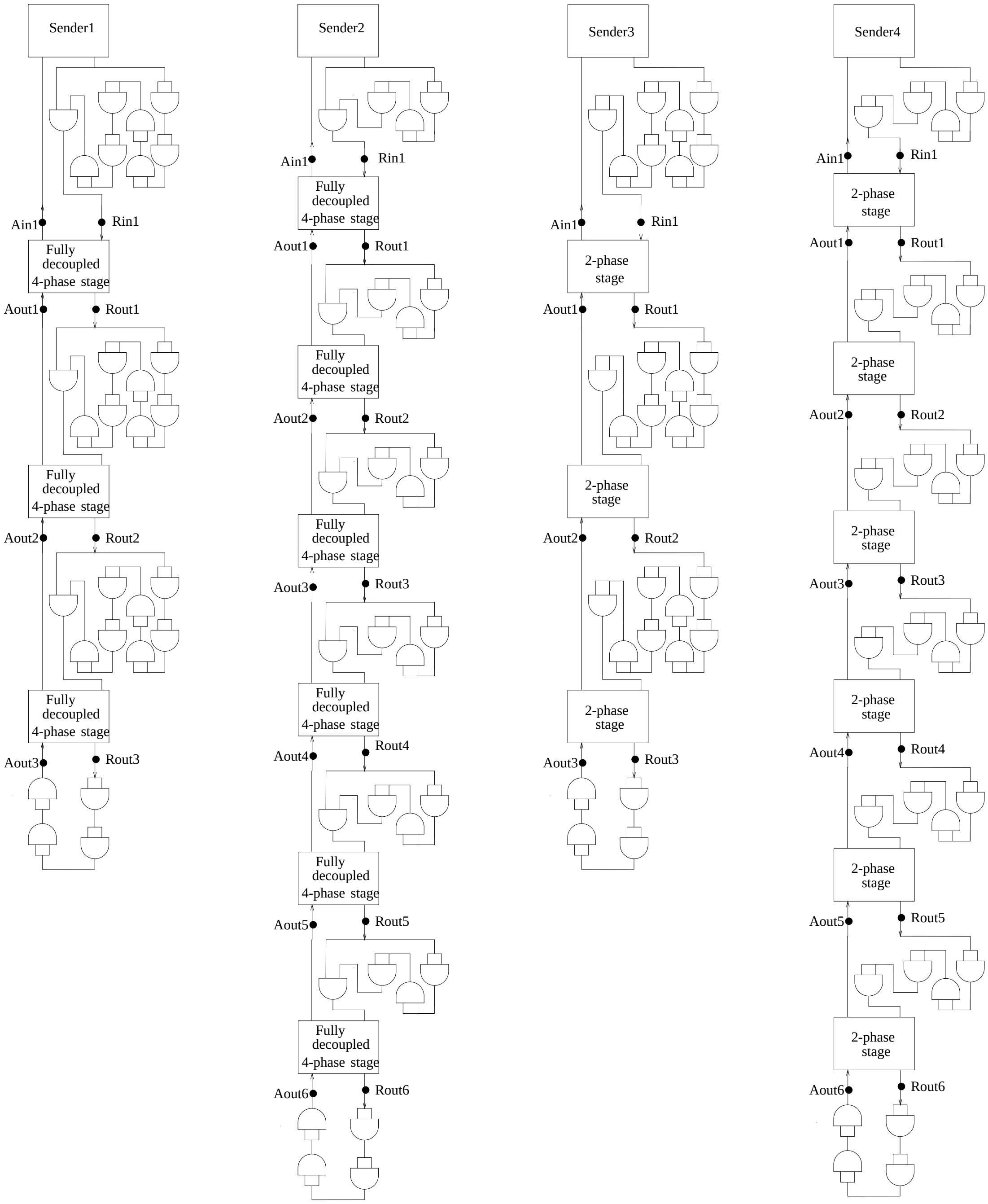


Figure 4.22: Pipeline structure used for simulations(5)

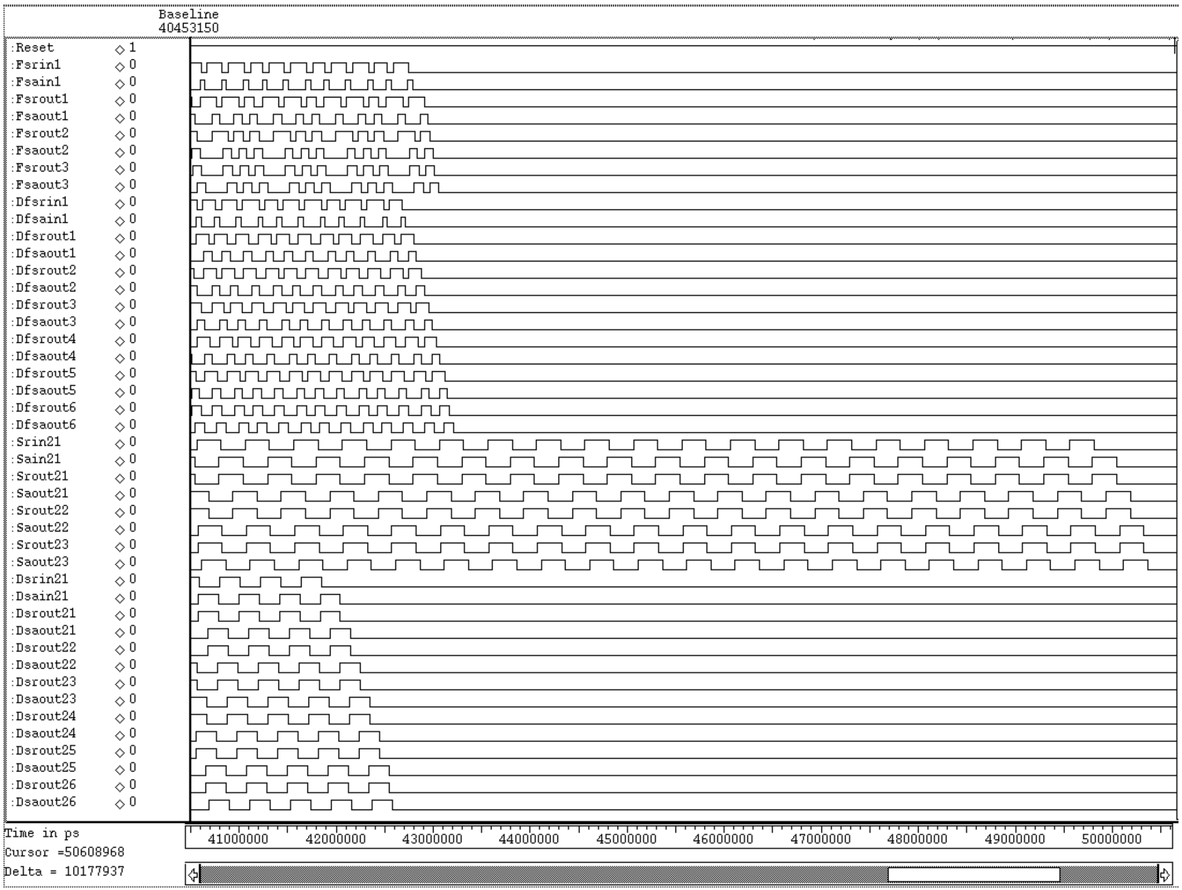


Figure 4.23: The simulation result(5)

Another previously described pipeline [Appl97] works with 2-phase signalling. The circuit is quite simple as shown in Figure 4.24. The behaviour of the control block is shown in Figure 4.25. When the **Control** signal is **Hi** the transitions on the **In** signal line will appear on the **Out** signal line. When the **Control** signal is **Lo** the **Out** signal line will stay in the same state it was in before the **Control** signal went **Lo**. This means its function is like a transparent latch. Therefore, a functionally equivalent circuit may be drawn as shown in Figure 4.26. The state pipeline is not a speed-independent circuit. Some timing constraints must be followed. Although this kind of circuit may be potentially faster than the 2-phase micropipeline shown in Figure 2.10 and the fully decoupled control circuit shown in Figure 4.10, much care is needed to check the circuit is safe. In Figure 4.26 the high-active transparent latches are used. This means that the data is hold if the **lt** signal is **Lo** and the latches are transparent if the **lt** signal is **Hi**. If two set of circuits shown in Figure 4.26 are connected in series the latch of the first stage holds data just a short time which is equal to the delay of a transparent latch. Therefore, delay must be added between **Ain** and **Rout** if the circuit contains logic processing. The time between two transitions of the **Rin** must be greater than or equal to the time of the signal through the transparent latch labelled **SEND**, the XOR labelled **UNTIL** and the Inverter. The time between the transitions on the **Ain** and a new request on the **Rin** must be greater than or equal to the time of the signal through the XOR labelled **UNTIL** and the Inverter.

4.2 Two-phase and Four-phase libraries

2- and 4-phase Control modules and some mathematical, logical, and relational operation components are needed to form the components of circuits during synthesis. To simulate the synthesized circuits the behavioural VHDL models of

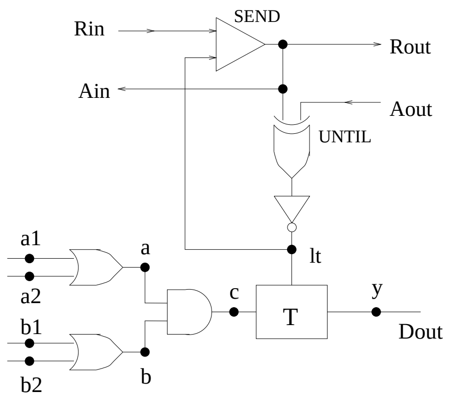


Figure 4.24: State pipeline

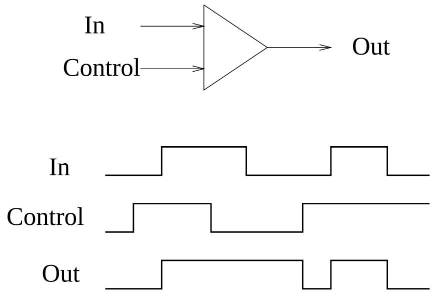


Figure 4.25: The behaviour of the control block used in the state pipeline

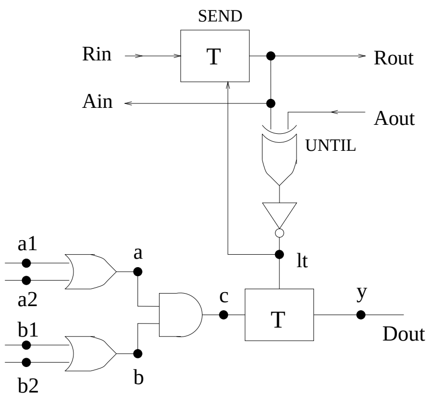


Figure 4.26: A circuit functionally equivalent to the state pipeline

these components must exist. The behavioural VHDL models of some components are shown in Appendix A. After using "VerilogIn" to load the synthesized circuits into Cadence schematics all components used must exist and have layout. Then the synthesized circuits can be laid out. The Cell library of 2- and 4-phase control modules and some components are shown in Appendix B.

Figure 4.27 shows the implementation of the 4-phase Select modules. The 'early' handshaking protocol is used in Figure 4.27(a). This uses the rising edge of the *req* signal to indicate 'data available' and the rising edge of the *ack* signal to indicate 'data latched' [Furb96c]. The falling edges are return to inactive state actions. As shown in Figure 4.27(a), the outputs of the two asymmetric C-gates are initially logic '0'. When the data is available and the signal *Bool* is logic '0', the rising edge of the *req* signal enables the output labelled 'False' to become logic '1'. Then the output of the NOR gate will fall to logic '0'. This will disable the other asymmetric C-gate from turning its output to logic '1'. The output labelled 'False' remains at logic '1' until the *req* signal returns to logic '0'. The output of the NOR gate will then become logic '1' to enable the subsequent rising edge of the *req* signal to be delivered to one of the outputs of the two asymmetric C-gates depending on the logic value of the *Bool* signal. Alternatively, the 'broad' handshaking protocol is used in Figure 4.27(b). This uses the rising edge of the *req* signal to indicate 'data available' and the falling edge of the *ack* signal to indicate 'data latched'. As shown in Figure 4.27(b), the signal *Bool* does not change between the rising edge of the *req* signal and the falling edge of the *ack* signal. This enables the rising and falling edges of the *req* signal to be delivered to the same output, labelled 'True' or 'False' depending on the logic value of the *Bool* signal.

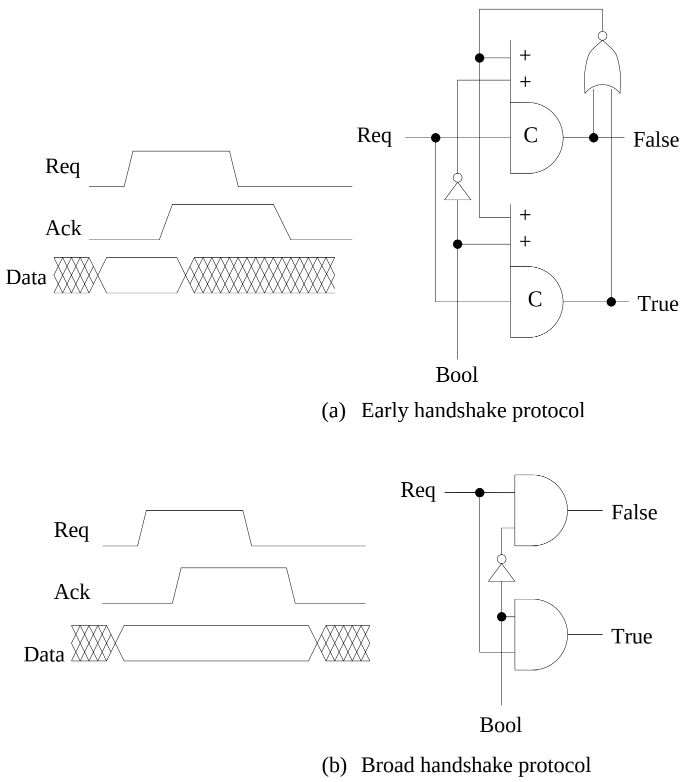


Figure 4.27: A 4-phase Select module

4.3 The design of the synthesizer

4.3.1 Configuration file

Circuits may not have only one set of inputs and outputs. This means circuits may have more than one signal called Rin, Ain, Rout, and Aout. Therefore, a configuration file is required to describe the bundle of the handshake signals and their corresponding data. For example, Figure 4.28 shows a circuit to be synthesized. There are two pairs of input data bundles and two pairs of output data bundles. A configuration file which contains the following information is required. Then the synthesizer may follow the data path and this information to produce the corresponding control circuits.

```
defbundle Rin1, Ain1, A, B, C;
defbundle Rin2, Ain2, D, E;
```

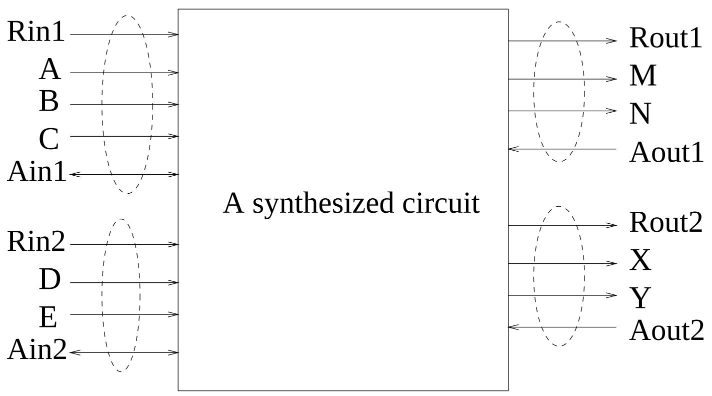


Figure 4.28: A synthesized circuit

```
defbundle Rout1, Aout1, M, N;
defbundle Rout2, Aout2, X, Y;
```

4.3.2 Options

Speed, cost, and power are three parameters which a synthesizer may attempt to optimize. Before proceeding to synthesis the target optimization must be chosen. If no option is specified the default option is followed. The default option is speed. More detail about optimization is given in chapter 5.

4.3.3 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. To implement the synthesizer, rules for partition, conversion, and the production of control circuits described in chapter 3 must be followed.

The "if A='0' then C:=B; else D:=E; end if;" statement may be implemented as shown in Figure 4.29 without initial value and Figure 4.30 with initial values. If this if statement is the only source of the values of the variables **C** and **D** there are two cases to consider. The first case is that there may be a mistake in the VHDL input file. Warning messages may be added to remind the user. The second case is that these variables are not used in unspecified conditions. The circuits can be implemented using the method for general assignments. Figure 4.29 shows this example. If the if statement is not the only source of the values of the variables **C** and **D** and there are some preceding statements which have specified values of the variables **C** and **D**, the circuit shown in Figure 4.30 keeps the values of the variables **C** and **D** unchanged if the corresponding circuit is executed.

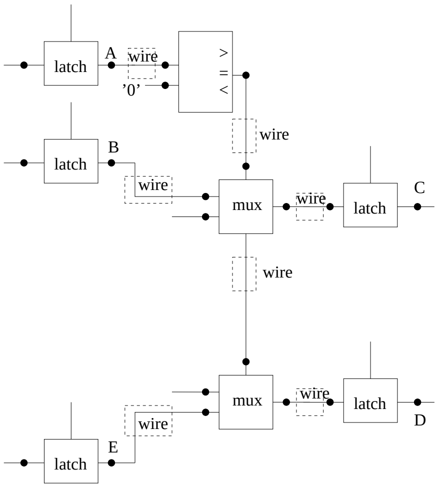


Figure 4.29: The implementation of the statement “if A='0' then C:=B; else D:=E; end if;” without initial value

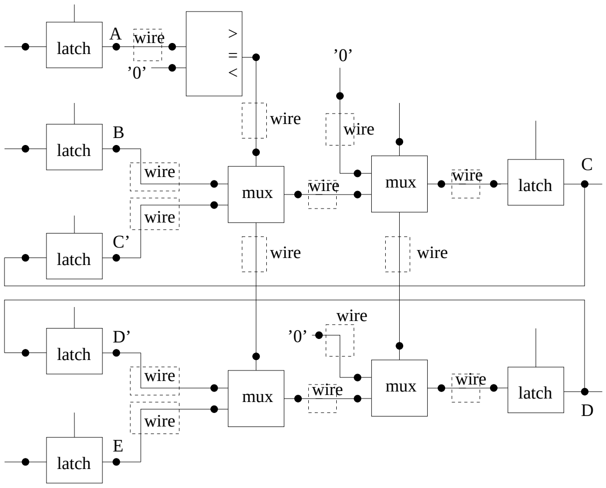


Figure 4.30: The implementation of the statement "if A='0' then C:=B; else D:=E; end if;" with initial values

A possible implementation of the statement "A := ?; while A='0' loop A:=B + C; A := B; end loop;" is shown in Figure 4.30.

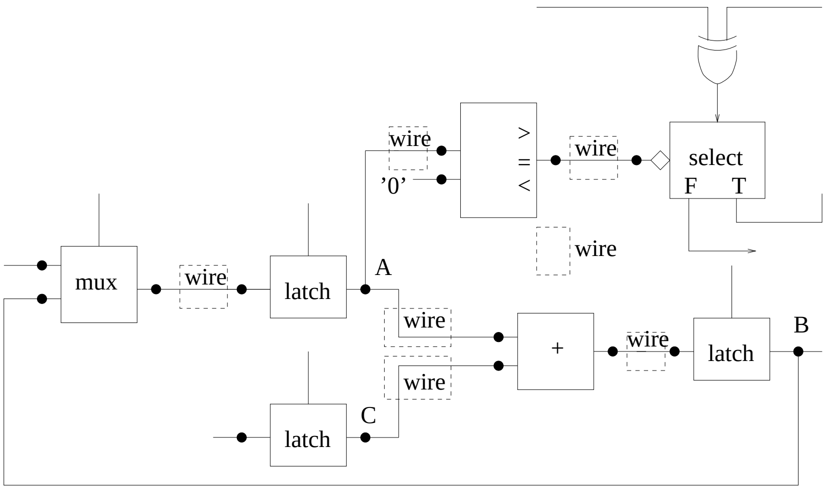


Figure 4.31: A possible implementation of the statement "A := ?; while A='0' loop A:=B + C; A := B; end loop;".

The main program has five PASSes, the role of each PASS is as follows:

PASS 1: create the registers corresponding to each variable.

PASS 2: partition all statements into a number of stages.

PASS 3: add a selector at the front of each register where required.

PASS 4: produce the circuits within each stage.

PASS 5: produce the structural VHDL files of each stage and a structural VHDL file of the top level.

The normalization subprogram is used to convert some internal component models to actual component models or change the terminal name of some components, such as " >= ", " <= ", "/ = ", "ltlatch", " + ", " − ". The RESET subprogram is used to connect the RESET signal of the TOP_STG to each stage and connect the RESET of each stage to the components within each stage. A subprogram which is used to produce assignments to enable the external signals to drive the internal signals or internal signals to drive external signals is implemented. A subprogram which is used to produce some internal signals for non-connections is implemented.

4.3.4 Functions

The following is a summary of functions which this version of the synthesizer can achieve.

1. It can synthesize micropipeline circuits whose configurations include fork, join, and serial.
2. It can support adders, subtractors, multipliers, dividers, and some logic operations. However, the components which can execute these mathematical and logical operations must exist in the library.
3. It can implement circuits with feedback.

Chapter 5

Case studies

5.1 The subsequent design work

A proposed design flow was shown in Figure 3.29. The 2-phase structural Verilog output of the test example shown in Figure 3.26 was successfully imported into the Design Framework II database (Cadence) using VerilogIn. To successfully import the structural Verilog output into Cadence all components used to compose the structural Verilog must exist in the Cadence library. The input and output pins of the Verilog components must be consistent with the Cadence components. Otherwise, the circuits cannot be imported into Cadence.

After importing the structural Verilog output into Cadence Verilog simulations are helpful to check whether the design is correct. However, the simulation may fail due to the Verilog views of the components with feedback not existing in the Cadence library. For example, a schematic contains components, c2 and toggle2 shown in Appendix B. If the Verilog views of c2 and toggle2 do not exist in the library the Verilog simulation for the schematic will fail. The reason is that the Verilog simulator cannot successfully simulate transistor circuits with feedback in some cases. It is necessary to create the Verilog views for each component with feedback if Verilog simulation is required before we layout the schematic.

The layout of the schematic was automatically generated using the function of Layout Synthesis. Then TimeMill was used to do a postlayout simulation to make sure that the design is correct. To do that TimeMill needs to read the HSPICE netlist of the layout. Before generating the HSPICE netlist the extracted view of the layout must first be generated. The layout of stg1 was successfully extracted and the Hspice netlist of the extracted view was also generated successfully. The expected output was obtained. Figure 5.1 shows the output of TimeMill simulation of the whole circuit.

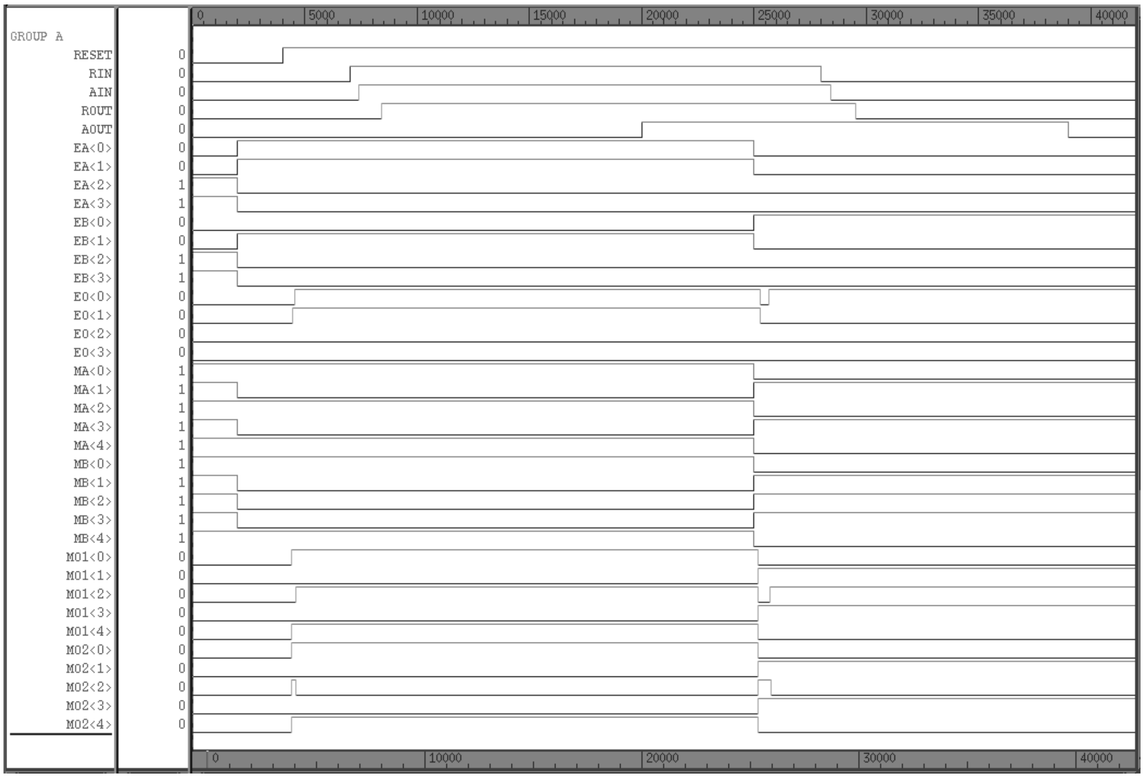


Figure 5.1: The output of TimeMill simulation of the whole circuit

5.2 While loop examples

5.2.1 A simple while loop example

```
ENTITY circuit IS
  PORT ( RIN, AOUT : IN  MVL;
        AIN, ROUT : OUT  MVL;
        A : IN MVL_VECTOR ( 4 DOWNTO 0 );
        E : OUT MVL_VECTOR ( 4 DOWNTO 0 );
        RESET : IN MVL );
END circuit;
architecture BEHAVIORAL of circuit is
begin
  U1: process
    variable Qrin, Qaout : MVL;
    variable y, w : MVL_VECTOR ( 4 DOWNTO 0 );
    begin
      y := a; w := "00101";
      while ( w >= "00001" ) loop
        y := y + "00001";  w := w - "00001";
      end loop;
      e <= y;
    end process U1;
end BEHAVIORAL;
```

Figure 5.2: A simple while loop example

Figure 5.2 shows a while loop example. The synthesizer read this input and produced the output shown in Appendix D.1. The schematic of the output is shown in Figure 5.3. The 2-phase while loop control circuit shown in Figure 3.21 is used in this example.

5.2.2 4-phase while loop control circuits

Figure 5.4 shows a 4-phase while loop control circuit which can perform the following loop operation. The circuit continuously performs the loop operation if the greater output of the comparator remains true.

```
w = A;
while ( w > "00101" ) loop
    w = w - "00001";
end loop;
```

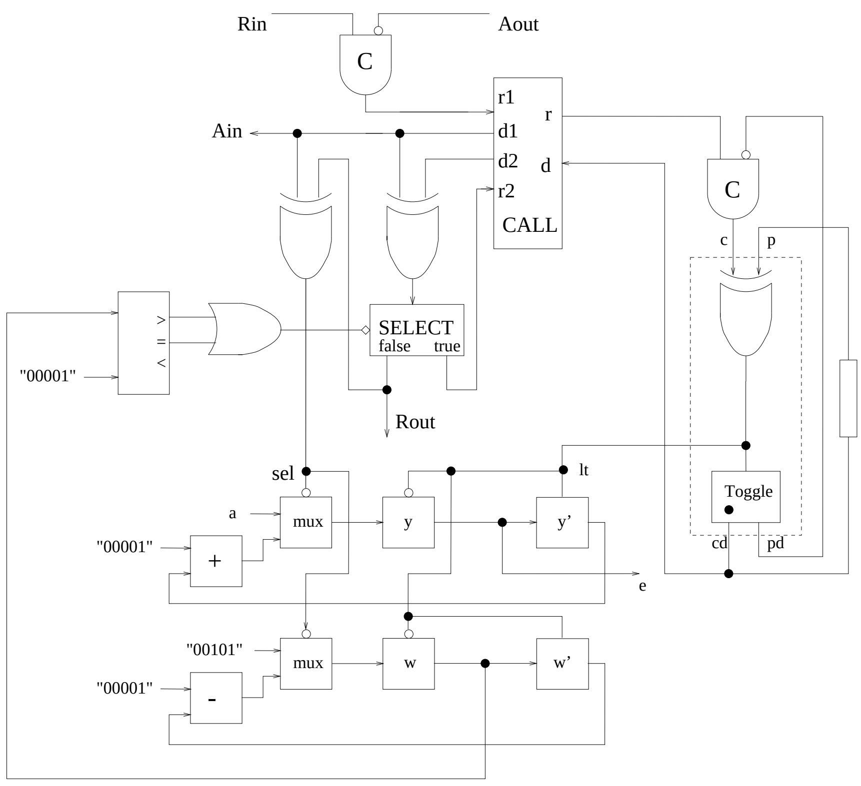


Figure 5.3: The schematic of the output of the synthesizer

Figure 5.5 shows the waveform of the structural VHDL simulation of the 4-phase while loop control circuit shown in Figure 5.4.

In the 4-phase handshaking protocol the request and acknowledge signals will return to their inactive state to start a new cycle. Activating the reset of related components is a simple and fast way to deliver a falling transition from the request input to the request output. Figure 5.6 shows a 4-phase while loop control circuit where the reset is activated if the **Rin** is **Low** and the **Aout** is **High**. The circuit can also perform the same loop operation as shown in Figure 5.4. It will terminate the loop operation if the 'equal' output of the comparator becomes true. The number of the inverters between the output of the asymmetric C-gate and the junction **p1** must be odd. Figure 5.7 shows the waveform of the structural VHDL simulation of the 4-phase while loop control circuit shown in Figure 5.6.

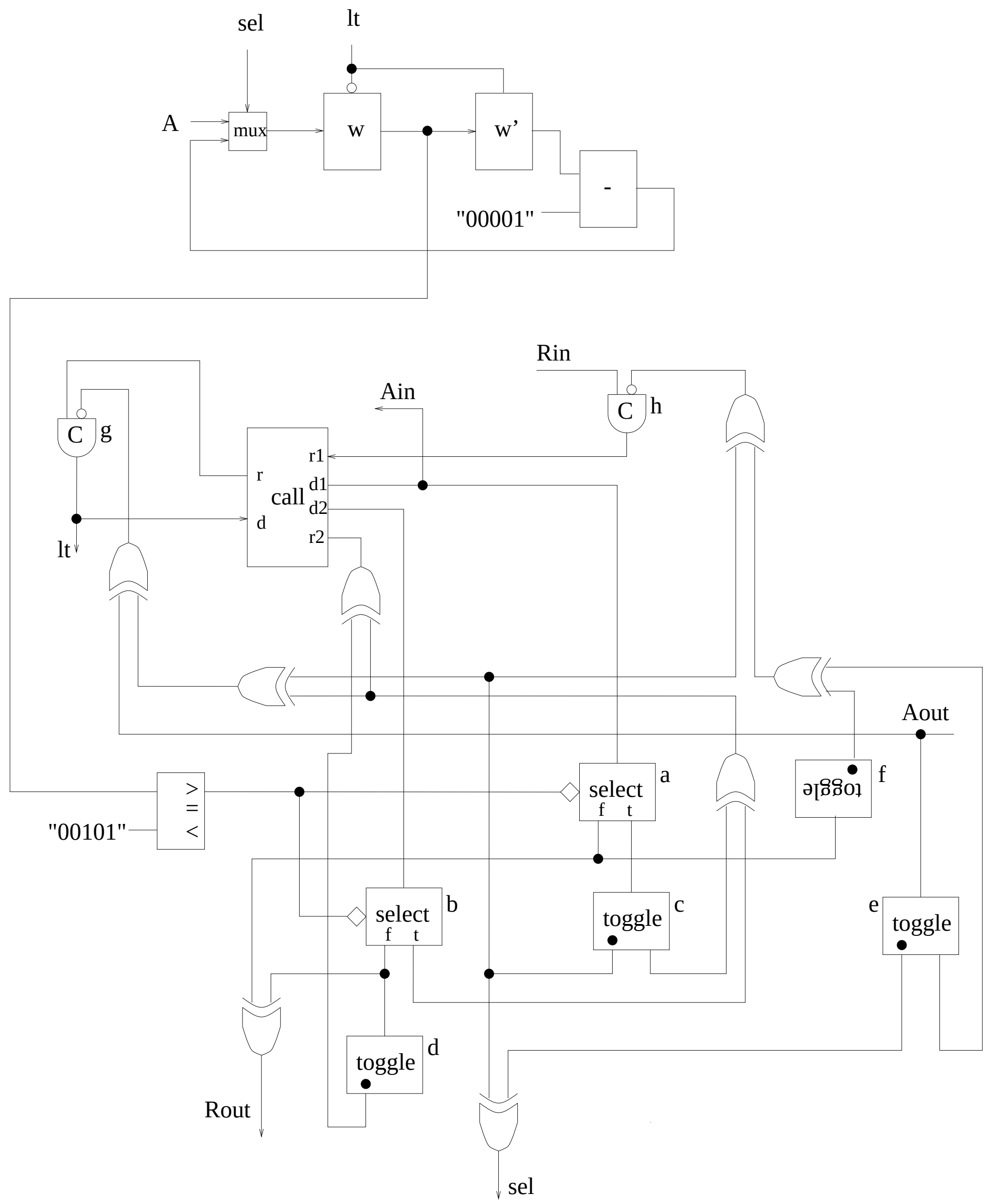


Figure 5.4: A 4-phase while loop control circuit(1)

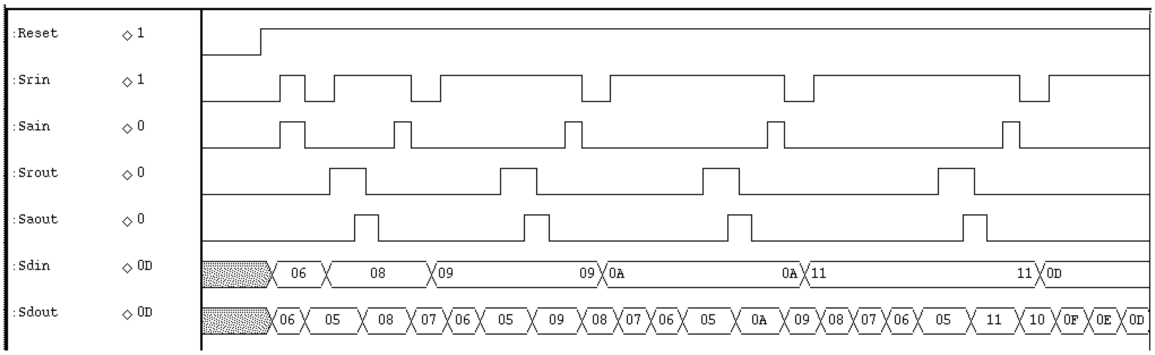


Figure 5.5: The waveform of the simulation of the 4-phase while loop control circuit of Figure 5.4

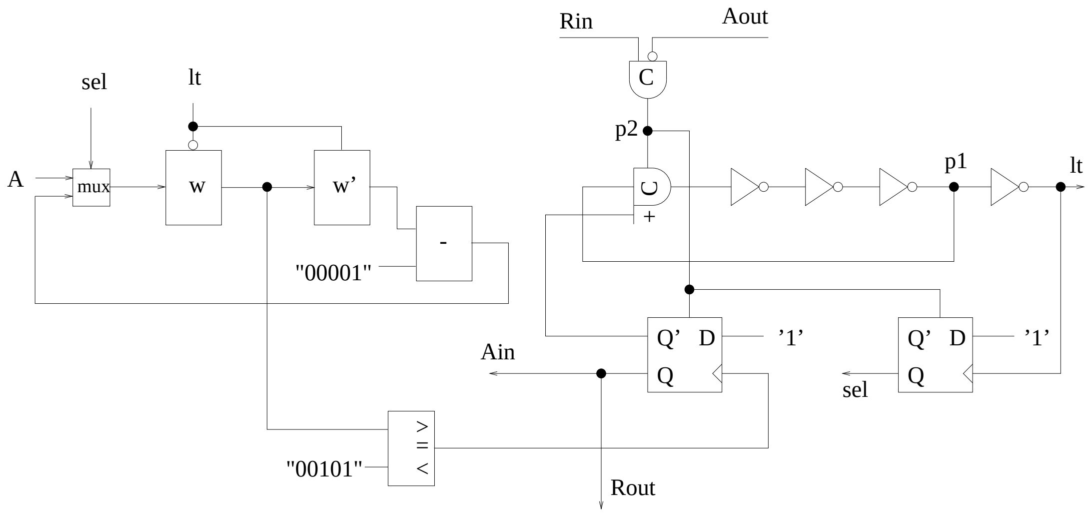


Figure 5.6: A 4-phase loop control circuit(2)

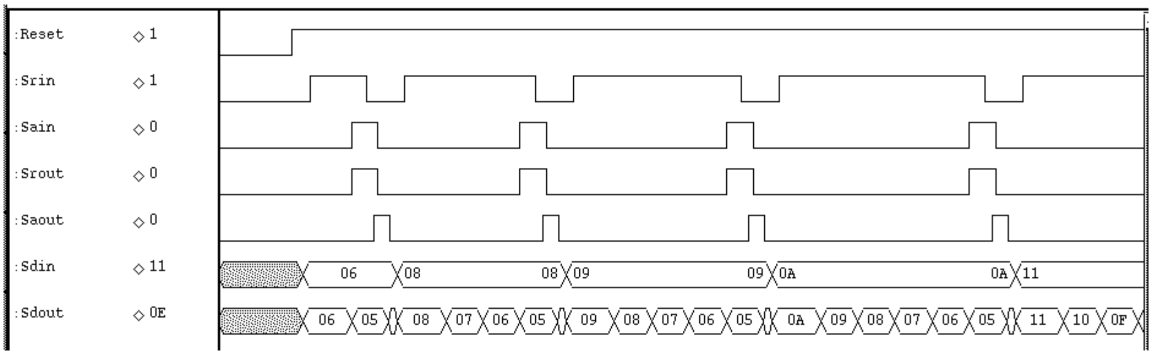


Figure 5.7: The waveform of the simulation of the 4-phase while loop control circuit of Figure 5.6

Figure 5.8 shows a 4-phase loop circuit which can perform the following loop operation. Figure 5.9 shows the waveform of the structural VHDL simulation of the circuit shown in Figure 5.8.

w = A;
while (w > ”00101”) loop
    s = s + ”00001”;
    w = w − ”00001”;
end loop;

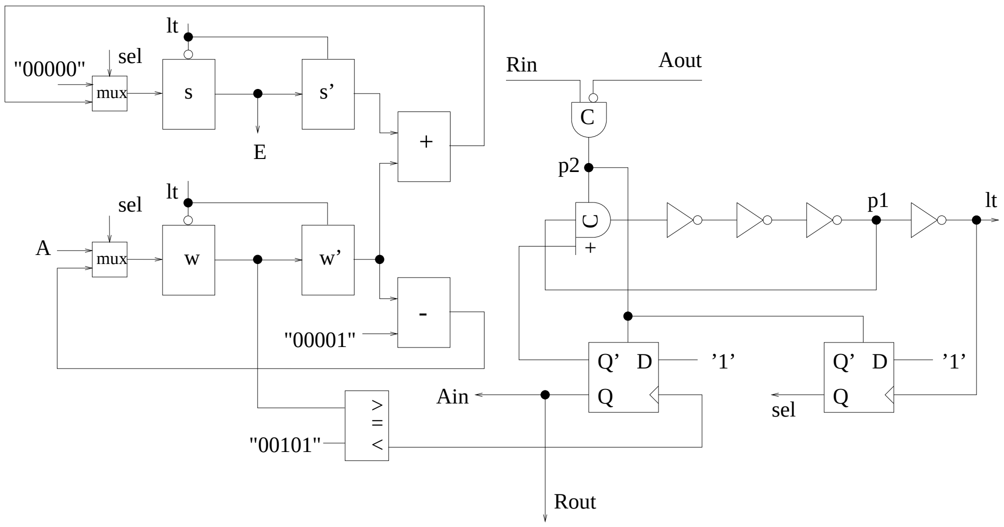


Figure 5.8: A 4-phase loop example

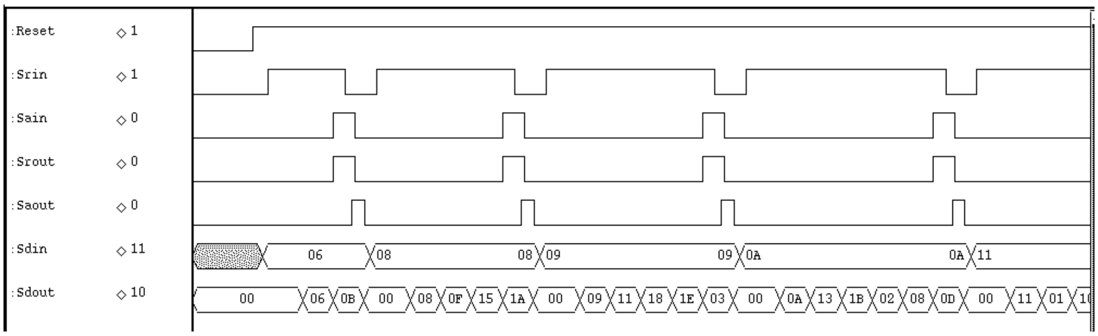


Figure 5.9: The waveform of the simulation of a 4-phase while loop control circuit

Two identical test pattern generators send data and handshaking signals to both control circuits shown in Figures 5.4 and 5.6. There are also two identical receivers connected to the output of both circuits. Figure 5.10 shows that the circuit of Figure 5.4 finished the simulation first.

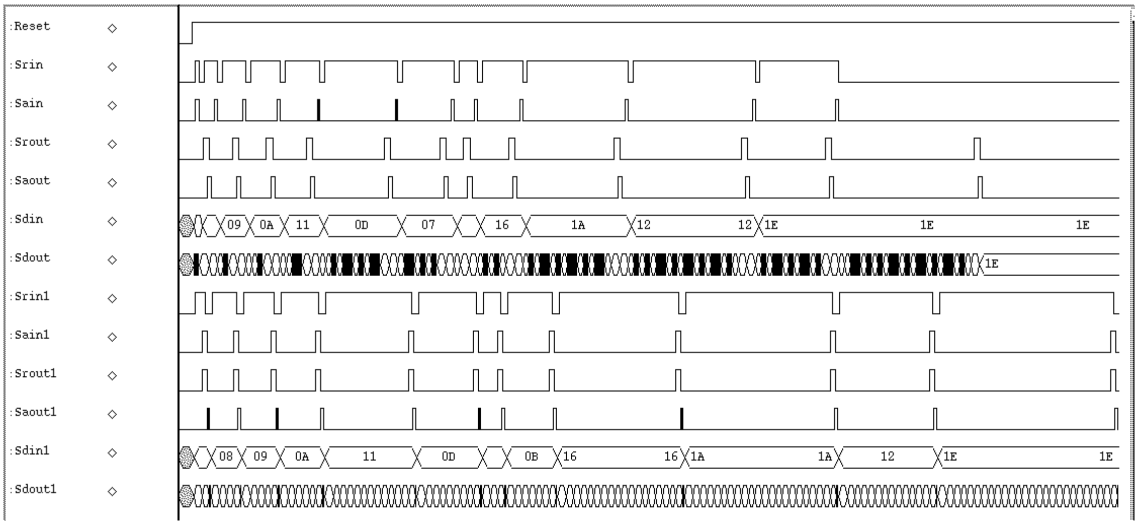


Figure 5.10: The waveform of the simulation of a comparison of two 4-phase loop control circuits

5.2.3 A while loop stage connects to another stage

The behavioural VHDL description shown in Figure 5.11 was partitioned into 2 stages by the synthesizer. The first stage contains the while loop operation. The second stage reads the fra_h and exp_h of the first stage and stores them into the fra_rm and exp_rm. The output, MO(0) connects to the fra_rm(0) and the outputs, MO(1) to (8) connect to the fra_rm(3) to (10). The 4-phase output of the synthesizer is shown in Appendix D.2.1. The 2-phase output of the synthesizer is shown in Appendix D.2.2. Figure 5.12 shows the VHDL simulation waveform of the whole circuit shown in D.2.2.1.

```
ENTITY circuit2 IS
   PORT ( RIN, AOUT : IN  MVL;
          AIN, ROUT : OUT  MVL;
          EA : IN MVL_VECTOR ( 0 TO 4 );
          MA : IN MVL_VECTOR ( 0 TO 14 );
          OP : IN MVL_VECTOR ( 1 DOWNTO 0 );
          EO : OUT MVL_VECTOR ( 0 TO 4 );
          MO : OUT MVL_VECTOR ( 0 TO 8 );
          RESET : IN MVL );
END circuit2;
architecture BEHAVIOR of circuit2 is
begin
  U1: process
      variable sign_c, sign_d, sign_s, comp : MVL;
      variable EXP_H, EXP_R, EXP_RM : MVL_VECTOR ( 0 TO 4 );
      variable FRA_H, FRA_R, FRA_RR, FRA_RM : MVL_VECTOR ( 0 TO 14 );
      begin
          fra_h := MA;      exp_h := EA;
          while ( fra_h = "000000000000001" ) loop
               fra_h := sh_left( fra_h, 1 );  exp_h := exp_h;
          end loop;
          fra_rm := fra_h;      exp_rm := exp_h;
          MO(0) <= fra_rm(0);   MO(1 to 8) <= fra_rm(3 to 10);
          EO <= exp_rm;
      end process U1;
end BEHAVIOR;
```

Figure 5.11: An example shows that a while loop stage connects to another stage.

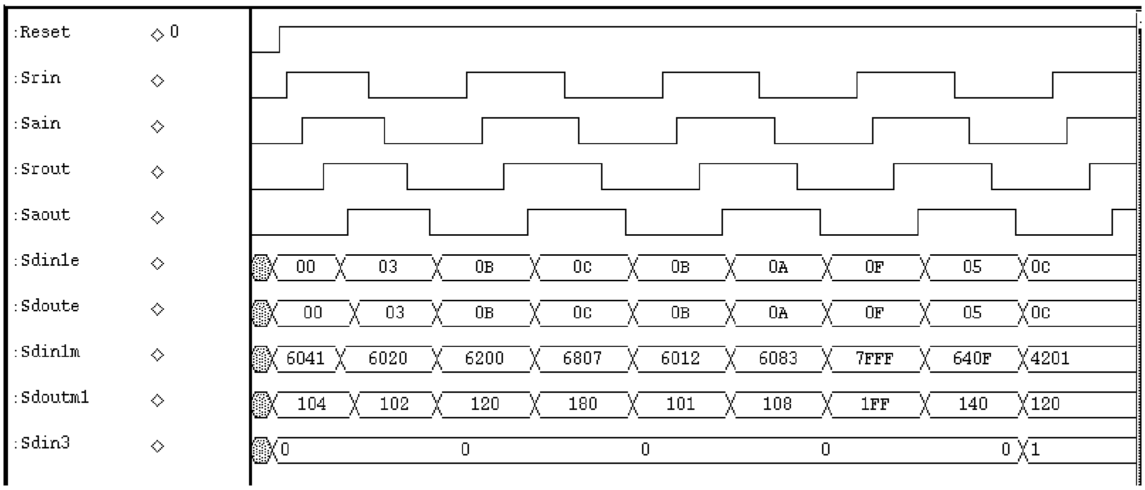


Figure 5.12: The waveform of the simulation of a while loop

5.3 A floating point Adder/subtractor

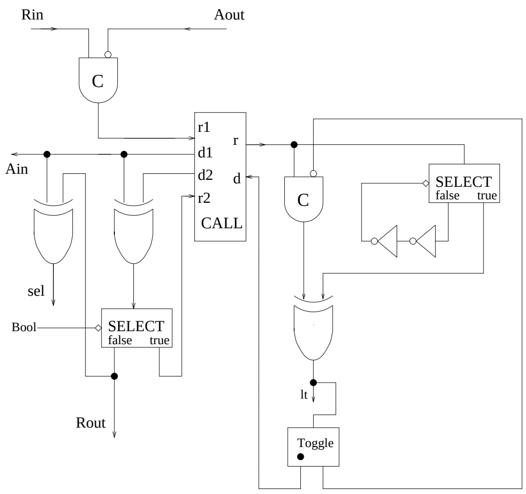


Figure 5.13: The 2-phase while loop control circuit

A test example of the behavioural description shown in Appendix E is a floating point Adder/subtractor with a 5-bit exponent a sign bit and a 8-bit mantissa. The external port definitions are RIN, AIN, ROUT, AOUT, RESET, EA, EB, MA, MB, OP, EO, MO. RIN, AIN, ROUT, and AOUT are two-phase handshaking control signals. EA, EB, MA, and MB are the exponents and mantissaes of two number respectively. The behavioural description was partitioned into 14 stages by the synthesizer. The 2-phase and 4-phase while loop control circuits shown in Figures 5.13 and 5.14 are used to implement the while circuits in the 14-bit floating point adder/subtractors. The top level of the synthesized circuits is shown in Figure 5.15.

As shown in the behavioural description in Appendix E the the *comp* is '$1'$ if $QEA >= QEB$. Otherwise, it is '$0'$. The fra_c is set to QMA if the *comp* is equal to '$1'$. Otherwise, it is set to QMB. Similarly, the fra_d is set to QMB if the *comp* is equal to '$1'$. Otherwise, it is set to QMB. Three computations, $fra\_c + fra_d$, $fra\_c - fra_d$ and $fra\_d - fra_c$ can be executed at the same time and they are denoted by fra_s1, fra_s2 and fra_s3. Table 5.1 shows how the sign bit and the sum/difference are obtained.

Table 5.1: The relationship between the sign bits and the sum/difference

OP OP	comp	sign bit of fra_c	sign bit of fra_d	carry of $fra\_c - fra\_d$	sign bit of sum/difference	sum/ difference
00	0	0	0		+	fra_s1
00	0	0	1		+	fra_s2
00	0	1	0		−	fra_s2
00	0	1	1		−	fra_s1
00	1	0	0		+	fra_s1
00	1	0	1	0	+	fra_s2
00	1	0	1	1	−	fra_s3
00	1	1	0	0	−	fra_s2
00	1	1	0	1	+	fra_s3
00	1	1	1		−	fra_s1
01	0	0	0		−	fra_s2
01	0	0	1		−	fra_s1
01	0	1	0		+	fra_s1
01	0	1	1		+	fra_s2
01	1	0	0	0	+	fra_s2
01	1	0	0	1	−	fra_s3
01	1	0	1		+	fra_s1
01	1	1	0		−	fra_s1
01	1	1	1	0	−	fra_s2
01	1	1	1	1	+	fra_s3

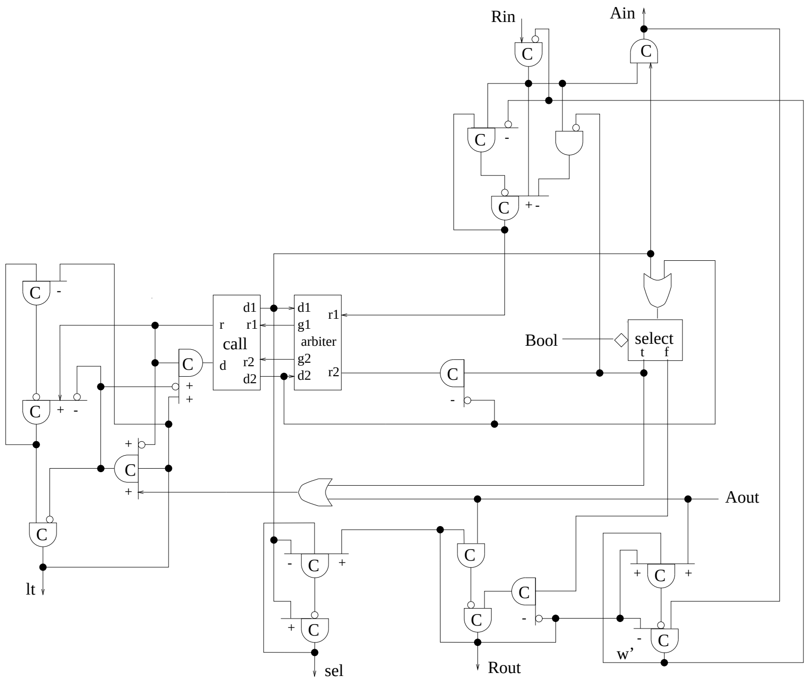


Figure 5.14: The 4-phase while loop control circuit

Different control circuits were used to implement the circuits. The two-phase control circuit was introduced in Section 2.5. The four-phase control circuits were described in Section 4.1. Both synthesized 2-phase and 4-phase 14-bit floating point adder/subtractors were simulated using the Leapfrog VHDL simulator and the Powermill correctively. The following data are used to the test the synthesized circuits.

Input					Output		Input					Output	
OP	EA	MA	EB	MB	EO	MO	OP	EA	MA	EB	MB	EO	MO
00	00	090	01	080	01	0C8	01	06	080	07	0A0	06	1C0
00	01	080	03	0A0	04	000	00	05	094	05	1C0	02	160
00	05	080	07	0C0	08	010	00	05	0C8	05	182	03	018
00	06	080	04	0E0	06	0F8	00	05	1C2	05	081	03	104
00	05	080	05	0A0	06	090	00	05	1C4	05	0C0	02	0E0
00	05	080	07	0D0	08	018	00	05	1F0	05	180	06	1B8
00	07	080	05	090	07	0E4	00	05	186	05	198	06	18F
00	02	080	06	090	06	0A8							

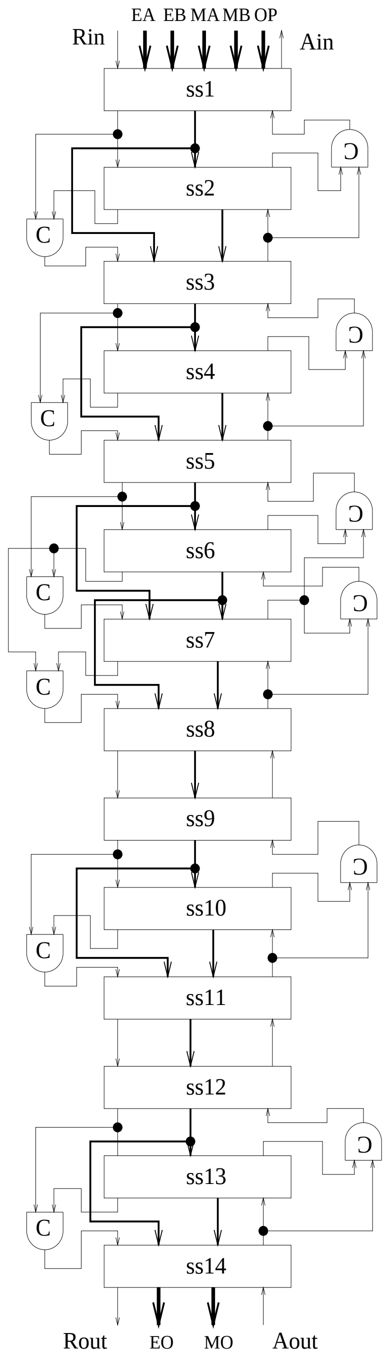


Figure 5.15: The floating point adder/subtractor

Table 5.2: The delays of the data path and the control path of different implementations

Stage Name	Data path delay (ns)	2-phase control path delay (ns)	4-phase simple control path delay (ns)	4-phase semi-control path delay (ns)	4-phase fully control path delay (ns)	4-phase long control path delay (ns)
stg1	15.2	9.8	5.6	4.0	6.4	8.0
stg2	21.0	9.8	5.6	4.0	6.4	8.0
stg3	5.8	9.8	5.6	4.0	6.4	8.0
stg4	8.2	21.6	26.6	26.6	26.6	26.6
stg5	0.0	9.8	5.6	4.0	6.4	8.0
stg6	58.8	9.8	5.6	4.0	6.4	8.0
stg7	23.2	9.8	5.6	4.0	6.4	8.0
stg8	0	9.8	5.6	4.0	6.4	8.0
stg9	19.8	9.8	5.6	4.0	6.4	8.0
stg10	18.8	9.8	5.6	4.0	6.4	8.0
stg11	5.8	21.6	26.6	26.6	26.6	26.6
stg12	4.8	9.8	5.6	4.0	6.4	8.0
stg13	0	9.8	5.6	4.0	6.4	8.0
stg14	19.8	9.8	5.6	4.0	6.4	8.0
Expected time for data output(ns)		268.8	258.6	252.0	261.8	269.8

Table 5.2 shows the data path and control path delays of different implementations of the floating point adder/subtractor. The bigger delay is chosen for each stage, and the sum of the 14 delays gives the expected time the data can be sent to the output. However, there are some additional delays on the interconnections between stages and some time for waiting to proceed to the next stage.

Table 5.3 shows the number of transistors and the run time of the 150 sets of data, the throughput, the latency and the energy of the synthesized floating point adder/subtractor.

Table 5.3: The performance of the synthesized floating point adder/subtractor

Circuit Name	Transistors (piece)	Run Time(μs)	Throughput (MHz)	Latency (ns)	Energy (fj)
2-phase	20358	26.88	5.68	874.5	156.12
4-p Simple	20190	34.36	4.41	591.8	168.36
4-p Semi	20646	36.94	4.11	730.8	183.77
4-p Fully	20746	28.56	5.34	739.4	168.69
4-p Long	21038	31.40	4.85	747.8	163.16

5.4 While loops contain handshaking signals

In Chapter 3 the while loop control circuit was introduced in Figure 3.21. However, the circuit works by holding the initial data when the circuit receives an **rin** signal and the circuit sends out an **rout** signal when the loop termination condition becomes true to inform the successor that data are available. There is no request signal inside the while loop. In some cases it is necessary that the while loop contains request signals. The subsequent subsections will concentrate on this issue. Four interesting cases are discussed here. They are as follows:

- There is an **Rin** event before the loop statement.
- There is an **Rout** event before the loop statement.
- There are both **Rin** and **Rout** events before the loop statement.
- There are no **Rin** and **Rout** events before the loop statement.

5.4.1 An Rin event before the loop statement

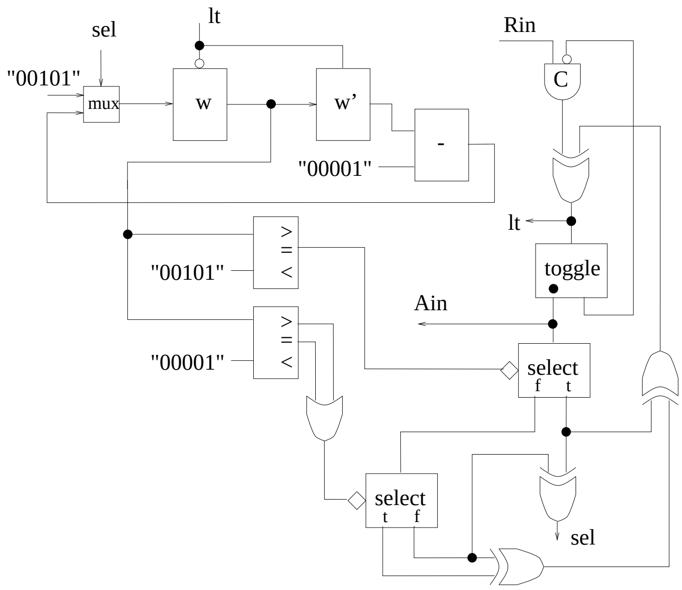


Figure 5.16: 2-phase loop with request input

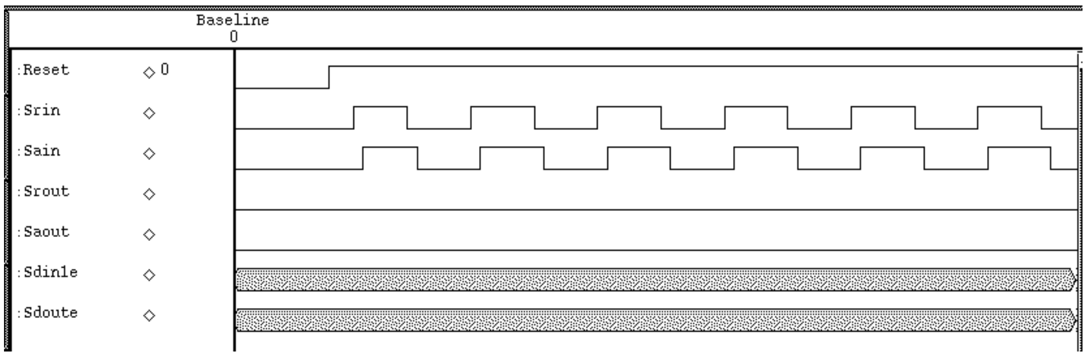


Figure 5.17: The waveform of the simulation of a loop with request input(1)

As shown in Figure 5.16 the signal, **sel**, will start from a logic '0'. The initial value, "00101" will pass through the multiplexer to the active low transparent latch, *w*. The signal, **lt**, will go to logic '1' to enable the latch, *w*, holding the data when the first Rin event arrives. When the latch, *w*, holds "00101", the 'equal' output of the upper comparator will become true. The event will be delivered to the output labelled 't' (true) of the Select module whose Bool signal is connected to the 'equal' output of the upper comparator. This will enable the signal, **sel**, to go to logic '1' as well as enabling the latch, w', to hold data and the latch, w, to become transparent. The input of the latch w is connected to the output of the subtractor if the signal **sel** is at logic '1'. When the next Rin event arrives the latch, w will hold "00100". Because the 'equal' output of the upper comparator becomes false, the event will be delivered to the output labelled 'f' (false) of the Select module and then to the output labelled 't' (true) of the other Select module due to the output of the OR gate being true. The event will enable the latch w to become transparent again. The latch w holds "00011" when the third Rin event arrives. The latch w holds "00010" when the fourth Rin event arrives. The latch w holds "00000" when the sixth Rin event arrives. Because the output of the OR gate is false, it will enable the event through the false outputs of the two Select modules and turn the signal **sel** to logic '0'. This means that the input of the latch w is connected to the initial value, "00101", again. The latch w will hold the initial value when the seventh Rin event arrives. Therefore, The bus connected to the output of the latch w has the values "00101", "00100", "00011", "00010", "00001", "00000", "00101", "00101", ... etc sequentially. The VHDL simulation waveform is shown in Figure 5.17.

A loop statement requires three control circuits. The first is for setting the initial values of each variable which is used inside the loop statement. The second is for the loop operation and the assignments inside the loop. The third part is for the assignments after the loop to pass the results to other variables. The control circuit of the corresponding latches for each variable of the initial assignments before the while loop statement can be put between the output labelled 't' of the right-top Select module and the inputs of the two XOR gates. The control circuit of the corresponding latches for each variable of the assignments inside the while loop statement can be put between the output labelled 'f' of the right-top Select module and the input of the left-bottom Select module. The control circuit of the corresponding latches for each variable of the result assignments after the while loop statement can be put between the output labelled 'f' of the left-bottom Select module and the inputs of the two XOR gates. The signal, sel remains logic '1' during the period between the event appeared on the output labelled 't' of the right-top Select module and the event appeared on the output labelled 'f' of the left-bottom Select module.

5.4.2 An Rout event before the loop statement

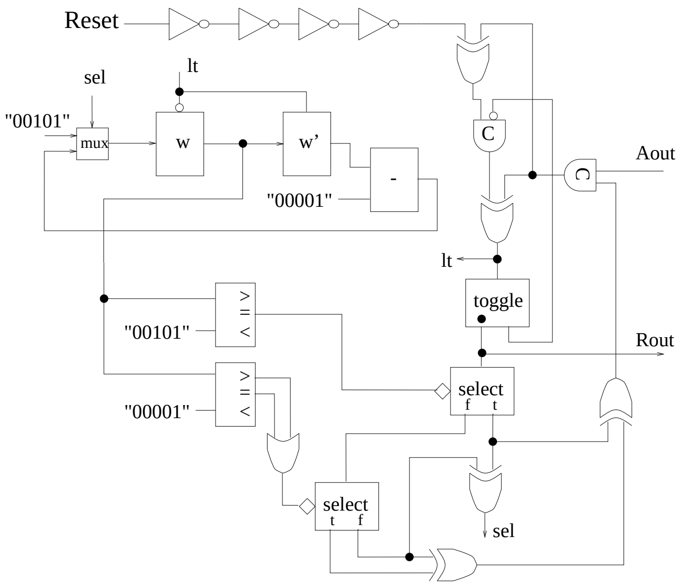


Figure 5.18: 2-phase loop with req out

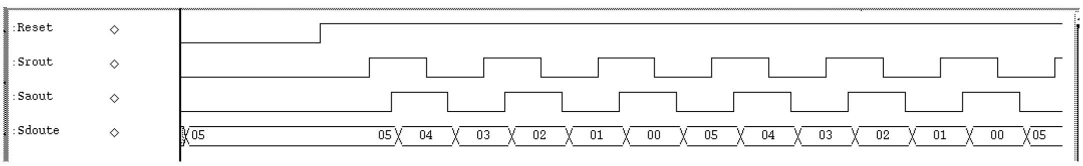


Figure 5.19: The waveform of the simulation of a loop with request(2)

As shown in Figure 5.18 the signal **sel** will start from a logic '0'. The initial value, "00101", will through the multiplexer to the active low transparent latch, w. The signal **lt** will go to logic '1' to enable the latch w holding the data when the Reset signal goes from logic '0' to logic '1'. Then an Rout signal is sent out from the dotted output of the Toggle to inform the successor stage that the output data is available. The dotted output of the Toggle module is also connected to the input of the Select module. Due to the latch w holding the "00101" the 'equal' output of the upper comparator will become true. The event will be delivered to the output labelled 't' (true) of the Select module whose Bool signal is connected to the 'equal' output of the upper comparator. This will enable the signal **sel** to go to logic '1'. It will also wait for an Aout signal from the successor stage to enable the latch w' to hold data and the latch w to become transparent. The input of the latch w is connected to the output of the subtractor if the signal **sel** is at logic '1'. The event on the output of the right hand side of the C gate will also enable the latch w to hold "00100". Because the 'equal' output of the upper comparator becomes false, the Rout event will also be delivered to the output labelled 'f' (false) of the Select module and then to the output labelled 't' (true) of the other Select module because the output of the OR gate is true. The event will wait for an Aout event to enable the latch w to become transparent again as well as to enable the subsequent Rout events. The latch w holds "00011" after the second Aout event and the second Rout event meet together at the C gate. The latch w holds "00010" after the third Aout event and the third Rout event meet together at the C gate. The latch w holds "00000" after the fifth Aout event and the fifth Rout event meet together at the C gate. Because the output of the OR gate is false, it will enable the Rout event through the false outputs of the two Select modules and set the signal **sel** to logic '0'. This means that the input of the latch w is connected to the initial value "00101" again. The latch w will hold the initial value after the sixth Aout event and the sixth Rout event meet together. Therefore, "00101", "00100", "00011", "00010", "00001", "00000", "00101", "00101", ... etc. are shown on the bus connected to the output of the latch, w sequentially. The VHDL simulation waveform is shown in Figure 5.19.

5.4.3 Both Rin and Rout events before the loop statement

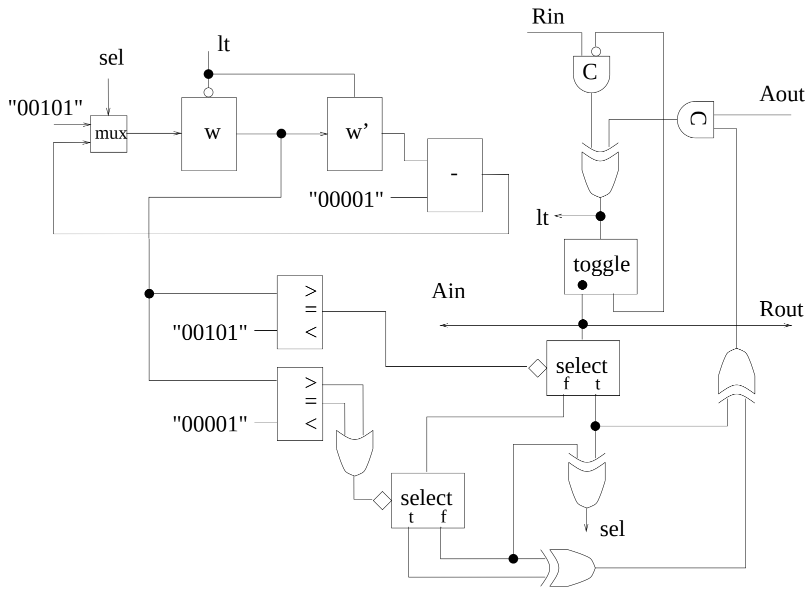


Figure 5.20: 2-phase loop with req in and req out

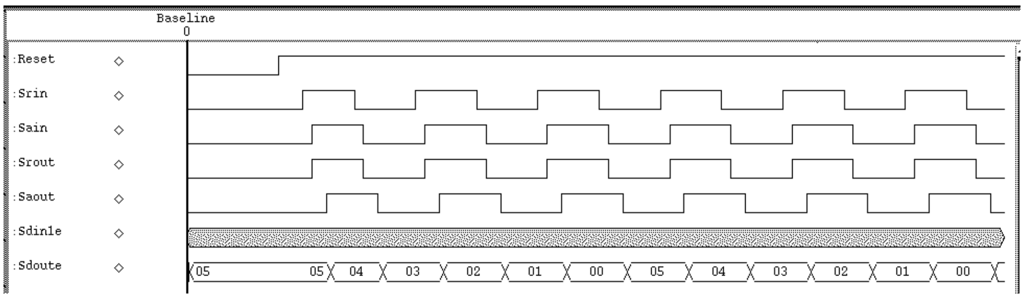


Figure 5.21: The waveform of the simulation of a loop with request(3)

As shown in Figure 5.20 the signal **sel** will start from a logic '0'. The initial value, "00101" will pass through the multiplexer to the active low transparent latch w. The signal **lt** will go to logic '1' to enable the latch w to hold the data when a Rin event is received. Then a Rout event is sent out from the dotted output of the Toggle to inform the successor stage that the output data is available. The dotted output of the Toggle module is also connected to the input of the Select module. Due to the latch w holding "00101" the 'equal' output of the upper comparator will become true. The event will be delivered to the output labelled 't' (true) of the Select module whose Bool signal is connected to the 'equal' output of the upper comparator. This will enable the signal **sel** to go to logic '1'. It will also wait for an Aout signal from the successor stage to enable the latch w' to hold data and the latch w to become transparent. The input of the latch w is connected to the output of the subtractor if the signal **sel** is at logic '1'. Therefore, the next Rin event enables the latch w to hold "00100". Because the 'equal' output of the upper comparator becomes false, the Rout event will also be delivered to the output labelled 'f' (false) of the Select module and then to the output labelled 't' (true) of the other Select module because the output of the OR gate is true. The event will wait for an Aout event to enable the latch w to become transparent again. The latch w holds "00011" after the second Aout event and the second Rout event meet together at the C gate and the third Rin event is received. The latch w holds "00010" after the third Aout event and the third Rout event meet together at the C gate and the fourth Rin event is received. The latch w holds "00000" after the fifth Aout event and the fifth Rout event meet together at the C gate and the sixth Rin event is received. Because the output of the OR gate is false, it will enable the Rout event through the false outputs of the two Select modules and set the signal **sel** to logic '0'. This means that the input of the latch w is connected to the initial value, "00101" again. The latch w will hold the initial value after the sixth Aout event and the sixth Rout event meet together and the seventh Rin event is received. Therefore, "00101", "00100", "00011", "00010", "00001", "00000", "00101", "00101", ... etc. are shown on the bus connected to the output of the latch w sequentially. The VHDL simulation waveform is shown in Figure 5.21.

5.4.4 No Rin and Rout events before the loop statement

As shown in Figure 5.22 the signal **sel** will start from a logic '0'. The initial value, "00101", will pass through the multiplexer to the active low transparent

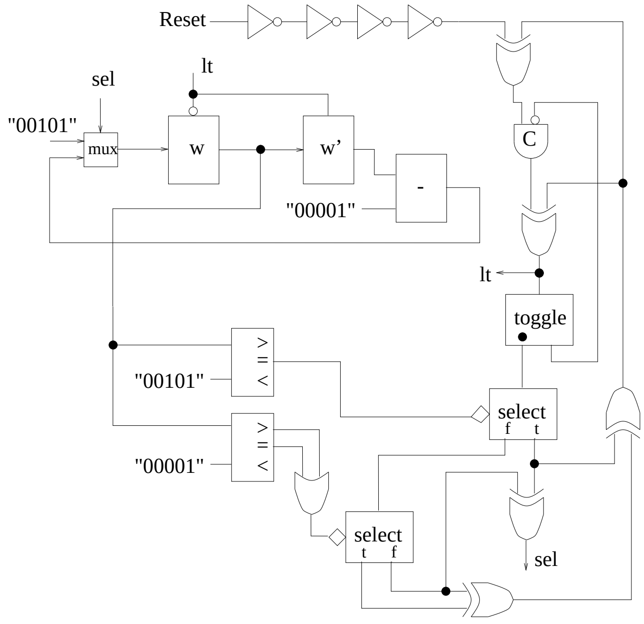


Figure 5.22: 2-phase loop without req

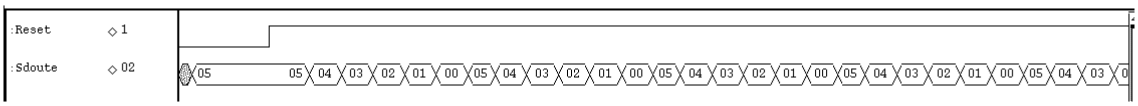


Figure 5.23: The waveform of the simulation of a loop without request

latch w. The signal **lt** will go to logic ‘1’ to enable the latch w holding the data when the Reset signal goes from logic ‘0’ to logic ‘1’. The dotted output of the Toggle module is connected to the input of the Select module. Due to the latch w holding “00101” the ‘equal’ output of the upper comparator will become true. The event will be delivered to the output labelled ‘t’ (true) of the Select module whose Bool signal is connected to the ‘equal’ output of the upper comparator. This will enable the signal **sel** to go to logic ‘1’. It will also enable the latch w' to hold data and the latch w to become transparent. The input of the latch w is connected to the output of the subtractor if the signal **sel** is at logic ‘1’. The event will also enable the latch w to hold “00100”. Because the ‘equal’ output of the upper comparator becomes false, the Rout event will also be delivered to the output labelled ‘f’ (false) of the Select module and then to the output labelled ‘t’ (true) of the other Select module because the output of the OR gate is true. The event will enable the latch w to become transparent again as well as enable the subsequent events. The latch w holds "00011" after the third event throughs the top XOR gate and the C gate. The latch w holds "00010" after the the fourth event throughs the top XOR gate and the C gate. The latch w holds "00000" after the sixth event through the top XOR gate and the C gate. Because the output of the OR gate is false, it will enable the Rout event through the false outputs of the two Select modules and set the signal **sel** to logic '0'. This means that the input of the latch w is connected to the initial value "00101" again. The latch w will hold the initial value after the seventh event throughs the top XOR gate and the C gate. Therefore, "00101", "00100", "00011", "00010", "00001", "00000", "00101", "00101", ... etc. are shown on the bus connected to the output of the latch w sequentially. The VHDL simulation waveform is shown in Figure 5.23.

5.5 The blocksorter

The following description shows a simple handshaking control with a simple computation. When the Reset signal is logic '0' it is in a reset loop to clear the AIN and ROUT signals. While the Reset signal is logic '1' the operating cycle is (1) waiting for an Rin transition; (2) computing; (3) sending AIN and Rout signals; (4) waiting for an Aout transition.

```
while ( RESET = '0' ) loop
   Qrin := '0'; Qaout := '0'; Sro := '0'; ROUT <= sro; AIN <='0';
end loop;
while ( Rin = Qrin ) loop end loop; Qrin := Rin;
if a > "00101" then b := a - "00010"; else b := a + "00011"; end if;
E <= b; AIN <= RIN; sro := not sro; Rout <= sro;
while ( Aout = Qaout ) loop end loop; Qaout := Aout;
```

A different synthesis procedure is applied to generate the circuits. Therefore, checking whether the loop statement contains handshaking controls is required before the circuit synthesis. In this case only the computation part of the description is required to convert into circuits and then a suitable control circuit is added. In the blocksorter [Berk93] example a computation loop contains handshaking controls.

A two-phase blocksorter can be described as follows:

```
while ( RESET = '0' ) loop
   Qrin := '0'; Qaout := '0'; Sro := '0'; ROUT <= sro; AIN <='0';
end loop;
while ( Rin = Qrin ) loop end loop; Qrin := Rin;
y := a; w := "00101";
AIN <= RIN;
while ( w >= "00001" ) loop
   while ( Rin = Qrin ) loop end loop; Qrin := Rin;
   x := a;  AIN <= RIN;
   if x > y then b := y; else b := x; end if; E <= b;
   sro := not sro; Rout <= sro;
   if x > y then y := x; else y := y; end if;
   while ( Aout = Qaout ) loop end loop; Qaout := Aout;
   w := w - "00001";
end loop;
b := y; E <= b;
sro := not sro; Rout <= sro;
while ( Aout = Qaout ) loop end loop; Qaout := Aout;
```

To get a correct simulation the description contains some handshaking control loop and initial assignments. However, this causes the synthesis work to be more difficult. If the reset loop, the handshaking control loop and initial assignments are put into procedures there is no effect on the simulation. But it is easier to remove these procedures and to convert the description into the circuits. The previous blocksorter description can be re-written as follows:

```
ENTITY blocksort1 IS
  PORT ( RIN, AOUT : IN  MVL; AIN, ROUT : OUT  MVL;
       A : IN MVL_VECTOR ( 4 DOWNTO 0 ); E : OUT MVL_VECTOR ( 4 DOWNTO 0 );
       RESET : IN MVL );
END blocksort1;
architecture BEHAVIORAL of blocksort1 is
begin
 U1: process
    variable Qrin, Qaout, sro : MVL;
    variable X, Y, B, W : MVL_VECTOR ( 4 DOWNTO 0 );
 PROCEDURE  req_in (Qrin: INOUT MVL) IS
   BEGIN while ( Rin = Qrin ) loop  wait for 1 ns; end loop; Qrin := Rin; END req_in;
 PROCEDURE  req_out (sro: INOUT MVL) IS
   BEGIN sro := not sro;  Rout <= sro; END req_out;
 PROCEDURE  ack_out (Qaout: INOUT MVL) IS
   BEGIN while ( Aout = Qaout ) loop  wait for 1 ns; end loop; Qaout := Aout; END ack_out;
 PROCEDURE  ack_in IS
   BEGIN AIN <= RIN; END ack_in;
 PROCEDURE  reset_q (Qrin, Qaout, sro: INOUT MVL) IS
   BEGIN while ( RESET = '0' ) loop wait for 1 ns; Qrin := '0'; Qaout := '0';
      sro := '0'; wait for 1 ns; ROUT <= sro; AIN <='0'; end loop; END reset_q;
 begin
   wait for 1 ns; reset_q(Qrin, Qaout, sro); wait for 1 ns;
   req_in(Qrin); y := a; w := "10100";  -- w := "00101";  wait for 1 ns;  ack_in;
   while ( w >= "10000" ) loop   -- while ( w >= "00001" ) loop
     wait for 1 ns; req_in(Qrin); x := a; wait for 1 ns; ack_in;
     if x > y then b := y; else b := x; end if;
     E <= b; wait for 1 ns; req_out(sro); wait for 1 ns;
     if x > y then y := x; else y := y; end if;
     ack_out(Qaout);  w := w - "10000";  -- w := w - "00001";
   end loop;
   wait for 1 ns; b := y; E <= b; wait for 1 ns; req_out(sro); ack_out(Qaout);
 end process U1;
end BEHAVIORAL;
```

We may count the number of appearances of the *req_in*, *ack_in*, *req_out* and *ack_out* signals in the description. A *req_in* and an *ack_in* appear before the while loop. A *req_in*, an *ack_in*, a *req_out* and an *ack_out* appear inside the while loop. A *req_out* and an *ack_out* appear after the while loop. The while loop contains four handshaking signals. In the previous simple while loop example the loop contains no handshaking signal. The handshaking signal number inside a while loop can be used to recognize for generating different while loop circuits. The handshaking signal numbers of the blocksorter description are shown as follows:

```
req_in = 1
        y := a; w := "00101";
ack_in = 1
        while ( w >= "00001" ) loop
req_in = 2
          x := a;
ack_in = 2
          if x > y then b := y; else b := x; end if;
          E <= b;
req_out = 1
          if x > y then y := x; else y := y; end if;
ack_out = 1
          w := w - "00001";
        end loop;
        b := y;
        E <= b;
req_out = 2
ack_out = 2
```

The synthesizer read the blocksorter behavioural description and produced the structural VHDL files of the circuit shown in Figure 5.24.

The synthesized 2-phase blocksorter was simulated using the test program shown in Appendix F.2 and the configuration program shown in Appendix F.1. The correct result was obtained. The simulation result is shown in Figure 5.25.

6 blocksorter stages connected together were also simulated using the test program shown in Appendix F.3 and the configuration program shown in Appendix F.4. The simulation result is shown in Figure 5.26.

A 4-phase blocksorter circuit is shown in Figure 5.27. **"Return to zero"** is required in the 4-phase handshaking protocol. The circuit enclosed in the dashed line and labelled $M3$ is used to ensure that the latches which are connected to the control signal **lt y** still hold data and that the signal connected to the input labelled **d** of the Call module becomes **logical '0'** when the output **r** of the Call module becomes **logical '0'**. The latches will be clear when the signal labelled **CX** becomes **logical '1'** and wait for holding the next data. The circuit enclosed in the dashed line and labelled $M1$ is used to ensure that the control signal connected to the input **r1** of the Arbiter module becomes **logical '0'** when

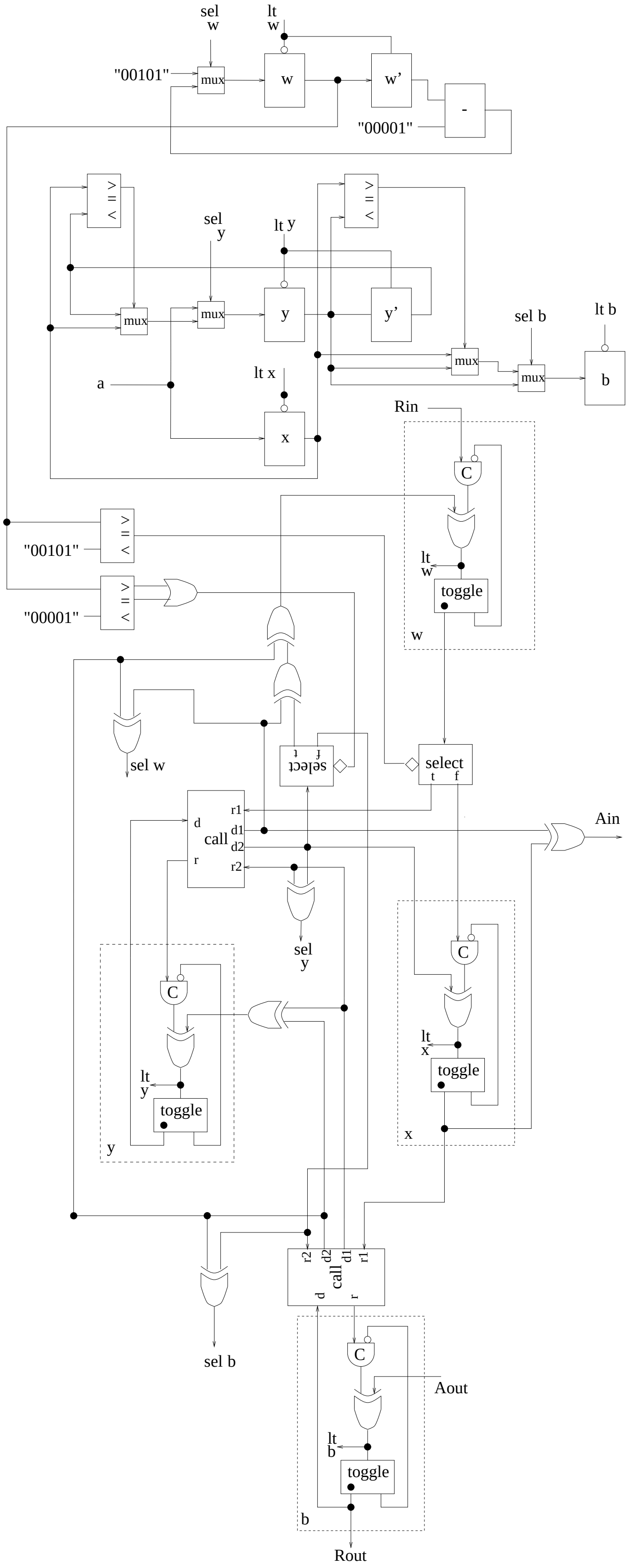


Figure 5.24: The synthesized 2-phase Blocksorter

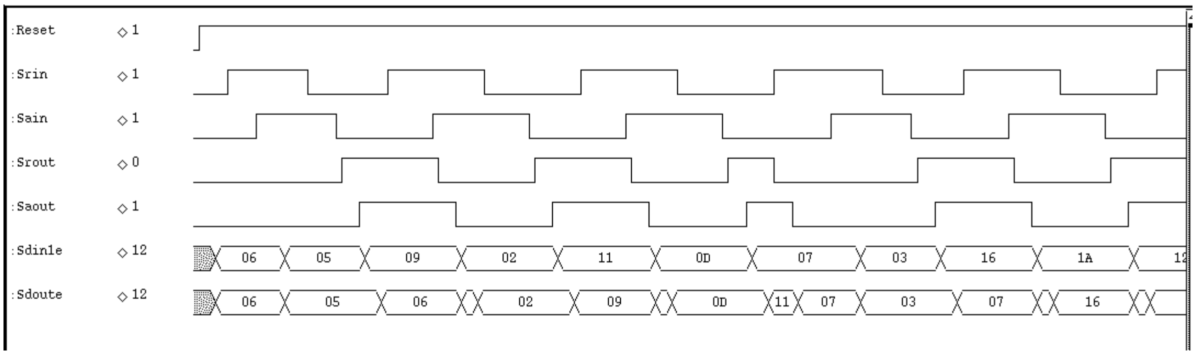


Figure 5.25: The waveform of the simulation of the synthesized blocksorter

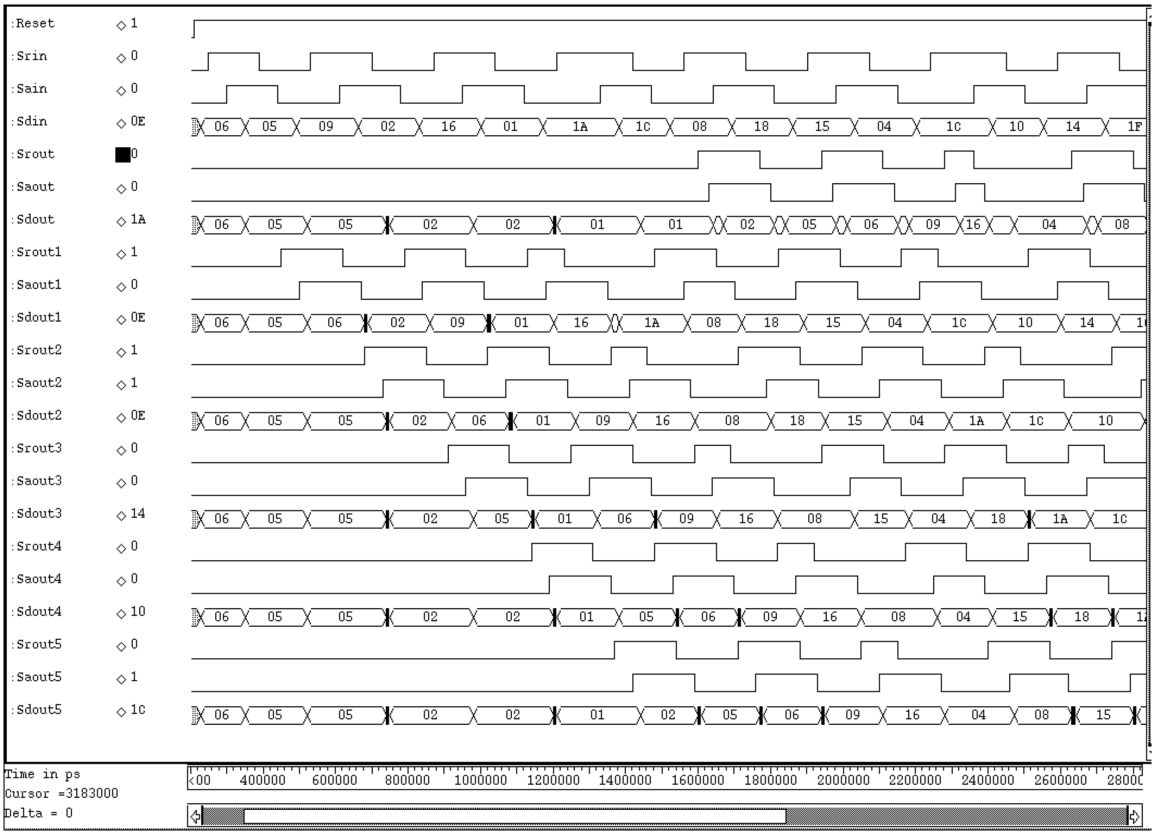


Figure 5.26: The waveform of the simulation of 6 synthesized blocksorters

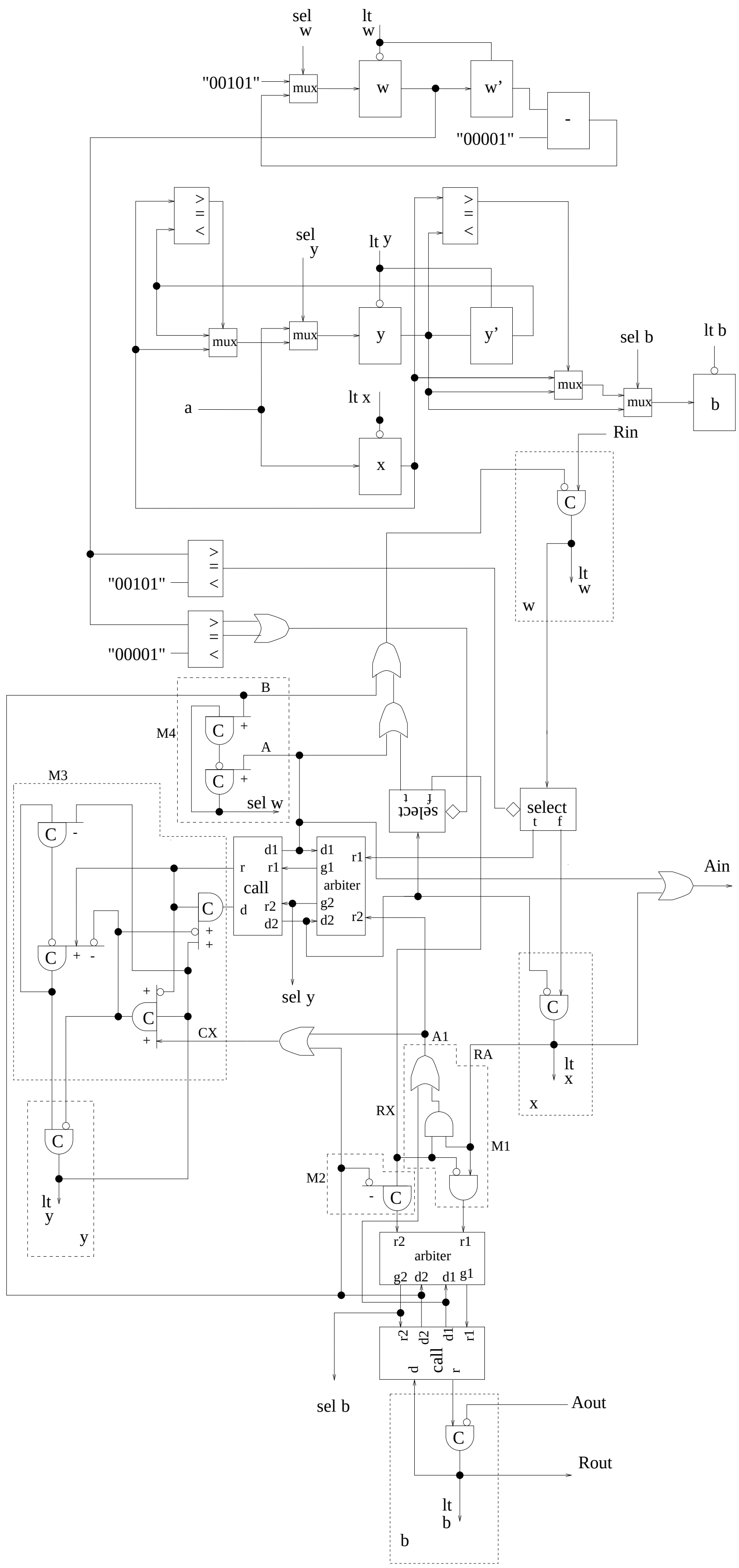


Figure 5.27: The synthesized 4-phase Blocksorter

the signal labelled **RX** becomes **logical '1'**. This ensures that the complete cycle signal of the input **d** of the Call module is sent to the output **d1** and then the request of **RX** can be sent into the Call module. The circuit also ensures that the signal labelled $A1$ stays **logical '1'** if both signals labelled RX and RA are **logical '1'** or the output **d1** of the Call module is **logical '1'**. When the data is held and the output **d2** of the Call module becomes **logical '1'** it ensures that $Rin-$ goes through and the signal RA becomes **logical '0'** as well as the signal labelled $A1$ becomes **logical '0'**. The circuit enclosed in the dashed line and labelled $M2$ is used to ensure that the request of the signal **RX** cab be held until the corresponding completion signal is received.

The circuits labelled $M1$ and $M3$ are required if the simple latch control circuit (see Figure 4.7) is applied. The circuits labelled $M1$, $M2$ and $M3$ are required if the semi-decoupled control circuit (see Figure 4.9) is applied. The circuits labelled $M2$ and bf $M3$ are required if the fully-decoupled control circuit (see Figure 4.10) and the long-hold control circuit (see Figure 4.11) are applied. The circuit enclosed in the dashed line and labelled $M4$ is to produce the select signal of the multiplexer at the front of the low active transparent latch **w**. The signal **sel w** goes **high** when the signal labelled **A** becomes **high** and it goes **low** when the signal labelled **B** becomes **high**.

Table 5.4 shows the number of the transistors and the run time of the 120 sets of data, the throughput, the latency and the energy of the synthesized blocksorter circuit. Table 5.5 shows the number of the transistors and the run time of the 120 sets of data, the throughput and the latency of the 6 synthesized blocksorter circuits were connected in series.

Table 5.4: The performance of the synthesized blocksorter

Circuit Name	Transistors (piece)	Run Time(μs)	Throughput (MHz)	Latency (ns)	Energy (fj)
2-phase	2878	15.53	7.80	261.3	10.78
4-p Simple	2596	19.74	6.19	228.0	12.13
4-p Semi	2766	22.33	5.41	264.4	14.23
4-p Fully	2782	19.29	6.27	220.0	14.72
4-p Long	2878	20.78	5.82	226.3	13.65

Table 5.5: The performance of the synthesized blocksorter

Circuit Name	Transistors (piece)	Run Time(μs)	Throughput (MHz)	Latency (ns)
2-phase	17268	16.70	7.71	1273.1
4-p Simple	15576	21.84	5.81	1328.1
4-p Semi	16596	25.0	5.09	1575.7
4-p Fully	16692	20.48	6.25	1353.2
4-p Long	17268	24.24	5.24	1534.9

5.6 Further work

A simple processor example, Stump [Ende98a], is shown in Appendix G. It is a 16-bit mini-risc processor which has 8 registers. The 4-bit condition code register, the instruction set and its 3 formats, the shift operations and the conditional branch instructions are also given in Appendix G. The behavioural VHDL description shown in Appendix G.2 can be used to describe the asynchronous operation of the processor. The following programs are used to test how the model works.

```
      LD R4,0020H              LOOP: LD R4,0020H
      LD R1,(0021H)                  LD R1,(0021H)
      LD R2,(0022H)                  LD R2,(0022H)
      ADD R3, R1, R2                 ADD R3, R1, R2
      LD (0023H), R3                 LD (0023H), R3
LOOP: BAL LOOP                       BAL LOOP

          (A)                            (B)
```

A test program which contains the machine code of the above program (A) and some initial memory data and the behavioural memory model is shown in Appendix G.3. The simulation result of the behavioural asynchronous Stump description is shown in Figure 5.28.

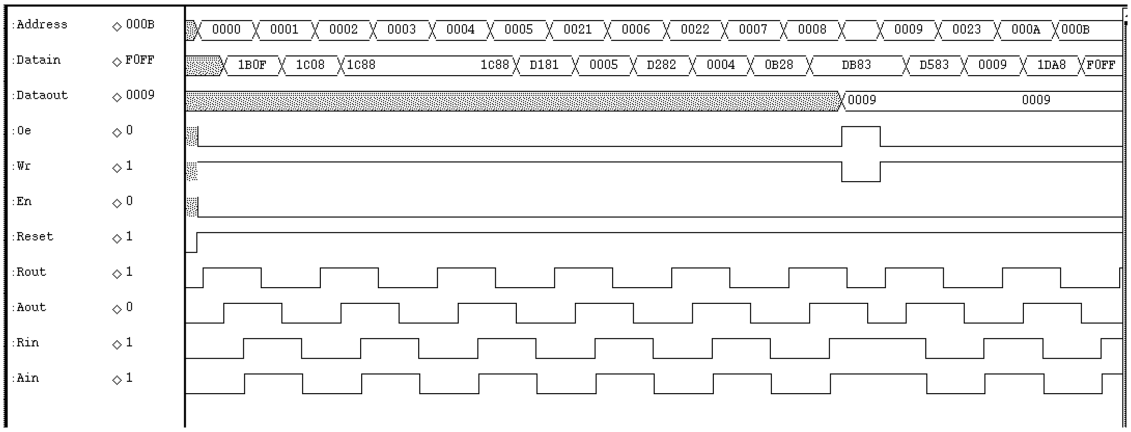


Figure 5.28: The waveform of the simulation of a behavioural 2-phase Stump processor

The synthesizer read the behavioural VHDL description shown in Appendix G.2 and generated two-phase and four-phase Stump processor circuits. The synthesized two-phase structural Stump processor is shown in Figure 5.29. When the test program shown in Appendix G.3 was used to simulate the synthesized two-phase structural Stump processor the correct result was obtained. Different four-phase latch control circuits shown in Figures 4.7, 4.9, 4.10 and 4.11 were used to generate various four-phase implementations of the Stump processor from the same description shown in Appendix G.3. Two changes are required for the four-phase implementations. The first one is for the start circuit of the processor. This is shown in Figure 5.30. Because a rising transition and a falling transition have to flow into the **r1** input of the Call module at the top of the circuit in Figure 5.29, an XOR and a two-phase Toggle are required to enable the rising transition of Reset to pass through the XOR and into the Call module. Then a corresponding rising transition is sent to the **Rin** of the stage labelled **ss4**. After the **ss4** stage holds data the stage sends a rising transition to the **d** input of the

Table 5.6: The performance of the Stump processors

Circuit Name	Transistors (piece)	Run Time(μs)	Throughput (KIPS)	Latency (ns)	Energy (fj)
2-phase	171166	220.0	1090.7	687.2	1530.57
4-p Simple	170428	378.9	633.4	1148.4	1329.71
4-p Semi	170922	411.8	582.8	1249.1	1514.44
4-p Fully	171028	237.3	1011.3	738.0	1109.13
4-p Long	171350	240.9	996.1	748.1	1067.5

Call module. Immediately there is a rising transition on the output labelled **d1**. This rising transition goes through the Toggle and shows on its dotted output. Then the rising transition goes through the XOR and becomes a falling transition. This falling transition goes through the **ss4** stage and appears on the output labelled **d1**. Then the falling transition flows into the Toggle. A rising transition is present on the non-dotted output. This rising transition is used as a control signal of the multiplexer inside **ss4** stage, which is used to switch the latches connected to the initial values or the connections from the stage labelled **ss16**. The non-dotted output of the Toggle will remain at logic **Hi** except when Reset goes **Lo**. The second change is to remove the XOR whose inputs are connected to the signals labelled **r2** and **d2** of the Call module at the bottom of the circuit shown in Figure 5.29. A wire must be connected between the signal labelled **r2** and the control signal of the multiplexer inside stage **ss16**. This change is due to the four-phase protocol being used and a rising transition and a falling transition are present on the **r2** input of the Call module. If this change is not made there will be two four-phase events on the output of the XOR. A four-phase structural Stump processor circuit is shown in Figure 5.31.

Table 5.6 shows the number of the transistors and the run time of the 240 instructions, the throughput, the latency and the energy of the synthesized Stump processors using different control latch circuits (see Figures 2.10, 4.7, 4.9, 4.10 and 4.11).

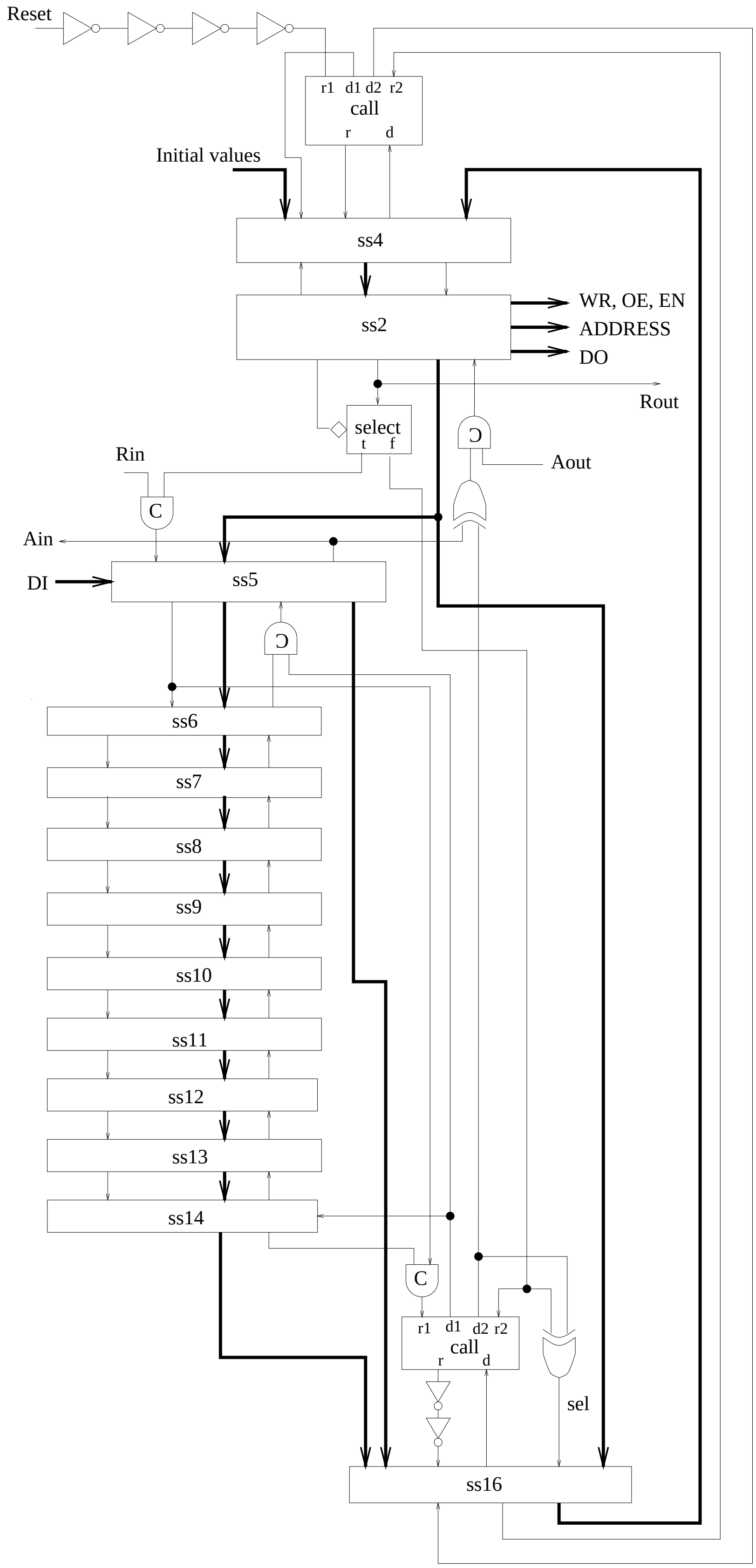


Figure 5.29: A two-phase structural Stump processor

5.6. Further work

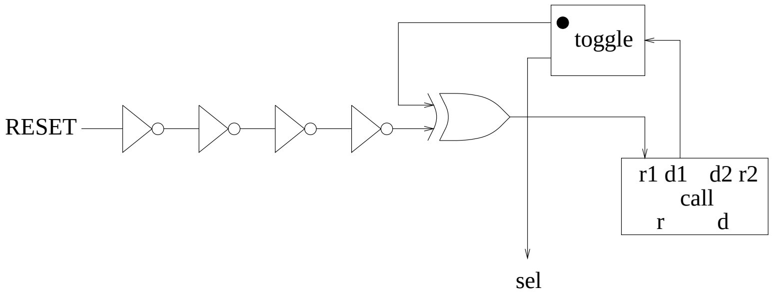


Figure 5.30: A change is required for the 4-phase handshaking protocol

5.6.1 Return to zero problems

There is no Arbiter in the circuit shown in Figure 5.29 and the simulations ran correctly on the leapfrog simulator. When four-phase control circuits are used within each stage and four-phase control modules are also used for the connections between stages it is necessary to put an Arbiter at the front of the Call module which is connect to the **ss16** stage as shown in Figure 5.31. Because the **ss5** stage is connected to both **ss6** and **ss16** stages, when the rising transition flows into the Call module a rising transition is sent to **ss6** stage. When stage **ss6** holds data the acknowledge rising transition is through **d** and **d1** to **ss14** and the C gate which is connected to the **Aout** of the **ss5** stage. The acknowledge rising transition goes through **ss14**, **ss13**, ... etc. When the **ss5** stage receives the transition the falling transition is sent out to the **ss6** stage and C gate which is connected to the **r1** of the Call module. At this moment another rising transition may be sent to the **r2** input of the Call module. The previous calling sequence for **r1** is not finished yet. The rising transition appears on the **r2** input will cause the Call module to go wrong. Each four-phase event consists of a rising transition and a falling transition. Both transitions must flow through the control path. The time from **r1** going **Hi** until **d1** going **Hi** and going **Lo** may be longer than that in two-phase counterpart. To avoid such circuit errors an Arbiter is required to connected to the Call module at the front of the **ss16** stage. Because only the initial event needs to go through **r1** on the Call module at the front of the **ss4**

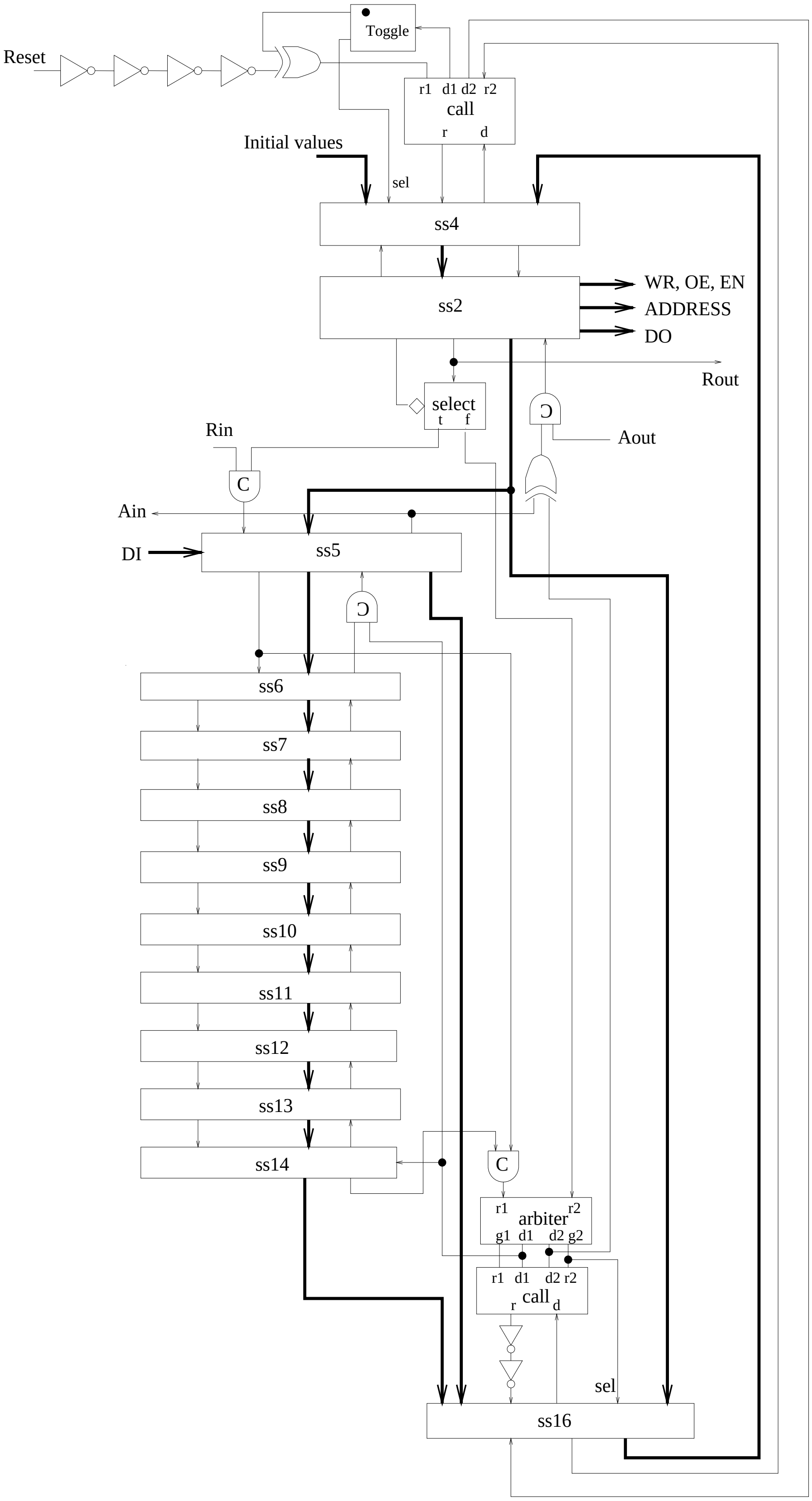


Figure 5.31: A four-phase structural Stump processor

stage, at the other times only the event sent from the **ss16** stage will go to **r2** on that Call module. An Arbiter is not necessary for that Call module.

A four-phase Call circuit [Liu97] is shown in Figure 5.32. Two 3-input asymmetric C-gates (see Figure 4.8) are used to send the rising transitions and falling transitions of **D** to **D1** or **D2** depending whether the event on the **R** output was generated from **R1** or **R2**. A 2-input asymmetric C-gate is used to send the rising transition and falling transition of **R1** or **R2** to the **R** output. Initially **R1** and **R2** are **low**. When a rising transition arrives on the **R1** input a rising transition is sent out from the **R** output. After a rising transition is received on the **D** input this causes a rising transition to be sent out from the output of the upper 3-input asymmetric C-gate on **D1**. If a rising transition arrives on the **R2** input before **R1** falls, **R2**, **R** and **D** will be at logical **'1'** and therefore a rising transition is sent out from the **D2** output in error. No transition corresponding to that on **R2** is sent out from the **R** output, and a deadlock may happen. A four-phase Arbiter costs about 40 transistors. The easy way to remove this problem is to connect an Arbiter at the front of the Call circuit. Alternatively, a more complex Call circuit can be used.

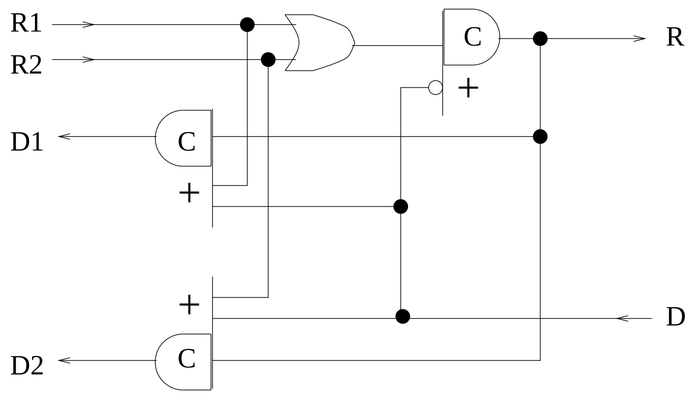


Figure 5.32: A four-phase call circuit

When the fully decoupled four-phase control circuit is applied the simulation of the processor stopped due to the Select module. The reason is that the **Rout** of the **ss2** stage sends out a rising transition to the Select module and the Select module will deliver this rising transition to one of its two outputs depending on the boolean value which is produced from the **ss2** stage. If the boolean value is a logic '0' the rising transition is sent to the output labelled **f**. Then the rising transition goes to the **r2** input of the Arbiter. The Arbiter sends the rising transition out from **g2** and the rising transition flows into the **r2** input of the Call module. Finally the rising transition arrives at the **Rin** input of the **ss16** stage. When the **ss16** stage holds data the acknowledge rising transition will be sent out. It will go through the **d** and **d2** of the Call module as well as an XOR and a C gate and arrive on **Aout** of the **ss2** stage. After the **ss2** stage receives a rising transition from **Aout** the data in the **ss2** stage may be changed. If the data is changed the boolean value of the Select module is also changed. Therefore, the falling transition sent from **Rout** of the **ss2** stage may be delivered to the wrong place. The simulation shows that the falling transition which should be sent to the **false** path was sent to the **true** path. This caused the simulation to fail. A four-phase Select module which remembers the boolean value at the time of sending the rising transition is required. After using this Select module the simulations ran successfully.

Due to the fact that there is no return to zero in a two-phase protocol, a two-phase protocol may be a better choice than a four-phase protocol for communication between stages.

5.6.2 Sharing resources

Figure 5.33 shows that the **stg4** stage contains two computations, a 16-bit adder and a 17-bit adder. A resource sharing implementation may be applied to save cost. A 17-bit adder stage can be created and calling this adder stage twice may get the same result. To implement resource sharing a re-partitioning of the stages is required. Latches are put at the input and output ends of the computations. After putting latches at the input and output ends of the computations inside the **stg4** stage and adding control circuits for the latches some new stages are created as shown in Figure 5.34. The **stg3** stage forks into the **stg4**, **stg31** and **stg35** stages and the **stg3**, **stg32** and **stg36** stages join into the **stg4** stage.

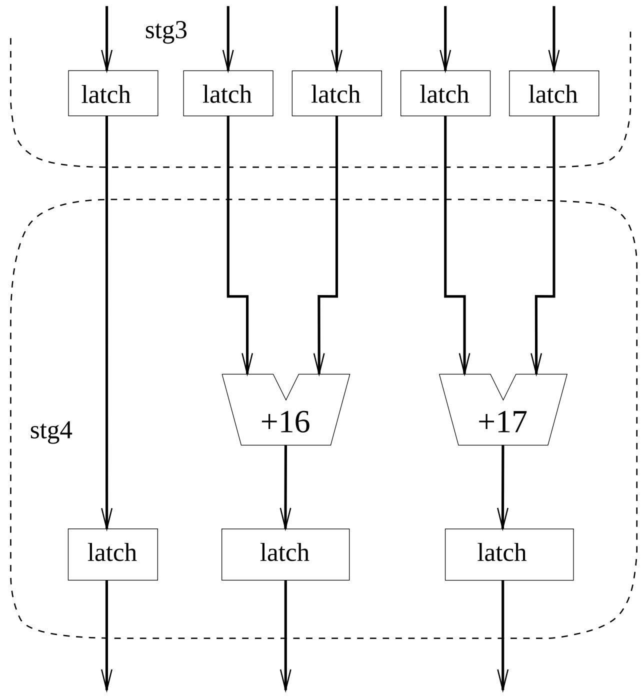


Figure 5.33: A stage(stg4) contains a 16-bit adder and a 17-bit adder

Some Arbiter and Call modules are required to build the circuit for connecting different source stages to the resource stage. For example, if there are five source stages Figure 5.35 shows the Arbiter and Call circuits which can be used to connect the five source stages to the resource stage. The source stage which is

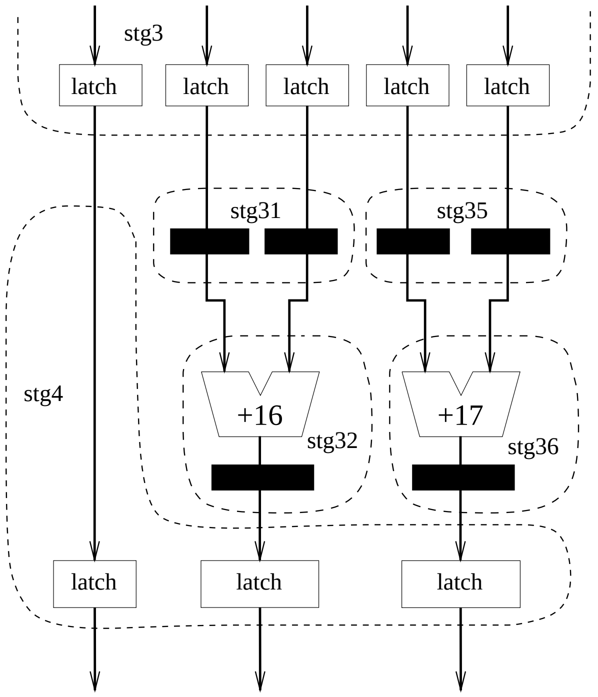


Figure 5.34: Some latches are connected at the inputs and outputs of adders.

identified as '1' has to connect its **Rout** and **Aout** to the **r1** and **d1** inputs which are labelled **1**. The source stage which is identified as '2' has to connect its **Rout** and **Aout** to the **r2** and **d2** inputs which are labelled **2**, and so on. Two sets of multiplexer circuits as shown in Figure 5.35 are required to connect the data from different source stages to the computation device inside the resource stage. The source stage which is identified as '1' has to connect two data outputs to the two multiplexers which are labelled **1**. The source stage which is identified as '4' has to connect two data outputs to the two multiplexers which are labelled **4**. The signals labelled **s3** are connected together. They are used to control the multiplexers. These connections of the multiplexer controls are for the four-phase protocol.

Select and XOR circuits are also required to connect the **Rout** and **Aout** of the resource stage to different output stages. Figure 5.36 shows Select and XOR circuits which can be used to connect a resource stage to five output stages. The output stage which is identified as '3' has to connect its **Rin** and **Ain** to the

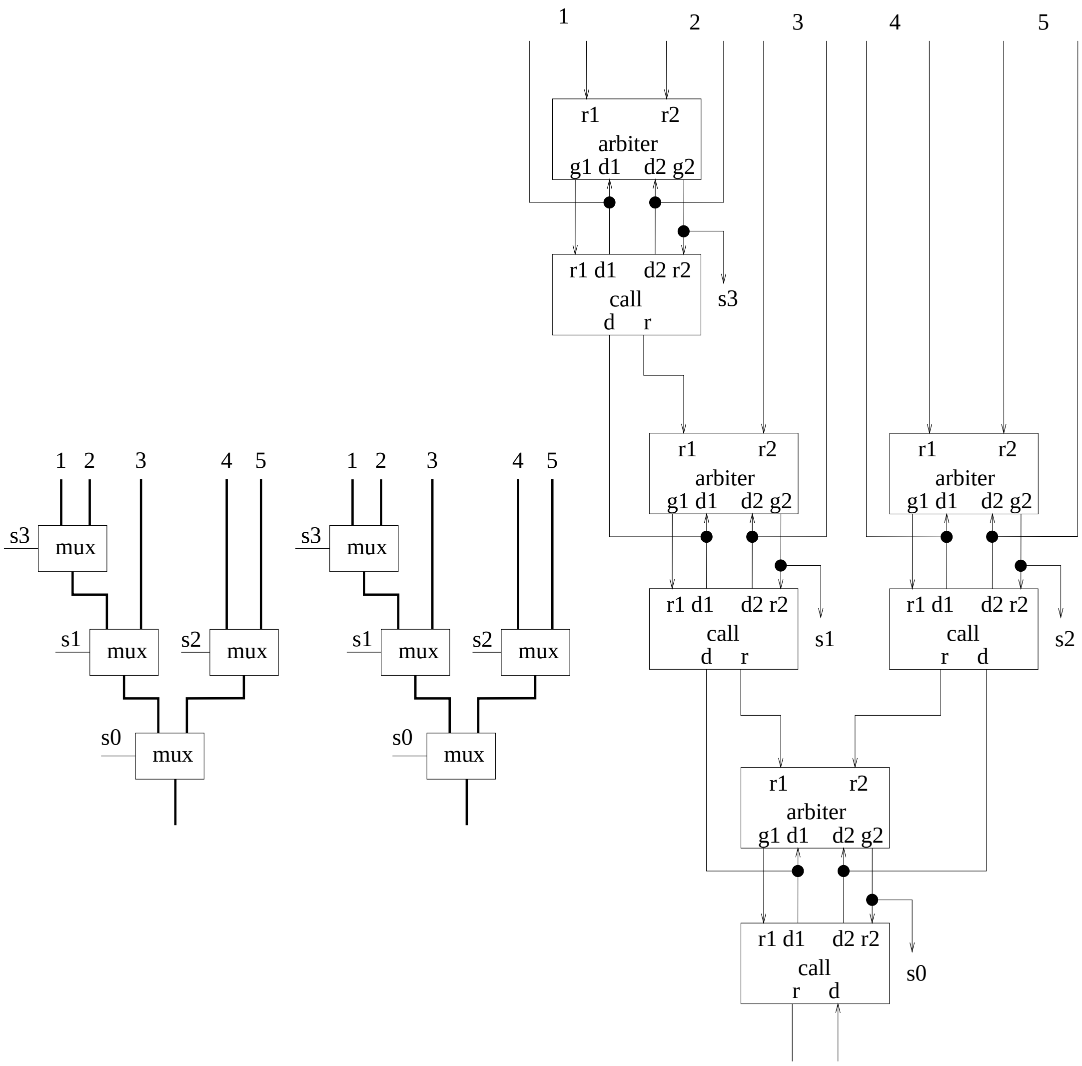


Figure 5.35: The Arbiters, Calls and Muxs are connected to control the resource stage receiving requests from different stages.

Select and XOR circuits which are labelled **3**. The inputs labelled **s1, s2, s3, s4** of the latch shown in Figure 5.36 are generated from the circuits shown in Figure 5.35. The outputs labelled **b1, b2, b3, b4** of the latch shown in Figure 5.36 are used to decide where the events should be sent to. They are connected to the inputs labelled **b1, b2, b3, b4** of the Select circuits.

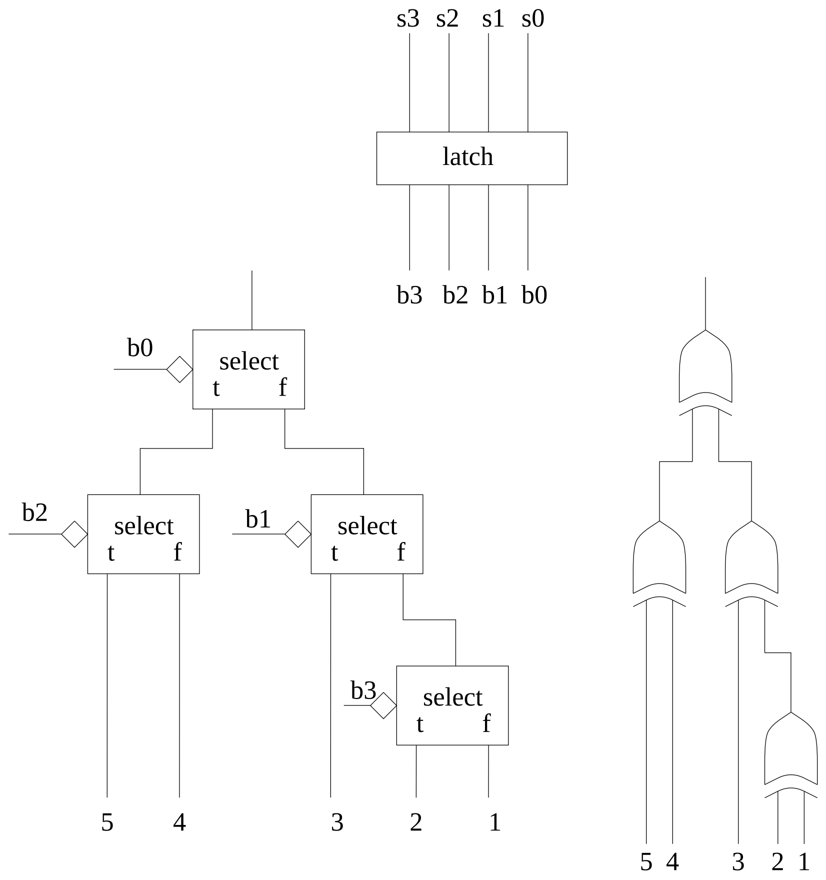


Figure 5.36: The selects and XORs are used to deliver the request from the resource stage to the corresponding stage.

Using the above techniques a 17-bit adder resource stage can be created and called by two source stages, **stg31** and **stg35**. The circuit is shown in Figure 5.37. The resource stage is connected to two output stages, **stg32** and **stg36**. The two-phase protocol is used in this circuit. The **stg31** stage is connected to the Arbiter, Call and Mux circuits at the position labelled **1**. The **stg35** stage is connected to the Arbiter, Call and Mux circuits at the position labelled **2**. The **stg32** stage is connected to the Select and XOR circuits at the position labelled

1. The **stg36** stage is connected to the Select and XOR circuits at the position labelled **2**. Those non-connected inputs of the MUXs may need to be connected to logic '0' or '1' depending on what the computations are.

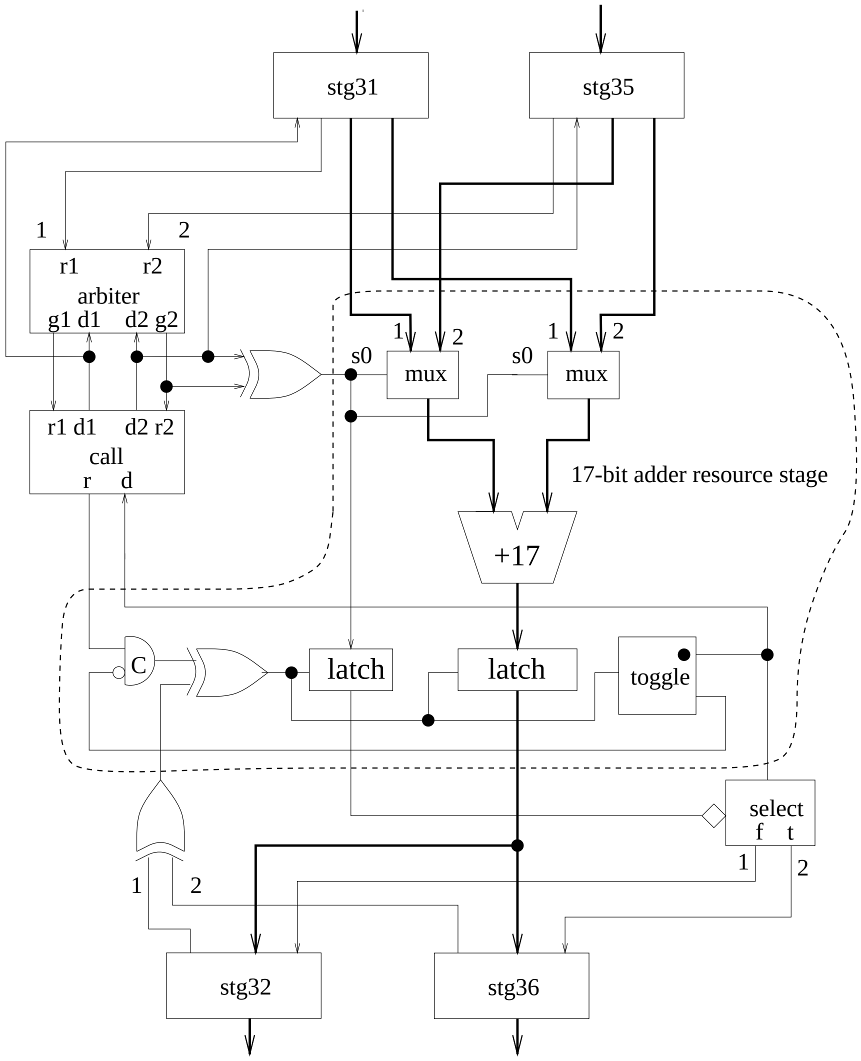
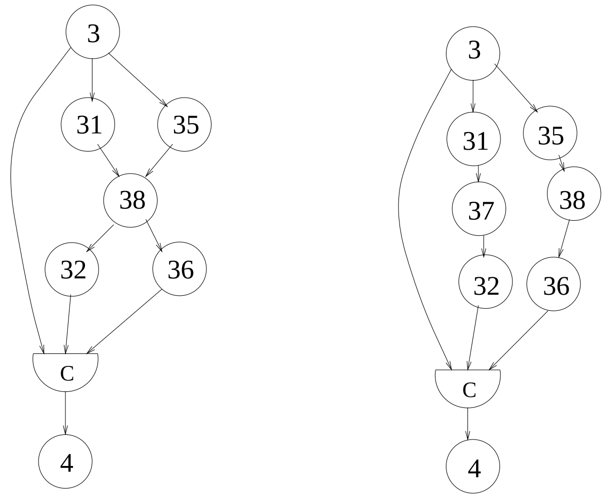


Figure 5.37: The stages (stg31 and stg35) use a resource stage (a 17-bit adder) to generate two computations in different time.

The circuit in Figure 5.34 may be denoted by Figure 5.38(a) or Figure 5.38(b) depending on whether one resource is used or two resources are used. Both circuits which are implemented using a two-phase protocol run correctly. However, Figure 5.38(a) has deadlock if a four-phase protocol is used to implement the circuit. The four-phase circuit of Figure 5.38(b) can run correctly without deadlock. The reason is that the rising transition of the **stg3** stage is sent to the **stg4** stage, the **stg31** stage and the **stg35** stage at the same time. The acknowledge rising transitions of the **stg31** stage and the **stg35** stage are sent out when these two stages hold data. However, the **stg4** stage is waiting for the rising transitions from the **stg32** stage and the **stg36** stage. If the **stg31** stage first calls the resource stage, **stg38** the Rin of **stg38** will not return to zero due to the **stg31** stage can not get the falling transition on its **Rin**. To avoid the deadlock the method shown in Figures 5.39, 5.40 and 5.41 can be applied.



(a) 2 sources and 1 resources (b) 2 sources and 2 resources

Figure 5.38: The stages (stg31 and stg35) use a resource stage (a 17-bit adder) to generate two computations in different time.

The example shown in Figure 5.39 illustrates how the steps work. The first step is to create some new latches for those data from the **stg5** stage but not connected to the **stg41** stage. These latches form a new stage called **stg45**. Make the connections between the **stg5** and the new stage. Remember the destination stage from the **stg5** is the **stg6** stage. After processing the circuit of Figure 5.39 is changed as shown in Figure 5.40.

The circuit of Figure 5.38(a) can use the above method to avoid deadlock. However, the circuit of Figure 5.40 still contains deadlock. A second step can be followed.

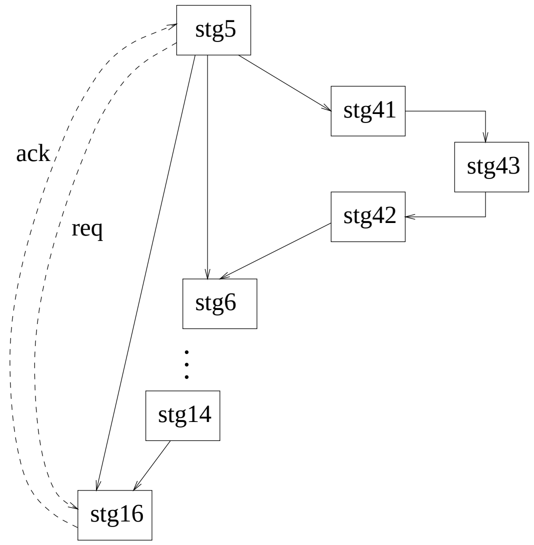


Figure 5.39: To avoid deadlock in 4-phase circuits some additional stages are required in the path(1).

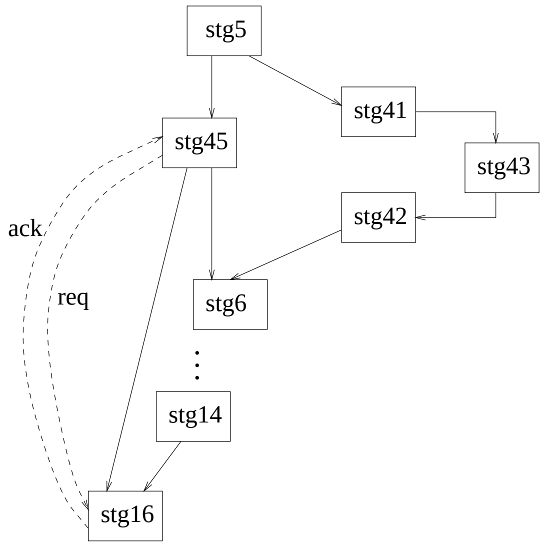


Figure 5.40: To avoid deadlock in 4-phase circuits some additional stages are required in the path(2).

The second step is to create some new latches for those data from the **stg45** stage but not connected to the **stg6** stage. These latches form a new stage called **stg46**. Make the connections between **stg45** and the new stage. The final circuit is shown in Figure 5.41. Now the circuit shown in Figure 5.41 can run correctly without deadlock.

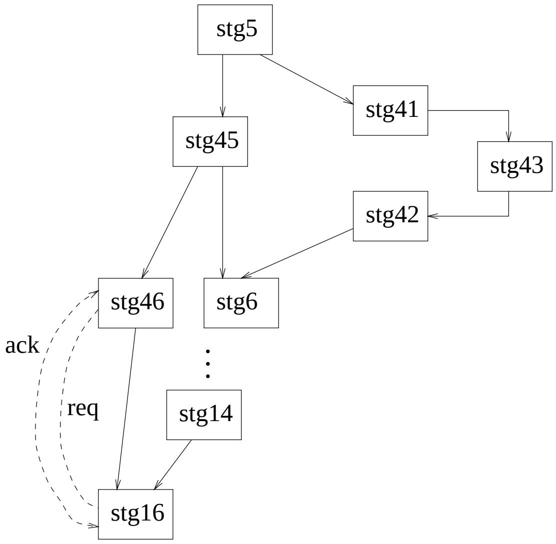


Figure 5.41: To avoid deadlock in 4-phase circuits some additional stages are required in the path(3).

The synthesized Stump processors of Figures 5.29 and 5.31 contain some 16-bit and 17-bit adders as well as some 16-bit and 17-bit subtractors. One choice is that a 17-bit adder and a 17-bit subtractor process all addition and subtraction operations within different stages. The other is that a 16-bit and a 17-bit adders as well as a 16-bit and a 17-bit subtractors process all addition and subtraction operations within different stages. The two-phase circuit (Figure 5.29) of the Stump processor can be converted into resource shared circuits with the exact same size or with a different size. These circuits were simulated correctly using the leapfrog simulator.

The resource shared circuits for the above two cases using four-phase control for the Stump processor circuit (Figure 5.31) have to use the above process to avoid deadlock. An example of the resource shared circuits with common sizes of the

Stump processor circuit is shown in Figure 5.42. **47** and **48** are resource stages. They are a 17-bit adder and a 17-bit subtractor. The circuit shown in Figure 5.42 is after processing. Only synthesized circuits using the fully decoupled and the long hold control circuits were simulated correctly using the leapfrog simulator and PowerMill. The reason is that the falling transitions of **Rout** in the fully decoupled and the long hold control circuits can be sent out before the falling transition arrives on **Rin**. Therefore, the falling transitions of **Aout** in the fully decoupled and the long hold control circuits can arrive before the falling transition arrives on **Rin**. However, it is necessary to ensure that the falling transitions of **Rout** in the simple and semi-decoupled control circuits are sent out after the falling transition arrives on **Rin**. Deadlocks may happen.

For example, as shown in Figure 5.42, **stg11** sends rising transitions to **stg26**, **stg28**, **stg30**, **stg32**, **stg34**, **stg36**, **stg38** and **stg40**. Then **stg26** sents the request transition to **stg47**. After **stg47** holds data it sends a rising transition to **stg27**. **stg27** is one of eight stages which connect to **stg12**. It is necessary to wait for these eight stages to send requests to **stg12**. Then **stg12** will send the acknowledge to these eight stages. However, four stages will still wait to send requests to **stg47**. On the other hand, **stg29**, **stg35**, **stg37**, **stg39** are still waiting for data from **stg47**. It is then impossible to get the rising transition from the **stg12**. As shown in Figure 5.43, **Rout** of **stg27** stays at logical **'1'** and the logical **'0'** on **Rin** of **stg27** cannot pass the C-gate without a rising transition on **Aout** of **stg27**. Eight extra buffer stages are required to connect from **stg27**, **stg29**, **stg31**, **stg33**, **stg35**, **stg37**, **stg39** and **stg41** to **stg12** individually. In total twelve extra buffer stages are required to ensure that the resource shared four-phase simple and semi-decoupled Stump processors can operate properly.

Table 5.7 shows the performance of the resource shared four-phase fully decoupled and long hold Stump processors.

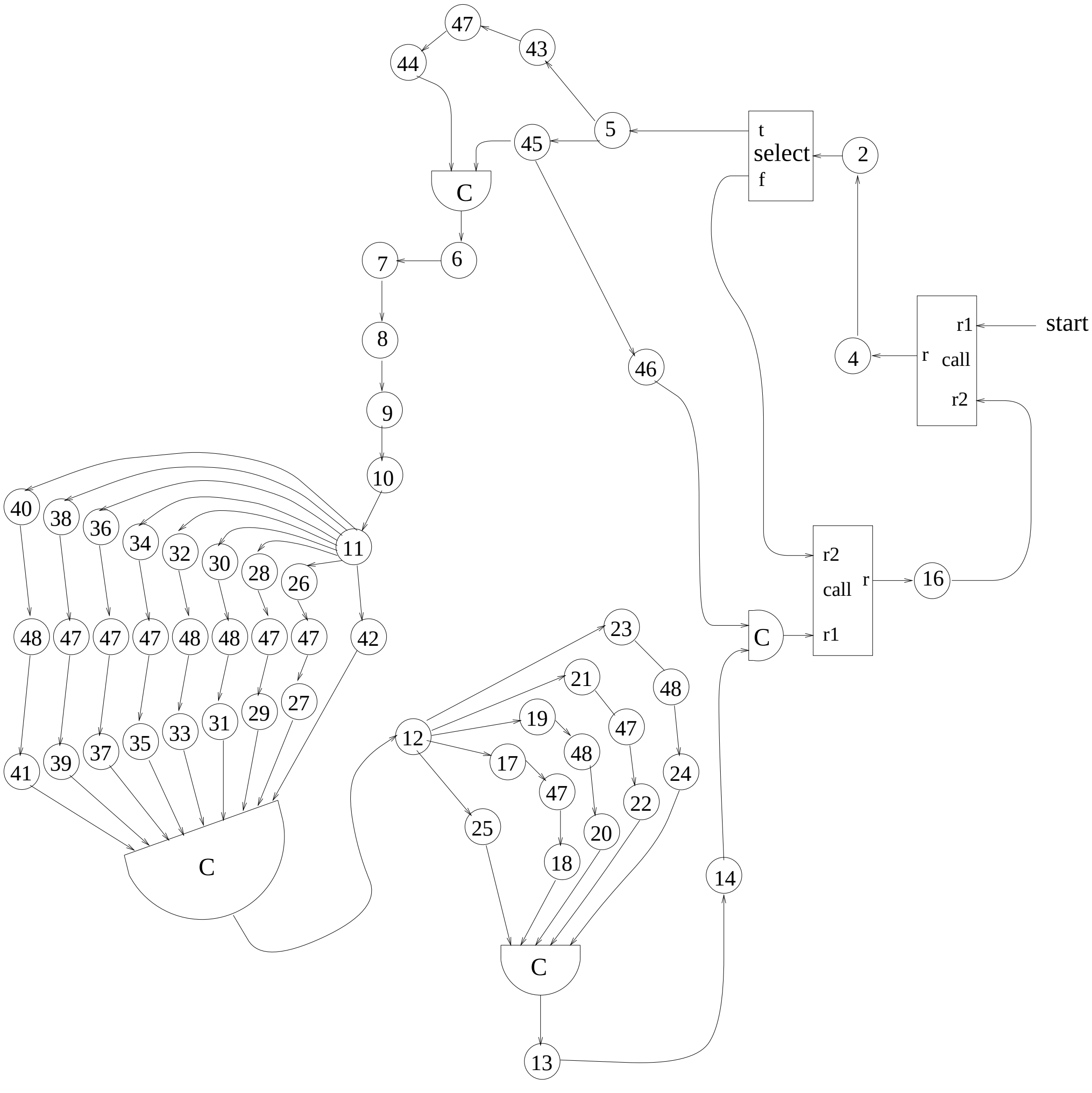


Figure 5.42: Resource stages (47 and 48) are called from different stages without deadlock.

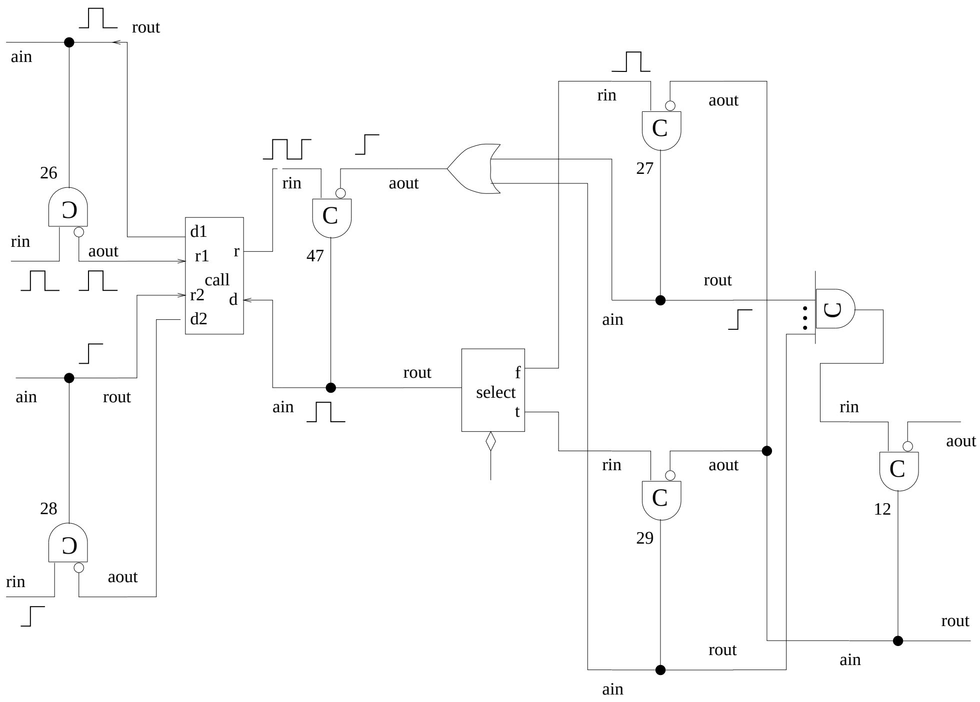


Figure 5.43: Four stages of the simple control circuits

Table 5.7: The performance of the resource shared Stump processors

Circuit Name	Transistors (piece)	Run Time(μs)	Throughput (KIPS)	Latency (ns)	Energy (fj)
2-phase 4 resources	184934	415.9	577.1	1299.4	1760.59
2-phase 2 resources	183412	536.8	447.1	1678.0	1834.65
4-p Fully 4 resources	193396	467.4	513.5	1460.5	1547.82
4-p Fully 2 resources	191374	617.9	388.4	1931.3	1686.65
4-p Long 4 resources	194586	461.3	520.3	1441.2	1487.89
4-p Long 2 resources	192512	611.1	392.8	1910.2	1626.53

Chapter 6

Verification, evaluation and discussion

Some experimental results, a comparison of two-phase and four-phase circuits, the optimization procedure and an analysis of the synthesized circuits are presented in this chapter.

6.1 Verification

Currently, verification is performed by checking that the simulation results produced by the behavioural VHDL description are the same as the simulation results produced by the structural VHDL description. Verilog simulation may be used to verify that the synthesized circuits are correctly imported into the Cadence database. TimeMill simulation is used to verify that the layout of the synthesized circuit is correct. The VHDL component library is used for the VHDL simulation of the synthesized circuits. The Cadence component library is used for Verilog simulation of the synthesized circuits in the Cadence environment. The Cadence component library is also used for generating the layout of the synthesized circuits in the Cadence environment. To produce correct designs it is important to ensure that both libraries contain correct components. During the work the author discovered an error in TimeMill simulation. After inverters were added at the **Lt** input of the transparent latches the correct simulation result was obtained. The problem is due to the delay on the control path being less than the delay on the data path. This caused wrong data to be latched.

The two-phase and four-phase control modules (see subsections 2.5.2 and 4.1.3) were implemented using standard logic gates in both VHDL and Cadence libraries. Some schematics of these control circuits are shown in Appendix B. Tables 6.1, 6.2 and 6.3 show the delay time and the number of transistors for standard logic gates and the two-phase and four-phase control modules. The synthesizer can check the data path delay against the control path delay using this delay information. The same kind of errors can be avoided.

Table 6.1: Standard Logic Gates

Name	Description	Number of Transistors	Gate Delay(ns)
not1	Inverter	2	1
and2	2-input AND	6	2.4
and3	3-input AND	8	2.8
and4	4-input AND	10	3.2
or2	2-input OR	6	2.4
or3	3-input OR	8	2.8
or4	4-input OR	10	3.2
nand2	2-input NAND	4	1.4
nand3	3-input NAND	8	1.8
nand4	4-input NAND	10	2.2
nor2	2-input NOR	4	1.4
nor3	3-input NOR	8	1.8
nor4	4-input NOR	10	2.2
xor2	2-input XOR	14	4.2
xnor2	2-input XNOR	12	3.2

6.2 Evaluation

An evaluation of the synthesizer in terms of performance, advantages and disadvantages is presented in this section.

Table 6.2: The properties of 2-phase control modules

Name	Description	Number of Transistors	Gate Delay(ns)
arbiter2	2-input Arbiter	138	10.8 17.4
toggle2	2-output Toggle	38	5.0
select1	Select	86	9.4 9.8
xor2	2-input XOR	14	4.2
c2	2-input C element	32	3.6
call2	2-input Call	106	R 4.2 D1/D2 7.8

Table 6.3: The properties of 4-phase control modules

Name	Description	Number of Transistors	Gate Delay(ns)
arbiter2_4	2-input Arbiter	40	5.6 10 (R1→G1/R2→G2) 1.4 (D1/D2→G1/G2)
toggle2_4	2-output Toggle	78	2.8 3.2
select1_4	Select	38	3.2
xor2	2-input XOR	14	4.2
c3	3-input C element	44	4.4
c2_m1	2-input Asymmetric C element with a input labelled −	16	2.8 3.2
c2_p1	2-input Asymmetric C element with a input labelled +	20	3.2
c3_m1	3-input Asymmetric C element with a input labelled −	42	4.0
c3_p1_m1	3-input Asymmetric C element with 2 inputs labelled + and −	32	3.6
c4_p1_m1	4-input Asymmetric C element with 2 inputs labelled + and −	44	4.4
c3_p2	3-input Asymmetric C element with 2 inputs labelled +	16	2.8 3.2
c4	4-input C element	34	4.0 6.4
c5	5-input C element	44	6.4 6.8
c9	9-input C element	72	6.8 7.2
or2	2-input OR	6	2.4
call2_4	2-input Call	24	2.4 (R1/R2→R) 2.8 (D→D1/D2)

6.2.1 Performance

The current version of the synthesizer can partition behavioural VHDL descriptions into several stages depending on whether the computations can be executed concurrently. The partition rules were described in Chapter 3. Basic assignments, IF statements and loop statements may be used for describing the behaviour of the synthesized system. The synthesizer follows conversion rules to produce the internal circuit models. The function of the statements within each stage is used to generate the corresponding control circuits. The information used includes:

- Whether only assignments are in the stage;
- Whether an unbalanced IF statement is in the stage;
- Whether a WHILE loop statement is in the stage;
- Whether a FOR loop statement is in the stage; and
- Whether a loop statement contains handshaking signals in the stage.

Before generating the synthesized circuits the synthesizer calculates the delays on the control paths and the data paths for each stage. Each component model has to contain its delay time. A terminal queue and a device stack are required to find an execution ordering for calculating the delay times on the data paths and the control paths. The maximum delay time is used for the data path. The minimum delay time is used for the control path.

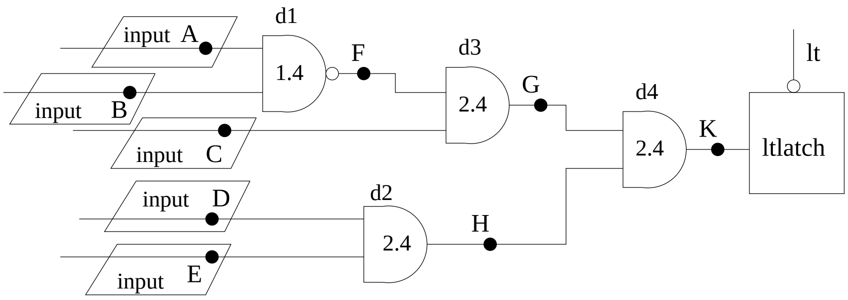


Figure 6.1: A circuit to show how the terminal queue and the device stack work.

For example, to get the delay time at the input labelled K of the transparent latch shown in Figure 6.1 the device labelled $d4$ is pushed onto the device stack and its two inputs, terminals G and H are inserted onto the terminal queue first as shown in Figure 6.2(b). Then the terminal labelled G is read from the terminal queue. The device whose output is connected to the terminal labelled G, i.e., the device labelled $d3$, will be pushed onto the device stack and its input, terminal F, is inserted onto the terminal queue as shown in Figure 6.2(C). Because the input terminal labelled C of the device labelled $d3$ is an input of the whole circuit, it is not necessary to insert it into the terminal queue. After the terminal labelled H is read from the terminal queue the device whose output is connected to the terminal labelled H, i.e., the device labelled $d2$, will be pushed onto the device stack as shown in Figure 6.2(d). However, it is not necessary to insert both of its inputs labelled D and E into the terminal queue for the same reason. After the terminal labelled F has been read from the terminal queue the device labelled $d1$ will be pushed onto the device stack as shown in Figure 6.2(e).

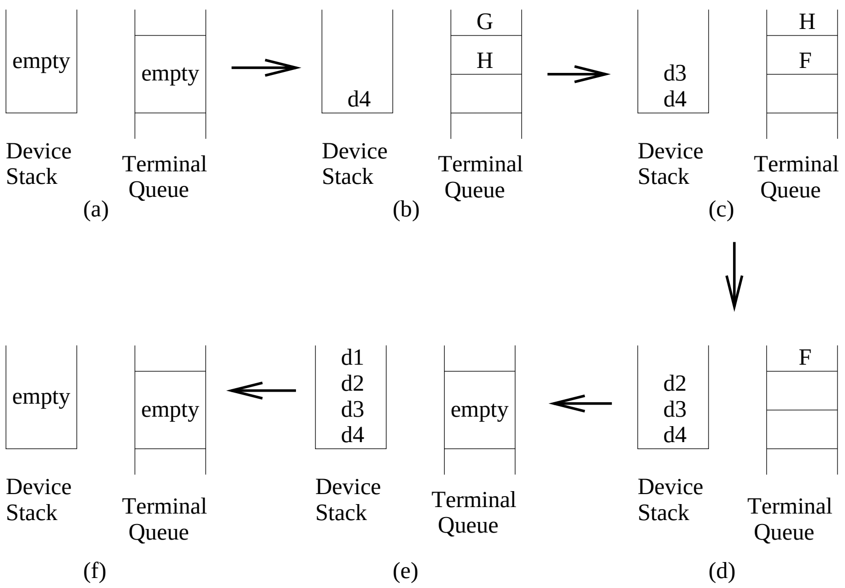


Figure 6.2: The change of the terminal queue and the device stack of the example shown in Figure 6.1.

When the terminal queue is empty the devices in the device stack will be popped out to calculate the delay time. At the beginning the delay times inside terminals labelled A, B, C, D and E are zero. The calculation of the delay time inside the terminal labelled F becomes 1.4 ns after the device labelled $d1$ is popped out. The calculation of the delay time inside the terminal labelled H becomes 2.4 ns after the device labelled $d2$ is popped out. The calculation of the delay time inside the terminal labelled G becomes 3.8 ns after the device labelled $d3$ is popped. The calculation of the delay time inside the terminal labelled K becomes 6.2 ns after the device labelled $d4$ is popped out. Therefore, the data path delay is 6.2 ns. If the delay on the data paths is greater than the difference between the delay on the control paths and the safety delay margin an even number of inverters are added to the control path, the number chosen depending on the difference between the delays on the data path and the control path to ensure the correct data can be latched. For example, if the delay on the data path is 21 ns and the delay on the control path is 15.4 ns 12 inverters are added to the control path if the delay of the inverter is 1 ns and the assumed safety delay margin is 5.2 ns.

Both structural VHDL and Verilog files of the synthesized circuits are generated.

6.2.2 Advantages

One advantage of this synthesizer is that VHDL is used for both input and output descriptions. Before the actual components are available the synthesized circuits can be simulated using the VHDL models in Appendix A. The behavioural models and the synthesized circuits can also be simulated to check their correctness at an early design stage. Another output from the synthesizer is structural Verilog files which are used to import the synthesized circuits into the Cadence database. This offers a convenient path to continue the subsequent design work.

6.2.3 Disadvantages

The VHDL behaviour may be complex for a bigger system. This problem may be overcome if a graphical input interface which can produce the bundled handshaking controls automatically is available. However, it is a fact that the behavioural VHDL model can be simulated easily. Attention must be paid to check that the VHDL component library is consistent with the Cadence component library. Attention is also required to ensure that the VHDL and Verilog outputs are consistent.

6.3 A comparison of two phase and four phase implementations

6.3.1 Comparing 2-phase and 4-phase

In a 2-phase protocol rising and falling transitions have the same meaning. They represent request events or acknowledge events. For each bundled data communication only one transition is required on the request line and one on the acknowledge line. A return to zero action is not necessary. It is quite simple to think about the behaviour of a 2-phase micropipeline circuit. For example, if an event is on the input $R1$ of the Call module shown in Figure 4.5 an event will appear at the output R shortly if a 2-phase protocol is applied. To send another event to the input $R2$ without error it is only necessary to wait for the completion event appearing on the output $D1$. However, if a 4-phase protocol is applied a rising edge on the input $R1$ of the Call module will give a rising edge at the output R shortly, but to send another rising edge to the input $R2$ without error it is not only necessary to wait for the rising edge appearing on the output $D1$ but it is also necessary to wait for the falling edge appearing on the output

$D1$. If the computation times are the same for both cases the time between an event on the input $R1$ and the completion event appearing on the output $D1$ should be equal to the time between a rising edge on the input $R1$ and the corresponding rising edge appearing on the output $D1$. There is an additional time for the falling edge to appear on the output $D1$. However, the implementation of the 4-phase Call module may be simpler than the 2-phase one. Sometimes it may be easy and clear to construct the 2-phase micropipeline circuits first and then to construct the 4-phase micropipeline circuits by following the structure of the 2-phase micropipeline circuits.

Most digital components are level-sensitive. The 4-phase control circuit shown in Figure 4.7 is quite simple. The cost is low compared with the 2-phase control circuit shown in Figure 2.10. Whether the cost and area of the 4-phase control circuits are low and small depends on the implementation technique applied.

From the experiments in Chapter 4, repartitioning the computations of a 2-phase micropipeline circuit into more stages can give much speedup. Repartitioning the computations of a 4-phase micropipeline circuit into more stages gives less speedup.

6.3.2 The advantages of using 2-phase control

In a 2-phase micropipeline it is easier to think about the behaviour and design of the circuit. Due to there being no return to zero actions in 2-phase micropipelines they are potentially faster than 4-phase micropipelines.

6.3.3 The disadvantages of using 2-phase control

If the two-phase event-controlled storage element is used the switch circuits for each data bit require more area and cost. Standard latches are level-triggered components. If the control circuit shown in Figure 2.10 is used an additional cost and area are required for the 'Toggle' module.

6.3.4 The advantages of using 4-phase control

The return to zero actions carry no meaning in some 4-phase protocols. If asymmetric delay techniques are applied or the return to zero actions are implemented by a 'RESET' 4-phase micropipelines have good performance. Most digital components are level-sensitive. The control modules for 4-phase are simple and have low cost.

6.3.5 The disadvantages of using 4-phase control

Enough delay must be included to ensure that the previous handshaking actions are complete. Test programs in VHDL are more complex than those for 2-phase control.

6.4 Experimental results

Blocksorter

A 14-bit floating adder/subtractor, a blocksorter, and a simple simple Stump processor were used to test the synthesizer in Chapter 5. The properties of these synthesized circuits are shown in Tables 5.3, 5.4, 5.5, 5.6 and 5.7. 120 sets of test data were sent to the synthesized blocksorter for the Leapfrog simulation. **Rout** of the blocksorter is connected to the **Aout** input directly. A new request was sent to the blocksorter when the test program received a transition from the **Ain** output. The run time is the time difference between the **request-out** signal

Circuit	2-phase	4-p simple	4-p semi	4-p fully	4-p long
Energy	10.78	12.13	14.23	14.72	13.65

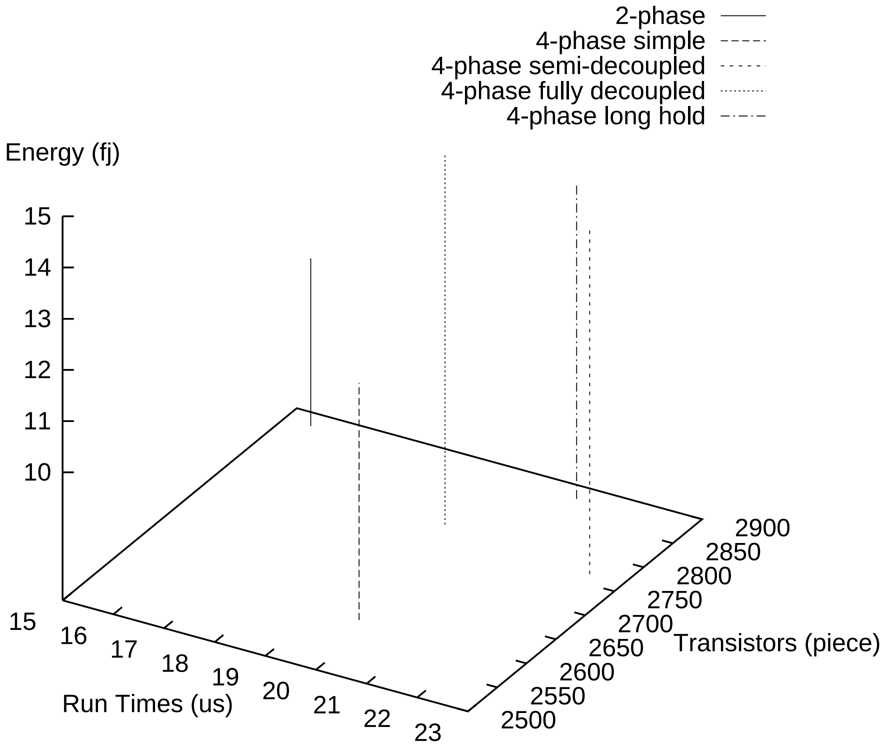


Figure 6.3: The performance of the synthesized blocksorter circuits

of the 120th data on **Rout** and the **request-in** signal of the first data on **Rin**. The energy information was obtained from PowerMill simulation for twelve sets of test data.

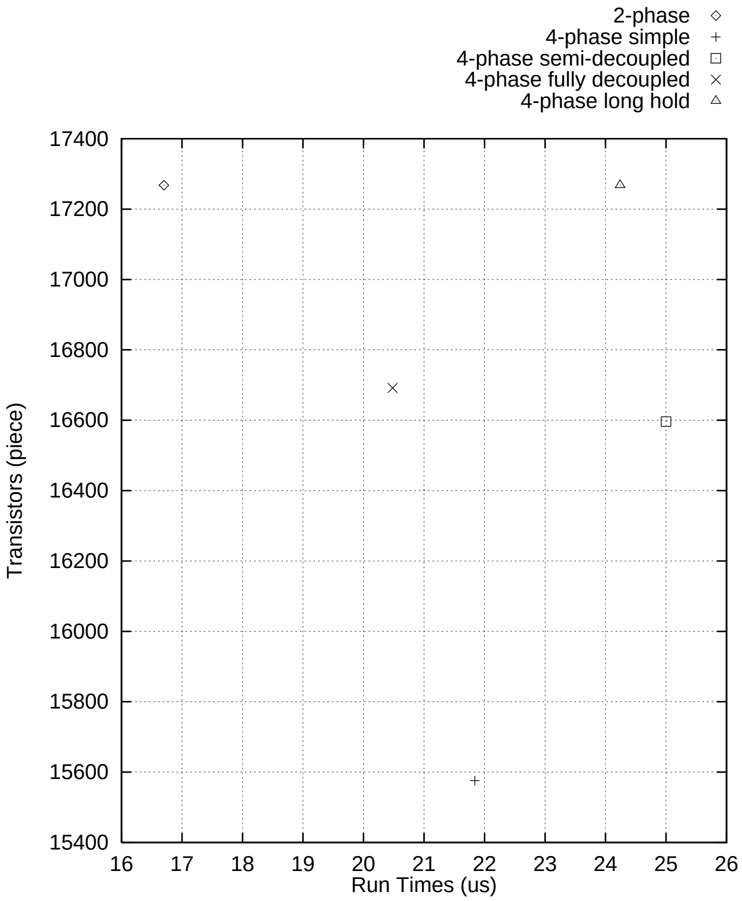


Figure 6.4: The run time and the transistor number of the synthesized blocksorter circuits

As shown in Figures 6.3, 6.4 and 6.5 the 2-phase blocksorter is the fastest and its power consumption is the lowest. The 2-phase blocksorter also has high throughput and high latency. But the cost is high. The blocksorter using the 4-phase simple latch control circuit has a small number of transistors. Therefore, its power consumption is low. The blocksorter using the 4-phase fully decoupled latch control circuit is the fastest of the different 4-phase circuits. It also has low latency. This means that the time between the **request-in** signal on **Rin** and the **request-out** signal on **Rout** is short. However, a waiting time is required in asynchronous design if the circuit is busy. The blocksorter using the 4-phase semi-decoupled control circuit does not have good performance here. The only advantage of the blocksorter using the 4-phase long hold control circuit is that it is fast.

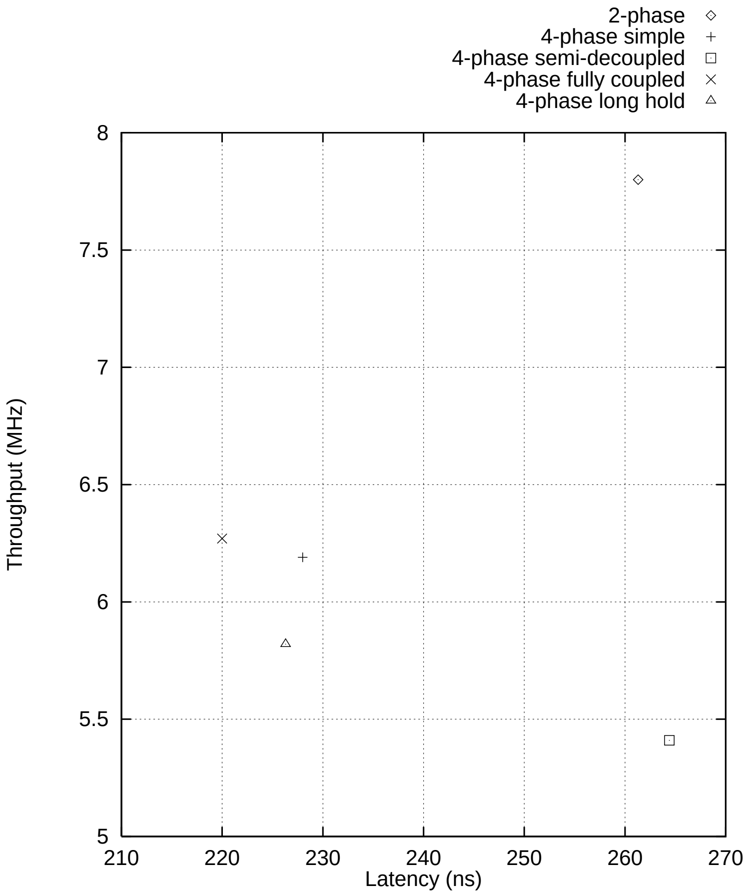


Figure 6.5: The throughput and latency of the synthesized blocksorter circuits

14-bit floating adder/subtractor

Circuit	2-phase	4-p simple	4-p semi	4-p fully	4-p long
Energy	156.12	168.36	183.77	168.69	163.16

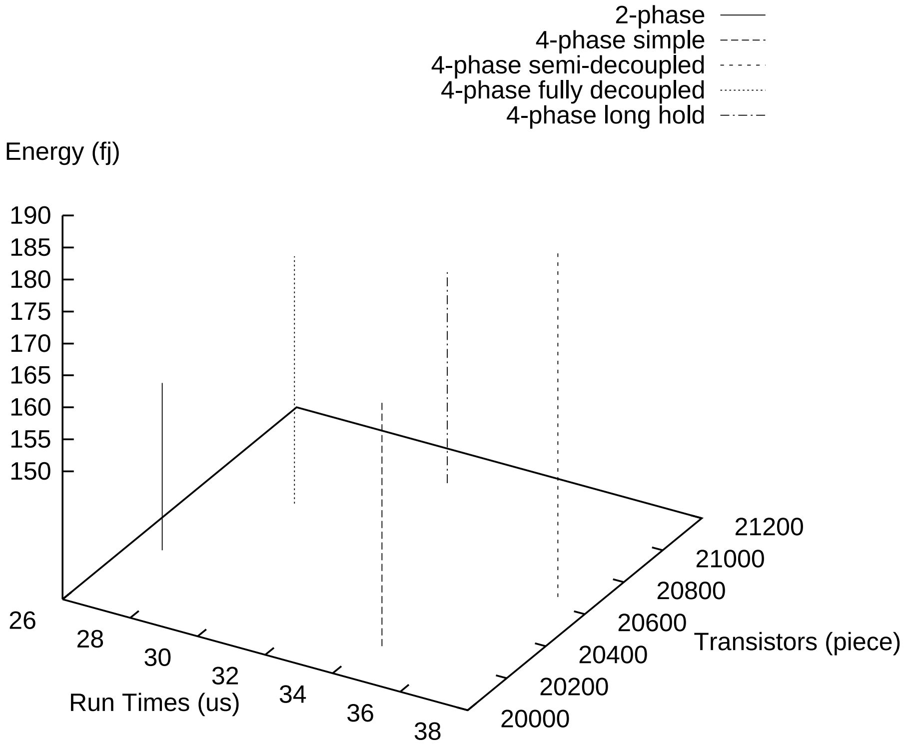


Figure 6.6: The performance of the synthesized 14-bit floating point adder/subtractors

150 sets of test data were sent to the synthesized floating point adder/subtractor for the Leapfrog simulation. **Rout** of the floating point adder/subtractor is connected to the **Aout** input directly. A new request was sent to the floating point adder/subtractor when the test program received a transition from the **Ain** output. The run time is the time difference between the **request-out** signal of the 150th data on **Rout** and the **request-in** signal of the first data on **Rin**. The energy information was obtained from PowerMill simulation for fifteen sets of test data.

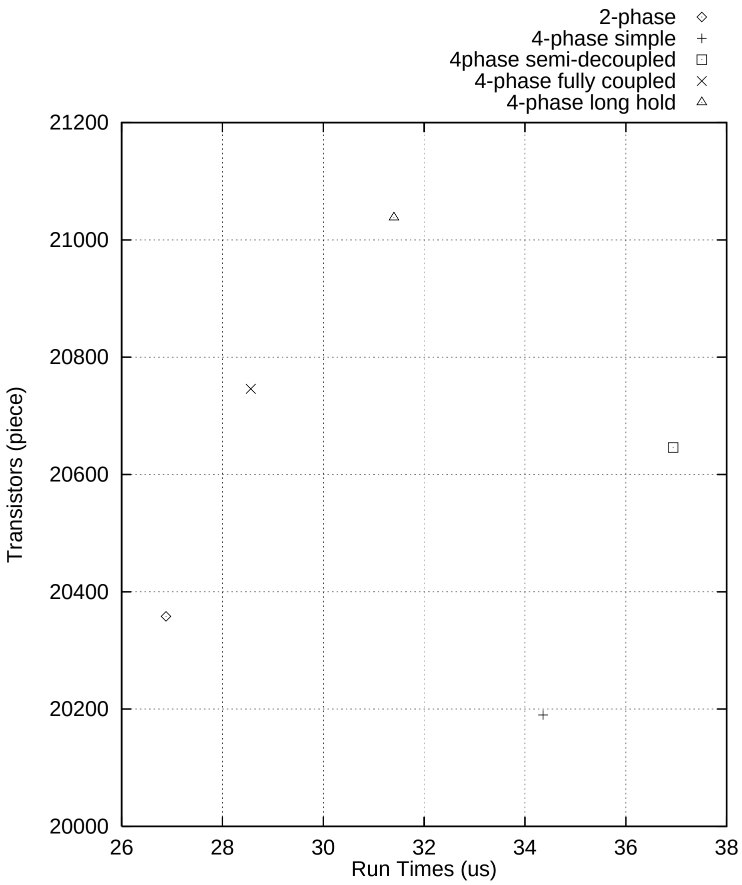


Figure 6.7: The performance of the synthesized 14-bit floating point adder/subtractors

As shown in Figures 6.6, 6.7 and 6.8 the fast circuit is the 2-phase implementation. The two-phase circuit also has high throughput, high latency and low power consumption. For this circuit the 2-phase design is not especially expensive, unlike the blocksorter. The floating point adder/subtractor using the 4-phase simple latch control circuit is the cheapest and has low latency. The 4-phase semi-decoupled circuit is slowest and has low throughput. Again the 4-phase fully decoupled circuit is the fastest of the different 4-phase circuits. It also has high throughput. The 4-phase long hold circuit has high cost.

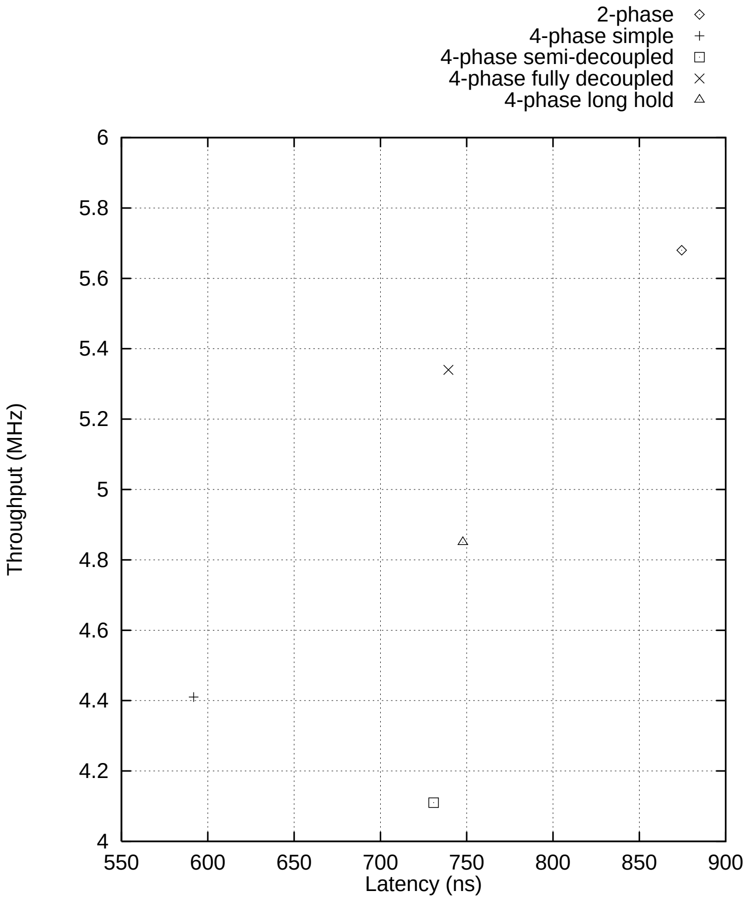


Figure 6.8: The throughput and latency of the synthesized 14-bit floating point adder/subtractors

Stump processor

A loop program with twelve instructions were used to test the synthesized Stump processors shown Figures 5.29 and 5.31. **Rout** of the synthesized Stump processor is connected to the **Aout** input directly. Memory data and a request or acknowledge was sent to the Stump processor by the test program when the Stump processor fetches instructions or reads data. The run time is the time difference between the 321th **request-out** signal on **Rout** and the first **request-out** signal on **Rout**. This is the time for 240 instructions to be executed. The energy information was obtained from PowerMill simulation for 12 instructions.

Circuit	2-p	4-p simple	4-p semi	4-p fully	4-p long
Energy	1530.57	1329.71	1514.44	1109.13	1067.50

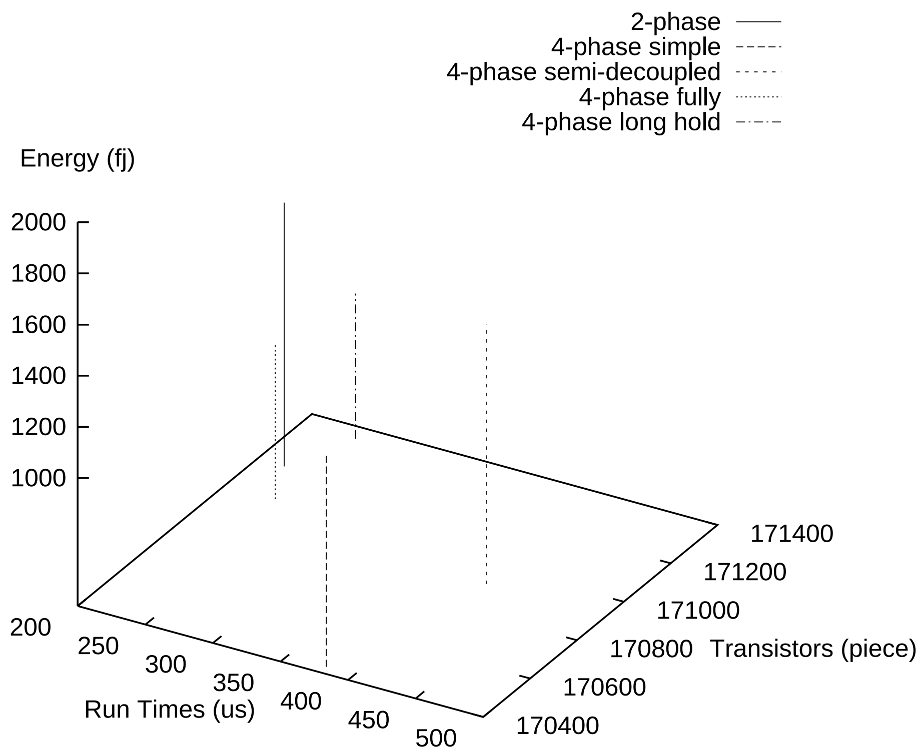


Figure 6.9: The performance of the original synthesized Stump processors

As shown in Figures 6.9, 6.10 and 6.11 the fast circuit is still the 2-phase implementation. The two-phase circuit also has high throughput and low latency. As before the 4-phase fully decoupled circuit is the fastest of the different 4-phase circuits. It also has high throughput. The 4-phase simple circuit is the cheapest circuit but it is slow. The synthesized Stump processor using the 4-phase semi-decoupled circuit is slowest and has low throughput as well as high latency. The 4-phase long hold circuit has low power consumption but high cost. However, its speed is not slow.

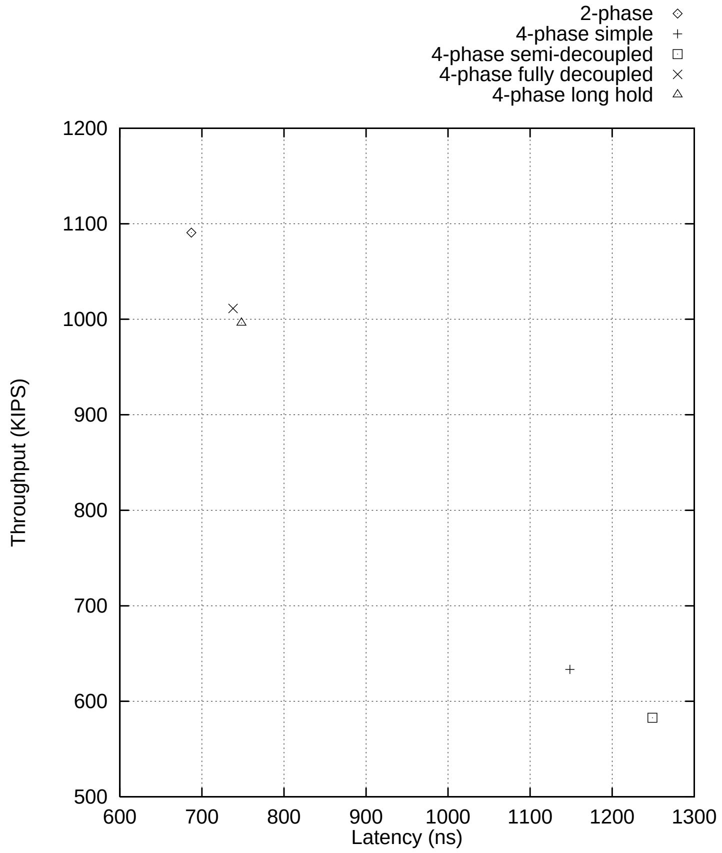


Figure 6.10: The latency and throughput of the original synthesized Stump processors

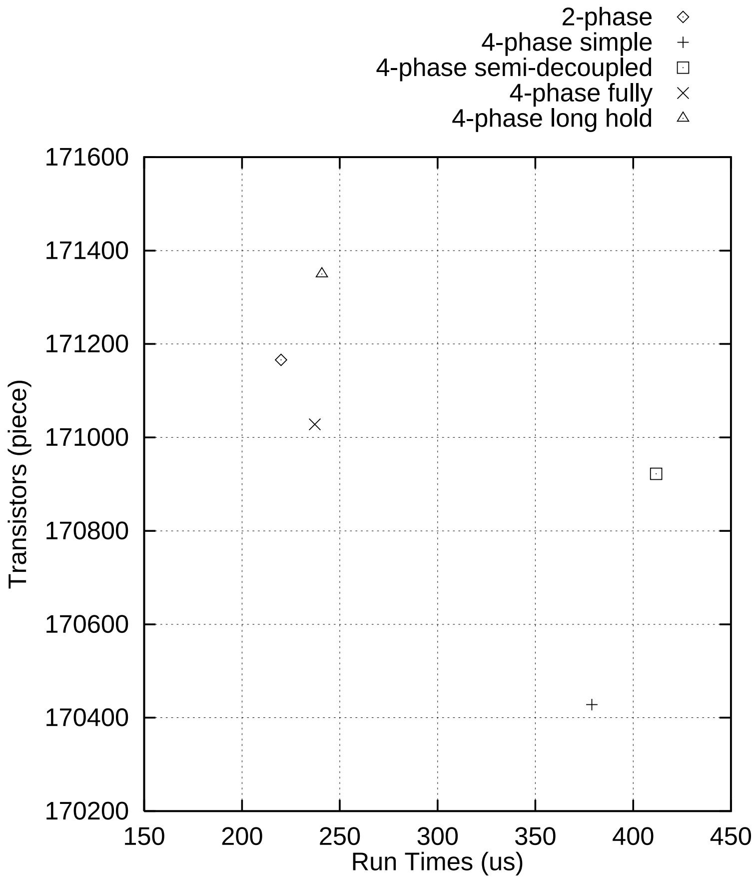


Figure 6.11: The performance of the original synthesized Stump processors

Sharing resources

Resource sharing was described in Chapter 5. The two-phase, four-phase fully-decoupled and four-phase long-hold Stumps were used to test this technique. The performance of the resource shared circuits are shown in Table 5.7.

Circuit	2-p 4-resource	4-p fully 4-resource	4-p long 4-resource	2-p 2-resource	4-p fully 2-resource	4-p long 2-resource
Energy	1760.59	1547.82	1487.89	1834.65	1686.65	1626.53

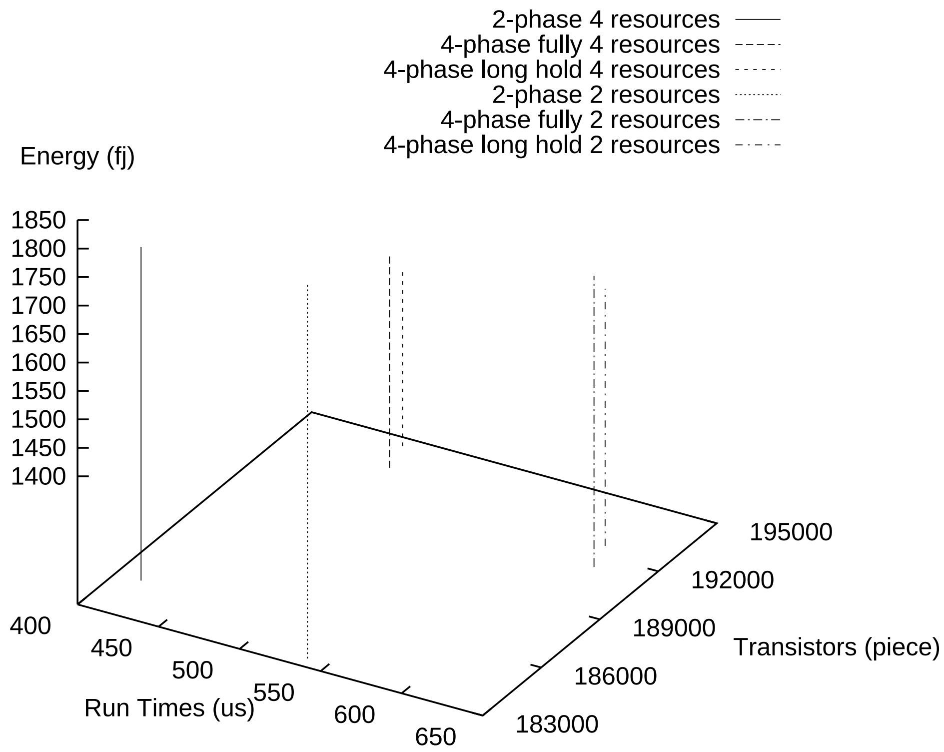


Figure 6.12: The performance of the shared resource Stump processors

As shown in Figures 6.12 and 6.13 the power consumption is high and the run time is increased by 50 percent compared to the original designs. This is just a demonstration of how the method works. If man expensive components are required in a circuit this method can be applied to reduce costs. It is clear that the two-phase control circuit is still fast.

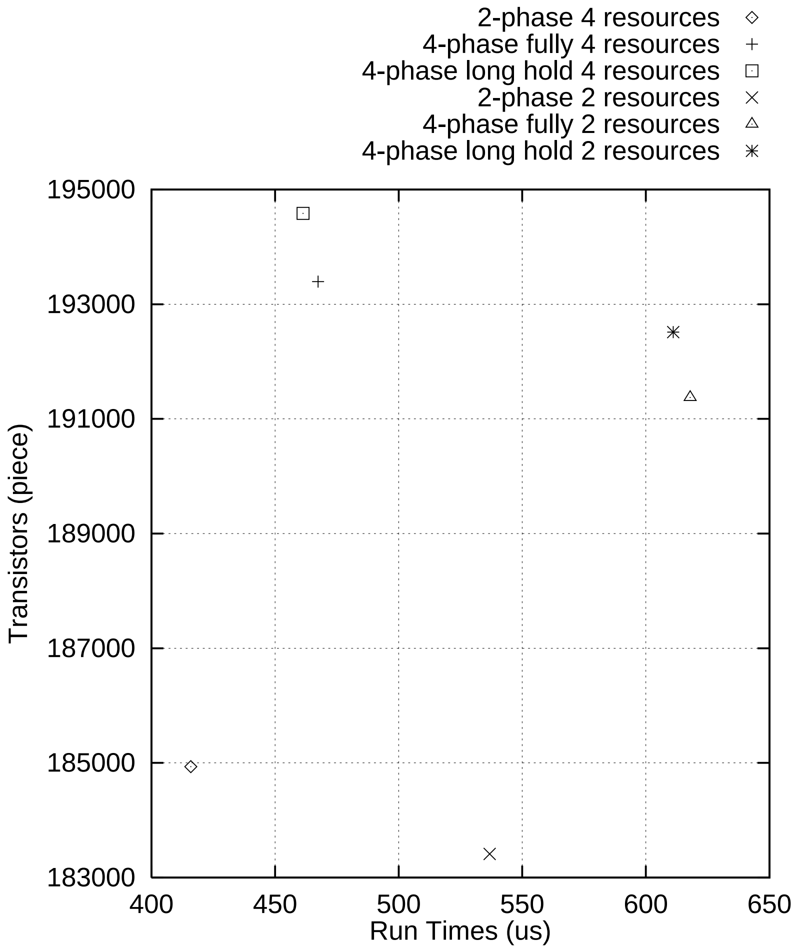


Figure 6.13: The performance of the shared resource Stump processors

Asymmetric delay

Figure 4.19 shows an asymmetric delay. The asymmetric delay was also applied to improved the performance of the circuits. The performance of the synthesized Stump using the asymmetric delay is shown in Table 6.4.

Circuit	4-p simple	4-p semi	4-p fully	4-p long
Energy	1332.53	1520.86	1109.53	1059.34

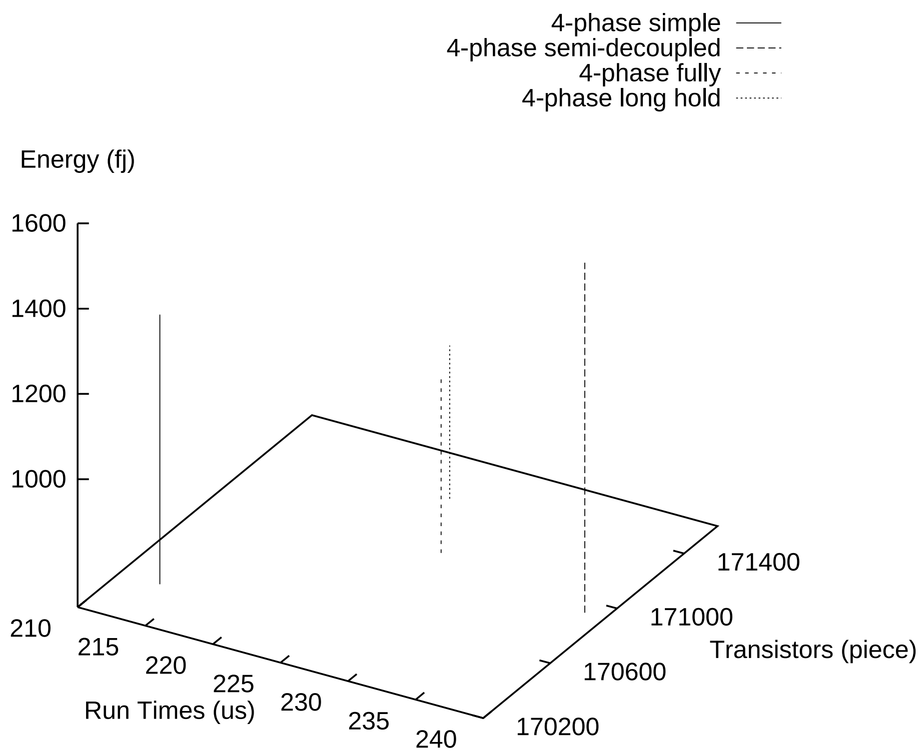


Figure 6.14: The performance of the Stump processors using asymmetric delay

As shown in Figures 6.14 and 6.15 the Stump processor using the four-phase simple control circuit is 40 percent faster than the original synthesized circuit. However, it is necessary to deal with asymmetric delays very carefully. If the asymmetric delay is applied it is necessary that no transition can arrive on the input within double the delay time. The reason is that some unwanted transitions

Table 6.4: The performance of the Stump processors with asymmetric delay

Circuit Name	Transistors (piece)	Run Time(μs)	Throughput (KIPS)	Latency (ns)	Energy (fj)
4-p Simple	170448	213.0	1127.0	662.4	1332.53
4-p Semi	170928	238.5	1006.4	740.8	1520.86
4-p Fully	171040	226.5	1059.6	670.2	1109.53
4-p Long	171350	223.3	1074.7	658.4	1059.34

appear on the control signal after the **AND** operation. The four-phase fully decoupled and long hold control circuits can easily meet such problems. The time between the rising transition and the falling transition of the control signal is about 70 ns. If the data path delay is bigger and the asymmetric delay is used the circuit may go wrong.

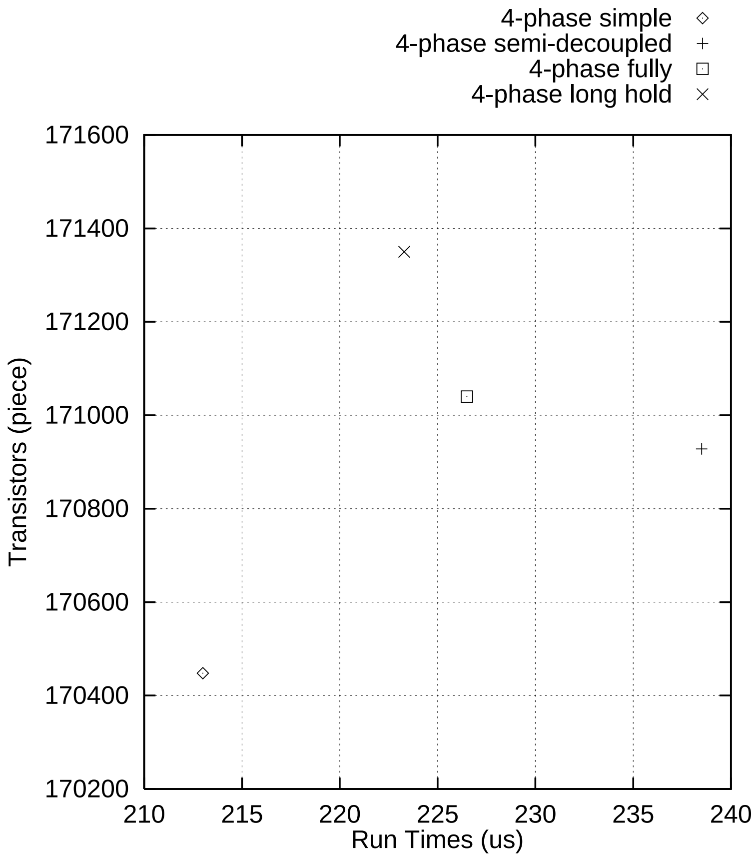


Figure 6.15: The performance of the Stump processors using asymmetric delay

6.5 Optimization

If two stages only contain assignments, the first is the only input stage to the second, and the second is the only output stage from the first, then these two stages can be merged. If the first one is called $stageA$ and the second is called $stageB$, the procedure to merge the two stages is as follows:

1. Connect the outputs of each latch to the sources of its inputs inside $stageA$ and remove these latches as well as redundant connections.
2. Remove the outputs of $stageA$ and the inputs of $stageB$ from the device list.
3. Remove the control circuits of $stageA$.
4. Remove the interconnections between $stageA$ and $stageB$ from the network device list.

However, one more condition must be satisfied if the simple 4-phase or the semi-decoupled 4-phase control circuits of Figures 4.7 and 4.9 are applied. This is that if the first stage is an output stage of the second then the two stages cannot be merged. The 2-phase Stump processor shown in Figure 5.29 can be merged as shown in Figure 6.16. The performance of the Stump processors after merge is shown in Table 6.5.

Table 6.5: The performance of the Stump processors after merge(1)

Circuit Name	Transistors (piece)	Run Time(μs)	Throughput (KIPS)	Latency (ns)	Energy (fj)
2-phase	137226	159.8	1502.3	499.2	522.37
4-p Simple	140016	289.8	828.1	878.7	558.77
4-p Semi	140206	295.9	811.2	898.2	613.95
4-p Fully	137088	262.2	915.3	801.2	417.46
4-p Long	137292	266.8	899.5	811.5	404.62

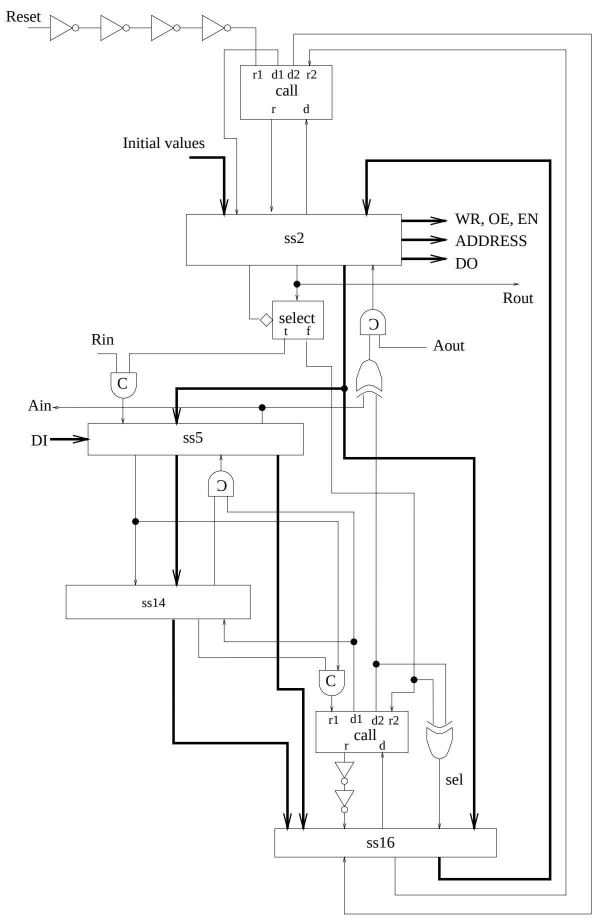


Figure 6.16: A merged two-phase structural Stump processor

As shown in Figures 6.17, 6.18 and 6.19 the performance of the synthesized circuits is improved. However, the speed of the merged four-phase fully decoupled and long hold Stump processors are decreased. The long hold Stump processors has low power consumption. The two-phase one is the fastest.

Circuit	2-phase	4-p simple	4-p semi	4-p fully	4-p long
Energy	522.37	558.77	613.95	417.46	404.62

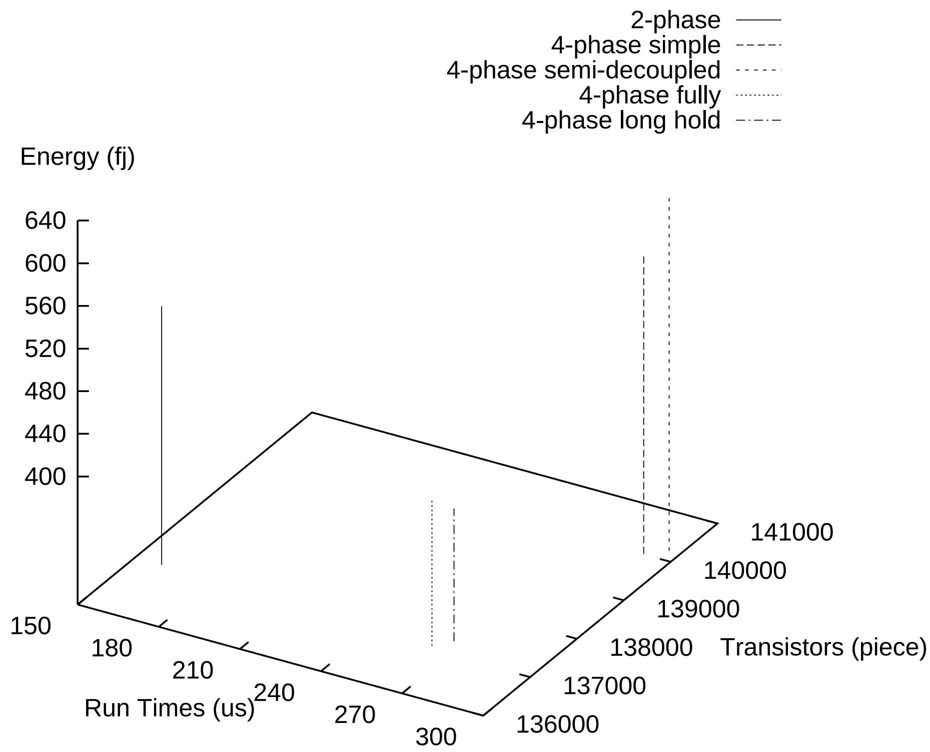


Figure 6.17: The power information of the merged Stump processors

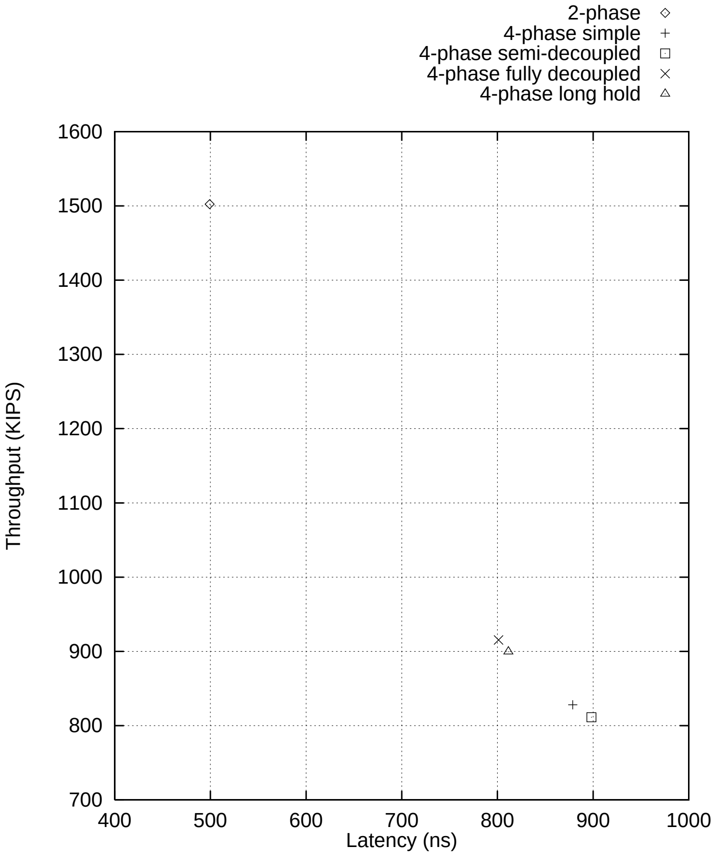


Figure 6.18: The latency and throughput of the merged Stump processors

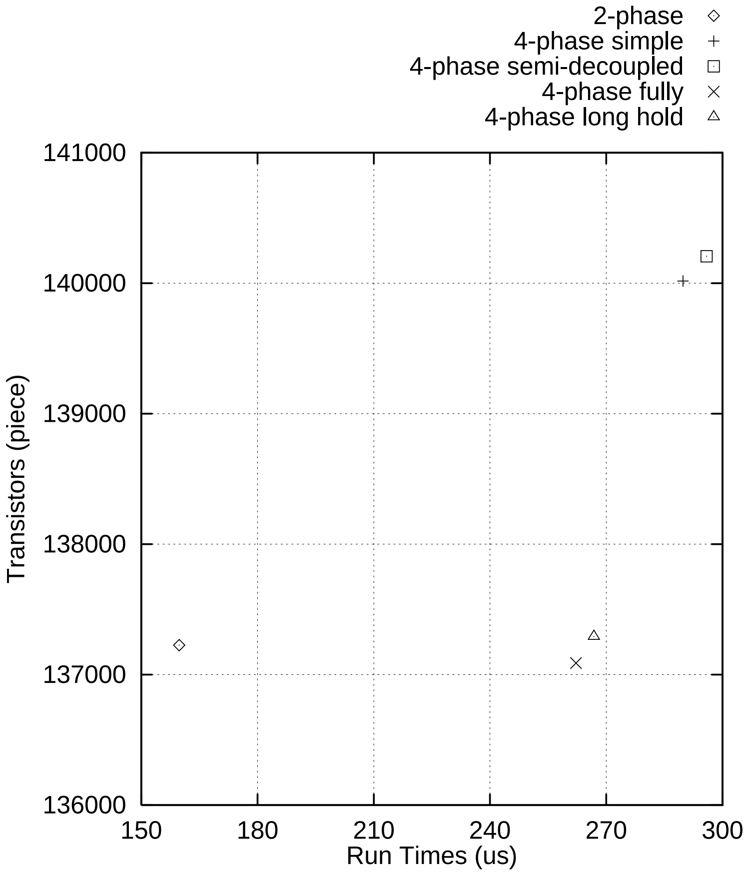


Figure 6.19: The run time of the merged Stump processor

The synthesized Stump processors using the four-phase simple control latch circuit and the four-phase semi-decoupled control circuit have 5 stages. The synthesized Stump processors using the four-phase fully decoupled control circuit and the four-phase long hold control circuit only have 4 stages. For the comparison Table 6.6 shows the performance of the 5-stage fully decoupled Stump processor and the 5-stage long hold Stump processor.

Circuit Name	4-p simple	4-p semi	4-p fully	4-p long	4-p fully 5 stages	4-p long 5 stages
Energy	558.77	613.95	417.46	404.62	481,38	442.08

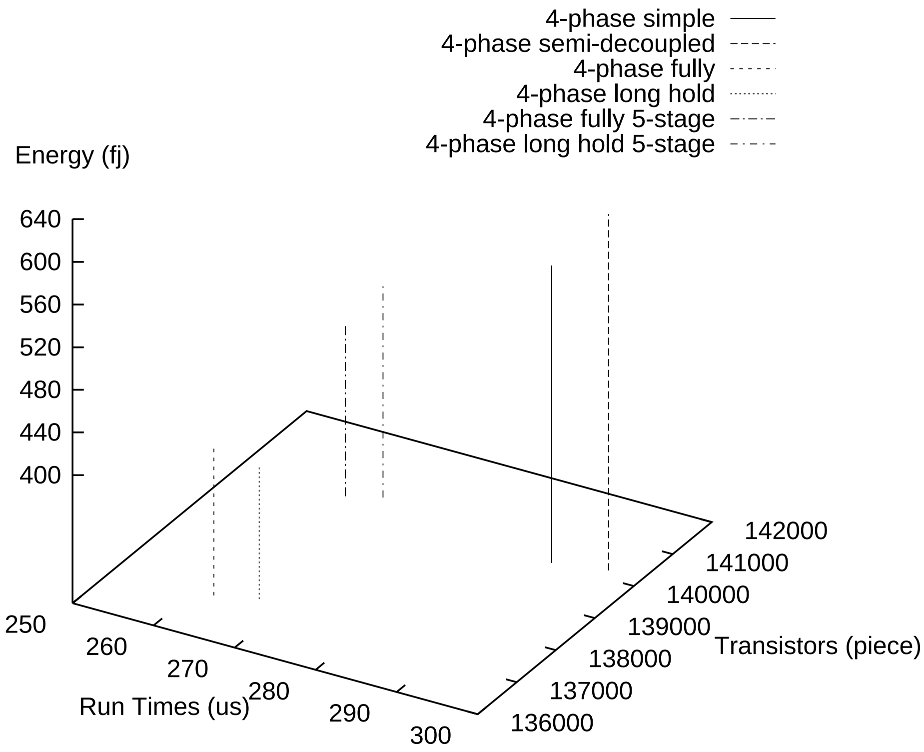


Figure 6.20: The power information of the 4-phase merged Stump processors

As shown in Figures 6.20 and 6.21 the speed of the merged four-phase Stump processors using the simple and semi-decoupled control circuits is faster than the

Table 6.6: The performance of the Stump processors after merge(2)

Circuit Name	Transistors (piece)	Run Time(μs)	Throughput (KIPS)	Latency (ns)	Energy (fj)
2-phase	137226	159.8	1502.3	499.2	522.37
4-p Simple	140016	289.8	828.1	878.7	558.77
4-p Semi	140206	295.9	811.2	898.2	613.95
4-p Fully	137088	262.2	915.3	801.2	417.46
4-p Long	137292	266.8	899.5	811.5	404.62
4-p Fully 5	140252	263.2	911.8	801.8	481.38
4-p Long 5	140466	266.8	899.5	812.0	442.08

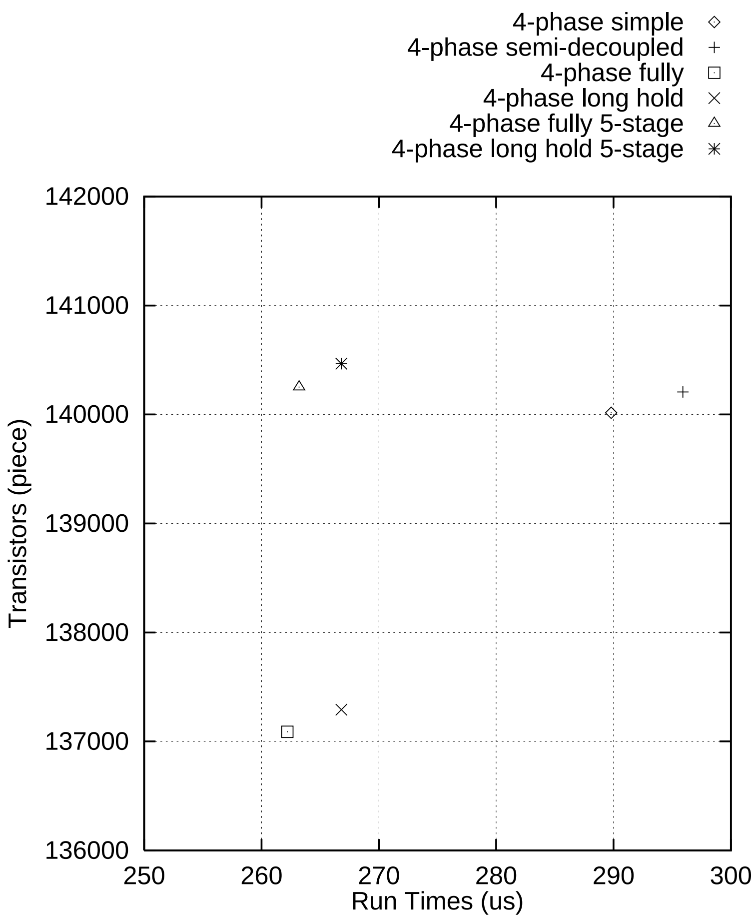


Figure 6.21: The performance of the 4-phase merged Stump processors

original synthesized Stump processors. However, the speed of the merged four-phase Stump processors using the fullly decoupled and long hold control circuits is slower than the speed of the original synthesized Stump processors. There is no much difference between the 5-stage and 4-stage Stump processors using the fully decoupled and long hold control circuits in speed. The 5-stage Stump processors using the fully decoupled and long hold control circuits require more transistors than the merged four-phase Stump processors using the simple and semi-decoupled control circuits. Anyway the merged four-phase Stump processors are cheaper than the original synthesized Stump processors and have lower power consumption.

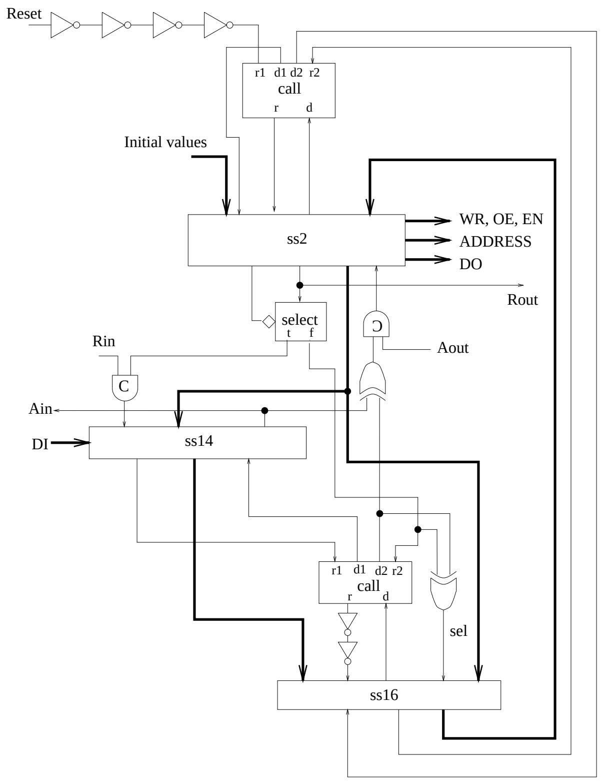


Figure 6.22: A further merged two-phase structural Stump processor

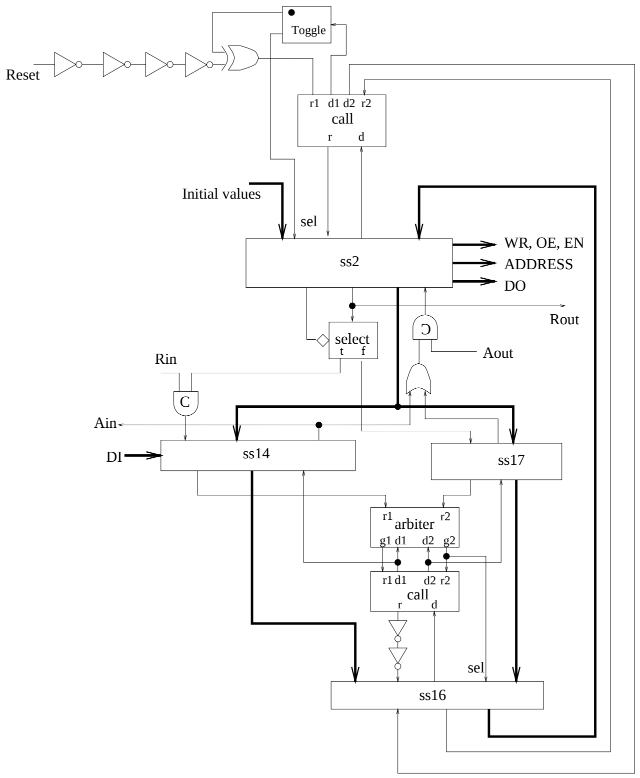


Figure 6.23: A merged four-phase structural Stump processor

If a modification is made in the behavioural description the processor can be further merged as shown in Figure 6.22. If some statements are added after the corresponding **ELSE** of the **false** of **Select** module, the four-phase Stump processors using the simple latch control circuit can also be further merged. The 4-phase Stump processors using the simple and semi-decoupled latch control circuits can be further merged as shown in Figure 6.23. The performance of these further merged Stump processors are shown in Table 6.7, Figures 6.24, 6.25 and 6,26. The run time of the 2-phase Stump processor is about 155.1 μs. The 4-phase long hold circuit gives low power consumption.

Table 6.7: The performance of the Stump processors after optimization

Circuit Name	Transistors (piece)	Run Time(μs)	Throughput (KIPS)	Latency (ns)	Energy (fj)
2-phase before	137226	159.8	1502.3	499.2	522.37
4-p Simple before	140016	289.8	828.1	878.7	558.77
4-p Fully before	137088	262.2	915.3	801.2	417.46
4-p Long before	137292	266.8	899.5	811.5	404.62
2-phase after	134286	155.1	1547.9	484.5	461.59
4-p Simple after	137128	240.4	998.1	751.4	434.85
4-p Fully after	134152	244.6	981.1	764.5	365.19
4-p Long after	134342	248.5	965.6	776.7	362.33

Circuit Name	2-p merged	4-p fully merged	2-p modified modified	4-p simple modified	4-p fully modified	4-p long modified
Energy	522.37	417.46	461.59	434.85	365.19	362.33

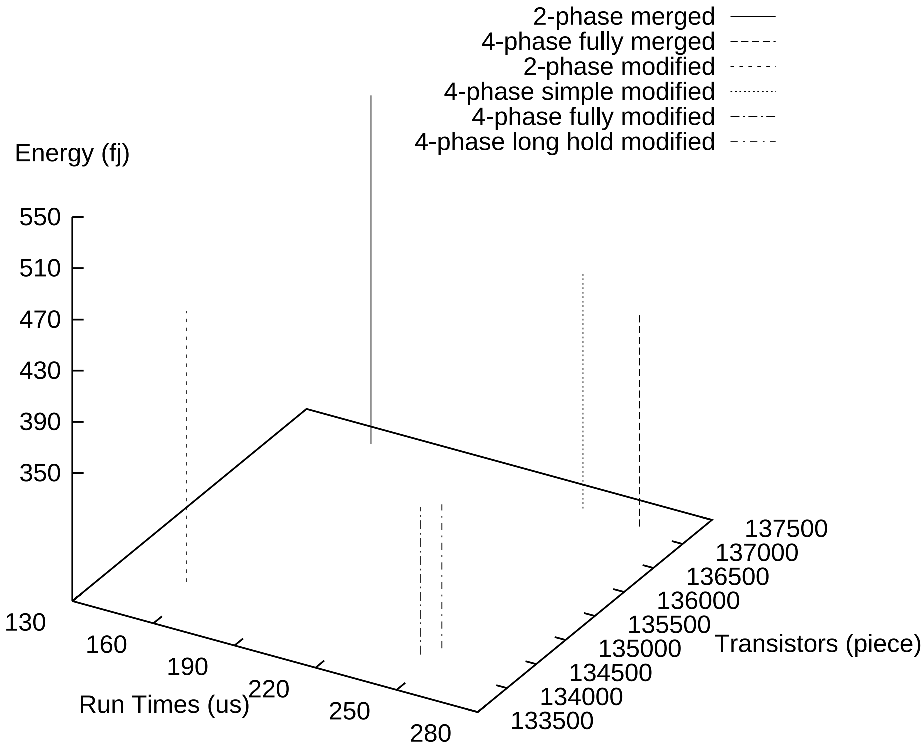


Figure 6.24: The power information of the further merged Stump processors

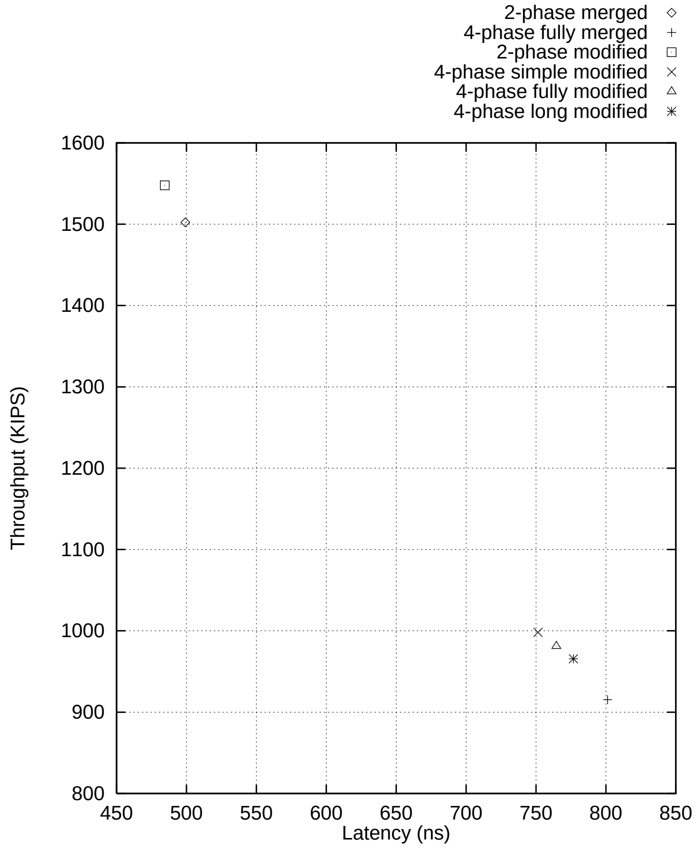


Figure 6.25: The latency and throughput of the further merged Stump processors

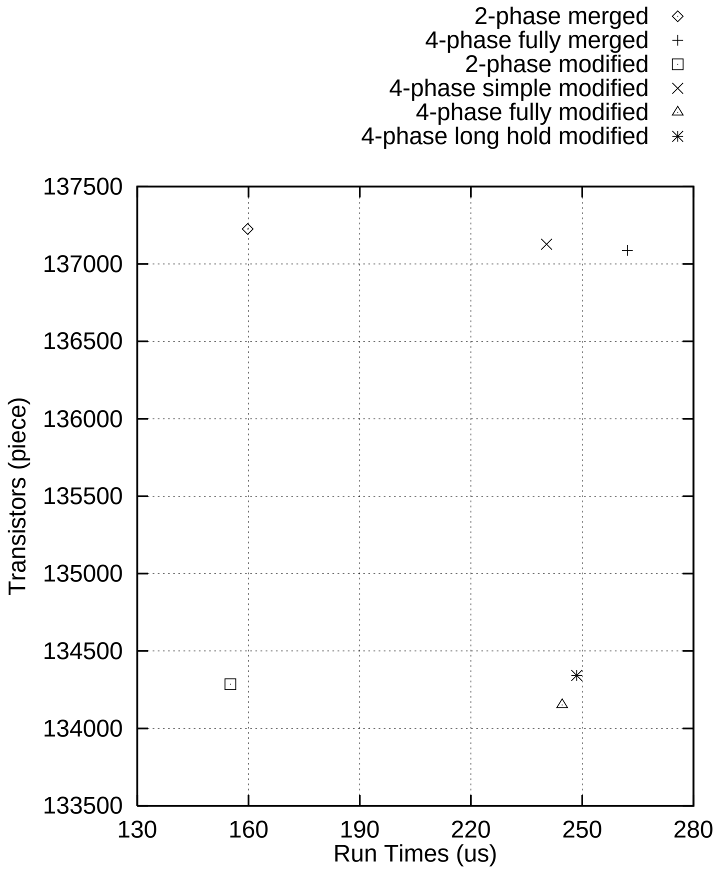


Figure 6.26: The run time of the simple Stump processor

6.6 Discussion

6.6.1 Verilog Simulation

To successfully simulate the synthesized circuits in Verilog all the Library Cells must have Verilog Models. If there is no Verilog Model the circuit will expand down to the transistor level (the lowest level). This transistor level is built in the Verilog Language, and is just a switching gate. For some simple cases, like an inverter, there is no simulation problem. However, most other cases such as circuits having feedback, cannot be simulated. So the general practice is that all the Library Cells should have Verilog Models. Otherwise, you cannot use the Verilog Simulator; instead you can use an other tool such as TimeMill.

6.6.2 Merge problem

Some notes about the data signals which are merged in stage B_4 of Figure 2.14 are given as follows:

1. It is easier to identify that data signals come from either stage B_2 or stage B_3 if dual rail encoding is applied. The select signal of the multiplexers in stage B_4 is also easy to generate.

2. It is necessary to use the control signals to control the multiplexers to switch the data signals from stage B_2 or stage B_3 if single rail encoding and the bundled data convention are used.

3. The control signals are usually slower than the data signals. However, some delays may be added between the multiplexers and the latches to ensure that the the correct data are caught.

4. In order to obtain a faster circuit two identical computation and latch

circuits may exist in stage B_4. Then the control signals are used to select the output to be used. However, the cost grows due to there being two identical circuits.

5. The main reason to use a selective connection as shown in Figure 2.14 is that the difference between the processing times on the two paths is great. Otherwise, **fork-join** connections or multiplexers are used.

6.6.3 Fan-out

It is impossible for a device to produce an infinite output current. Thus, there is a limitation of each device (i.e. the “fan-out”), which represents how many output devices a device is able to connect to. In particular, the **‘lt’** signal is often connected to a lot of latches within a stage. The parameters can be set in the Device objects of the circuit model which is constructed by the synthesizer. Checking the number of devices connected to each component output can find ”fan-out” errors. Adding buffer components can remove these errors. It is necessary to ensure that there is no “fan-out” error in the component library. Otherwise, the layout of the synthesized circuit will have ”fan-out” errors. However, postlayout simulation using TimeMill can find this kind of error.

6.7 Analysis

Relative to the original synthesized Stump processors, optimization reduces the transistor counts by 19.5% $\sim$ 21.6%. The power saved is about 66.1% $\sim$ 69.8%. The run time of the two-phase and the simple control Stump processors have an improvement of 29.5% and 36.6%. The throughput of the two-phase and the simple control Stump processors have an improvement of 41.9% and 57.6%. The latency of the two-phase and the simple control Stump processors is reduced by

29.5% and 34.6%. Figures 6.27, 6.28, 6.29, 6.30 and 6.31 show the performance of different synthesized circuits. They show that the two-phase circuits have good performance.

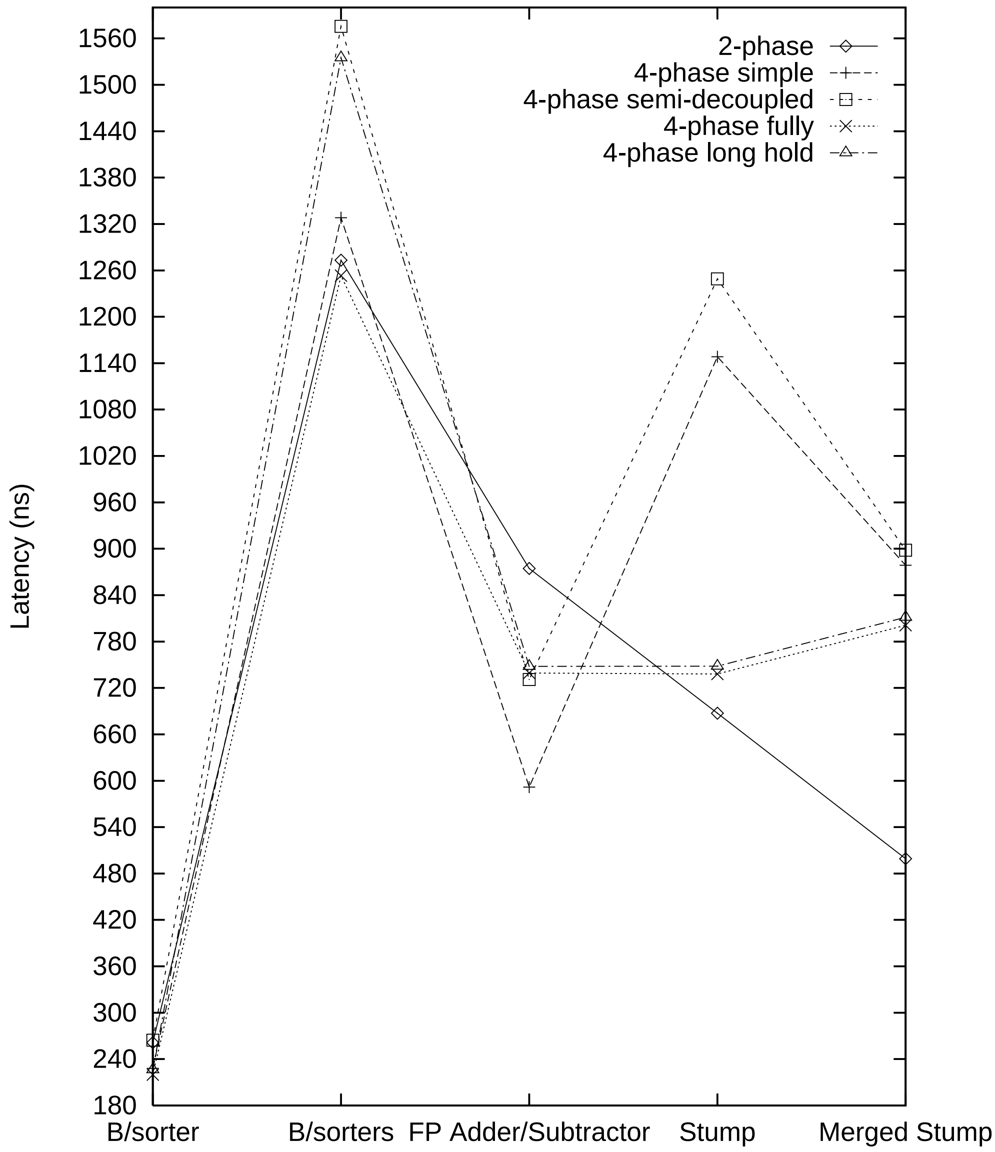


Figure 6.27: The latencies of different control circuits

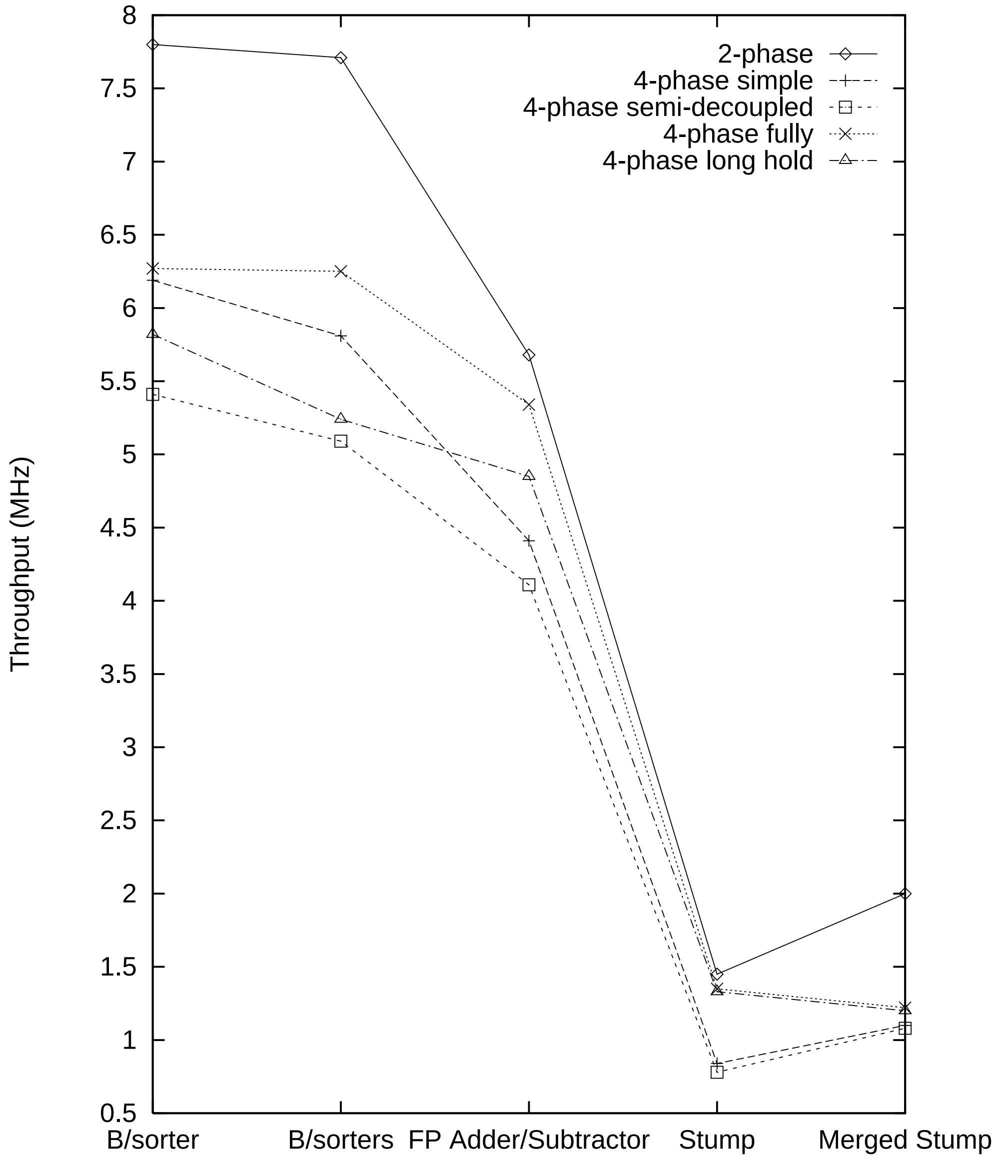


Figure 6.28: The throughputs of different control circuits

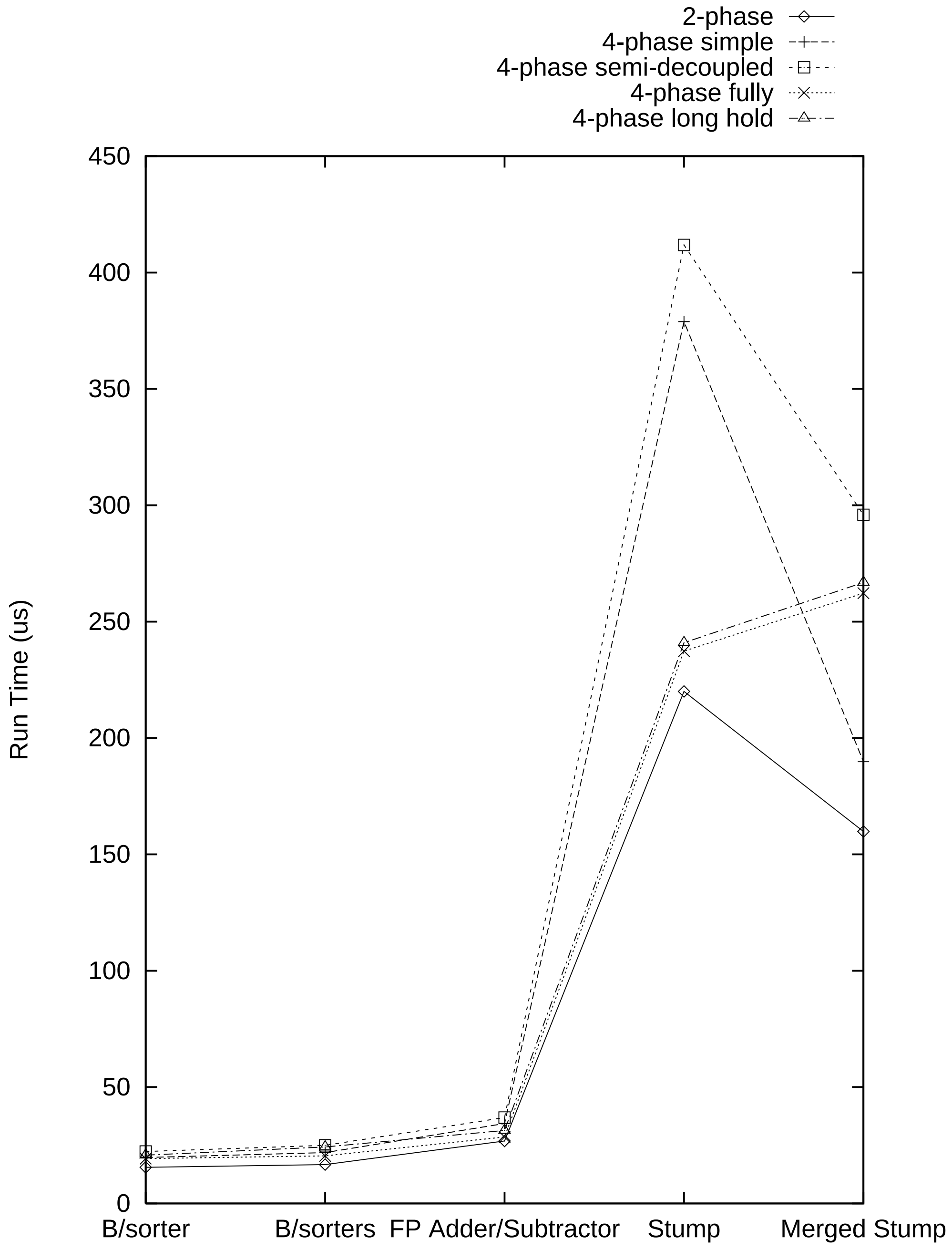


Figure 6.29: The run times of different control circuits

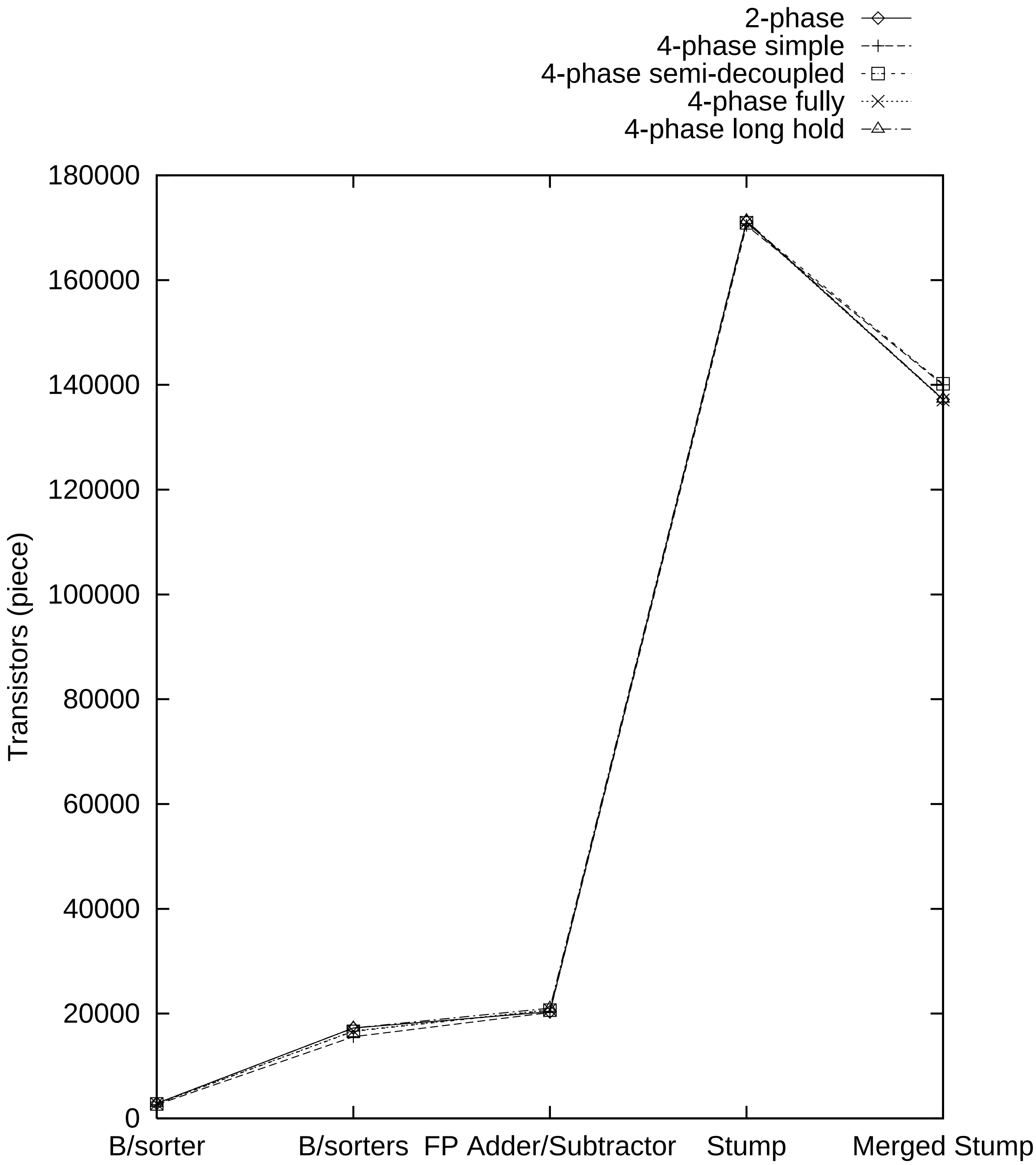


Figure 6.30: The transistors of different control circuits

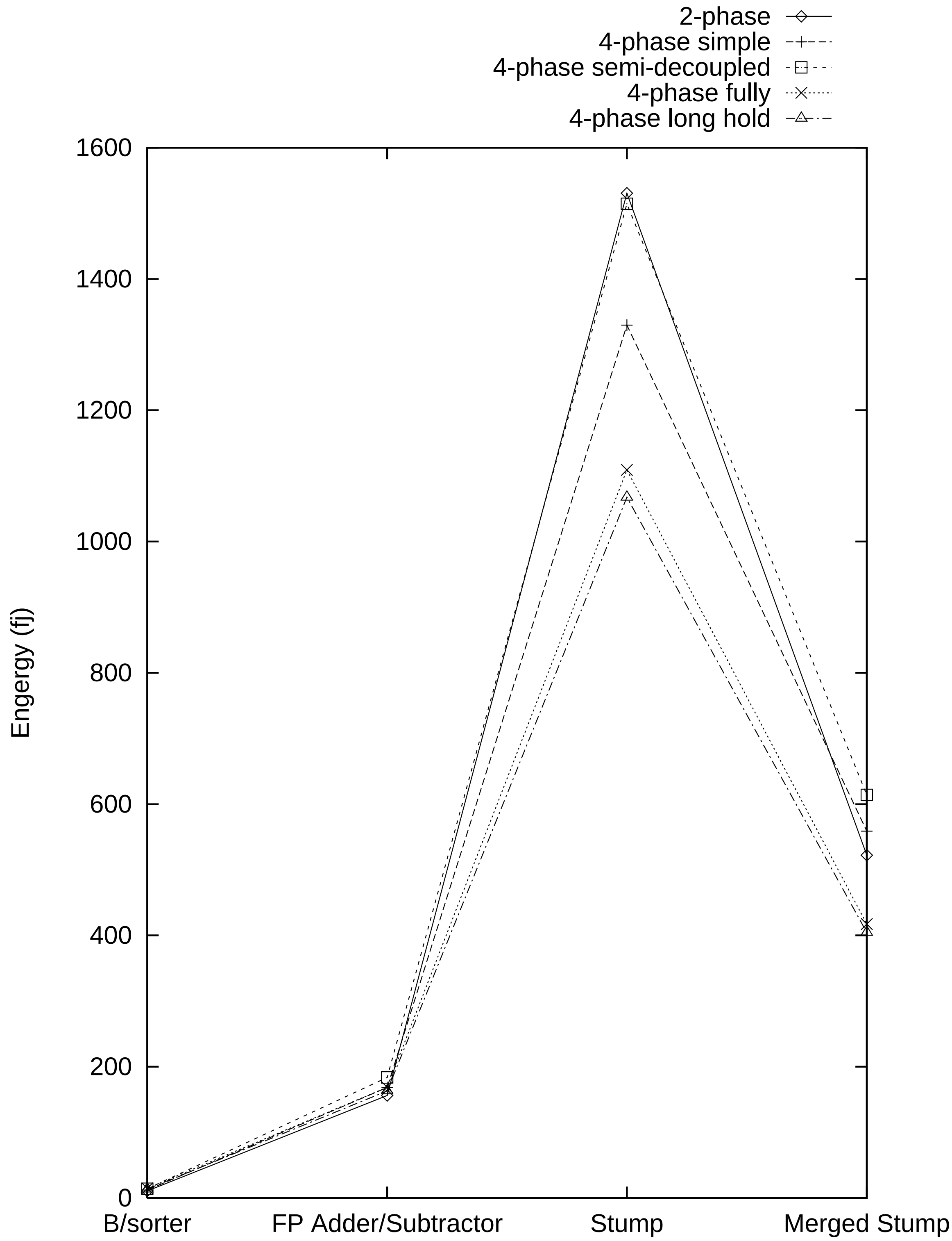


Figure 6.31: The energy of different control circuits

Figures 6.32, 6.33, 6.34, 6.35 and 6.36 show the comparison of the performance of two-phase circuits and the best performance of the four-phase circuits. It is very clear that the two-phase micropipeline circuits offer better performance than the four-phase designs. However, the four-phase circuits have the potential to have a good performance as shown in Figure 5.6. The author would like to further study techniques for the optimization of the four-phase control circuits.

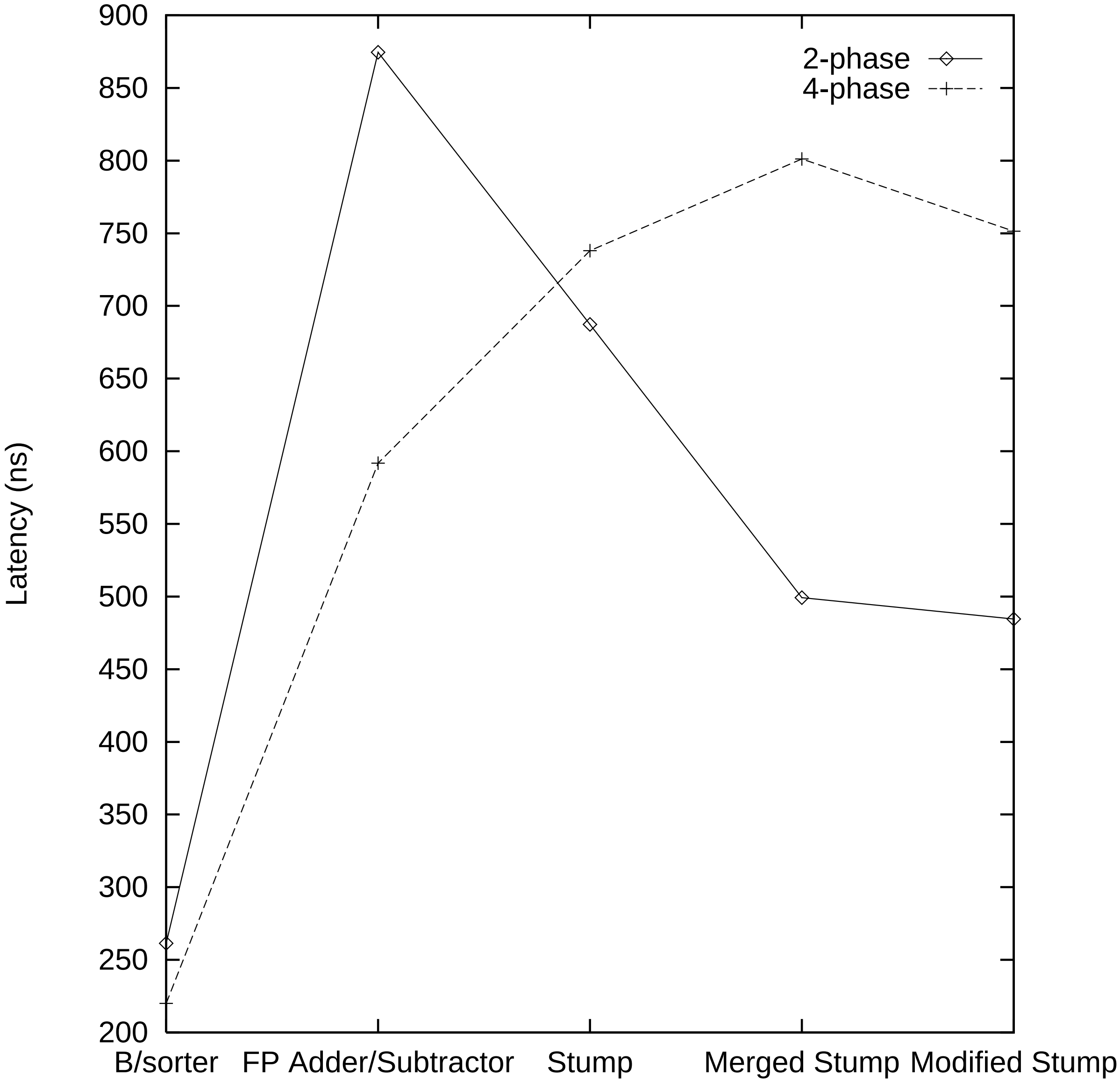


Figure 6.32: The latencies of different control circuits

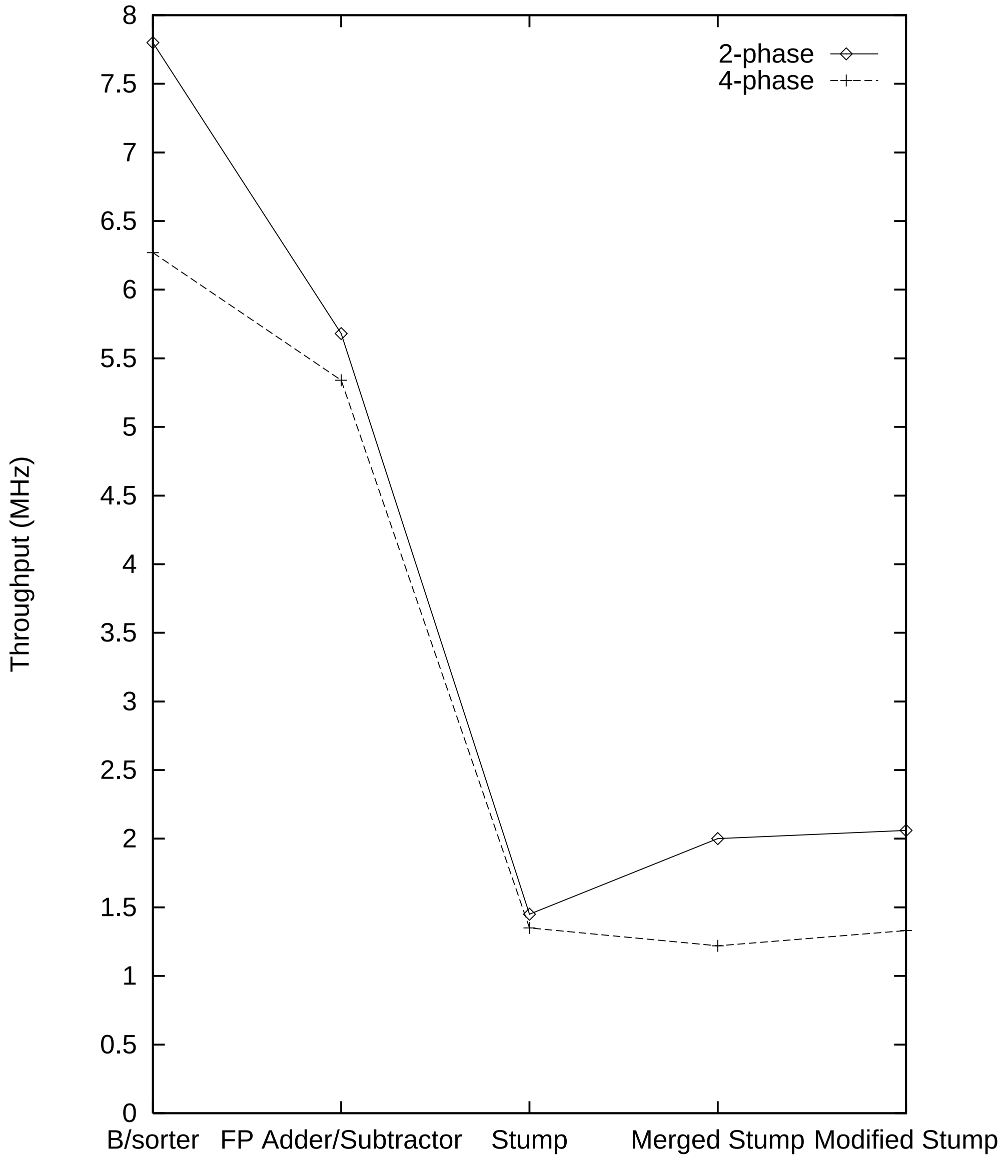


Figure 6.33: The throughputs of different control circuits

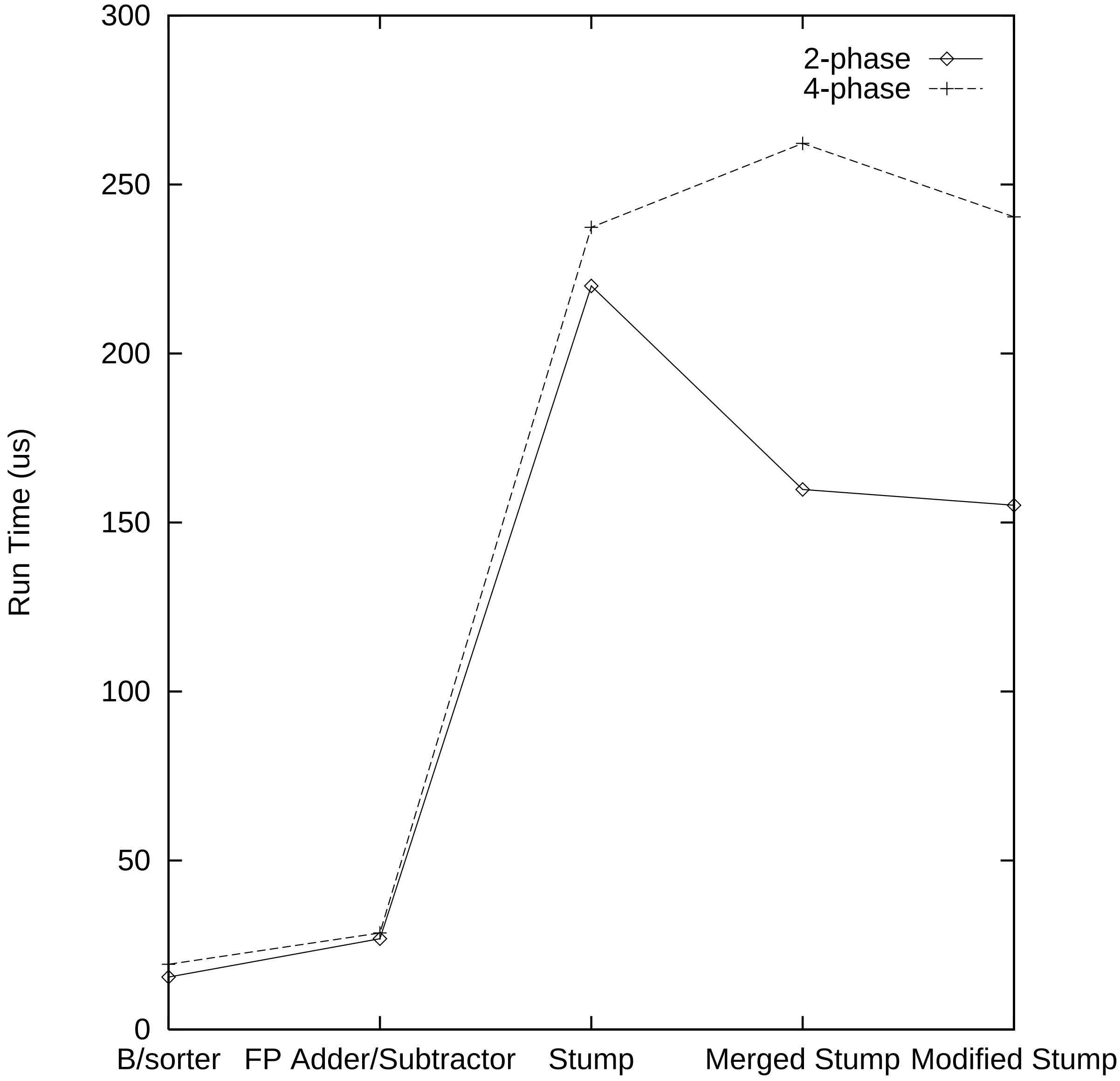


Figure 6.34: The run times of different control circuits

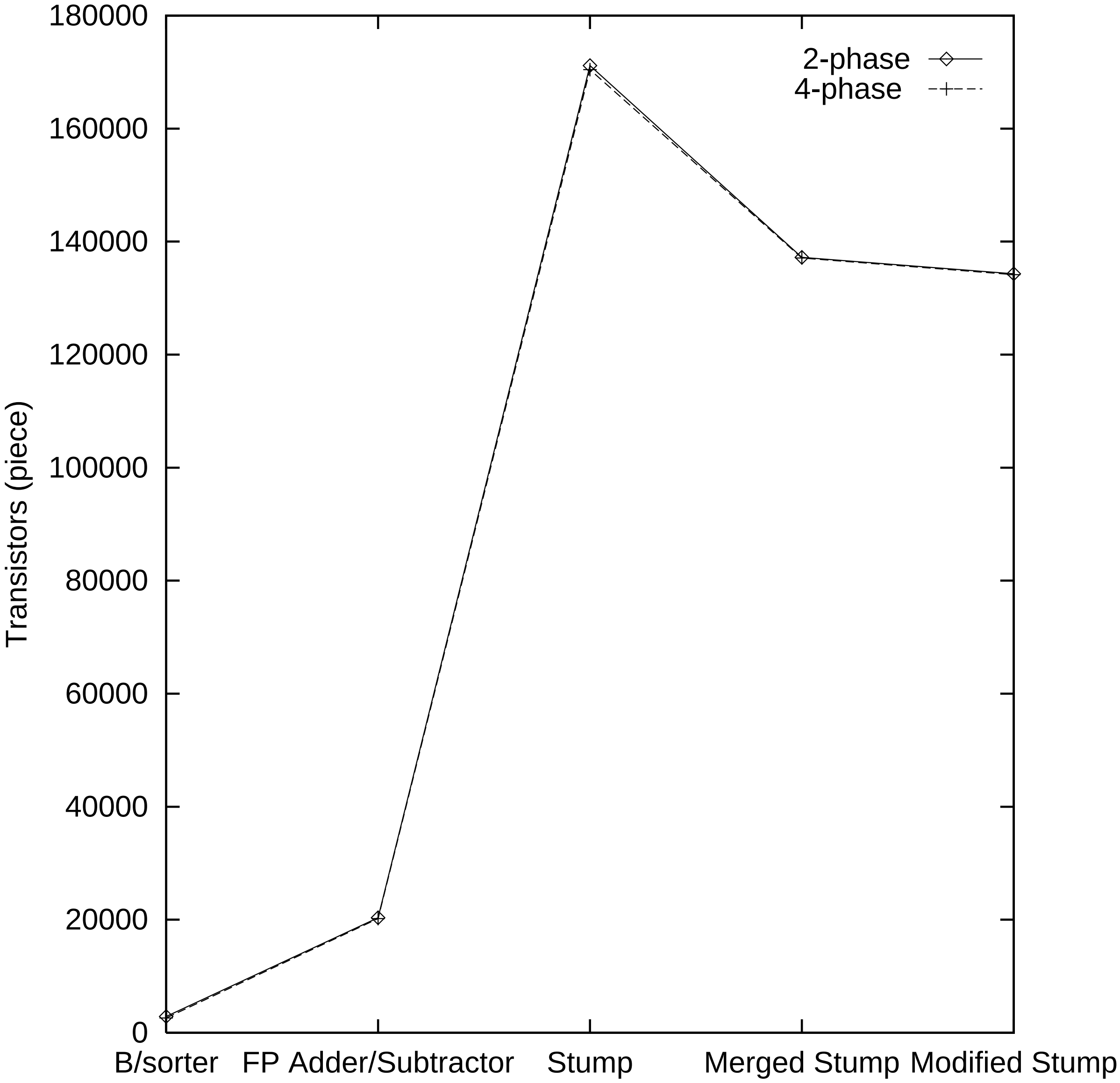


Figure 6.35: The transistors of different control circuits

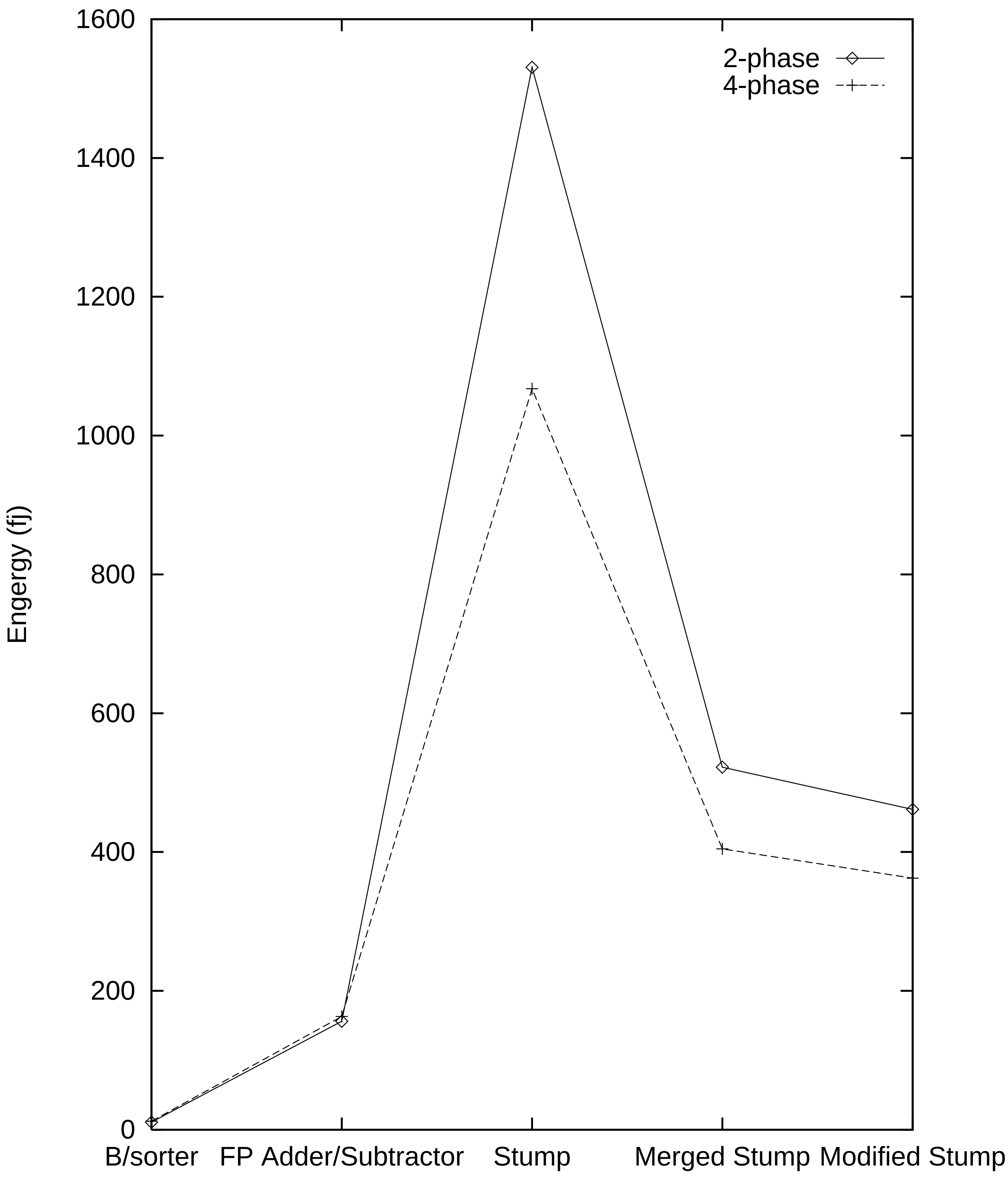


Figure 6.36: The energy of different control circuits

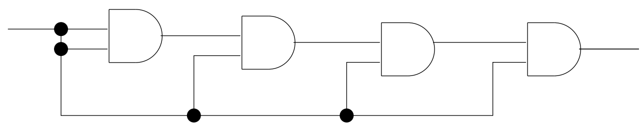


Figure 6.37: The asymmetric delay with faster reset

The experimental results show that the two-phase circuits have good speed performance. This is due to the rising and falling transitions of the 4-phase circuits following the same routes. To improve performance asymmetric delays can be built using the circuit shown in Figure 6.37.

Only one instruction flows through the pipeline of the Stump processors shown in Figures 5.29 and 5.31. This is why the performance of the experimental results was only 1.55 MIPS. However, this meets the requirement of the behavioural description. Further investigation is required to ensure that multiple instructions are able to flow through the synthesized Stump pipelines.

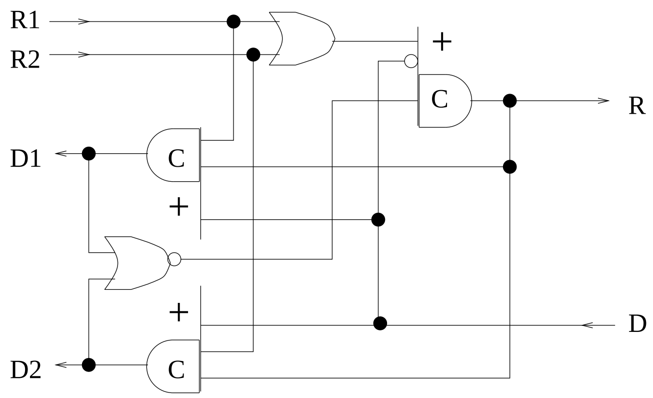


Figure 6.38: A four-phase Call module [Liu97]

The 4-stage circuit using the four-phase simple and semi-decoupled latch control shown in Figure 6.23 required an Arbiter at the input to the top Call module if the **Rout+** of **ss16** is sent out before **d1** becomes logical **'0'** and the Call modules shown in Appendix B and Figure 5.32 are applied. Alternatively, deadlock is avoided if the Call module [Liu97] shown in Figure 6.38 is applied.

6.8 Comparisons with other synthesis methods

6.8.1 Tangram

Handshake circuits described by van Berkel are used as an intermediate representation for asynchronous circuits compiled from Tangram programs based on CSP [Berk93, Scha95]. The circuits generated by this method do not use a micropipeline design style. VHDL is not applied here. In Tangram channels are used for input, output and synchronisation communications. Assignments, skip and stop commands, sequencing and concurrency commands as well as conditional and repetition commands are used for describing the behaviour of a circuit. The Tangram compilation function defines the syntax directed mapping from Tangram commands into their handshake circuit implementation. In the composition of a handshake circuit, a number of handshake components are connected together by channels.

The synthesizer presented in this thesis partitions the VHDL statements into several stages depending on whether statements can be executed concurrently and whether the special control circuit is required. Then the internal circuit models for the datapath are constructed. The suitable control circuits are generated depending on the function for each stage. The interconnections between stages are then constructed. Finally, the structural VHDL and Verilog files of the circuits are produced. Assignments, IF statements, loop statements and basic mathematical and logical operations are available. The behavioural simulation can be performed before synthesis.

6.8.2 Balsa

Balsa is a CSP-like language based on the Tangram VLSI programming language [Bard96, Bard97, Bard98, Bard20]. Balsa is an extension of Tangram. Balsa can be compiled into handshake circuits expressed in the Breeze intermediate language. Then Breeze can be transformed into a gate level netlist (a structural Verilog netlist of standard cells). The gate level netlist can be imported into the Cadence Design Framework II database. Gate level simulation and final silicon production is carried out within Cadence. Breeze can also be transformed into the LARD [Ende98, Ende98a] modelling language to allow behavioural simulation. The circuits generated by this method are also not micropipelined design style. VHDL is also not used. There is no tool for Balsa simulation. Attention is also required to ensure that the Balsa program and its LARD model are consistent.

A Balsa Stump processor was synthesized. However, the author lacks experience on the Balsa tool. The performance is summarised as follows:

```
For 240 instructions:

  Run time:      2.078 sec.
  Throughput:    115.5 KIPS
  Latency:       6497.9 ns
  Transistors:   191164
  Energy:        706.84 fj
```

The speed of the Balsa Stump processor is quite slow. However, the parallel operator, || can be used to improve the performance. Further investigation has undertaken. Some components used for the Balsa Stump processor have to be **reset** before the simulation starts if the simulation is run on the VHDL simulator. Otherwise, those undefined values may cause the simulation goes wrong.

6.8.3 Synopsys

A synchronous version Stump processor was synthesized by the Synopsys tool. The performance is summarised as follows:

```
For 240 instructions:

  Clock rate: 12.5 MHz

  Run time:      153.6 us
  Throughput:    1.56 MIPS
  Latency:       480.0 ns
  Transistors:   45112
  Energy:        902.90 fj
```

The speed of the synchronous Stump processor is slightly better than the two-phase Stump processor. The two-phase and four-phase Stump processors have better power consumption than the synchronous Stump processor. The synchronous Stump processor is very cheap and has very low latency.

When all statements used for synthesizing the two-phase and four-phase Stump processors were placed into only one **process**. Then the synthesized synchronous processor could not work correctly. After placing the statements into 5 different **process**es and generating 5 different clocks for each **process** the correct synchronous Stump processor was synthesized successfully from the Synopsys design_analyzer.

6.8.4 Other CSP based methods

The circuits generated by Burns's synthesis method of transforming concurrent programs based on a variant of CSP into self-timed circuits is also not micropipeline design style [Burn88]. OCCAM is used as the input language in Brunvand's translation and transformation method [Brun89, Brun91, Brun91a]. OCCAM(async) is used as the input language on Nedelchev's silicon compiler which converts a high-level programming language to asynchronous CMOS circuits [Nede95].

Chapter 7

Conclusions

This thesis has presented some engineering work on developing a micropipeline synthesizer. The work presented in this thesis demonstrates that VHDL can be used to describe the behaviour of micropipelined systems. It is also suitable for representing the circuits output by 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. In order to synthesize correct circuits the input description must be correct first. The simulation mechanism of VHDL is able to assist in the discovery of potential design errors at an early stage. Though the nature of the work is mainly engineering, there are some significant new insights gained in the course of the work.

7.1 Contributions

The partition rules, conversion rules and rules of generating control circuits were derived. VHDL models of event-driven logic modules and some standard logic function elements were created using standard logic gates. Then a synthesizer which converts the behavioural VHDL into 2-phase and 4-phase structural

VHDL and VERILOG in a micropipeline style has been implemented using C++. The VHDL output is to verify the design against its behavioural VHDL using the VHDL simulator. The VERILOG output is to import into the Cadence database for the subsequent design work. A design flow for the construction of micropipeline circuits using the synthesizer was proposed.

A comparison of 2- and 4-phase micropipeline circuits in VHDL was presented. From the simulations of the various configurations some insight is obtained. It is summarized as follows:

- Two-phase circuits have good performance on speed. When a concurrent reset is not applied on the four-phase circuits and the rising and falling transitions follow the same routes as the two-phase circuits, it is more complex to build the circuits to ensure both the rising and falling transitions can flow through the control path correctly. On the other hand, asymmetric C-gates are useful for building circuits for the selection signals of the multiplexers.
- Four-phase circuits have better performance on power consumption. Four-phase circuits using the long-hold latch control circuit are best.
- In **resource sharing** the two-phase implementations have better performance than the four-phase implementations. There is no **return to zero** problem. **fork** and **join** cost nothing to the two-phase implementations. Add some addition buffer stages the four-phase implementations using the fully decoupled and long hold latch control circuits can also implement **resource sharing**. However, the four-phase implementations using the simple and semi-decoupled latch control circuits require more buffer stages to avoid deadlock.
- When the stages are connected in a feed back loop the number of stages can be 2 for the two-phase circuits and the four-phase circuits using the fully

decoupled and long hold latch control circuits. However, the minimum number of stages is 3 for the four-phase circuits using the simple and semi-decoupled latch control circuits.

Both VHDL simulation and postlayout simulation using TimeMill for the synthesized circuits obtained correct results. Due to the synthesized circuits containing both data paths and control circuits, delays must be added automatically on the control path to ensure that the data signals are faster than the control signals.

A VHDL design environment for micropipelines has been established. The VHDL behaviour may be complex for a bigger system. However, it is very easy to simulate the behavioural VHDL model using a VHDL simulator at an early design stage.

Experimental results show that:

Stump processor

The fastest speed is 1.55 MIPS for the two-phase synthesized processor and the lowest power consumption is 362.33 fj for the synthesized four-phase long hold processor.

14-bit floating point adder/subtractor

The fastest speed is 5.68 MHz and the lowest power consumption is 156.12 fj for the two-phase floating point adder/subtractor. The floating point adder/subtractor using the four-phase simple latch control circuit has the lowest the transistor count.

Blocksorter

The fastest speed is 7.80 MHz and the lowest power consumption is 10.78 fj for the two-phase synthesized Blocksorter. Blocksorter using the four-phase simple latch control circuit has the lowest the transistor count.

Different merged and shared resource techniques were presented to demonstrate the properties of the two-phase and four-phase circuits with different latch control circuits.

7.2 Future work

The experimental results show that the two-phase circuits have good performance in speed. This is due to the rising and falling transitions of the 4-phase circuits following the same routes. To improve the performance asymmetric delays can be built using the circuit shown in Figure 6.37. The loop control circuit shown in Figure 5.6 will be applied in future investigations.

Only one instruction flows through the pipelines of the Stump processors shown in Figures 5.29 and 5.31. This it why the performance shown in the experimental results was as low as 1.55 MIPS. However, it meets the requirements of the behavioural description. Further investigation is required to ensure that multiple instructions are able to flow through the synthesized Stump circuits.

The author believes that the results in this thesis show that VHDL can be helpfully used in various design stages. This section briefly describes some important research problems addressed by the author.

7.2.1 Test vectors

Due to the fact that the design flow requires simulation at various design levels, such as VHDL behaviour, VHDL structure, schematic and layout, VHDL, Verilog and TimeMill simulators are used for the simulations. A lot of time was spent generating the various test vectors. It is worth studying test pattern generation techniques for various micropipeline systems. We believe that it is necessary in the future to develop an automatic test vector generation system to support the various micropipeline design levels.

7.2.2 Interfacing with other design tools

Figure 3.29 shows the interface between the synthesizer and other tools such as LARD [Ende98, Ende98a], Yellow [Barr96, Barr97], and Balsa [Bard96, Bard97, Bard98]. Further work is required to develop conversion programs between VHDL and these asynchronous hardware description languages.

7.2.3 Standard asynchronous components

Standard components for synchronous design are available on the open market. However, components for asynchronous design, such as C gates, Toggles, ... etc., are not. Researchers have to create their own component libraries before they start their research projects. Creating a set of standard components with different views is helpful for the prospects for asynchronous design.

7.2.4 Structural VHDL support

Cadence schematics can be imported from some HDLs, for example Verilog. VHDL is an IEEE standard, but Cadence doesn't support a VHDL-In function to import structural VHDL into the Cadence database. The only use for structural VHDL is for simulation. It is necessary to develop techniques to support importing structural VHDLs into other design systems.

7.3 Fully delay-insensitive implementation

A difficult engineering problem with the synthesis method presented in this thesis is the insertion of delays in the control path. In some cases the delay may reduce the performance of the circuits. For example, the four-phase fully decoupled circuits have high speed performance, but if unsuitable delay is added on the control path the performance of the circuits becomes poor. Therefore, a dual-rail technique could be added as an option in the data path implementation to synthesize fully delay-insensitive pipelines, thereby avoiding this problem.

Appendix A

VHDL models

A.1 A C gate

```
library ieee;  use ieee.std_logic_arith.all;  use ieee.std_logic_1164.all;
ENTITY c2 IS PORT ( R, A, B : IN std_logic; Y: OUT std_logic ); END c2;
architecture BEHAVIORAL of C2 is
    component nand3 PORT ( A, B, C: IN std_logic; Y: OUT std_logic ); end component;
    for all: nand3 use entity work.nand3(behavioral);
    signal s1, s2, s3, s4 : std_logic;
begin
    Y <= s4;
    u1: nand3 PORT map ( A=> R, B => A, C => B, Y => s1 );
    u2: nand3 PORT map ( A=> R, B => B, C => s4, Y => s2 );
    u3: nand3 PORT map ( A=> R, B => A, C => s4, Y => s3 );
    u4: nand3 PORT map ( A=> s1, B => s2, C => s3, Y => s4 );
end BEHAVIORAL;
```

A.2 A XOR2 gate

```
library ieee;  use ieee.std_logic_arith.all;  use ieee.std_logic_1164.all;
ENTITY XOR2 IS PORT (A, B : IN std_logic; Y: OUT std_logic); END XOR2;
architecture BEHAVIORAL of XOR2 is
begin Y <= A xor B after 4.2 NS;
end BEHAVIORAL;
```

A.3 An OR2

```
library ieee;  use ieee.std_logic_arith.all;  use ieee.std_logic_1164.all;
ENTITY OR2 IS PORT (A, B: IN std_logic; C: OUT std_logic); END OR2;
architecture BEHAVIORAL of OR2 is
begin C <= A or B after 2.4 NS;
end BEHAVIORAL;
```

A.4 A 2-phase Toggle

```
library ieee;  use ieee.std_logic_arith.all;  use ieee.std_logic_1164.all;
ENTITY toggle2 IS PORT ( R, I : IN std_logic; Yd, Yn: OUT std_logic ); END toggle2;
architecture BEHAVIORAL of toggle2 is
    component nand2 PORT ( A, B: IN std_logic; Y: OUT std_logic ); end component;
    for all: nand2 use entity work.nand2(behavioral);
    component nand3 PORT ( A, B, C: IN std_logic; Y: OUT std_logic ); end component;
    for all: nand3 use entity work.nand3(behavioral);
    component nand4 PORT ( A, B, C, D: IN std_logic; Y: OUT std_logic ); end component;
    for all: nand4 use entity work.nand4(behavioral);
    signal s1, s2, s3, s4, s5, s6 : std_logic;
begin
    Yn <= s1;   Yd <= s6;
    u1: nand2 PORT map ( A=> s4, B => s4, Y => s1 );
    u2: nand3 PORT map ( A=> s1, B => R, C => I, Y => s2 );
    u3: nand4 PORT map ( A=> s4, B => R, C => I, D => s4, Y => s3 );
    u4: nand2 PORT map ( A=> s3, B => s6, Y => s4 );
    u5: nand3 PORT map ( A=> R, B => s2, C => s6, Y => s5 );
    u4: nand2 PORT map ( A=> s5, B => s3, Y => s6 );
end BEHAVIORAL;
```

A.5 A 4-phase Toggle

```
library ieee;  use ieee.std_logic_arith.all;  use ieee.std_logic_1164.all;
ENTITY toggle2_4 IS PORT ( R, I : IN std_logic; Yd, Yn: OUT std_logic ); END toggle2_4;
architecture BEHAVIORAL of toggle2_4 is
    component nand2 PORT ( A, B: IN std_logic; Y: OUT std_logic ); end component;
    for all: nand2 use entity work.nand2(behavioral);
    component nand3 PORT ( A, B, C: IN std_logic; Y: OUT std_logic ); end component;
    for all: nand3 use entity work.nand3(behavioral);
    component nor2 PORT ( A, B, C, D: IN std_logic; Y: OUT std_logic ); end component;
    for all: nor2 use entity work.nor2(behavioral);
    component not1 PORT ( A: IN std_logic; B: OUT std_logic ); end component;
    for all: not1 use entity work.not1(behavioral);
    signal s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, s14 : std_logic;
begin
    Yn <= s14;   Yd <= s13;
    u1: nand2 PORT map ( A=> I, B => s2, Y => s1 );
    u2: nand3 PORT map ( A=> s1, B => R, C => I, Y => s2 );
    u3: nand3 PORT map ( A=> s2, B => I, C => s4, Y => s3 );
    u4: nand3 PORT map ( A=> s3, B => R, C => s11, Y => s4 );
    u5: not1 PORT map ( A=> s10, B => s5 );
    u6: nand3 PORT map ( A=> I, B => s5, C => s12, Y => s6 );
    u7: nand2 PORT map ( A=> I, B => s13, Y => s7 );
    u8: nand3 PORT map ( A=> I, B => s10, C => s12, Y => s8 );
    u9: nand2 PORT map ( A=> I, B => s14, Y => s9 );
   u10: nand2 PORT map ( A=> s2, B => s11, Y => s10 );
   u11: nand3 PORT map ( A=> s10, B => s3, C => R, Y => s11 );
   u12: nor2 PORT map ( A=> s13, B => s14, Y => s12 );
   u13: nand2 PORT map ( A=> s6, B => s7, Y => s13 );
   u14: nand2 PORT map ( A=> s8, B => s9, Y => s14 );
end BEHAVIORAL;
```

A.6 A 2-phase Call

```
library ieee;  use ieee.std_logic_arith.all;  use ieee.std_logic_1164.all;
ENTITY call2 IS PORT ( CL, R1, R2, D : IN std_logic; R, D1, D2: OUT std_logic ); END call2;
architecture BEHAVIORAL of call2 is
    component nand3 PORT ( A, B, C: IN std_logic; Y: OUT std_logic ); end component;
    for all: nand3 use entity work.nand3(behavioral);
    component xor2 PORT ( A, B: IN std_logic; Y: OUT std_logic ); end component;
    for all: xor2 use entity work.xor2(behavioral);
    signal s1, s2, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, s14, s15 : std_logic;
begin
    D1 <= s10;     D2 <= s11;
    u1: xor2 PORT map ( A=> D, B => s11, Y => s1 );
    u2: xor2 PORT map ( A=> s10, B => D, Y => s2 );
    u3: xor2 PORT map ( A=> R1, B => R2, Y => R );
    u4: nand3 PORT map ( A=> CL, B => R1, C => s1, Y => s4 );
    u5: nand3 PORT map ( A=> CL, B => s1, C => s10, Y => s5 );
    u6: nand3 PORT map ( A=> CL, B => R1, C => s10, Y => s6 );
    u7: nand3 PORT map ( A=> CL, B => s2, C => R2, Y => s7 );
    u8: nand3 PORT map ( A=> CL, B => R2, C => s11, Y => s8 );
    u9: nand3 PORT map ( A=> CL, B => s2, C => s11, Y => s9 );
   u10: nand3 PORT map ( A=> s4, B => s5, C => s6, Y => s10 );
   u11: nand3 PORT map ( A=> s7, B => s8, C => s9, Y => s11 );
end BEHAVIORAL;
```

A.7 A 4-phase Call

```
library ieee;  use ieee.std_logic_arith.all;  use ieee.std_logic_1164.all;
ENTITY call2_4 IS PORT ( R1, R2, D : IN std_logic; R, D1, D2: OUT std_logic ); END call2_4;
architecture BEHAVIORAL of call2_4 is
    component not1 PORT ( A: IN std_logic; B: OUT std_logic ); end component;
    for all: not1 use entity work.not1(behavioral);
    component nor3 PORT ( A, B, C: IN std_logic; Y: OUT std_logic ); end component;
    for all: nor3 use entity work.nor3(behavioral);
    component or2 PORT ( A, B: IN std_logic; Y: OUT std_logic ); end component;
    for all: or2 use entity work.or2(behavioral);
    signal s1, s2, s3 : std_logic;
begin
    D1 <= s2;      D2 <= s3;
    u1:not1 PORT map ( A=> D, B => s1 );
    u2: nor3 PORT map ( A=> R2, B => s1, C => s3, Y => s2 );
    u3: nor3 PORT map ( A=> s2, B => s1, C => R1, Y => s3 );
    u4: or2 PORT map ( A=> R1, B => R2, Y => R );
end BEHAVIORAL;
```

A.8 A 4-phase Select

```
library ieee;  use ieee.std_logic_arith.all;  use ieee.std_logic_1164.all;
ENTITY select1_4 IS PORT ( Bool, I : IN std_logic; Yt, Yf : OUT std_logic ); END select1_4;
architecture BEHAVIORAL of select1_4 is
    component nand3 PORT ( A, B, C: IN std_logic; Y: OUT std_logic ); end component;
    for all: nand3 use entity work.nand3(behavioral);
    component not1 PORT ( A: IN std_logic; B: OUT std_logic ); end component;
    for all: not1 use entity work.not1(behavioral);
    component nand2 PORT ( A, B: IN std_logic; Y: OUT std_logic ); end component;
    for all: nand2 use entity work.nand2(behavioral);
    component nor2 PORT ( A, B: IN std_logic; Y: OUT std_logic ); end component;
    for all: nor2 use entity work.nor2(behavioral);
    signal s1, s2, s3, s4, s5, s6, s7, s8 : std_logic;
begin
    Yf <= s7;      Yt <= s8;
    u1: not1 PORT map ( A=> Bool, B => s1 );
    u2: nand3 PORT map ( A=> I, B => s1, C => s6, Y => s2 );
    u3: nand2 PORT map ( A=> I, B => s7, Y => s3 );
    u4: nand3 PORT map ( A=> I, B => Bool, C => s6, Y => s4 );
    u5: nand2 PORT map ( A=> I, B => s8, Y => s5 );
    u6: nor2 PORT map ( A=> s7, B => s8, Y => s6 );
    u7: nand2 PORT map ( A=> s2, B => s3, Y => s7 );
    u8: nand2 PORT map ( A=> s4, B => s5, Y => s8 );
end BEHAVIORAL;
```

A.9 A Comparator1

```
library ieee;  use ieee.std_logic_arith.all;  use ieee.std_logic_1164.all;
ENTITY comparator1 IS PORT ( a0, b0 : IN std_logic; g, e, l : OUT std_logic ); END comparator1;
architecture BEHAVIORAL of comparator1 is
    component not1 PORT ( A: IN std_logic; B: OUT std_logic ); end component;
    for all: not1 use entity work.not1(behavioral);
    component nor2 PORT ( A, B: IN std_logic; Y: OUT std_logic ); end component;
    for all: nor2 use entity work.nor2(behavioral);
    signal s1, s2, s3, s4 : std_logic;
begin
    g <= s3;   l <= s4;
    u1: not1 PORT map ( A=> a0, B => s1 );
    u2: not1 PORT map ( A=> b0, B => s2 );
    u3: nor2 PORT map ( A=> s1, B => b0, Y => s3 );
    u4: nor2 PORT map ( A=> a0, B => s2, Y => s4 );
    u5: nor2 PORT map ( A=> s3, B => s4, Y => e );
end BEHAVIORAL;
```

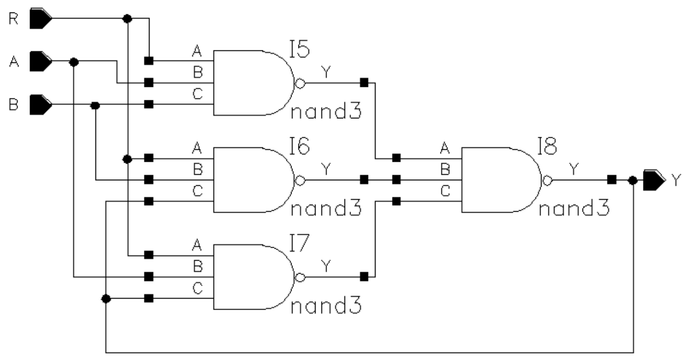
Appendix B

Component library

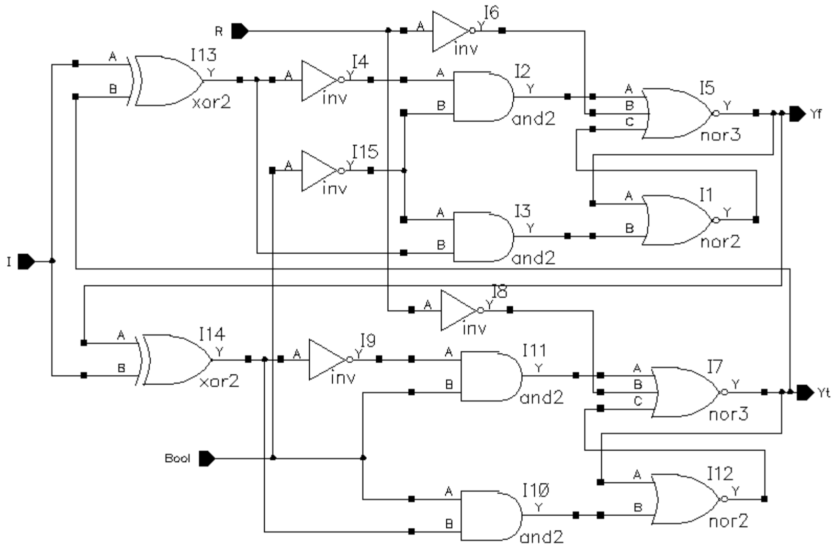
Components [Pave94][Liu97] presented in this appendix are in Cadence library and used for generating Hspice netlists.

B.1 2-phase control modules

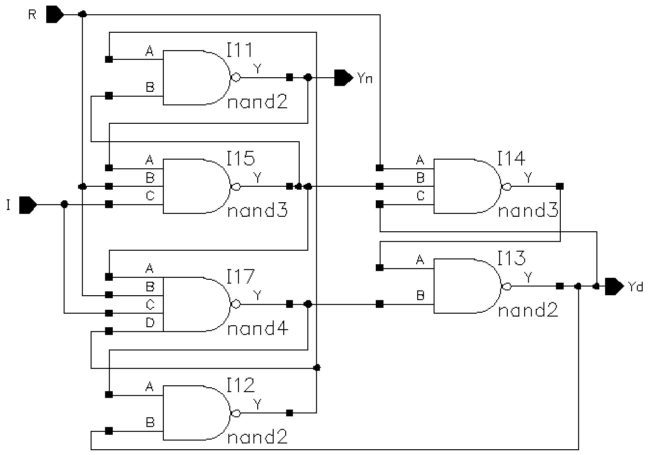
2-input C gate



2-phase Select module

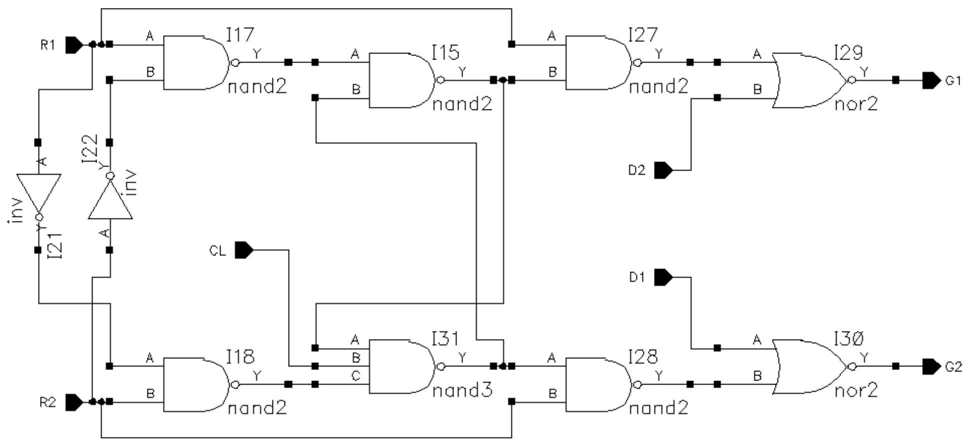


2-phase Toggle module

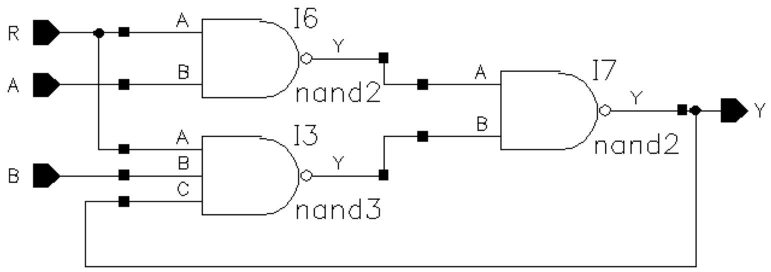


B.2 4-phase control modules

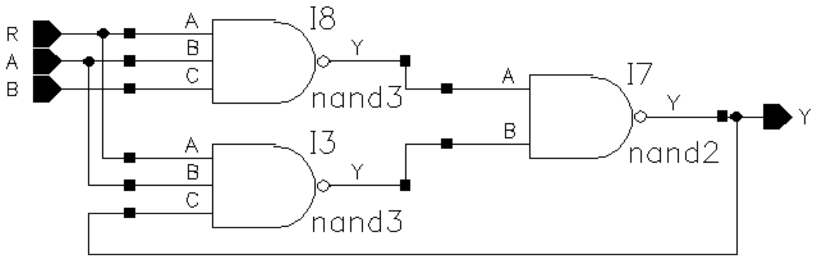
4-phase Arbiter



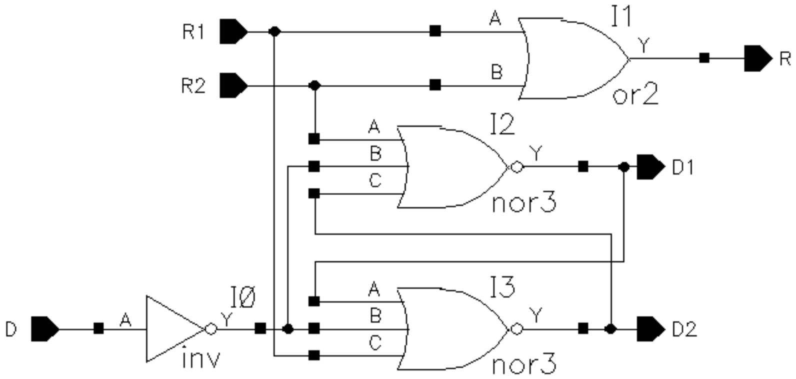
2-input Asymmetric C gate(c2_m1)



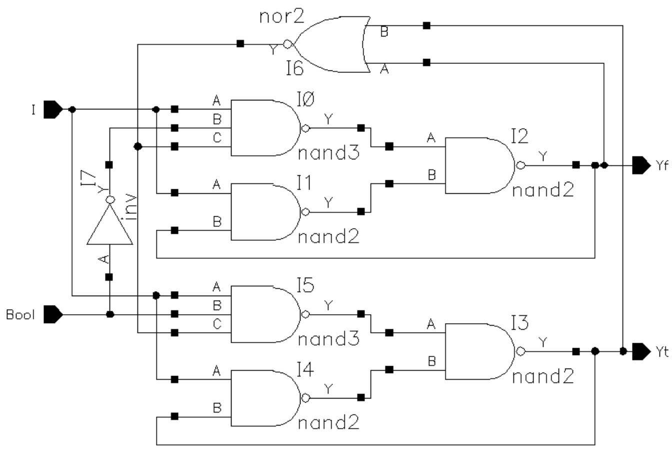
2-input Asymmetric C gate(c2_p1)



4-phase Call module



4-phase Select module



Appendix C

VHDL statements

Only some statements shown in this appendix are used to synthesize the circuits. However, other statements may also be available to synthesize the circuits in the new version.

C.1 Statements

Signal assignment statement

$signal <= expression;$

Variable assignment statement

$variable := expression;$

Entity declaration

entity entity_name **is**

[**port** (external_signal_declaration);]

end entity_name;

The external_signal_declaration) must contain **Reset, Rin, Ain** with **IN** mode and **Rout, Aout** with **OUT** mode. These signals are used for the handshaking control with the external input and output circuits.

External_signal_declaration

list_of_external_signals mode type { ; list_of_external_signals mode type }

Architecture declaration

architecture architecture_name **of** entity_name **is**

begin

architecture_body

end architecture_name;

Variable declaration

variable list_of_variable_names : type_name;

IF statement

if condition **then**

sequential statements

[**else** sequential statements]

end if;

FOR loop statement

for identifier **in** range **loop**

sequential statements

end loop;

WHILE loop statement

while boolean-expression **loop** sequential statements **end loop**;

C.2 Handshaking controls

The following procedures must be placed in the beginning of the **process** statement to deal with the handshaking control during the simulation. They will be removed before the synthesis.

```
procedure  req_in (Qrin: INOUT std_logic) IS
    begin
          while ( Rin = Qrin ) loop  wait for 1 ns;  end loop;
          Qrin := Rin;
    end req_in;

procedure  req_out (sro: INOUT std_logic) IS
    begin
          sro := not sro;  Rout <= sro;
    end req_out;
procedure  ack_out (Qaout: INOUT std_logic) IS
    begin
        while ( Aout = Qaout ) loop
           wait for 1 ns;
        end loop;
        Qaout := Aout;
    end ack_out;

procedure  ack_in IS
    begin
        Ain <= Rin;
    end ack_in;

procedure  reset_q (Qrin, Qaout, sro: INOUT std_logic) IS
    begin
           while ( Reset = '0' ) loop
              wait for 1 ns;
                Qrin := '0'; Qaout := '0'; sro := '0';
              wait for 1 ns;
                Rout <= sro;  Ain <= '0';
          end loop;
    end reset_q;
```

When the above procedures are declared **reset_q(Qrin, Qaout, sro);** , **req_out(sro);** , **ack_out(Qaout);** , **req_in(Qrin);** and **ack_in;** can be inserted into the circuit behaviural descritipn properly to deal with the handshaking control during the simulation. However, they can not be removed from the description before the synthesis if the these procedure calls inside loop statements. The reason is that they are used to identify what control circuits are used.

Appendix D

While loop examples

D.1 The output of Example 1

D.1.1 VHDL Entity of the whole circuit

```
LIBRARY XL;  USE XL.XL_STD.ALL;
ENTITY CIRCUIT IS PORT (
  RIN, AOUT : IN MVL;
  AIN, ROUT : OUT MVL;
  A : IN MVL_VECTOR ( 4 DOWNTO 0 );
  E : OUT MVL_VECTOR ( 4 DOWNTO 0 );
  RESET : IN MVL );
END CIRCUIT;
```

D.1.2 VHDL Architecture of the whole circuit

```
architecture STRUCTURAL of CIRCUIT is
  component stg1
     PORT ( Reset_stg1 : IN MVL; Rin_stg1 : IN MVL; Aout_stg1 : IN MVL;
        Ain_stg1 : OUT MVL; Rout_stg1 : OUT MVL;
        U71I0_0: IN MVL; U71I1_0: IN MVL; U71I2_0: IN MVL; U71I3_0: IN MVL;
        U71I4_0: IN MVL; Y_0: OUT MVL; Y_1: OUT MVL; Y_2: OUT MVL;
        Y_3: OUT MVL; Y_4: OUT MVL );
  end component;  for Ustg1: stg1 use entity work.stg1(structural);
  signal bus51: MVL_VECTOR( 0 to 4);  signal bus52: MVL_VECTOR( 0 to 4);
  signal wire11: MVL; signal wire12: MVL; signal wire13: MVL;
  signal wire14: MVL; signal wire15: MVL;
begin
  wire13 <= RESET; wire11 <= RIN; wire12 <= AOUT; ROUT <= wire15; AIN <= wire14;
  bus51( 0 ) <= A(0); bus51( 1 ) <= A(1); bus51( 2 ) <= A(2); bus51( 3 ) <= A(3); bus51( 4 ) <= A(4);
  E(0) <= bus52( 0 );  E(1) <= bus52( 1 );  E(2) <= bus52( 2 ); E(3) <= bus52( 3 );  E(4) <= bus52( 4 );
  Ustg1: stg1 PORT map ( Reset_stg1 => wire13, Rin_stg1 => wire11,  Aout_stg1 => wire12,
         Ain_stg1 => wire14,  Rout_stg1 => wire15, U71I0_0 => bus51( 0 ), U71I1_0 => bus51( 1 ),
         U71I2_0 => bus51( 2 ), U71I3_0 => bus51( 3 ), U71I4_0 => bus51( 4 ), Y_0 => bus52( 0 ),
         Y_1 => bus52( 1 ), Y_2 => bus52( 2 ), Y_3 => bus52( 3 ), Y_4 => bus52( 4 ) );
end STRUCTURAL;
```

D.1.3 VHDL Entity of the stg1

```
library XL;  use XL.XL_STD.all;
ENTITY stg1 IS
  PORT ( Reset_stg1 : IN MVL; Rin_stg1, Aout_stg1 : IN MVL; Ain_stg1, Rout_stg1 : OUT MVL;
        U71I0_0, U71I1_0, U71I2_0, U71I3_0, U71I4_0: IN MVL;
        Y_0, Y_1, Y_2, Y_3, Y_4: OUT MVL );
END stg1;
```

D.1.4 VHDL Architecture of the stg1

```
architecture STRUCTURAL of stg1 is
 component adder5 PORT ( C0: IN MVL; A0: IN MVL; A1: IN MVL; A2: IN MVL; A3: IN MVL;
  A4: IN MVL; B0: IN MVL; B1: IN MVL; B2: IN MVL; B3: IN MVL; B4: IN MVL;
  S0: OUT MVL; S1: OUT MVL; S2: OUT MVL; S3: OUT MVL; S4: OUT MVL; C: OUT MVL );
 end component; for U11: adder5 use entity work.adder5(behavioral);
 component c2 PORT ( R: IN MVL; A: IN MVL; B: IN MVL; Y: OUT MVL ); end component;
 for U21: c2 use entity work.c2(behavioral); for U22: c2 use entity work.c2(behavioral);
 component call2 PORT ( CL: IN MVL; R1: IN MVL; D1: OUT MVL; R2: IN MVL;
  D2: OUT MVL; R: OUT MVL; D: IN MVL );
 end component; for U31: call2 use entity work.call2(behavioral);
 component comparator5 PORT ( A0: IN MVL; A1: IN MVL; A2: IN MVL; A3: IN MVL;
  A4: IN MVL; B0: IN MVL; B1: IN MVL; B2: IN MVL; B3: IN MVL;
  B4: IN MVL; G: OUT MVL; E: OUT MVL; L: OUT MVL );
 end component; for U41: comparator5 use entity work.comparator5(behavioral);
 component htlatch5 PORT ( Lt: IN MVL; I0: IN MVL; I1: IN MVL; I2: IN MVL; I3: IN MVL;
  I4: IN MVL; Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL; Y3: OUT MVL; Y4: OUT MVL );
 end component; for U51: htlatch5 use entity work.htlatch5(behavioral);
 for U52: htlatch5 use entity work.htlatch5(behavioral);
 component ltlatch5 PORT ( Lt: IN MVL; I0: IN MVL; I1: IN MVL; I2: IN MVL; I3: IN MVL;
  I4: IN MVL; Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL; Y3: OUT MVL; Y4: OUT MVL );
 end component; for U61: ltlatch5 use entity work.ltlatch5(behavioral);
 for U62: ltlatch5 use entity work.ltlatch5(behavioral);
 component mux2x5 PORT ( S: IN MVL; I0_0: IN MVL; I0_1: IN MVL; I1_0: IN MVL;
  I1_1: IN MVL; I2_0: IN MVL; I2_1: IN MVL; I3_0: IN MVL; I3_1: IN MVL; I4_0: IN MVL;
  I4_1: IN MVL; Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL; Y3: OUT MVL; Y4: OUT MVL );
 end component; for U71: mux2x5 use entity work.mux2x5(behavioral);
 for U72: mux2x5 use entity work.mux2x5(behavioral);
 component not1 PORT ( A: IN MVL; B: OUT MVL ); end component;
 for U81: not1 use entity work.not1(behavioral); for U82: not1 use entity work.not1(behavioral);
 component or2 PORT ( A: IN MVL; B: IN MVL; Y: OUT MVL ); end component;
 for U91: or2 use entity work.or2(behavioral);
 component select1 PORT ( R: IN MVL; Bool: IN MVL; I: IN MVL; Yt: OUT MVL; Yf: OUT MVL );
 end component; for U101: select1 use entity work.select1(behavioral);
 component subtractor5 PORT ( C0: IN MVL; A0: IN MVL; A1: IN MVL; A2: IN MVL; A3: IN MVL;
  A4: IN MVL; B0: IN MVL; B1: IN MVL; B2: IN MVL; B3: IN MVL; B4: IN MVL; S0: OUT MVL;
  S1: OUT MVL; S2: OUT MVL; S3: OUT MVL; S4: OUT MVL; C: OUT MVL );
 end component; for U111: subtractor5 use entity work.subtractor5(behavioral);
 component toggle2 PORT ( R: IN MVL; I: IN MVL; Yd: OUT MVL; Yn: OUT MVL );
 end component; for U121: toggle2 use entity work.toggle2(behavioral);
 component xor2 PORT ( A: IN MVL; B: IN MVL; Y: OUT MVL ); end component;
 for U131: xor2 use entity work.xor2(behavioral); for U132: xor2 use entity work.xor2(behavioral);
 for U133: xor2 use entity work.xor2(behavioral);  signal bus51: MVL_VECTOR( 0 to 4);
 signal bus52: MVL_VECTOR( 0 to 4); signal bus53: MVL_VECTOR( 0 to 4);
 signal bus54: MVL_VECTOR( 0 to 4); signal bus55: MVL_VECTOR( 0 to 4);
 signal bus56: MVL_VECTOR( 0 to 4); signal bus57: MVL_VECTOR( 0 to 4);
 signal bus58: MVL_VECTOR( 0 to 4); signal bus59: MVL_VECTOR( 0 to 4);
 signal bus510: MVL_VECTOR( 0 to 4); signal bus511: MVL_VECTOR( 0 to 4);
 signal wire13: MVL; signal wire14: MVL; signal wire15: MVL; signal wire16: MVL;
 signal wire17: MVL; signal wire18: MVL; signal wire19: MVL; signal wire110: MVL;
 signal wire111: MVL; signal wire112: MVL; signal wire113: MVL; signal wire114: MVL;
 signal wire115: MVL; signal wire116: MVL; signal wire117: MVL; signal wire118: MVL;
 signal wire119: MVL; signal wire120: MVL; signal wire121: MVL; signal wire122: MVL;
```

```
 signal bus512: MVL_VECTOR( 0 to 4); signal wire11: MVL; signal wire12: MVL;
begin  wire120 <= '0';  wire121 <= '0'; bus55( 0 ) <= '0'; bus55( 1 ) <= '0'; bus55( 2 ) <= '1';
 bus55( 3 ) <= '0'; bus55( 4 ) <= '1'; bus57( 0 ) <= '0'; bus57( 1 ) <= '0'; bus57( 2 ) <= '0';
 bus57( 3 ) <= '0'; bus57( 4 ) <= '1'; bus59( 0 ) <= '0'; bus59( 1 ) <= '0'; bus59( 2 ) <= '0';
 bus59( 3 ) <= '0'; bus59( 4 ) <= '1'; bus511( 0 ) <= '0'; bus511( 1 ) <= '0'; bus511( 2 ) <= '0';
 bus511( 3 ) <= '0'; bus511( 4 ) <= '1'; wire13 <= Reset_stg1; wire11 <= Rin_stg1;
 wire12 <= Aout_stg1; Rout_stg1 <= wire112; Ain_stg1 <= wire18; Y_0 <= bus51( 0 );
 Y_1 <= bus51( 1 ); Y_2 <= bus51( 2 ); Y_3 <= bus51( 3 ); Y_4 <= bus51( 4 );
 U11: adder5 PORT map ( C0 => wire120, A0 => bus58( 0 ), A1 => bus58( 1 ), A2 => bus58( 2 ),
    A3 => bus58( 3 ), A4 => bus58( 4 ), B0 => bus59( 0 ), B1 => bus59( 1 ), B2 => bus59( 2 ),
    B3 => bus59( 3 ), B4 => bus59( 4 ), S0 => bus510( 0 ), S1 => bus510( 1 ),
    S2 => bus510( 2 ), S3 => bus510( 3 ), S4 => bus510( 4 ), C => open );
 U21: c2 PORT map ( R => wire13, A => wire11, B => wire15, Y => wire14 );
 U22: c2 PORT map ( R => wire13, A => wire110, B => wire114, Y => wire113 );
 U31: call2 PORT map ( CL => wire13, R1 => wire14, D1 => wire18,
    R2 => wire111, D2 => wire19, R => wire110, D => wire116 );
 U41: comparator5 PORT map ( A0 => bus56( 0 ), A1 => bus56( 1 ), A2 => bus56( 2 ),
    A3 => bus56( 3 ), A4 => bus56( 4 ), B0 => bus57( 0 ), B1 => bus57( 1 ), B2 => bus57( 2 ),
    B3 => bus57( 3 ), B4 => bus57( 4 ), G => wire118, E => wire119, L => open );
 U51: htlatch5 PORT map ( Lt => wire115, I0 => bus54( 0 ), I1 => bus54( 1 ), I2 => bus54( 2 ),
    I3 => bus54( 3 ),  I4 => bus54( 4 ), Y0 => bus56( 0 ), Y1 => bus56( 1 ),
    Y2 => bus56( 2 ), Y3 => bus56( 3 ), Y4 => bus56( 4 ) );
 U52: htlatch5 PORT map ( Lt => wire115, I0 => bus51( 0 ), I1 => bus51( 1 ),
    I2 => bus51( 2 ), I3 => bus51( 3 ), I4 => bus51( 4 ), Y0 => bus58( 0 ),
    Y1 => bus58( 1 ), Y2 => bus58( 2 ), Y3 => bus58( 3 ), Y4 => bus58( 4 ) );
 U61: ltlatch5 PORT map ( Lt => wire115, I0 => bus52( 0 ), I1 => bus52( 1 ),
    I2 => bus52( 2 ), I3 => bus52( 3 ), I4 => bus52( 4 ), Y0 => bus51( 0 ),
    Y1 => bus51( 1 ), Y2 => bus51( 2 ), Y3 => bus51( 3 ), Y4 => bus51( 4 ) );
 U62: ltlatch5 PORT map ( Lt => wire115, I0 => bus53( 0 ), I1 => bus53( 1 ),
    I2 => bus53( 2 ), I3 => bus53( 3 ), I4 => bus53( 4 ), Y0 => bus54( 0 ),
    Y1 => bus54( 1 ), Y2 => bus54( 2 ), Y3 => bus54( 3 ), Y4 => bus54( 4 ) );
 U71: mux2x5 PORT map ( S => wire16, I0_0 => U71I0_0, I0_1 => bus510( 0 ),
    I1_0 => U71I1_0, I1_1 => bus510( 1 ), I2_0 => U71I2_0, I2_1 => bus510( 2 ),
    I3_0 => U71I3_0, I3_1 => bus510( 3 ), I4_0 => U71I4_0, I4_1 => bus510( 4 ),
    Y0 => bus52( 0 ), Y1 => bus52( 1 ), Y2 => bus52( 2 ), Y3 => bus52( 3 ),
    Y4 => bus52( 4 ) );
 U72: mux2x5 PORT map ( S => wire16, I0_0 => bus55( 0 ), I0_1 => bus512( 0 ),
    I1_0 => bus55( 1 ), I1_1 => bus512( 1 ), I2_0 => bus55( 2 ), I2_1 => bus512( 2 ),
    I3_0 => bus55( 3 ), I3_1 => bus512( 3 ), I4_0 => bus55( 4 ), I4_1 => bus512( 4 ),
    Y0 => bus53( 0 ), Y1 => bus53( 1 ), Y2 => bus53( 2 ), Y3 => bus53( 3 ),
    Y4 => bus53( 4 ) );
 U81: not1 PORT map ( A => wire12, B => wire15 );
    U82: not1 PORT map ( A => wire117, B => wire114 );
 U91: or2 PORT map ( A => wire118, B => wire119, Y => wire122 );
 U101: select1 PORT map ( R => wire13, Bool => wire122, I => wire17,
    Yt => wire111, Yf => wire112 );
 U111: subtractor5 PORT map ( C0 => wire121, A0 => bus56( 0 ), A1 => bus56( 1 ),
    A2 => bus56( 2 ), A3 => bus56( 3 ), A4 => bus56( 4 ), B0 => bus511( 0 ),
    B1 => bus511( 1 ), B2 => bus511( 2 ), B3 => bus511( 3 ), B4 => bus511( 4 ),
    S0 => bus512( 0 ), S1 => bus512( 1 ), S2 => bus512( 2 ), S3 => bus512( 3 ),
    S4 => bus512( 4 ), C => open );
 U121: toggle2 PORT map ( R => wire13, I => wire115, Yd => wire116, Yn => wire117 );
 U131: xor2 PORT map ( A => wire18, B => wire112, Y => wire16 );
 U132: xor2 PORT map ( A => wire18, B => wire19, Y => wire17 );
 U133: xor2 PORT map ( A => wire113, B => wire116, Y => wire115 );
end STRUCTURAL;
```

D.1.5 Verilog of the whole circuit

```
module stg1 (Reset_stg1, Rin_stg1, Aout_stg1, Ain_stg1, Rout_stg1, U71I0_0, U71I1_0, U71I2_0,
U71I3_0, U71I4_0, Y_0, Y_1, Y_2, Y_3, Y_4);
input Reset_stg1; input Rin_stg1; input Aout_stg1; output Ain_stg1; output Rout_stg1;
input U71I0_0; input U71I1_0; input U71I2_0; input U71I3_0; input U71I4_0; output Y_0;
output Y_1; output Y_2; output Y_3; output Y_4; endmodule
module buffer1 (Y, A); output Y; input A; endmodule
module log_vdd (vdd1); output vdd1; endmodule
module log_gnd (gnd1); output gnd1; endmodule
module CIRCUIT (RIN, AOUT, AIN, ROUT, A, E, RESET);
input RIN; input AOUT; output AIN; output ROUT; input [4:0] A; output [4:0] E; input RESET;
wire [0:4] bus51; wire [0:4] bus52; wire wire11; wire wire12; wire wire13; wire wire14;
wire wire15; buffer1 B1 (.Y(wire13), .A(RESET)); buffer1 B2 (.Y(wire11), .A(RIN));
buffer1 B3 (.Y(wire12), .A(AOUT)); buffer1 B4 (.Y(ROUT), .A(wire15));
buffer1 B5 (.Y(AIN), .A(wire14)); buffer1 B6 (.Y(bus51[0]), .A(A[0]));
buffer1 B7 (.Y(bus51[1]), .A(A[1])); buffer1 B8 (.Y(bus51[2]), .A(A[2]));
buffer1 B9 (.Y(bus51[3]), .A(A[3])); buffer1 B10 (.Y(bus51[4]), .A(A[4]));
buffer1 B11 (.Y(E[0]), .A(bus52[0])); buffer1 B12 (.Y(E[1]), .A(bus52[1]));
buffer1 B13 (.Y(E[2]), .A(bus52[2])); buffer1 B14 (.Y(E[3]), .A(bus52[3]));
buffer1 B15 (.Y(E[4]), .A(bus52[4]));
stg1 Ustg1 (.Reset_stg1(wire13), .Rin_stg1(wire11), .Aout_stg1(wire12), .Ain_stg1(wire14),
    .Rout_stg1(wire15), .U71I0_0(bus51[0]), .U71I1_0(bus51[1]), .U71I2_0(bus51[2]),
    .U71I3_0(bus51[3]), .U71I4_0(bus51[4]), .Y_0(bus52[0]), .Y_1(bus52[1]),
    .Y_2(bus52[2]), .Y_3(bus52[3]), .Y_4(bus52[4]));  endmodule
```

D.1.6 Verilog of the stg1

```
module adder5 (C0, A0, A1, A2, A3, A4, B0, B1, B2, B3, B4, S0, S1, S2, S3, S4, C);
input C0; input A0; input A1; input A2; input A3; input A4; input B0; input B1; input B2;
input B3; input B4; output S0; output S1; output S2; output S3; output S4; output C;
endmodule    module c2 (R, A, B, Y); input R; input A; input B; output Y; endmodule
module call2 (CL, R1, D1, R2, D2, R, D); input CL; input R1;
output D1; input R2; output D2; output R; input D; endmodule
module comparator5 (A0, A1, A2, A3, A4, B0, B1, B2, B3, B4, G, E, L);
input A0; input A1; input A2; input A3; input A4; input B0; input B1;
input B2; input B3; input B4; output G; output E; output L; endmodule
module htlatch5 (Lt, I0, I1, I2, I3, I4, Y0, Y1, Y2, Y3, Y4);
input Lt; input I0; input I1; input I2; input I3; input I4;
output Y0; output Y1; output Y2; output Y3; output Y4; endmodule
module ltlatch5 (Lt, I0, I1, I2, I3, I4, Y0, Y1, Y2, Y3, Y4);
input Lt; input I0; input I1; input I2; input I3; input I4;
output Y0; output Y1; output Y2; output Y3; output Y4; endmodule
module mux2x5 (S, I0_0, I0_1, I1_0, I1_1, I2_0, I2_1, I3_0, I3_1, I4_0, I4_1, Y0, Y1,
Y2, Y3, Y4); input S; input I0_0; input I0_1; input I1_0; input I1_1; input I2_0;
input I2_1; input I3_0; input I3_1; input I4_0; input I4_1; output Y0; output Y1;
output Y2; output Y3; output Y4; endmodule
module not1 (A, B); input A; output B; endmodule   module or2 (A, B, Y); input A;
input B; output Y; endmodule   module select1 (R, Bool, I, Yt, Yf);
input R; input Bool; input I; output Yt; output Yf; endmodule
module subtractor5 (C0, A0, A1, A2, A3, A4, B0, B1, B2, B3, B4, S0, S1, S2,
S3, S4, C); input C0; input A0; input A1; input A2; input A3; input A4;
input B0; input B1; input B2; input B3; input B4; output S0; output S1;
output S2; output S3; output S4; output C; endmodule
module toggle2 (R, I, Yd, Yn); input R; input I; output Yd; output Yn; endmodule
module xor2 (A, B, Y); input A; input B; output Y; endmodule  module buffer1 (Y, A);
output Y; input A; endmodule  module log_vdd (vdd1); output vdd1; endmodule
module log_gnd (gnd1); output gnd1; endmodule
module stg1 (Reset_stg1, Rin_stg1, Aout_stg1, Ain_stg1, Rout_stg1, U71I0_0,
U71I1_0, U71I2_0, U71I3_0, U71I4_0, Y_0, Y_1, Y_2, Y_3, Y_4); input Reset_stg1;
input Rin_stg1; input Aout_stg1; output Ain_stg1; output Rout_stg1; input U71I0_0;
input U71I1_0; input U71I2_0; input U71I3_0; input U71I4_0; output Y_0; output Y_1;
output Y_2; output Y_3; output Y_4; wire [0:4] bus51; wire [0:4] bus52;
wire [0:4] bus53; wire [0:4] bus54; wire [0:4] bus55; wire [0:4] bus56;
wire [0:4] bus57; wire [0:4] bus58; wire [0:4] bus59; wire [0:4] bus510;
wire [0:4] bus511; wire [0:4] bus512; wire wire11; wire wire12; wire wire13; wire wire14;
wire wire15; wire wire16; wire wire17; wire wire18; wire wire19; wire wire110;
wire wire111; wire wire112; wire wire113; wire wire114; wire wire115; wire wire116;
wire wire117; wire wire118; wire wire119; wire wire120; wire wire121; wire wire122;
log_gnd LL1 (.gnd1(wire120)); log_gnd LL2 (.gnd1(wire121));
log_gnd LL3 (.gnd1(bus55[0])); log_gnd LL4 (.gnd1(bus55[1]));
log_vdd LH1 (.vdd1(bus55[2])); log_gnd LL5 (.gnd1(bus55[3]));
log_vdd LH2 (.vdd1(bus55[4])); log_gnd LL6 (.gnd1(bus57[0]));
```

```
log_gnd LL7 (.gnd1(bus57[1])); log_gnd LL8 (.gnd1(bus57[2]));
log_gnd LL9 (.gnd1(bus57[3])); log_vdd LH3 (.vdd1(bus57[4]));
log_gnd LL10 (.gnd1(bus59[0])); log_gnd LL11 (.gnd1(bus59[1]));
log_gnd LL12 (.gnd1(bus59[2])); log_gnd LL13 (.gnd1(bus59[3]));
log_vdd LH4 (.vdd1(bus59[4])); log_gnd LL14 (.gnd1(bus511[0]));
log_gnd LL15 (.gnd1(bus511[1])); log_gnd LL16 (.gnd1(bus511[2]));
log_gnd LL17 (.gnd1(bus511[3])); log_vdd LH5 (.vdd1(bus511[4]));
buffer1 B1 (.Y(wire13), .A(Reset_stg1));  buffer1 B2 (.Y(wire11), .A(Rin_stg1));
buffer1 B3 (.Y(wire12), .A(Aout_stg1));  buffer1 B4 (.Y(Rout_stg1), .A(wire112));
buffer1 B5 (.Y(Ain_stg1), .A(wire18));  buffer1 B6 (.Y(Y_0), .A(bus51( 0 )));
buffer1 B7 (.Y(Y_1), .A(bus51( 1 )));  buffer1 B8 (.Y(Y_2), .A(bus51( 2 )));
buffer1 B9 (.Y(Y_3), .A(bus51( 3 )));  buffer1 B10 (.Y(Y_4), .A(bus51( 4 )));
adder5 U11 (.C0(wire120), .A0(bus58[0]), .A1(bus58[1]), .A2(bus58[2]), .A3(bus58[3]),
   .A4(bus58[4]), .B0(bus59[0]), .B1(bus59[1]), .B2(bus59[2]), .B3(bus59[3]), .B4(bus59[4]),
   .S0(bus510[0]), .S1(bus510[1]), .S2(bus510[2]), .S3(bus510[3]), .S4(bus510[4]), .C(open1));
c2 U21 (.R(wire13), .A(wire11), .B(wire15), .Y(wire14));
c2 U22 (.R(wire13), .A(wire110), .B(wire114), .Y(wire113));
call2 U31 (.CL(wire13), .R1(wire14), .D1(wire18), .R2(wire111),
   .D2(wire19), .R(wire110), .D(wire116));
comparator5 U41 (.A0(bus56[0]), .A1(bus56[1]), .A2(bus56[2]), .A3(bus56[3]), .A4(bus56[4]),
   .B0(bus57[0]), .B1(bus57[1]), .B2(bus57[2]), .B3(bus57[3]), .B4(bus57[4]), .G(wire118),
   .E(wire119), .L(open2));
htlatch5 U51 (.Lt(wire115), .I0(bus54[0]), .I1(bus54[1]), .I2(bus54[2]), .I3(bus54[3]),
   .I4(bus54[4]), .Y0(bus56[0]), .Y1(bus56[1]), .Y2(bus56[2]), .Y3(bus56[3]), .Y4(bus56[4]));
htlatch5 U52 (.Lt(wire115), .I0(bus51[0]), .I1(bus51[1]), .I2(bus51[2]), .I3(bus51[3]),
   .I4(bus51[4]), .Y0(bus58[0]), .Y1(bus58[1]), .Y2(bus58[2]), .Y3(bus58[3]), .Y4(bus58[4]));
ltlatch5 U61 (.Lt(wire115), .I0(bus52[0]), .I1(bus52[1]), .I2(bus52[2]), .I3(bus52[3]),
   .I4(bus52[4]), .Y0(bus51[0]), .Y1(bus51[1]), .Y2(bus51[2]), .Y3(bus51[3]), .Y4(bus51[4]));
ltlatch5 U62 (.Lt(wire115), .I0(bus53[0]), .I1(bus53[1]), .I2(bus53[2]), .I3(bus53[3]),
   .I4(bus53[4]), .Y0(bus54[0]), .Y1(bus54[1]), .Y2(bus54[2]), .Y3(bus54[3]), .Y4(bus54[4]));
mux2x5 U71 (.S(wire16), .I0_0(U71I0_0), .I0_1(bus510[0]), .I1_0(U71I1_0), .I1_1(bus510[1]),
   .I2_0(U71I2_0), .I2_1(bus510[2]), .I3_0(U71I3_0), .I3_1(bus510[3]), .I4_0(U71I4_0),
   .I4_1(bus510[4]), .Y0(bus52[0]), .Y1(bus52[1]), .Y2(bus52[2]), .Y3(bus52[3]), .Y4(bus52[4]));
mux2x5 U72 (.S(wire16), .I0_0(bus55[0]), .I0_1(bus512[0]), .I1_0(bus55[1]), .I1_1(bus512[1]),
   .I2_0(bus55[2]), .I2_1(bus512[2]), .I3_0(bus55[3]), .I3_1(bus512[3]), .I4_0(bus55[4]),
   .I4_1(bus512[4]), .Y0(bus53[0]), .Y1(bus53[1]), .Y2(bus53[2]), .Y3(bus53[3]), .Y4(bus53[4]));
not1 U81 (.A(wire12), .B(wire15)); not1 U82 (.A(wire117), .B(wire114));
or2 U91 (.A(wire118), .B(wire119), .Y(wire122));
select1 U101 (.R(wire13), .Bool(wire122), .I(wire17), .Yt(wire111), .Yf(wire112));
subtractor5 U111 (.C0(wire121), .A0(bus56[0]), .A1(bus56[1]), .A2(bus56[2]), .A3(bus56[3]),
   .A4(bus56[4]), .B0(bus511[0]), .B1(bus511[1]), .B2(bus511[2]), .B3(bus511[3]),
   .B4(bus511[4]), .S0(bus512[0]), .S1(bus512[1]), .S2(bus512[2]), .S3(bus512[3]),
   .S4(bus512[4]), .C(open3));
toggle2 U121 (.R(wire13), .I(wire115), .Yd(wire116), .Yn(wire117));
xor2 U131 (.A(wire18), .B(wire112), .Y(wire16));
xor2 U132 (.A(wire18), .B(wire19), .Y(wire17));
xor2 U133 (.A(wire113), .B(wire116), .Y(wire115)); endmodule
```

D.2 The output of Example 2

D.2.1 4-phase output

D.2.1.1 VHDL of the whole circuit

```
LIBRARY XL;  USE XL.XL_STD.ALL;
ENTITY circuit2 IS PORT ( RIN, AOUT : IN MVL; AIN, ROUT : OUT MVL; EA : IN MVL_VECTOR ( 0 TO 4 );
  MA : IN MVL_VECTOR ( 0 TO 14 ); OP : IN MVL_VECTOR ( 1 DOWNTO 0 ); EO : OUT MVL_VECTOR ( 0 TO 4 );
  MO : OUT MVL_VECTOR ( 0 TO 8 ); RESET : IN MVL );  END circuit2;
architecture STRUCTURAL of circuit2 is
  component stg1 PORT ( Reset_stg1 : IN MVL; Rin_stg1 : IN MVL; Aout_stg1 : IN MVL; Ain_stg1 : OUT MVL;
    Rout_stg1 : OUT MVL; U81I0_0: IN MVL; U81I1_0: IN MVL; U81I2_0: IN MVL; U81I3_0: IN MVL; U81I4_0: IN MVL;
    U81I5_0: IN MVL; U81I6_0: IN MVL; U81I7_0: IN MVL; U81I8_0: IN MVL; U81I9_0: IN MVL; U81I10_0: IN MVL;
    U81I11_0: IN MVL; U81I12_0: IN MVL; U81I13_0: IN MVL; U81I14_0: IN MVL; U91I0_0: IN MVL; U91I1_0: IN MVL;
    U91I2_0: IN MVL; U91I3_0: IN MVL; U91I4_0: IN MVL; FRA_H_0: OUT MVL; FRA_H_1: OUT MVL; FRA_H_2: OUT MVL;
    FRA_H_3: OUT MVL; FRA_H_4: OUT MVL; FRA_H_5: OUT MVL; FRA_H_6: OUT MVL; FRA_H_7: OUT MVL;
    FRA_H_8: OUT MVL; FRA_H_9: OUT MVL; FRA_H_10: OUT MVL; FRA_H_11: OUT MVL; FRA_H_12: OUT MVL;
    FRA_H_13: OUT MVL; FRA_H_14: OUT MVL; EXP_H_0: OUT MVL; EXP_H_1: OUT MVL; EXP_H_2: OUT MVL;
    EXP_H_3: OUT MVL; EXP_H_4: OUT MVL );
  end component; for Ustg1: stg1 use entity work.stg1(structural);
  component stg2 PORT ( Reset_stg2 : IN MVL; Rin_stg2 : IN MVL; Aout_stg2 : IN MVL; Ain_stg2 : OUT MVL;
    Rout_stg2 : OUT MVL; U21I0: IN MVL; U21I1: IN MVL; U21I2: IN MVL; U21I3: IN MVL; U21I4: IN MVL; U21I5: IN MVL;
    U21I6: IN MVL; U21I7: IN MVL; U21I8: IN MVL; U21I9: IN MVL; U21I10: IN MVL; U21I11: IN MVL; U21I12: IN MVL;
    U21I13: IN MVL; U21I14: IN MVL; U31I0: IN MVL; U31I1: IN MVL; U31I2: IN MVL; U31I3: IN MVL; U31I4: IN MVL;
    FRA_RM_0: OUT MVL; FRA_RM_3: OUT MVL; FRA_RM_4: OUT MVL; FRA_RM_5: OUT MVL; FRA_RM_6: OUT MVL;
    FRA_RM_7: OUT MVL; FRA_RM_8: OUT MVL; FRA_RM_9: OUT MVL; FRA_RM_10: OUT MVL; EXP_RM_0: OUT MVL;
    EXP_RM_1: OUT MVL; EXP_RM_2: OUT MVL; EXP_RM_3: OUT MVL; EXP_RM_4: OUT MVL );
  end component; for Ustg2: stg2 use entity work.stg2(structural);
  signal bus151: MVL_VECTOR( 0 to 14); signal bus152: MVL_VECTOR( 0 to 14); signal bus51: MVL_VECTOR( 0 to 4);
  signal bus52: MVL_VECTOR( 0 to 4); signal bus53: MVL_VECTOR( 0 to 4); signal bus81: MVL_VECTOR( 0 to 7);
  signal wire11: MVL; signal wire12: MVL; signal wire13: MVL; signal wire14: MVL;
  signal wire15: MVL; signal wire16: MVL; signal wire17: MVL; signal wire18: MVL;
begin    wire13 <= RESET; wire11 <= RIN; wire12 <= AOUT; ROUT <= wire18; AIN <= wire15; bus51( 0 ) <= EA(0);
  bus51( 1 ) <= EA(1); bus51( 2 ) <= EA(2); bus51( 3 ) <= EA(3); bus51( 4 ) <= EA(4); bus151( 0 ) <= MA(0);
  bus151( 1 ) <= MA(1); bus151( 2 ) <= MA(2); bus151( 3 ) <= MA(3); bus151( 4 ) <= MA(4); bus151( 5 ) <= MA(5);
  bus151( 6 ) <= MA(6); bus151( 7 ) <= MA(7); bus151( 8 ) <= MA(8); bus151( 9 ) <= MA(9); bus151( 10 ) <= MA(10);
  bus151( 11 ) <= MA(11); bus151( 12 ) <= MA(12); bus151( 13 ) <= MA(13); bus151( 14 ) <= MA(14); open <= OP(0);
  open <= OP(1); EO(0) <= bus53( 0 ); EO(1) <= bus53( 1 ); EO(2) <= bus53( 2 ); EO(3) <= bus53( 3 );
  EO(4) <= bus53( 4 ); MO(0) <= wire14; MO(1) <= bus81( 0 ); MO(2) <= bus81( 1 ); MO(3) <= bus81( 2 );
  MO(4) <= bus81( 3 ); MO(5) <= bus81( 4 ); MO(6) <= bus81( 5 ); MO(7) <= bus81( 6 ); MO(8) <= bus81( 7 );
  Ustg1: stg1 PORT map ( Reset_stg1 => wire13, Rin_stg1 => wire11, Aout_stg1 => wire17, Ain_stg1 => wire15,
   Rout_stg1 => wire16, U81I0_0 => bus151( 0 ),  U81I1_0 => bus151( 1 ), U81I2_0 => bus151( 2 ),
   U81I3_0 => bus151( 3 ), U81I4_0 => bus151( 4 ), U81I5_0 => bus151( 5 ), U81I6_0 => bus151( 6 ),
   U81I7_0 => bus151( 7 ), U81I8_0 => bus151( 8 ), U81I9_0 => bus151( 9 ), U81I10_0 => bus151( 10 ),
   U81I11_0 => bus151( 11 ), U81I12_0 => bus151( 12 ), U81I13_0 => bus151( 13 ), U81I14_0 => bus151( 14 ),
   U91I0_0 => bus51( 0 ), U91I1_0 => bus51( 1 ), U91I2_0 => bus51( 2 ), U91I3_0 => bus51( 3 ), U91I4_0 => bus51( 4 ),
   FRA_H_0 => bus152( 0 ), FRA_H_1 => bus152( 1 ), FRA_H_2 => bus152( 2 ), FRA_H_3 => bus152( 3 ),
   FRA_H_4 => bus152( 4 ), FRA_H_5 => bus152( 5 ), FRA_H_6 => bus152( 6 ), FRA_H_7 => bus152( 7 ),
   FRA_H_8 => bus152( 8 ), FRA_H_9 => bus152( 9 ), FRA_H_10 => bus152( 10 ), FRA_H_11 => bus152( 11 ),
   FRA_H_12 => bus152( 12 ), FRA_H_13 => bus152( 13 ), FRA_H_14 => bus152( 14 ), EXP_H_0 => bus52( 0 ),
   EXP_H_1 => bus52( 1 ), EXP_H_2 => bus52( 2 ), EXP_H_3 => bus52( 3 ), EXP_H_4 => bus52( 4 ) );
  Ustg2: stg2 PORT map ( Reset_stg2 => wire13, Rin_stg2 => wire16, Aout_stg2 => wire12, Ain_stg2 => wire17,
   Rout_stg2 => wire18, U21I0 => bus152( 0 ), U21I1 => bus152( 1 ), U21I2 => bus152( 2 ), U21I3 => bus152( 3 ),
   U21I4 => bus152( 4 ), U21I5 => bus152( 5 ), U21I6 => bus152( 6 ), U21I7 => bus152( 7 ), U21I8 => bus152( 8 ),
   U21I9 => bus152( 9 ), U21I10 => bus152( 10 ), U21I11 => bus152( 11 ), U21I12 => bus152( 12 ),
   U21I13 => bus152( 13 ), U21I14 => bus152( 14 ), U31I0 => bus52( 0 ), U31I1 => bus52( 1 ), U31I2 => bus52( 2 ),
   U31I3 => bus52( 3 ), U31I4 => bus52( 4 ), FRA_RM_0 => wire14, FRA_RM_3 => bus81( 0 ), FRA_RM_4 => bus81( 1 ),
   FRA_RM_5 => bus81( 2 ), FRA_RM_6 => bus81( 3 ), FRA_RM_7 => bus81( 4 ), FRA_RM_8 => bus81( 5 ),
   FRA_RM_9 => bus81( 6 ), FRA_RM_10 => bus81( 7 ), EXP_RM_0 => bus53( 0 ), EXP_RM_1 => bus53( 1 ),
   EXP_RM_2 => bus53( 2 ), EXP_RM_3 => bus53( 3 ), EXP_RM_4 => bus53( 4 ) );
end STRUCTURAL;
```

D.2.1.2 VHDL of the stg1

```
library XL;  use XL.XL_STD.all;
ENTITY stg1 IS
  PORT ( Reset_stg1 : IN MVL; Rin_stg1, Aout_stg1 : IN MVL; Ain_stg1, Rout_stg1 : OUT MVL;
       U81I0_0, U81I1_0, U81I2_0, U81I3_0, U81I4_0, U81I5_0, U81I6_0, U81I7_0, U81I8_0, U81I9_0,
       U81I10_0, U81I11_0, U81I12_0, U81I13_0, U81I14_0, U91I0_0, U91I1_0, U91I2_0, U91I3_0,
       U91I4_0: IN MVL; FRA_H_0, FRA_H_1, FRA_H_2, FRA_H_3, FRA_H_4, FRA_H_5, FRA_H_6, FRA_H_7,
       FRA_H_8, FRA_H_9, FRA_H_10, FRA_H_11, FRA_H_12, FRA_H_13, FRA_H_14,
       EXP_H_0, EXP_H_1, EXP_H_2, EXP_H_3, EXP_H_4: OUT MVL );
END stg1;
architecture STRUCTURAL of stg1 is
  component c2 PORT ( R: IN MVL; A: IN MVL; B: IN MVL; Y: OUT MVL ); end component;
  for U11: c2 use entity work.c2(behavioral); for U12: c2 use entity work.c2(behavioral);
  component call2 PORT ( CL: IN MVL; R1: IN MVL; D1: OUT MVL; R2: IN MVL;
       D2: OUT MVL; R: OUT MVL; D: IN MVL );
  end component; for U21: call2 use entity work.call2(behavioral);
  component comparator15 PORT ( A0: IN MVL; A1: IN MVL; A2: IN MVL; A3: IN MVL; A4: IN MVL;
       A5: IN MVL; A6: IN MVL; A7: IN MVL; A8: IN MVL; A9: IN MVL; A10: IN MVL; A11: IN MVL;
       A12: IN MVL; A13: IN MVL; A14: IN MVL; B0: IN MVL; B1: IN MVL; B2: IN MVL; B3: IN MVL;
       B4: IN MVL; B5: IN MVL; B6: IN MVL; B7: IN MVL; B8: IN MVL; B9: IN MVL; B10: IN MVL;
       B11: IN MVL; B12: IN MVL; B13: IN MVL; B14: IN MVL; G: OUT MVL; E: OUT MVL; L: OUT MVL );
  end component; for U31: comparator15 use entity work.comparator15(behavioral);
  component htlatch15 PORT ( Lt: IN MVL; I0: IN MVL; I1: IN MVL; I2: IN MVL; I3: IN MVL;
       I4: IN MVL; I5: IN MVL; I6: IN MVL; I7: IN MVL; I8: IN MVL; I9: IN MVL; I10: IN MVL;
       I11: IN MVL; I12: IN MVL; I13: IN MVL; I14: IN MVL; Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL;
       Y3: OUT MVL; Y4: OUT MVL; Y5: OUT MVL; Y6: OUT MVL; Y7: OUT MVL; Y8: OUT MVL;
       Y9: OUT MVL; Y10: OUT MVL; Y11: OUT MVL; Y12: OUT MVL; Y13: OUT MVL; Y14: OUT MVL );
  end component; for U41: htlatch15 use entity work.htlatch15(behavioral);
  component htlatch5 PORT ( Lt: IN MVL; I0: IN MVL; I1: IN MVL; I2: IN MVL; I3: IN MVL;
       I4: IN MVL; Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL; Y3: OUT MVL; Y4: OUT MVL );
  end component; for U51: htlatch5 use entity work.htlatch5(behavioral);
  component ltlatch15 PORT ( Lt: IN MVL; I0: IN MVL; I1: IN MVL; I2: IN MVL; I3: IN MVL;
       I4: IN MVL; I5: IN MVL; I6: IN MVL; I7: IN MVL; I8: IN MVL; I9: IN MVL; I10: IN MVL;
       I11: IN MVL; I12: IN MVL; I13: IN MVL; I14: IN MVL; Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL;
       Y3: OUT MVL; Y4: OUT MVL; Y5: OUT MVL; Y6: OUT MVL; Y7: OUT MVL; Y8: OUT MVL;
       Y9: OUT MVL; Y10: OUT MVL; Y11: OUT MVL; Y12: OUT MVL; Y13: OUT MVL; Y14: OUT MVL );
  end component; for U61: ltlatch15 use entity work.ltlatch15(behavioral);
  component ltlatch5 PORT ( Lt: IN MVL; I0: IN MVL; I1: IN MVL; I2: IN MVL; I3: IN MVL;
       I4: IN MVL; Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL; Y3: OUT MVL; Y4: OUT MVL );
  end component; for U71: ltlatch5 use entity work.ltlatch5(behavioral);
  component mux2x15 PORT ( S: IN MVL; I0_0: IN MVL; I0_1: IN MVL; I1_0: IN MVL; I1_1: IN MVL;
       I2_0: IN MVL; I2_1: IN MVL; I3_0: IN MVL; I3_1: IN MVL; I4_0: IN MVL; I4_1: IN MVL;
       I5_0: IN MVL; I5_1: IN MVL; I6_0: IN MVL; I6_1: IN MVL; I7_0: IN MVL; I7_1: IN MVL;
       I8_0: IN MVL; I8_1: IN MVL; I9_0: IN MVL; I9_1: IN MVL; I10_0: IN MVL; I10_1: IN MVL;
       I11_0: IN MVL; I11_1: IN MVL; I12_0: IN MVL; I12_1: IN MVL; I13_0: IN MVL;
       I13_1: IN MVL; I14_0: IN MVL; I14_1: IN MVL; Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL;
       Y3: OUT MVL; Y4: OUT MVL; Y5: OUT MVL; Y6: OUT MVL; Y7: OUT MVL; Y8: OUT MVL;
       Y9: OUT MVL; Y10: OUT MVL; Y11: OUT MVL; Y12: OUT MVL; Y13: OUT MVL; Y14: OUT MVL );
  end component; for U81: mux2x15 use entity work.mux2x15(behavioral);
  component mux2x5 PORT ( S: IN MVL; I0_0: IN MVL; I0_1: IN MVL; I1_0: IN MVL; I1_1: IN MVL;
       I2_0: IN MVL; I2_1: IN MVL; I3_0: IN MVL; I3_1: IN MVL; I4_0: IN MVL; I4_1: IN MVL;
       Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL; Y3: OUT MVL; Y4: OUT MVL );
  end component; for U91: mux2x5 use entity work.mux2x5(behavioral);
  component not1 PORT ( A: IN MVL; B: OUT MVL ); end component;
  for U101: not1 use entity work.not1(behavioral); for U102: not1 use entity work.not1(behavioral);
  component select1 PORT ( R: IN MVL; Bool: IN MVL; I: IN MVL; Yt: OUT MVL; Yf: OUT MVL );
  end component; for U111: select1 use entity work.select1(behavioral);
  for U112: select1 use entity work.select1(behavioral);
```

```
  component toggle2 PORT ( R: IN MVL; I: IN MVL; Yd: OUT MVL; Yn: OUT MVL ); end component;
  for U121: toggle2 use entity work.toggle2(behavioral);
  for U122: toggle2 use entity work.toggle2(behavioral);
  for U123: toggle2 use entity work.toggle2(behavioral);
  for U124: toggle2 use entity work.toggle2(behavioral);
  component xor2 PORT ( A: IN MVL; B: IN MVL; Y: OUT MVL ); end component;
  for U131: xor2 use entity work.xor2(behavioral); for U132: xor2 use entity work.xor2(behavioral);
  for U133: xor2 use entity work.xor2(behavioral); for U134: xor2 use entity work.xor2(behavioral);
  for U135: xor2 use entity work.xor2(behavioral); for U136: xor2 use entity work.xor2(behavioral);
  for U137: xor2 use entity work.xor2(behavioral); for U138: xor2 use entity work.xor2(behavioral);
  signal buffer11: MVL_VECTOR( 0 to 0); signal bus151: MVL_VECTOR( 0 to 14);
  signal bus152: MVL_VECTOR( 0 to 14); signal bus153: MVL_VECTOR( 0 to 14);
  signal bus154: MVL_VECTOR( 0 to 14); signal bus155: MVL_VECTOR( 0 to 14);
  signal bus51: MVL_VECTOR( 0 to 4); signal bus52: MVL_VECTOR( 0 to 4);
  signal bus53: MVL_VECTOR( 0 to 4); signal wire11: MVL;
  signal wire12: MVL; signal wire13: MVL; signal wire14: MVL; signal wire15: MVL; signal wire16: MVL;
  signal wire17: MVL; signal wire18: MVL; signal wire19: MVL; signal wire110: MVL; signal wire111: MVL;
  signal wire112: MVL; signal wire113: MVL; signal wire114: MVL; signal wire115: MVL; signal wire116: MVL;
  signal wire117: MVL; signal wire118: MVL; signal wire119: MVL; signal wire120: MVL; signal wire121: MVL;
  signal wire122: MVL; signal wire123: MVL; signal wire124: MVL; signal wire125: MVL; signal wire126: MVL;
  signal wire127: MVL; signal wire128: MVL; signal wire129: MVL; signal wire130: MVL;
begin
  bus153( 1 ) <= bus154( 0 ); bus153( 2 ) <= bus154( 1 ); bus153( 3 ) <= bus154( 2 );
  bus153( 4 ) <= bus154( 3 ); bus153( 5 ) <= bus154( 4 ); bus153( 6 ) <= bus154( 5 );
  bus153( 7 ) <= bus154( 6 ); bus153( 8 ) <= bus154( 7 ); bus153( 9 ) <= bus154( 8 );
  bus153( 10 ) <= bus154( 9 ); bus153( 11 ) <= bus154( 10 ); bus153( 12 ) <= bus154( 11 );
  bus153( 13 ) <= bus154( 12 ); bus153( 14 ) <= bus154( 13 ); buffer11( 0 ) <= wire129;
  wire130 <= buffer11( 0 ); bus154( 14 ) <= '0'; bus155( 0 ) <= '0'; bus155( 1 ) <= '0';
  bus155( 2 ) <= '0'; bus155( 3 ) <= '0'; bus155( 4 ) <= '0'; bus155( 5 ) <= '0';
  bus155( 6 ) <= '0'; bus155( 7 ) <= '0'; bus155( 8 ) <= '0'; bus155( 9 ) <= '0';
  bus155( 10 ) <= '0'; bus155( 11 ) <= '0'; bus155( 12 ) <= '0'; bus155( 13 ) <= '0';
  bus155( 14 ) <= '1'; wire13 <= Reset_stg1; wire11 <= Rin_stg1;  wire12 <= Aout_stg1;
  Rout_stg1 <= wire115; Ain_stg1 <= wire116; FRA_H_0 <= bus151( 0 ); FRA_H_1 <= bus151( 1 );
  FRA_H_2 <= bus151( 2 ); FRA_H_3 <= bus151( 3 ); FRA_H_4 <= bus151( 4 ); FRA_H_5 <= bus151( 5 );
  FRA_H_6 <= bus151( 6 ); FRA_H_7 <= bus151( 7 ); FRA_H_8 <= bus151( 8 ); FRA_H_9 <= bus151( 9 );
  FRA_H_10 <= bus151( 10 ); FRA_H_11 <= bus151( 11 ); FRA_H_12 <= bus151( 12 );
  FRA_H_13 <= bus151( 13 ); FRA_H_14 <= bus151( 14 ); EXP_H_0 <= bus52( 0 ); EXP_H_1 <= bus52( 1 );
  EXP_H_2 <= bus52( 2 ); EXP_H_3 <= bus52( 3 );  EXP_H_4 <= bus52( 4 );
  U11: c2 PORT map ( R => wire13, A => wire11, B => wire16, Y => wire14 );
  U12: c2 PORT map ( R => wire13, A => wire118, B => wire17, Y => wire15 );
  U21: call2 PORT map ( CL => wire13, R1 => wire14, D1 => wire116,
        R2 => wire19, D2 => wire117, R => wire118, D => open );
  U31: comparator15 PORT map ( A0 => bus153( 0 ), A1 => bus153( 1 ), A2 => bus153( 2 ), A3 => bus153( 3 ),
        A4 => bus153( 4 ), A5 => bus153( 5 ), A6 => bus153( 6 ), A7 => bus153( 7 ), A8 => bus153( 8 ),
        A9 => bus153( 9 ), A10 => bus153( 10 ), A11 => bus153( 11 ), A12 => bus153( 12 ),
        A13 => bus153( 13 ), A14 => bus153( 14 ), B0 => bus155( 0 ), B1 => bus155( 1 ),
        B2 => bus155( 2 ), B3 => bus155( 3 ), B4 => bus155( 4 ), B5 => bus155( 5 ), B6 => bus155( 6 ),
        B7 => bus155( 7 ), B8 => bus155( 8 ), B9 => bus155( 9 ), B10 => bus155( 10 ),
        B11 => bus155( 11 ), B12 => bus155( 12 ), B13 => bus155( 13 ), B14 => bus155( 14 ),
        G => open, E => wire130, L => open );
  U41: htlatch15 PORT map ( Lt => wire15, I0 => bus151( 0 ), I1 => bus151( 1 ), I2 => bus151( 2 ),
        I3 => bus151( 3 ), I4 => bus151( 4 ), I5 => bus151( 5 ), I6 => bus151( 6 ), I7 => bus151( 7 ),
        I8 => bus151( 8 ), I9 => bus151( 9 ), I10 => bus151( 10 ), I11 => bus151( 11 ),
        I12 => bus151( 12 ), I13 => bus151( 13 ), I14 => bus151( 14 ), Y0 => bus153( 0 ),
        Y1 => bus154( 0 ), Y2 => bus154( 1 ), Y3 => bus154( 2 ), Y4 => bus154( 3 ), Y5 => bus154( 4 ),
        Y6 => bus154( 5 ), Y7 => bus154( 6 ), Y8 => bus154( 7 ), Y9 => bus154( 8 ), Y10 => bus154( 9 ),
        Y11 => bus154( 10 ), Y12 => bus154( 11 ), Y13 => bus154( 12 ), Y14 => bus154( 13 ) );
```

```
  U51: htlatch5 PORT map ( Lt => wire15, I0 => bus52( 0 ), I1 => bus52( 1 ), I2 => bus52( 2 ),
         I3 => bus52( 3 ), I4 => bus52( 4 ), Y0 => bus53( 0 ), Y1 => bus53( 1 ), Y2 => bus53( 2 ),
         Y3 => bus53( 3 ), Y4 => bus53( 4 ) );
  U61: ltlatch15 PORT map ( Lt => wire15, I0 => bus152( 0 ), I1 => bus152( 1 ), I2 => bus152( 2 ),
         I3 => bus152( 3 ), I4 => bus152( 4 ), I5 => bus152( 5 ), I6 => bus152( 6 ), I7 => bus152( 7 ),
         I8 => bus152( 8 ), I9 => bus152( 9 ), I10 => bus152( 10 ), I11 => bus152( 11 ),
         I12 => bus152( 12 ), I13 => bus152( 13 ), I14 => bus152( 14 ), Y0 => bus151( 0 ),
         Y1 => bus151( 1 ), Y2 => bus151( 2 ), Y3 => bus151( 3 ), Y4 => bus151( 4 ), Y5 => bus151( 5 ),
         Y6 => bus151( 6 ), Y7 => bus151( 7 ), Y8 => bus151( 8 ), Y9 => bus151( 9 ), Y10 => bus151( 10 ),
         Y11 => bus151( 11 ), Y12 => bus151( 12 ), Y13 => bus151( 13 ), Y14 => bus151( 14 ) );
  U71: ltlatch5 PORT map ( Lt => wire15, I0 => bus51( 0 ), I1 => bus51( 1 ), I2 => bus51( 2 ),
         I3 => bus51( 3 ), I4 => bus51( 4 ), Y0 => bus52( 0 ), Y1 => bus52( 1 ), Y2 => bus52( 2 ),
         Y3 => bus52( 3 ), Y4 => bus52( 4 ) );
  U81: mux2x15 PORT map ( S => wire113, I0_0 => U81I0_0, I0_1 => bus154( 0 ), I1_0 => U81I1_0,
         I1_1 => bus154( 1 ), I2_0 => U81I2_0, I2_1 => bus154( 2 ), I3_0 => U81I3_0, I3_1 => bus154( 3 ),
         I4_0 => U81I4_0, I4_1 => bus154( 4 ), I5_0 => U81I5_0, I5_1 => bus154( 5 ), I6_0 => U81I6_0,
         I6_1 => bus154( 6 ), I7_0 => U81I7_0, I7_1 => bus154( 7 ), I8_0 => U81I8_0, I8_1 => bus154( 8 ),
         I9_0 => U81I9_0, I9_1 => bus154( 9 ), I10_0 => U81I10_0, I10_1 => bus154( 10 ), I11_0 => U81I11_0,
         I11_1 => bus154( 11 ), I12_0 => U81I12_0, I12_1 => bus154( 12 ), I13_0 => U81I13_0,
         I13_1 => bus154( 13 ), I14_0 => U81I14_0, I14_1 => bus154( 14 ), Y0 => bus152( 0 ),
         Y1 => bus152( 1 ), Y2 => bus152( 2 ), Y3 => bus152( 3 ), Y4 => bus152( 4 ), Y5 => bus152( 5 ),
         Y6 => bus152( 6 ), Y7 => bus152( 7 ), Y8 => bus152( 8 ), Y9 => bus152( 9 ), Y10 => bus152( 10 ),
         Y11 => bus152( 11 ), Y12 => bus152( 12 ), Y13 => bus152( 13 ), Y14 => bus152( 14 ) );
  U91: mux2x5 PORT map ( S => wire113, I0_0 => U91I0_0, I0_1 => bus53( 0 ), I1_0 => U91I1_0,
         I1_1 => bus53( 1 ), I2_0 => U91I2_0, I2_1 => bus53( 2 ), I3_0 => U91I3_0, I3_1 => bus53( 3 ),
         I4_0 => U91I4_0, I4_1 => bus53( 4 ), Y0 => bus51( 0 ), Y1 => bus51( 1 ), Y2 => bus51( 2 ),
         Y3 => bus51( 3 ), Y4 => bus51( 4 ) );
  U101: not1 PORT map ( A => wire111, B => wire16 ); U102: not1 PORT map ( A => wire18, B => wire17 );
  U111: select1 PORT map ( R => wire13, Bool => wire129, I => wire116, Yt => wire119, Yf => wire120 );
  U112: select1 PORT map ( R => wire13, Bool => wire129, I => wire117, Yt => wire121, Yf => wire122 );
  U121: toggle2 PORT map ( R => wire13, I => wire12, Yd => wire123, Yn => wire124 );
  U122: toggle2 PORT map ( R => wire13, I => wire119, Yd => wire125, Yn => wire126 );
  U123: toggle2 PORT map ( R => wire13, I => wire120, Yd => wire127, Yn => open );
  U124: toggle2 PORT map ( R => wire13, I => wire122, Yd => wire128, Yn => open );
  U131: xor2 PORT map ( A => wire110, B => wire12, Y => wire18 );
  U132: xor2 PORT map ( A => wire114, B => wire128, Y => wire19 );
  U133: xor2 PORT map ( A => wire125, B => wire114, Y => wire110 );
  U134: xor2 PORT map ( A => wire112, B => wire125, Y => wire111 );
  U135: xor2 PORT map ( A => wire127, B => wire124, Y => wire112 );
  U136: xor2 PORT map ( A => wire123, B => wire125, Y => wire113 );
  U137: xor2 PORT map ( A => wire121, B => wire126, Y => wire114 );
  U138: xor2 PORT map ( A => wire122, B => wire120, Y => wire115 );
end STRUCTURAL;
```

D.2.1.3 VHDL of the stg2

```
library XL;  use XL.XL_STD.all;
ENTITY stg2 IS
   PORT ( Reset_stg2 : IN MVL;
         Rin_stg2, Aout_stg2 : IN MVL;
         Ain_stg2, Rout_stg2 : OUT MVL;
         U21I0, U21I1, U21I2, U21I3, U21I4, U21I5, U21I6, U21I7,
         U21I8, U21I9, U21I10, U21I11, U21I12, U21I13, U21I14,
         U31I0, U31I1, U31I2, U31I3, U31I4: IN MVL;
         FRA_RM_0, FRA_RM_3, FRA_RM_4, FRA_RM_5, FRA_RM_6,
         FRA_RM_7, FRA_RM_8, FRA_RM_9, FRA_RM_10,
         EXP_RM_0, EXP_RM_1, EXP_RM_2, EXP_RM_3, EXP_RM_4: OUT MVL );
END stg2;
architecture STRUCTURAL of stg2 is
   component c2
      PORT ( R: IN MVL A: IN MVL B: IN MVL Y: OUT MVL );
   end component;
   for U11: c2 use entity work.c2(behavioral);
   component ltlatch15
      PORT ( Lt: IN MVL I0: IN MVL I1: IN MVL I2: IN MVL;
         I3: IN MVL I4: IN MVL I5: IN MVL I6: IN MVL;
         I7: IN MVL I8: IN MVL I9: IN MVL I10: IN MVL;
         I11: IN MVL I12: IN MVL I13: IN MVL I14: IN MVL;
         Y0: OUT MVL Y1: OUT MVL Y2: OUT MVL Y3: OUT MVL;
         Y4: OUT MVL Y5: OUT MVL Y6: OUT MVL Y7: OUT MVL;
         Y8: OUT MVL Y9: OUT MVL Y10: OUT MVL Y11: OUT MVL;
         Y12: OUT MVL Y13: OUT MVL Y14: OUT MVL );
   end component;
   for U21: ltlatch15 use entity work.ltlatch15(behavioral);
   component ltlatch5
      PORT ( Lt: IN MVL I0: IN MVL I1: IN MVL I2: IN MVL;
         I3: IN MVL I4: IN MVL Y0: OUT MVL Y1: OUT MVL;
         Y2: OUT MVL Y3: OUT MVL Y4: OUT MVL );
   end component;
   for U31: ltlatch5 use entity work.ltlatch5(behavioral);
   component not1 PORT ( A: IN MVL; B: OUT MVL ); end component;
   for U41: not1 use entity work.not1(behavioral);
   signal wire11: MVL; signal wire12: MVL; signal wire13: MVL;
   signal wire14: MVL; signal wire15: MVL;
begin
   wire15 <= Reset_stg2; wire13 <= Rin_stg2; wire14 <= Aout_stg2;
   Rout_stg2 <= wire11; Ain_stg2 <= wire11;
   U11: c2 PORT map ( R => wire15, A => wire13, B => wire12, Y => wire11 );
   U21: ltlatch15 PORT map ( Lt => wire11, I0 => U21I0, I1 => U21I1,
            I2 => U21I2, I3 => U21I3, I4 => U21I4, I5 => U21I5, I6 => U21I6,
            I7 => U21I7, I8 => U21I8, I9 => U21I9, I10 => U21I10,
            I11 => U21I11, I12 => U21I12, I13 => U21I13, I14 => U21I14,
            Y0 => FRA_RM_0, Y1 => open, Y2 => open, Y3 => FRA_RM_3,
            Y4 => FRA_RM_4, Y5 => FRA_RM_5, Y6 => FRA_RM_6, Y7 => FRA_RM_7,
            Y8 => FRA_RM_8, Y9 => FRA_RM_9, Y10 => FRA_RM_10, Y11 => open,
            Y12 => open, Y13 => open, Y14 => open );
   U31: ltlatch5 PORT map ( Lt => wire11, I0 => U31I0, I1 => U31I1,
            I2 => U31I2, I3 => U31I3, I4 => U31I4, Y0 => EXP_RM_0,
            Y1 => EXP_RM_1, Y2 => EXP_RM_2, Y3 => EXP_RM_3, Y4 => EXP_RM_4 );
   U41: not1 PORT map ( A => wire14, B => wire12 );
end STRUCTURAL;
```

D.2.2 2-phase output

D.2.2.1 VHDL of the whole circuit

```
LIBRARY XL; USE XL.XL_STD.ALL;
ENTITY circuit2 IS PORT ( RIN, AOUT : IN MVL; AIN, ROUT : OUT MVL;
EA : IN MVL_VECTOR ( 0 TO 4 ); MA : IN MVL_VECTOR ( 0 TO 14 ); OP : IN MVL_VECTOR ( 1 DOWNTO 0 );
EO : OUT MVL_VECTOR ( 0 TO 4 ); MO : OUT MVL_VECTOR ( 0 TO 8 ); RESET : IN MVL );
END circuit2;
architecture STRUCTURAL of circuit2 is
  component stg1 PORT ( Reset_stg1 : IN MVL; Rin_stg1 : IN MVL; Aout_stg1 : IN MVL; Ain_stg1 : OUT MVL;
      Rout_stg1 : OUT MVL; U81I0_0: IN MVL; U81I1_0: IN MVL; U81I2_0: IN MVL; U81I3_0: IN MVL;
      U81I4_0: IN MVL; U81I5_0: IN MVL; U81I6_0: IN MVL; U81I7_0: IN MVL; U81I8_0: IN MVL;
      U81I9_0: IN MVL; U81I10_0: IN MVL; U81I11_0: IN MVL; U81I12_0: IN MVL; U81I13_0: IN MVL;
      U81I14_0: IN MVL; U91I0_0: IN MVL; U91I1_0: IN MVL; U91I2_0: IN MVL; U91I3_0: IN MVL;
      U91I4_0: IN MVL; FRA_H_0: OUT MVL; FRA_H_1: OUT MVL; FRA_H_2: OUT MVL; FRA_H_3: OUT MVL;
      FRA_H_4: OUT MVL; FRA_H_5: OUT MVL; FRA_H_6: OUT MVL; FRA_H_7: OUT MVL; FRA_H_8: OUT MVL;
      FRA_H_9: OUT MVL; FRA_H_10: OUT MVL; FRA_H_11: OUT MVL; FRA_H_12: OUT MVL;
      FRA_H_13: OUT MVL; FRA_H_14: OUT MVL; EXP_H_0: OUT MVL; EXP_H_1: OUT MVL;
      EXP_H_2: OUT MVL; EXP_H_3: OUT MVL; EXP_H_4: OUT MVL );
  end component; for Ustg1: stg1 use entity work.stg1(structural);
  component stg2 PORT ( Reset_stg2 : IN MVL; Rin_stg2 : IN MVL; Aout_stg2 : IN MVL; Ain_stg2 : OUT MVL;
      Rout_stg2 : OUT MVL; U21I0: IN MVL; U21I1: IN MVL; U21I2: IN MVL; U21I3: IN MVL; U21I4: IN MVL;
      U21I5: IN MVL; U21I6: IN MVL; U21I7: IN MVL; U21I8: IN MVL; U21I9: IN MVL; U21I10: IN MVL;
      U21I11: IN MVL; U21I12: IN MVL; U21I13: IN MVL; U21I14: IN MVL; U31I0: IN MVL; U31I1: IN MVL;
      U31I2: IN MVL; U31I3: IN MVL; U31I4: IN MVL; FRA_RM_0: OUT MVL; FRA_RM_3: OUT MVL;
      FRA_RM_4: OUT MVL; FRA_RM_5: OUT MVL; FRA_RM_6: OUT MVL; FRA_RM_7: OUT MVL;
      FRA_RM_8: OUT MVL; FRA_RM_9: OUT MVL; FRA_RM_10: OUT MVL; EXP_RM_0: OUT MVL;
      EXP_RM_1: OUT MVL; EXP_RM_2: OUT MVL; EXP_RM_3: OUT MVL; EXP_RM_4: OUT MVL );
  end component; for Ustg2: stg2 use entity work.stg2(structural);
  signal bus151: MVL_VECTOR( 0 to 14); signal bus152: MVL_VECTOR( 0 to 14); signal bus51: MVL_VECTOR( 0 to 4);
  signal bus52: MVL_VECTOR( 0 to 4); signal bus53: MVL_VECTOR( 0 to 4); signal bus81: MVL_VECTOR( 0 to 7);
  signal wire11: MVL; signal wire12: MVL; signal wire13: MVL; signal wire14: MVL; signal wire15: MVL;
  signal wire16: MVL; signal wire17: MVL; signal wire18: MVL;
begin
  wire13 <= RESET; wire11 <= RIN; wire12 <= AOUT; ROUT <= wire18; AIN <= wire15; bus51( 0 ) <= EA(0);
  bus51( 1 ) <= EA(1); bus51( 2 ) <= EA(2); bus51( 3 ) <= EA(3); bus51( 4 ) <= EA(4); bus151( 0 ) <= MA(0);
  bus151( 1 ) <= MA(1); bus151( 2 ) <= MA(2); bus151( 3 ) <= MA(3); bus151( 4 ) <= MA(4); bus151( 5 ) <= MA(5);
  bus151( 6 ) <= MA(6); bus151( 7 ) <= MA(7); bus151( 8 ) <= MA(8); bus151( 9 ) <= MA(9); bus151( 10 ) <= MA(10);
  bus151( 11 ) <= MA(11); bus151( 12 ) <= MA(12); bus151( 13 ) <= MA(13); bus151( 14 ) <= MA(14); open <= OP(0);
  open <= OP(1); EO(0) <= bus53( 0 ); EO(1) <= bus53( 1 ); EO(2) <= bus53( 2 ); EO(3) <= bus53( 3 );
  EO(4) <= bus53( 4 ); MO(0) <= wire14; MO(1) <= bus81( 0 ); MO(2) <= bus81( 1 ); MO(3) <= bus81( 2 );
  MO(4) <= bus81( 3 ); MO(5) <= bus81( 4 ); MO(6) <= bus81( 5 ); MO(7) <= bus81( 6 ); MO(8) <= bus81( 7 );
  Ustg1: stg1 PORT map ( Reset_stg1 => wire13, Rin_stg1 => wire11, Aout_stg1 => wire17, Ain_stg1 => wire15,
       Rout_stg1 => wire16, U81I0_0 => bus151( 0 ), U81I1_0 => bus151( 1 ), U81I2_0 => bus151( 2 ), U81I3_0 => bus151( 3 ),
       U81I4_0 => bus151( 4 ), U81I5_0 => bus151( 5 ), U81I6_0 => bus151( 6 ), U81I7_0 => bus151( 7 ),
       U81I8_0 => bus151( 8 ), U81I9_0 => bus151( 9 ), U81I10_0 => bus151( 10 ), U81I11_0 => bus151( 11 ),
       U81I12_0 => bus151( 12 ), U81I13_0 => bus151( 13 ), U81I14_0 => bus151( 14 ), U91I0_0 => bus51( 0 ),
       U91I1_0 => bus51( 1 ), U91I2_0 => bus51( 2 ), U91I3_0 => bus51( 3 ), U91I4_0 => bus51( 4 ),
       FRA_H_0 => bus152( 0 ), FRA_H_1 => bus152( 1 ), FRA_H_2 => bus152( 2 ), FRA_H_3 => bus152( 3 ),
       FRA_H_4 => bus152( 4 ), FRA_H_5 => bus152( 5 ), FRA_H_6 => bus152( 6 ), FRA_H_7 => bus152( 7 ),
       FRA_H_8 => bus152( 8 ), FRA_H_9 => bus152( 9 ), FRA_H_10 => bus152( 10 ), FRA_H_11 => bus152( 11 ),
       FRA_H_12 => bus152( 12 ), FRA_H_13 => bus152( 13 ), FRA_H_14 => bus152( 14 ), EXP_H_0 => bus52( 0 ),
       EXP_H_1 => bus52( 1 ), EXP_H_2 => bus52( 2 ), EXP_H_3 => bus52( 3 ), EXP_H_4 => bus52( 4 ) );
  Ustg2: stg2 PORT map ( Reset_stg2 => wire13, Rin_stg2 => wire16, Aout_stg2 => wire12, Ain_stg2 => wire17,
       Rout_stg2 => wire18, U21I0 => bus152( 0 ), U21I1 => bus152( 1 ), U21I2 => bus152( 2 ), U21I3 => bus152( 3 ),
       U21I4 => bus152( 4 ), U21I5 => bus152( 5 ), U21I6 => bus152( 6 ), U21I7 => bus152( 7 ), U21I8 => bus152( 8 ),
       U21I9 => bus152( 9 ), U21I10 => bus152( 10 ), U21I11 => bus152( 11 ), U21I12 => bus152( 12 ),
       U21I13 => bus152( 13 ), U21I14 => bus152( 14 ), U31I0 => bus52( 0 ), U31I1 => bus52( 1 ), U31I2 => bus52( 2 ),
       U31I3 => bus52( 3 ), U31I4 => bus52( 4 ), FRA_RM_0 => wire14, FRA_RM_3 => bus81( 0 ),
       FRA_RM_4 => bus81( 1 ), FRA_RM_5 => bus81( 2 ), FRA_RM_6 => bus81( 3 ), FRA_RM_7 => bus81( 4 ),
       FRA_RM_8 => bus81( 5 ), FRA_RM_9 => bus81( 6 ), FRA_RM_10 => bus81( 7 ), EXP_RM_0 => bus53( 0 ),
       EXP_RM_1 => bus53( 1 ), EXP_RM_2 => bus53( 2 ), EXP_RM_3 => bus53( 3 ), EXP_RM_4 => bus53( 4 ) );
end STRUCTURAL;
```

D.2.2.2 VHDL of the stg1

```
library XL;  use XL.XL_STD.all;
ENTITY stg1 IS
   PORT ( Reset_stg1 : IN MVL; Rin_stg1, Aout_stg1 : IN MVL; Ain_stg1, Rout_stg1 : OUT MVL;
        U81I0_0, U81I1_0, U81I2_0, U81I3_0, U81I4_0, U81I5_0, U81I6_0, U81I7_0, U81I8_0, U81I9_0, U81I10_0,
        U81I11_0, U81I12_0, U81I13_0, U81I14_0, U91I0_0, U91I1_0, U91I2_0, U91I3_0, U91I4_0: IN MVL;
        FRA_H_0, FRA_H_1, FRA_H_2, FRA_H_3, FRA_H_4, FRA_H_5, FRA_H_6, FRA_H_7, FRA_H_8, FRA_H_9, FRA_H_10,
        FRA_H_11, FRA_H_12, FRA_H_13, FRA_H_14, EXP_H_0, EXP_H_1, EXP_H_2, EXP_H_3, EXP_H_4: OUT MVL );
END stg1;
architecture STRUCTURAL of stg1 is
   component c2 PORT ( R: IN MVL; A: IN MVL; B: IN MVL; Y: OUT MVL ); end component;
   for U11: c2 use entity work.c2(behavioral); for U12: c2 use entity work.c2(behavioral);
   component call2 PORT ( CL: IN MVL; R1: IN MVL; D1: OUT MVL; R2: IN MVL; D2: OUT MVL; R: OUT MVL; D: IN MVL );
   end component; for U21: call2 use entity work.call2(behavioral);
   component comparator15 PORT ( A0: IN MVL; A1: IN MVL; A2: IN MVL; A3: IN MVL; A4: IN MVL; A5: IN MVL; A6: IN MVL;
        A7: IN MVL; A8: IN MVL; A9: IN MVL; A10: IN MVL; A11: IN MVL; A12: IN MVL; A13: IN MVL; A14: IN MVL;
        B0: IN MVL; B1: IN MVL; B2: IN MVL; B3: IN MVL; B4: IN MVL; B5: IN MVL; B6: IN MVL; B7: IN MVL; B8: IN MVL;
        B9: IN MVL; B10: IN MVL; B11: IN MVL; B12: IN MVL; B13: IN MVL; B14: IN MVL; G: OUT MVL; E: OUT MVL; L: OUT MVL );
   end component; for U31: comparator15 use entity work.comparator15(behavioral);
   component htlatch15 PORT ( Lt: IN MVL; I0: IN MVL; I1: IN MVL; I2: IN MVL; I3: IN MVL; I4: IN MVL; I5: IN MVL;
        I6: IN MVL; I7: IN MVL; I8: IN MVL; I9: IN MVL; I10: IN MVL; I11: IN MVL; I12: IN MVL; I13: IN MVL; I14: IN MVL;
        Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL; Y3: OUT MVL; Y4: OUT MVL; Y5: OUT MVL; Y6: OUT MVL; Y7: OUT MVL;
        Y8: OUT MVL; Y9: OUT MVL; Y10: OUT MVL; Y11: OUT MVL; Y12: OUT MVL; Y13: OUT MVL; Y14: OUT MVL );
   end component; for U41: htlatch15 use entity work.htlatch15(behavioral);
   component htlatch5 PORT ( Lt: IN MVL; I0: IN MVL; I1: IN MVL; I2: IN MVL; I3: IN MVL;
        I4: IN MVL; Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL; Y3: OUT MVL; Y4: OUT MVL );
   end component; for U51: htlatch5 use entity work.htlatch5(behavioral);
   component ltlatch15 PORT ( Lt: IN MVL; I0: IN MVL; I1: IN MVL; I2: IN MVL; I3: IN MVL; I4: IN MVL; I5: IN MVL;
        I6: IN MVL; I7: IN MVL; I8: IN MVL; I9: IN MVL; I10: IN MVL; I11: IN MVL; I12: IN MVL; I13: IN MVL; I14: IN MVL;
        Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL; Y3: OUT MVL; Y4: OUT MVL; Y5: OUT MVL; Y6: OUT MVL; Y7: OUT MVL;
        Y8: OUT MVL; Y9: OUT MVL; Y10: OUT MVL; Y11: OUT MVL; Y12: OUT MVL; Y13: OUT MVL; Y14: OUT MVL );
   end component; for U61: ltlatch15 use entity work.ltlatch15(behavioral);
   component ltlatch5 PORT ( Lt: IN MVL; I0: IN MVL; I1: IN MVL; I2: IN MVL; I3: IN MVL;
        I4: IN MVL; Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL; Y3: OUT MVL; Y4: OUT MVL );
   end component; for U71: ltlatch5 use entity work.ltlatch5(behavioral);
   component mux2x15 PORT ( S: IN MVL; I0_0: IN MVL; I0_1: IN MVL; I1_0: IN MVL; I1_1: IN MVL; I2_0: IN MVL;
        I2_1: IN MVL; I3_0: IN MVL; I3_1: IN MVL; I4_0: IN MVL; I4_1: IN MVL; I5_0: IN MVL; I5_1: IN MVL;
        I6_0: IN MVL; I6_1: IN MVL; I7_0: IN MVL; I7_1: IN MVL; I8_0: IN MVL; I8_1: IN MVL; I9_0: IN MVL;
        I9_1: IN MVL; I10_0: IN MVL; I10_1: IN MVL; I11_0: IN MVL; I11_1: IN MVL; I12_0: IN MVL; I12_1: IN MVL;
        I13_0: IN MVL; I13_1: IN MVL; I14_0: IN MVL; I14_1: IN MVL; Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL;
        Y3: OUT MVL; Y4: OUT MVL; Y5: OUT MVL; Y6: OUT MVL; Y7: OUT MVL; Y8: OUT MVL; Y9: OUT MVL;
        Y10: OUT MVL; Y11: OUT MVL; Y12: OUT MVL; Y13: OUT MVL; Y14: OUT MVL );
   end component; for U81: mux2x15 use entity work.mux2x15(behavioral);
   component mux2x5 PORT ( S: IN MVL; I0_0: IN MVL; I0_1: IN MVL; I1_0: IN MVL; I1_1: IN MVL;
        I2_0: IN MVL; I2_1: IN MVL; I3_0: IN MVL; I3_1: IN MVL; I4_0: IN MVL; I4_1: IN MVL;
        Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL; Y3: OUT MVL; Y4: OUT MVL );
   end component; for U91: mux2x5 use entity work.mux2x5(behavioral);
   component not1 PORT ( A: IN MVL; B: OUT MVL ); end component;
   for U101: not1 use entity work.not1(behavioral); for U102: not1 use entity work.not1(behavioral);
   component select1 PORT ( R: IN MVL; Bool: IN MVL; I: IN MVL; Yt: OUT MVL; Yf: OUT MVL ); end component;
   for U111: select1 use entity work.select1(behavioral);
   component toggle2 PORT ( R: IN MVL; I: IN MVL; Yd: OUT MVL; Yn: OUT MVL ); end component;
   for U121: toggle2 use entity work.toggle2(behavioral);
   component xor2 PORT ( A: IN MVL; B: IN MVL; Y: OUT MVL ); end component;
   for U131: xor2 use entity work.xor2(behavioral); for U132: xor2 use entity work.xor2(behavioral);
   for U133: xor2 use entity work.xor2(behavioral);
   signal bus151: MVL_VECTOR( 0 to 14); signal bus152: MVL_VECTOR( 0 to 14); signal bus153: MVL_VECTOR( 0 to 14);
   signal bus154: MVL_VECTOR( 0 to 14); signal bus155: MVL_VECTOR( 0 to 14); signal bus51: MVL_VECTOR( 0 to 4);
   signal bus52: MVL_VECTOR( 0 to 4); signal bus53: MVL_VECTOR( 0 to 4); signal wire11: MVL; signal wire12: MVL;
   signal wire13: MVL; signal wire14: MVL; signal wire15: MVL; signal wire16: MVL; signal wire17: MVL; signal wire18: MVL;
   signal wire19: MVL; signal wire110: MVL; signal wire111: MVL; signal wire112: MVL; signal wire113: MVL;
   signal wire114: MVL; signal wire115: MVL; signal wire116: MVL; signal wire117: MVL; signal wire118: MVL;
```

```
begin
  bus153( 1 ) <= bus154( 0 ); bus153( 2 ) <= bus154( 1 ); bus153( 3 ) <= bus154( 2 ); bus153( 4 ) <= bus154( 3 );
  bus153( 5 ) <= bus154( 4 ); bus153( 6 ) <= bus154( 5 ); bus153( 7 ) <= bus154( 6 ); bus153( 8 ) <= bus154( 7 );
  bus153( 9 ) <= bus154( 8 ); bus153( 10 ) <= bus154( 9 ); bus153( 11 ) <= bus154( 10 ); bus153( 12 ) <= bus154( 11 );
  bus153( 13 ) <= bus154( 12 ); bus153( 14 ) <= bus154( 13 ); bus154( 14 ) <= ’0’; bus155( 0 ) <= ’0’; bus155( 1 ) <= ’0’;
  bus155( 2 ) <= ’0’; bus155( 3 ) <= ’0’; bus155( 4 ) <= ’0’; bus155( 5 ) <= ’0’; bus155( 6 ) <= ’0’; bus155( 7 ) <= ’0’;
  bus155( 8 ) <= ’0’; bus155( 9 ) <= ’0’; bus155( 10 ) <= ’0’; bus155( 11 ) <= ’0’; bus155( 12 ) <= ’0’; bus155( 13 ) <= ’0’;
  bus155( 14 ) <= ’1’; wire13 <= Reset_stg1; wire11 <= Rin_stg1; wire12 <= Aout_stg1; Rout_stg1 <= wire112;
  Ain_stg1 <= wire18; FRA_H_0 <= bus152( 0 ); FRA_H_1 <= bus152( 1 ); FRA_H_2 <= bus152( 2 ); FRA_H_3 <= bus152( 3 );
  FRA_H_4 <= bus152( 4 ); FRA_H_5 <= bus152( 5 ); FRA_H_6 <= bus152( 6 ); FRA_H_7 <= bus152( 7 ); FRA_H_8 <= bus152( 8 );
  FRA_H_9 <= bus152( 9 ); FRA_H_10 <= bus152( 10 ); FRA_H_11 <= bus152( 11 ); FRA_H_12 <= bus152( 12 );
  FRA_H_13 <= bus152( 13 ); FRA_H_14 <= bus152( 14 ); EXP_H_0 <= bus51( 0 ); EXP_H_1 <= bus51( 1 );
  EXP_H_2 <= bus51( 2 ); EXP_H_3 <= bus51( 3 ); EXP_H_4 <= bus51( 4 );
  U11: c2 PORT map ( R => wire13, A => wire11, B => wire15, Y => wire14 );
  U12: c2 PORT map ( R => wire13, A => wire110, B => wire114, Y => wire113 );
  U21: call2 PORT map ( CL => wire13, R1 => wire14, D1 => wire18, R2 => wire111, D2 => wire19,
      R => wire110, D => wire116 );
  U31: comparator15 PORT map ( A0 => bus153( 0 ), A1 => bus153( 1 ), A2 => bus153( 2 ), A3 => bus153( 3 ),
      A4 => bus153( 4 ), A5 => bus153( 5 ), A6 => bus153( 6 ), A7 => bus153( 7 ), A8 => bus153( 8 ),
      A9 => bus153( 9 ), A10 => bus153( 10 ), A11 => bus153( 11 ), A12 => bus153( 12 ), A13 => bus153( 13 ),
      A14 => bus153( 14 ), B0 => bus155( 0 ), B1 => bus155( 1 ), B2 => bus155( 2 ), B3 => bus155( 3 ),
      B4 => bus155( 4 ), B5 => bus155( 5 ), B6 => bus155( 6 ), B7 => bus155( 7 ), B8 => bus155( 8 ),
      B9 => bus155( 9 ), B10 => bus155( 10 ), B11 => bus155( 11 ), B12 => bus155( 12 ), B13 => bus155( 13 ),
      B14 => bus155( 14 ), G => open, E => wire118, L => open );
  U41: htlatch15 PORT map ( Lt => wire115, I0 => bus152( 0 ), I1 => bus152( 1 ), I2 => bus152( 2 ), I3 => bus152( 3 ),
      I4 => bus152( 4 ), I5 => bus152( 5 ), I6 => bus152( 6 ), I7 => bus152( 7 ), I8 => bus152( 8 ), I9 => bus152( 9 ),
      I10 => bus152( 10 ), I11 => bus152( 11 ), I12 => bus152( 12 ), I13 => bus152( 13 ), I14 => bus152( 14 ),
      Y0 => bus153( 0 ), Y1 => bus154( 0 ), Y2 => bus154( 1 ), Y3 => bus154( 2 ), Y4 => bus154( 3 ), Y5 => bus154( 4 ),
      Y6 => bus154( 5 ), Y7 => bus154( 6 ), Y8 => bus154( 7 ), Y9 => bus154( 8 ), Y10 => bus154( 9 ),
      Y11 => bus154( 10 ), Y12 => bus154( 11 ), Y13 => bus154( 12 ), Y14 => bus154( 13 ) );
  U51: htlatch5 PORT map ( Lt => wire115, I0 => bus51( 0 ), I1 => bus51( 1 ), I2 => bus51( 2 ), I3 => bus51( 3 ),
      I4 => bus51( 4 ), Y0 => bus53( 0 ), Y1 => bus53( 1 ), Y2 => bus53( 2 ), Y3 => bus53( 3 ), Y4 => bus53( 4 ) );
  U61: ltlatch15 PORT map ( Lt => wire115, I0 => bus151( 0 ), I1 => bus151( 1 ), I2 => bus151( 2 ), I3 => bus151( 3 ),
      I4 => bus151( 4 ), I5 => bus151( 5 ), I6 => bus151( 6 ), I7 => bus151( 7 ), I8 => bus151( 8 ), I9 => bus151( 9 ),
      I10 => bus151( 10 ), I11 => bus151( 11 ), I12 => bus151( 12 ), I13 => bus151( 13 ), I14 => bus151( 14 ),
      Y0 => bus152( 0 ), Y1 => bus152( 1 ), Y2 => bus152( 2 ), Y3 => bus152( 3 ), Y4 => bus152( 4 ), Y5 => bus152( 5 ),
      Y6 => bus152( 6 ), Y7 => bus152( 7 ), Y8 => bus152( 8 ), Y9 => bus152( 9 ), Y10 => bus152( 10 ),
      Y11 => bus152( 11 ), Y12 => bus152( 12 ), Y13 => bus152( 13 ), Y14 => bus152( 14 ) );
  U71: ltlatch5 PORT map ( Lt => wire115, I0 => bus52( 0 ), I1 => bus52( 1 ), I2 => bus52( 2 ), I3 => bus52( 3 ),
      I4 => bus52( 4 ), Y0 => bus51( 0 ), Y1 => bus51( 1 ), Y2 => bus51( 2 ), Y3 => bus51( 3 ), Y4 => bus51( 4 ) );
  U81: mux2x15 PORT map ( S => wire16, I0_0 => U81I0_0, I0_1 => bus154( 0 ), I1_0 => U81I1_0, I1_1 => bus154( 1 ),
      I2_0 => U81I2_0, I2_1 => bus154( 2 ), I3_0 => U81I3_0, I3_1 => bus154( 3 ), I4_0 => U81I4_0,
      I4_1 => bus154( 4 ), I5_0 => U81I5_0, I5_1 => bus154( 5 ), I6_0 => U81I6_0, I6_1 => bus154( 6 ),
      I7_0 => U81I7_0, I7_1 => bus154( 7 ), I8_0 => U81I8_0, I8_1 => bus154( 8 ), I9_0 => U81I9_0,
      I9_1 => bus154( 9 ), I10_0 => U81I10_0, I10_1 => bus154( 10 ), I11_0 => U81I11_0, I11_1 => bus154( 11 ),
      I12_0 => U81I12_0, I12_1 => bus154( 12 ), I13_0 => U81I13_0, I13_1 => bus154( 13 ), I14_0 => U81I14_0,
      I14_1 => bus154( 14 ), Y0 => bus151( 0 ), Y1 => bus151( 1 ), Y2 => bus151( 2 ), Y3 => bus151( 3 ),
      Y4 => bus151( 4 ), Y5 => bus151( 5 ), Y6 => bus151( 6 ), Y7 => bus151( 7 ), Y8 => bus151( 8 ),
      Y9 => bus151( 9 ), Y10 => bus151( 10 ), Y11 => bus151( 11 ), Y12 => bus151( 12 ),
      Y13 => bus151( 13 ), Y14 => bus151( 14 ) );
  U91: mux2x5 PORT map ( S => wire16, I0_0 => U91I0_0, I0_1 => bus53( 0 ), I1_0 => U91I1_0, I1_1 => bus53( 1 ),
      I2_0 => U91I2_0, I2_1 => bus53( 2 ), I3_0 => U91I3_0, I3_1 => bus53( 3 ), I4_0 => U91I4_0,
      I4_1 => bus53( 4 ), Y0 => bus52( 0 ), Y1 => bus52( 1 ), Y2 => bus52( 2 ), Y3 => bus52( 3 ), Y4 => bus52( 4 ) );
  U101: not1 PORT map ( A => wire12, B => wire15 ); U102: not1 PORT map ( A => wire117, B => wire114 );
  U111: select1 PORT map ( R => wire13, Bool => wire118, I => wire17, Yt => wire111, Yf => wire112 );
  U121: toggle2 PORT map ( R => wire13, I => wire115, Yd => wire116, Yn => wire117 );
  U131: xor2 PORT map ( A => wire18, B => wire112, Y => wire16 );
  U132: xor2 PORT map ( A => wire18, B => wire19, Y => wire17 );
  U133: xor2 PORT map ( A => wire113, B => wire116, Y => wire115 );
end STRUCTURAL;
```

D.2.2.3 VHDL of the stg2

```
library XL;  use XL.XL_STD.all;
ENTITY stg2 IS
  PORT ( Reset_stg2 : IN MVL; Rin_stg2, Aout_stg2 : IN MVL; Ain_stg2, Rout_stg2 : OUT MVL;
       U21I0, U21I1, U21I2, U21I3, U21I4, U21I5, U21I6, U21I7, U21I8, U21I9, U21I10, U21I11,
       U21I12, U21I13, U21I14, U31I0, U31I1, U31I2, U31I3, U31I4: IN MVL;
       FRA_RM_0, FRA_RM_3, FRA_RM_4, FRA_RM_5, FRA_RM_6, FRA_RM_7, FRA_RM_8, FRA_RM_9,
       FRA_RM_10, EXP_RM_0, EXP_RM_1, EXP_RM_2, EXP_RM_3, EXP_RM_4: OUT MVL );
END stg2;
architecture STRUCTURAL of stg2 is
  component c2 PORT ( R: IN MVL; A: IN MVL; B: IN MVL;  Y: OUT MVL ); end component;
  for U11: c2 use entity work.c2(behavioral);
  component ltlatch15 PORT ( Lt: IN MVL; I0: IN MVL; I1: IN MVL; I2: IN MVL; I3: IN MVL;
       I4: IN MVL; I5: IN MVL; I6: IN MVL; I7: IN MVL; I8: IN MVL; I9: IN MVL; I10: IN MVL;
       I11: IN MVL; I12: IN MVL; I13: IN MVL; I14: IN MVL; Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL;
       Y3: OUT MVL; Y4: OUT MVL; Y5: OUT MVL; Y6: OUT MVL; Y7: OUT MVL; Y8: OUT MVL; Y9: OUT MVL;
       Y10: OUT MVL; Y11: OUT MVL; Y12: OUT MVL; Y13: OUT MVL; Y14: OUT MVL );
  end component;
  for U21: ltlatch15 use entity work.ltlatch15(behavioral);
  component ltlatch5 PORT ( Lt: IN MVL; I0: IN MVL; I1: IN MVL; I2: IN MVL; I3: IN MVL;
       I4: IN MVL; Y0: OUT MVL; Y1: OUT MVL; Y2: OUT MVL; Y3: OUT MVL; Y4: OUT MVL );
  end component;
  for U31: ltlatch5 use entity work.ltlatch5(behavioral);
  component not1 PORT ( A: IN MVL; B: OUT MVL ); end component;
  for U41: not1 use entity work.not1(behavioral);
  component toggle2 PORT ( R: IN MVL; I: IN MVL; Yd: OUT MVL; Yn: OUT MVL ); end component;
  for U51: toggle2 use entity work.toggle2(behavioral);
  component xor2 PORT ( A: IN MVL; B: IN MVL; Y: OUT MVL ); end component;
  for U61: xor2 use entity work.xor2(behavioral);
  signal wire11: MVL; signal wire12: MVL; signal wire13: MVL; signal wire14: MVL;
  signal wire15: MVL; signal wire16: MVL; signal wire17: MVL; signal wire18: MVL;
begin
  wire18 <= Reset_stg2; wire16 <= Rin_stg2; wire17 <= Aout_stg2; Rout_stg2 <= wire14; Ain_stg2 <= wire14;
  U11: c2 PORT map ( R => wire18, A => wire16, B => wire13, Y => wire11 );
  U21: ltlatch15 PORT map ( Lt => wire12, I0 => U21I0, I1 => U21I1, I2 => U21I2, I3 => U21I3, I4 => U21I4,
         I5 => U21I5, I6 => U21I6, I7 => U21I7, I8 => U21I8, I9 => U21I9, I10 => U21I10, I11 => U21I11,
         I12 => U21I12, I13 => U21I13, I14 => U21I14, Y0 => FRA_RM_0, Y1 => open, Y2 => open, Y3 => FRA_RM_3,
         Y4 => FRA_RM_4, Y5 => FRA_RM_5, Y6 => FRA_RM_6, Y7 => FRA_RM_7, Y8 => FRA_RM_8,
         Y9 => FRA_RM_9, Y10 => FRA_RM_10, Y11 => open, Y12 => open, Y13 => open, Y14 => open );
  U31: ltlatch5 PORT map ( Lt => wire12, I0 => U31I0, I1 => U31I1, I2 => U31I2, I3 => U31I3, I4 => U31I4,
         Y0 => EXP_RM_0, Y1 => EXP_RM_1, Y2 => EXP_RM_2, Y3 => EXP_RM_3, Y4 => EXP_RM_4 );
  U41: not1 PORT map ( A => wire15, B => wire13 );
  U51: toggle2 PORT map ( R => wire18, I => wire12, Yd => wire14, Yn => wire15 );
  U61: xor2 PORT map ( A => wire11, B => wire17, Y => wire12 );
end STRUCTURAL;
```

Appendix E

A Micropipelined Floating Point Adder/Subtractor

A behavioural description of a micropipelined floating point adder/Subtractor is shown in this appendix. The description is used to to test the synthesizer in Chapter 5.

A behavioural description

```
ENTITY F_P_Processor1 IS
  PORT ( RIN, AOUT : IN  MVL; AIN, ROUT : OUT  MVL; EA, EB : IN MVL_VECTOR ( 0 TO 4 );
       MA, MB : IN MVL_VECTOR ( 0 TO 8 ); OP : IN MVL_VECTOR ( 1 DOWNTO 0 );
       EO : OUT MVL_VECTOR ( 0 TO 4 ); MO : OUT MVL_VECTOR ( 0 TO 8 ); RESET : IN MVL );
END F_P_Processor1;
architecture BEHAVIOR of F_P_Processor1 is
begin U1: process
    variable sign_c, sign_d, sign_s, comp : MVL;
    variable QEA, QEB, EXP_DIFF1, EXP_DIFF2, EXP_DIFF, EXP_DIFFA : MVL_VECTOR ( 0 TO 4 );
    variable EXP_C, EXP_E, EXP_F, EXP_N, EXP_H, EXP_R, EXP_RM : MVL_VECTOR ( 0 TO 4 );
    variable QMA, QMB : MVL_VECTOR ( 0 TO 9 );
    variable FRA_C, FRA_D, FRA_DD, FRA_D1 : MVL_VECTOR ( 0 TO 12 );
    variable FRA_SC, FRA_SD, FRA_S, FRA_S1, FRA_S2, FRA_S3 : MVL_VECTOR ( 0 TO 13 );
    variable FRA_E, FRA_F, FRA_H, FRA_R, FRA_RR, FRA_RM : MVL_VECTOR ( 0 TO 14 );
    begin  QEA := EA; QEB := EB; QMA(0) := MA(0);  QMA(2 to 9) := MA(1 to 8); QMB(0) := MB(0); QMB(2 to 9) := MB(1 to 8);
        if EA = "00000" then QMA(1) := '0'; else QMA(1) := '1'; end if;
        if EB = "00000" 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 9) := QMA; fra_dd(0 to 9) := QMB;
        else exp_c := QEB; fra_c(0 to 9) := QMB; fra_dd(0 to 9) := QMA; end if;
        fra_c(10 to 12) := "000";  fra_dd(10 to 12) := "000"; fra_d1(0 to 10) := fra_dd(0 to 10);
        fra_d1(11) := fra_dd(11) or fra_dd(12); fra_d1(12) := fra_dd(12); exp_diffa := exp_diff;
        while ( exp_diffa > "00000" ) loop fra_d1 := sh_right( fra_d1, 1 ); exp_diffa := exp_diffa - "00001"; end loop;
        fra_sc(1 to 12) := fra_c(1 to 12); fra_sd(1 to 12) := fra_d1(1 to 12); sign_c := fra_c(0); sign_d := fra_d1(0);
        if sign_c = '1' then fra_sc(0) := '0'; else fra_sc(0) := fra_c(0);  end if;
        if sign_d = '1' then fra_sd(0) := '0'; else fra_sd(0) := fra_d1(0); end if;
        fra_s1(0 to 13) := fra_sc + fra_sd; fra_s2(0 to 13) := fra_sc - fra_sd; fra_s3(0 to 13) := fra_sd - fra_sc;
        if op = "00" then if sign_c = '0' and sign_d = '0' then fra_s := fra_s1;
                   else if sign_c = '0' and sign_d = '1' then fra_s := fra_s2;
                      else if sign_c = '1' and sign_d = '0' then fra_s := fra_s3;
                         else fra_s := fra_s1; end if; end if; end if;
        else if op = "01" then if sign_c = '0' and sign_d = '0' then fra_s := fra_s2;
                      else if sign_c = '0' and sign_d = '1' then fra_s := fra_s1;
                         else if sign_c = '1' and sign_d = '0' then fra_s := fra_s1;
                            else fra_s := fra_s3; end if; end if; end if;
           else fra_s := fra_sc + fra_sd; end if; end if;
        sign_s := fra_s(0);
        if op = "00" then if sign_s = '0' then if sign_c = '0' then if sign_d = '0' then fra_e(0) := '0';
                                                   else fra_e(0) := '0'; end if;
                                       else if sign_d = '0' then  fra_e(0) := '0';
                                          else fra_e(0) := '1'; end if; end if;
                   else if sign_c = '0' then if sign_d = '0' then fra_e(0) := fra_s(0);
                                          else fra_e(0) := '1'; end if;
                      else if sign_d = '0' then fra_e(0) := '1';
                         else fra_e(0) := fra_s(0); end if; end if; end if;
        else if op = "01" then if sign_s = '0' then if sign_c = '0' then if sign_d = '0' then fra_e(0) := '0';
                                                      else fra_e(0) := '0'; end if;
                                          else if sign_d = '0' then fra_e(0) := '1';
                                             else fra_e(0) := '0'; end if; end if;
                      else if sign_c = '0' then if sign_d = '0' then fra_e(0) := '1';
                                             else fra_e(0) := fra_s(0); end if;
                         else if sign_d = '0' then fra_e(0) := fra_s(0);
                            else fra_e(0) := '1'; end if; end if; end if;
           else fra_e(0) := fra_s(0); end if; end if;
        fra_e(1 to 13) := fra_s(1 to 13); fra_e(14) := '0'; exp_e := exp_c;
        if fra_e(1) = '1' then fra_f := sh_right( fra_e, 1 ); exp_f := exp_e + '1'; else fra_f := fra_e; exp_f := exp_e; end if;
        if fra_f = "0" then exp_n := "0"; else exp_n := exp_f; end if;
        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_rr(12) := fra_r(12) or fra_r(13) or fra_r(14);
        if ( fra_r(11) = '1' and fra_rr(12) = '0' and fra_r(10) = '1' ) or ( fra_r(11) = '1' and fra_rr(12) = '1' )
           then fra_rr(0 to 10) := fra_r(0 to 10) + '1'; else fra_rr(0 to 10) := fra_r(0 to 10); end if;
        if fra_rr(1) = '1' then fra_rm := sh_right( fra_rr, 1 ); exp_rm := exp_r + '1'; else fra_rm := fra_rr; exp_rm := exp_r; end if;
        MO(0) <= fra_rm(0); MO(1 to 8) <= fra_rm(3 to 10); EO <= exp_rm;
    end process U1;
end BEHAVIOR;
```

Appendix F

Blocksorter test

F.1 A configuration program for the blocksorter

```
use WORK.blocksort1_BENCH;
configuration blocksort1_CONF_struc of blocksort1_BENCH is
  for BEHAVIOR
     for U2 : blocksort1 use entity work.blocksort1(structural);
     end for;
  end for;
end blocksort1_CONF_struc;
```

F.2 A test program for the blocksorter

```
library XL;  use XL.XL_STD.all;
ENTITY blocksort1_BENCH IS END blocksort1_BENCH;
library XL;  use XL.XL_STD.all;  use XL.XL_Gates.all;
architecture BEHAVIOR of blocksort1_BENCH is
  component blocksort1 PORT ( RIN: IN  MVL; AIN: OUT  MVL; ROUT: OUT  MVL; AOUT: IN  MVL;
     A: IN MVL_VECTOR ( 4 DOWNTO 0 ); E: OUT MVL_VECTOR ( 4 DOWNTO 0 ); RESET: IN MVL );
  end component;
  signal RESET, SRin, SAin, SRout, SAout : MVL := '0'; signal SDin1E, SDoutE : MVL_VECTOR ( 4 DOWNTO 0 );
  subtype TEST_VECTORS1 is MVL_VECTOR( 4 DOWNTO 0 );
  type VECTOR_SET1 is array ( NATURAL range <> ) of TEST_VECTORS1;
  constant MY_VECTOR_SET1 : VECTOR_SET1 := ( "00110", "00101", "01001", "00010", "10001", "01101",
      "00111", "00011", "10110", "11010", "10010", "11110");
   begin
   U2: blocksort1 PORT map ( RIN => SRIN, AIN => SAIN, ROUT => SROUT, AOUT => SAOUT,
                        A => SDIN1E, E => SDOUTE, RESET => RESET  );
  U1: process
     variable Qain, S1 : MVL := '0';
     begin
       wait for 200 ns; wait for 30 ns;
       SDIN1E <= MY_VECTOR_SET1 (0); S1 := '1'; wait for 20 ns; SRIN <= not SAIN;
       for j in 1 to 11 loop wait for 20 ns;
          while ( S1 = '1' ) loop wait for 1 ns;
            if ( not SAIN = Qain ) then wait for 20 ns; S1 := '0'; Qain := SAIN; end if;
          end loop;
          wait for 30 ns; SDIN1E <= MY_VECTOR_SET1 (j); wait for 20 ns; S1 := '1';
          SRIN <= not SAIN after 20 ns; wait for 40 ns;
       end loop;
       while ( S1 = '1' ) loop wait for 1 ns;
         if ( not SAIN = Qain ) then wait for 20 ns; S1 := '0'; Qain := SAIN; end if;
       end loop;  wait;
  end process U1;
  U3: process
      variable Qrout, S2 : MVL := '0';
      begin
       for j in 0 to 11 loop  S2 := '1'; wait for 1 ns;
          while ( S2 = '1' ) loop wait for 1 ns;
            if ( not SROUT = Qrout ) then Qrout := SROUT; S2 := '0'; SAOUT <= SROUT after 30 ns; end if;
          end loop; wait for 40 ns;
       end loop;  wait;
  end process U3;
  reset_process: process
      begin
       RESET <= '0'; wait for 200 ns; reset <= '1'; wait;
  end process;
end BEHAVIOR;
```

F.3 A test program for the 6-stage blocksorter

```
library XL;  use XL.XL_STD.all;
ENTITY F6blocksort_BENCH IS   END F6blocksort_BENCH;
library XL;  use XL.XL_STD.all;  use XL.XL_Gates.all;
architecture BEHAVIOR of F6blocksort_BENCH is
  component blocksort1 PORT ( RIN: IN  MVL; AIN: OUT  MVL; ROUT: OUT  MVL; AOUT: IN  MVL;
     A: IN MVL_VECTOR ( 4 DOWNTO 0 ); E: OUT MVL_VECTOR ( 4 DOWNTO 0 ); RESET: IN MVL );
  end component;
  signal RESET, SRin, SAin, SRout, SAout : MVL := '0';
  signal SRout1, SAout1, SRout2, SAout2, SRout3, SAout3, SRout4, SAout4, SRout5, SAout5 : MVL := '0';
  signal SDin, SDout : MVL_VECTOR ( 4 DOWNTO 0 );
  signal SDout1, SDout2, SDout3, SDout4, SDout5 : MVL_VECTOR ( 4 DOWNTO 0 );
  subtype TEST_VECTORS1 is MVL_VECTOR( 4 DOWNTO 0 ) ;
  type VECTOR_SET1 is array ( NATURAL range <> ) of TEST_VECTORS1 ;
  constant MY_VECTOR_SET1 : VECTOR_SET1 := ( "00110", "00101", "01001", "00010", "10110", "00001",
  "11010", "11100", "01000", "11000", "10101", "00100", "11100", "10000", "10100", "11111", "10111", "01110" );
  begin
  U2: blocksort1 PORT map ( RIN => SRIN, AIN => SAIN, ROUT => SROUT1,
                    AOUT => SAOUT1, A => SDIN, E => SDOUT1, RESET => RESET );
  U21: blocksort1 PORT map ( RIN => SROUT1, AIN => SAOUT1, ROUT => SROUT2,
                    AOUT => SAOUT2, A => SDOUT1, E => SDOUT2, RESET => RESET );
  U22: blocksort1 PORT map ( RIN => SROUT2, AIN => SAOUT2, ROUT => SROUT3,
                    AOUT => SAOUT3, A => SDOUT2, E => SDOUT3, RESET => RESET );
  U23: blocksort1 PORT map ( RIN => SROUT3, AIN => SAOUT3, ROUT => SROUT4,
                    AOUT => SAOUT4, A => SDOUT3, E => SDOUT4, RESET => RESET );
  U24: blocksort1 PORT map ( RIN => SROUT4, AIN => SAOUT4, ROUT => SROUT5,
                    AOUT => SAOUT5, A => SDOUT4, E => SDOUT5, RESET => RESET );
  U25: blocksort1 PORT map ( RIN => SROUT5, AIN => SAOUT5, ROUT => SROUT,
                    AOUT => SAOUT, A => SDOUT5, E => SDOUT, RESET => RESET );
 U1: process variable Qain, S1 : MVL := '0';
    begin
     wait for 200 ns; wait for 20 ns; SDIN <= MY_VECTOR_SET1 (0); S1 := '1'; wait for 20 ns; SRIN <= not SAIN;
     for j in 1 to 17 loop wait for 20 ns;
        while ( S1 = '1' ) loop wait for 1 ns;
           if ( not SAIN = Qain ) then wait for 20 ns; S1 := '0'; Qain := SAIN; end if; end loop; wait for 30 ns;
        SDIN <= MY_VECTOR_SET1 (j); wait for 20 ns; S1 := '1'; SRIN <= not SAIN after 20 ns; wait for 40 ns;
     end loop;
     while ( S1 = '1' ) loop wait for 1 ns;
       if ( not SAIN = Qain ) then wait for 20 ns; S1 := '0'; Qain := SAIN; end if;
     end loop; wait; end process U1;
 U3: process variable Qrout, S2 : MVL := '0';
    begin
     for j in 0 to 17 loop S2 := '1'; wait for 1 ns;
        while ( S2 = '1' ) loop wait for 1 ns;
           if ( not SROUT = Qrout ) then Qrout := SROUT; S2 := '0'; SAOUT <= SROUT after 30 ns; end if;
        end loop; wait for 40 ns;
     end loop; wait; end process U3;
 reset_process: process begin RESET <= '0'; wait for 200 ns; reset <= '1'; wait; end process;
end BEHAVIOR;
```

F.4 A configuration program for the 6-stage blocksorter

```
use WORK.F6blocksort_BENCH;
configuration F6blocksort_CONF_struc of F6blocksort_BENCH is
  for BEHAVIOR
    for all : blocksort1 use entity work.blocksort1(structural);
    end for;
  end for;
end F6blocksort_CONF_struc;
```

Appendix G

A simple asynchronous microprocessor

G.1 STUMP model

A 16-bit mini-risc processor, STUMP [Ende98a] has 8 registers. R0 is always zero and can be used as a source operand allowing Move instructions to be synthesized from an Add instruction. R0 may be written to, but the result is always discarded allowing Compare instruction to be synthesized from Subtract instructions. R7 is the program counter. The 4-bit condition code register contains Sign Flag(N), Zero Flag(Z), Overflow Flag(V) and Carry-Out Flag(C) shown in Table G.1. The condition code register is updated to reflect the result of the ALU operation when arithmetic and logical instructions are executed. The condition code register is not changed for Load, Store, and Branch instructions. The instruction set and its 3 formats are shown as Tables G.2, G.3, G.4 and G.5. Table G.6 shows shift operations. Table G.7 shows conditional branch instructions.

Table G.1: The Condition code register

Bit 3	Bit 2	Bit 1	Bit 0
Sign Flag (N)	Zero Flag (Z)	Overflow Flag (V)	Carry-Out Flag (C)

Table G.2: Instruction set

Instruction Code	Instruction	Explanation
000	ADD	2's complement add
001	ADC	2's complement add with carry-in
010	SUB	2's complement subtract
011	SBC	2's complement subtract with borrow
100	AND	Bitwise AND of two 16-bit words
101	OR	Bitwise OR of two 16-bit words
110	LD/ST	Load Reg. from memory or Store Reg. to memory
111	Bcc	Branch if condition cc is satisfied

Table G.3: Instruction format: Type 1: 2 Source registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INSTR			0	LD CC	DST			SRC A			SRC B			SHIFT	

Table G.4: Instruction format: Type 2: 1 Source register, 1 immediate value

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INSTR			1	LD CC	DST			SRC A			Immediate				

Table G.5: Instruction format: Type 3: Conditional Branch

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Condition				Offset							

Table G.6: Shift operations

Operation	Instr Bit 1	Instr Bit 0	SRC A, bit 15:=	Carry-out:=
No shift	0	0	A15	0
ASR	0	1	A15	A0
ROR	1	0	A0	A0
RRC	1	1	C_{in}	A0

Table G.7: Conditional Branch Instructions

Mnemonic	Bits 11:8	Condition	
BAL	0000	Always	
BNV	0001	Never	
BHI	0010	$C + Z = 0$	comparison: unsigned arithmetic
BLS	0011	$C + Z = 1$	
BCC	0100	$C = 0$	overflow test: unsigned arithmetic
BCS	0101	$C = 1$	
BNE	0110	$Z = 0$	zero test
BEQ	0111	$Z = 1$	
BVC	1000	$V = 0$	overflow test: signed arithmetic
BVS	1001	$V = 1$	
BPL	1010	$N = 0$	comparison: signed arithmetic
BMI	1011	$N = 1$	
BGE	1100	$N'.V + N.V' = 0$	
BLT	1101	$N'.V + N.V' = 1$	
BGT	1110	$(N'.V + N.V') + Z = 0$	
BLE	1111	$(N'.V + N.V') + Z = 1$	

Table G.8: Instruction Phases

Phase 1	Fetch	The next instruction is fetch from memory
Phase 2	Decode/Execute	The instruction is decoded and executed
Phase 3	Write	ALU operations are written back into the Reg. bank Load/Store operations access memory

G.2 A behavioural description of the asynchronous STUMP

```
ENTITY Processor_asyn IS
  PORT ( RESET: IN MVL; ADDRESS: OUT MVL_vector ( 15 DOWNTO 0 ); DO: OUT MVL_vector ( 15 DOWNTO 0 );
    WR: OUT MVL; OE: OUT MVL; EN: OUT MVL; ROUT : OUT MVL; AOUT : IN MVL;
    DI: IN MVL_vector ( 15 DOWNTO 0 ); RIN : IN MVL; AIN : OUT MVL ); END Processor_asyn;
ARCHITECTURE behavioral OF Processor_asyn IS
BEGIN
U1: PROCESS  VARIABLE RA, RB, RD : MVL_vector(2 DOWNTO 0);
   VARIABLE RDD, s1_RDD, s2_RDD, s3_RDD, s4_RDD, s5_RDD, s6_RDD, s7_RDD, s8_RDD, s6_RA,
          s7_RA, s7_RB, s8_RA, s8_RB : MVL_vector(2 DOWNTO 0);
   VARIABLE reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7 : MVL_vector(15 DOWNTO 0);
   VARIABLE s1_reg0, s1_reg1, s1_reg2, s1_reg3, s1_reg4, s1_reg5, s1_reg6, s1_reg7 : MVL_vector(15 DOWNTO 0);
   VARIABLE s2_reg0, s2_reg1, s2_reg2, s2_reg3, s2_reg4, s2_reg5, s2_reg6, s2_reg7 : MVL_vector(15 DOWNTO 0);
   VARIABLE s3_reg0, s3_reg1, s3_reg2, s3_reg3, s3_reg4, s3_reg5, s3_reg6, s3_reg7 : MVL_vector(15 DOWNTO 0);
   VARIABLE s4_reg0, s4_reg1, s4_reg2, s4_reg3, s4_reg4, s4_reg5, s4_reg6, s4_reg7 : MVL_vector(15 DOWNTO 0);
   VARIABLE s5_reg0, s5_reg1, s5_reg2, s5_reg3, s5_reg4, s5_reg5, s5_reg6, s5_reg7 : MVL_vector(15 DOWNTO 0);
   VARIABLE s6_reg0, s6_reg1, s6_reg2, s6_reg3, s6_reg4, s6_reg5, s6_reg6, s6_reg7 : MVL_vector(15 DOWNTO 0);
   VARIABLE s6s_reg0, s6s_reg1, s6s_reg2, s6s_reg3, s6s_reg4, s6s_reg5, s6s_reg6, s6s_reg7,
          s7_reg0, s7_reg1, s7_reg2, s7_reg3, s7_reg4, s7_reg5, s7_reg6, s7_reg7 : MVL_vector(15 DOWNTO 0);
   VARIABLE s8_reg0, s8_reg1, s8_reg2, s8_reg3, s8_reg4, s8_reg5, s8_reg6, s8_reg7 : MVL_vector(15 DOWNTO 0);
   VARIABLE s9_reg0, s9_reg1, s9_reg2, s9_reg3, s9_reg4, s9_reg5, s9_reg6, s9_reg7 : MVL_vector(15 DOWNTO 0);
   VARIABLE current_instruction, s4_current_instruction, s5_current_instruction, s6_current_instruction,
          s7_current_instruction, s8_current_instruction, s9_current_instruction,
          mem_address, mdr, data_out, OPA, OPB, carry, result : MVL_vector ( 15 DOWNTO 0 );
   VARIABLE OPA1, OPB1, OPC1, result1 : MVL_vector ( 16 DOWNTO 0 );
   VARIABLE OPA2, OPB2, OPC2, result2 : MVL_vector ( 15 DOWNTO 0 );
   VARIABLE opcode, s7_opcode, s8_opcode : MVL_vector ( 2 DOWNTO 0 );
   VARIABLE branchcode : MVL_vector ( 3 DOWNTO 0 ); VARIABLE shiftcode : MVL_vector ( 1 DOWNTO 0 );
   VARIABLE condition_codes, s1_condition_codes, s2_condition_codes, s4_condition_codes, s5_condition_codes,
          s6_condition_codes, s7_condition_codes, s8_condition_codes, s9_condition_codes : MVL_vector ( 3 DOWNTO 0 );
   VARIABLE LDCC, BIT12, update, Cys, Cin, bccc, s6_bccc : MVL;
   VARIABLE fetch, s1_fetch, s2_fetch, s3_fetch, s4_fetch, s5_fetch, s6_fetch, s7_fetch, s8_fetch, s9_fetch : MVL;
   VARIABLE sro, Qrin, Qaout, Qen, Qwr, qoe, w, s1_w, s2_w, s3_w, s4_w, s5_w, s6_w, s7_w, s8_w, s9_w : MVL;
PROCEDURE req_in (Qrin: INOUT MVL) IS
        BEGIN while ( Rin = Qrin ) loop wait for 1 ns; end loop; Qrin := Rin; END req_in;
PROCEDURE req_out (sro: INOUT MVL) IS BEGIN sro := not sro; Rout <= sro; END req_out;
PROCEDURE ack_out (Qaout: INOUT MVL) IS
        BEGIN while ( Aout = Qaout ) loop wait for 1 ns; end loop; Qaout := Aout; END ack_out;
PROCEDURE ack_in IS BEGIN AIN <= RIN; END ack_in;
PROCEDURE reset_q (Qrin, Qaout, sro: INOUT MVL) IS BEGIN while ( RESET = '0' ) loop
   wait for 1 ns; Qrin := '0'; Qaout := '0'; sro := '0'; wait for 1 ns; ROUT <= sro; AIN <='0'; end loop; END reset_q;
BEGIN reset_q(Qrin, Qaout, sro); reg0 := "0000000000000000"; reg1 := "0000000000000000";
  reg2 := "0000000000000000"; reg3 := "0000000000000000"; reg4 := "0000000000000000";
  reg5 := "0000000000000000"; reg6 := "0000000000000000"; reg7 := "0000000000000000";
  condition_codes := "0000"; fetch := '1'; w := '0';
  while RESET = '1' loop
    -- memory_read
    s1_reg0 := reg0; s1_reg1 := reg1; s1_reg2 := reg2; s1_reg3 := reg3; s1_reg4 := reg4; s1_reg5 := reg5; s1_reg6 := reg6;
    s1_reg7 := reg7; s1_condition_codes := condition_codes; s1_w := w; s1_fetch := fetch; s1_RDD := RDD;
    IF fetch = '1' THEN mem_address := reg7; ELSE mem_address := result; END IF;
    IF w = '1' THEN qEN := '0'; qOE := '1'; qWR := '0'; ELSE  qEN := '0'; qOE := '0'; qWR := '1'; END IF;
    DO <= data_out; ADDRESS <= mem_address; EN <= qen; OE <= qoe; WR <= qwr; wait for 5 ns;
    req_out(sro); ack_out(Qaout); WAIT FOR 10 ns; -- Fetch instruction and increment PC
    IF not s1_w = '1' THEN req_in(Qrin); mdr := DI; s2_reg0 := s1_reg0; s2_reg1 := s1_reg1; s2_reg2 := s1_reg2;
      s2_reg3 := s1_reg3; s2_reg4 := s1_reg4; s2_reg5 := s1_reg5; s2_reg6 := s1_reg6; s2_reg7 := s1_reg7;
      s2_condition_codes := s1_condition_codes; s2_w := s1_w; s2_fetch := s1_fetch; s2_RDD := s1_RDD;
      ack_in; WAIT FOR 10 ns; -- Decode and set operands
      s3_reg1 := s2_reg1; s3_reg2 := s2_reg2; s3_reg3 := s2_reg3; s3_reg4 := s2_reg4; s3_reg5 := s2_reg5;
      s3_reg6 := s2_reg6; s3_condition_codes := s2_condition_codes; s3_w := s2_w; s3_fetch := s2_fetch;
```

```
IF s2_fetch = '1' THEN s3_reg7 := s2_reg7 + "0000000000000001"; s3_reg0 := "0000000000000000";
              ELSE s3_reg7 := s2_reg7; s3_reg0 := s2_reg0; END IF;
s3_RDD := s2_RDD; current_instruction := mdr; s4_reg0 := s3_reg0; s4_reg1 := s3_reg1; s4_reg2 := s3_reg2;
s4_reg3 := s3_reg3; s4_reg4 := s3_reg4; s4_reg5 := s3_reg5; s4_reg6 := s3_reg6; s4_reg7 := s3_reg7;
s4_condition_codes := s3_condition_codes; s4_w := s3_w; s4_fetch := s3_fetch;
branchcode := current_instruction(11 DOWNTO 8 ); s4_current_instruction := current_instruction;
s4_RDD := s3_RDD; s5_reg0 := s4_reg0; s5_reg1 := s4_reg1; s5_reg2 := s4_reg2; s5_reg3 := s4_reg3;
s5_reg4 := s4_reg4; s5_reg5 := s4_reg5; s5_reg6 := s4_reg6; s5_reg7 := s4_reg7;
s5_condition_codes := s4_condition_codes; s5_w := s4_w; s5_fetch := s4_fetch; s5_RDD := s4_RDD;
s5_current_instruction := s4_current_instruction;
shiftcode := s4_current_instruction(1 DOWNTO 0); RA := s4_current_instruction(7 DOWNTO 5);
IF branchcode = "0000" THEN bccc := '1'; -- JAL
ELSE IF branchcode = "0001" THEN bccc := '0'; -- JNV
    ELSE IF branchcode = "0010" THEN    -- JHI
          IF (s4_condition_codes(0) OR s4_condition_codes(2))='0' THEN bccc := '1';
          ELSE bccc := '0'; END IF;
        ELSE IF branchcode = "0011" THEN IF (s4_condition_codes(0) OR s4_condition_codes(2))='1' -- JLS
                          THEN bccc := '1'; ELSE bccc := '0'; END IF;
            ELSE IF branchcode = "0100" THEN    -- JCC
                  IF (s4_condition_codes(0)='0') THEN bccc := '1'; ELSE bccc := '0'; END IF;
                ELSE IF branchcode = "0101" THEN    -- JCS
                      IF (s4_condition_codes(0)='1') THEN bccc := '1'; ELSE bccc := '0'; END IF;
                    ELSE IF branchcode = "0110" THEN    -- JNE
                          IF (s4_condition_codes(2)='0') THEN bccc := '1'; ELSE bccc := '0'; END IF;
                        ELSE IF branchcode = "0111" THEN    -- JEQ
                              IF (s4_condition_codes(2)='1') THEN bccc := '1'; ELSE bccc := '0'; END IF;
                            ELSE IF branchcode = "1000" THEN    -- JVC
                                  IF (s4_condition_codes(1)='0') THEN bccc := '1'; ELSE bccc := '0'; END IF;
                                ELSE IF branchcode = "1001" THEN    -- JVS
                                      IF (s4_condition_codes(1)='1') THEN bccc := '1'; ELSE bccc := '0'; END IF;
                                    ELSE IF branchcode = "1010" THEN    -- JPL
                                          IF (s4_condition_codes(3)='0') THEN bccc := '1'; ELSE bccc := '0'; END IF;
                                        ELSE IF branchcode = "1011" THEN    -- JMI
                                              IF (s4_condition_codes(3)='1') THEN bccc := '1'; ELSE bccc := '0'; END IF;
                                            ELSE IF branchcode = "1100" THEN    -- JGE
                                                  IF ((NOT(s4_condition_codes(3)) AND s4_condition_codes(1)) OR
                                                     (s4_condition_codes(3) AND NOT(s4_condition_codes(1))))='0' THEN
                                                     bccc := '1'; ELSE bccc := '0'; END IF;
                                                ELSE IF branchcode = "1101" THEN    -- JLT
                                                      IF ((NOT(s4_condition_codes(3)) AND s4_condition_codes(1)) OR
                                                         (s4_condition_codes(3) AND NOT(s4_condition_codes(1))))='1' THEN
                                                         bccc := '1'; ELSE bccc := '0'; END IF;
                                                    ELSE IF branchcode = "1110" THEN    -- JGT
                                                          IF (((NOT(s4_condition_codes(3)) AND s4_condition_codes(1)) OR
                                                             (s4_condition_codes(3) AND NOT(s4_condition_codes(1)))) OR
                                                              s4_condition_codes(2))='0' THEN bccc := '1'; ELSE bccc := '0';
                                                          END IF;
                                                        ELSE IF branchcode = "1111" THEN    -- JLE
                                                              IF (((NOT(s4_condition_codes(3)) AND s4_condition_codes(1)) OR
                                                                 (s4_condition_codes(3) AND NOT(s4_condition_codes(1)))) OR
                                                                  s4_condition_codes(2))='1' THEN bccc := '1';
                                                              ELSE bccc := '0'; END IF; END IF; END IF; END IF; END IF;
END IF; END IF; END IF; END IF; END IF; END IF; END IF; END IF; END IF; END IF; END IF; END IF;
-- shift
s6_reg0 := s5_reg0; s6_reg1 := s5_reg1; s6_reg2 := s5_reg2; s6_reg3 := s5_reg3; s6_reg4 := s5_reg4;
s6_reg5 := s5_reg5; s6_reg6 := s5_reg6; s6_reg7 := s5_reg7; s6_condition_codes := s5_condition_codes;
s6_w := s5_w; s6_fetch := s5_fetch; s6_current_instruction := s5_current_instruction; s6_bccc := bccc;
s6_RA := RA; opcode := s5_current_instruction(15 DOWNTO 13 ); BIT12 := s5_current_instruction(12);
RB := s5_current_instruction(4 DOWNTO 2); s6_RDD := s5_RDD;
IF shiftcode = "00" THEN Cys := '0'; s6s_reg0 := s5_reg0; s6s_reg1 := s5_reg1; s6s_reg2 := s5_reg2;
    s6s_reg3 := s5_reg3; s6s_reg4 := s5_reg4; s6s_reg5 := s5_reg5; s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
ELSE IF shiftcode = "01" THEN IF RA = "000" THEN Cys := s5_reg0(0);
```

```
s6s_reg0(14 DOWNTO 0) := s5_reg0(15 DOWNTO 1); s6s_reg1 := s5_reg1; s6s_reg2 := s5_reg2;
s6s_reg3 := s5_reg3; s6s_reg4 := s5_reg4; s6s_reg5 := s5_reg5; s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
      ELSE IF RA = "001" THEN Cys := s5_reg1(0); s6s_reg0 := s5_reg0;
          s6s_reg1(14 DOWNTO 0) := s5_reg1(15 DOWNTO 1); s6s_reg2 := s5_reg2; s6s_reg3 := s5_reg3;
          s6s_reg4 := s5_reg4; s6s_reg5 := s5_reg5; s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
        ELSE IF RA = "010" THEN Cys := s5_reg2(0); s6s_reg0 := s5_reg0; s6s_reg1 := s5_reg1;
            s6s_reg2(14 DOWNTO 0) := s5_reg2(15 DOWNTO 1); s6s_reg3 := s5_reg3;
            s6s_reg4 := s5_reg4; s6s_reg5 := s5_reg5; s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
          ELSE IF RA = "011" THEN Cys := s5_reg3(0); s6s_reg0 := s5_reg0; s6s_reg1 := s5_reg1;
              s6s_reg2 := s5_reg2; s6s_reg3(14 DOWNTO 0) := s5_reg3(15 DOWNTO 1);
              s6s_reg4 := s5_reg4; s6s_reg5 := s5_reg5; s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
            ELSE IF RA = "100" THEN Cys := s5_reg4(0); s6s_reg0 := s5_reg0; s6s_reg1 := s5_reg1;
                s6s_reg2 := s5_reg2; s6s_reg3 := s5_reg3;
                s6s_reg4(14 DOWNTO 0) := s5_reg4(15 DOWNTO 1);
                s6s_reg5 := s5_reg5; s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
              ELSE IF RA = "101" THEN Cys := s5_reg5(0); s6s_reg0 := s5_reg0; s6s_reg1 := s5_reg1;
                  s6s_reg2 := s5_reg2; s6s_reg3 := s5_reg3; s6s_reg4 := s5_reg4;
                  s6s_reg5(14 DOWNTO 0) := s5_reg5(15 DOWNTO 1);
                  s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
                ELSE IF RA = "110" THEN Cys := s5_reg6(0); s6s_reg0 := s5_reg0;
                    s6s_reg1 := s5_reg1; s6s_reg2 := s5_reg2; s6s_reg3 := s5_reg3;
                    s6s_reg4 := s5_reg4; s6s_reg5 := s5_reg5;
                    s6s_reg6(14 DOWNTO 0) := s5_reg6(15 DOWNTO 1); s6s_reg7 := s5_reg7;
                  ELSE Cys := s5_reg7(0); s6s_reg0 := s5_reg0; s6s_reg1 := s5_reg1;
                    s6s_reg2 := s5_reg2; s6s_reg3 := s5_reg3; s6s_reg4 := s5_reg4;
                    s6s_reg5 := s5_reg5; s6s_reg6 := s5_reg6;
                    s6s_reg7(14 DOWNTO 0) := s5_reg7(15 DOWNTO 1); END IF;
      END IF; END IF; END IF; END IF; END IF; END IF;
 ELSE IF shiftcode = "10" THEN IF RA = "000" THEN Cys := s5_reg0(0);
          s6s_reg0(14 DOWNTO 0) := s5_reg0(15 DOWNTO 1); s6s_reg0(15) := s5_reg0(0);
          s6s_reg1 := s5_reg1; s6s_reg2 := s5_reg2; s6s_reg3 := s5_reg3;
          s6s_reg4 := s5_reg4; s6s_reg5 := s5_reg5; s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
        ELSE IF RA = "001" THEN Cys := s5_reg1(0); s6s_reg0 := s5_reg0;
            s6s_reg1(14 DOWNTO 0) := s5_reg1(15 DOWNTO 1); s6s_reg1(15) := s5_reg1(0);
            s6s_reg2 := s5_reg2; s6s_reg3 := s5_reg3; s6s_reg4 := s5_reg4;
            s6s_reg5 := s5_reg5; s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
          ELSE IF RA = "010" THEN Cys := s5_reg2(0); s6s_reg0 := s5_reg0; s6s_reg1 := s5_reg1;
              s6s_reg2(14 DOWNTO 0) := s5_reg2(15 DOWNTO 1); s6s_reg2(15) := s5_reg2(0);
              s6s_reg3 := s5_reg3; s6s_reg4 := s5_reg4; s6s_reg5 := s5_reg5;
              s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
            ELSE IF RA = "011" THEN Cys := s5_reg3(0); s6s_reg0 := s5_reg0; s6s_reg1 := s5_reg1;
                s6s_reg2 := s5_reg2; s6s_reg3(14 DOWNTO 0) := s5_reg3(15 DOWNTO 1);
                s6s_reg3(15) := s5_reg3(0); s6s_reg4 := s5_reg4; s6s_reg5 := s5_reg5;
                s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
              ELSE IF RA = "100" THEN Cys := s5_reg4(0); s6s_reg0 := s5_reg0;
                  s6s_reg1 := s5_reg1; s6s_reg2 := s5_reg2; s6s_reg3 := s5_reg3;
                  s6s_reg4(14 DOWNTO 0) := s5_reg4(15 DOWNTO 1);
                  s6s_reg4(15) := s5_reg4(0); s6s_reg5 := s5_reg5;
                  s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
                ELSE IF RA = "101" THEN Cys := s5_reg5(0); s6s_reg0 := s5_reg0;
                    s6s_reg1 := s5_reg1; s6s_reg2 := s5_reg2; s6s_reg3 := s5_reg3;
                    s6s_reg4 := s5_reg4; s6s_reg5(14 DOWNTO 0) := s5_reg5(15 DOWNTO 1);
                    s6s_reg5(15) := s5_reg5(0); s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
                  ELSE IF RA = "110" THEN Cys := s5_reg6(0); s6s_reg0 := s5_reg0;
                      s6s_reg1 := s5_reg1; s6s_reg2 := s5_reg2; s6s_reg3 := s5_reg3;
                      s6s_reg4 := s5_reg4; s6s_reg5 := s5_reg5;
                      s6s_reg6(14 DOWNTO 0) := s5_reg6(15 DOWNTO 1);
                      s6s_reg6(15) := s5_reg6(0); s6s_reg7 := s5_reg7;
                    ELSE Cys := s5_reg7(0); s6s_reg0 := s5_reg0; s6s_reg1 := s5_reg1;
                      s6s_reg2 := s5_reg2; s6s_reg3 := s5_reg3; s6s_reg4 := s5_reg4;
                      s6s_reg5 := s5_reg5; s6s_reg6 := s5_reg6;
                      s6s_reg7(14 DOWNTO 0) := s5_reg7(15 DOWNTO 1);
```

```
                                        s6s_reg7(15) := s5_reg7(0); END IF;
                END IF; END IF; END IF; END IF; END IF; END IF;
            ELSE IF shiftcode =  "11" THEN IF RA = "000" THEN Cys := s5_reg0(0);
                                    s6s_reg0(14 DOWNTO 0) := s5_reg0(15 DOWNTO 1);
                                    s6s_reg0(15) := s5_condition_codes(0); s6s_reg1 := s5_reg1; s6s_reg2 := s5_reg2;
                                    s6s_reg3 := s5_reg3; s6s_reg4 := s5_reg4; s6s_reg5 := s5_reg5;
                                    s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
                                ELSE IF RA = "001" THEN Cys := s5_reg1(0); s6s_reg0 := s5_reg0;
                                        s6s_reg1(14 DOWNTO 0) := s5_reg1(15 DOWNTO 1);
                                        s6s_reg1(15) := s5_condition_codes(0); s6s_reg2 := s5_reg2; s6s_reg3 := s5_reg3;
                                        s6s_reg4 := s5_reg4; s6s_reg5 := s5_reg5; s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
                                    ELSE IF RA = "010" THEN Cys := s5_reg2(0); s6s_reg0 := s5_reg0; s6s_reg1 := s5_reg1;
                                            s6s_reg2(14 DOWNTO 0) := s5_reg2(15 DOWNTO 1);
                                            s6s_reg2(15) := s5_condition_codes(0); s6s_reg3 := s5_reg3;
                                            s6s_reg4 := s5_reg4; s6s_reg5 := s5_reg5;
                                            s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
                                        ELSE IF RA = "011" THEN Cys := s5_reg3(0); s6s_reg0 := s5_reg0;
                                                s6s_reg1 := s5_reg1; s6s_reg2 := s5_reg2;
                                                s6s_reg3(14 DOWNTO 0) := s5_reg3(15 DOWNTO 1);
                                                s6s_reg3(15) := s5_condition_codes(0); s6s_reg4 := s5_reg4;
                                                s6s_reg5 := s5_reg5; s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
                                            ELSE IF RA = "100" THEN Cys := s5_reg4(0); s6s_reg0 := s5_reg0;
                                                    s6s_reg1 := s5_reg1; s6s_reg2 := s5_reg2; s6s_reg3 := s5_reg3;
                                                    s6s_reg4(14 DOWNTO 0) := s5_reg4(15 DOWNTO 1);
                                                    s6s_reg4(15) := s5_condition_codes(0);
                                                    s6s_reg5 := s5_reg5; s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
                                                ELSE IF RA = "101" THEN Cys := s5_reg5(0); s6s_reg0 := s5_reg0;
                                                        s6s_reg1 := s5_reg1; s6s_reg2 := s5_reg2;
                                                        s6s_reg3 := s5_reg3; s6s_reg4 := s5_reg4;
                                                        s6s_reg5(14 DOWNTO 0) := s5_reg5(15 DOWNTO 1);
                                                        s6s_reg5(15) := s5_condition_codes(0);
                                                        s6s_reg6 := s5_reg6; s6s_reg7 := s5_reg7;
                                                    ELSE IF RA = "110" THEN Cys := s5_reg6(0); s6s_reg0 := s5_reg0;
                                                            s6s_reg1 := s5_reg1; s6s_reg2 := s5_reg2;
                                                            s6s_reg3 := s5_reg3; s6s_reg4 := s5_reg4;
                                                            s6s_reg5 := s5_reg5;
                                                            s6s_reg6(14 DOWNTO 0) := s5_reg6(15 DOWNTO 1);
                                                            s6s_reg6(15) := s5_condition_codes(0);
                                                            s6s_reg7 := s5_reg7;
                                                        ELSE Cys := s5_reg7(0); s6s_reg0 := s5_reg0;
                                                            s6s_reg1 := s5_reg1; s6s_reg2 := s5_reg2;
                                                            s6s_reg3 := s5_reg3; s6s_reg4 := s5_reg4;
                                                            s6s_reg5 := s5_reg5; s6s_reg6 := s5_reg6;
                                                            s6s_reg7(14 DOWNTO 0) := s5_reg7(15 DOWNTO 1);
                                                            s6s_reg7(15) := s5_condition_codes(0); END IF;
END IF; END IF; END IF; END IF; END IF; END IF; END IF; END IF; END IF; END IF;
s7_reg0 := s6_reg0; s7_reg1 := s6_reg1; s7_reg2 := s6_reg2; s7_reg3 := s6_reg3; s7_reg4 := s6_reg4;
s7_reg5 := s6_reg5; s7_reg6 := s6_reg6; s7_reg7 := s6_reg7; s7_condition_codes := s6_condition_codes;
s7_w := s6_w; s7_fetch := s6_fetch; s7_current_instruction := s6_current_instruction; s7_RA := s6_RA;
s7_RB := RB; s7_opcode := opcode; s7_RDD := s6_RDD;
IF opcode = "111" THEN -- if type 3 instruction
    IF s6_bccc = '1' THEN IF s6_current_instruction(7) = '1' THEN
            OPB(15 DOWNTO 7) := "111111111"; OPB(6 DOWNTO 0) := current_instruction(6 DOWNTO 0);
        ELSE OPB(15 DOWNTO 7) := "000000000"; OPB(6 DOWNTO 0) := current_instruction(6 DOWNTO 0); END IF;
    ELSE OPB := s6_reg0; END IF; OPA := s6_reg7;
ELSE IF BIT12 = '0' THEN  --  type 1 instruction
        IF RA = "000" THEN OPA := s6s_reg0; ELSE IF RA = "001" THEN OPA := s6s_reg1;
        ELSE IF RA = "010" THEN OPA := s6s_reg2; ELSE IF RA = "011" THEN OPA := s6s_reg3;
            ELSE IF RA = "100" THEN OPA := s6s_reg4; ELSE IF RA = "101" THEN OPA := s6s_reg5;
                ELSE IF RA = "110" THEN OPA := s6s_reg6; ELSE OPA := s6s_reg7; END IF;
        END IF; END IF; END IF; END IF; END IF; END IF;
        IF RB = "000" THEN OPB := s6_reg0; ELSE IF RB = "001" THEN OPB := s6_reg1;
```

```
        ELSE IF RB = "010" THEN OPB := s6_reg2; ELSE IF RB = "011" THEN OPB := s6_reg3;
          ELSE IF RB = "100" THEN OPB := s6_reg4; ELSE IF RB = "101" THEN OPB := s6_reg5;
            ELSE IF RB = "110" THEN OPB := s6_reg6; ELSE OPB := s6_reg7; END IF;
        END IF; END IF; END IF; END IF; END IF; END IF;
      ELSE  --  type 2 instruction
        IF RA = "000" THEN OPA := s6_reg0; ELSE IF RA = "001" THEN OPA := s6_reg1;
        ELSE IF RA = "010" THEN OPA := s6_reg2; ELSE IF RA = "011" THEN OPA := s6_reg3;
          ELSE IF RA = "100" THEN OPA := s6_reg4; ELSE IF RA = "101" THEN OPA := s6_reg5;
            ELSE IF RA = "110" THEN OPA := s6_reg6; ELSE OPA := s6_reg7; END IF;
        END IF; END IF; END IF; END IF; END IF; END IF;
        IF s6_current_instruction(4) = '1' THEN
          OPB(15 DOWNTO 5) := "11111111111"; OPB(4 DOWNTO 0) := s6_current_instruction(4 DOWNTO 0);
        ELSE OPB(15 DOWNTO 5) := "00000000000"; OPB(4 DOWNTO 0) := s6_current_instruction(4 DOWNTO 0);
        END IF; END IF; END IF;
-- Execute instruction and sort out condition codes
s8_reg0 := s7_reg0; s8_reg1 := s7_reg1; s8_reg2 := s7_reg2; s8_reg3 := s7_reg3; s8_reg4 := s7_reg4;
s8_reg5 := s7_reg5; s8_reg6 := s7_reg6; s8_reg7 := s7_reg7; s8_condition_codes := s7_condition_codes;
s8_w := s7_w; s8_fetch := s7_fetch; s8_current_instruction := s7_current_instruction; s8_RDD := s7_RDD;
s8_opcode := s7_opcode; LDCC := s7_current_instruction(11); RD := s7_current_instruction(10 DOWNTO 8);
IF s7_opcode = "000" THEN  -- ADD
  OPA1(16) := '0'; OPB1(16) := '0'; OPA1( 15 DOWNTO 0 ) := OPA; OPB1( 15 DOWNTO 0 ) := OPB;
  result1 := OPA1 + OPB1; result := result1(15 DOWNTO 0); OPA2(15) := '0'; OPB2(15) := '0';
  OPA2( 14 DOWNTO 0 ) := OPA( 14 DOWNTO 0 ); OPB2( 14 DOWNTO 0 ) := OPB( 14 DOWNTO 0 );
  result2 := OPA2 + OPB2; carry(15) := result1(16); carry(14) := result2(15);
ELSE IF s7_opcode = "001" THEN -- ADC
    OPA1(16) := '0'; OPB1(16) := '0'; OPC1(16 DOWNTO 1) := "0000000000000000";
    OPC1(0) := s7_condition_codes(0); OPA1( 15 DOWNTO 0 ) := OPA;OPB1( 15 DOWNTO 0 ) := OPB;
    result1 := OPA1 + OPB1; result1 := result1 + OPC1; result := result1(15 DOWNTO 0);
    OPA2(15) := '0'; OPB2(15) := '0'; OPC2(15 DOWNTO 1) := "000000000000000";
    OPC2(0) := s7_condition_codes(0); OPA2( 14 DOWNTO 0 ) := OPA( 14 DOWNTO 0 );
    OPB2( 14 DOWNTO 0 ) := OPB( 14 DOWNTO 0 ); result2 := OPA2 + OPB2;
    result2 := result2 + OPC2; carry(15) := result1(16); carry(14) := result2(15);
  ELSE IF s7_opcode = "010" THEN -- SUB
      OPA1(16) := '0'; OPB1(16) := '0'; OPA1( 15 DOWNTO 0 ) := OPA; OPB1( 15 DOWNTO 0 ) := OPB;
      result1 := OPA1 - OPB1; result := result1(15 DOWNTO 0); OPA2(15) := '0'; OPB2(15) := '0';
      OPA2( 14 DOWNTO 0 ) := OPA( 14 DOWNTO 0 ); OPB2( 14 DOWNTO 0 ) := OPB( 14 DOWNTO 0 );
      result2 := OPA2 - OPB2; carry(15) := result1(16); carry(14) := result2(15);
    ELSE IF s7_opcode = "011" THEN -- SBC
        OPA1(16) := '0';  OPB1(16) := '0'; OPC1(16 DOWNTO 1) := "0000000000000000";
        OPC1(0) := s7_condition_codes(0); OPA1( 15 DOWNTO 0 ) := OPA; OPB1( 15 DOWNTO 0 ) := OPB;
        result1 := OPA1 - OPB1; result1 := result1 - OPC1; result := result1(15 DOWNTO 0);
        OPA2(15) := '0'; OPB2(15) := '0'; OPC2(15 DOWNTO 1) := "000000000000000";
        OPC2(0) := s7_condition_codes(0); OPA2( 14 DOWNTO 0 ) := OPA( 14 DOWNTO 0 );
        OPB2( 14 DOWNTO 0 ) := OPB( 14 DOWNTO 0 ); result2 := OPA2 - OPB2;
        result2 := result2 - OPC2; carry(15) := result1(16); carry(14) := result2(15);
      ELSE IF s7_opcode = "100" THEN -- AND
          result := OPA AND OPB; carry := "0000000000000000"; ELSE IF s7_opcode = "101" THEN -- OR
          result := OPA OR OPB;  carry := "0000000000000000"; ELSE IF s7_opcode = "110" THEN --ST/LD
          OPA1(16) := '0';  OPB1(16) := '0'; OPA1( 15 DOWNTO 0 ) := OPA; OPB1( 15 DOWNTO 0 ) := OPB;
          result1 := OPA1 + OPB1; result := result1(15 DOWNTO 0);
        ELSE --Bcc opcode = "111"
          OPA1(16) := '0'; OPB1(16) := '0'; OPA1( 15 DOWNTO 0 ) := OPA; OPB1( 15 DOWNTO 0 ) := OPB;
          result1 := OPA1 + OPB1; result := result1(15 DOWNTO 0);
END IF; END IF; END IF; END IF; END IF; END IF; END IF;
-- Write
IF s8_fetch = '0' THEN
  IF s8_RDD = "000" THEN s9_reg0 := mdr; s9_reg1 := s8_reg1; s9_reg2 := s8_reg2; s9_reg3 := s8_reg3;
    s9_reg4 := s8_reg4; s9_reg5 := s8_reg5; s9_reg6 := s8_reg6; s9_reg7 := s8_reg7;
  ELSE IF s8_RDD = "001" THEN s9_reg0 := s8_reg1; s9_reg1 := mdr; s9_reg2 := s8_reg2; s9_reg3 := s8_reg3;
      s9_reg4 := s8_reg4; s9_reg5 := s8_reg5; s9_reg6 := s8_reg6; s9_reg7 := s8_reg7;
    ELSE IF s8_RDD = "010" THEN s9_reg0 := s8_reg0; s9_reg1 := s8_reg1; s9_reg2 := mdr; s9_reg3 := s8_reg3;
        s9_reg4 := s8_reg4; s9_reg5 := s8_reg5; s9_reg6 := s8_reg6; s9_reg7 := s8_reg7;
```

```
            ELSE IF s8_RDD = "011" THEN s9_reg0 := s8_reg0; s9_reg1 := s8_reg1; s9_reg2 := s8_reg2;
                s9_reg3 := mdr; s9_reg4 := s8_reg4; s9_reg5 := s8_reg5; s9_reg6 := s8_reg6; s9_reg7 := s8_reg7;
              ELSE IF s8_RDD = "100" THEN s9_reg0 := s8_reg0; s9_reg1 := s8_reg1; s9_reg2 := s8_reg2;
                  s9_reg3 := s8_reg3; s9_reg4 := mdr; s9_reg5 := s8_reg5; s9_reg6 := s8_reg6; s9_reg7 := s8_reg7;
                ELSE IF s8_RDD = "101" THEN s9_reg0 := s8_reg0; s9_reg1 := s8_reg1; s9_reg2 := s8_reg2;
                    s9_reg3 := s8_reg3; s9_reg4 := s8_reg4; s9_reg5 := mdr; s9_reg6 := s8_reg6;
                    s9_reg7 := s8_reg7;
                  ELSE IF s8_RDD = "110" THEN s9_reg0 := s8_reg0; s9_reg1 := s8_reg1; s9_reg2 := s8_reg2;
                      s9_reg3 := s8_reg3; s9_reg4 := s8_reg4; s9_reg5 := s8_reg5; s9_reg6 := mdr;
                      s9_reg7 := s8_reg7;
                    ELSE s9_reg0 := s8_reg0; s9_reg1 := s8_reg1; s9_reg2 := s8_reg2; s9_reg3 := s8_reg3;
                      s9_reg4 := s8_reg4; s9_reg5 := s8_reg5; s9_reg6 := s8_reg6; s9_reg7 := mdr;
      END IF; END IF; END IF; END IF; END IF; END IF; END IF;
      s9_condition_codes := s8_condition_codes; s9_fetch := '1'; s9_w := s8_w; data_out := data_out;
    ELSE IF not ( s8_opcode(2) = '1' and s8_opcode(1) = '1' ) THEN -- ADD, ADC, SUB, SBC, AND, OR
        IF RD = "000" THEN s9_reg0 := result; s9_reg1 := s8_reg1; s9_reg2 := s8_reg2; s9_reg3 := s8_reg3;
          s9_reg4 := s8_reg4; s9_reg5 := s8_reg5; s9_reg6 := s8_reg6; s9_reg7 := s8_reg7;
        ELSE IF RD = "001" THEN s9_reg0 := s8_reg1; s9_reg1 := result; s9_reg2 := s8_reg2; s9_reg3 := s8_reg3;
            s9_reg4 := s8_reg4; s9_reg5 := s8_reg5; s9_reg6 := s8_reg6; s9_reg7 := s8_reg7;
          ELSE IF RD = "010" THEN s9_reg0 := s8_reg0; s9_reg1 := s8_reg1; s9_reg2 := result; s9_reg3 := s8_reg3;
              s9_reg4 := s8_reg4; s9_reg5 := s8_reg5; s9_reg6 := s8_reg6; s9_reg7 := s8_reg7;
            ELSE IF RD = "011" THEN s9_reg0 := s8_reg0; s9_reg1 := s8_reg1; s9_reg2 := s8_reg2; s9_reg3 := result;
                s9_reg4 := s8_reg4; s9_reg5 := s8_reg5; s9_reg6 := s8_reg6; s9_reg7 := s8_reg7;
              ELSE IF RD = "100" THEN s9_reg0 := s8_reg0; s9_reg1 := s8_reg1; s9_reg2 := s8_reg2;
                  s9_reg3 := s8_reg3; s9_reg4 := result; s9_reg5 := s8_reg5; s9_reg6 := s8_reg6; s9_reg7 := s8_reg7;
                ELSE IF RD = "101" THEN s9_reg0 := s8_reg0; s9_reg1 := s8_reg1; s9_reg2 := s8_reg2;
                    s9_reg3 := s8_reg3; s9_reg4 := s8_reg4; s9_reg5 := result; s9_reg6 := s8_reg6;
                    s9_reg7 := s8_reg7;
                  ELSE IF RD = "110" THEN s9_reg0 := s8_reg0; s9_reg1 := s8_reg1; s9_reg2 := s8_reg2;
                      s9_reg3 := s8_reg3; s9_reg4 := s8_reg4; s9_reg5 := s8_reg5; s9_reg6 := result;
                      s9_reg7 := s8_reg7;
                    ELSE s9_reg0 := s8_reg0; s9_reg1 := s8_reg1; s9_reg2 := s8_reg2; s9_reg3 := s8_reg3;
                      s9_reg4 := s8_reg4; s9_reg5 := s8_reg5; s9_reg6 := s8_reg6; s9_reg7 := result;
        END IF; END IF; END IF; END IF; END IF; END IF; END IF;
        IF LDCC = '1' THEN s9_condition_codes(3) := result(15);
          IF result = "0000000000000000" THEN s9_condition_codes(2) := '1'; ELSE s9_condition_codes(2) := '0';
          END IF; s9_condition_codes(1) := carry(15) xor carry(14); s9_condition_codes(0) := carry(15);
        ELSE s9_condition_codes := s8_condition_codes; END IF;
      ELSE s9_condition_codes := s8_condition_codes;
        IF ( s8_opcode(2) = '1' and s8_opcode(1) = '1' and s8_opcode(0) = '1' ) THEN  -- Bcc
          s9_reg0 := s8_reg0; s9_reg1 := s8_reg1; s9_reg2 := s8_reg2; s9_reg3 := s8_reg3; s9_reg4 := s8_reg4;
          s9_reg5 := s8_reg5; s9_reg6 := s8_reg6; s9_reg7 := result;
        ELSE s9_reg0 := s8_reg0; s9_reg1 := s8_reg1; s9_reg2 := s8_reg2; s9_reg3 := s8_reg3; s9_reg4 := s8_reg4;
          s9_reg5 := s8_reg5; s9_reg6 := s8_reg6; s9_reg7 := s8_reg7; END IF; END IF;
      IF ( s8_opcode(2) = '1' and s8_opcode(1) = '1' and s8_opcode(0) = '0' ) THEN
        IF LDCC = '1' THEN -- ST to memory
          IF RD = "000" THEN data_out := s8_reg0; ELSE IF RD = "001" THEN data_out := s8_reg1;
          ELSE IF RD = "010" THEN data_out := s8_reg2; ELSE IF RD = "011" THEN data_out := s8_reg3;
            ELSE IF RD = "100" THEN data_out := s8_reg4; ELSE IF RD = "101" THEN data_out := s8_reg5;
              ELSE IF RD = "110" THEN data_out := s8_reg6; ELSE data_out := s8_reg7;
          END IF; END IF; END IF; END IF; END IF; END IF; END IF; s9_w := '1'; s9_fetch := '0';
        ELSE -- LD from memory
          s9_w := '0'; data_out := data_out; s9_fetch := '0'; END IF;
      ELSE data_out := data_out; s9_fetch := s8_fetch; s9_w := s8_w; END IF; END IF;
    s9_current_instruction := s8_current_instruction; RDD := RD; wait for 1 ns;
  ELSE  -- IF not s1_w = '1'
    s9_w := '0'; s9_fetch := '1'; data_out := data_out; s9_condition_codes := s1_condition_codes; s9_reg0 := s1_reg0;
    s9_reg1 := s1_reg1; s9_reg2 := s1_reg2; s9_reg3 := s1_reg3; s9_reg4 := s1_reg4; s9_reg5 := s1_reg5;
    s9_reg6 := s1_reg6; s9_reg7 := s1_reg7; END IF; wait for 1 ns; fetch := s9_fetch; w := s9_w; data_out := data_out;
  condition_codes := s9_condition_codes; reg0 := s9_reg0; reg1 := s9_reg1; reg2 := s9_reg2; reg3 := s9_reg3;
  reg4 := s9_reg4; reg5 := s9_reg5; reg6 := s9_reg6; reg7 := s9_reg7;
END loop; wait for 1 ns; END process U1; END behavioral;
```

G.3 A test program for the asynchronous STUMP model

```
entity Processor_asyn_test is end;
LIBRARY ieee; USE ieee.std_logic_1164.all; USE ieee.std_logic_arith.all;
architecture behavioral of Processor_asyn_test is
  component Processor_asyn PORT ( RESET: IN std_logic; ADDRESS, DO: OUT std_logic_vector (15 DOWNTO 0);
    WR, OE, EN: OUT std_logic; ROUT: OUT std_logic; AOUT: IN std_logic;
    DI: IN std_logic_vector (15 DOWNTO 0); RIN : IN std_logic; AIN : OUT std_logic);
  end component;
  signal address, datain, dataout : std_logic_vector (15 DOWNTO 0) ;
  signal oe, wr, en, Reset, rout, aout, rin, ain : std_logic;
  subtype memory_address is integer range 0 to 63 ;
  type memory_array is array ( memory_address ) of std_logic_vector(15 downto 0) ;
begin
Reset_Process: process begin  Reset <= '0';  wait for 260 ns;  Reset <= '1'; wait; end process Reset_Process;
cpu: Processor_asyn port map ( RESET => reset, ADDRESS => address, DO => dataout,
      WR => wr, OE => oe, EN => en, ROUT => rout, AOUT => aout, DI => datain, RIN => rin, AIN => ain);
mem: process  variable memory : memory_array :=
( "0001101100001111",  -- 0000  1B0F            ADD R3, R0+0FH
  "0001110000001000",  -- 0001  1C08            ADD R4, R0+08H
  "0001110010001000",  -- 0002  1C88            ADD R4, R4+08H
  "0001110010001000",  -- 0003  1C88            ADD R4, R4+08H
  "0001110010001000",  -- 0004  1C88            ADD R4, R4+08H
  "1101000110000001",  -- 0005  D181            LD R1, (R4+01H)
  "1101001010000010",  -- 0006  D282            LD R2, (R4+02H)
  "0000101100101000",  -- 0007  0B28            LD R3, (R1+R2)
  "1101101110000011",  -- 0008  DB83             ST R3, (R4+03H)
  "1101010110000011",  -- 0009  D583            LD R5, (R4+03H)
  "0001110110101000",  -- 000A  1DA8             ADD R5, R5+08H
  "1111000011111111",  -- 000B  F0FF     LOOP: BAL LOOP
  "0000000000000000", "0000000000000000", "0000000000000000",  "0000000000000000",  -- 000FH
  "0000000000000000", "0000000000000000", "0000000000000000",  "0000000000000000",
  "0000000000000000", "0000000000000000", "0000000000000000",  "0000000000000000",
  "0000000000000000", "0000000000000000", "0000000000000000",  "0000000000000000",
  "0000000000000000", "0000000000000000", "0000000000000000",  "0000000000000000",  -- 001FH
  "0000000000000000", "0000000000000101", -- 0021  05
  "0000000000000100",                     -- 0022  04
  "0000000000110011",                     -- 0023  33
  "0000000000000000", "0000000000000000", "0000000000000000",  "0000000000000000",
  "0000000000000000", "0000000000000000", "0000000000000000",  "0000000000000000",
  "0000000000000000", "0000000000000000", "0000000000000000",  "0000000000000000",
  "0000000000000000", "0000000000000000", "0000000000000000",  "0000000000000000",
  "0000000000000000", "0000000000000000", "0000000000000000",  "0000000000000000",
  "0000000000000000", "0000000000000000", "0000000000000000",  "0000000000000000",
  "0000000000000000", "0000000000000000", "0000000000000000",  "0000000000000000" );
  variable addr: integer; variable qwr, qoe, qen: std_logic;
  variable qdata : std_logic_vector(15 downto 0); variable qrout, qain, sro, r: std_logic := '0';
   begin
     while RESET = '0' loop
       Qrout := '0'; Qain := '0'; sro := '0'; Rin <= sro; Aout <='0'; wait for 1 ns; END loop;
     IF ( not Rout = Qrout ) THEN Qrout := Rout; addr := to_integer( address );
        qdata := dataout; qwr := wr; qen := en; qoe := oe;
        IF qen = '0' and qwr = '0' THEN memory( addr ) := qdata; END IF;
        IF qen = '0' and qoe = '0' THEN r := '1'; END IF;
        wait for 20 ns; Aout <= Rout;  END IF;
     IF r = '1' THEN IF qain = sro THEN r := '0'; qdata := memory( addr );
            datain <= qdata; wait for 20 ns; sro := not sro; Rin <= sro; END IF; END IF;
     IF ( not Ain = Qain ) THEN Qain := Ain; END IF; wait for 1 ns;
   end process mem;
end behavioral;
```

Bibliography

[Akel90] Akella, V. and Gopalakrishnan, G., "Hierarchical Action Refinement: A Methodology for Compiling Asynchronous Circuits from A Concurrent HDL", 1990 Computer Hardware Description Languages Workshop.

[Akel92] Akella, V., "An Integrated Framework for the Automatic Synthesis of Efficient Self-timed Circuits from Behavioral Specifications", PhD Thesis, Department of Computer Science, University of Utah, 1992.

[Akel92a] Akella, V. and Gopalakrishnan, G., "SHILPA: A High-Level Synthesis System for Self-Timed Circuits", Proc. International Conf. Computer-Aided Design (ICCAD), IEEE Computer Society Press, November 1992, pp. 587-591.

[Appl97] Appleton, S.S., Morton, S.V., and Liebelt, M.J., "Two-Phase Asynchronous Pipeline Control", Proceedings of 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems, Eindhoven, The Netherlands, April 7-10, 1997, pp.12-21.

[AsynBib] Asynchronous Online Bibliography,

http://www.win.tue.nl/cs/pa/wsinap/async.html.

[AsynWeb] Asynchronous Logic Home Page,

http://www.cs.man.ac.uk/amulet/async/index.html.

[AsynTool] Asynchronous Tools available on the Internet,

http://www.cs.man.ac.uk/amulet/async/async_tools.html.

[Bard96] Bardsley, A., "An Asynchronous Logic Synthesiser", Final year project report, Department of Computer Science, University Of Manchester, UK, May 1996.

[Bard97] Bardsley, A. and Edwards, D., "Compiling the language Balsa to Delay Insensitive Hardware", International Conference on Hardware Description Languages and their Applications, edited by C. D. Kloos and E. Cerny, IFIP, 20-25 April 1997, Spain, pp. 89-91.

[Bard20] Bardsley, A., "Implementing Balsa Handshake Circuits", PhD Thesis, Dept. of Computer Science, University of Manchester, 2000.

[Bard98] Bardsley, A., “Balsa: An Asynchronous Circuit Synthesis System”, MPhil Thesis, Dept. of Computer Science, University of Manchester, 1998.

[Barr96] Barringer, H. and Fellows, D., Gough, G., Jinks, P., Marsden, B., and Williams, A., “Design and Simulation in Rainbow: A Framework for Asynchronous Micropipeline Circuits”, Proceedings of the European Simulation Symposium, Vol. 2, Society for Computer Simulation International, October 1996, pp. 567-571.

[Barr97] Barringer, H. and Fellows, D., Gough, G., Williams, A., “Abstract Modelling of Asynchronous Micropipeline Systems using Rainbow”, International Conference on Hardware Description Languages and their Applications, edited by C. D. Kloos and E. Cerny, IFIP, 20-25 April 1997, Spain, pp. 285-304.

[Berk88a] van Berkel, C.H. and Rem, M. and Saeijs, W. J. J. R., “VLSI Programming”, ICCD ’88: IEEE International Conference on Computer Design: VLSI in Computer & Processors, 1988, Philips Research Laboratories, pp. 152-156.

[Berk88b] van Berkel, C.H. and Saeijs, W. J. J. R., “Compilation of Communicating Processes into Delay-Insensitive Circuits”, ICCD ’88: IEEE International Conference on Computer Design: VLSI in Computer & Processors, 1988, Philips Research Laboratories, IEEE Computer Society Press, pp. 157-162.

[Berk93] van Berkel, K., “Handshake Circuits – An Asynchronous Architecture for VLSI programming, 1993, Cambridge University Press.

[Brun89] Brunvand, E. and Sproull, R. F., “Translating Concurrent Communicating Programs into Delay-Insensitive Circuits”, Carnegie Mellon, Technical Report No. CMU-CS-89-126, April, 1989.

[Brun91] Brunvand, E., “Translating Concurrent Communicating Programs into Asynchronous Circuits”, PhD Thesis, Carnegie Mellon University, CMU-CS-91-198, 1991.

[Brun91a] Brunvand, E. K. and Starkey, M., “An Integrated Environment for the Design and Simulation of Self Timed Systems”, edited by A. Halaas and P. B. Denyer, VLSI 91, IFIP, August 1991, pp. 4a.2.

[Burn87] Burns, M.S., “Automated Compilation of Concurrent Programs into self-timed Circuits”, Caltech, Caltech-CS-TR-88-2, 1987.

[Burn88] Burns, Steven M. and Martin, Alain J., “Syntax-directed Translation of Concurrent Programs into Self-timed Circuits”, Proceedings of the Fifth MIT Conference on Advanced Research in VLSI, edited by J. Allen and F. Leighton, MIT Press, 1988, pp. 35-50.

[Burn88a] Burns, S. M. and Martin, A. J., “Synthesis of Self-Timed Circuits by Program Transformation”, The Fusion of Hardware Design and Verification”, edited by Milne, G. J., Elsevier Science Publishers, 1988, pp. 99-116.

[Burn91] Burns, S. M., “Performance Analysis and Optimization of Asynchronous Circuits”, PhD Thesis, California Institute of Technology, CS-TR-91-01, 1991.

[Brzo95a] Brzozowski, J. A., Hauck, S., and Seger, C.-J., “Design of Asynchronous Circuits”, chapter 15, in “Asynchronous Circuits”, Springer-Verlag, 1995.

[Cade] Cadence, “Design Data Translators Reference”, September 1994.

[Calv86] Calvo, J. and Acha, J.L. and Valencia, M., “Asynchronous Modular Arbiter”, IEEE Transactions on Computers, Vol. c-35, No. 1, January 1986, pp. 67-70.

[Chan73] Chaney, T. J. and Molnar, C. E., “Anomalous Behavior of Synchronizer and Arbiter Circuits”, IEEE Transactions on Computers, Vol. C-22, No. 4, April 1973, pp. 421-422.

[Chu85] Chu, T. A., “A Design Methodology for Concurrent VLSI System”, Proceedings of the International Conference on Computer Design, IEEE (ICCD), 1985, pp. 407-410.

[Chu86] Chu, T.-A., “On the models for designing VLSI asynchronous digital systems”, Integration, the VLSI journal, Vol. 4, No. 2, June 1986, pp. 99-113.

[Chu86b] Chu, T.-A. and Glasser, L. A., “Synthesis of Self-timed Control Circuits form Graphs: An Example”, Proc. International Conf. Computer Design (ICCD), IEEE Computer Society Press, 1986, pp. 565-571.

[Chu87] Chu, T.-A., “Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications”, PhD Thesis, MIT/LCS/TR-393, MIT Laboratory for Computer Science, June 1987.

[Chun98] Chung, S.-H., “The Design of a Branch Target Cache for an Asynchronous Microprocessor”, Master Thesis, Computer Science Department, University of Manchester, UK, 1998.

[Clar67] Clark, W. A., “Macromodular Computer Systems”, AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference, Atlantic City, NJ, Vol. 30, Academic Press, 1967, pp. 335-336.

[Clar74] Clark, W. A. and Molnar, C. E., “Macromodular Computer Systems”, Chapter 3, Computers in Biomedical Research, Vol. IV, edited by Stacy, Ralph W. and Waxman, Bruce D., Academic Press, 1974, pp. 45-85.

[Coat93] Coates, B., Stevens, K.S., "Automatic Synthesis of Fast Compact Self-timed Control Circuits", Proceedings of the IFIP working conference on Asynchronous Design Methodologies, Manchester, England, 1993.

[Davi93c] Davis, A., "Practical Asynchronous Design", Proceedings of the VII Banff Workshop: Asynchronous Hardware Design, Banff, Canada, 1993.

[Davi95] Davis, A., Nowick, S.M., "Asynchronous Circuit Design: Motivation, Background, & Methods", in "Asynchronous Digital Circuit Design", edited by G. Birtwistle and A. Davis, Springer, 1995, pp. 1-49.

[Day91] Day, P., "A Micropipelined Multiplier Design", ACiD-WG/EXACT Workshop on Asynchronous Data Processing, Eindhoven University of Technology, Veldhoven, The Netherlands, December 1992.

[Day95] Day, P., and Woods, J.V., "Investigations into Micropipeline Latch Design Styles", IEEE Transactions on VLSI Systems, Vol. 3, No. 2, pp. 264-272, June 1995.

[Dijk75] Dijkstra, E. W., "Garded Commands, Non-Determinacy, and the Formal Derivation of Programs", Communications of the ACM, Vol. 18, 1975, pp. 453-457.

[Dill92] Dill, David L. and Nowick, Steven M. and Sproull, Robert F., "Specification and Automatic Verification of Self-Timed Queues", Formal Methods in System Design, Vol. 1, No. 1, July 1992, pp. 29-60.

[Eber89a] Ebergen, J. C., "Translating Programs into Delay-Insensitive Circuits", Technical Report: CWI Tract 56, Centre for Mathematics and Computer Science, Amsterdam, 1989.

[Ende93] Endecott, P.B., "Processor Architectures for Power Efficiency and Asynchronous Implementation", MSc Thesis, Dept. of Computer Science, University of Manchester, 1993.

[Ende95] Endecott, P.B., "Parallel Structures for Asynchronous Microprocessors", IEEE Computer Society, TCCA Newsletter, October 1995.

[Ende98] Endecott, P.B. and Furber, S.B., "Modelling and Simulation of Asynchronous Systems using the LARD Hardware Description Language", Proceedings of the 12th European Simulation Multiconference, Manchester, June 1998, Society for Computer Simulation International, pp. 39-43.

[Ende98a] Endecott, P.B. and Furber, S.B., "Behavioural Modelling of Asynchronous Systems for Power and Performance Analysis", Proceedings of PATMOS'98, Lyngby, Denmark, October 6-9 1998.

[Epic95] EPIC, "TimeMill User Manual", Release 3.2, EPIC, 1995.

[Epic96] EPIC, "PowerMill User Manual", Release 3.4, EPIC, 1996.

[Furb93] Furber, S.B., "AMULET1 - An Asynchronous ARM Processor", Symposium Record of Hot Chips V, Stanford University, CA, U.S.A., August, 1993.

[Furb93a] Furber, S.B., Day, P., Garside, J.D., Paver, N.C. and Woods, J.V., "A Micropipelined ARM", Proceedings of the IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration (VLSI'93), Grenoble, France, September 1993. Ed. Yanagawa, T. and Ivey, P.A. Pub. North Holland.

[Furb93c] Furber, S.B., "Computing without Clocks", Proceedings of the VII Banff Workshop: Asynchronous Hardware Design, Banff, Canada, 1993.

[Furb94] Furber, S.B., Day, P., Garside, J.D., Paver, N.C. and Woods, J.V., "AMULET1: A Micropipelined ARM", Proceedings of CompCon'94, IEEE Computer Society Press, CompCon'94, San Francisco, March 1994.

[Furb94a] Furber, S.B., Day, P., Garside, J.D., Paver, N.C., Temple, S. and Woods, J.V., "The Design and Evaluation of an Asynchronous Microprocessor", Proceedings of ICCD 94 (1994) pp 217-220. ICCD '94 Boston, Massachusetts Oct. 10-12th 1994.

[Furb95] Furber, S.B., "Computing without Clocks: Micropipelining the ARM Processor", in "Asynchronous Digital Circuit Design", edited by G. Birtwistle and A. Davis, Springer Verlag, pp.211-262.

[Furb95a] Furber, S.B., "An Introduction to Asynchronous Design: The Return of Asynchronous Logic", Department of Computer Science, University of Manchester, UK, 1995.

(http://www.cs.man.ac.uk/amulet/async/async_desc.html)

[Furb96] Furber, S.B., Day, P., Garside, J.D., Paver, N.C., and Temple, S., "AMULET2e", EMSYS'96 - OMI Sixth Annual Conference, Berlin, 23-25 September 1996. IOS Press, ISBN 90 5199 300 5.

[Furb96b] Furber, S.B., "Breaking Step - the Return of Asynchronous Logic", (23K compressed) Keynote talk - IEE Colloquium on the Design and Test of Asynchronous Systems Savoy Place, London, 28 February 1996, pp. 1/1 - 1/4 UK ISSN 0963-3308 ref 1996/040.

[Furb96c] Furber, S.B., and and Liu, J., "Dynamic Logic in Four-Phase Micropipelines", Proceedings: Async'96, Aizu-Wakamatsu, Japan, March 18-21 1996.

[Furb96d] Furber, S.B., and Day, P., "Four-Phase Micropipeline Latch Control Circuits", IEEE Transactions on VLSI Systems, vol. 4 no. 2, June 1996 pp. 247-253, ISSN 1063-8210.

[Furb96e] Furber, S. B., "ARM System Architecture", Addison Wesley Longman, 1996.

[Furb97] Furber, S.B., Garside, J.D., Temple, S., Liu, J., Day, P., and Paver, N.C., "AMULET2e: An Asynchronous Embedded Controller", Proceedings of Async '97, April 1997, pp. 290-299.

[Furb98] Furber, S.B., Garside, J.D., Gilbert, D.A., "AMULET3: A Hight-performance Self-Timed ARM Microprocessor", Proceedings of ICCD 98 (1998).

[Gajs92] Gajski, D., Dutt, N., Wu, A. and Lin, S., "High-Level Synthesis – Introduction to Chip and System Design", Kluwer Academic Publishers, 1992.

[Gars93] Garside, J. D., "A CMOS VLSI Implementation of an Asynchronous ALU", edited by S. Furber and D. Edwards, Asynchronous Design Methodologies, IFIP Transactions, Vol. A-28, Elsevier Science Publishers, 1993, pp. 181-207.

[Gars96a] Garside, J.D., "AMULET2e", Symposium Record of Hot Chips 8, Stanford University, CA, U.S.A., August, 1996.

[Gino90] Ginosar, R. and Michell, N., "On the Potential of Asynchronous Pipelined Processors", Computer Architecture News, Vol. 18, No. 4, December 1990, pp. 27-34.

[Gopa90] Gopalakrishnan, G. and Jain, P., "Some Recent Asynchronous System Design Methodologies", Dept. of C.S., Univ. of Utah, UU-CS-TR-90-016, October 1990.

[Gopa92] Gopalakrishnan, G. and Liebchen, A., "Dynamic Recordering of High Latency Transactions Using a Modified Micropipeline", Proceedings of ICCD-92, IEEE Computer Society Press, October 1992, pp. 336-340.

[Gopa93a] Gopalakrishnan, G. and Akella, V., "Specification, Simulation, and Synthesis of Self-Timed Circuits", Proc. Hawaii International Conf. System Sciences", Vol. I, IEEE Computer Society Press, pp 399-408, January 1993.

[Gopa93b] Gopalakrishnan, G., "Some Unusual Micropipeline Circuits", Dept. of Computer Science, Univ. of Utah, Technical Report No. UUCS-93-015, July 1993.

[Gree88] Greenstreet, M. R. and Steiglitz, K., "Throughput of Long Self-Timed Pipelines", Princeton, CS-TR-190-88, November 1988.

[Hack72] Hack, M., "Analysis of Production Schemata by Petri Nets", Project MAC, MIT ,1972, Technical Report TR 94.

[Hauc95] Hauck, S., "Asynchronous Design Methodologies: An Overview", Proceedings of the IEEE, Vol.83, No.1, pp.69-93, January 1995.

[Henn90] Hennessy, J. L. and Patterson, D. A.: Computer Architecture – A Quantitative Approach, Morgan Kaufmann Publishers Inc., 1990.

[Hoar78] Hoare, C.A.R., "Communicating Sequential Processes", Communications of the ACM", Vol. 21, No. 8, August 1978, pp. 666-677.

[Hoar85] Hoare, C. A. R., "Communicating Sequential Processes", Prentice-Hall, 1985.

[Huff54] Huffman, D. A., "The Synthesis of Sequential Switching Circuits", Journal of the Franklin Institute, Vol. 257, No. 3, March 1954, pp. 161-190.

[Huff57] Huffman, D. A., "The Design and Use of Hazard-free Switching Networks", Journal of the Association for Computing Machinery, Vol. 4 No.41, pp.47-62, January 1957.

[Ieee87] IEEE Std, IEEE Standard VHDL Language Reference Manual, Standard 1076-1987, IEEE Inc., 1988.

[Inmo83] Inmos, Occam Programming Manual, 1983.

[Jose90b] Josephs, M. B. and Udding, J. T., "Delay-Insensitive Circuits: An Algebraic Approach to their Design", Lecture Notes in Computer Science, edited by J. C. M. Baeten and J. W. Klop, Vol. 458, Springer-Verlag, August 1990, pp. 342-366.

[Jose91] Josephs, M. B. and Udding, J. T., "An Algebra for Delay-Insensitive Circuits", Workshop on Computer-Aided Verification, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, edited by Robert Kurshan and Edmund M. Clarke, AMS-ACM, Vol. 3, 1991, pp. 147-175.

[Kell74] Keller, R. M., "Towards a Theory of Universal Speed-Independent Modules", IEEE Transactions on Computers, Vol. C-23, No. 1, January 1974, pp. 21-33.

[Kell93] Kelly, R., "Asynchronous Design Aspects of High-Performance Logic Architectural Modelling of a Bipolar Asynchronous Microprocessor", MSc Thesis, Dept. of Computer Science, Manchester University, 1993.

[Kinn76] Kinniment, D.J. and Woods, J.V., "Synchronisation and arbitration circuits in digital systems", Proc. IEE", Vol. 123, No. 10, October 1976, pp. 961-966.

[Komo88] Komori, S. and Takata, H. and Tamura, T. and Asai, F. and Ohno, T. and Tomisawa, O. and Yamasaki, T. and Shima, K. and Asada, K. and Terada, H., "An Elastic Pipeline Mechanism by Self-Timed Circuits", IEEE journal of Solid-State Circuits, Vol. 23, No. 1, February 1988, pp. 111-117.

[Lin91] Lin, K.-J. and Lin, C.-S., "Automatic Synthesis of Asynchronous Circuits", Proc. 28th ACM/IEEE Design Automation Conference, IEEE Computer Society Press, 1991, pp. 296-301.

[Lin92] Lin, K.-J. and Lin, C.-S., "A Realization Algorithm of Asynchronous Circuits from STG", Proc. European Conference on Design Automation (EDAC), IEEE Computer Society Press, 1992, pp. 322-326.

[Liu97] Liu, J., "Arithmetic and Control Components for an Asynchronous System", PhD Thesis, Dept. of Computer Science, University of Manchester, 1997.

[Mano91] Mano, M. M., "Digital Design", Prentice-Hall, 1991.

[Mart86] Martin, A. J., "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits", Distributed Computing, Vol. 1, No. 4, 1986, pp. 226-234.

[Mart89] Martin, A. J., "Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits", Caltech Computer Science, 1989, Caltech-CS-TR-89-1.

[Mart89d] Martin, A. J., "Formal Program Transformations for VLSI Circuit Synthesis", in "Formal Development of Programs and Proofs", edited by Dijkstra, Edsger W., Addison-Wesley, 1989, pp. 59-80.

[Mart90d] Martin, A. J., "Synthesis of Asynchronous VLSI Circuits", Chapter 6, in "Formal Methods for VLSI Design", edited by J. Straunstrup, North-Holland, 1990, pp. 237-283.

[Mart91] Martin, A. J. and Hazewindus, P. J., "Testing Delay-Insensitive Circuits", Advanced Research in VLSI: Proceedings of the 1991 UC Santa Cruz Conference, edited by Séquin, Carlo H., MIT Press, 1991, pp. 118-132.

[Mart92] Martin, A. J., "Asynchronous Datapaths and the Design of an Asynchronous Adder", Formal Methods in System Design", Vol. 1, No. 1, July 1992, pp. 117-137.

[Mehr92] Mehra, R., "Micropipelined Cache Design Strategies for an Asynchronous Microprocessor", MSc Thesis, Dept. of Computer Science, University of Manchester, 1992.

[Mehr95] Mehra, R. and Garside, J.D., "A Cache Line Fill Circuit for a Micropipelined Asynchronous Microprocessor", IEEE Computer Society, TCCA Newsletter, October 1995.

[Meng89] Meng, T. H.-Y. and Brodersen, R. W. and Messerschmitt, D. G., "Automatic Synthesis of Asynchronous Circuits from High-Level Specifications", IEEE Trans. on Computer-Aided Design, Vol. 8, No. 11, November 1989, pp. 1185-1205.

[Mill65] Miller, R. E., "Switching Theory Vol. II: Sequential Circuits and Machines", John Wiley, 1965.

[Miro97] Mirotznik, M. S. and Prather, D., "How to choose EM software", IEEE Spectrum Magazine, December 1997.

[Moln83] Molnar, C. E. and Fang, T.-P., "Synthesis of Reliable Speed-Independent Circuit Modules: I. General Method for Specification of Module-Environment Interaction and Derivation of a Circuit Realization", Technical Memorandum, No. 297, Computer Systems Laboratory, Institute for Biomedical Computing, Washington Univ., St. Louis, MO, 1983.

[Moln85] Molnar, C. E. and Fang, T.-P. and Rosenberger, F. U., "Synthesis of Delay-Insensitive Modules", 1985 Chapel Hill Conference on Very Large Scale Integration, edited by Fuchs, Henry, Computer Science Press, 1985, pp. 67-86.

[Mull57] Muller, D. E. and Bartky, W. S., "A Theory of Asynchronous Circuits", Proceedings of an International Symposium on the Theory of Switching, Harvard University Press, April 1957, pp. 204-243.

[Nede95] Nedelchev, I. M., "Asynchronous VLSI Design", PhD Thesis, University of Surrey, UK, June 1995.

[Niel88] Niessen, C. and van Berkel, C.H. and and Rem, M. and Saeijs, W. J. J. R., "VLSI Programming and Silicon Compilation; A Novel Approach from Philips Research", ICCD '88: IEEE International Conference on Computer Design: VLSI in Computer & Processors, 1988, Philips Research Laboratories, pp. 150-151.

[Nowi91] Nowick, S. M. and Dill, D. L., "Automatic Synthesis of Locally-Clocked Asynchronous State Machines", 1991 Int. Conf. on Computer-Aided Design (Proceedings of ICCAD-91), IEEE Computer Society Press, Computer Systems Laboratory, Stanford University, November 1991, pp. 318-321.

[Nowi91a] Nowick, S. M. and Dill, D. L., "Synthesis of Asynchronous State Machines Using a Local Clock", 1991 Int. Conf. on Computer Design (Proceedings of ICCD-91), Computer Systems Laboratory, Stanford University, IEEE Computer Society Press, October 1991, pp. 192-197.

[Nowi92c] Nowick, S. M. and Dill, D. L., "Asynchronous State Machine Synthesis Using a Local Clock", 1992 MCNC Int. Workshop on Logic Synthesis, Computer Systems Laboratory, Stanford University, 1992.

[Orns67] Ornstein, S. M. and Stucki, M. J. and Clark, W. A., "A Functional Description of Macromodules", AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference, Atlantic City, NJ, Vol. 30, Academic Press, 1967, pp. 337-355.

[Pave92] Paver, N. C. and Day, P. and Furber, S. B. and Garside, J. D. and Woods, J. V., "Register Locking in an Asynchronous Microprocessor", Proc. International Conf. Computer Design (ICCD), IEEE Computer Society Press, October 1992, pp. 351-355.

[Pave94] Paver, N. C., "The Design and Implementation of an Asynchronous Microprocessor", PhD Thesis, Department of Computer Science, University of Manchester, June 1994.

[Pear75] Pearce, R.C. and Field, J.A. and Little, W.D., "Asynchronous Arbiter Module", IEEE Transactions on Computers, Vol. c-24, September 1975, pp. 931-932.

[Pete77] Peterson, J. L., "Petri Nets", Computing Surveys, Vol. 9, No. 3, September 1977, pp. 223-253.

[Pete81] Peterson, J. L., "Petri Net Theory and the Modeling of Systems", Prentice-Hall, 1981.

[Plum72] Plummer, W. W., "Asynchronous Arbiters", IEEE Transactions on Computers, Vol. c-21, January 1972, pp. 37-42.

[Rush95] Rushton, A., "VHDL For Logic Synthesis – An Introductory Guide For Achieving Design Requirements", McGraw-Haill, 1995.

[Seit70] Seitz, C. L., "Asynchronous Machines Exhibiting Concurrency", Conference Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, 1970.

[Scha95] Schalij, F. D., "Tangram", Tech. Rep. UR 008/93 (1995), Philips Electronics N.V..

[Seit80] Seitz, C. L., "System Timing", Chapter 7, in "Introduction to VLSI Systems", edited by Mead, Carver A. and Conway, Lynn A., Addison-Wesley, 1980.

[Suth89] Sutherland, I. E., "Micropipelines", The 1988 Turing Award Lecture, Communications of the ACM, Vol. 32, No. 6, pp. 720-738, January 1989.

[Tan92] Tan, S.-Y., "High Level Modelling of Micropipelines", Master Thesis, University of Manchester, UK, October 1992.

[Tan94] Tan, S.-Y., "A Simulation Model of Micropipelines", Journel of National Taipei Institute of Technology, Vol. 27-2, Taiwan, R.O.C, July 1994, pp. 115-147.

[Tan98] Tan, S.-Y. and Furber, S. and Yen, W.-F., "The Design of an Asynchronous VHDL Synthesizer", Proceedings of Design, Automation and Test in Europe Conference 1998 (DATE 98), Paris, February 1998, pp. 44-51.

[Theo94] Theodoropoulos, G. and Woods, J.V., "Building Parallel Distributed Models for Asynchronous Computer Architectures", Proceedings of the World Transputer Congress 1994, Como, Italy, September 1994, pp 285-301.

[Theo95a] Theodoropoulos, G., Strategies for the Modelling and Simulation of Asynchronous Computer Architectures", PhD Thesis, Dept. of Computer Science, University of Manchester, 1995.

[Unge69] Unger, S. H., "Asynchronous Sequential Switching Circuits", Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[Vanb90] Vanbekbergen, "Optimized Synthesis of Synchronous Control Circuits from Graph-Theoretic Specifications", Proceedings of the International Conference on Conputer-Aided Design, 1990, pp. 184-187.

[Vanb93] Vanbekbergen, P., "Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications", PhD Thesis, Catholic University of Leuven, September 1993.

[Will90] Williams, T. E. and Horowitz, M. A., "Bipolar Circuit Elements providing Self-Completion-Indication", IEEE journal of Solid-State Circuits, Vol. 25, No. 1, January 1990, pp. 309-312.

[Will92] Williams, T. E., "Analyzing and Improving the Latency and Throughput Performance of Self-Timed Pipelines and Rings", Proceedings of the IEEE International Conference on Circuits and Systems (ISCAS 92), May 1992.

[Wood97] Woods, J.V., Day, P., Furber, S.B., Garside, J.D., Paver, N.C., and Temple, S., "AMULET1: An Asynchronous ARM Microprocessor", IEEE Transactions on Computers, Vol. 46, No. 4, April 1997, pp.385-398.

[Yako92] Yakovlev, A. V., "On Limitations and Extensions of STG model for Designing Asynchronous Control Circuits", Proc. International Conf. Computer Design (ICCD), IEEE Computer Society Press, October 1992, pp. 396-400.

[Yun92] Yun, K.Y. and Dill, D.L. and Nowick, S.M., "Practical Generalizations of Asynchronous State Machines", Computer Systems Laboratory, Stanford University, CSL-TR-92-544, July, 1992.

[Yun92a] Yun, K. Y. and Dill, D. L. and Nowick, S. M., "Synthesis of 3D Asynchronous State Machines", Proc. International Conf. Computer Design (ICCD), IEEE Computer Society Press, October 1992, pp. 346-350.