Arithmetic and Control Components for an Asynchronous System

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

1997

Jianwei Liu

Department of Computer Science

Table of Contents

List of Figures

List of Tables

Abstract

This thesis describes arithmetic components (an adder and a multiplier) and control components which have been designed and implemented for AMULET3i, a commercial asynchronous embedded system chip incorporating the third generation asynchronous ARM processor (AMULET3).

A novel carry arbitration scheme is proposed (and has been patented) for parallel adder circuits. This new scheme provides an efficient encoding in which the carry is generated by arbitrating several input carry requests, exploiting the associativity of the carry computation. Post-layout simulation, in a 0.35 micron triple metal CMOS technology, shows that the adder for AMULET3i takes 1.8 ns to complete the computation of a 32-bit addition.

The multiplier design uses the modified Booth's algorithm integrated with a new encoding technique for adjusting the product result of an unsigned number multiplication. An adjustment value is made on the least significant 32-bit positions. Post-layout simulation, in a 0.35 micron triple metal CMOS technology, shows that the multiplier for AMULET3i takes 11.2 ns (2.8 ns $\times$ 4 cycles) to complete the computation of a 32-bit long multiplication in the worst case.

Organizing these arithmetic components efficiently into a four-phase asynchronous pipeline is investigated and a set of speed-independent latch control circuits is then proposed. Additionally, a set of control modules for four-phase micropipelines is presented. These two sets of control components can be used to construct complex and powerful asynchronous systems.

Declaration

No portion of the work referred to in the 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 and the Ownership of Intellectual Property Rights

(1) Copyright in text of this thesis rests with the Author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the Author and lodged in the John Rylands University Library of Manchester. Details may be obtained from the Librarian. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.

(2) The ownership of any intellectual property rights which may be described in this thesis is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

Further information on the conditions under which disclosures and exploitation may take place is available from the Head of Department of Computer Science.

The Author

The author was awarded the degrees of B.Sc and M.Sc, both in Electrical and Electronic Engineering, at Shenyang Institute of Technology and Harbin Institute of Technology, China, in 1984 and 1987, respectively. Significant experience was gained from involvement in 16-bit and 32-bit microprocessor, FPGA and ASIC chip designs at the Northeast Microelectronics Institute, China, from 1987 to 1993. He became interested in asynchronous design, VHDL design and formal approaches to hardware design when at the Technical University of Denmark, as a visiting scholar, from 1993 to 1994. He obtained an M.Sc degree in Computer Science from the University of Manchester in 1995. This thesis reports the results of the work undertaken during the AMULET project at the Computer Science Department of the University of Manchester.

Acknowledgements

First and foremost, I am grateful to my supervisor, Steve Furber, who has been an invaluable source of inspiration and continuous support in every aspect of this work. I could not hope for a better supervisor.

During my PhD study, I have also been with the Cogency Technology, Inc., as a VLSI design engineer. Thanks are due to Nigel Paver, Paul Day, Craig Farnsworth, Dave Jackson and Warren Lien for their support. Again, special thanks go to Nigel Paver for his commenting on the thesis structure and continuous support.

Thanks are due also to my adviser, Doug Edwards, for his encouragement and support, and to Steve Temple, Jim Garside, Dave Gilbert, Phil Endecott, Oleg Petlin and Julian Skidmore for their kind help. Thanks also to David Lloyd for his commenting on a draft of this thesis, to John Bainbridge and Suck-Heui Chung for discussions on the PETRIFY tool, and to Sun-Yen Tan for interesting conversations in Chinese. The other members of the AMULET group are acknowledged for their contributions at the regular Monday afternoon meetings.

I would also like to acknowledge with gratitude the grants from ORS and URS.

Last, but not least, I owe much gratitude to Jørgen Staunstrup, who brought me into the world of asynchronous design when I stayed at the Technical University of Denmark as a visiting scholar from 1993 to 1994.

Dedication

To

My parents — Pengqi Liu and Shuhua Zhang

My wife — Li He

And my sons — Dan Liu and Ying Liu

献给：

我的父母 -- 刘鹏起 和 张淑华

我的妻子 -- 何莉

我的儿子 -- 刘丹 和 刘英

Introduction 1

The real world is asynchronous by nature. It is, thus, logical to build digital systems in an asynchronous way, exploiting the potential advantages of this inherent property of asynchrony to their fullest. However, synchronous design styles have been preferred and have dominated digital systems for the last three decades. This is not surprising for two reasons. Firstly, synchronous design is easier to understand and easier to implement, which are attractive characteristics. Secondly, asynchronous design was usually considered less disciplined and more anarchic, which frightens most designers away.

With the rapid development of synchronous digital systems, however, there is evidence that we are beginning to hit some of the fundamental limitations of synchrony. It is becoming ever more difficult to establish global synchrony within today's chips, let alone from chip to chip. It is becoming unacceptable for global synchrony to burn increasing power, especially for power-sensitive applications where short battery life is the bane of the users. It is becoming a huge task for a digital system to be maintained and for its components to be replaced or reused. High noise emission and Electro-Magnetic Interference (EMI) are also increasingly becoming concerns in mobile communication applications.

Asynchronous design [1] has made a come-back in recent years, showing a number of advantages [2,3] over synchronous design. There are no clock related problems because global synchrony has been removed. Performance can be better as it is based on the average case rather than the worst case. Power consumption can be lower since power is only consumed when needed. Large digital systems can easily be maintained due to the high modularity and composability as each block can be designed without knowledge of the timing characteristics of any of the other blocks. Also, the low noise emission and good Electro-Magnetic Compatibility (EMC) of asynchronous systems are of potential use in mobile communication applications since increasingly rigorous EMI compliance specifications and testing can be more easily satisfied.

With asynchronous design becoming widely recognized after a world-wide resurgence of interest, it seems that it is expanding beyond its initial area of interest (which was primarily in academic research) into industry. However, there is still confusion surrounding the claimed advantages as there are very few demonstrable chips available to assess and therefore to endorse the asynchronous design methodology. The outcomes for most claims are still to be answered, though some are obvious.

The AMULET (Asynchronous Microprocessor Using Low Energy Techniques) group was established late in 1990, led by Professor Steve Furber, to investigate the claimed advantages and the feasibility of designing large asynchronous systems. The objective is to realize asynchronous microprocessors with lower power consumption and higher performance than is currently available using synchronous design techniques. Rather than adding to the theoretical work, an engineering approach was adopted and this has contributed to the growing pool of asynchronous knowledge during the last seven years.

The first milestone was AMULET1 [4-8] in 1994, an asynchronous implementation of the ARM 32-bit RISC microprocessor [9,10]. It demonstrated the feasibility of building an asynchronous system at the levels of complexity of current synchronous digital systems with the resources and tools readily available to synchronous designers.

The second milestone was AMULET2e [11] in 1996, an asynchronous embedded system chip which includes a significantly enhanced version of AMULET1. Its performance and power efficiency are competitive with the industry leading synchronous ARM designs. The AMULET2e work established a path to the commercial exploitation of asynchronous design.

AMULET3i, a commercial asynchronous embedded system chip for communication applications, is currently under development. This will be a significant milestone: the first fully asynchronous embedded system going into a commercially viable product.

The main objective of the work described in this thesis is to design high performance and low power arithmetic components (an adder and a multiplier) and control components for AMULET3i. An adder and a multiplier have been designed and implemented down to the layout level; these are two basic arithmetic blocks which are critical to the performance of the processor core. A set of control components for four-phase micropipelines, namely the pipeline latch control circuits, have been proposed, which can be used to organize arithmetic components efficiently into a micropipeline. Additionally, another set of control components, namely four-phase control modules, is also presented as basic building blocks. These two set of control components can be used to construct complex and powerful asynchronous systems.

Thesis overview

Due to the engineering nature of my PhD work, there is considerable detail which could easily blur the picture of the basic ideas. Instead, only the key ideas and relevant information are given here. Some engineering detail can be found in the circuit schematics and layout, presented in the appendices. Background information for asynchronous design is provided in chapter 2. The body of the work is divided into two main parts. The first part includes the arithmetic components, the AMULET3i adder in chapter 3 and the AMULET3i multiplier in chapter 4. The other part deals with the control components, a set of four-phase micropipeline latch control circuits in chapter 5 and a set of four-phase control modules in chapter 6. Each chapter is self-contained.

Addition is one of the most important arithmetic operations performed frequently within both general purpose and digital signal processing systems and an adder is therefore an important arithmetic component. A novel carry arbitration scheme is proposed (and has been patented [12]) for parallel adder circuits in chapter 3. This scheme provides an efficient encoding in which the carry is generated by arbitrating several input carry requests, exploiting the associativity of the carry computation. The new coding is a logically redundant superset of the conventional carry process. Departing from this general coding, certain modifications which reduce the redundancy can easily be made where this simplifies the implementation. The proposed carry arbitration scheme not only leads to high speed adders due to the reduction in the required layers of logic, but also offers a regular and compact layout and uniform fan-in and fan-out loadings. To demonstrate the feasibility and effectiveness of the new scheme, a 32-bit adder for AMULET3i has been designed and implemented down to the layout level.

Multiplication is another of the most common arithmetic operations. In chapter 4, the multiplier design for AMULET3i is presented, in which attention is focused on CMOS circuit design techniques. The AMULET3i multiplier can process two classes of multiply instructions: a normal 32-bit result and a long 64-bit result; both types of multiply instruction can also optionally perform an accumulate operation. A new encoding technique has been employed to adjust the final result of an unsigned number multiply operation. The design uses the modified Booth's algorithm [13,14] and eight bits are scanned at a time. A new 4-2 Counter with an enable control has been proposed. High speed circuit design techniques including the "true single-phase clocking registers" [15] are used. Some of this chapter is based on previous work by the author described in his M.Sc thesis [16].

As the four-phase micropipeline design style [17-19] was adopted for AMULET3i, the design of arithmetic components, the adder and the multiplier, are similar to clocked designs in some ways. However there are some subtle differences between the two; this is obvious in the multiplier design where the asynchronous nature has been exploited. The fundamental difference lies in the control mechanisms, which are described in chapter 5 and chapter 6.

The AMULET designs are based on Sutherland's micropipelines [20]. Although micropipelines were originally conceived with two-phase control, most recent work uses four-phase control mainly for performance reasons. The change from two-phase control to four-phase control leaves many choices open regarding the organization of the asynchronous pipelines. Chapter 5 explores these control schemes for asynchronous pipelines and presents a set of pipeline latch control circuits. All of the proposed pipeline latch control circuits are speed-independent, which is verified using the FORCAGE tool [21]. Low power considerations and the use of dynamic logic are also discussed in this chapter.

To ease the design of asynchronous systems based on four-phase micropipelines, a set of basic control modules is required. Such a set is proposed in chapter 6. Arbiters, which are non-trivial and tricky to implement, are also included. The specifications of these four-phase control modules are carried out using Petri Nets [22]. These basic control modules, together with the pipeline latch control circuits, can construct complex and powerful asynchronous systems including forking or joining multiple pipelines. All of the proposed control modules are speed-independent, which is verified using the PETRIFY tool [23-26].

A brief description of AMULET3i is given in chapter 7 in the hope of providing the big picture into which the components described in the previous chapters can be placed.

Conclusions are finally made in chapter 8.

Contributions

The main contributions made in this thesis are:

❑ In chapter 3, a high performance, low power asynchronous 32-bit adder for AMULET3i has been designed and implemented down to the layout level. The design uses a novel carry arbitration scheme (which has been patented) exploiting the associativity of the carry computation.

- In chapter 4, a high performance, low power asynchronous 32-bit multiplier for AMULET3i has been designed and implemented down to the layout level. The design employs the modified Booth's algorithm integrated with a new encoding technique for adjusting the product result of an unsigned number multiply operation.

- In chapter 5, a set of speed-independent latch control circuits has been proposed for asynchronous pipelines. These pipeline latch control circuits provide a framework within which arithmetic components can be efficiently organized.

- In chapter 6, a set of speed-independent control modules has been proposed. These control modules provide basic building blocks which can be used to construct complex and powerful asynchronous systems.

Background 2

In this chapter, we highlight some aspects of asynchronous design. Asynchronous design here refers to the design of digital circuits which operate correctly without relying on global clocks for synchronization. It is not possible to offer a comprehensive overview here; instead a brief introduction to the basic concepts is provided. The micropipeline design style and the AMULET project are then overviewed, which are of interest here because they form the background for the rest of the work described in this thesis. A full treatment of other asynchronous design styles can be founded elsewhere [2,3].

2.1 Introduction

A binary digital circuit uses two distinct values, 0 and 1. This is an ideal model. In reality, there are no true digital circuits, but only analog circuits which approximate to digital behaviours. No matter how *quick* the transitions the digital signals make, there are not only 0's and 1's but also undefined values between 0 and 1. These undefined values, when they occur, may not be recognized or may be interpreted in different ways by a digital circuit. As a result the digital circuit may behave unexpectedly. The period of this time uncertainty of a transition can be interpreted as "delay", and unexpected phenomena in a digital circuit due to the existence of delays are called "hazards'. To avoid such hazards, we must wait and evaluate digital signals only at well-defined reference points. Generally, digital design methodologies fall into two categories according to how these reference points are defined. The synchronous design methodology uses global clock signals as reference points, whereas the asynchronous design methodology employs the elapse of time or local control signals as reference points.

Historically, most early asynchronous designs used the elapse of time as reference points, based on some *real* delay assumptions on circuit elements or wires. The design process is much the same as synchronous design. It postulates many local clock signals based on the elapse of time between the changes of circuit signals. These postulated local clock signals are used to define reference points, which can be variable and controlled by adjusting delays in circuit elements or wires. Though much effort has been expended during the last three decades on this design approach, there are some fundamental problems that are hard to deal with. As a result, this design style is viewed as less disciplined and more anarchic than synchronous design, and this view has frightened most designers away in the past and still generates an adverse reaction.

However, most current asynchronous designs have abandoned the old ad hoc method based on *real* delay assumptions on circuit elements or wires. Instead, they use unbounded delay assumptions, which means a circuit always operates correctly under any distribution of circuit element delays or wire delays. Though this seems very pessimistic, it resolves all the delay-related problems that would otherwise arise. At the same time, the performance of a circuit is not compromised and even may be improved since concurrent operations can easily be exploited. Another benefit is that the circuit correctness issue is separated from delays and as a result circuit verification becomes easy, which is increasingly important for a complex system. Current asynchronous design is very systematic and well disciplined.

2.2 Basic concepts

A few key concepts and a taxonomy of asynchronous design are introduced and defined informally here; these are fundamental to the understanding of asynchronous design. Formal definitions are beyond the scope of this thesis and can be found elsewhere.

2.2.1 Delay models

The *bounded delay* model assumes that there is an upper bound on the delay of a circuit element or a wire.

The *unbounded delay* model assumes that there is no upper bound on the delay of a circuit element or a wire.

2.2.2 Circuit classification

Timed circuits are circuits whose correct operation is dependent on the delays in circuit elements and wires.

Speed-independent circuits are circuits whose correct operation is independent of the delays in circuit elements, and wire delays are assumed to be zero.

Delay-insensitive circuits are circuits whose correct operation is independent of the delays in both circuit elements and wires.

Quasi-delay-insensitive circuits are delay-insensitive circuits augmented with isochronic forks.

(*Isochronic forks* are sets of interconnecting wires where the delay difference between the branches is zero or negligible compared to the circuit element delays.)

2.2.3 Hazards and races

A *static hazard* is a single transition of a signal which should remain constant.

A *dynamic hazard* is a multiple transition of a signal which should change only once.

A *function hazard* is inherent in the specification of the logic function.

A *logic hazard* depends on the particular implementation of the logic function.

An *essential hazard* is inherent in the specification of the finite state machine.

A *non-essential hazard* (also called a *race*) depends on the particular state encoding.

A *noncritical race* is where all transient states settle to the same final state.

A *critical race* is where different transient states may lead to the different final states.

2.2.4 Metastability and arbitration

The *metastability* problem [27] is the phenomenon of the unusually long delay in the logic decision time between two values 0 and 1 for bistable systems such as flip-flops. When two asynchronous inputs to a bistable system arrive at very nearly the same time, a discrete decision must be made from a continuous range of input possibilities. It is fundamentally impossible to make this decision reliably within a bounded time. The delay may theoretically be an indefinite amount of time [28-30]. *Arbitration* is the mechanisms whereby a bistable system responds to either one input or the other.

Though metastability is an inevitable problem, the resulting metastable states can be resolved internally to maintain valid logic levels at the circuit interface using analog circuit techniques. The *mutual exclusion circuit* (MUTEX) [31] has this property and is used for making a non-deterministic decision between asynchronous calling requests.

It is worth noting here that the probability of failure of synchronous designs can never be zero and it must be accepted that whenever an asynchronous signal is input there is some chance of failure, though the probability can be made small with careful design techniques. However, this is not the case in asynchronous designs; an asynchronous circuit can be designed always to operate correctly, though it will require an unbounded time to resolve in the worst case.

2.2.5 Circuit specifications

Generally speaking, there are two broad classes of asynchronous design specification styles: state-based and event-based approaches.

Asynchronous finite state specifications are *Huffman state machines* [32,33] or extended Huffman state machines such as *Burst Mode* state machine [34,35]. Huffman circuits operate in *fundamental mode*, which assumes that only one input can change at a time, and succeeding input changes must not occur until the entire circuit settles into be a stable state. Relaxing the condition of only one input change in fundamental mode, burst-mode circuits allow multiple input changes as a burst. Another operation mode is called the *input/output mode* [36], which assumes that further external input changes can be applied as soon as the expected outputs have responded the current inputs. *Total state specifications* [37,38] are referred to as *Muller state graphs*, from which the semantics of event-based models are derived. *Trace theory* [39,40] is an abstract and formal description of a Muller state graph.

Event specifications are referred to as Petri Net [22] specifications, and include I-nets [41], Signal Transition Graphs (STG) [42,43], and Change Diagrams (CD) [21,44]. *Petri Net* specifications are a mathematical formalism to describe the behaviour of systems with concurrency, causality and conflicts between events.

I-nets are restricted Petri Nets in which interface signal names are assigned to transitions. *Signal Transition Graphs* are interpreted Petri Nets whose transitions are labelled as signal value changes. Similar to STGs, *Change diagrams* are interpreted Petri Nets, but allow OR-type signal transitions and disengageable arcs for nonrepeating signal transitions.

2.2.6 Signalling protocols

A *handshake* is a procedure where one signal (the request signal) makes a transition and a second signal (the acknowledge signal) makes a transition as a response.

Links are sets of request and acknowledgement wires used for communications through handshaking between different blocks.

The *two-phase* [20] protocol uses one handshake along a link for one transaction between two blocks. As a result, rising and falling signal transitions are equivalent,

The *four-phase* [17-19] protocol uses two handshakes along a link for one transaction between two blocks. There are variant schemes (see chapter 5) based on this protocol.

2.2.7 Data representation

Bundled data [20] comprises a set of data wires and an associated control signal that indicates the validity of the data. The data wires and the control wire are constructed such that stable data are available at the receiver before the control signal makes an indication of valid data. The relationship between the data and control delays required to ensure correct operation is referred to as the *bundling constraint*.

Coded data systems hide timing information in the data itself. There are many ways to encode data [45]. One well-known method is the *dual-rail* code [46] that requires two wires to encode a single bit of data. A transition can occur on either one wire or the other and not on both wires.

2.2.8 Synthesis

The type of specification usually determines the style of synthesis which can be used to generate the asynchronous circuit. State-based and event-based specifications have corresponding synthesis approaches: state-based and event-based synthesis. These two synthesis approaches are often used to design controllable asynchronous modules. Once a set of asynchronous modules is at hand, large asynchronous systems can be built up from these modules. Syntax directed program translations for specifications using CSP like programming languages [47] such as Tangram [48] are examples of this approach to building circuits from a library of modules. Although state-based or event-based design techniques can be applied directly to large asynchronous systems, they have not been very successful and practical for VLSI applications. Note that some designs are combinations of state-based and event-based design approaches.

2.3 Sutherland’s micropipelines

Micropipelines were introduced by Ivan Sutherland in his 1988 Turing Award lecture [20], and are a framework for building asynchronous pipelines. Micropipelines are composed of a bounded delay datapath operated by an unbounded delay two-phase control circuit.

Data passes on a bus from sender to receiver and is associated with a *Request* wire indicating when the data is valid. There is an *Acknowledge* wire from the receiver to the sender which indicates whether the data has been received. (see figure 2-1). The data wires and the request signalling wire must be treated as a bundle; the data must reach the receiver prior to the request event. Rising and falling transitions of request and acknowledge wires are equivalent, carrying the same information.

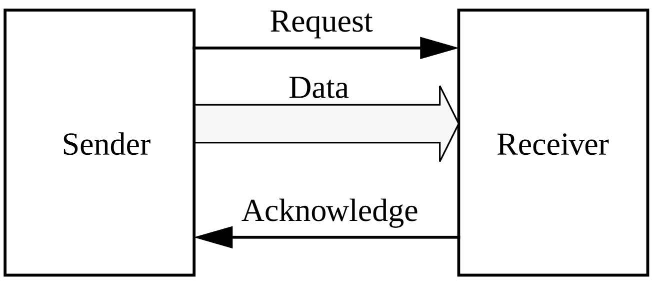


Figure 2-1: A bundled data interface

2.3.1 Event control modules

Figure 2-2 illustrates a basic set of event control blocks proposed by Sutherland which can be “programmed” to build complex and powerful asynchronous systems. These basic building blocks were designed using I-nets [41].

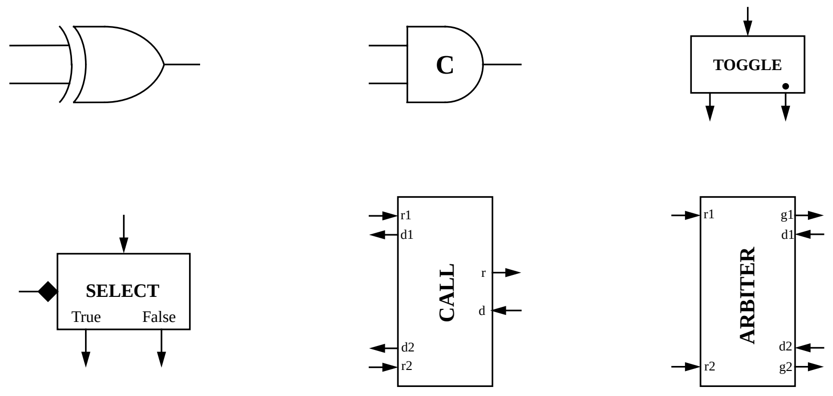


Figure 2-2: Micropipeline event logic modules

The **XOR** gate acts as the OR function for events. A transition on either input results in a transition on its output. For correct operation events must not arrive simultaneously on both inputs. XOR modules are often called **MERGE** elements because they merge two event streams into one.

The **Muller C-gate** acts as the AND function for events. A transition will occur at the output only when there have been transitions at both of the inputs. Muller C-gates are often called **RENDEZVOUS** elements because they make events at the output wait until events have been received on both inputs.

The **TOGGLE** module steers incoming events to its outputs alternately; the first event to arrive is issued to the output marked with a dot, the second to the unmarked output, and so on.

The **SELECT** module steers incoming events to one of two outputs according to the Boolean value of its diamond input. The Boolean value must be set up before the incoming event that it steers, a requirement similar to the bundling constraint.

The **CALL** module allows two processes to share a common resource, similar to a procedure call in software. The calling processes must be mutually exclusive; if they are not, they must access the call block through an arbiter.

The **ARBITER** module is used to control the interaction between two asynchronous event streams. As the two streams can issue requests at any time, the arbitration logic is inherently prone to metastability. The metastable states must be resolved internally to maintain valid logic levels at the interface of the module.

2.3.2 Event-controlled storage element

Event-controlled storage elements are needed to build a complete micropipeline circuit. Figure 2-3 shows an implementation of an event-controlled storage element and the symbol used to denote it.

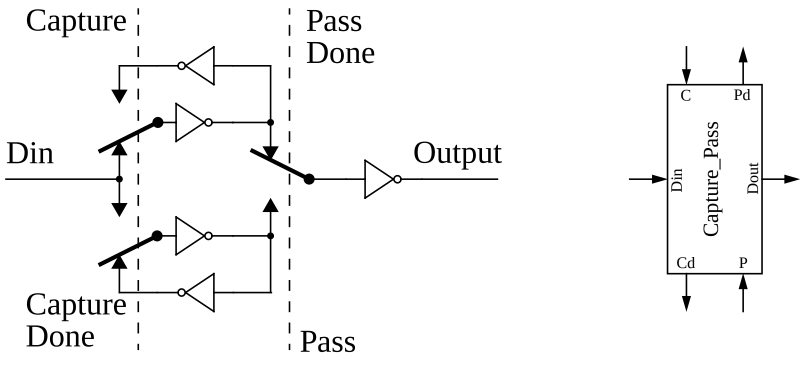


Figure 2-3: Event-controlled storage element

The input is initially connected to the output; it is transparent when empty and does not behave as a storage element at all. An event on the “capture” wire flips the two switches, and as a result a loop is formed containing two inverters, causing the data to be latched. This loop is still connected to the output, which therefore carries the previously latched value and does not follow subsequent input changes. An event on the “capture done” wire is issued after the switches have flipped. An event on the “pass” wire flips the other switch and as a result the element is returned to the transparent state and ready for the next coming transaction. Similarly, an event on the “pass done” wire is issued after the switch has flipped.

2.3.3 Micropipeline FIFO

A micropipeline with no processing in it, which is simply a FIFO, can be built as shown in figure 2-4. A data value can be entered into the FIFO from the left by signalling an event on the *Rin* wire, whereupon it will ripple down the FIFO and eventually will be fed out through the wire *Rout*.

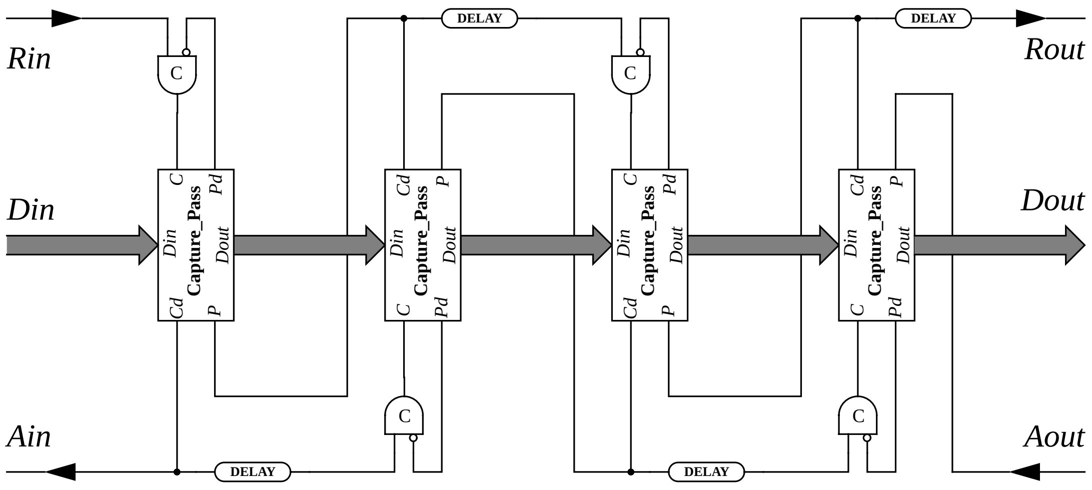


Figure 2-4: Micropipeline FIFO

One of the elegant features of a micropipeline FIFO is its elasticity. Data can be inserted into or removed from a FIFO at any rate bounded from zero to a maximum defined by the throughput parameter. The maximum insertion rate at the input end and the maximum removal rate at the output end can be achieved at the same time. However, in this condition, the percentage occupancy of the FIFO remains unchanged, and is determined by how fast the request signal passes forward and the acknowledge signal returns backward. If the request signal and acknowledge signal travel at the same rate, which is the most common case for a micropipeline FIFO, the percentage occupancy is only 50%.

Therefore, if we want to sustain high throughput for a long time, more FIFO stages should be used than might be expected. This is why an asynchronous micropipeline FIFO is often deeper than its synchronous counterpart for the same application.

2.3.4 Micropipelines with processing

The simple micropipeline FIFO can be extended to interpose processing logic between micropipeline FIFO stages, as shown in figure 2-5. The operation of this micropipeline with processing operates in a similar manner to the micropipeline FIFOs. The delay in the request event path must match the logic processing delay in order to preserve the data bundling convention.

More complex structures including forking and merging multiple pipelines can be built with the aid of other event control modules.

2.4 The AMULET project

It is our belief that asynchronous designs should be justified not only on a theoretical significance but also by their practical implications. This is also the motivation behind the AMULET project.

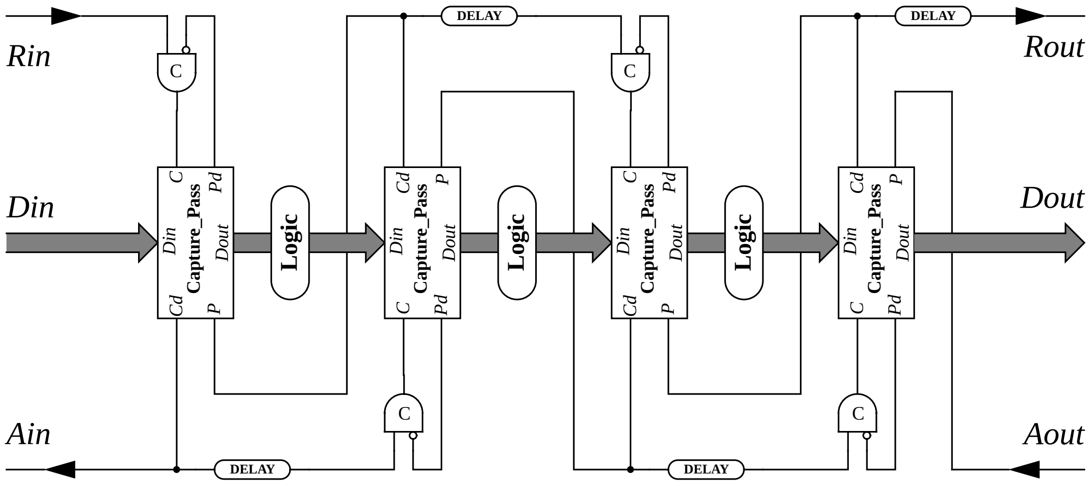


Figure 2-5: Basic micropipeline structure

2.4.1 AMULET1 chip

In 1994 Professor Steve Furber’s AMULET group at the University of Manchester took delivery of the AMULET1 processor, the first asynchronous implementation of a commercial processor architecture. The AMULET1 chips are code compatible with the ARM 32-bit RISC processor.

The design used the two-phase micropipeline style and includes several novel features such as the register locking mechanism [49], the instruction prefetching with its “colour” management of non-determinism and the data dependent ALU operations [50]. The chips were fabricated on two CMOS processes: a 1 µm process at ES2 and a 0.7 µm process at GEC Plessey Semiconductors.

Table 2-1 shows a summary of the characteristics of the AMULET1 chips with those of ARM6 for comparison. The chips demonstrate robustness to variations in temperature and voltage supply. The AMULET1 chip demonstrated the feasibility of building an asynchronous digital system at the levels of complexity of current synchronous digital systems.

Table 2-1: Characteristics of AMULET1 [4]

	AMULET1 (a)	AMULET1 (b)	ARM6
Process	1 μm	0.7 μm	1 μm
Area (mm^2)	5.5×4.1	3.9×2.9	4.1×2.7
Transistors	58,374	58,374	33,494
Performance	20.5 kDhry.	40 kDhry.	31 kDhry
Power	152 mW	N/A	148 mW
MIPS/W	77	N/A	120
Conditions	5 volt, 20 °C	5 volt, 20 °C	5 volt, 20 MHz

2.4.2 AMULET2e chip

Two years later, the AMULET group took delivery of the AMULET2e embedded system chip. AMULET2e is aimed at the embedded control market, and includes AMULET2 (a significantly enhanced version of AMULET1), 4 Kbytes of RAM which can also be configured to operate as a cache, a counter-timer for real-time reference, a flexible memory interface and various configuration and control registers. The design includes several novel features such as the load and register forwarding, branch target prediction, and the “halt” mode. The design uses the four-phase micropipeline design style. The chips were fabricated in a 0.5 μm triple metal CMOS technology.

Table 2-2 shows a summary of the characteristics of AMULET2e with those of ARM710 and ARM810 for comparison. AMULET2e is the first asynchronous processor whose performance and power-efficiency are competitive with the industry-leading clocked

ARM designs. One remarkable feature of AMULET2e is that the power consumption drops to nearly zero with the “halt” function enabled.

Table 2-2: Characteristics of AMULET2e [11]

	ARM710	AMULET2e	ARM810
Process	0.6 µm 2LM	0.5 µm 3LM	0.5 µm 3LM
Area (mm^2)	32	41	76
Transistors	570,295	454,000	836,022
Cache	8 K 4-way	4K 64-way	8K 64-way
MIPS	23	40	86
Power	120 mW	150 mW	500 mW
MIPS/W	192	250	172
Conditions	3.3 volt, 25 MHz	3.3 volt, 20 °C	3.3 volt, 72 MHz

2.4.3 AMULET3i

AMULET3i, an asynchronous embedded system chip which incorporates the third generation asynchronous ARM processor (AMULET3), is currently under development. Different from its predecessors, AMULET1 and AMULET2e, AMULET3i is aimed to be a commercially viable product for communication applications. This will be a significant step (see chapter 7).

Adder design 3

In this chapter a novel carry arbitration scheme is proposed (and has been patented) for parallel adder circuits. The proposed scheme provides an efficient encoding in which the carry is generated by arbitrating several input carry requests, exploiting the associativity of the carry computation. The new scheme not only leads to high speed adders due to a reduction in the required layers of logic, but also offers a regular and compact layout and uniform fan-in and fan-out loadings. To demonstrate the feasibility and effectiveness of the proposed scheme, a 32-bit adder for AMULET3i has been designed. Post-layout simulation, in a 0.35 micron triple metal CMOS technology, shows that it takes 1.8 ns to complete the computation of a 32-bit addition.

3.1 Introduction

Addition is one of the most important arithmetic operations performed frequently within both general purpose and digital signal processing systems. A problem with designing high speed adder circuits is that the most significant bits of the result are logically and physically dependent upon the carry output values from the least significant bits. The consequence of this sequential dependency is that addition operations tend to be relatively slow. This has been widely recognized, and adder design has been studied extensively for decades. Generally, the basis of adder designs is still either carry generation and carry propagation [51-55] or carry selection based on all possible results being available [56,57]. In recent years carry free additions achieved by employing redundant number systems have received considerable attention [58,59]. In an effort to develop adder circuits that are capable of operating at high speed a carry arbitration scheme for parallel adders is proposed. The new scheme provides an efficient encoding in which the carry is generated by arbitrating several input carry requests, exploiting the associativity of the carry computation.

3.2 Carry arbitration

The interesting and difficult task in an adder circuit is the computation of the carry bits. For an addition of two 1-bit numbers a_i and b_i, the carry c_i can be evaluated as shown in table 3-1. There are two general cases defined by the values of a_i and b_i. The first case, where there is a carry request, arises when both operand bits are equal. A 1-carry request occurs if both inputs are 1, whereas a 0-carry request occurs if both inputs are 0. The second case, where there is no carry request, arises when the operand bits have different values. The letter *u* indicates there is no carry request. Carry computation is similar to the logic behaviour when connecting wires a_i and b_i together. If they have the same value, then the result follows. If they are different, the result is undefined.

Table 3-1: Carry request

a_i, b_i	c_i
0 0	0
1 1	1
0 1	u
1 0	u

3.2.1 Two-way carry arbiter

One input pair may or may not make a carry request. If two input pairs (a_i, b_i) and (a_j, b_j) are considered together, they may issue carry requests at the same time. Therefore, there is a need to arbitrate these two carry requests. Figure 3-1 shows a two-way carry arbiter. The input pair (a_i, b_i) can make a *non-maskable* carry request, where *non-maskable* means that a carry request from the input pair (a_i, b_i) must always be granted service to the output c_i. The input pair (a_j, b_j) can make *maskable* carry requests, where *maskable* means that a carry request from the input pair (a_j, b_j) may be masked by the input pair (a_i, b_i). Only when there is no non-maskable carry request from the input pair (a_i, b_i) is a maskable carry request from the input pair (a_j, b_j) granted service to the output c_i. The truth table required to implement two-way carry arbiters is illustrated in table 3-2.

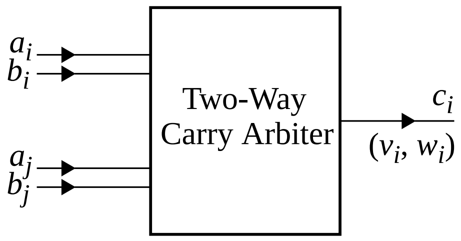


Figure 3-1: Two-way carry arbiter

Table 3-2: Two-way carry requests

a_i, b_i	a_j, b_j	c_i
0 0	- -	0
1 1	- -	1
0 1 (or 1 0)	0 0	0
0 1 (or 1 0)	1 1	1
0 1 (or 1 0)	0 1 (or 1 0)	u

The output carry c_i can be encoded using two wires (v_i, w_i) as shown in table 3-3. Equations EQ-1 and EQ-2 satisfy table 3-2 and table 3-3.

Table 3-3: Dual-rail code

c_i	v_i, w_i
0	0 0
1	1 1
u	0 1 (or 1 0)

$$v_i = a_ib_i + (a_i + b_i)a_j \qquad \text{(EQ-1)}$$

$$w_i = a_ib_i + (a_i + b_i)b_j \qquad \text{(EQ-2)}$$

Figure 3-2 shows a 4-bit carry computation using two-way carry arbiters. The solid dots represent two-way carry arbiters. The carry output values of the high order bits is generated by arbitrating carry requests from their low order bits. High order bit carry requests have priority over low order bit carry requests. For any carry output bits, there must exist a path to every low order input operand bits, which reflects the fact that the carries shall propagate across all the way of the word length of the operands.

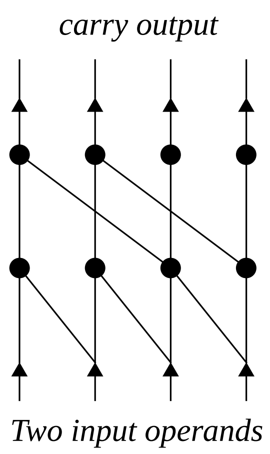


Figure 3-2: 4-bit carry computation

The proposed scheme is similar to but different from the scheme proposed by Brent and Kung [52]. Firstly, the computation logic needed for carry generate g_i and carry propagate p_i in the Brent and Kung adders is not necessary in our scheme. This leads to a reduction of the required layers of logic and hence high speed carry generation. Secondly, only single-rail signals need to be routed instead of dual-rail signals if the signals v_i and w_i are predicted to be equal (which indicates that the carry has been generated, either a 1-carry request or a 0-carry request). This results in a reduction of chip area, especially in the final row of the carry computation where more room is needed to accommodate signals crossing from the least significant bits to the most significant bits. Finally and more importantly, group adders in a carry select adder can be eliminated using the modified implementation of carry arbiters as we will see later.

In fact, the Brent and Kung scheme can be viewed as a special encoding of our scheme as shown in table 3-4. The two signal pairs (g_i, p_i) and (g_j, p_j) generated from the input pairs (a_i, b_i) and (a_j, b_j) can be seen as new input pairs. The new input pair (g_i, p_i) issues a 0-carry request when they are both 0, a 1-carry request when g_i is 1, and no carry request when p_i is 1. Note that g_i and p_i are mutually exclusive. In other words, the case of (g_i, p_i) with the value (1, 1) is removed by the Brent and Kung encoding.

Table 3-4: (g, p) carry requests

g_i, p_i (a_i, b_i)	g_j, p_j (a_j, b_j)	c_i
0 0 (0 0)	- - (- -)	0
1 0 (1 1)	- - (- -)	1
0 1 (0 1 or 1 0)	0 0 (0 0)	0
0 1 (0 1 or 1 0)	1 0 (1 1)	1
0 1 (0 1 or 1 0)	0 1 (0 1 or 1 0)	u

The carry request output c_i is encoded here as shown in table 3-5. Equations EQ-3 and EQ-4 give the behaviour defined by table 3-4 and table 3-5.

Table 3-5: The Brent and Kung carry code

c_i	v_i, w_i
0	0 0
1	1 1
u	0 1

$$v_i = g_i + p_i g_j \quad \text{(EQ-3)}$$

$$w_i = p_i p_j \quad \text{(EQ-4)}$$

Equations EQ-3 and EQ-4 are the key ideas of the well known Brent and Kung adders. It is clear that the computation logic for carry generate g_i and carry propagate p_i is wasteful except for understanding how the carries are generated and propagated. By encoding the input pair a_i and b_i to the carry generate g_i and propagate p_i, the advantage in our scheme of some signals being routed in single-rail form is lost because the dual-rail signals g_i and p_i are always required in the Brent and Kung scheme.

3.2.2 Three-way carry arbiter

A three-way carry arbiter is shown in figure 3-3. As before, the input pair (a_i, b_i) can issue a non-maskable carry request. The input pairs (a_j, b_j) and (a_k, b_k) can both make maskable carry requests at any time, possibly at the same time. However, the input pair (a_j, b_j) has priority over the input pair (a_k, b_k). Only when there is no non-maskable carry request from the input pair (a_i, b_i) is a maskable carry request from the input pair (a_j, b_j) granted service to the output c_i. Only when there is no non-maskable carry request from the input pair (a_i, b_i) and no maskable carry request from the input pair (a_j, b_j) is a maskable carry request from the input pair (a_k, b_k) granted service to the output c_i.

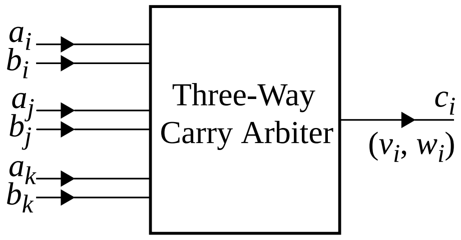


Figure 3-3: Three-way carry arbiter

The truth table required to implement three-way carry arbiters is shown in table 3-6. Equations EQ-5 and EQ-6 give the behaviour defined by table 3-3 and table 3-6.

Table 3-6: Three-way carry requests

a_i, b_i	a_j, b_j	a_k, b_k	c_i
0 0	- -	- -	0
1 1	- -	- -	1
0 1 (or 1 0)	0 0	- -	0
0 1 (or 1 0)	1 1	- -	1
0 1 (or 1 0)	0 1 (or 1 0)	0 0	0
0 1 (or 1 0)	0 1 (or 1 0)	1 1	1
0 1 (or 1 0)	0 1 (or 1 0)	0 1 (or 1 0)	u

$$v_i = a_ib_i + (a_i + b_i)(a_jb_j + (a_j + b_j)a_k) \qquad \text{(EQ-5)}$$

$$w_i = a_ib_i + (a_i + b_i)(a_jb_j + (a_j + b_j)b_k) \qquad \text{(EQ-6)}$$

Figure 4 shows a 9-bit carry computation using three-way carry arbiters. The addition of an n-bit binary number using three-way carry arbiters can be performed in a time proportional to $O(log_3n)$, and therefore is more efficient than using two-way carry arbiters where the computation time is $O(log_2n)$. It is worth noting here that there is a difference in complexity between two-way and three-way carry arbiters, which should be taken into account when comparing them.

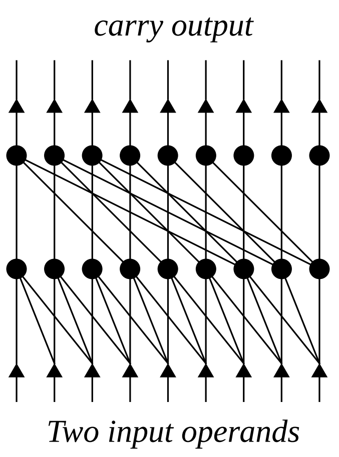


Figure 3-4: *9*-bit carry computation

The algorithm as shown in the above diagram is very elegant, and follows a very simple rule:

$t = 3$; while $(c_i = u)$ $\{c_i = c_{i-t}; t = 3t;\}$

Here t is the number of input pairs of carry arbiters used, and is three for this case. In the bottom line, the carries are computed just by looking at the three bits and hold either u or the correct carries. In the top line, the carry computation covers more bits and reach the point where all of the bit positions have been examined, therefore all of the carries are generated.

3.2.3 Carry arbiters with more than three ways

Using the same approach, carry arbiters with any number of pairs of input signals can be derived. Theoretically, it will be appreciated that a single carry arbitration circuit could be responsive to *n* pairs of input signals ($n > 3$). However, carry arbiters with more than four ways are not usually of practical interest. Firstly, too many series transistors are needed to implement these arbiters, which leads to inefficient CMOS designs. Secondly, the arbiter cell layout can easily become too large for the bit pitch of a datapath.

The circuit which implements a 9-bit carry computation as shown in figure 3-4 can be, in fact, considered as a nine-way carry arbiter, which is built up using three-way carry arbiters.

Now it may be questioned why the new term “carry arbitration” has been introduced to describe a circuit whose function is purely combinational. The introduction of this new term serves to explain the idea, since it is difficult to use the conventional terms “generate”, “kill” and “propagate” to describe the new coding.

In a sense, the new coding is a logically-redundant superset of the conventional carry process. Departing from this general coding, certain modifications (which reduce the redundancy) can easily be made where this simplifies the implementation as we will see later in section 3.4.

3.3 Parallel prefix computation

In this section the verification of the adder design using the proposed scheme is carried out formally by taking an n-bit addition using two-way carry arbiters as an example. Let $(a_n, a_{n-1}, \ldots, a_1)$ and $(b_n, b_{n-1}, \ldots, b_1)$ be n-bit binary input operands with output carries $(c_n, c_{n-1}, \ldots, c_1)$, and let c_0 be the initial input carry bit. We define an operator “*o*” [60] here as follows:

$$(a, b)o(a', b') = (ab + (a + b)b', ab + (a + b)b')$$

Lemma 1: Let

$$(v_i, w_i) = \begin{cases} (c_1, c_1) & \text{if } i = 1 \\ (a_i, b_i)o(v_{i-1}, w_{i-1}) & \text{if } 2 \le i \le n \end{cases}$$

where $c_1 = a_1b_1 + (a_1 + b_1)c_0$.

Then $c_i = v_i = w_i$ for $i = 1, 2, \ldots, n$.

Proof: We prove the lemma by induction on i.

It is obvious that the above equation holds true for $i = 1$.

If $i > 1$ and $c_{i-1} = v_{i-1} = w_{i-1}$, then

$(v_i, w_i) = (a_i, b_i)o(v_{i-1}, w_{i-1})$

$= (a_i, b_i)o(c_{i-1}, c_{i-1})$

$= (a_ib_i + (a_i + b_i)c_{i-1}, a_ib_i + (a_i + b_i)c_{i-1})$

$= (c_i, c_i)$

Thus, the equation holds true by induction.

Lemma 2: The operator "o" is associative.

Proof: For any three (a_3, b_3), (a_2, b_2) and (a_1, b_1),

$[(a_3, b_3)o(a_2, b_2)]o(a_1, b_1) =$

$[(a_3b_3 + (a_3 + b_3)a_2), (a_3b_3 + (a_3 + b_3)b_2)]o(a_1, b_1) =$

$(((a_3b_3 + (a_3 + b_3)a_2)(a_3b_3 + (a_3 + b_3)b_2) +$

$((a_3b_3 + (a_3 + b_3)a_2) + (a_3b_3 + (a_3 + b_3)b_2))a_1),$

$((a_3b_3 + (a_3 + b_3)a_2)(a_3b_3 + (a_3 + b_3)b_2) +$

$((a_3b_3 + (a_3 + b_3)a_2) + (a_3b_3 + (a_3 + b_3)b_2))b_1)) =$

$(((a_3b_3 + (a_3 + b_3)a_2b_2) + (a_3b_3 + (a_3 + b_3)(a_2 + b_2))a_1),$

$((a_3b_3 + (a_3 + b_3)a_2b_2) + (a_3b_3 + (a_3 + b_3)(a_2 + b_2))b_1)) =$

$((a_3b_3 + (a_3 + b_3)(a_2b_2 + (a_2 + b_2)a_1)),$

$(a_3b_3 + (a_3 + b_3)(a_2b_2 + (a_2 + b_2)b_1))) =$

$(a_3, b_3)\mathrm{o}(a_2b_2 + (a_2 + b_2)a_1, a_2b_2 + (a_2 + b_2)b_1) =$

$(a_3, b_3)o[(a_2, b_2)o(a_1, b_1)]$

Thus, the operator "*o*" is associative.

This lemma provides the foundation for using tree structures to generate carries since the signals v_i and w_i can be computed in any order from the given input values. This is the key idea for the proposed scheme.

Lemma 3: The operator "*o*" is not commutative.

This can easily be proved by inspection that $(1, 1)o(0, 0) \neq (0, 0)o(1, 1)$. This lemma implies that carry arbitration should perform in a prioritized way.

3.4 Implementation

Figure 3-5 shows a static CMOS implementation of a two-way carry arbiter. Note that the outputs v_i and w_i are complemented signals. However, the arbiter is quite symmetrical and implementing the next stage in inverse logic is straightforward. The signals through two arbiters are naturally positive true, so no inverters are needed.

Figure 3-6 shows a pass-transistor based implementation of a two-way carry arbiter. This implementation has an additional feature. The output v_i is zero if and only if the output w_i is zero, and the output w_i is one if and only if the output v_i is one. This provides another view of the arbiter. When the outputs v_i and w_i are different this means that there

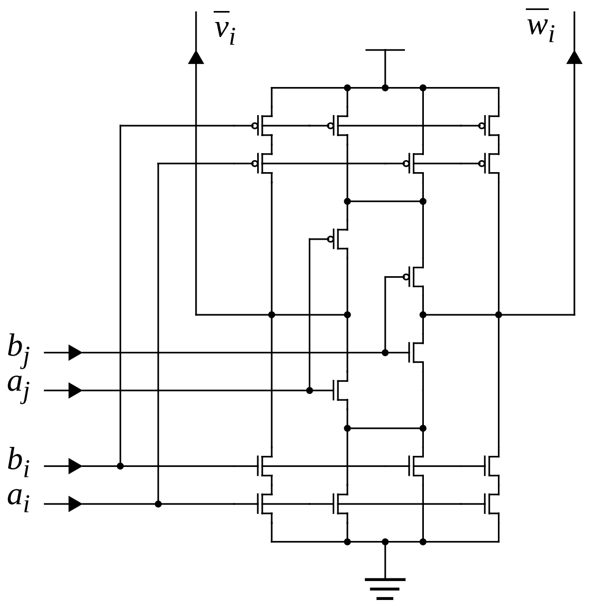


Figure 3-5: Static implementation of a two-way carry arbiter

are no carry requests from the inputs as described previously. Furthermore we can view the output v_i as the carry out generated with a one carry-in and the output w_i as the carry out generated with a zero carry-in. The implementation in figure 3-5 does not distinguish which is the carry out generated with a one carry-in and which with a zero carry-in, since each output can be zero or one independent of the other output. The AND and OR gates in figure 3-6 serve as an input conversion from (0 1) to (1 0). The signals after these two gates, e.g., (o_j, z_j), take one of the three values (0 0), (1 1) and (1 0).

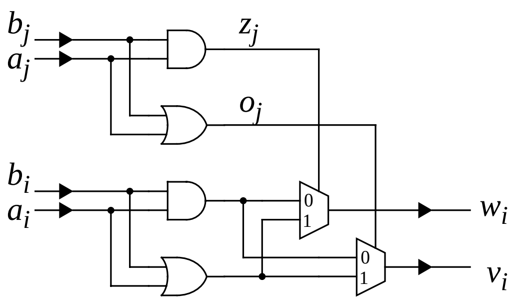


Figure 3-6: Pass-transistor based implementation

Three-way carry arbiters and four-way carry arbiters may be advantageous if dynamic CMOS techniques are used. Figure 3-7 shows a direct dynamic CMOS implementation [61-63] of a three-way carry arbiter. Instead of using a global precharge control signal, local incoming input signals are used for this purpose. The operation of the circuit is such that the nodes *n1* and *n2* are precharged high when the inputs a_i and b_i are low during the reset phase of the control handshake and will conditionally discharge during the evaluation phase in a self-timed design. The inverters are required for the next stage and also served to maintain proper drive strength.

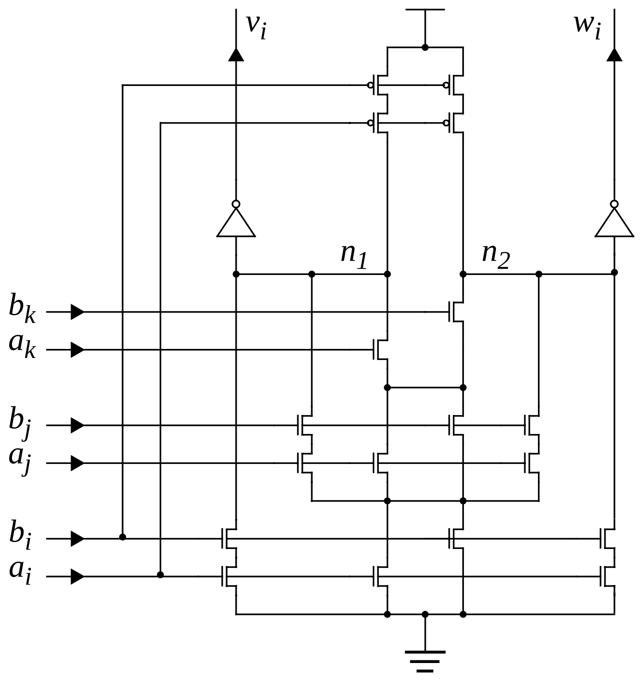


Figure 3-7: Direct implementation of a three-way carry arbiter

Figure 3-8 gives a modified version of the three-way carry arbiter by reducing the redundancy of the new coding. We assume here that every input pair takes one of the three values (0 0), (1 1) and (1 0), and (0 1) has already been transformed to (1 0) as described previously. The output v_i is the carry out generated with a one carry-in and the output w_i is the carry out generated with a zero carry-in if no carry requests issue from input signals. This results in the elimination of group adders in a carry select adder (see section 3.6) and is the main feature of our scheme.

However, the use of the modified implementation needs the input conversion from (0 1) to (1 0). Fortunately this causes no problem; the conversion is simple. It consists of one 2-input NAND and one 2-input NOR gate per bit. For practical reasons, gates are normally necessary anyway to isolate the signals from the main input buses. The difference here is that NAND and NOR gates are used instead of inverters. If the two input buses are designed using a precharged structure, the outputs from the NAND and NOR gates are naturally low (as required in the dynamic implementation) when the buses are precharged high. Furthermore, these NAND and NOR gates can be reused for logic operations in an ALU design.

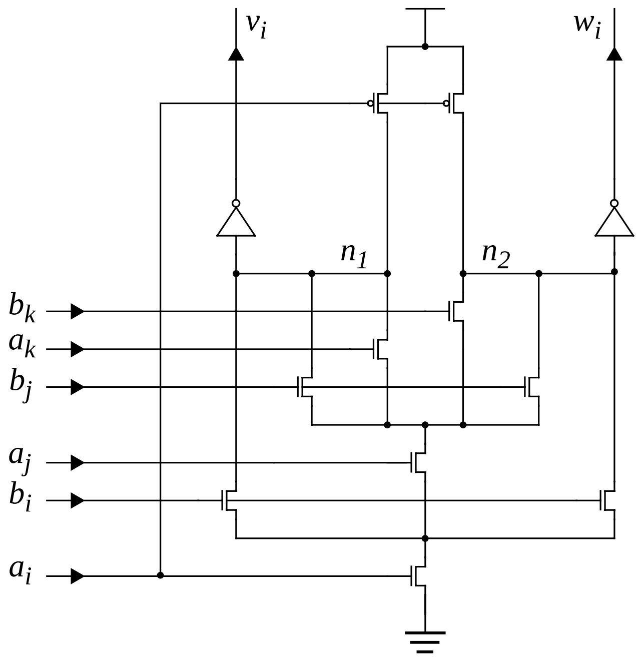


Figure 3-8: Modified implementation of a three-way carry

It could be questioned here whether there is a real difference between this new scheme eliminating the value (0 1) compared with the Brent and Kung scheme which does not use the value (1 1). How can we claim that the new arrangement without the formation of generate and propagate terms has an advantage after adding initial NAND and NOR gates? The answer lies in observing that the constraint of not using the value (1 1) is inherent in the Brent and Kung scheme and therefore an initial formation of the generate and propagate terms is required, whereas the constraint of not using the value (0 1) in the modified implementation of the carry arbitration scheme is introduced as an optimization rather than enforced. The optimization leads to the benefit of eliminating group adders in a carry select adder (see section 3.6) and also results directly in a simplified circuit.

3.5 Refinement of the Manchester carry chain

One simple application of the new scheme is given in this section, where it is used to refine the Manchester carry chain. In the next section, another application is given, which is to simplify the design of carry select adders.

A wide variety of addition schemes and their implementations are available to serve different performance/cost requirements. One of them is the well known Manchester carry chain [31], which is often found in custom datapaths combined with the carry skip scheme. However a problem with the Manchester carry chain is that too many pass transistors are in series along the carry chain, which degrades the performance especially in CMOS designs with a low supply voltage. To avoid this problem, buffers are usually used to divide the carry chain into several sets of series pass transistors as shown in figure 3-9.

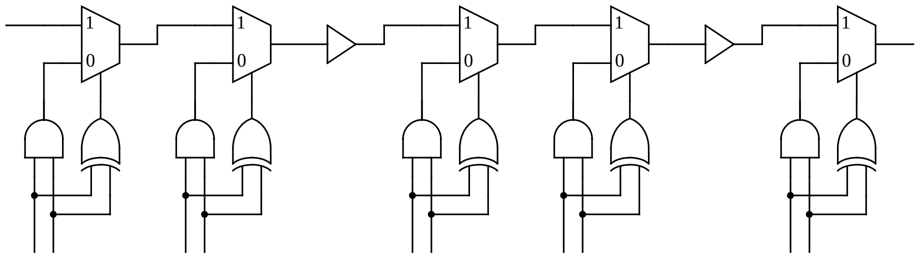


Figure 3-9: Manchester carry chain with buffers

Instead of using buffers to limit the number of pass transistors in series, the carry chain can be rearranged using the part of the circuit in figure 3-6 based on the concept of carry arbitration. Figure 3-10 shows a new carry chain in which the output of one set of series pass transistors is connected to the control gate of the next stage. By so doing, we avoid the series connection of pass transistors without any overhead. It is worth noting that a double pass-transistor logic design style [64] should be used in order to exploit this new carry chain fully.

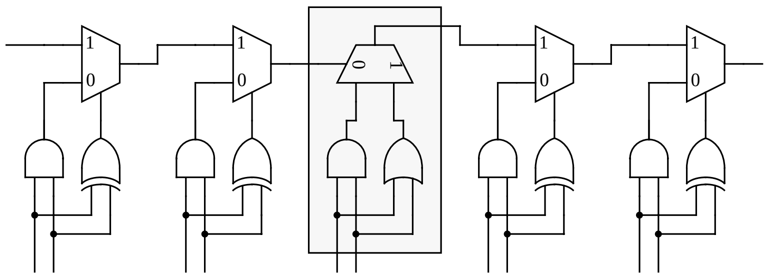


Figure 3-10: Manchester carry chain without buffers

Obviously, this new implementation of the Manchester carry chain can be derived directly from the truth table without any knowledge of the carry arbitration scheme. The new implementation was found during the development of the carry arbitration scheme.

3.6 Simplification of carry select adders

Figure 3-11 shows an adder design using the conventional carry select scheme [57]. The inputs are divided into d-bit (or possibly variable width) groups. Two d-bit adders are needed per group. One is an adder with a zero carry-in and the other with a one carry-in. The carry generator is responsible for generating the boundary carries for all groups, which are then used to select the appropriate sum using a multiplexer.

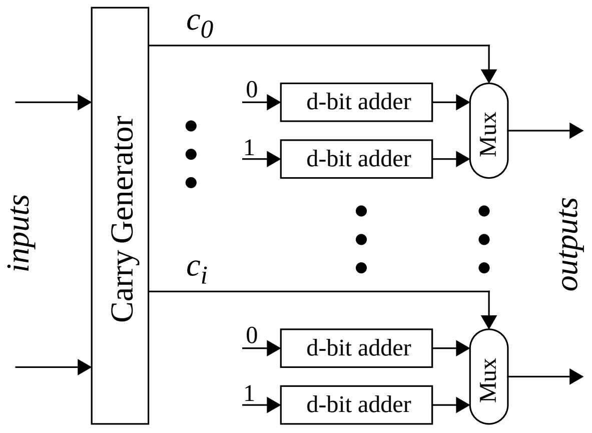


Figure 3-11: Carry select adder

Design decisions must be made to choose the appropriate group widths in order to balance the worst case delays of both the carry generator and the group adders. If the group adders are made too long, the decreasing delays in the carry generator are exceeded by the increasing delays of the group adders. If the group adders are made too short, the logical complexity of the carry generator increases and its delay determines the total adder delay. Usually a mechanism for carry computation with low complexity, such as the Manchester carry chain, is chosen in the group adders. So the group cannot be made long (normally less than or equal to 8 bits) due to its linearly increasing delay. This leads to the increasing complexity of the carry generators. Carry generators designed using conventional approaches consume much chip area and power as well as limiting the ultimate performance that can be achieved.

If carry arbiters, modified according to the circuit in figure 3-6 or figure 3-8, are used as elements to design the carry generator, the group adders can be eliminated as shown in figure 3-12. The output v_i is the carry out generated with a one carry-in and the output w_i is the carry out generated with a zero carry-in if no carry requests issue from input signals. Choosing the length of the group adders becomes unnecessary since the group adders are not required at all. This results in a significant reduction of chip area, especially when the groups are made long, since group adders also need a mechanism for carry computation.

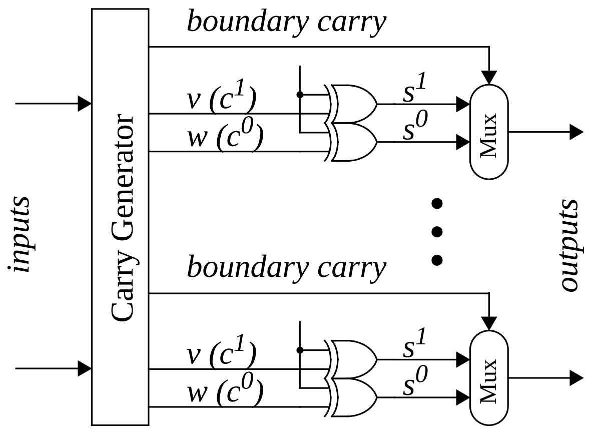


Figure 3-12: New carry select adder

The intermediate signals v_i and w_i in the carry generator are elegantly reused for generating the two intermediate sums. If the signals v_i and w_i are equal (meaning that the carry has been generated), the final result is independent of the boundary carry since the two intermediate sums are equal. If the signals v_i and w_i are different, the two intermediate sums with the signals v_i and w_i as inputs are those with a one carry-in and a zero carry-in, respectively. Therefore the boundary carry can choose one of these two intermediate sum results to use as the final sum result. It is clear that these two intermediate signals v_i and w_i have dynamic meanings, and this is the main feature of the proposed scheme. It is worth noting that the carry generator itself is much simplified and optimized by using the proposed scheme

3.7 Adder design for AMULET3i

A 32-bit adder for AMULET3i has been designed, using the architecture in figure 3-12, to demonstrate the proposed scheme. The whole adder is visualized (but not divided) as consisting of four 8-bit long groups. Figure 3-13 illustrates the block diagram for the AMULET3i adder.

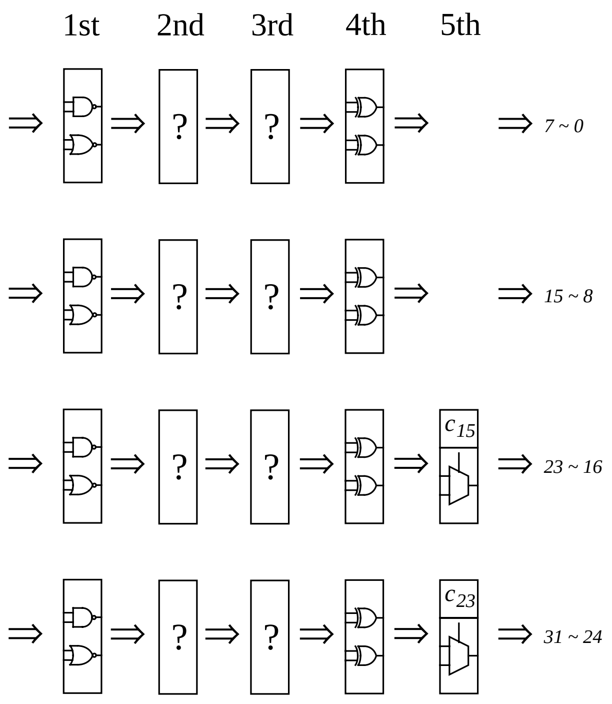


Figure 3-13: AMULET3i adder block diagram

The AMULET3i adder compromises one row of conversion circuits containing 2-input NAND and NOR gates and two rows of three-way carry arbiters to generate all the intermediate signals v_i and w_i. Additionally, two extra three-way carry arbiters are needed to compute the boundary carries. These operate in parallel with the XOR gates (the 4th and 5th levels are mostly operating in parallel).

3.8 Circuit design

An efficient three-way carry arbiter design is the key to the whole adder design. The dynamic implementation of a three-way carry arbiter as shown in figure 3-8 was initially chosen since dynamic circuits offer the benefits of increased speed and lower switched capacitance. However, dynamic circuits are sensitive to noise when both the NMOS pull down and the PMOS pull up networks are in the off state. Additional devices as shown in figure 3-14 are, in practice, incorporated into dynamic circuits to combat noise. There is then the problem that the dynamic circuit with the additional device might demonstrate a considerable performance disadvantage since the NMOS pull down network must overdrive the additional device.

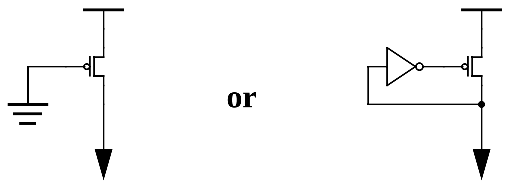


Figure 3-14: Devices for dynamic circuits

We look firstly at the static implementation of a three-way carry arbiter as shown in figure 3-15 before moving on to an alternative implementation. In the case of this fully complementary CMOS circuit, the size of the p-type transistors should be 2 ~ 3 times greater than that of the n-type transistors to compensate for the typically 2 ~ 3 times slower speed of the p-type transistors. As a result, this circuit consumes a large area and is quite slow due to its large input capacitance. The problem can easily be solved by making the size of all the p-typed transistors minimum. However, this change makes the rise time of the circuit dramatically increase.

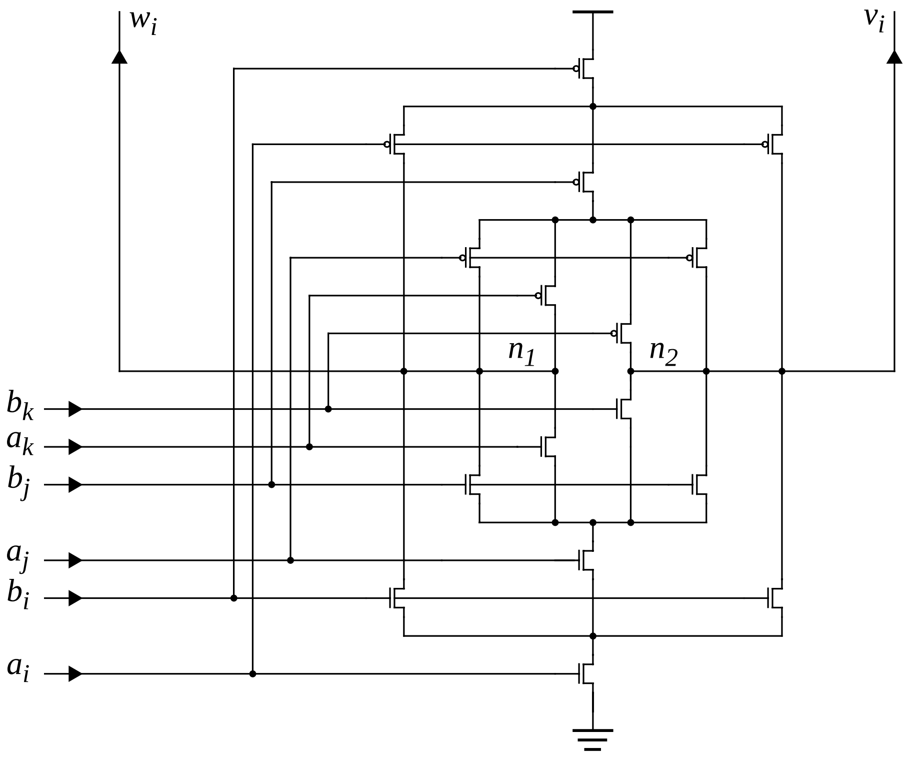


Figure 3-15: Static Implementation of a three-way carry arbiter

The original idea of dynamic circuits can be reintroduced here but all the p-type transistors are retained. Figure 3-16 shows a new implementation combining both static and dynamic circuit properties. Two p-type transistors P1 and P2 are introduced for precharging. While this may seem like a foolish idea at first, it has some merit. Although the new implementation is almost the same as the static implementation apart from the two extra p-type transistors at the circuit level, the operation of the two circuits is totally different.

All the p-type transistors except these two precharge transistors are minimum sized in the new implementation. The p-type transistors in the original static implementation should be oversized by 2 ~ 3 times compared with the size of the n-type transistors to keep the rise time in line with the fall time. The large input capacitance due to the oversized p-type transistor therefore requires a previous stage with more drive strength. This inevitably results in degraded performance and increased power consumption.

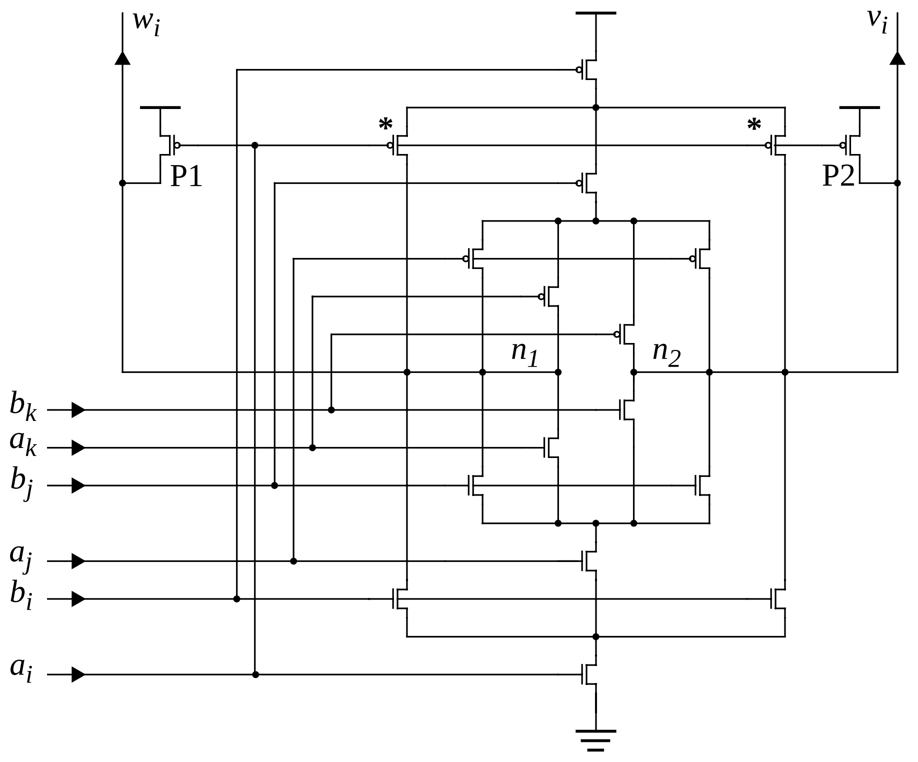


Figure 3-16: New Implementation of a three-way carry arbiter

The new implementation behaves both statically and dynamically, thus having the advantages of these two types of circuit. The transistors marked with an asterisk can, in fact, be eliminated. This very efficient carry arbiter circuit provides a firm foundation for the realisation of a high speed AMULET3i adder.

The three-way carry arbiter shown in figure 3-16 was analysed using HSPICE on extracted layout under the conditions of 3.3 volt supply voltage and 100 ˚C temperature. The simulation results are given in table 3-7. The estimation of power consumption of a circuit is difficult since it is a function of not only its inputs but also of their history. For the sake of simplicity, the power consumption was measured under the assumption of 100% input activity.

Table 3-7: Simulation results of the three-way carry arbiter

	delay	power
typical process case	0.35 ns	72 µW @ 100 MHz
		153 µW@ 200 MHz
worst process corner	0.44 ns	71 µW @ 100 MHz
		148 µW @ 200 MHz

3.9 Layout design

The technology on which the AMULET3i adder is based, is a 0.35 micron triple metal CMOS process. The minimum drawn width is 0.4 micron.

The layout of the AMULET3i adder uses a full-custom style for the datapath, where the circuit and layout are optimized. The bit pitch in the datapath is 82 λ. Data flow is routed horizontally in metal3, while control flow is relayed vertically in metal2. Metal1 is used for local interconnections in cells. The global power rails use metal1 and metal3, and the local power rails use metal2.

3.10 Evaluation

An evaluation of the AMULET3i adder in terms of performance, power consumption and silicon area is presented in this section.

3.10.1 Performance

The critical path covers one NAND/NOR gate stage, three three-way carry arbiter stages and one multiplexer stage. The critical delay is about 1.8 ns under worst-case conditions (*Vdd* = 3.3V, *Vss*=0.1V, slow-slow process corner, at 100 $^{\circ}$C temperature). This results in a 460 MHz computational speed with a 20% engineering margin.

3.10.2 Power consumption

The estimation of power consumption is a difficult problem as it is a strong function of the inputs and their history. A rough estimate of power consumption is given based on some assumptions. It is highly unlikely that all data bits will change for every data value. Based on the assumptions that half the data bits on average will change and that the dynamic switching power is 90% of the total power, the power estimate of the datapath is about 8 and 17 mW operating at 100 and 200 MHz (under typical process conditions), respectively.

3.10.3 Silicon area

The silicon area of the datapath is 686 $\lambda \times 2624\ \lambda$ ($137.2 \times 524.8\ \mu m^2$). Figure 3-17 shows the physical layout of the datapath of the AMULET3i adder, and illustrates its regular structure.

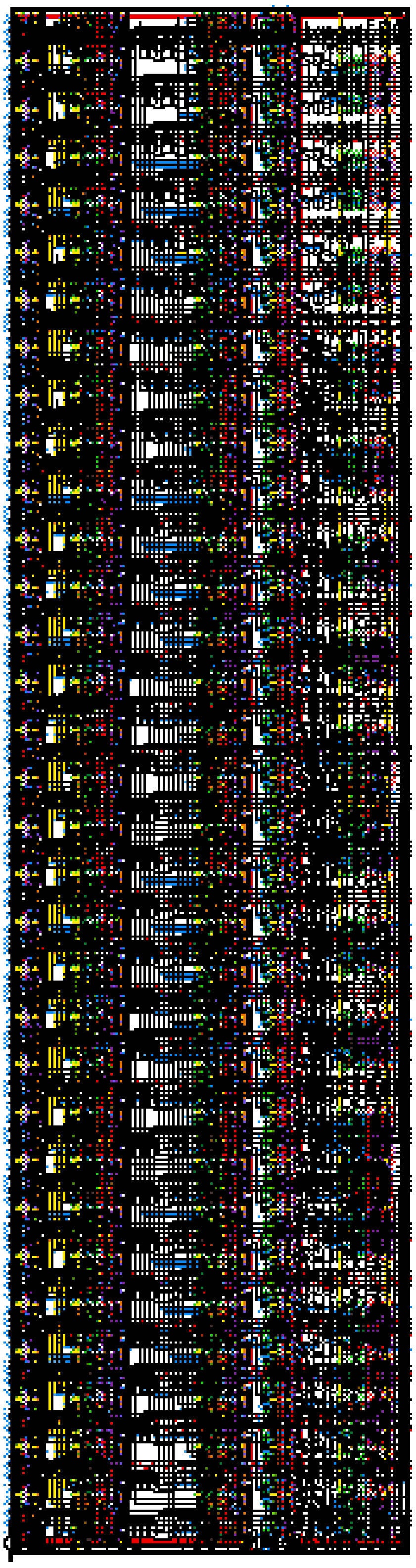


Figure 3-17: Physical layout of the adder datapath

3.11 Summary

A carry arbitration scheme is proposed (and has been patented) for parallel adder circuits. The proposed scheme provides an efficient encoding in which the carry is generated by arbitrating several input carry requests, exploiting the associativity of the carry computation. The new scheme not only leads to high speed adders due to the reduction in the required layers of logic, but also offers a regular and compact layout and uniform fan-in and fan-out loadings.

CMOS implementations of carry arbiters have been derived and modified. The meaning of the modified version is twofold. If the intermediate signals v_i and w_i are equal, it means that the carry has been generated. If they are different, it means that there are no carry requests from the input signals. The intermediate signal v_i can be viewed as the carry out generated with a one carry-in and the intermediate signals w_i as the carry out generated with a zero carry-in.

A new implementation of a three-way carry arbiter has been developed, which behaves both statically and dynamically, thus having the advantages of both static and dynamic circuits.

Two applications of the scheme are given in this chapter. One is to refine the Manchester carry chain. Another is to simplify carry select adders.

A high performance, low power asynchronous 32-bit adder with a reasonable hardware resource has been developed for AMULET3i, demonstrating the feasibility and effectiveness of the new scheme. It takes 1.8 ns to complete a 32-bit addition and occupies 137.2 μm × 524.8 μm of chip area in a 0.35 μm triple metal CMOS technology. The power estimate of the datapath is about 8 and 17 mW operating at 100 and 200 MHz (under typical process conditions), respectively.

It is worth noting that the proposed scheme is general and can be applied to both asynchronous design and synchronous design. The new scheme was used in the adder design for the ARM Piccolo DSP processor [65].

Multiplier design 4

This chapter presents the design of a multiplier for AMULET3i. Attention is focused on CMOS circuit design techniques. We start with an introduction to basic algorithms for multiplication. The asynchronous multiplier for AMULET2e is then reviewed, as this formed the starting point for the design of the AMULET3i multiplier. Finally, the design of an asynchronous multiplier for AMULET3i is developed which uses the modified Booth's algorithm integrated with a new encoding technique for adjusting the product result of an unsigned number multiplication. Post-layout simulation, in a 0.35 micron triple metal CMOS technology, shows that it takes 11.2 ns (2.8 ns $\times$ 4 cycles) to complete the computation of a 32-bit long multiplication in the worst case.

4.1 Introduction

The general principle by which computers carry out multiplication is quite simple. The multiplication of two 1-bit binary numbers is even simpler than addition since there is no need for the carry to propagate. Consider the multiplication of two unsigned numbers using the ordinary paper-and-pencil method. Figure 4-1 illustrates a dot representation [66] for the multiplication of two 8-bit unsigned numbers. Roughly speaking, the number of dots reflects the amount of hardware in a parallel multiplier or the processing time for a serial multiplier. The height of the dot diagram relates to the latency for carrying out the multiplication. The paper-and-pencil method comprises two distinct steps. Firstly, all the partial products are generated simultaneously, then they are added together proceeding column-wise from right to left. Although conceptually simple, a direct mechanical implementation of the paper-and-pencil method would lead to a very inefficient design [67] due to the asymmetry between different columns.

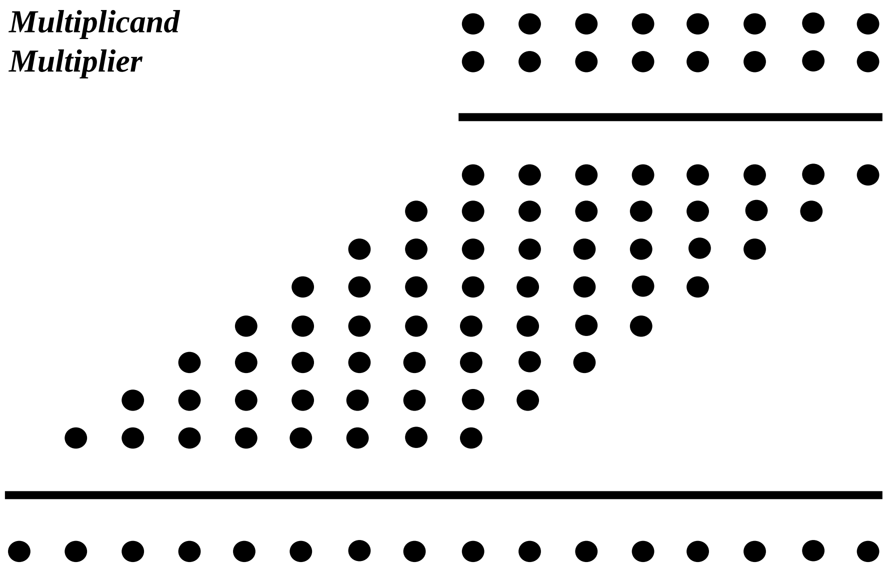


Figure 4-1: Dot representation of 8 $\times$ 8 bit add and shift multiplication

Looking row-wise, there is a degree of symmetry in terms of the number of dots, though they have different weights in each row. It is thus desirable to proceed row-wise from top to bottom for VLSI implementations, either sequentially or using parallel hardware. The scheme derived from a straightforward application of the paper-and-pencil method is essentially a process of repeated adds (conditionally adding the multiplicand to a running partial product) and shifts. Therefore there are two basic approaches to improving the speed of multiplication: making each addition faster, and reducing the number of additions required. An additional technique is to use an “early out” scheme [68], which depends upon the operands presented.

4.1.1 Making each addition faster

A simple multiplier using the scheme derived from the paper-and-pencil method is illustrated in figure 4-2. The multiplier and multiplicand are initially placed in registers *A* and *B*, respectively; register *P* which holds the partial product is initially 0. Each multiply step consists of replacing *P* with the sum of *P* and *B* (AND-gated by the least significant bit of *A*), and then shifting *P* and *A* together one bit right at a time.

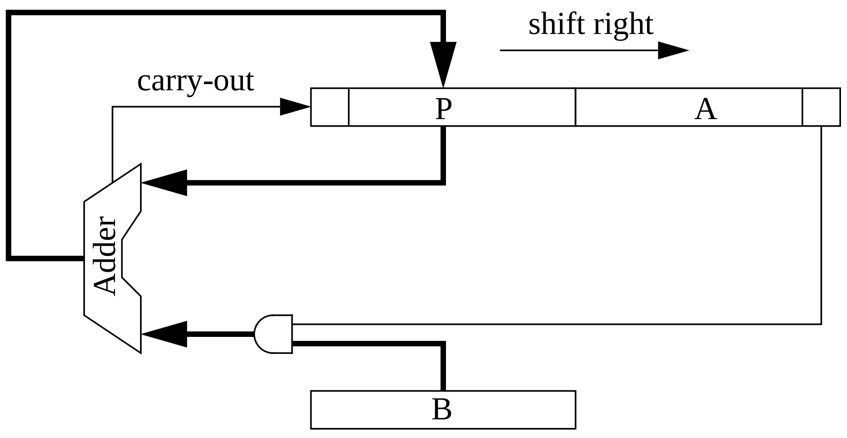


Figure 4-2: A simple multiplier

Obviously, the time necessary for carry propagation imposes the ultimate limit on the speed of addition and thus multiplication. All the techniques for faster adders can be used here to speed up multiplication. However, multiplication is a special case of repetitive addition in which the intermediate results of all but the last addition are not of any interest. So it is not necessary for the carries to propagate during every multiply step. Instead, the carries generated during one step may be saved and used again in the next step with an appropriate shift. In this way, a partial sum and a partial (saved) carry together present the partial product. Thus each multiply step needs only the time required for a 1-bit addition since all the carry bits are passed from internal intermediate signals to outputs. Only on the last step need the carries be propagated to completion instead of being saved. A carry-save multiplier is illustrated in figure 4-3.

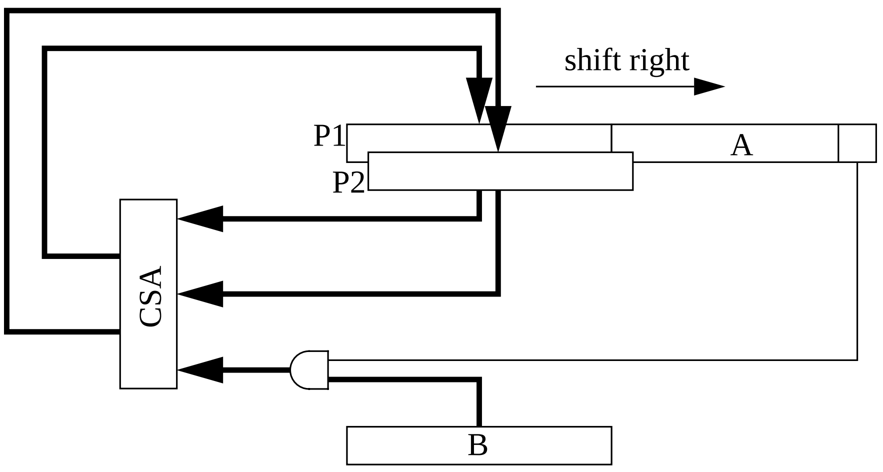


Figure 4-3: A carry-save multiplier

Alternatively, redundant number systems [58] can be used to achieve addition without carry propagation. Take the radix-2 redundant representation as an example, which has a digit set $\{\overline{1}, 0, 1\}$ where $\overline{1}$ denotes -1. An n-bit redundant number $Y = [y_{n-1},\ldots,y_0]$ has the value $\sum_{i=0}^{n-1} y_i \times 2^i$, where y_i belongs to $\{\overline{1}, 0, 1\}$. This is similar to an unsigned binary representation except that y_i can be $\overline{1}$. The key idea to avoid carry propagation when adding two redundant numbers is to set the intermediate sum to 0 or 1 when there is a negative carry from the next lower order position and to set the intermediate sum to 0 or $\overline{1}$ when there is a positive carry from the next lower order position. By so doing, there is no need to know the lower order carry to obtain the carry as the intermediate sum and carry from the next lower order position cannot both be 1 and -1 at the same time.

It is worth noting that since the partial product has been replaced by a partial sum and a partial carry, the carry-save scheme in effect employs a redundant concept. The difference is that the carry-save scheme uses the digit set {0, 1, 2, 3} instead of $\{\bar{1}, 0, 1\}$ since the combination of a partial sum and a partial carry results in four values of unsigned number.

4.1.2 Reducing the number of additions required

One way to reduce the number of additions required is to use multi-operand additions (more than three operands), which can add many numbers simultaneously, instead of just two or three at a time. A Wallace tree [69] is well known for its optimal computation time. However, its implementation is often too expensive to justify the speed obtained. Several tree or array structures derived from the Wallace tree have been proposed by trading speed for regularity [70-72].

Another way to reduce the number of additions required is to skip over any contiguous string of 1s and 0s in the multiplier, rather than form a partial product for each bit. The original Booth's algorithm [13] is based on this idea.

Taking a 32-bit two's complement number as an example. A 32-bit signed word A = $(a_{31}a_{30} \ldots a_1\, a_0)$ can be expressed as:

$$A = -2^{31}a_{31} + \sum_{i=0}^{30} 2^i a_i$$

The principle of the original Booth's algorithm is to rewrite this number as:

$$A = -2^{31}a_{31} + \sum_{i=0}^{30} 2^i a_i = \sum_{i=0}^{31} 2^i (a_{i-1} - a_i) = \sum_{i=0}^{31} 2^i k_i$$

where a_{-1} is a dummy bit that is equal to zero, and k_i (= a_{i-1} - a_i) belongs to the digit set of {$\bar{1}$, 0, 1}. Thus, the original Booth's algorithm may be viewed as a conversion of the multiplier representation from a conventional code into a redundant code. The redundant code is {1, 0, 1}, and the radix is two. The radix (r = 2^b) determines how many bits (b) of multiplier are retired in an iteration.

A redundant addition or carry-save addition scheme encodes the *multiplicand* using a redundant representation, while the original Booth's algorithm encodes the *multiplier* using a redundant representation. It is worth noting that the radix of the algorithm and the radix of the number representation are not the same concept.

A slightly different algorithm, called the modified Booth's algorithm [14], considers groups of bits of the multiplier rather than skipping over arbitrarily long strings. The multiplier bits are divided into two-bit groups. Three bits are scanned at a time, two bits from the present group and the third bit being the higher-order bit of the next lower-order group.

The principle of the modified Booth's algorithm is to rearrange a number as:

$$A = -2^{31}a_{31} + \sum_{i=0}^{30} 2^i a_i = \sum_{i=0}^{15} 2^{2i}(a_{2i-1} + a_{2i} - 2a_{2i+1}) = \sum_{i=0}^{15} 2^{2i}k_i$$

where a_{-1} is a dummy bit that is equal to zero, and k_i (= a_{2i-1} + a_{2i} - $2a_{2i+1}$) belongs to the digit set of {-2, -1, 0, +1, +2}. Thus, the modified Booth's algorithm may be viewed as a conversion of the multiplier representation from a conventional code into a redundant code. The redundant code is {-2, -1, 0, 1, 2}, and the radix is four. A radix 4 algorithm retires 2 bits of multiplier in an iteration.

The modified Booth's algorithm is described in table 4-1.

Table 4-1: Modified Booth algorithm

Group	Action
0 0 0	0
0 0 1	+1
0 1 0	+1
0 1 1	+2
1 0 0	-2
1 0 1	-1
1 1 0	-1
1 1 1	0

The modified Booth's algorithm is more commonly used than the original Booth's algorithm since VLSI implementations favour its fixed shift of the multiplier in each iteration. The modified Booth's algorithm halves the number of additions that have to be performed compared with the simple paper-and-pencil method, therefore speeding up the multiplication.

An additional technique that may be used to further reduce the number of additions is to check in each multiply step whether the shifted multiplier register contains only 1s or 0s, and, if so, to terminate the multiply process early. Note that the final result must be correctly aligned.

4.2 AMULET2e multiplier

The AMULET2e multiplier has been described elsewhere [16], so only a summary is presented here. Figure 4-4 shows the organisation of the AMULET2e multiplier.

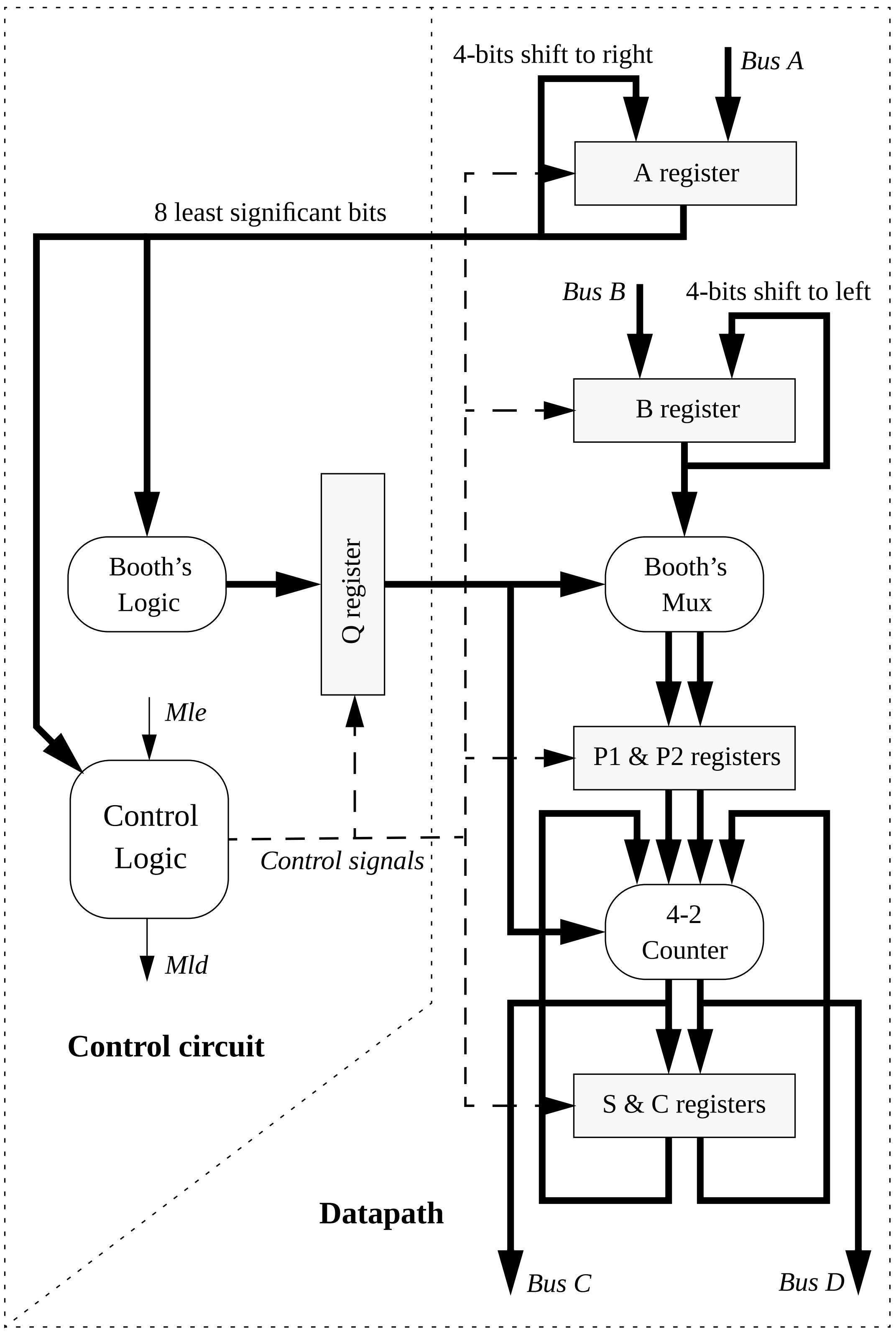


Figure 4-4: AMULET2e multiplier organization

❑ The AMULET2e multiplier is a 32-bit normal multiplier, which means that the final result is the least significant 32 bits of the 64-bit product. One benefit from this sort of multiplier is that both unsigned and signed number multiplications give the same result. The AMULET2e multiplier does not detect overflow and leaves it to software either to constrain the operands to ensure there is no overflow or to perform explicit checks (as required by the ARM instruction set definition).

❑ The AMULET2e multiplier uses the modified Booth's algorithm. Two stages of the Booth's algorithm are performed in each cycle by shifting four bits at a time. The AMULET2e multiplier employs an "early out" scheme, which depends on the operands provided, hence achieving statistical speed improvement and saving power.

❑ An iterative structure was chosen combined with a pipeline technique in the AMULET2e multiplier to reduce the hardware cost by increasing hardware utilization. The partial products in the AMULET2e multiplier remain at a fixed alignment to avoid difficulty when selecting the final result in "early out" cases. Instead, the multiplicand and multiplier shift left and right, respectively.

❑ The AMULET2e multiplier uses the high speed, low power true single-phase clocking (TSPC) methodology and pass-transistor logic style. Novel 4-2 Counters are used which are symmetric with respect to their inputs and outputs. Transistors with small size were favoured for low power.

❑ The AMULET2e multiplier was designed in a 0.5 μm three metal CMOS process technology. The layout is regular and compact with a datapath area of only 320×710 μm^2. The working chip has a 6.5 ns multiplier cycle time [11].

4.3 Multiply support for AMULET3i

AMULET3i supports two classes of multiply instruction: a normal 32-bit result and a long 64-bit result. Both types of multiply instruction can also optionally perform an accumulate operation.

4.3.1 Normal multiply

There are two normal multiply instructions, producing 32-bit results:

MUL

The MUL instruction multiples the values of two registers together, truncates the result to 32 bits, and stores the result in a third register.

MLA

The MLA instruction multiples the values of two registers together, adds the value of a third register, truncates the result to 32 bits, and stores the result into a fourth register.

Both instructions can operate on signed or unsigned numbers since only the least significant 32 bits of the product result are stored in the destination register and the type of the operands does not affect this value.

4.3.2 Long multiply

There are four long multiply instructions, producing 64 bit results:

SMULL & *UMULL*

These two instructions multiply the values of two registers together and store the 64 bit result in a third and a fourth register. There are signed (SMULL) and unsigned (UMULL) variants. The signed variants produce a different result in the most significant 32 bits if either or both of the source operands is negative.

SMLAL & UMLAL

These two instructions multiply the values of two registers together, add the 64 bit value from a third and a fourth register and store the 64 bit result back into those (third and fourth) registers. There are again signed (SMLAL) and unsigned (UMLAL) variants. These two instructions perform a long multiply and accumulate.

4.4 Multiplier organization

The target for the multiplier design for AMULET3i is a 2 times speed improvement compared with the AMULET2e multiplier, with a reasonable area increase. Latency and chip area were considered the most important parameters to be minimized. The AMULET3i multiplier is not optimized for low power since multiplication instructions are not very often used compared with other instructions for general purpose applications. However, low power was kept in mind during the development of the design.

4.4.1 First design iteration

The first design decision was to use the modified Booth's algorithm, processing 8 bits at a time. The reasons are twofold. Firstly, based on the evaluation of the AMULET2e multiplier, this approach is likely to meet the speed target. Secondly, an 8-bit scheme, just having four cases (caused by early outs) to choose from, simplifies the product result select compared with the eight cases arising from the “early out” scheme with 4 bits at a time. This difficulty was avoided in the AMULET2e multiplier by shifting the multiplicand left while the partial product remains fixed, since the most significant 32 bits of a product result can be thrown away. However, as the multiplier for AMULET3i supports long multiply instructions, the difficulty cannot easily be avoided as the multiplicand should remain fixed here while the partial products are shifted right.

The second design decision was to define an iterative structure for the AMULET3i multiplier. It is possible to implement a fast parallel 32-bit multiplier, however, a significant amount of hardware would be needed. On the other hand, serial multipliers use less area but are quite slow. A serial/parallel iterative structure was chosen as a good compromise for the AMULET3i multiplier.

The initial design is shown in figure 4-5. A 64 bit accumulate value can be used to initialise one of the partial product registers *P1* and *P2* (the most significant 32 bits and the least significant 32 bits of an accumulate value are in *P1L* and *P1H* or *P2L* and *P2H*, respectively). Multiplier data can be stored into the least significant 32 bits of either of the partial product registers *P1* or *P2*. The most significant 32 bits of one of the partial product registers *P1* or *P2* is unused and should be initialised to 0. This initial version of the design presents a minimum hardware requirement.

4.4.2 Encoding technique

As described previously, the multiplier for AMULET3i should support both unsigned and signed numbers. In fact, the modified Booth’s algorithm can also be used with an

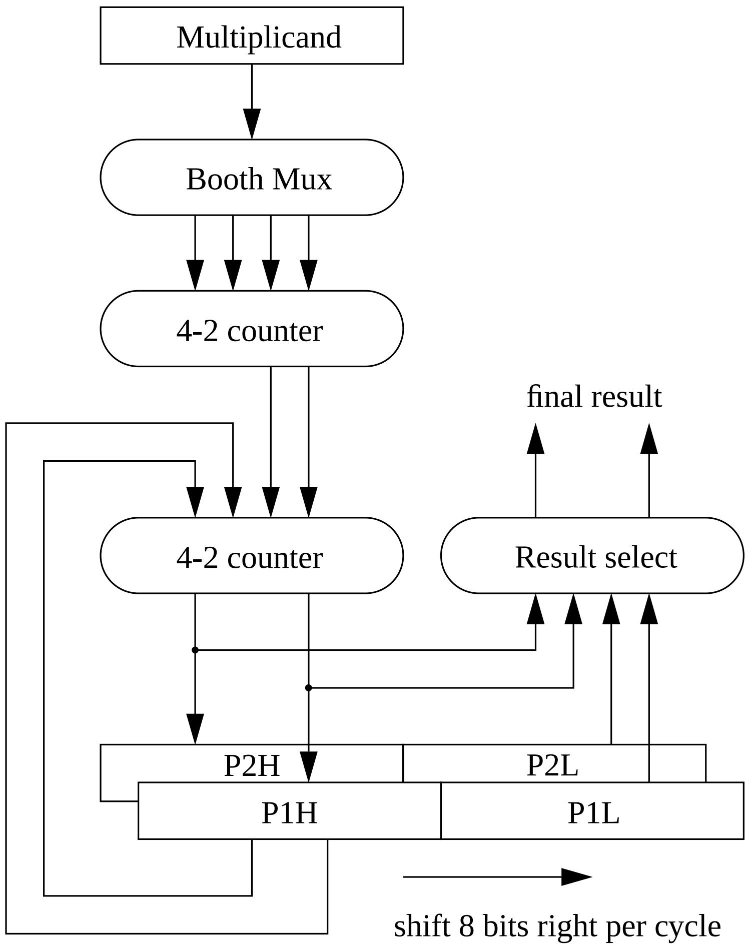


Figure 4-5: First version

unsigned system. For an unsigned number multiply operation, an extra action must be performed to adjust the product result. The conventional equation of the modified Booth's algorithm for an unsigned number is, in the case of a 32-bit number, to rearrange an unsigned number $A = (a_{31}a_{30} \dots a_1\, a_0)$ as:

$$A = 2^{31}a_{31} + \sum_{i=0}^{30} 2^i a_i = \sum_{i=0}^{15} 2^{2i}(a_{2i-1} + a_{2i} - 2a_{2i+1}) + 2^{32}a_{31} = \sum_{i=0}^{15} 2^{2i}k_i + 2^{32}a_{31}$$

where a_{-1} is a dummy bit that is equal to zero, and k_i $(= a_{2i-1} + a_{2i} - 2a_{2i+1})$ belongs to the digit set of {-2, -1, 0, +1, +2}. Obviously, an adjustment value (a multiplicand value) can initially be put into either register *P1L* or *P2L* to represent the term $2^{32}a_{31}$. However, this cannot easily be done since one of the registers *P1L* and *P2L* is used for the most significant 32 bits of an accumulate value and the other is used for a multiplier operand. One observation is that one of the registers *P1H* and *P2H* is for the least significant 32 bits of an accumulate value and the other is left unused. The new idea introduced here is to put an adjustment value in one of the registers *P1H* and *P2H*.

A signed or unsigned number can be expressed as:

$$A = 2^{31}a_{31} + \sum_{i=0}^{30} 2^i a_i = \sum_{i=0}^{15} 2^{2i}(b_{2i} + b_{2i+1} - 2b_{2i+2}) + a_0 = \sum_{i=0}^{15} 2^{2i}k_i + a_0$$

where $b_0 = 0$, $b_i = a_i$ $(1 \le i \le 31)$, and $b_{32} = sign \times a_{31}$. The *sign* bit indicates that signed numbers are used if it is 1 and unsigned numbers are used if it is 0. In this way, an adjustment value can initially put into either register *P1H* or *P2H* to present the term a_0.

4.4.3 Second design iteration

From figure 4-5, a multiply cycle should cover the delay of two 4-2 Counters, one Booth mux cell and one register. In order to improve the speed, a common pipeline technique can be used, as shown in the figure 4-6. Two additional pipeline registers are added to the initial version. This does not cause a big increase in hardware since part of registers can be merged efficiently into the preceding 4-2 Counter as we will see later in the circuit design. However, the pipeline register causes a one clock cycle skew between the partial products and the signals before the pipeline registers since the partial product registers are shift registers. A multiplexer can be used before the partial product registers to solve the skew problem as is frequently done in clocked designs. The alternative approach is to

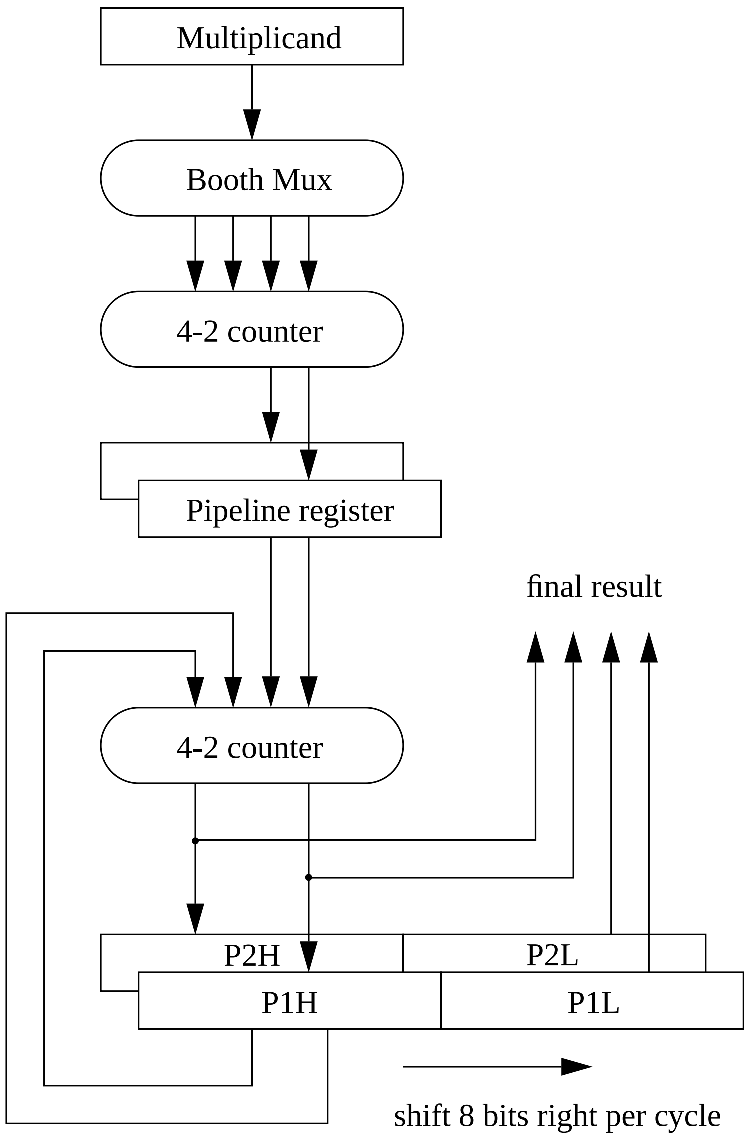


Figure 4-6: Second version

use gated registers (conditional clocking) for partial products. Only on the first cycle are the partial registers disabled and the contents of the registers remains unshifted, therefore the partial registers are naturally aligned with the incoming signals from the pipeline registers after the first cycle.

The first approach (using a multiplexer) will suffer a hardware overhead, whereas the second approach (conditional clocking) will violate the high speed true single-phase clocking methodology we will use in the circuit design as the clock signals for the pipeline registers and the partial product registers have to be separated due to the gated clock requirement for the partial product registers. The two above approaches were heavily influenced by the clocked design methodology.

In fact, the skew problem can easily be solved within the asynchronous framework by making the pipeline registers initially transparent. It will be seen that the first cycle time must cover the whole path delay just as in the non-pipelined case. However, this does not matter for an asynchronous design which can have variable cycle times. This is an example of how nicely an asynchronous design can solve problems which can only be solved with much effort in clocked designs.

Another change is that a final shifter for “early out” cases is not used since there is difficulty in the layout stage. Though the number of tracks for buses is ten per bit pitch, six buses must be reserved for global use and only four local buses are available for the multiplier. As a result, the final result can be quickly shifted out instead.

4.4.4 Sign extension

Due to the two negative terms (-1 and -2) in the modified Booth’s algorithm, the sign bit (the most significant bit) of the partial products has to be extended up to the most significant bit of the expected result. This means that in a real circuit implementation the sign bit has to be broadcast up to the most significant bit of the expected result and this may cause both decreased circuit speed, since a heavy capacitance load arises from the high fan-out of the sign bit, and increased layout area. The scheme presented below avoids these drawbacks.

Consider a number A_{ext} of a k-bit signed partial product $A = (a_{k-1}a_{k-2} \ldots a_1\ a_0)$, which must be sign extended by s bits. Its value is:

$$A_{ext} = -2^{s+k-1}a_{k-1} + \sum_{k-1}^{s+k-2} 2^i a_{k-1} + \sum_{i=0}^{k-2} 2^i a_i$$

The above equation can be rearranged as:

$$A_{ext} = \sum_{k-1}^{s+k-1} 2^i + 2^{k-1}(1 - a_{k-1}) + \sum_{i=0}^{k-2} 2^i a_i$$

From the equation, instead of direct sign extension, constant 1s (the first term) can be added at the most significant s+1 bit positions of the number A_{ext} and the inverted a_{k-1} (the second term) replaces the original a_{k-1}. All the constant 1s of the partial products can be pre-calculated as a adjustment value.

4.5 Circuit design

The true single-phase clocking (TSPC) methodology [15] and pass-transistor logic style [73-77] were chosen for the circuit design in order to achieve high performance and low power. One main advantage of the true single-phase clocking methodology is that the clock skew problem of complementary phase or multi-phase clocking schemes is avoided. Another advantage is its low power consumption as only one enabling signal is required. Pass-transistor logic style is flexible for the design of arithmetic components.

4.5.1 Booth mux cell design

The modified Booth's algorithm examines three bits of the multiplier at a time to determine whether to add 0, +1, +2, -1, or -2 times the multiplicand. The Booth mux cell performs this function, and it steers the appropriate multiplicand value to the output.

Figure 4-7 shows the circuit of the Booth mux cell used in the AMULET3i multiplier. Some effort was expended to ensure that only one path from the input to the output is on at any time, minimising short circuit currents for low power reasons.

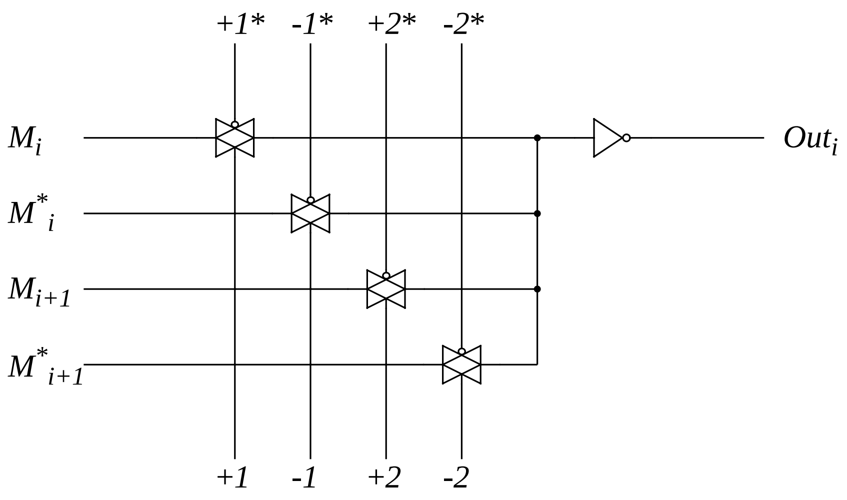


Figure 4-7: Booth mux cell

The Booth mux cell was analysed using HSPICE on extracted layout under the conditions of 3.3 volt supply voltage and 100 °C temperature. The simulated results are given in table 4-2. The estimation of power consumption of a circuit is difficult since it is a function of not only its inputs but also of their history. For the sake of simplicity, the power consumption was measured under the assumption of 100% input activity.

Table 4-2: Simulation results on the Booth mux cell

	delay	power
typical process case	0.61 ns	41 µW @ 100 MHz
		87 µW @ 200 MHz
worst process corner	0.72 ns	37 µW @ 100 MHz
		78 µW @ 200 MHz

4.5.2 4-2 Counter design

4-2 Counters [78-82] are used to speed up the partial product compression process. The main advantage of 4-2 counters over the more familiar 3-2 counters (i.e., full adders) is that their structure is analogous to a binary tree, which leads to regular layout and improved speed. Logically, a 4-2 counter consists of two full adders as shown in figure 4-8 and has four XOR gate delays. Since the *Cout* signal is independent of the *Cin* signal, there is no propagation problem when several 4-2 counters are abutted into the same row; this is the key idea behind 4-2 counters. A 4-2 counter is similar to but different from a 5-3 counter. A 5-3 counter has three different weights for the outputs, while a 4-2 counter has two different weights for the outputs.

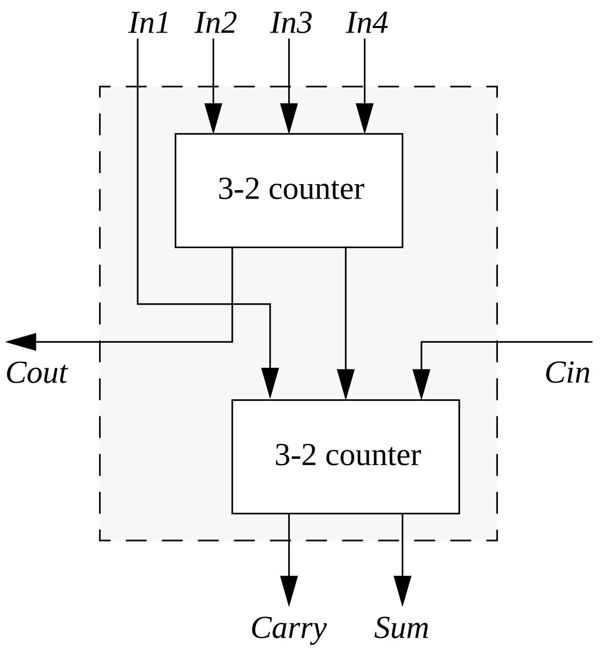


Figure 4-8: 4-2 Counter structure

With careful design, following the truth table as shown in table 4-3, one XOR gate delay can be saved. Figure 4-9 and figure 4-10 show the new 4-2 Counter with and without enable control, respectively. A 4-2 Counter with enable control includes the functionality of the pipeline register (see section 4.5.3). This inclusion is natural and without hardware overhead; just two more n-type transistors are introduced.

The circuits use pass-transistor logic and borrow a common practice from analog designs in which noise immunity is achieved by using quasi-differential signals. The interfacing signals are singled-ended and internal signals are complementary.

Normally the enable signal is high and the circuit behaves statically. The sum and carry delays are balanced for decreasing glitches; this is also desirable since both signals are on the critical path. This is different from the case of adder designs where the carry delay should be minimized since it is on the critical path and the sum delay is off the critical path.

Table 4-3: Truth table for 4-2 Counters

The number of inputs high	*Cin*	*Cout*	*Sum*	*Carry*
0	0	0	0	0
1	0	0	1	0
2	0	1/0^(note)	0	0/1^(note)
3	0	1	1	0
4	0	1	0	1
0	1	0	1	0
1	1	0	0	1
2	1	0/1^(note)	1	1/0^(note)
3	1	1	0	1
4	1	1	1	1
(note) — either *Cout* or *Carry* may be one or zero, but not both.				

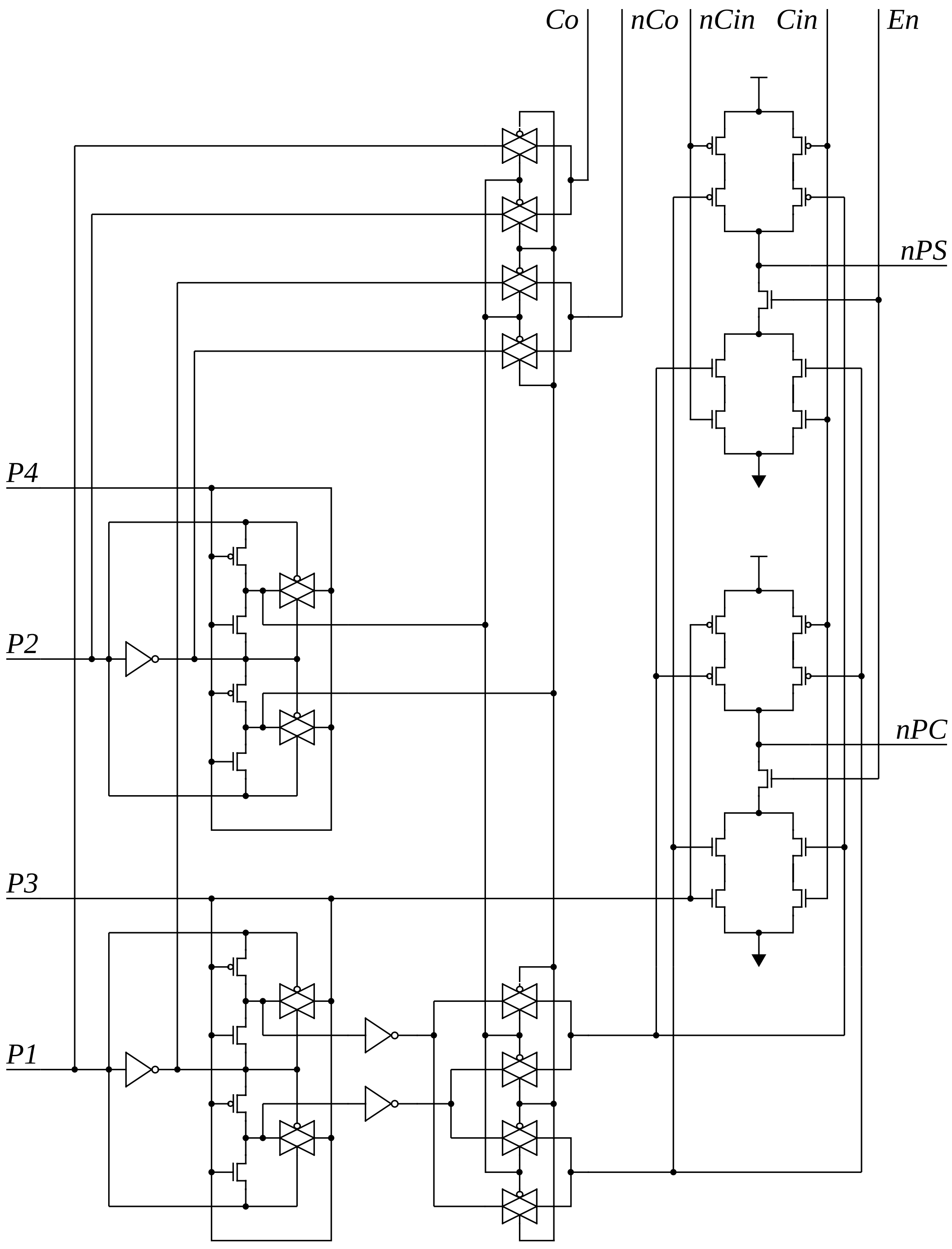


Figure 4-9: 4-2 Counter with enable control

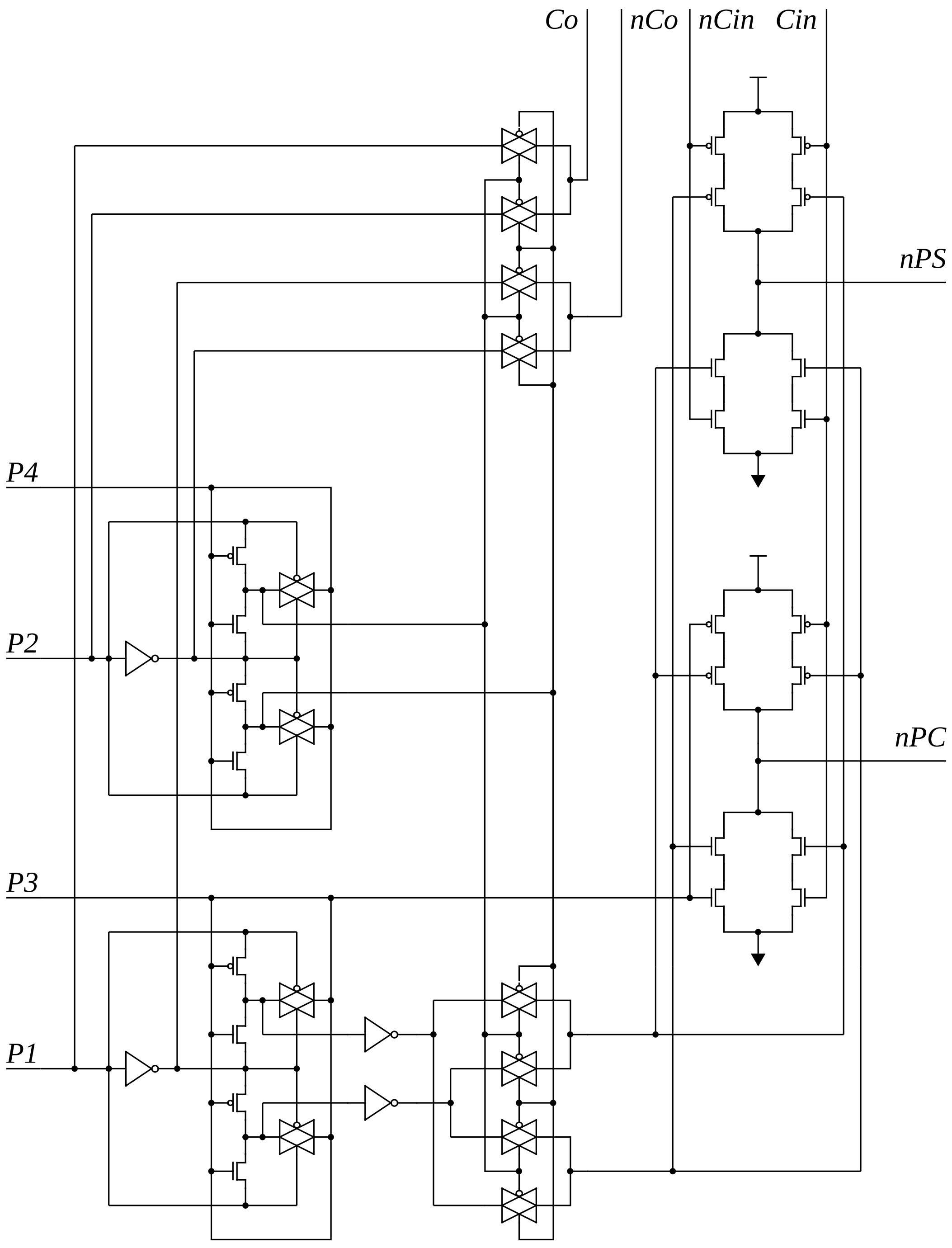


Figure 4-10: 4-2 Counter without enable control

The two 4-2 Counters were analysed using HSPICE on extracted layout under the conditions of 3.3 volt supply voltage and 100 °C temperature. The simulation results are given in table 4-4 and table 4-5. For the sake of simplicity, the power consumption was measured under the assumption that one input is active.

Table 4-4: Simulation results on the 4-2 Counter with enable control

	delay	power
typical process case	1.10 ns	319 µW @ 100 MHz
		644 µW @ 200 MHz
worst process corner	1.40 ns	302 µW @ 100 MHz
		611 µW @ 200 MHz

Table 4-5: Simulation results on the 4-2 Counter without enable control

	delay	power
typical process case	0.97 ns	300 µW @ 100 MHz
		606 µW @ 200 MHz
worst process corner	1.24 ns	285 µW @ 100 MHz
		574 µW @ 200 MHz

4.5.3 Pipeline register design

Figure 4-11 shows the circuit of a pipeline register. The first enabled inverting stage predischarges the node *n1* low and the second enabled inverting stage is opaque when the enable signal *En* is high. At the time that *En* falls, the node *n1* is either pulled high (input *In* low) through two pull-up transistors or remains low (input *In* high), and this level is then stored into the dynamic node *n2* through the second transparent inverting stage when *En* is low. Normally the enable signal *En* is high. Since the signals *Lt* and *nLt* are initially high and low, respectively, which allows the input *In* to propagate down to the node *n2*, the node *n2* has static behaviour. It is obvious that the initially transparent pipeline register not only solves the skew problem (see “Second design iteration” on page 80), but also makes the node *n2* static; otherwise some effort would have to be put into ensuring that the node *n2* was static rather than “floating”. There is no node in the circuit that is in the floating state for an arbitrary long time. It is worth noting that one enabled inverting stage required for a negative edge triggered TSPC register is merged into the last stage of the previous 4-2 Counter.

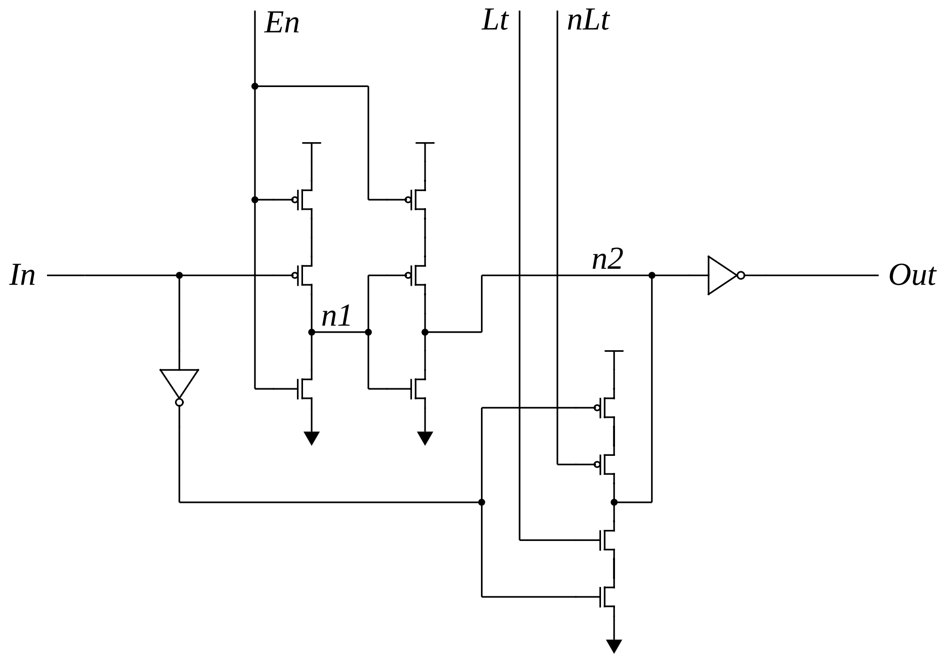


Figure 4-11: Pipeline register

The pipeline register was analysed using HSPICE on extracted layout under the conditions of 3.3 volt supply voltage and 100 °C temperature. The simulation results are given in table 4-6. For the sake of simplicity, the power consumption was measured under the assumption of 100% input activity.

.

Table 4-6: Simulation results on the pipeline register

	delay		power
	In $\longrightarrow$ *En*	*En* $\longrightarrow$ *Out*	
typical process case	0.0 ns	0.46 ns	58 µW @ 100 MHz
			92 µW @ 200 MHz
worst process corner	0.0 ns	0.68 ns	44 µW @ 100 MHz
			83 µW @ 200 MHz

4.5.4 Partial product register design

Figure 4-12 shows the circuit of the partial product register, which comprises three clocked inverting stages. The first stage is transparent and the third stage is opaque when *En* is high. On the other hand, the first stage is opaque and the third stage is transparent when *En* is low. At the time that *En* is high, node *n1* in the second stage is predischarged low. When *En* falls, the node *n1* is either pulled high or remains low, and this level is then transfer into the third stage.

The partial product register also provides a direct load capability. Initially the node *n2* is made static high by the signal *nZ*, and it can then be conditionally discharge depending on the signals *D* and *Lt*.

The partial product register was analysed using HSPICE on extracted layout under the conditions of 3.3 volt supply voltage and 100 °C temperature. The simulation results are given in table 4-7. For the sake of simplicity, the power consumption was measured under the assumption of 100% input activity.

.

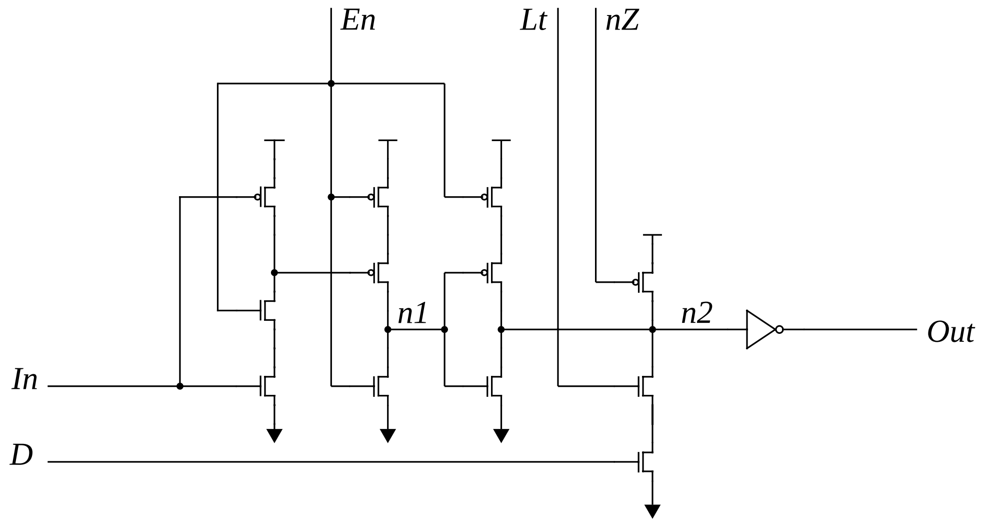


Figure 4-12: Partial product register

Table 4-7: Simulation results on the partial product register

	delay		power
	In $\rightarrow$ *En*	*En* $\rightarrow$ *Out*	
typical process case	0.19 ns	0.47 ns	39 µW @ 100 MHz
			63 µW @ 200 MHz
worst process corner	0.23 ns	0.65 ns	31 µW @ 100 MHz
			57 µW @ 200 MHz

4.5.5 Low power design

The multiplier for AMULET3i is not optimized for low power, however low power was kept in mind during the whole process of design development.

Dynamic logic [83,84] is favourable for low power due to its lower switched capacitance. However, a direct application of dynamic logic in an asynchronous design will cause a state-loss problem since an asynchronous design allows activity to cease for an arbitrarily long time. Therefore low power designs often employ dynamic logic with additional latches or charge-retention circuits to give pseudo-static behaviour. These additions increase the cost and power consumption of the dynamic circuits, thereby compromising their potential advantages. Circuits used for the AMULET3i multiplier are dynamic logic without the above-mentioned encumbrances whilst still retaining externally static behaviour.

The true single-phase clocking methodology has been adopted in the circuit design. The reasons are threefold. Firstly, its dynamic logic which can be integrated with static behaviour is desirable for both low power and high speed. Secondly, only one enabling signal is required and the minimum size and number of transistors are needed in the TSPC registers. Thirdly, it is easy to integrate some logic into a TSPC register to reduce the hardware complexity and overall delay and therefore save power.

To minimize the physical capacitance for low power, transistors are made small whenever this is possible. Cells for the AMULET3i multiplier usually comprise two stages. The first stage contains transistors with the smallest size possible to minimize the required area and power, whereas the second stage uses transistors with greater sizes to ensure that they have the drive capability for their capacitive load.

Reducing the activity of nodes with a large capacitive load is another approach adopted for low power. An early out technique is used, which not only gives a statistical speed improvement but also saves power. Attention is also given to minimise short circuit currents during the circuit design [85].

4.6 Layout design

The layout design of the AMULET3i multiplier uses a full-custom style for the datapath, where the circuit and layout of almost every transistor is optimized, and a standard cell style for the control logic, where the layout is automatically placed and routed using Compass Design Automation tools [86]. When the layout of a cell was complete, it was verified against the corresponding schematic (LVS) and then simulated using HSPICE.

The full-custom style is used in order to exploit the regularity of the datapath by designing only one "bit slice". The height of the bit slice in the datapath design is 82 λ for the AMULET3i multiplier. The number of tracks available for buses is ten per bit slice. Four tracks are for local routing and the other six for through buses. Data flow is routed horizontally in metal3, while control flow is relayed vertically in metal2. Both metal1 and metal2 are used for local interconnect in cells. The global power rails use metal1 and metal3, and the local power rails use metal2.

The overall height and width of the standard cells for AMULET2e are 112 λ and a multiple of 8 λ, respectively. This means that the connectors of a cell must have an 8 λ spacing and a 4 λ horizontal margin to either side of a cell. By taking into account existing open vertical routing tracks inside the standard cells, the routing over cell algorithm helps to reduce the final chip size.

4.7 Evaluation

An evaluation of the AMULET3i multiplier in terms of performance, power consumption and silicon area is presented in this section.

4.7.1 Performance

The critical path in the first pipeline stage includes one Booth mux cell, one 4-2 Counter (with enable control) and one pipeline register and the critical delay is about 2.8 ns under worst-case conditions (*Vdd* = 3.3V, *Vss*=0.1V, slow-slow process corner, at 100 ˚C temperature). The critical path in the second pipeline stage includes one 4-2 Counter (without enable control), one partial product register and one multiplexer for the final result and the critical delay is about 2.6 ns under worst-case conditions.

The delays of the two pipeline stages are well matched. This results in a 300 MHz computational speed with a 20% engineering margin.

4.7.2 Power consumption

The estimation of power consumption is a difficult problem since it is a strong function of the inputs and their history. A rough estimate of power consumption is given based on some assumptions. It is highly unlikely that all data bits will change for every data value. Based on the assumptions that half the data bits on average will change and that the dynamic switching power is 90% of the total power, the power estimate of the datapath is about 40 and 82 mW operating at 100 and 200 MHz (under typical process conditions), respectively.

4.7.3 Silicon area

The silicon area of the datapath is 2082 $\lambda \times 3198\ \lambda$ ($416.4 \times 639.6\ \mu m^2$). Figure 4-13 shows the physical layout of the datapath of the AMULET3i multiplier, and illustrates its regular structure.

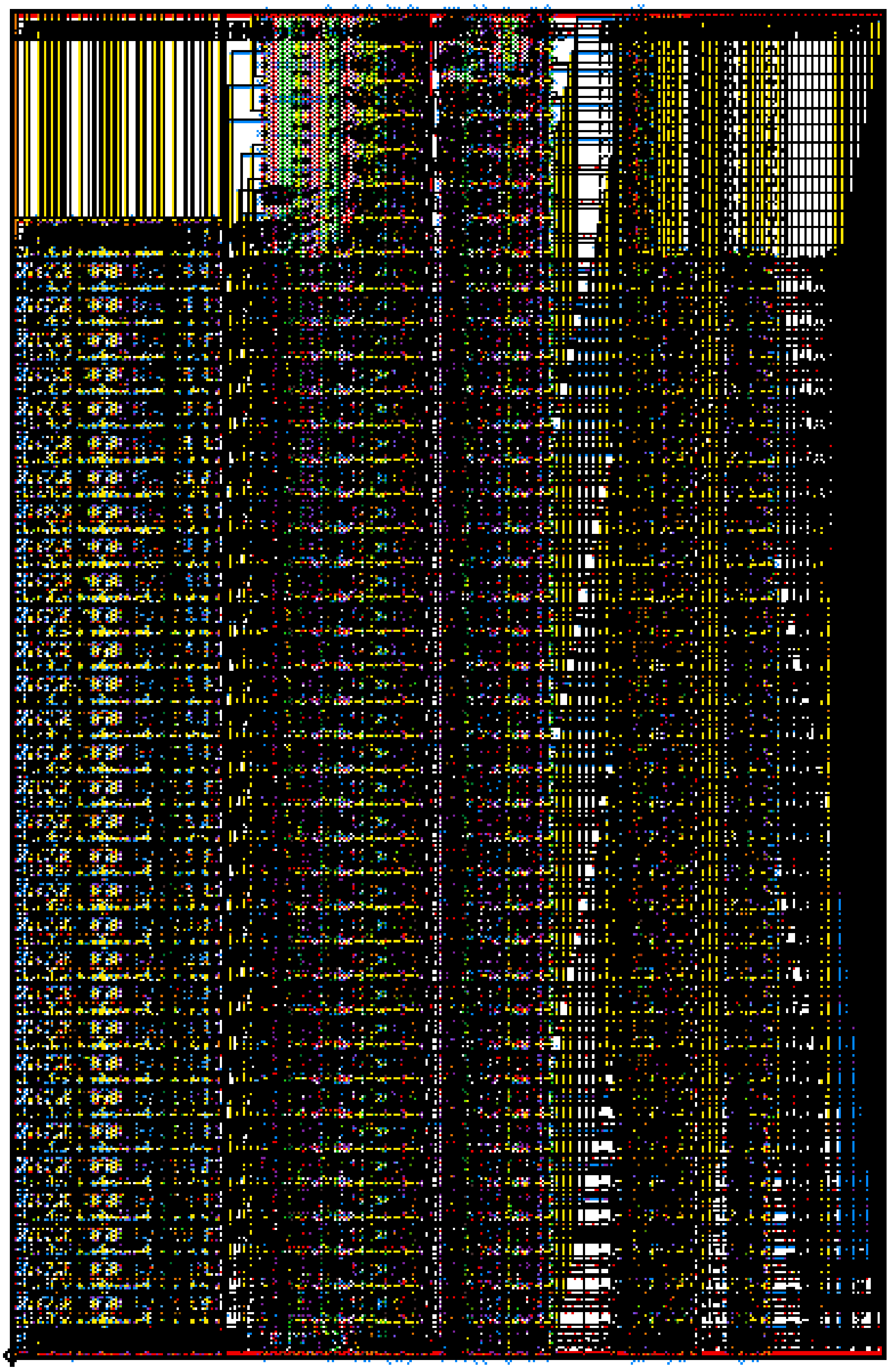


Figure 4-13: Physical layout of the multiplier datapath

4.8 Summary

A high performance, low power asynchronous 32 bit multiplier with a reasonable hardware resource has been developed for AMULET3i. The design uses the modified Booth's algorithm with 8 bits at a time with an iterative structure. An "early out" scheme is employed.

The pipeline registers are made initially transparent to avoid the data skew problem caused by introducing one pipeline stage. An new coding scheme is used to adjust the product result of an unsigned number multiplication. An adjustment value is made on the least significant 32-bit positions.

The true single-phase clocking methodology and pass-transistor logic style are chosen for circuit design. A new 4-2 counter circuit has been incorporated.

The AMULET3i multiplier presents a minimum hardware requirement given performance constraints and is designed for low power.

Post-layout simulation, in a 0.35 micron triple metal CMOS technology, shows that it takes 11.2 ns (2.8 ns $\times$ 4 cycles) to complete the computation of a 32-bit multiplication in the worst case. The power estimate of the datapath is about 40 and 82 mW operating at 100 and 200 MHz (under typical process conditions), respectively. The layout is regular and compact with a datapath area of only $416.4 \times 639.6\ \mu m^2$.

Taken individually, the characteristics above are not novel. What is new is the manner in which the AMULET3i multiplier has been designed to combine elegantly all these algorithm and circuit design techniques within an asynchronous framework.

Four-phase pipeline control 5

This chapter explores the design of four-phase control schemes for asynchronous pipelines. The study is focused mainly on the four-phase micropipeline design style which uses conventional level-sensitive data latches. Low power considerations and the use of dynamic logic are also discussed. All of the proposed pipeline latch control circuits are speed-independent, and this has been verified using the FORCAGE tool [21]. Simulation results in a 0.35 micron triple metal CMOS technology are presented.

5.1 Introduction

Micropipelines were introduced by Ivan Sutherland in his 1988 Turing Award lecture [20], and are a practical way to build asynchronous pipelines. Micropipelines are viewed as being composed of a control circuit employing the two-phase handshake protocol and a datapath using the bounded delay model.

The AMULET1 asynchronous processor, developed by Professor Steve Furber's AMULET group at the University of Manchester, used the two-phase micropipeline design techniques. However its successors, AMULET2e and AMULET3i, abandoned two-phase control in favour of four-phase control, mainly for performance reasons.

The four-phase micropipeline design space may be roughly categorized by viewing along three dimensions: the data-validity scheme, the logic activation configuration, and the decoupling degree. These three dimensions have the possible values of: **E**arly, **B**road or **L**ate; **R**equest-activate or **A**cknowledge-activate; **U**n-decoupled, **S**emi-decoupled or **F**ully-decoupled, respectively. A three-character shorthand notation can therefore be used to convey the category for a particular design. For example, the abbreviation **ERF** would signify a circuit which employs the **E**arly data-validity scheme, uses a **R**equest signal to activate combinational logic, and is **F**ully-decoupled.

5.2 Data-validity scheme

Figure 5-1 shows a general micropipeline stage structure. The latch control circuit communicates with neighbouring pipeline stages on both its input link (*Rin*, *Ain*) and its output link (*Rout*, *Aout*). The control link (*E*, *D*) connects with associated combinational logic. In addition to these three handshake links, a latch control wire (*Lt*) is needed to open and close the latch when low and high, respectively. The pipeline latches are configured as transparent when empty and we will return to this later.

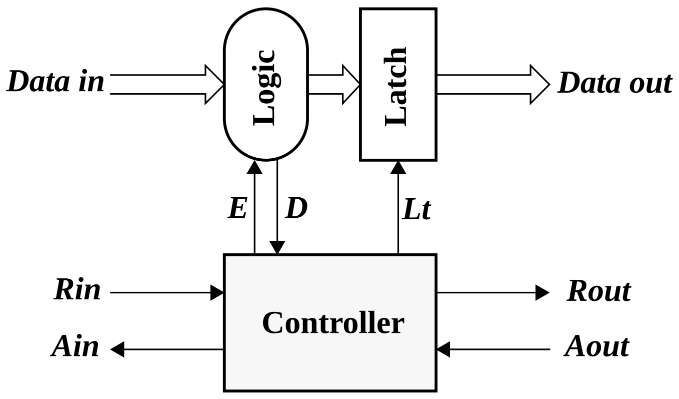


Figure 5-1: Micropipeline stage structure

The four-phase micropipeline design uses two successive handshakes for completing one communication process between neighbouring pipeline stages. There is a choice to be made as to which edge (rising or falling) of each handshake signal indicates the validity of data. This leaves us with three possible data-validity schemes, “early” [17,18], “broad” [19] or “late”, which are depicted in figure 5-2. It is worth noting that all these schemes take the micropipeline view that the sender of the data initiates the transfer.

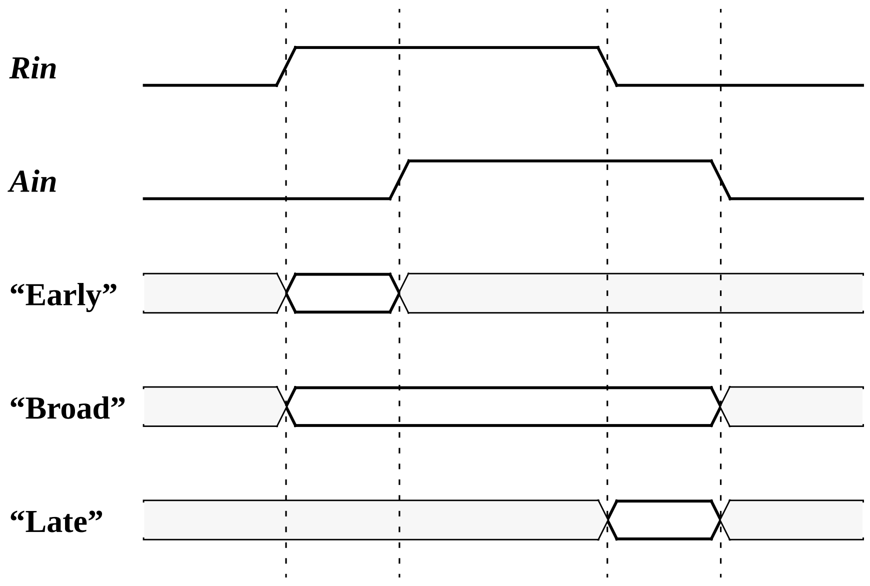


Figure 5-2: Three data-validity schemes

Initially, the *Rin* and *Ain* wires are both low. The “early” data-validity scheme uses the rising edge of the *Rin* wire to indicate “data available” and the rising edge of the *Ain* wire to indicate “data latched”. Then the *Rin* wire is returned low, whereafter the *Ain* wire is also returned low. The first handshake from *Rin* high to *Ain* high is called the “processing” or “evaluation” phase, during which the data remains valid. Data can change after the first handshake. The second handshake from *Rin* low to *Ain* low is called the “recovery” or “reset” phase, which is redundant and carries no meaning.

The “broad” data-validity scheme uses the rising edge of the *Rin* wire to indicate “data available” and the falling edge of the *Ain* wire to indicate “data latched”. Data must be guaranteed valid throughout two successive handshake processes. No “evaluation” or “reset” phases are distinguished.

The “late” data-validity scheme uses the falling edge of the *Rin* wire to indicate “data available” and the falling edge of the *Ain* wire to indicate “data latched”. The first handshake from *Rin* high to *Ain* high is called the “preset” phase, which is redundant and carries no meaning. The second handshake from *Rin* low to *Ain* low is called “processing” or “evaluation” phase, during which the data remains valid. Since the “late” data-validity scheme is rarely used, we focus only on the “early” and “broad” data-validity schemes and omit further consideration of the “late” scheme in this thesis.

5.3 Logic activation configuration

The rising edge of *Rin*, which indicates “data available” in both “early” and “broad” data-validity schemes, is usually used to activate combinational logic. This common arrangement is referred to as a “request-activate” configuration as shown in figure 5-3.

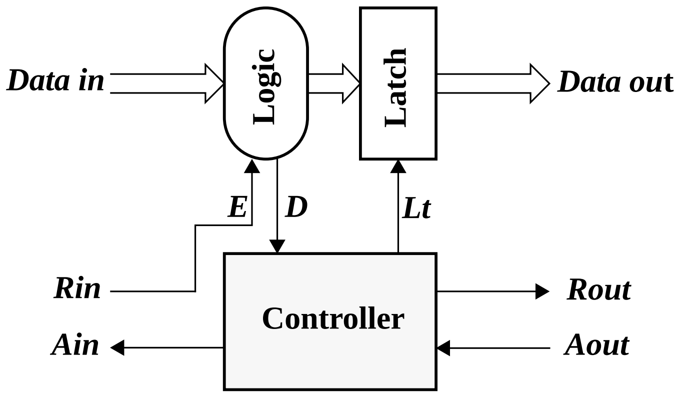


Figure 5-3: “Request-activate” configuration

Instead of using the *Rin* wire, the "broad" data-validity scheme has the choice of using the *Ain* wire to activate combinational logic as the data remains valid during the whole handshaking process. This new arrangement is referred to as an "acknowledge-activate" configuration as shown in figure 5-4, and provides an efficient framework for low power design using dynamic logic (see section 5.13).

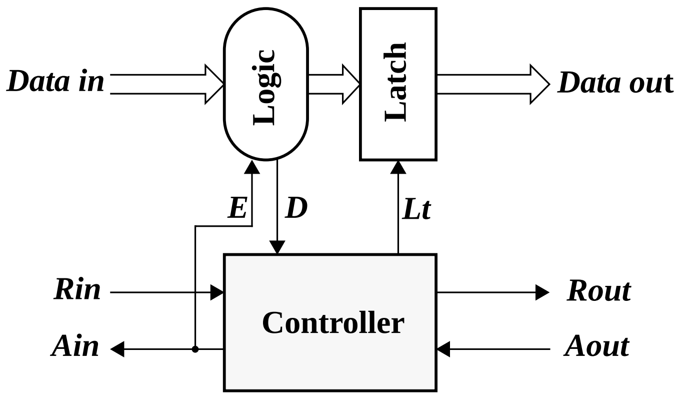


Figure 5-4: "Acknowledge-activate" configuration

5.4 Decoupling degree

Conceptually, the decoupling degree is used to describe how the input link interacts with the output link. For the sake of discussion, three terms are defined here. The first handshake is called "initiated" and the second handshake "completed". "Suspended" is between "initiated" and "completed".

A micropipeline stage is said to be un-decoupled if it satisfies the following two conditions: (1) a new communication coming along its input link cannot be "initiated" until the current communication going along its output link has been "completed", (2) and it is "suspended" if the new communication along its output link has not been "initiated". A micropipeline stage becomes semi-decoupled by getting rid of the first condition, and it becomes fully-decoupled by also removing the second condition. A new communication along the input link of a fully-decoupled latch control circuit may be "completed" before the new communication along its output link has been "initiated".

5.5 ERU latch control circuit

The specification of the latch control circuit is described using a Signal Transition Graph (STG) which shows the causal relationships between the signal transitions. An STG for an ERU latch control circuit is shown in figure 5-5. The dashed arrows indicate dependencies that the environment (usually the neighbouring stages) must observe and the solid arrows represent internal orderings; both must be maintained to ensure that the corresponding circuit is speed-independent. The "tokens" drawn next to certain arcs represent an initial "marking". A particular transition can fire only when there is a token on each of its input arcs and a token is placed on each of its output arcs after it fires.

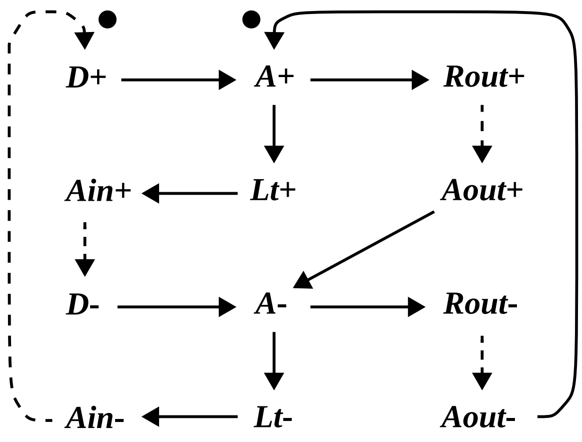


Figure 5-5: STG of the ERU latch control circuit

The state graph may be derived from the STG and then an implementation from the state graph, but in this simple case it may be seen by inspection that the circuit in figure 5-6 is an implementation of the STG in figure 5-5. There should be one closure and one opening of the latch before one communication has been “completed” for this latch control circuit. Thus the latch can only be closed when the next stage latch is open since *Aout* must be low (the next latch is open) before *Lt* can go high. In the case when data is inserted into the pipeline at a greater rate than it is removed from the pipeline, the pipeline will eventually fill. A full micropipeline has alternate closed and open latches (and therefore only alternate stages can be occupied), similar to master-slave latches in synchronous designs. This effectively halves the asynchronous pipeline depth. Therefore this design is not of practical interest, and it is used here only as a starting point.

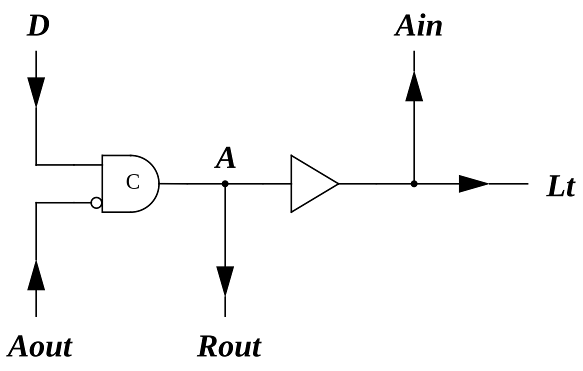


Figure 5-6: ERU latch control circuit

5.6 ERS latch control circuit

An STG specification for an ERS latch control circuit [18] is shown in figure 5-7. It is worth noting that an internal variable (*A*) is introduced on purpose. The variable (*A*) is used to record when the input link is ready to proceed. It is expected that there will be dozens of latched data and a buffer is to be needed to maintain reasonable drive strength. This buffer reflects the need for the latch to close before the input link is “initiated”. It could, perhaps, be argued that some delay should be built into the path from *D* to *Rout*.

However, there is no need for the latch to close before *Rout* is signalled so long as the data have propagated through the latch which is transparent when empty. This argument reflects, in fact, the constraint of the bounded delay model. Therefore the delay from *D* to *Rout* must be no shorter than the propagation delay through the latch for the correct operation of the circuit, which is almost always satisfied with confidence.

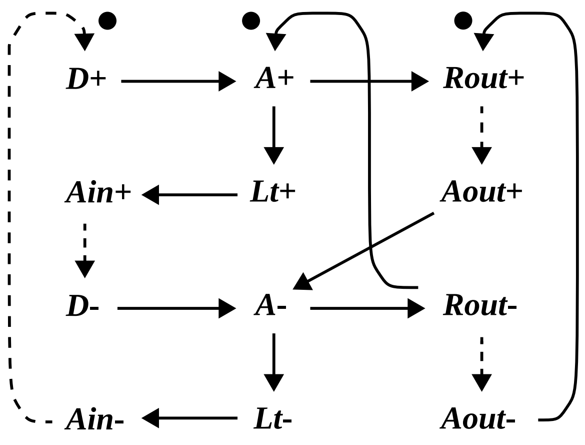


Figure 5-7: STG of the ERS latch control circuit

To obtain formally an implementation of an STG specification, the STG is first transformed into the state graph by applying the underlying Petri net rules to construct the reachability tree. The state graph should have the CSC (Complete State Coding) property, then logic equations for the output variables can be derived. Figure 5-8 shows an implementation of the ERS latch control circuit [18]. The notation used here for asymmetric C-gates follows that used in previous work [18]. 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 "-". This notation is illustrated in figure 5-9 which shows a possible transistor level implementation of an asymmetric C-gate.

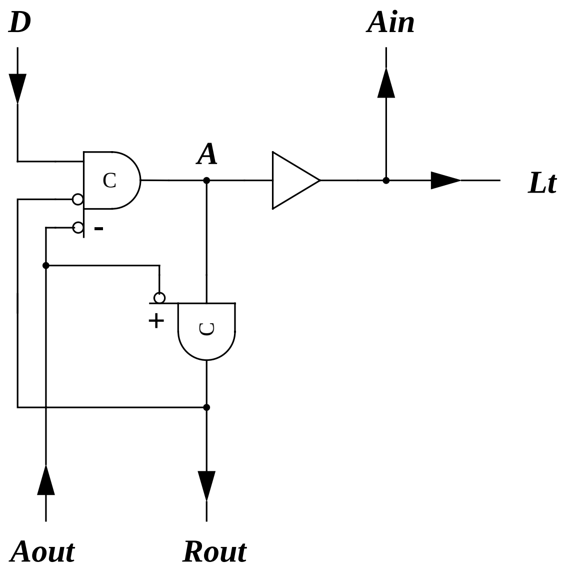


Figure 5-8: ERS latch control circuit

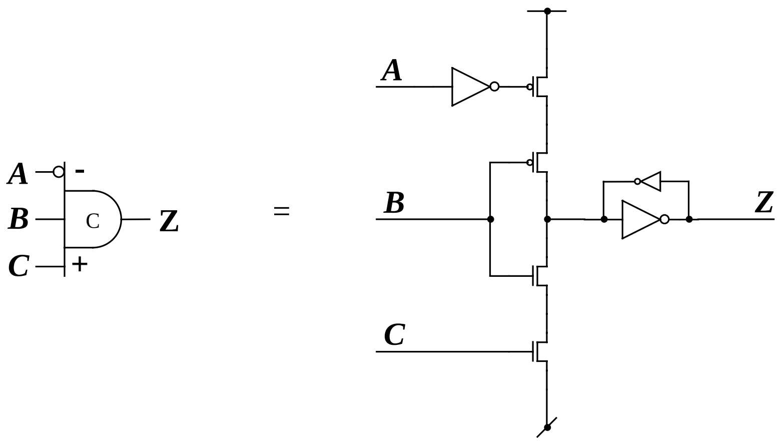


Figure 5-9: Asymmetric C-gate notation

With the ERS circuit a new communication on the input link can be “initiated” before the current communication on the output link has been “completed”, but it is “suspended” until the new communication on the output link has been “initiated”. This means that one communication should cover two “evaluation” processes and can therefore be performed in a time proportional to the sum of the two processing logic delays.

5.7 ERF latch control circuit

An STG specification for an ERF latch control circuit is shown in figure 5-10. Note that the buffer falling delay from *Lt* high to *Lt* low is removed from the input link path. This is significant since the buffer delay, especially in a wide datapath where the capacitive loading is large, has an adverse effect on the handshake delays.

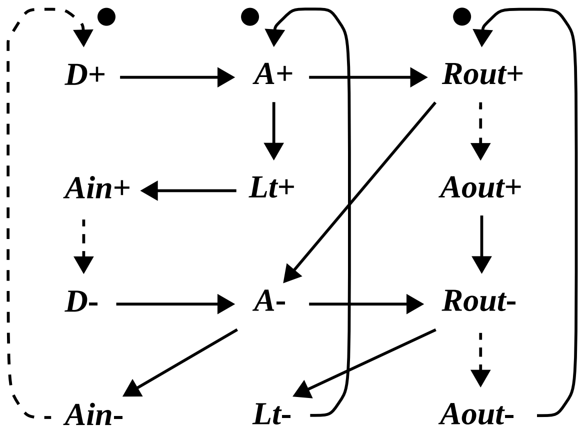


Figure 5-10: STG of the ERF latch control circuit

The ERF latch control circuit is shown in figure 5-11. A new communication along the input link can be "completed" before the new communication along the output link has been "initiated". The essence of a fully-decouped latch control circuit is to break the sequential operational dependency between its input side and its output side in order to allow them to run concurrently on either side. A clocked pipeline is, in some senses, fully-decouped, but it should use an edge-triggered as one pipeline stage to isolate its input flow from its output flow. It is obvious that asynchronous pipelines are more efficient in terms of the number of latches required, especially when a wide datapath or a deep pipeline is involved. It should be mentioned here that early asynchronous designs [87] used edge-triggered latches, simply following the practice of the clocked design.

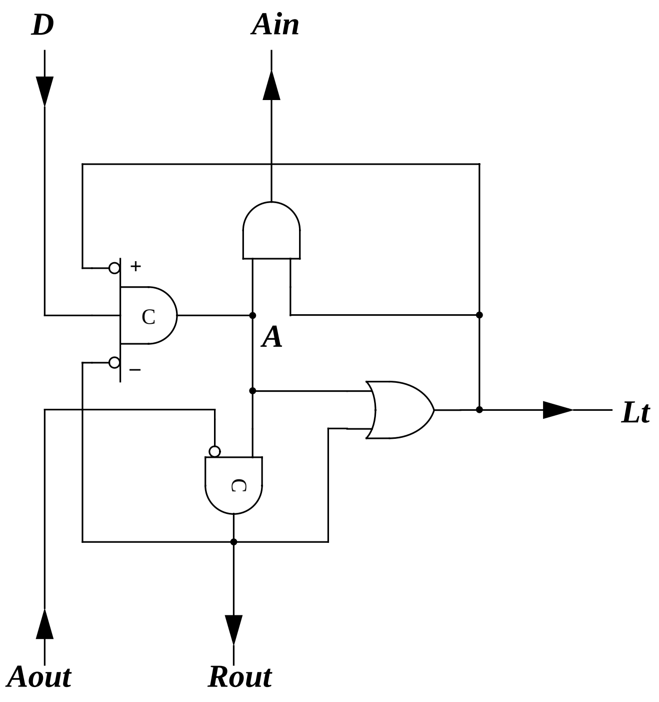


Figure 5-11: ERF latch control circuit

5.8 BRU latch control circuit

For the sake of comparison, an STG specification and implementation of a BRU latch control circuit are shown in figure 5-12 and figure 5-13, respectively.

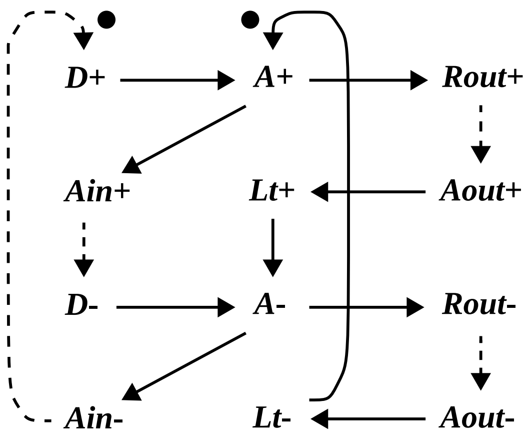


Figure 5-12: STG of the BRU latch control circuit

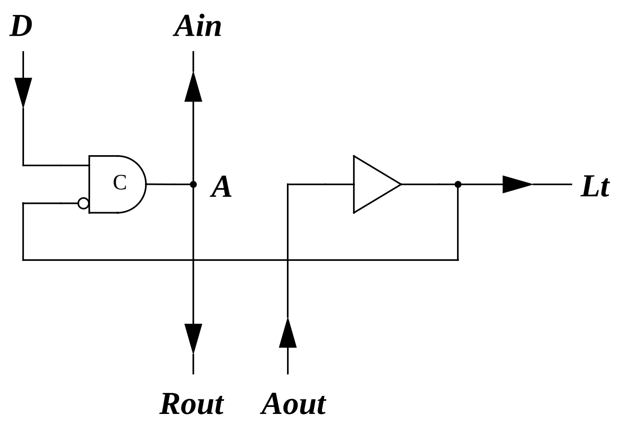


Figure 5-13: BRU latch control circuit

5.9 BRS latch control circuit

An STG specification and implementation of a BRS latch control circuit are shown in figure 5-14 and figure 5-15, respectively. The BRS latch control circuit is very similar to the ERS one. However, the buffer delay directly contributes to the input link delay in the ERS latch control circuit, whereas the buffer delay is "invisible" from the input link and moved into the output link in the BRS one.

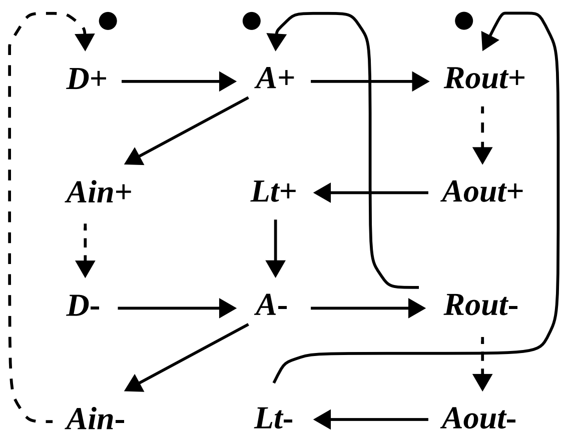


Figure 5-14: STG of the BRS latch control circuit

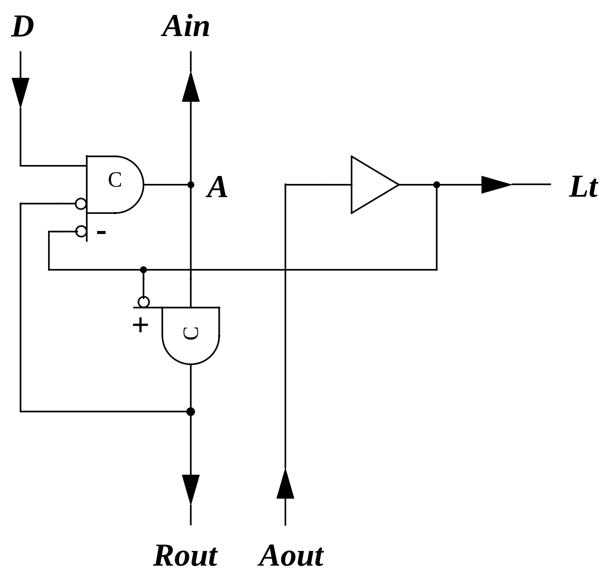


Figure 5-15: BRS latch control circuit

The BRS latch control circuit has the same drawback as the ERS one: the pipeline cycle time increases by twice the processing logic delay. It is of potential use only in FIFO applications.

5.10 BRF latch control circuit

An STG specification of a BRF latch control circuit is shown in figure 5-16. For the input link (*D*, *Ain*), the path from *Ain* low to *D* high is the critical arc since the evaluation process is by assumption much longer than internal handshake transitions. Similarly, for the output link (*Rout*, *Aout*), the path from *Rout* high to *Aout* high is the critical arc. By now, an intuitive feel for fully-decoupling is that operations on these two critical paths should not be dependent on each other. In other words, there is no simple loop that contains these two arcs in the STG specifications. By so doing, two neighbouring combinational logic functions can be performed in parallel at all times.

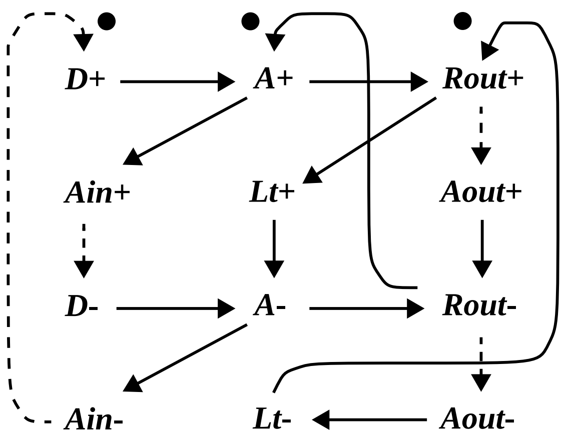


Figure 5-16: STG of the BRF latch control circuit

Figure 5-17 shows an implementation of the BRF latch control circuit [19]. The emphasis of asynchronous pipeline designs is on maximum allowable concurrency, which was kept in mind during the development of these latch control circuits. Only slight differences in STG specifications may lead to very different latch control circuits.

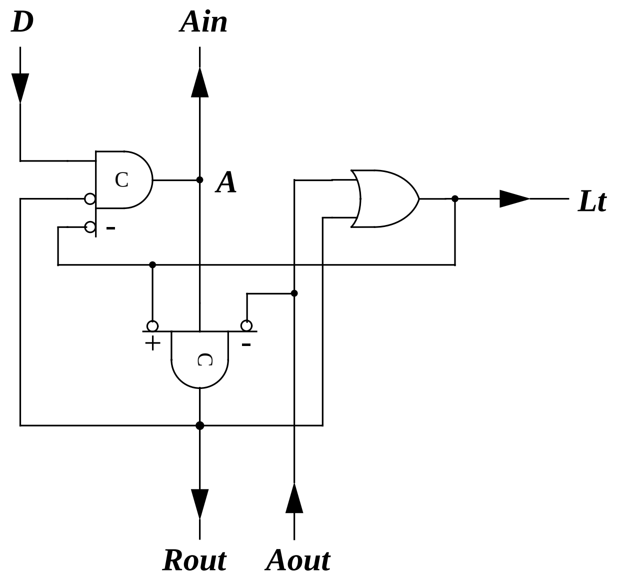


Figure 5-17: BRF latch control circuit

5.11 BAS & BAF latch control circuits

By now, we may sense the key difference between the "early" and "broad" data-validity schemes, which lies in the decision point on when to issue the acknowledge signal *Ain*. For the "early" data-validity schemes, only after the data has been latched is the acknowledge *Ain* issued. However for the "broad" data-validity schemes, the acknowledge *Ain* can be issued before the data has been latched. The key idea of the "broad" data-validity scheme is to make the first handshake as fast as possible and the associated combinational logic is sidelined from the pipeline (see figure 5-4). The request signal *Rin* is no longer entitled to activate the combinational logic since it may return low independently of whether the evaluation phase is complete or not. Instead, the acknowledge *Ain* can take the job. It could, perhaps, be argued that the point of activation of the combinational logic has been delayed and the performance will suffer. However, firstly, the delay is marginal since the first handshake is fast. Secondly, if it is still an issue, another arrangement can be made as shown in figure 5-18.

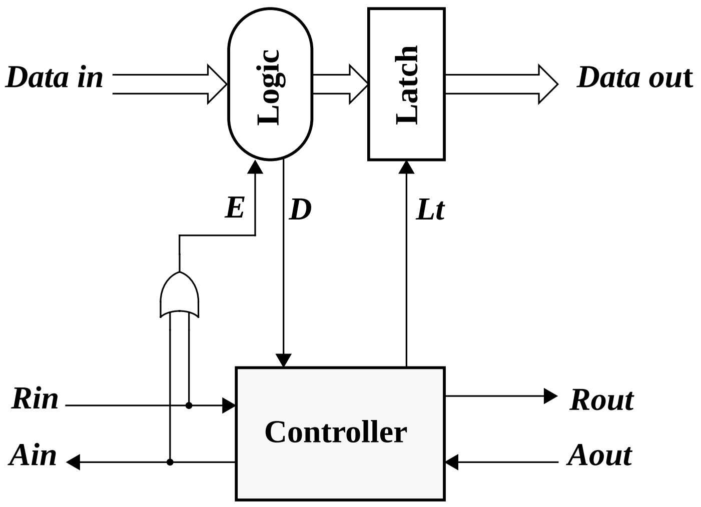


Figure 5-18: Another "Acknowledge-activate" configuration

STG specifications for a BAS and a BAF latch control circuit are shown in figure 5-19 and figure 5-20, respectively. Implementations of a BAS and a BAF latch control circuit are shown in figure 5-21 and figure 5-22, respectively. These two latch control circuits are almost the same as their request-activate counterparts but have an extra input. They can be used to exploit the advantage of dynamic logic for low power designs as we will

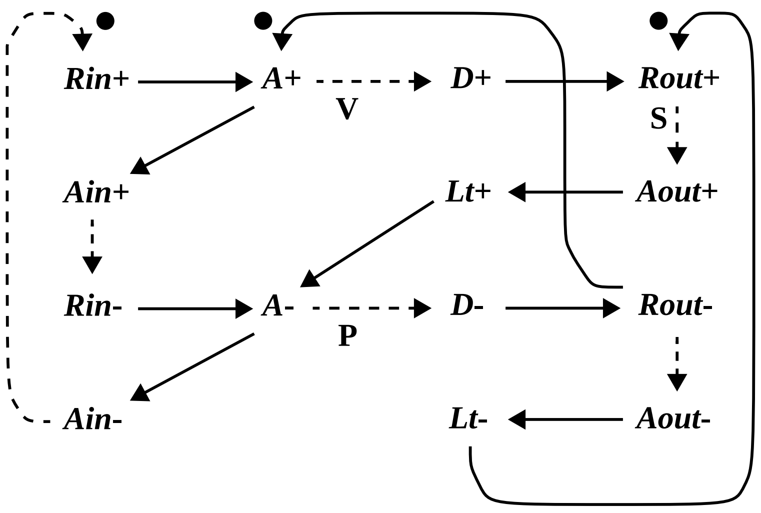


Figure 5-19: STG of the BAS latch control circuit

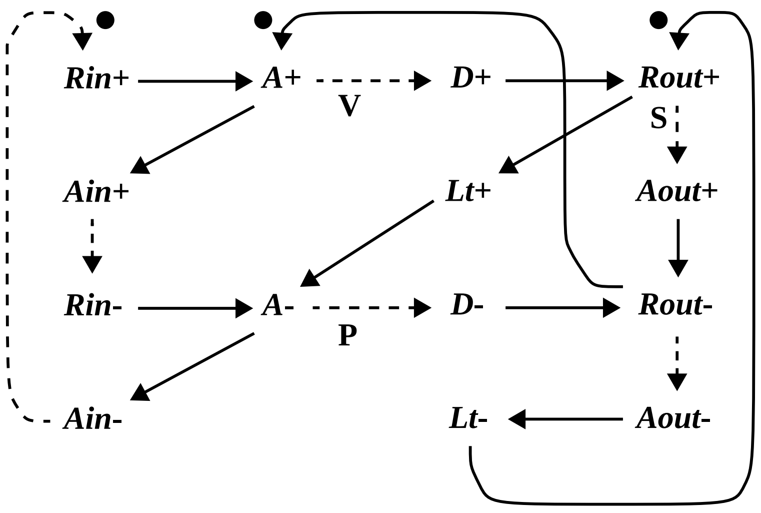


Figure 5-20: STG of the BAF latch control circuit

discuss in the next section. It should be noted here that up to now all the combinational circuits presented earlier are assumed to be static by default. Some effort must be made before dynamic circuits can be used.

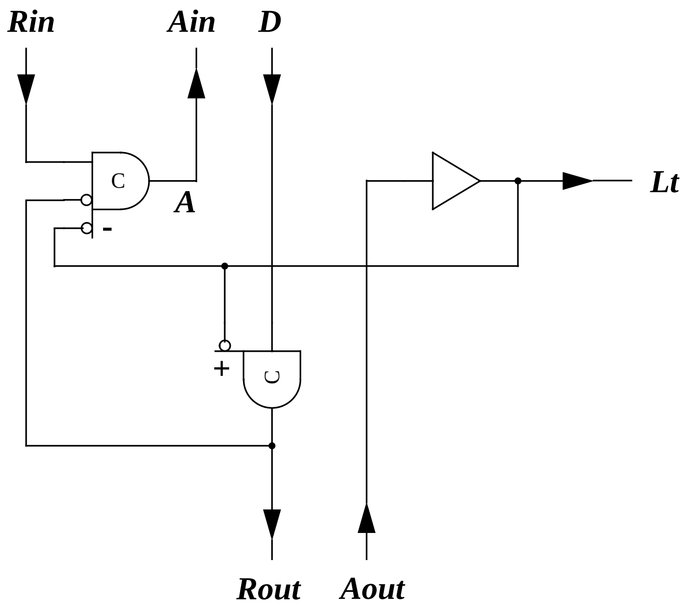


Figure 5-21: BAS latch control circuit

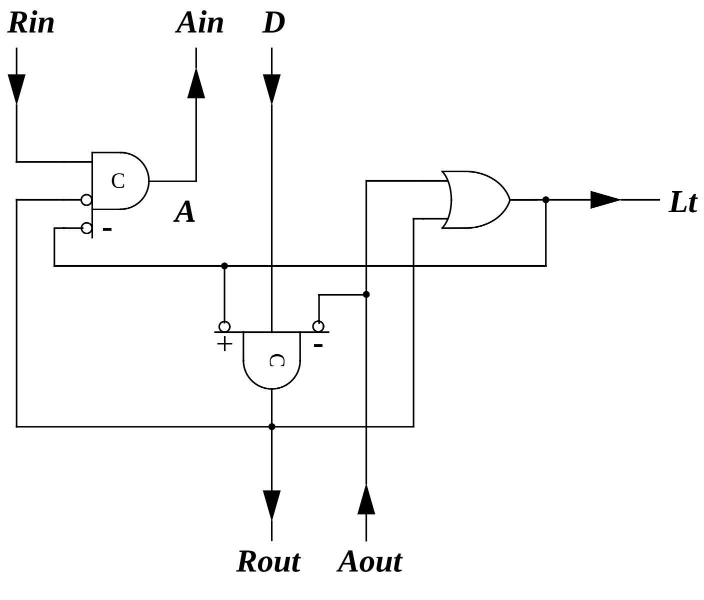


Figure 5-22: BAF latch control circuit

5.12 Interfacing

There are occasions where it may be desirable to use both “early” and “broad” latch control circuits. For example, the BAS or BAF latch control circuit for low power designs using dynamic logic should be used together with other latch control circuits to ensure that the end condition is satisfied. (see section 5.13).

To interface a “broad” latch control circuit into an “early” latch controller would appear to be rather straightforward, since the “broad” scheme is more than sufficient to cover the input specification of the “early” scheme. However there must be a converter when interfacing an “early” latch control circuit into a “broad” one. An STG specification and implementation of a converter are shown in figure 5-23 and figure 5-24, respectively.

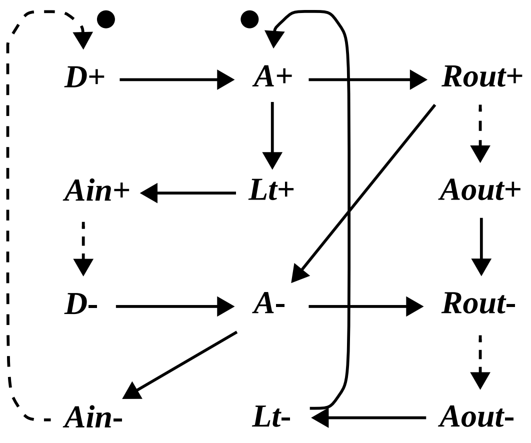


Figure 5-23: STG of the Converter

It should be noted that a broad latch control circuit can be used for cases where the early protocol is used. However, the operation of the circuit is totally sequential, which is undesirable from the performance perspective. Therefore appropriate latch control circuits should be used for particular application cases.

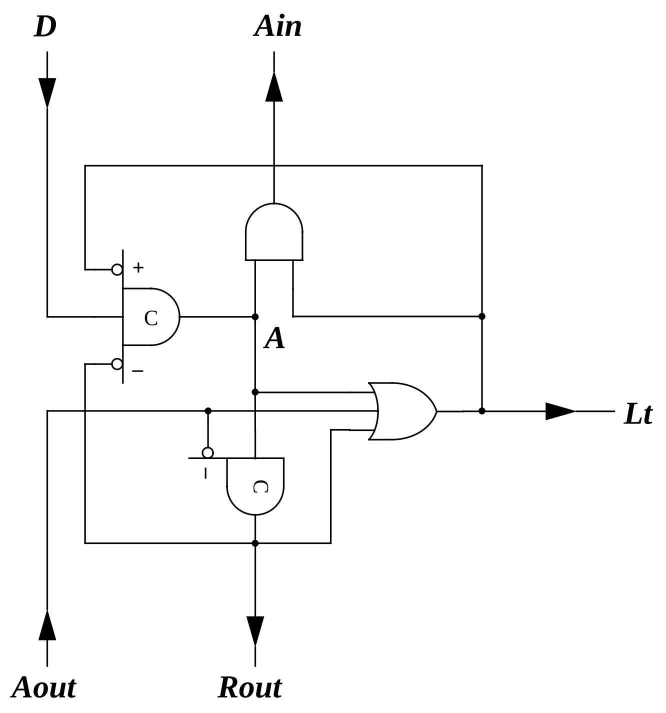


Figure 5-24: Converter circuit

With the Converter circuit a new communication along the input link is not subject to being blocked and will be completed as long as it has been initiated. This property is useful to ensure the end condition that we will discuss in the next section.

5.13 Low-power design using dynamic logic

The micropipeline design style configures the pipeline latches as transparent when empty. The motivation for this comes from both performance and testability. First, transparent latches steer the inputs directly to the outputs, thus reducing the latency of the pipeline. Secondly, they make the datapath have a combinational behaviour in its initial state, offering good testability of the datapath logic. However, this comes at a price. Data and glitches can be broadcast down the pipelines, thus wasting power.

Dynamic circuits can be used to localise the data flow to solve the above problem [88]. The obstruction of data flow is achieved since the dynamic logic is held during the precharged phase. Additionally, dynamic circuits offer the benefits of increased speed and lower switched capacitance. Therefore low power designs often employ dynamic logic, especially in the datapath design.

However, there is a difficulty in directly using dynamic circuits in asynchronous designs since the asynchronous control can stall in any state for any time. Leakage currents cause the output of dynamic circuits to be valid for a short time; therefore evaluation cannot begin until the output latch is free. The inputs must also be held stable until evaluation is complete, so during evaluation both the input and the output latches are required by the intervening dynamic logic, resulting in at most *50%* of the logic being active at any time.

Although additional latches or charge-retention circuits can be used to make dynamic circuits pseudo-static, these additions increase the cost and power consumption of the dynamic circuits, thereby compromising their potential advantages.

The new idea introduced here is to observe that it is not strictly necessary for the output latch to be free before evaluation begins; it is only necessary to know that it will become free “soon”. Here “soon” is interpreted as any period which is not subject to arbitrary delay and is within the dynamic storage time of the output nodes. This relaxation of the evaluation start time allows a significant improvement in the pipeline’s performance.

The dynamic logic begins evaluation when its enable (*E*) goes high and it indicates a valid output on a “done” signal (*D*). When its enable is low it is precharged, and precharge completion is signalled by the “done” signal going low. (see figure 5-4).

For the BAS or BAF latch control circuit, the acknowledge wire *Ain* is indeed a confirmation signal which indicates that the output latch will be free “soon”. “Soon” is just the result of internal self-timed delays only, and is determined by the evaluate phase (*V*) and the precharge phase (*P*) together with a few internal control delays. Here the assumption is that the pipeline stage is connected to similar neighbours. We argue that a stall can only occur between *Rout* high and *Aout* high on the arrow marked *S* in figure 5-19 or figure 5-20. If this is true, the property is propagated back to the input, and hence, by induction, along a pipeline of similar stages. Only the end conditions remain to be checked. This condition is satisfied by using the Converter (see figure 5-24).

5.14 Simulation results

The latch control circuits have been laid out using 0.*35* micron triple metal CMOS technology and simulated using HSPICE operating at worst-case conditions (*Vdd* = 3.3V, *Vss* = 0.1V, slow-slow process corner, at 100 °C) and driving a 32 bit latch. The simulation results are shown in table 5-1.

Table 5-1: HSPICE simulation results

Parameter	ERS	ERF	BRS	BRF	BAS	BAF
FIFO Cycle Time	3.7 ns	4.4 ns	3.6 ns	4.0 ns	3.6 ns	4.0 ns
FIFO Response	8.6 ns	10.1 ns	8.0 ns	3.7 ns	8.0 ns	3.7 ns
Proc. Cycle Time	10.1 ns	7.7 ns	10.0 ns	7.1 ns	7.0 ns	7.2 ns
Proc. Response	18.5 ns	10.2 ns	17.5 ns	3.8 ns	8.9 ns	3.9 ns

A micropipeline with no processing in it is a FIFO and its cycle time gives an upper bound on the potential throughput. The response time is measured by stalling the output of a *3* stage pipeline until it is full, and then seeing how long it takes from releasing the stall until the input starts moving. The corresponding results for a micropipeline with processing in it are established by inserting combinational logic into the pipeline with an evaluation time of 3.0 ns and reset time of 0.3 ns. The test circuit is shown in figure 5-25.

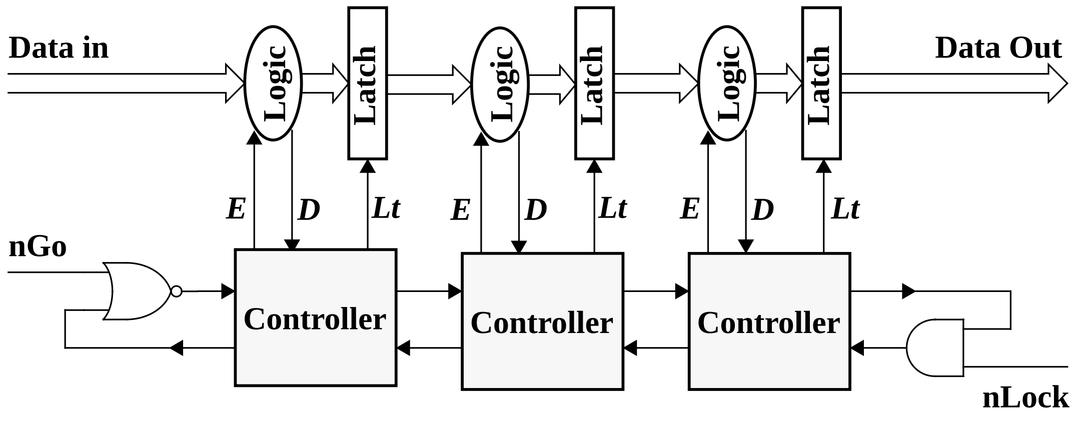


Figure 5-25: Test circuit

5.15 Discussion

The simulation results show that the cycle times of the ERS and BRS latch control circuits increase by approximately twice the processing delay, indicating both the processing delay on the input side and that on the output side are included. The cycle times of the other four latch control circuits just increase by the evaluation delay, indicating the processing delay on only one side is included. Here we now see how the different decoupling techniques have affected the resulting cycle times.

It is quite interesting that the BAS latch control circuit behaves in a “fully-decoupled” way. This is due to the fact that the point when the combinational logic begins evaluation has been moved in the acknowledge-activate configuration. This reflects the fact that the combinational logic is pulled out of the input link path and put aside. By so doing, the handshake process of the input link is in fact isolated from that of the output link. The change in the activation mechanism for the combinational logic makes the difference between semi-decoupled and fully-decouped behaviours.

The response times of the BRF and BAF latch control circuits is a lot smaller than the other four latch control circuits. The reason stems from the fact that when a confirmation signal (*Ain*) goes high this propagates very quickly backwards up the pipeline, allowing every pipeline stage to begin evaluation at almost the same time. For other latch control circuits, each pipeline stage must wait to clear the interlock before the initiating action is taken. Obviously, this is a very important factor in the performance of asynchronous pipelines which has unfortunately been ignored in the past. The response time relates to how fast a bubble [88] travels back up a pipeline. The detailed analysis of bubbles making self-timed pipelines fast can be found in [89].

It seems that the BRS and BAS latch control circuits will give the best performance in FIFO applications. However, it takes a long time to start moving after the full pipeline is released. Therefore, the BRF and BAF latch control circuits are suitable for both FIFO applications and pipelines including processing logic.

It is clear that the circuits using the broad protocol give better performance than those employing the early protocol. Among latch control circuits described above, the BRF and BAF latch control circuits are the best choice.

The BAF latch control circuits can be used to exploit the advantages of dynamic logic for low power designs. However, the end condition (a stall can only occur between *Rout* high to *Aout* high) must be met. For the BAF latch control circuit, this condition can easily be met by using the Converter circuit (see section 5.12).

All of the latch control circuit in this chapter are speed-independent, and were verified using the FORCAGE tool.

5.16 Summary

The design of control schemes for asynchronous pipelines has been studied. The study focused mainly on the four-phase micropipeline design style which uses conventional level-sensitive data latches. A set of speed-independent latch control circuits has been presented. Verification was carried out using the FORCAGE tool.

The BRF and BAF latch control circuits are the best choice for both FIFO applications and pipelines including processing logic. The ERF, BRF, BAS and BAF latch control circuits behave in the “fully-decoupled” way, where the cycle time increases by just one evaluation time. The BRF and BAF latch control circuits give the good response time.

The circuits using the broad protocol give better performance than those employing the early protocol. The acknowledge-activation configuration allows dynamic logic to be easily exploited for low power design. Dynamic logic retains externally static behaviour without additional latches or charge-retention circuits (allowing activity to cease without loss of state), and hence power can be saved.

Four-phase control modules 6

This chapter presents a set of control modules for four-phase micropipelines. Arbiters, which are non-trivial and tricky to design, are also included. These control modules, together with the pipeline latch control circuits described in the previous chapter, can be used to construct complex and powerful asynchronous systems including forking or joining multiple micropipelines. All of the proposed four-phase control modules are speed-independent, and this has been verified using the PETRIFY tool [23-26].

6.1 Introduction

In order to build asynchronous systems based on four-phase micropipelines, a set of basic control modules is required. Such a set is proposed here and shown in figure 6-1. The first element is the **CALL** module, which enables two processes to share a common resource. The two calling requests must be mutually exclusive. If they are not, the **ARBITER** module must be used instead. It is worth emphasizing that unlike in the synchronous case, an asynchronous arbiter always operates correctly. The **JOIN** and **FORK** modules are used to join and fork multiple control flows or pipelines, respectively. The **SELECT** module comes with two versions: one with a control link and one with a Boolean guard. The input Boolean guard must be prepared prior to the incoming handshakes on the input link and must remain stable during the handshaking process (restricted and guaranteed by the environment). The SELECT module steers incoming input handshakes to one of two outputs, depending on the handshake result along the control link or the Boolean value. The **TOGGLE** module steers incoming four-phase handshakes to alternate outputs. All of these four-phase control modules are speed-independent, and this has been verified using the PETRIFY tool.

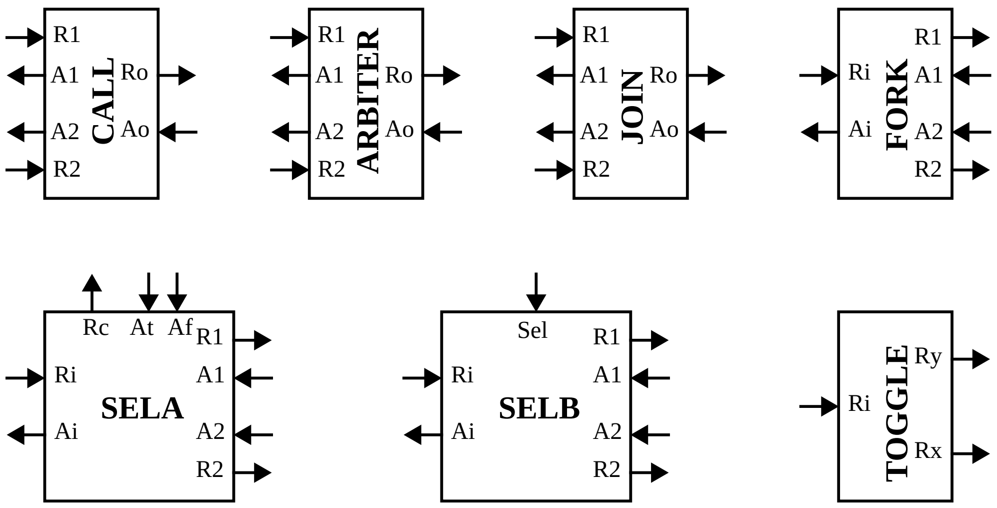


Figure 6-1: Four-phase control modules

This set of control modules provides the basic building blocks, which can be used to construct other control modules and asynchronous systems. The circuit implementations presented here are not claimed to be optimal. It should be appreciated that optimizations can be made if input constraints (determined by the environment) are known a priori to designers. A CALL module is an example, where it is known that the two input requests are mutually exclusive as a result of the environmental constraints. An ARBITER module is more general as its input changes are unrestricted. However, the circuit implementation of a CALL module is much simpler than that of an ARBITER module.

The specifications of these control modules are described using Petri Nets (PN) [22]. The PETRIFY tool then takes and manipulates this initial specification. It either generates another PN which is simpler than the original description or synthesizes an optimized speed-independent asynchronous circuit. The original specification may not satisfy the requirement of Complete State Coding (CSC) [90] and may lead to different states with the same binary value when encoding. To resolve this state coding conflict the PETRIFY tool automatically inserts a new state signal. The rising and falling transitions of this new state signal are inserted in such way that the synthesized circuit is optimized according to a selected cost function.

6.2 CALL modules

The CALL module serves the role of the procedure call in software where a common subroutine is shared. This section describes three types of CALL module: pCALL, dCALL, and bCALL. The first two CALL modules use the four-phase early protocol, while the last employs the four-phase broad protocol. The whole four-phase handshaking process on one input link must be completed before the next process on the other input link starts. Otherwise, the circuit will operate improperly.

6.2.1 pCALL module

A specification and implementation for a CALL module, called pCALL, are shown in figure 6-2 and figure 6-3, respectively. The pCALL module allows concurrent processing on the input link and resetting on the output link. However, the input and output links are not allowed to reset in parallel, with the input link being first reset and the output link following.

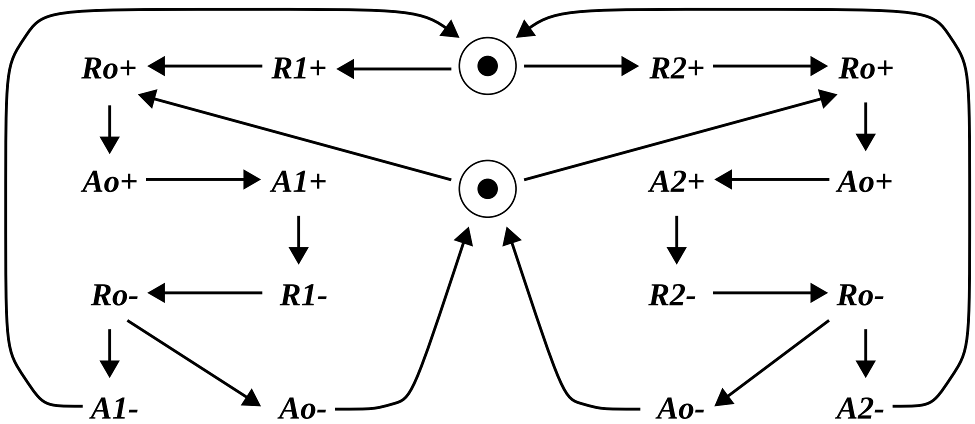


Figure 6-2: PN of the pCALL module

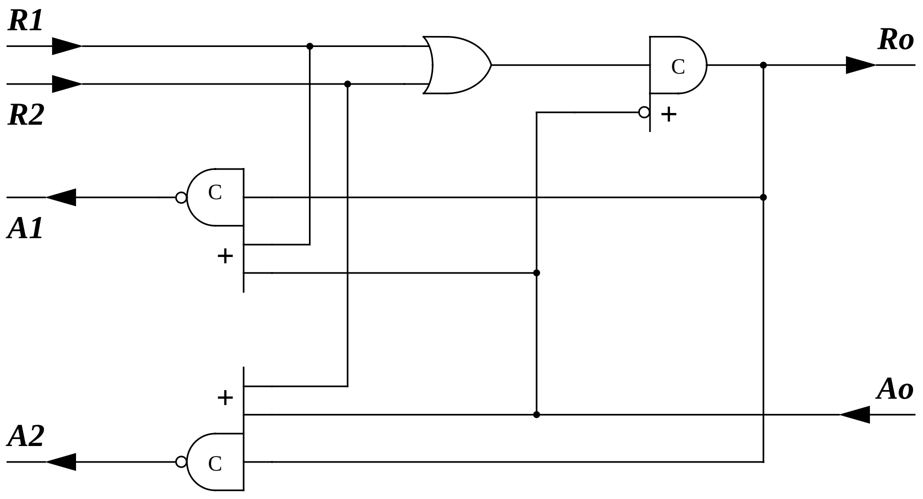


Figure 6-3: pCALL circuit implementation

6.2.2 dCALL module

A specification and implementation for a CALL module, called dCALL, are shown in figure 6-4 and figure 6-5, respectively. Like the pCALL module, the dCALL module allows concurrent processing on the input link and resetting on the output link.

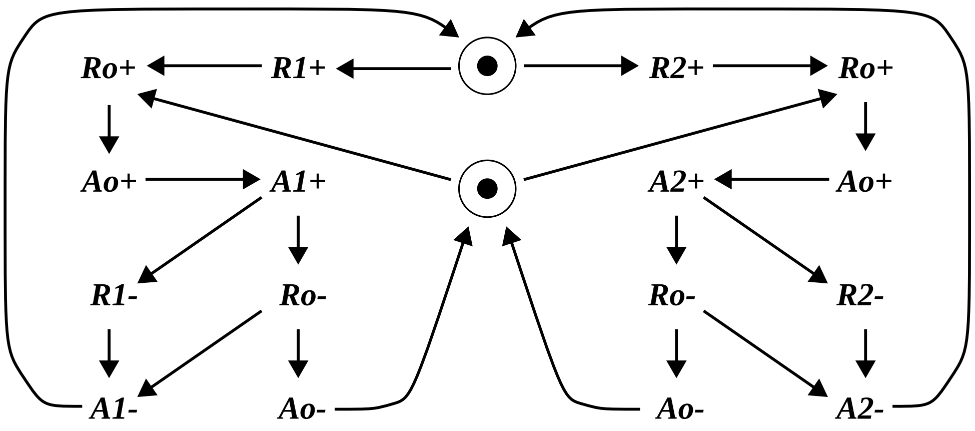


Figure 6-4: PN of the dCALL module

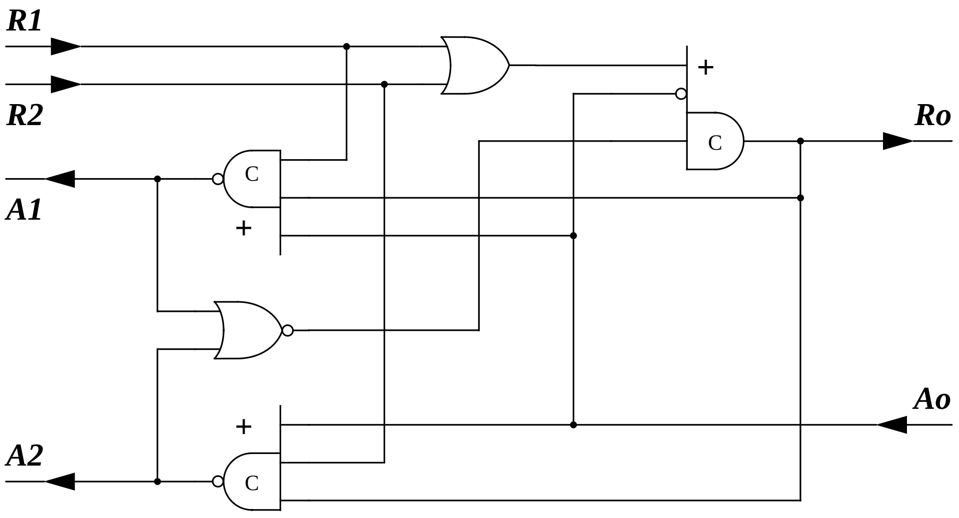


Figure 6-5: dCALL circuit implementation

Furthermore, concurrent resetting on both the input and output links are also allowed in the dCALL module. The resetting on the output link can start even before that on the input link. The output link has the property of self-resetting as soon as it has completed the calling procedure; resetting of the output link does not depend on an input reset request.

6.2.3 bCALL module

The first two CALL modules described above use the four-phase early protocol. There are occasions where it may be desirable to use the four-phase broad protocol, e.g., using dynamic logic for low-power design (see "Low-power design using dynamic logic" on page 115). A specification and implementation for a CALL module using the broad protocol, called bCALL, are shown in figure 6-6 and figure 6-7, respectively. The circuit is quite simple. It is worth noting that no processing or resetting phases are distinguished

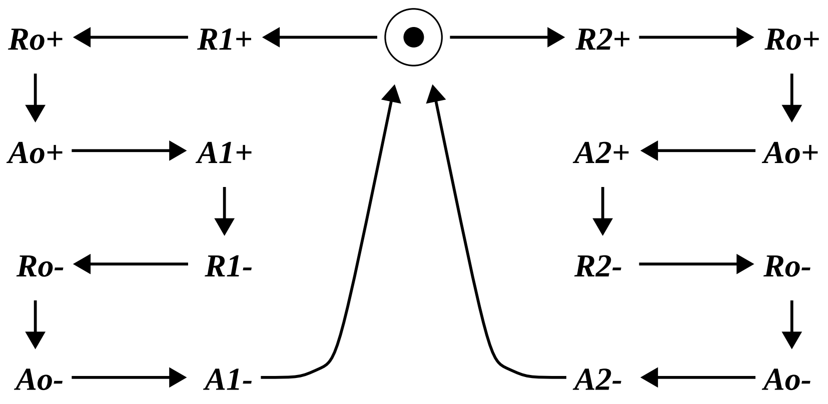


Figure 6-6: PN of the sCALL module

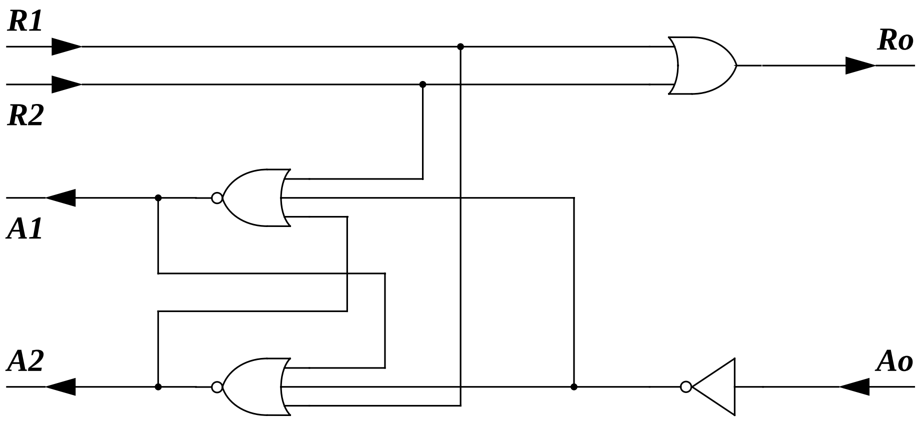


Figure 6-7: sCALL circuit implementation

in the broad protocol. The bCALL module can also be used for cases where the early protocol is used since the specification of the broad protocol is more than sufficient to cover that of the early protocol. However, the operation of the circuit is totally sequential, which is undesirable from the performance perspective.

6.3 ARBITER modules

The ARBITER module produces an exclusive grant to one of two asynchronous calling requests. As discussed in section 2.2.4 (see “Metastability and arbitration” on page 27), the ARBITER module is inherently prone to metastability. However, this metastable problem only affects the performance of the ARBITER module, not its functionality (only in the *asynchronous* case). Analog circuit techniques are used to keep the metastable states internal while maintaining valid logic levels at the interface. The mutual exclusion circuit (MUTEX) [31], as shown in figure 6-8, is such an analog circuit which makes a non-deterministic decision between two asynchronous requests. It comprises a cross-coupled NAND structure and a filter. The cross-coupled NAND structure may go metastable when the two inputs switch high at very nearly the same time. The filter conceals possible metastable states from the environment to maintain valid logic levels at the interface.

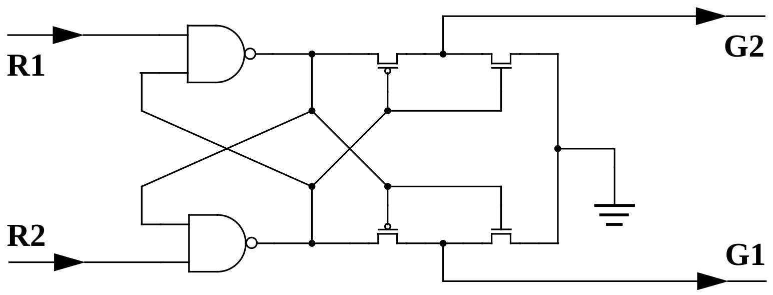


Figure 6-8: MUTEX circuit

This section describes three types of ARBITER module: pARBITER, dARBITER, and bARBITER. The first two ARBITER modules use the four-phase early protocol, while the last employs the four-phase broad protocol.

6.3.1 pARBITER module

A specification and implementation for an ARBITER module, called pARBITER, are shown in figure 6-9 and figure 6-10, respectively. The signals *G1* and *G2* are the outputs of the MUTEX element and internal signals of the pARBITER module. The two transitions (*Ro*+ $\longrightarrow$ *Ao*+) and (*Ro*- $\longrightarrow$ *Ao*-) are illustrated by the expressions (*Ro*+, *Ao*+) and (*Ro*-, *Ao*-), respectively, for the sake of brevity. As shown in [44], logic synthesis can produce speed-independent implementations only for specifications without conflicts on non-input signals. However, there is a conflict between the signals *G1* and *G2* in this specification and these two signals are internal (non-input) signals. We can get around this difficulty by treating the signals *G1* and *G2* as additional inputs [44] whose changes are restricted by the MUTEX element. The MUTEX element is considered to be part of the environment for the pARBITER module. This design trick is not restricted to conflicts on non-input signals and can also be applied to no-conflict cases. Well-defined modules can be treated in the same way as the MUTEX element and their outputs (also internal signals for a specification to be synthesized) are considered as additional inputs. By so doing, efficient implementations can be derived for some cases which otherwise may be difficult to synthesize.

The pARBITER module allows concurrent processing on the input link and resetting on the output link. However, the input and output links are not allowed to reset in parallel, with the input link being first reset and the output link following. Note that the signals *G1* and *G2* are often used to control a multiplexer to select the input data.

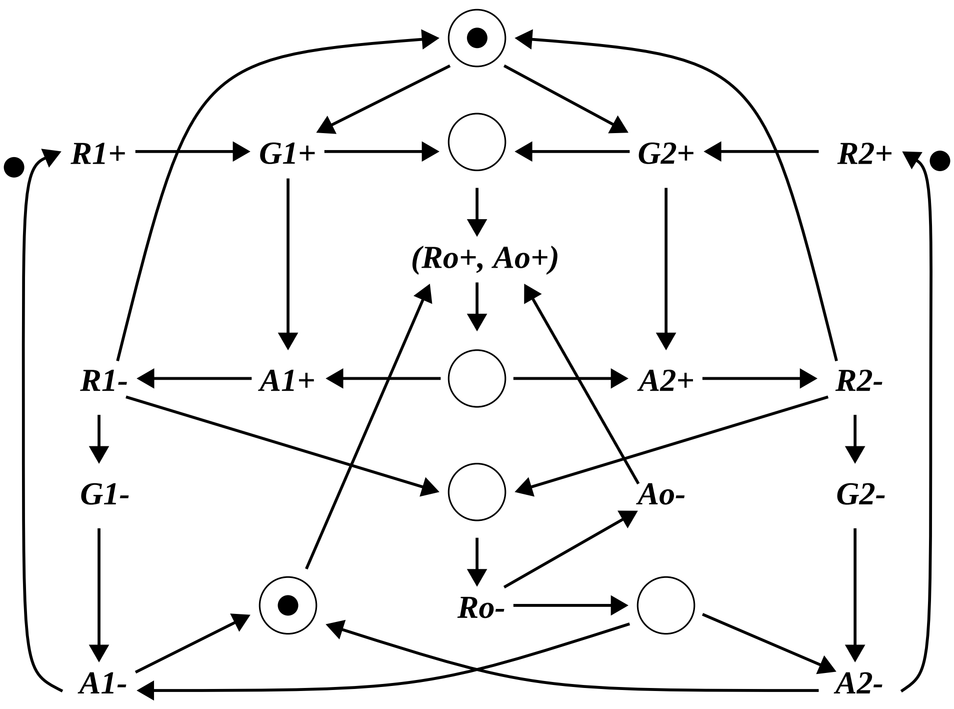


Figure 6-9: PN of the pARBITER module

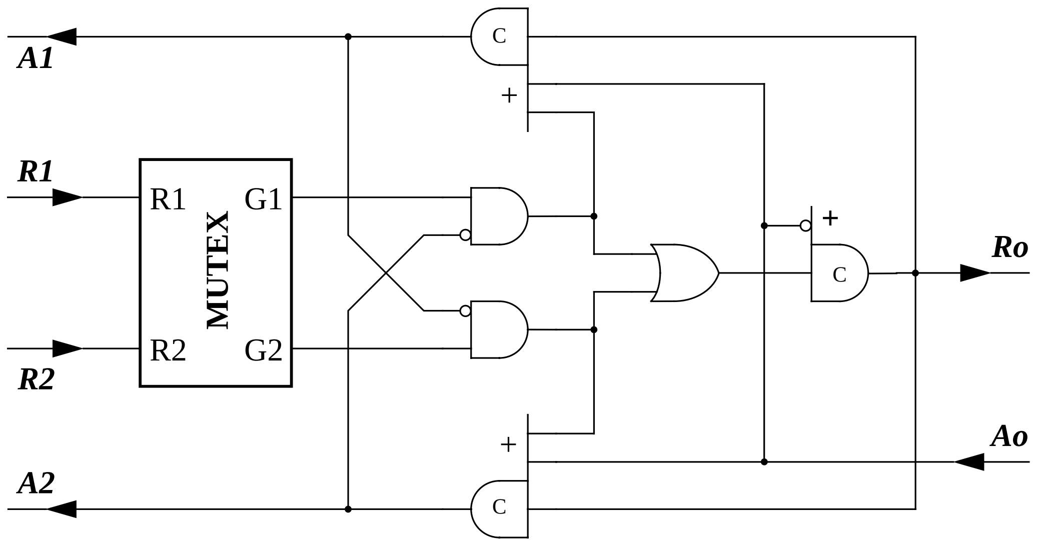


Figure 6-10: pARBITER circuit implementation

6.3.2 dARBITER module

A specification and implementation for an ARBITER module, called dARBITER, are shown in figure 6-11 and figure 6-12, respectively. Like the pARBITER module, the dARBITER module allows concurrent processing on the input link and resetting on the output link. Furthermore, concurrent resetting on both the input and output links are also allowed in the dARBITER module. The resetting on the output link can start even before that on the input link. The output link has the property of self-resetting as soon as it has completed the calling procedure; resetting of the output link does not depend on an input reset request.

Specifications with more concurrent operations lead, in general, to complex circuit implementations. This can bee seen from the development of the circuits above.

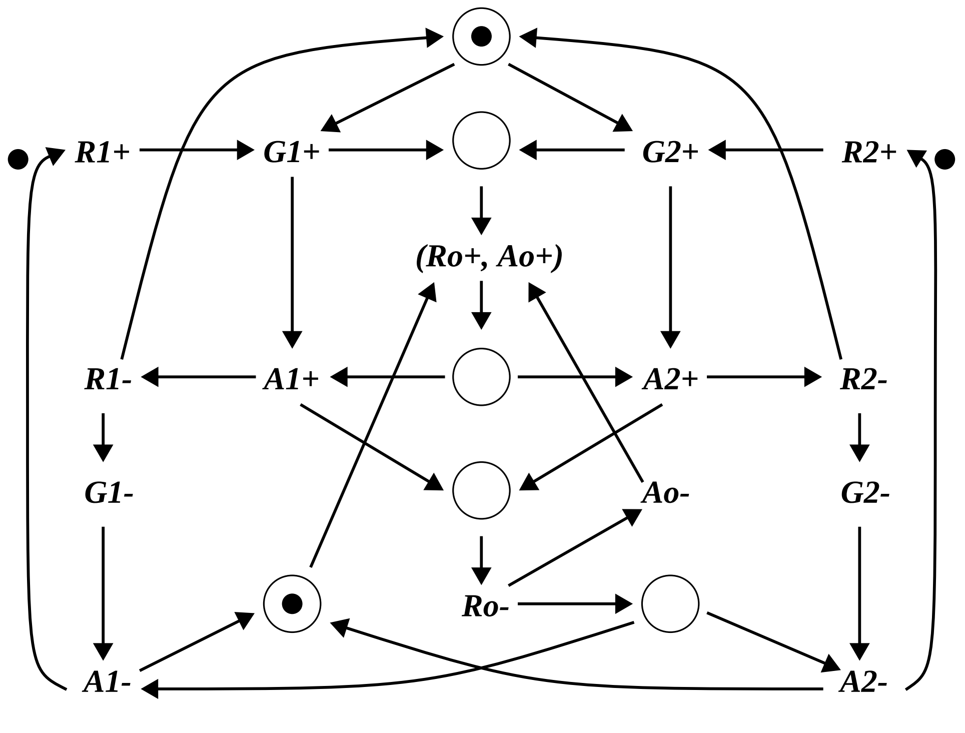


Figure 6-11: PN of the dARBITER module

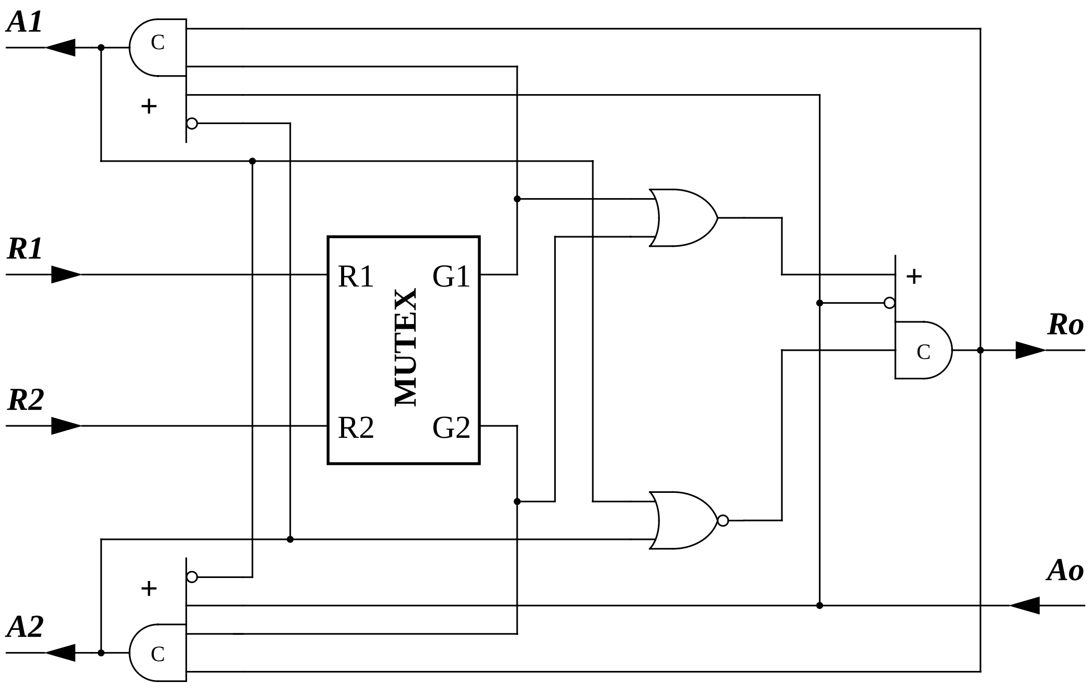


Figure 6-12: dARBITER circuit implementation

6.3.3 bARBITER module

The first two ARBITER modules described above use the four-phase early protocol. A specification and implementation for an ARBITER module using the broad protocol, called bARBITER, are shown in figure 6-13 and figure 6-14, respectively. The bARBITER module can also be used for cases where the early protocol is used since the specification of the broad protocol is more than sufficient to cover that of the early protocol. However, the operation of the circuit is totally sequential, which is undesirable from the performance perspective.

Generally, specifications using the broad protocol, e.g. the bARBITER, often have simpler circuit implementations than those using the early protocol.

All the ARBITER modules described above are fair arbiters [91], which means that a pending request on one input link must be granted after the granted request on the other input link has completed.

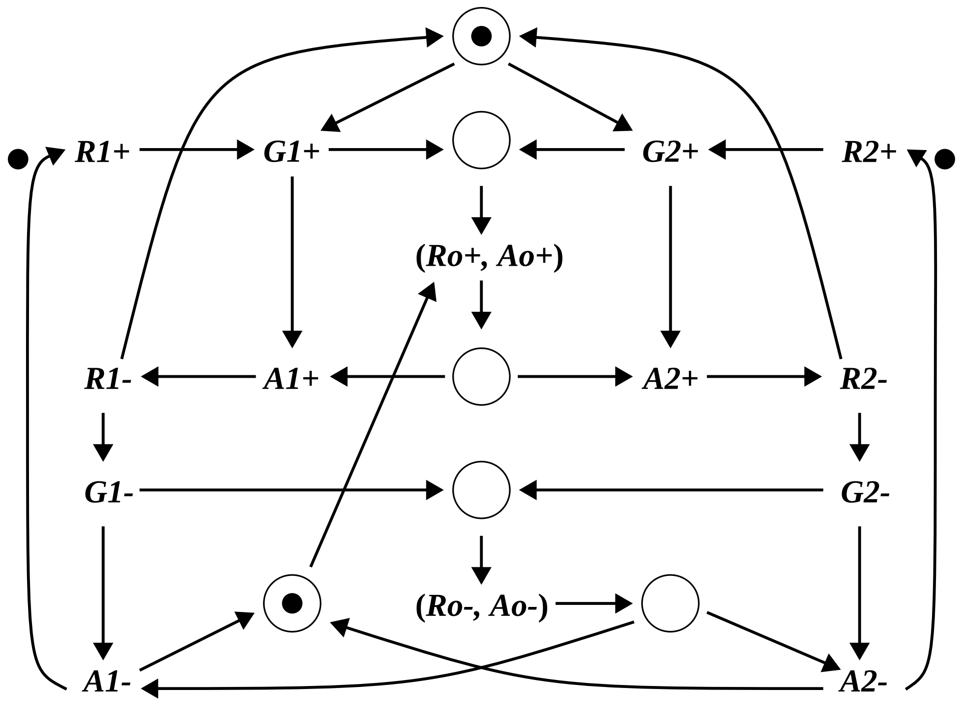


Figure 6-13: PN of the bARBITER

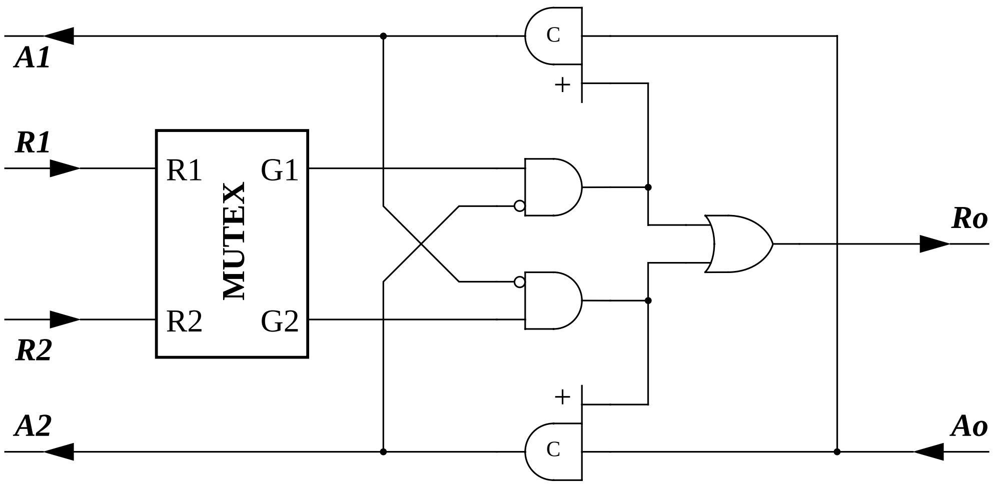


Figure 6-14: bARBITER circuit implementation

6.4 JOIN modules

The JOIN module synchronizes and concatenates two input links to the output link, and is used in organizing multiple control flows or pipelines. This section presents three types of JOIN module: pJOIN, dJOIN and bJOIN. The first two JOIN modules use the four-phase early protocol, while the last employs the four-phase broad protocol. As all of the PN specifications in this section and the following sections are quite straightforward they are omitted for the sake of brevity.

6.4.1 pJOIN module

Figure 6-15 shows a circuit implementation for a JOIN module, called pJOIN. The pJOIN module allows concurrent processing on the input link and resetting on the output link. However, the input and output links are not allowed to reset in parallel, with the input link being first reset and the output link following.

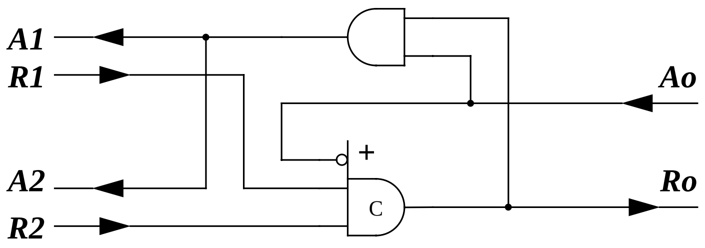


Figure 6-15: pJOIN circuit implementation

6.4.2 dJOIN module

Figure 6-16 shows a circuit implementation for a JOIN module, called dJOIN. Like the pJOIN module, the dJOIN module allows concurrent processing on the input link and resetting on the output link. Furthermore, concurrent resetting on both the input and output links are also allowed in the dJOIN module. The resetting on the output link can start even before that on the input link. The output link has the property of self-resetting as soon as it has completed the calling procedure; resetting of the output link does not depend on an input reset request.

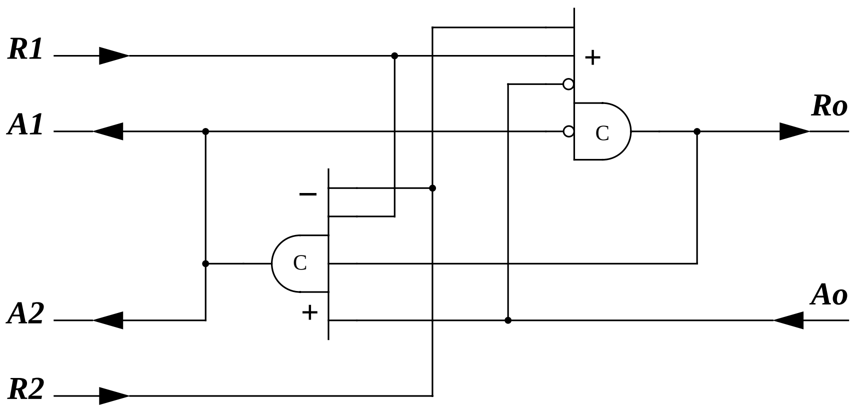


Figure 6-16: dJOIN circuit implementation

6.4.3 bJOIN module

The first two JOIN modules described above use the four-phase early protocol. A circuit for an ARBITER module using the broad protocol, called bARBITER, is shown in figure 6-17. The circuit is simple, and is similar to a C-gate. A difference is that the signal *Ro*, not *Ao*, is fed back internally in a C-gate.

The bARBITER module can also be used for cases where the early protocol is used since the specification of the broad protocol is more than sufficient to cover that of the early protocol. However, the operation of the circuit is totally sequential, which is undesirable from the performance perspective.

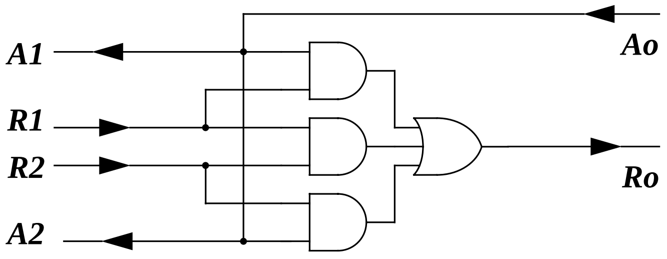


Figure 6-17: bJOIN circuit implementation

6.5 FORK modules

The FORK module is often used when there are multiple destinations. It is worth noting that the FORK module and the isochronic fork [92] discussed in section 2.2.2 are completely different concepts that have no relation to each other. This section presents three types of FORK module: pFORK, dFORK and bFORK. The first two FORK modules use the four-phase early protocol, while the last employs the broad protocol.

6.5.1 pFORK module

Figure 6-18 shows a circuit implementation for a FORK module, called pFORK. The pFORK module allows concurrent processing on the input link and resetting on the output link. However, the input and output links are not allowed to reset in parallel, with the input link being first and the output link following.

6.5.2 dFORK module

Figure 6-19 shows a circuit implementation for a FORK module, called dFORK. Like the pFORK module, the dFORK module allows concurrent processing on the input link

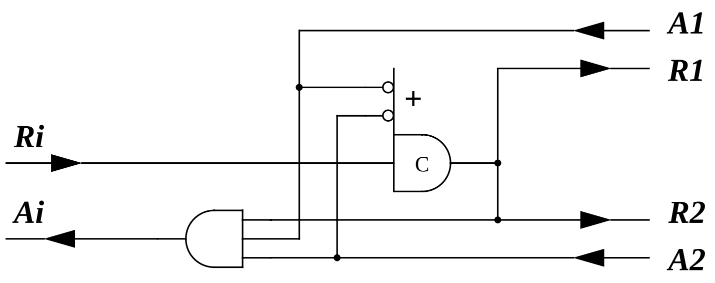


Figure 6-18: pFORK circuit implementation

and resetting on the output link. Furthermore, concurrent resetting on both the input and output links are also allowed in the dFORK module. The resetting on the output link can start even before that on the input link. The output link has the property of self-resetting as soon as it has completed the calling procedure; resetting of the output link does not depend on an input reset request.

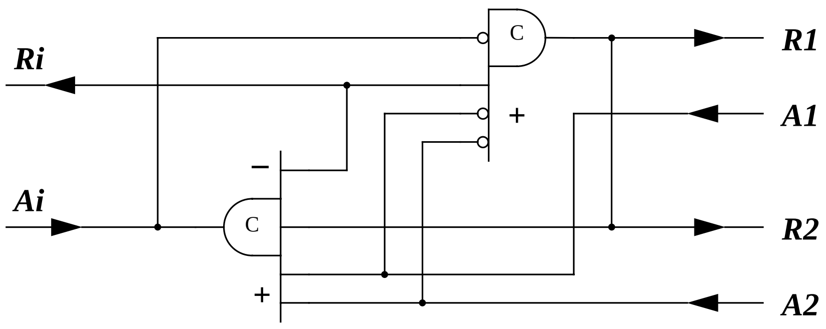


Figure 6-19: dFORK circuit implementation

6.5.3 bFORK module

The first two FORK modules described above use the four-phase early protocol. A circuit for a FORK module using the broad protocol, called bFORK, is shown in figure 6-20. The bFORK module can also be used for cases where the early protocol is used since the specification of the broad protocol is more than sufficient to cover that of the early protocol. However, the operation of the circuit is totally sequential, which is undesirable from the performance perspective.

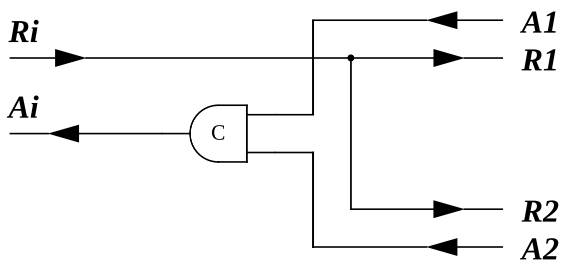


Figure 6-20: bFORK circuit implementation

6.6 SELA modules

This section presents three types of SELA modules: pSELA, dSELA and bSELA. All of these three SELA modules use a control link. The sSELA module serves the role of the if-else statement in programming languages. The input request first issues a handshake along the control link. If the returned value of the dual-rail acknowledge signal is true, the handshake will proceed along the output link (*R1*, *A1*); otherwise it goes along the output link (*R2*, *A2*). The first three SELA modules use the four-phase early protocol, while the last employs the four-phase broad protocol.

6.6.1 pSELA module

Figure 6-21 shows a circuit implementation for a SELA module, called pSELA. The pSELA module allows concurrent processing on the input link and resetting on the output link. However, the input and output links are not allowed to reset in parallel, with the input link being first and the output link following.

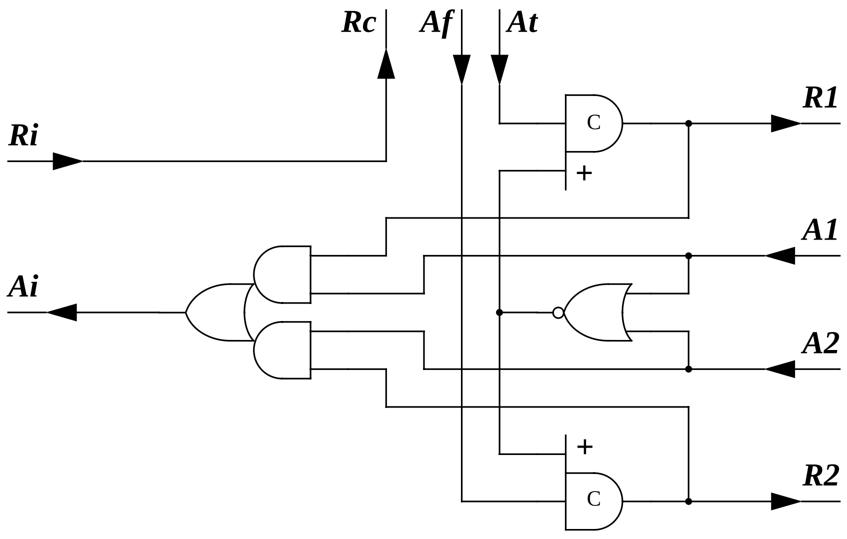


Figure 6-21: pSELA circuit implementation

6.6.2 dSELA module

Figure 6-22 shows a circuit implementation for a SELA module, called dSELA. Like the pSELA module, the dSELA module allows concurrent processing on the input link and resetting on the output link. Furthermore, concurrent resetting on both the input and output links are also allowed in the dSELA module. The resetting on the output link can start even before that on the input link. The output link has the property of self-resetting as soon as it has completed the calling procedure; resetting of the output link does not depend on an input reset request.

6.6.3 bSELA module

The first two SELA modules described above use the four-phase early protocol. A circuit for a SELA module using the broad protocol, called bSELA, is shown in figure 6-23. Note that the acknowledge signals of the control link are dual-rail encoded, so they are

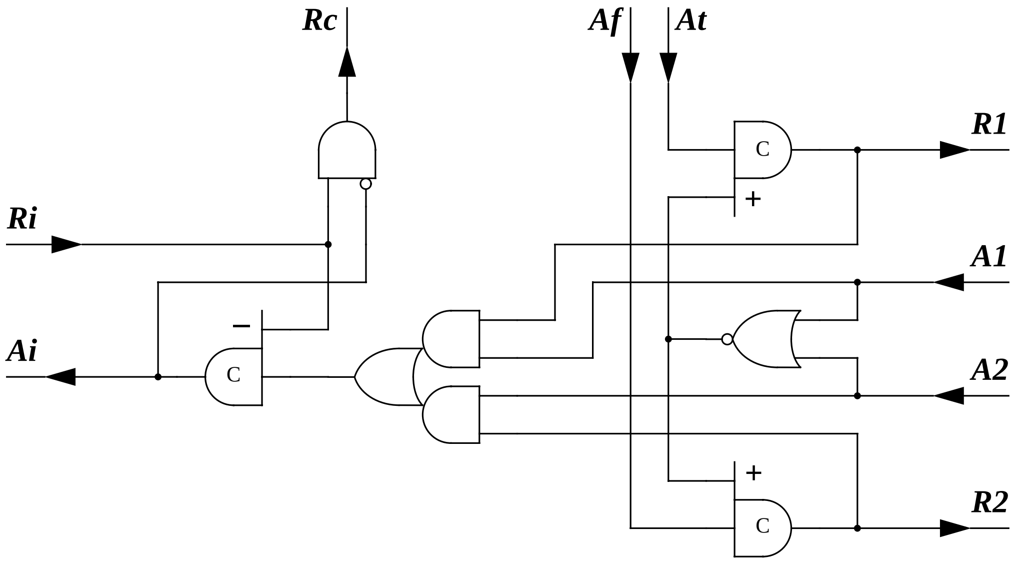


Figure 6-22: dSELA circuit implementation

able to convey a Boolean value and make the circuit implementation speed-independent. The bSELA module can also be used for cases where the early protocol is used since the specification of the broad protocol is more than sufficient to cover that of the early protocol. However, the operation of the circuit is totally sequential, which is undesirable from the performance perspective.

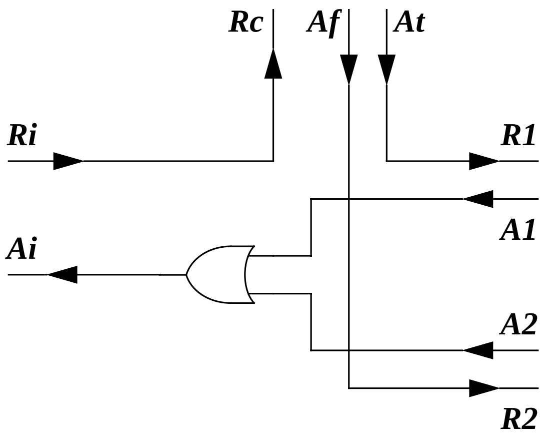


Figure 6-23: bSELA circuit implementation

6.7 SELB modules

There are often cases where a boolean guard is known prior to incoming input handshakes and remains stable during the process of handshaking. The SELB modules are designed for those cases. Figure 6-24 shows an implementation of the SELB module using the SELA module. This SELB module is still considered to be speed-independent as long as the Boolean guard is well controlled by the environment.

Generally, most speed-independent circuits are *robust*, where *robust* means that multiple input changes are allowed and the orders of input changes do not affect the behaviour of the circuit. This property is certainly desirable to designers. However, if the input changes of a specification are restricted by the environment and are known a priori to designers, the circuit implementation could be much simplified and more efficient.

By taking the nature of Boolean guard into account, simple and efficient circuit implementations of the SELB module can be derived; they are omitted here for the sake of brevity.

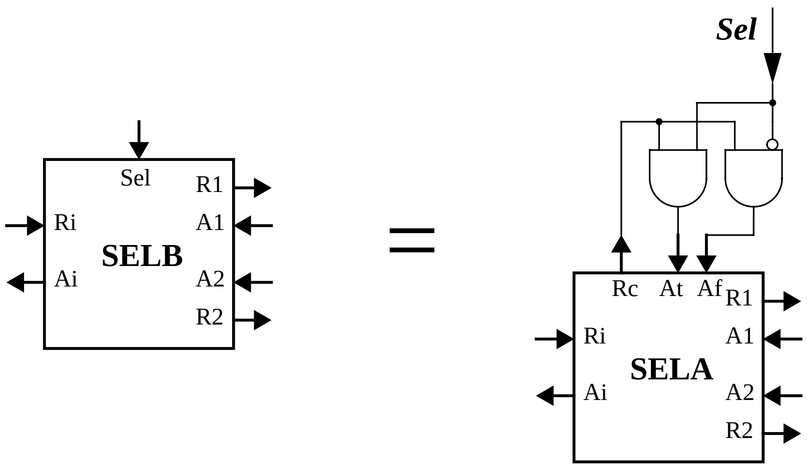


Figure 6-24: Implementation of the SELB modules

6.8 TOGGLE module

The TOGGLE module produces communications alternately on its two outputs in response to its input. The TOGGLE module is a useful building block and can be used to construct other control modules or even asynchronous systems. However, the TOGGLE module itself is the most difficult module to implement, though it appears to be quite simple. Many circuit implementations had been derived and then verified not to be speed independent. The difficult lies in the fact that circuit implementations tend to contain an inherent race hazard.

Figure 6-25 shows a circuit implementation for the TOGGLE module using NOR gates. Since the TOGGLE module is designed mainly as a basic building block for constructing other control modules, there are no associated acknowledge signals to form input or output links. Therefore, the environment must provide an input at a proper point only after the outputs have responded the previous input changes. Analysis of this circuit implementation has demonstrated that the operation is totally sequential, and races cannot happen as there is only one enabled transition in every possible state.

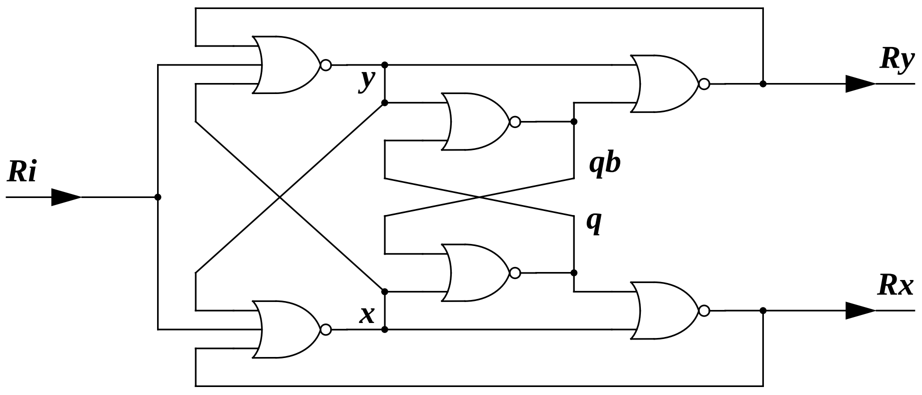


Figure 6-25: TOGGLE circuit implementation

6.9 An example: a counter

This section shows how an n-bit speed-independent counter is built using the TOGGLE modules as the building blocks. The worse case settling time of this counter is large since the carry may propagate from the low-order bit up to the high-order bit. However, only two bits change per operation on average [93]; the typical case is much faster than the worst case.

Figure 6-26 illustrates the diagram of the n-bit speed-independent counter. The carry stops at a bit position where the internal state variable (see figure 6-25) is zero; this is indicated by the transition along the *Rx* output of the TOGGLE module. If one bit stage is one, the transition happens along the *Ry* output which is connected to the next neighbour TOGGLE module. There are only two input states for the Completion Detector: either all are zeros or only one of them is one. When a change from one input state to the other is detected, it means the carry has completed its journey and the result is generated. The result lies in the internal state variable of the TOGGLE modules.

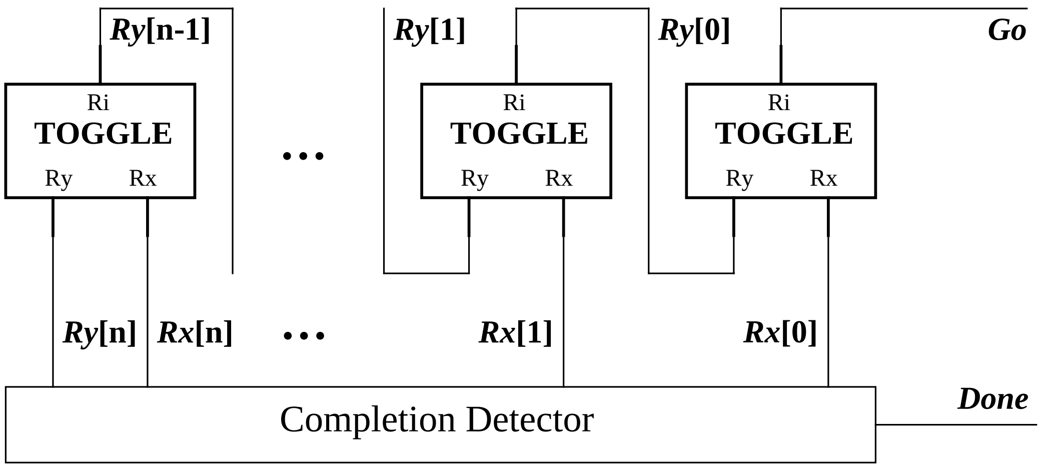


Figure 6-26: Speed-independent incrementer

6.10 Arbiter modules revisited

All of the arbiter circuits described in section 6.3 take the micropipeline view that the request signal initiates the data transfer (this is called a *push* channel). However, arbiters are often used in a bus structure, where the acknowledge signal initiates the data transfer (called a *pull* channel). One undesirable property of push arbiters is that the output request must wait whenever the MUTEX element goes metastable. Therefore the latency is unbounded, which is quite serious in some applications requiring low latency. This section presents two types of pull ARBITER module. The eARBITER modules use the four-phase early protocol, while the fARBITER module employs the broad protocol.

6.10.1 eARBITER module

A specification and implementation of a pull ARBITER module, called eARBITER, are shown in figure 6-27 and figure 6-28, respectively. The request signal *Ro* directly follows the input requests and it is not necessary to wait until the output signals *G1* and *G2* of the MUTEX element have been resolved when a metastable state occurs. The circuit has a bounded request latency, which is important for applications requiring low latency. Note that the place “p1” can accommodates two tokens, which the PETRIFY tool can deal with. However, other tools based on STGs have restrictions for multiple token cases, though a CD specification can describe this situation using OR-type signal transitions.

6.10.2 fARBITER module

A specification and implementation of a pull ARBITER module using the broad protocol, called fARBITER, are shown in figure 6-29 and figure 6-30, respectively.

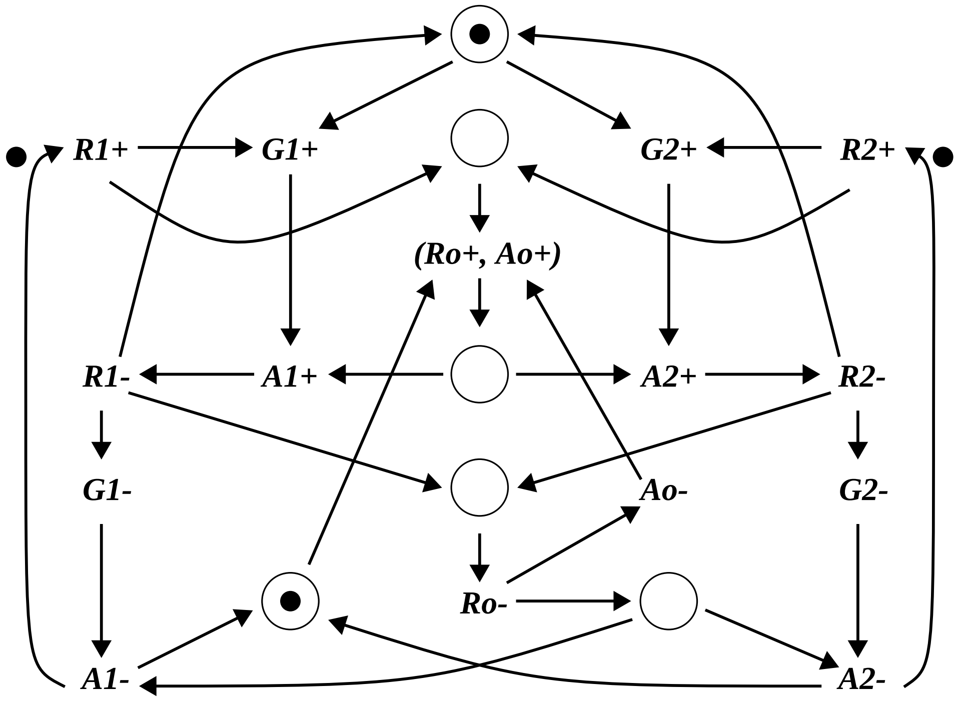


Figure 6-27: PN of the eARBITER module

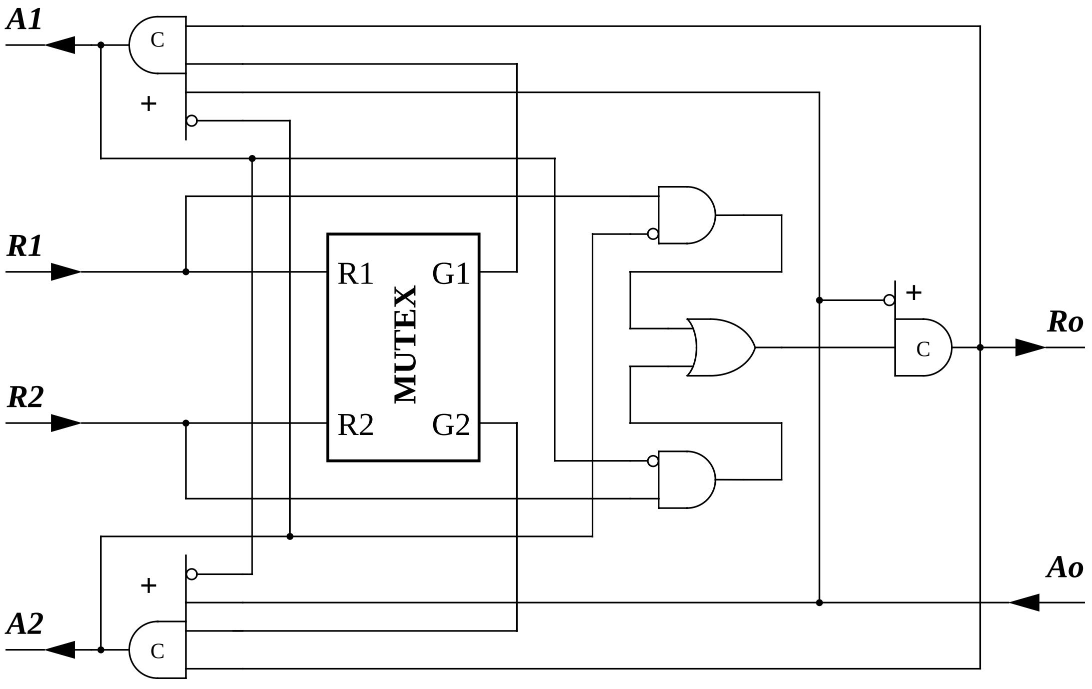


Figure 6-28: eARBITER circuit implementation

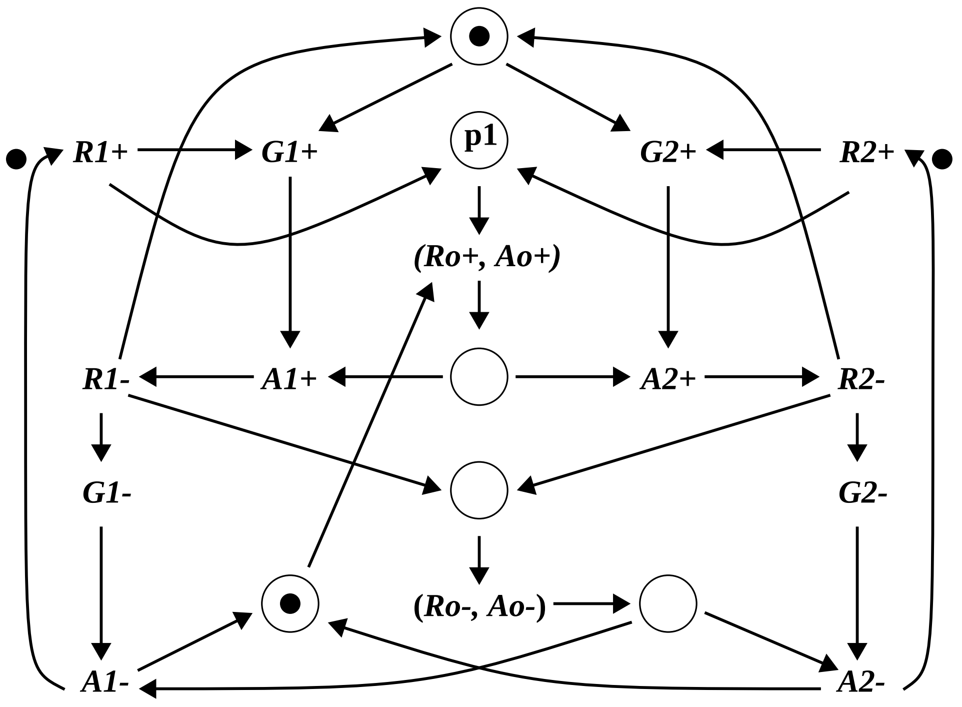


Figure 6-29: PN of the fARBITER module

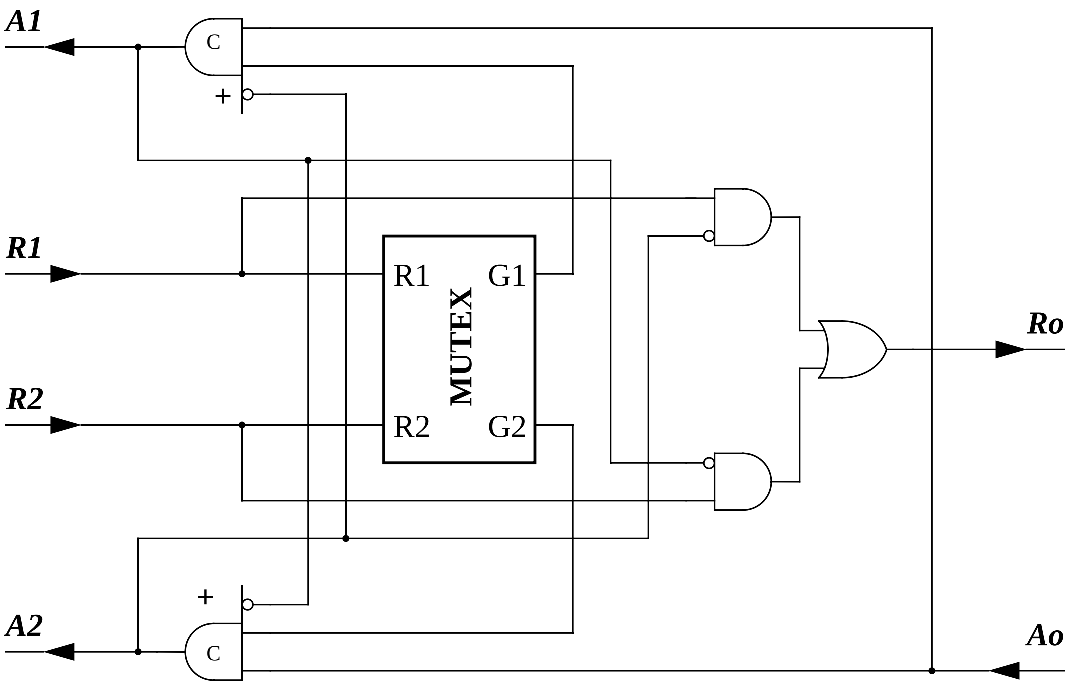


Figure 6-30: fARBITER circuit implementation

6.11 Modules with multiple input links

Up to now, all the modules presented have had at most two input links. There are often cases where multiple input links are required. Circuit implementations for modules with multiple input links can be derived following the procedure described in the previous sections. However, they are most practically built by using the corresponding two input link modules. This section examines the design of these modules with multiple input links. The design of a four-phase early protocol arbiter with multiple input links is taken as an example and discussed. The discussions can, in general, apply to other modules with multiple input links.

Figure 6-31 shows a tree arbiter with eight input links, where the solid dots represent the two input link arbiters. The following terms are defined for the sake of discussion. The top arbiter is called the *home node*, the bottom arbiters are called the *leaf nodes* and the arbiters between the home node and the leaf nodes are called the *directory nodes*. The input links connected to the same leaf node form a *leaf group*. The input links connected to the same directory node form a *directory group*. For an example, the input links *i1* and *i2* form a leaf group and input links *i5*, *i6*, *i7* and *i8* form a directory group.

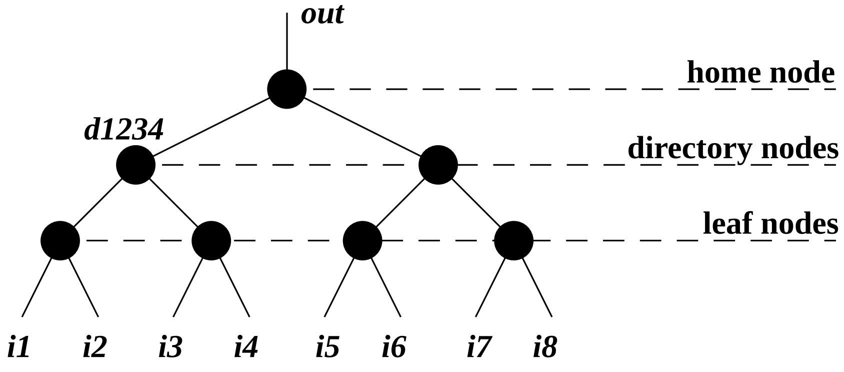


Figure 6-31: Tree arbiter

Imagine a case where all eight asynchronous input requests arrive and one input request (say, the input link *i1*) is granted and the other seven input requests are pending. We are interested in ***which*** input request will be granted and ***when*** after the granted input request from the input link *i1* is released.

All the ARBITER modules presented earlier are fair. In other words, a pending calling request must be granted after a bounded number of other input requests are granted. The bounded number is eight in this case. Seen from the home node, one input request from the directory group {*i5*, *i6*, *i7*, *i8*} will be granted after the input request from the input links *i1* is released because of the fair nature of the home node. Therefore, those input links should be put into different directory groups or different leaf groups if calling requests from those input links are likely to compete for a common resource. This is the first conclusion.

Suppose that sARBITER modules are used. The release of the calling request from the input link *i1* involves resetting all the nodes from the leaf node to the home node. All these resettings are sequential and thus delay a grant for other calling requests. Supposed that pARBITER modules are used. The situation will improve as the resetting of the output link of the home node and the falling transition of the input link of the directory node *d1234* can be in parallel. However, the circuit still waits for the falling transition of the calling request from the input link *i1*. This is unacceptable if the height of the tree structure is high. Supposed that dARBITER modules are used. The problem will be solved since the dARBITER module can reset the output link by itself as soon as it has completed the calling procedure and resetting of the output link does not depend on its input reset request (see “dARBITER module” on page 131). Therefore, dARBITER modules should be used to build an arbiter with multiple input links based on the tree structure. If other types of arbiters are used, the response time will be degraded. This is the second and also very important conclusion.

6.12 Summary

A set of control modules for four-phase micropipelines with different implementations has been presented. Arbiters, which are non-trivial and tricky to design, are also included. These control modules, together with the pipeline latch control circuits described in the previous chapter, can be used to construct complex and powerful asynchronous systems including forking or joining multiple pipelines. Also they can be used to construct other four-phase control modules. All of the proposed control modules are speed-independent, and this has been verified using the PETRIFY tool.

The design of an arbiter with multiple input links based on a tree structure has also been discussed. The dARBITER modules should be used to build an arbiter with multiple input links as their output links can be self-reset.

Petri nets have been shown to be an appropriate formalism for describing the behaviour of asynchronous systems with concurrency, causality and conflicts between events. Though most steps of the development of these control modules were processed by hand, the PETRIFY tool played a key role and was used to synthesize various implementations for comparison and analysis.

AMULET3i 7

AMULET3i is an asynchronous embedded system chip which incorporates the third generation asynchronous ARM processor (AMULET3). Different from its predecessors, AMULET1 and AMULET2e, AMULET3i is aimed to be a commercially viable product for communication applications. This will be a significant step. A brief description of AMULET3i and AMULET3 is given in this chapter in the hope of providing the big picture into which the components described in the previous chapters can be placed.

7.1 Introduction

As we said previously, it is our belief that asynchronous design must be justified on its practical significance rather than solely on a theoretical basis. The motivation behind the AMULET project is to demonstrate this practical significance.

AMULET1 demonstrated the feasibility of building an asynchronous system at the levels of complexity of current synchronous systems. AMULET2e proved the competitiveness of an asynchronous system compared with current synchronous systems, from both the power perspective and the performance perspective. AMULET3i will, in turn, put the asynchronous experience of the academic community into industrial practice.

7.2 AMULET3i

AMULET3i is a commercial asynchronous embedded system chip, whose organization is shown in figure 7-1. In addition to AMULET3 (the third generation asynchronous ARM processor), AMULET3i contains 8 Kbytes of RAM (which can also be configured

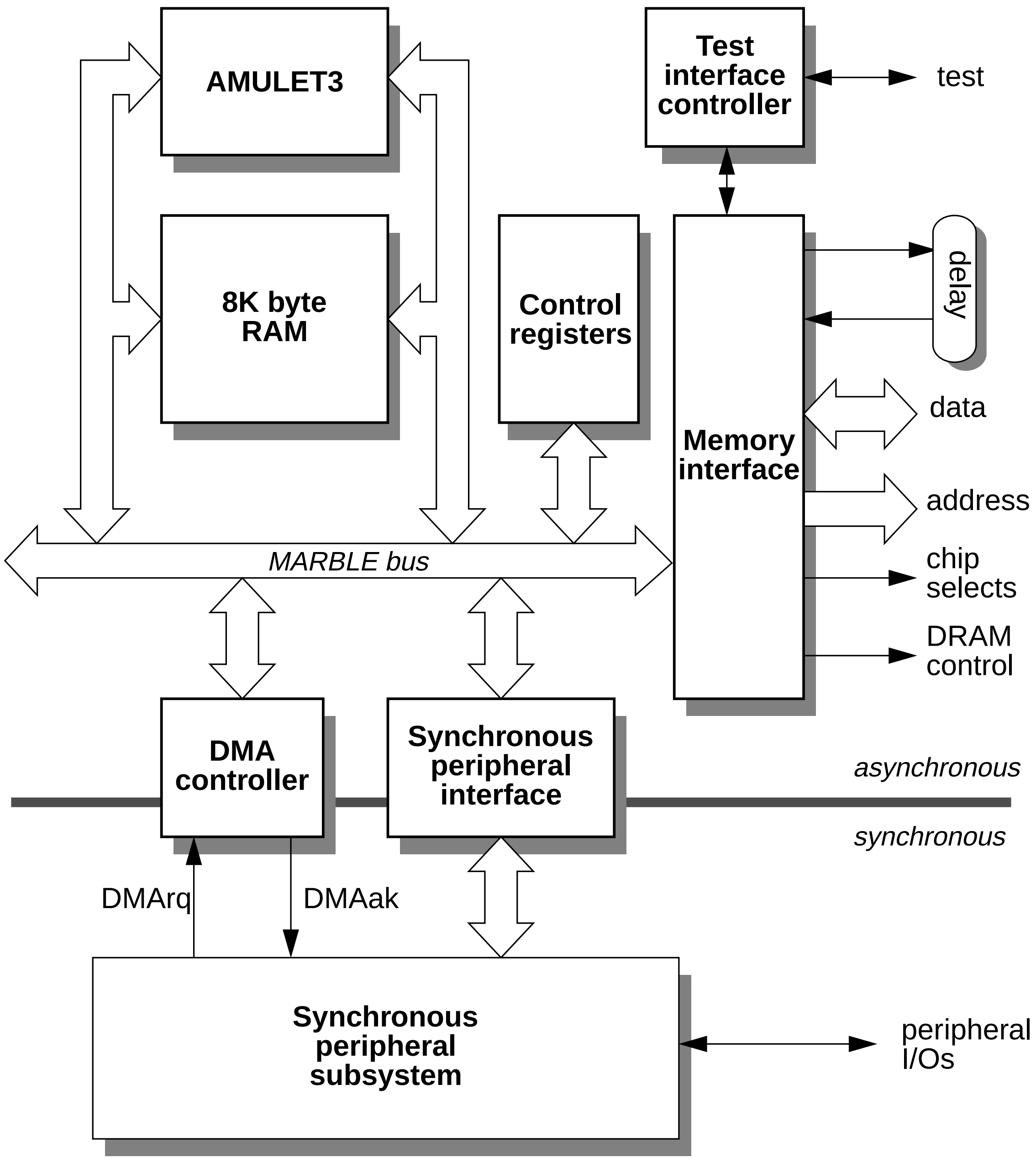


Figure 7-1: AMULET3i block diagram

as a cache), a DMA controller, a MARBLE (Manchester AsynchRonous Bus for Low Energy) bus [94], a flexible memory interface, a general synchronous peripheral interface, an on-chip synchronous peripheral subsystem, and various configuration and control registers. A test interface is also included to support the design for test strategy.

7.3 AMULET3

AMULET3 is the third generation asynchronous ARM processor. It implements the ARM architecture version 4 and supports the Thumb instruction set [95]. Figure 7-2 shows the block diagram of AMULET3, which consists of five major blocks. The

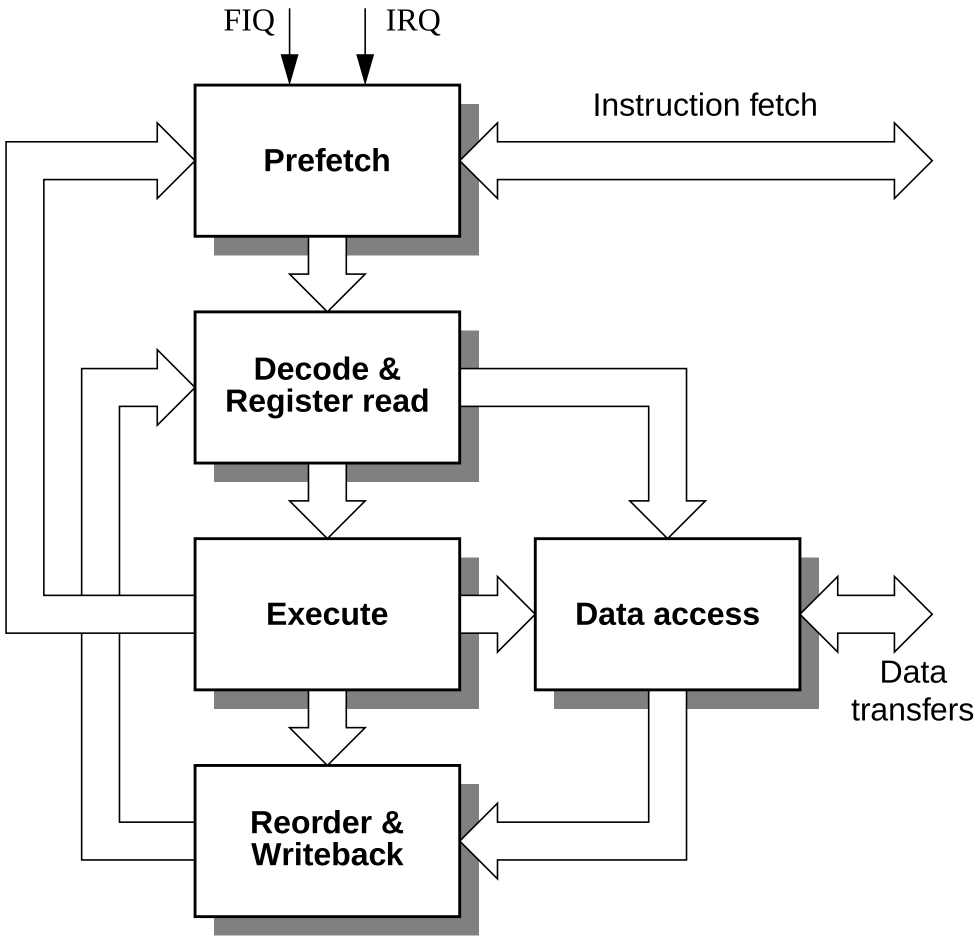


Figure 7-2: AMULET3 block diagram

detailed organization of AMULET3 is shown in Figure 7-3. (Note that figures 7-1, 7-2 and 7-3 are reproduced from the “Scoreboard” of the AMULET project with the kind permission of Professor Steve Furber). The design includes several novel features.

Firstly, a Harvard architecture is used and the data interface is sidelined from the main instruction flow. As a result, data transfer operations, especially multiple load and store instructions, can be decoupled from purely internal operations. Another benefit of this organization is that an interrupt can be dealt with in the Prefetch Unit rather than in the Decoder Unit and treated as a predicted branch, giving a fast interrupt response. As loaded values are reordered into the Register Bank and data aborts are allowed to be delayed, there is significant speculation following a load or store instruction without paying penalties for slow memory.

Secondly, instructions are allowed to execute out of order and a Reorder Buffer [96] (borrowed from superscalar design techniques) is used to hold results to be written back to the Register Bank in order. This reorder buffer is, in essence, an implementation of the register renaming mechanism. Therefore, result forwarding (not only the last result as in AMULET2e [11]) can be achieved in a deterministic and arbitration-free manner. It is worth noting that two Thumb instructions are fetched per bus cycle, which is another superscalar aspect of the design.

Finally, branch prediction and a halt mechanism are included. The halt mechanism is straightforward in asynchronous designs and achieves a three to four orders of magnitude power saving [11] in the idle state, whereas a synchronous design can only approach this power efficiency by stopping the clocks with considerable effort.

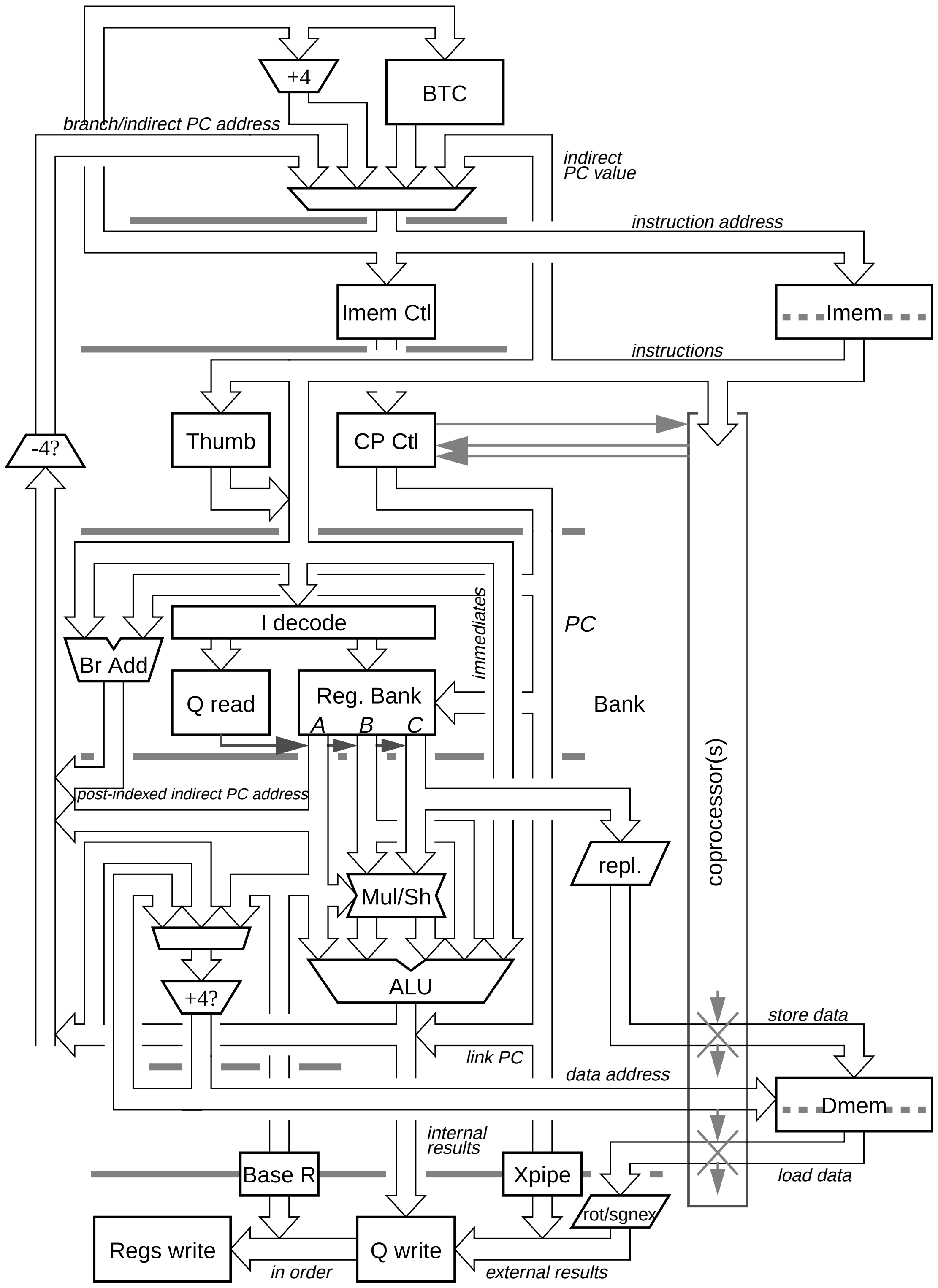


Figure 7-3: AMULET3 organisation

7.4 Execution unit

Figure 7-4 shows the block diagram of the Execution Unit. The ALU comprises a logic unit and an adder unit. The design of the adder unit is presented in chapter 3. The design of the multiplier is described in chapter 4. Multiplexers are used to implement the result forwarding mechanism, and choose operands either from the Register Bank or from the Reorder Buffer (which is also called the Queue).

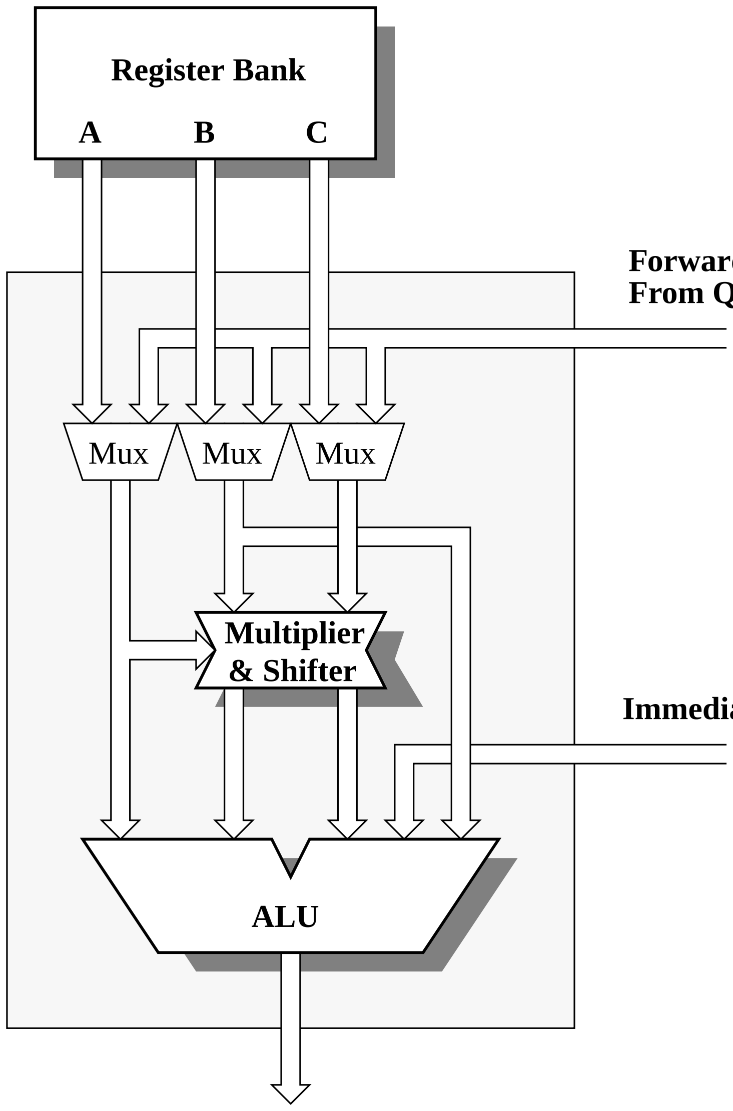


Figure 7-4: Execution pipeline organization

7.5 Implementation

Figure 7-5 shows an implementation oriented view of the AMULET3 datapath structure.

AMULET3i is designed using a 0.35 µm triple metal CMOS technology.

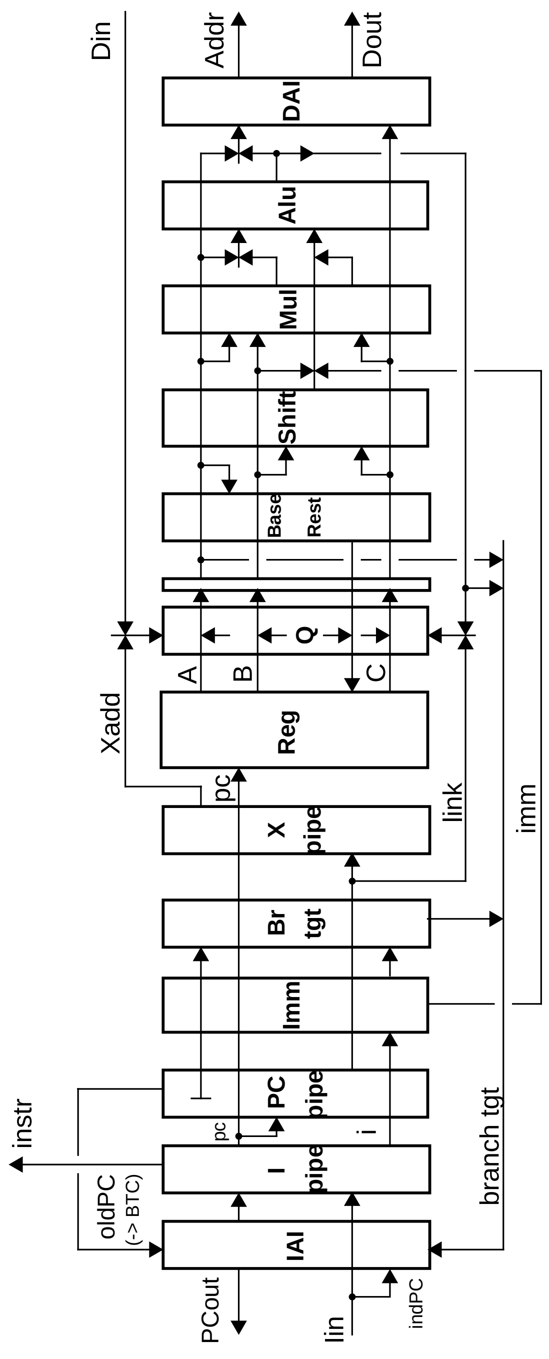


Figure 7-5: AMULET3 datapath structure

7.6 Summary

A brief description of AMULET3i and AMULET3 has been given. AMULET3i is an asynchronous embedded system chip, which is aimed to be commercially viable product for communication applications. AMULET3 is the third generation asynchronous ARM processor, which implements the ARM architecture version 4 and supports the Thumb instruction set. Clearly, the adder and the multiplier, described in the chapter 3 and 4, have directly contributed to AMULET3i. Two sets of asynchronous control circuits, described in the chapter 5 and 6, have also contributed to AMULET3i, but this is not clear in the pictures presented here.

Conclusions 8

This thesis has presented engineering work on asynchronous design. The arithmetic and control components were designed and implemented for AMULET3i, a commercial asynchronous embedded system chip for communication applications. The arithmetic components comprise an adder and a multiplier; these two are critical to the performance of the processor core. The control components consist of a set of pipeline latch control circuits and a set of control modules; all of these components are speed-independent. Though the nature of the work is mainly engineering, there are some significant new insights gained in the course of the work.

8.1 Contributions

A novel carry arbitration scheme was proposed (and has been patented) for parallel adder circuits. The proposed scheme provides an efficient encoding in which the carry is generated by arbitrating several input carry requests, exploiting the associativity of the carry computation. The new coding is a logically redundant superset of the conventional carry process. Departing from this general coding, certain modifications which reduce the redundancy can easily be made where this simplifies the implementation. The new scheme not only leads to high speed adders due to the reduction in the required layers of logic, but also offers a regular and compact layout and uniform fan-in and fan-out loadings. A high performance, low power 32-bit adder for AMULET3i has been designed using the new scheme and implemented down to the layout level. It takes 1.8 ns to complete a 32-bit addition and occupies 137.2 μm $\times$ 524.8 μm of chip area in a 0.35 μm triple metal CMOS technology. The power estimate of the datapath is about 8 and 17 mW operating at 100 and 200 MHz (under typical process conditions), respectively.

A high performance, low power asynchronous 32 bit multiplier with a reasonable hardware resource has been developed for AMULET3i. A new encoding technique has been used in the AMULET3i multiplier to adjust the product result of an unsigned number multiply operation. An adjustment value is made on the least significant 32-bit positions. A new 4-2 compressor with an enable control has been presented, together with several other circuit design techniques including the use of true single-phase clocking registers. The elegance of this multiplier is the manner in which the algorithm and the circuit implementation are well matched within the asynchronous framework. Post-layout simulation, in a 0.35 micron triple metal CMOS technology, shows that it takes 11.2 ns (2.8 ns $\times$ 4 cycles) to complete the computation of a 32-bit multiplication in the worst case. The power estimate of the datapath is about 40 and 82 mW operating at 100 and 200 MHz (under typical process conditions), respectively. The layout is regular and compact with a datapath area of only 416.4 $\times$ 639.6 μm^2.

A set of pipeline latch control circuits for four-phase asynchronous pipelines has been proposed. These can be used to organize arithmetic components efficiently into a micropipeline. All of the proposed pipeline latch control circuits are speed-independent, and this has been verified using the FORCAGE tool. A four-phase micropipeline can be configured either in a request-activated form or in an acknowledge-activated form. The latter is the framework within which dynamic logic can be exploited for low power.

A set of control modules has been proposed in order to ease the design of asynchronous systems based on four-phase micropipelines. Arbiters, which are non-trivial and tricky to implement, are also included. All of the proposed control modules are speed-independent, and this has been verified using the PETRIFY tool. These control modules, together with the pipeline latch control circuits, can be used to construct complex and powerful asynchronous systems.

8.2 Future work

There are some application areas where asynchronous designs are likely to demonstrate advantages. Our philosophy is still to prove that the theoretical benefits are practically realizable, and this is reflected in the engineering nature of work presented here.

There are two areas where asynchronous designs are attacking and are likely to win. The first is the low power market where short battery life is the bane of the user and the second is the mobile communication market where good EMC is required. Thus more future work is expected in these two areas.

8.2.1 Low power market

The field of low power designs using traditional clocked design methodologies has been plagued with fundamental difficulties. Global clock generation and distribution is blamed for a significant portion of the total power consumption in a synchronous CMOS circuit [97]. Though advanced power management can deal with clock gating and even shut down clocks, this comes at a price in terms of increased complexity. However, advanced power management is inherent within the asynchronous design methodology. Power is only consumed when needed. There are many other arguments which suggest an asynchronous design is a low power design. But the most convincing demonstration is the AMULET2e work, which reduces power below that achievable in the industry-leading clocked ARM designs.

It is worth noting that there is no single solution to the power consumption problem. A design should consider power at all levels of the design hierarchy, including the technology, layout, circuit, logic, design style, architectural and algorithmic levels [98-103].

8.2.2 Mobile communication market

In the early 19th Century, the French mathematician Jean-Baptiste Fourier proved that any reasonably behaved **periodic** function, *g(t)*, with frequency *f* can be constructed by summing a number of sines and cosines:

$$g(t) = c + \sum_{n=1}^{\infty} a_n \sin(2\pi nft) + \sum_{n=1}^{\infty} b_n \cos(2\pi nft)$$

where c is a constant, a_n and b_n are the sine and cosine amplitudes of the nth harmonics, which decrease as n increases. From the above equation, it is clear that a synchronous system produces “harmonic pollution” that aligns with harmonics of the clock, in addition to “fundamental noise” that aligns with the clock frequency. However, **periodic** operation is the fundamental property of synchronous systems and there is no way around this. Fortunately, asynchronous systems are **aperiodic** and therefore do not produce harmonic pollution (or produce negligible harmonic pollution). This very good EMC is a unique advantage of asynchronous systems. It is worth noting that an asynchronous system generates less fundamental noise compared with a similar synchronous system as it produces broadband distributed current without the high amplitude peaks. Recent work has shown that the magnitude of the current peak of a synchronous system is 2.5 times that of a similar asynchronous system [104]. With increasingly rigorous EMI compliance specifications and testing, good EMC properties will demonstrate another meritorious aspect of asynchronous design.

8.3 Asynchronous prospects

"It is possible that all the renewed interest in asynchronous techniques will come to nothing, though this seems unlikely. It is also possible that industry will suddenly see the asynchronous light and switch completely to the new approach. This seems even more unlikely! What seems more likely is that areas will be identified where asynchronous approaches have really worthwhile advantages; these will be niches in otherwise synchronous designs."

The above statement was made by Professor Steve Furber at a time shortly after the AMULET group was established. It still remains true today. In the intervening years, work in the AMULET group and elsewhere has moved asynchronous technology much closer to commercial reality. The research described in this thesis is expected to contribute to this movement, making the low power and EMC advantages inherent in asynchronous technology more accessible to the designers of products which need these benefits.

Bibliography

[1] Furber, S.B., “Computing without clocks”, Proceedings of the VII Banff Workshop: Asynchronous Hardware Design, Banff, Canada, 1993.

[2] Asynchronous Logic Home Page, http://www.cs.man.ac.uk/amulet/async/index.html.

[3] On-Line Asynchronous Bibliography, http://www.win.tue.nl/win/cs/pa/wsinap/async.html.

[4] 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, Boston, Massachusetts, October 1994.

[5] Furber, S.B., Day, P., Garside, J.D., Paver, N.C. and Woods, J.V., “A Micropipelined ARM”, Proceedings of the IFIP TC 10/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.

[6] Furber, S.B., Day, P., Garside, J.D., Paver, N.C. and Woods, J.V., “AMULET1: A Micropipelined ARM”, Proceedings of the IEEE Computer Conference, San Francisco, March 1994.

[7] Paver, N.C., “The design and implementation of an asynchronous microprocessor”, PhD thesis, University of Manchester, June 1994.

[8] Woods, J.V., Day, P., Furber, S.B., Garside, N.C., Paver, N.C. and Temple, S., “AMULET1: an asynchronous ARM microprocessor”, IEEE Transactions on Computers, vol. 46, April 1997, pp. 385-398.

[9] Furber, S.B., “VLSI RISC architecture and organization”, Marcel Dekker Inc., New York, 1989.

[10] Furber, S.B., “ARM System Architecture”, Addison Wesley Longman Limited, New York, 1996.

[11] 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, IEEE Computer Society Press, April 1997.

[12] Liu, J., “Digital adder circuits”,
UK Patent no. 9620526, November 1996.

[13] Booth, A.D., “A signed binary multiplication technique”,
Quarterly Journal of Mechanics and Applied Mathematics,
vol. 4, June 1951, pp. 236-240.

[14] MacSorley, O.L., “High-speed arithmetic in binary computers”,
Proceedings of the IRE, vol. 49, January 1961, pp. 67-91.

[15] Yuan, J. and Svensson, C., “High-speed CMOS circuit techniques,”
IEEE Journal of Solid-State Circuits, vol. 24, February 1989,
pp. 62-70.

[16] Liu, J., “The design of asynchronous multiplier”,
MSc thesis, University of Manchester, June 1995.

[17] Day, P. and Woods, J.V., “Investigation into micropipeline latch
design styles”, IEEE Transaction on VLSI Systems, vol. 3,
June 1995, pp. 264-272.

[18] Furber, S.B. and Day, P., “Four-phase micropipeline latch control
circuits”, IEEE Transaction on VLSI Systems, vol. 4, June 1996,
pp. 247-253.

[19] Furber, S. B. and Liu, J., “Dynamic logic in four-phase
micropipelines”, Proceedings of Async’96,
IEEE Computer Society Press, March 1996.

[20] Sutherland, I.E., “Micropipelines”, The 1988 Turing Award
Lecture, Communications of the ACM,vol. 32, June 1988,
pp. 720-738.

[21] Kishinevsk, M., Kondratyev, A., Taubin, A. and Varshavsky, V.,
“Concurrent hardware: the theory and practice of self-timed
design”, John Wiley & Sons, Inc., New York, 1994.

[22] Murata, T., “Petri nets: properties, analysis and applications”,
Proceedings of IEEE, vol. 77, April 1989, pp. 541-580.

[23] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L. and
Yakovlev, A., “Complete state encoding based on the theory of
regions”, Proceedings of Async’96, IEEE Computer Society Press,
March 1996.

[24] Cortadella, J., Kishinevsky, M., Lavagno, L. and Yakovlev, A.,
“Synthesizing Petri nets from state-based models”,
Proceedings of ICCAD’95, November 1995, pp. 164-171.

[25] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L. and
Yakovlev, A., “Methodology and tools for state encoding in
asynchronous circuit synthesis”, Proceedings of ACM/IEEE
Design Automation Conference, 1996.

[26] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L. and Yakovlev, A., “Technology mapping of speed-independent circuits based on combinational decomposition and resynthesis”, Proceedings of European Design and Test Conference, 1997.

[27] Chaney, T.J. and Molnar, C.E., “Anomalous behavior of synchronizer and arbiter circuits”, IEEE Transactions on Computers, vol. 22, April 1973, pp. 421-422.

[28] Couranz, G.R. and Wann, D.F., “Theoretical and experimental behaviour of synchronizers operating in the metastable region”, IEEE Transactions on Computers, vol. 24, June 1975, pp.604-616.

[29] Kinniment, D.J. and Woods, J.V., “Synchronisation and arbitration circuits in digital systems”, Proceedings of IEE, vol. 123, no. 10, October, 1976, pp. 961-966.

[30] Horstmann, J.U., Eichel, H.W. and Coates, R.L.,“Metastability behavior of CMOS ASIC flip-flops in theory and test”, IEEE Journal of Solid-State Circuits, vol. 24, February 1989, pp. 146-157.

[31] Mead, C. and Conway, L., “Introduction to VLSI systems”, Addison-Wesley, London, 1980.

[32] Huffman, D.A., “The synthesis of sequential switching circuits”, J.Franklin Institute, vol. 257, March/April 1954, pp. 161-190/275-303.

[33] Unger, S.H., “Asynchronous sequential switching circuits”, Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[34] Yun, K. and Dill, D., “Automatic synthesis of 3D asynchronous state machine”, Proceedings of ICCAD, 1992.

[35] Yun, K., Dill, D. and Norwick, S.M., “Synthesis of 3D asynchronous state machines”, Proceedings of ICCD, 1992.

[36] Lavagno, L. and Vincentelli, A.S., “Algorithms for synthesis and testing of asynchronous circuits”, Kluwer Academic Publishers, 1993.

[37] Muller, D.E. and Bartky, W.C., “A theory of asynchronous circuits”, Annals of Computing Laboratory of Harvard University, 1959, pp.204-243.

[38] Staunstrup, J., “A formal approach to hardware design”, Kluwer Academic Publishers, 1994.

[39] Ebergen, J.C., “Translating programs into delay-insensitive circuits”, PhD thesis, Eindhoven University of Technology, 1987.

[40] Ebergen, J. C., “A formal approach to designing delay-insensitive circuits”, Distributed Computing, vol. 5, July 1991, pp. 107-119.

[41] Molnar, C.E., Fang, T.P. and Rosenberger, F.U., “Synthesis of delay-insensitive modules”, Proceedings of the 1985 Chapel Hill Conference on Advanced Research in VLSI, 1985.

[42] Chu, T.A., Leung, C.K.C. and Wanuga, T.S., “A design methodology for concurrent VLSI systems”, Proceedings of ICCD, 1985.

[43] Rosenblum, L.Y. and Yakovlev, A.V., “Signals graph: from self-timed to timed ones”, International Workshop on Timed Petri Nets, Torino, Italy, 1985.

[44] Kishinevsky, M.A., Kondratyev, A.Y., Taubin, A.R. and Varshavsky, V.I., “On self-timed behavior verification”, Proceedings of TAU’92, March 1992.

[45] Verhoeff, T., “Delay-insensitive codes — an overview”, Distributed Computing, vol. 3, 1988, pp. 1-8.

[46] Sparsø, J. and Staunstrup, J., “Delay-insensitive multi-ring structures”, Integration, the VLSI Journal, vol. 15, 1993, pp. 313-340.

[47] Martin, A.J., “Programming in VLSI: from communicating processes to delay-insensitive circuits”, UT Year of Programming Series, Hoare, C.A.R., Ed., Addison-Wesley, 1990.

[48] Berkel, K.V., “Handshake circuits: an asynchronous architecture for VLSI programming”, volume 5, International Series on Parallel Computation, Cambridge University Press, 1993.

[49] Paver, N.C., “Condition detection in asynchronous pipelines”, UK Patent no. 9114513, October 1991.

[50] Garside, J.D., “A CMOS VLSI implementation of an asynchronous ALU”, Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England, 1993.

[51] Ling, H., “High-speed binary adder”, IBM J.Res.Development, vol. 25, 1981, pp. 156-166.

[52] Brent, R.P. and Kung, H.T., “A regular layout for parallel adders”, IEEE Transactions on Computers, vol. 31, 1982, pp. 260-264.

[53] Oklodzija, V.G. and Barnes, E.R., “On implementing addition in VLSI technology”, Parallel Distributed Computing, vol. 5, 1988, pp. 716-728.

[54] Lynch, T. and Swartzlander, E.E., “A spanning tree carry lookahead adder”, IEEE Transactions on Computers, vol. 41, 1992, pp. 931-939.

[55] Quach, N.T. and Flynn, M.J., “High-speed addition in CMOS”, IEEE Transactions on Computers, vol. 41, 1992, pp. 1612-1615.

[56] Sklansky, J, “Conditional-sum addition logic”, IRE Transactions on Electronic Computers, vol. 9, 1960, pp. 226-231.

[57] Bedrij, O.J., “Carry-select adder”, IRE Transactions on Electronic Computers,vol. 9, 1962, pp. 226-231.

[58] Avizienis, A., “Signed-digit number representations for fast parallel arithmetic”, IRE Transactions on Electronic Computers, vol. 10, September 1961, pp. 389-400.

[59] Srinivas, H.R. and Parhi, K.K., “A fast VLSI adder architecture”, IEEE Journal of Solid-State Circuits, vol. 27, June 1992, pp. 761-767.

[60] Ladner, R.E. and Fischer, M.J., “Parallel prefix computation”, J.ACM, vol. 27. 1980, pp. 831-838.

[61] Uyemura, J.P., “Circuit design for CMOS VLSI”, Kluwer Academic Publishers, 1992.

[62] Shoji, M., “CMOS digital circuit technology”, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1988.

[63] Weste, N.H.E. and Eshraghian, K., “Principle of CMOS VLSI design: a system perspective”, Addison-Wesley, Massachusetts, 1988.

[64] Suzuki, M. et al., “A 1.5 32-b CMOS ALU in double pass-transistor logic”, IEEE Journal of Solid-State Circuits, vol. 28, June 1993, pp. 1145-1151.

[65] Turley, J., “ARM tunes Piccolo for DSP performance”, Microprocessor Report, vol. 10, November 1996, pp. 17-20.

[66] Bewick, G. and Flynn, M.J., “Binary multiplication using partially redundant multiples”, Technical Report, CSL-TR-92-528, Computer Systems Laboratory, Standford University, June 1992.

[67] Omondi, A.R., “Computer arithmetic systems – algorithms, architecture and Implementation”, Prentice Hall International (UK) Limited, Cambridge, 1994.

[68] Day, P., “A micropipelined multiplier”, ACiD-WG/EXACT Workshop on Asynchronous Processing, Veldhoven, Netherlands, December 1992.

[69] Wallace, C.S., “A suggestion for parallel multipliers”, IEEE Transactions on Electronic Computer, vol. 13, 1964, pp. 14-17.

[70] Harata, Y., Nakamura, Y., Nagase, H., Takigawa, M. and Takagi, N., "A high-speed multiplier using a redundant binary adder tree", IEEE Journal of Solid-State Circuits, vol. 22, February 1987, pp. 28-34.

[71] Hennessy, J.L. and Patterson, D.A., "Computer architecture, a quantitative approach", Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[72] Zuras, D. and McAllister, W., "Balanced delay trees and combinatorial division in VLSI", IEEE Journal of Solid-State Circuits, vol. 21, October 1986, pp. 814-819.

[73] Wu, X. and Prosser, "Theory of transmission switches and its application to design of CMOS digital circuits", Int. J. Circuit Theory Application, vol. 20, 1992.

[74] Zhuang, N., and Wu, H., "A new design of the CMOS full adder", IEEE Journal Solid-State Circuit, vol. 27, May 1992, pp. 840-844.

[75] Wang, J.M., Fang, S.C. and Feng, W.S., "New efficient designs for XOR and XNOR functions on the transistor level", IEEE Journal of Solid-State Circuits, vol. 29, July 1994, pp. 780-786.

[76] Pasternak, J.H. and Salama, C.A.T., "Design of submicrometer CMOS differential pass-transistor logic circuits", IEEE Journal Solid-State Circuit, vol. 26, September 1991, pp. 1249-1258.

[77] Yano, K., Yamanaka, T., Nishida, T., Saito, M., Shimonigashi, K. and Shimizu, A., "A 3.8-ns CMOS 16×16-b multiplier using complementary pass-transistor logic", IEEE Journal Solid-State Circuit, vol. 25, April 1990, pp. 388-395.

[78] Santoro, M., "SPIM: a pipelined 64 × 64-bit iterative multiplier", IEEE Journal of Solid-State Circuits, vol. 24, April 1989, pp. 487-493.

[79] Nagamatsu, M., Tanaka, S., Mori, J., Hirano, K., Noguchi, T. and Hatanaka, K., "A 15-ns 32 × 32-b CMOS multiplier with an improved parallel structure", IEEE Journal of Solid-State Circuits, vol. 25, April 1990, pp. 494-497.

[80] Mori, J. et al., "A 10-ns 54 × 54 parallel structured full array multiplier with 0.5-µm CMOS technology", IEEE Journal of Solid-State Circuits, vol. 26, 1991, pp. 600-606.

[81] Goto, G, Sato, T., Nakajima, M. and Sukemura, T., "A 54 × 54 regularly structured tree multiplier", IEEE Journal of Solid-State Circuits, vol. 27, September 1992, pp. 1229-1236.

[82] Gerosa, G, Gary, S. and Dietz, C., "A 2.2W 80MHz superscalar RISC microprocessor", IEEE Journal of Solid-State Circuits, vol. 29, December 1994, pp. 1440-1454.

[83] Chandrakasan, A.P., Sheng, S. and Brodersen, E.W., “Low-power CMOS digital design”, IEEE Journal of Solid-State Corcuits, vol. 27, April 1992, pp. 473-484.

[84] Farnsworth, C., Edwards, D., Liu, J. and Sikand, S., “A hybrid asynchronous system design environment”, Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, May 1995.

[85] Veendrick, H.J.M., “Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits”, IEEE Journal of Solid-State Circuit, vol. 19, August 1984, pp. 468-473.

[86] User Manual, Compass Design Automation Inc., San Jose, U. S. A.

[87] Meng, T.H.Y., Brodersen, E.W and Messerschmitt, D.G., “Automatic synthesis of asynchronous circuits from high-level specifications”, IEEE Transactions on Computer-Aided Design, vol. 8, November 1989, pp. 1185-1204.

[88] Greenstreet, M.R. and Steiglitz, K., “Bubbles can make self-timed pipelines fast”, Journal of VLSI and Signal Processing, vol. 2, November 1990, pp. 139-148.

[89] Greenstreet, M.R., “STARI: A technique for high-bandwidth communication”, PhD thesis, Princeton University, 1993.

[90] Moon, C.W., “Synthesis and verification of asynchronous circuits from graphical specifications”, PhD thesis, Unversity of California at Berkeley, 1992.

[91] Martin, A.L., “Synthesis of asynchronous VLSI circuits”, Technical Report, TR-93-28, Computer Science Department, California Institute of Technology, 1993.

[92] Berkel, K.V., “Beware the isochronic fork”, Integration, the VLSI Journal, vol. 13, June 1992, pp. 103-128.

[93] Garside, J.D., “Micropipeline structures”, ACiD-WG.EXACT Workshop on Asynchronous Data Processing, the Netherlands, December 1992.

[94] Bainbridge, W.J. and Furber, S.B., “MARBLE: a proposed asynchronous system level macrocell bus”, Proceedings of the Second UK Asynchronous Forum, England, July 1997.

[95] ARM Architecture Reference Manual, Prentice Hall, Advanced RISC Machines Ltd. (ARM), 1996.

[96] Johnson, M., “Superscalar microprocessor design”, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[97] Dobberpuhl, D.W. et al.,
"A 200- MHz 64-b dual-issue CMOS microprocessor",
IEEE Journal of Solid-State Circuit,
vol. 27, November 1992, pp. 1555-1565.

[98] Liu, D. and Svensson, C., "Power consumption estimation in CMOS VLSI chips", IEEE Journal of Solid-State Circuits, vol. 29, June 1994, pp. 663-670.

[99] Liu, D. and Svensson, C., "Trading speed for low power by choice of supply and threshold voltages", IEEE Journal of Solid-State Circuits, vol. 28, January 1993, pp. 10-17.

[100] Nielsen, L.S., Niessen, C., Sparsø, J. and Berkel, K.V.,
"Low-power operation using self-timed circuits and adaptive scaling of the supply voltage", IEEE Transaction on VLSI Systems, vol. 2, December 1994, pp. 391-397.

[101] Burd, T., "Low-power CMOS library design methodology",
MSc thesis, University of California, Berkeley, 1994.

[102] Berkel, K.V., Burgess, R., Kessels, J.L.W., Peeters, A., Roncken, M. and Schalij, F., "A fully asynchronous low power error corrector for the DCC player", IEEE Journal of Solid-State Circuits,
vol. 29, December 1994, pp. 1429-1239.

[103] Chandrakasan, A.P., Burstein, A. and Broderson, R.W.,
"A low-power chipset for a portable multimedic I/O terminal",
IEEE Journal of Solid-State Circuits,
vol. 29, December 1994, pp. 1415-1428.

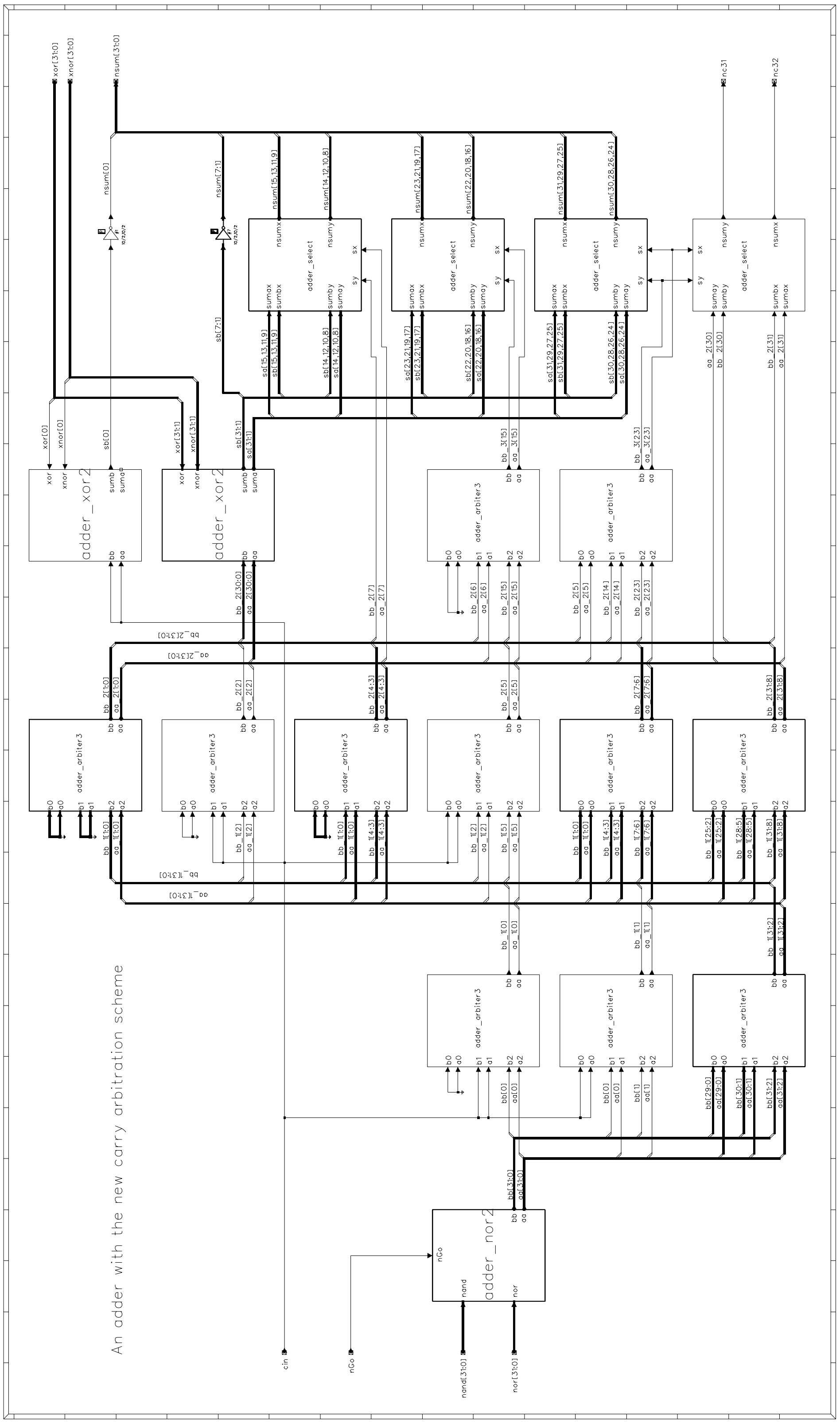
[104] "Cogency pushes asynchronous logic",
Microprocessor Report, vol. 11, October 1997.

Adder schematics A

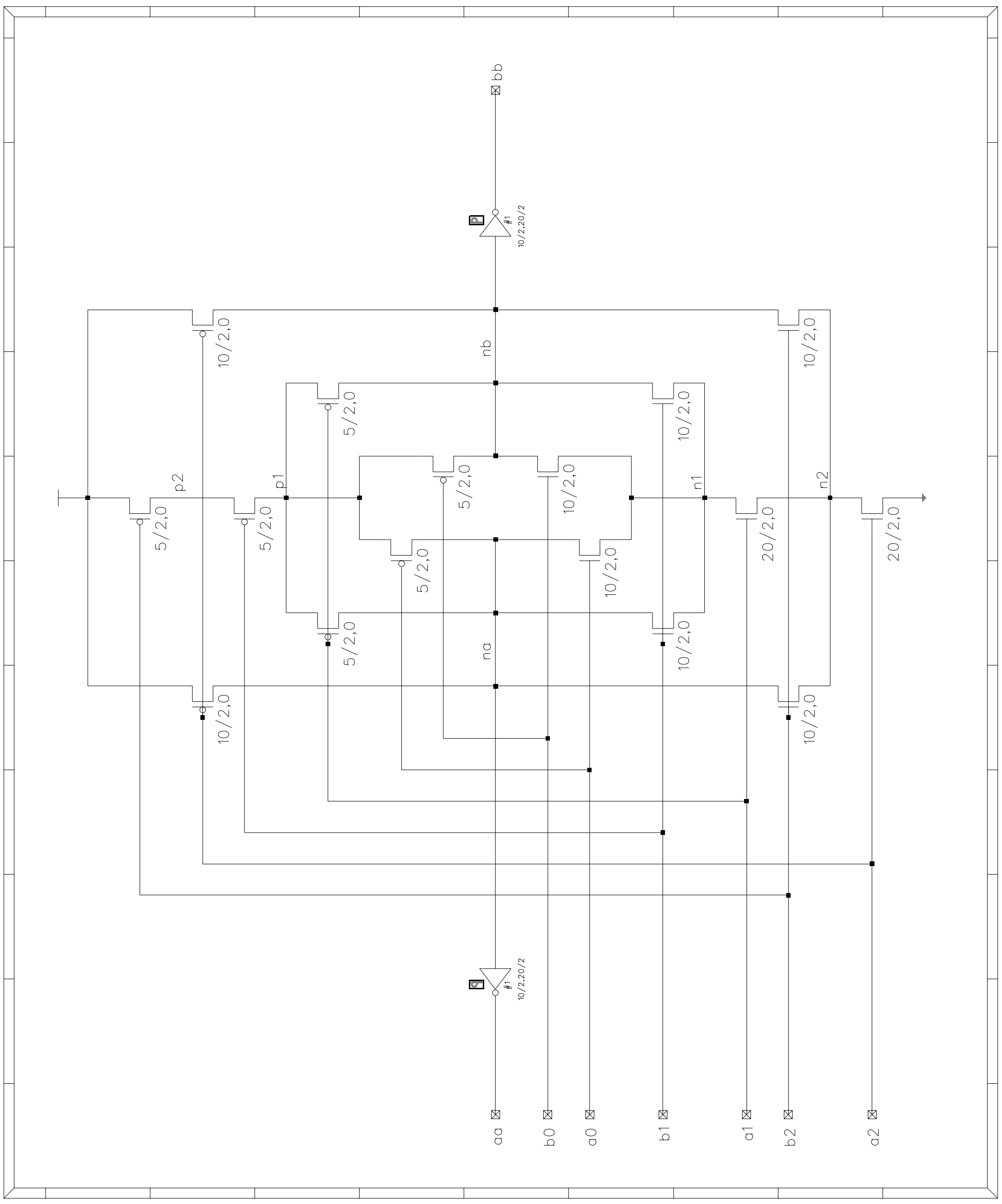
This appendix contains the schematics of some of the cell library for the AMULET3 adder. Below is a list:

- ❏ **adder_datapath**
- ❏ **adder_arbiter3**
- ❏ **adder_nor2**
- ❏ **adder_xor2**
- ❏ **adder_select**

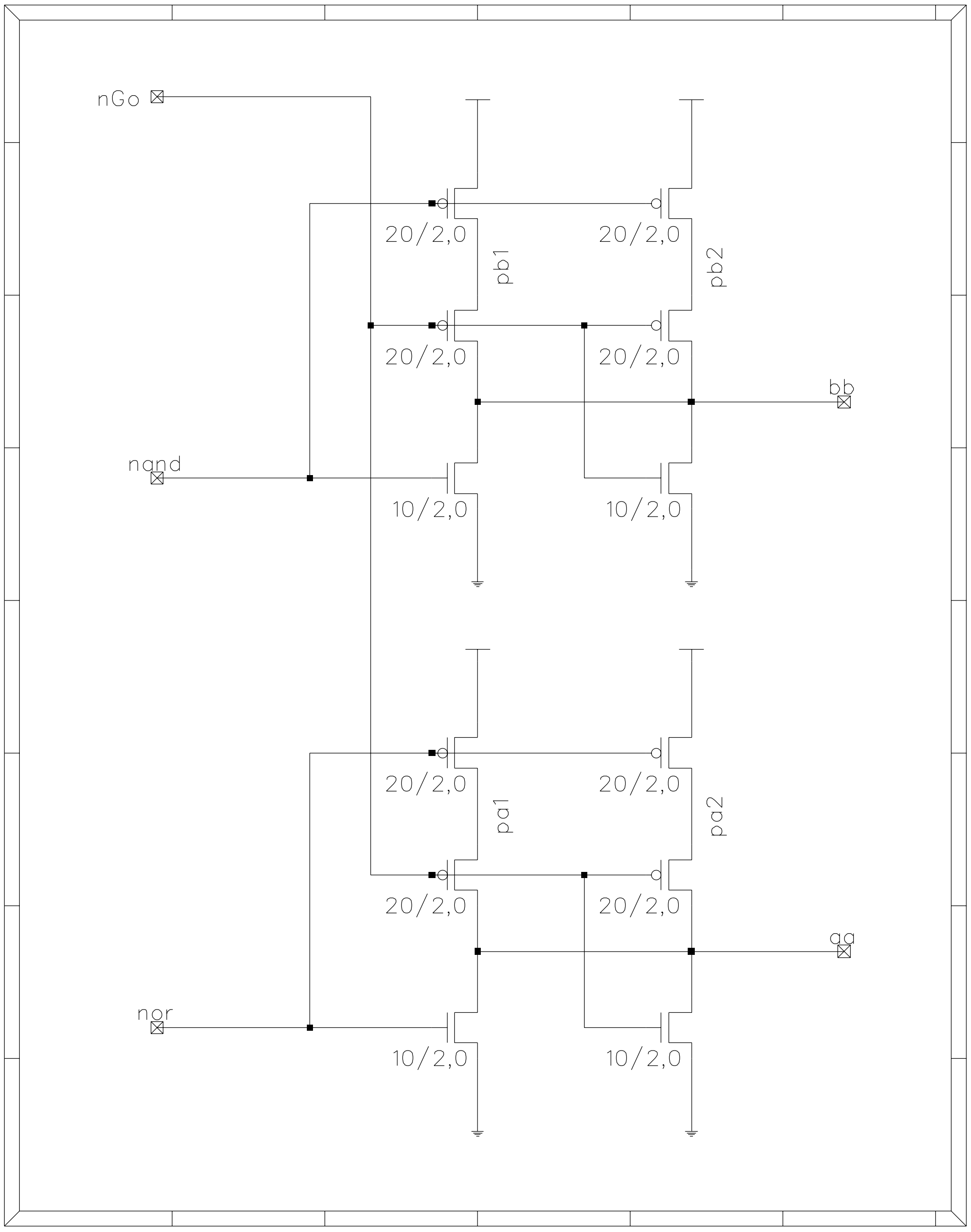
A.1 adder_datapath



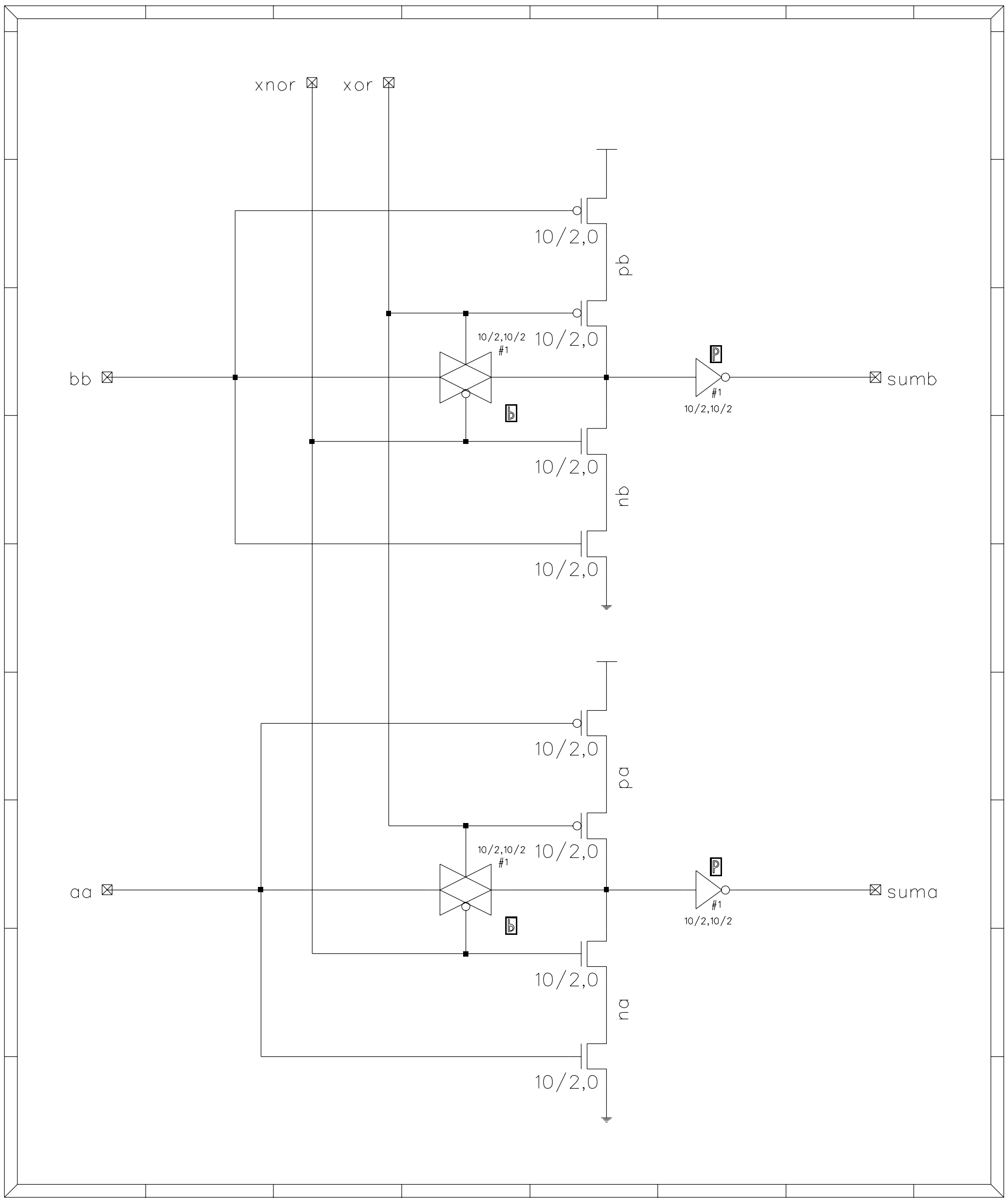
A.2 adder_arbiter3



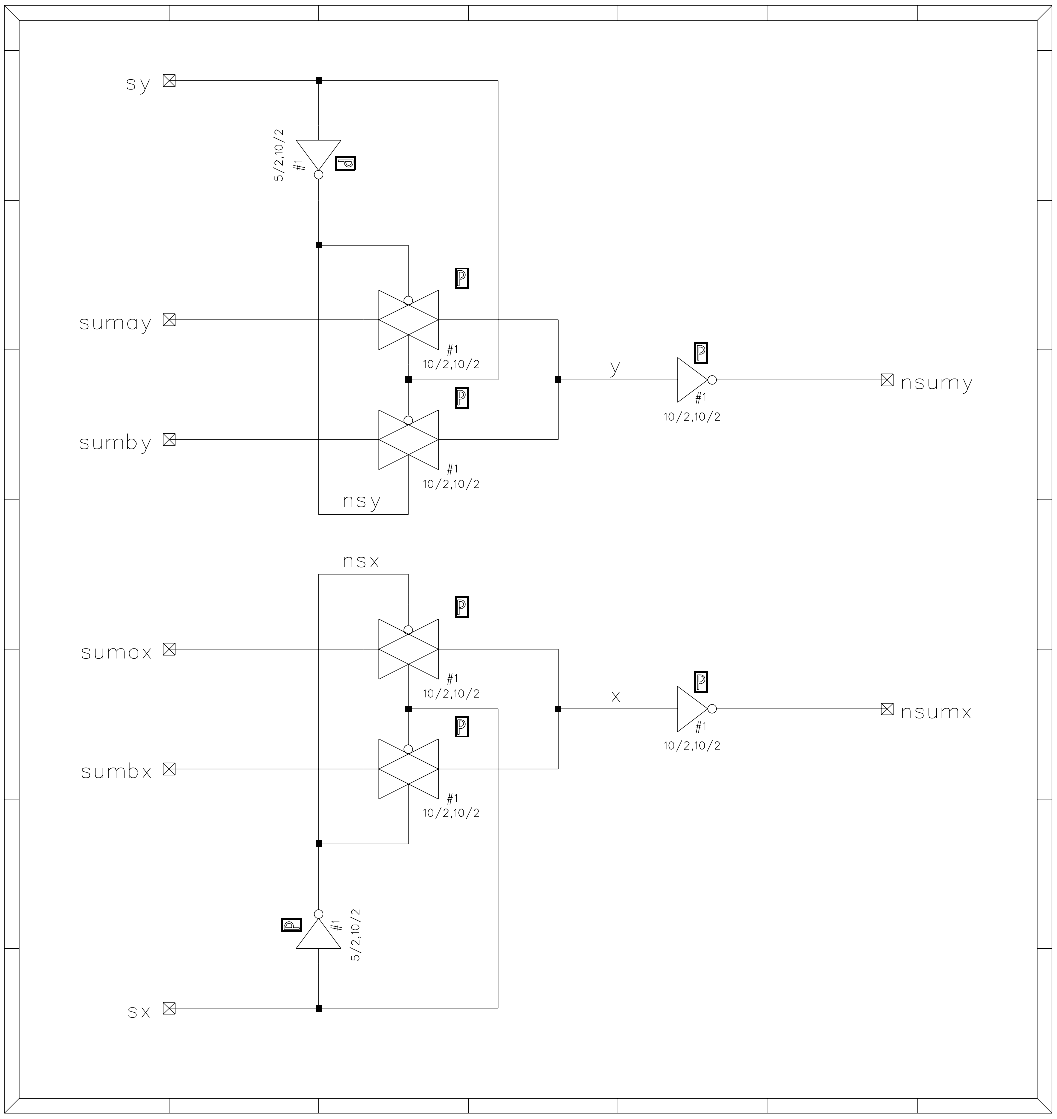
A.3 adder_nor2



A.4 adder_xor2



A.5 adder_select



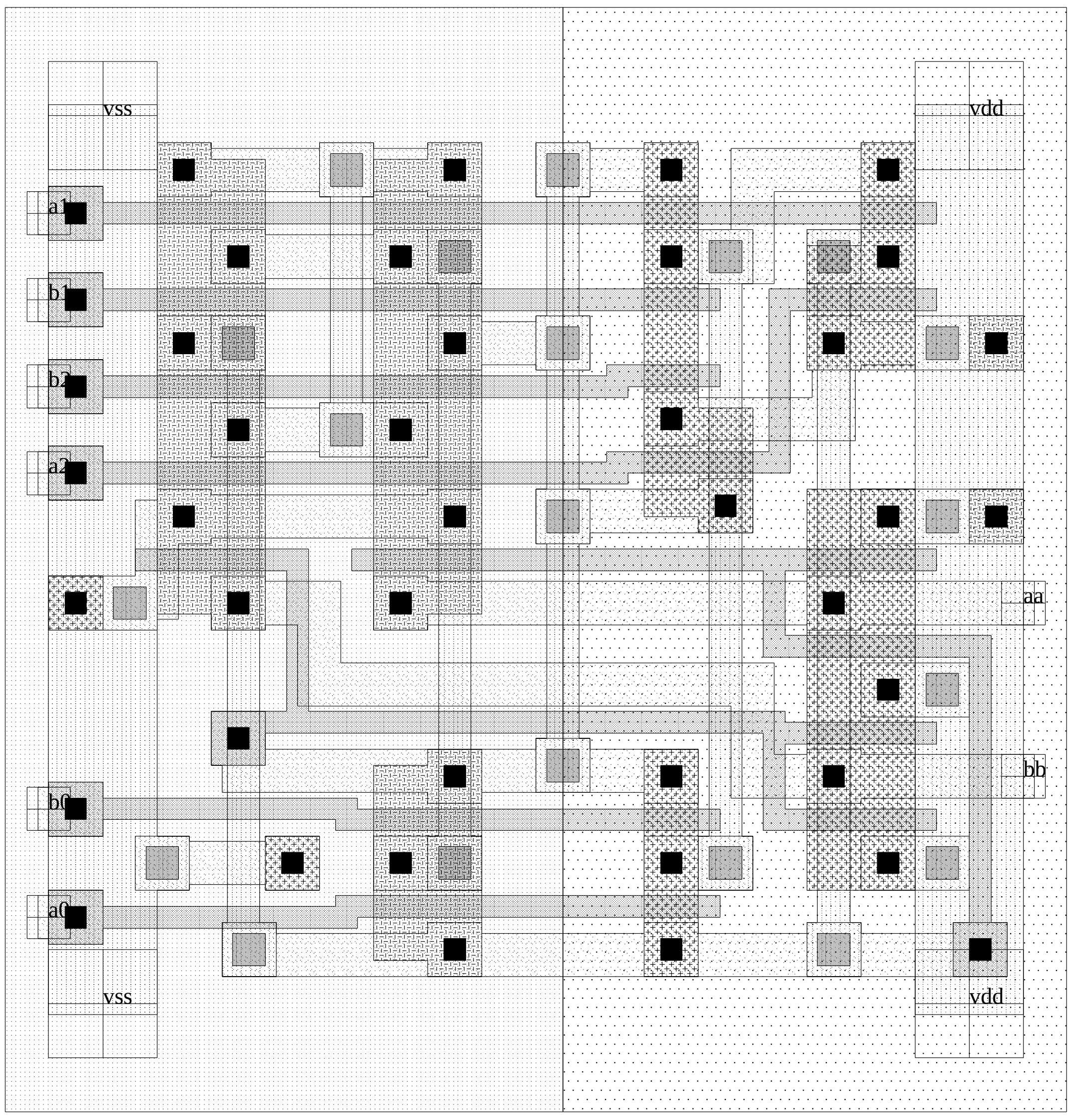
Adder layouts B

This appendix contains the layouts of some of the cell library for the AMULET3 adder.

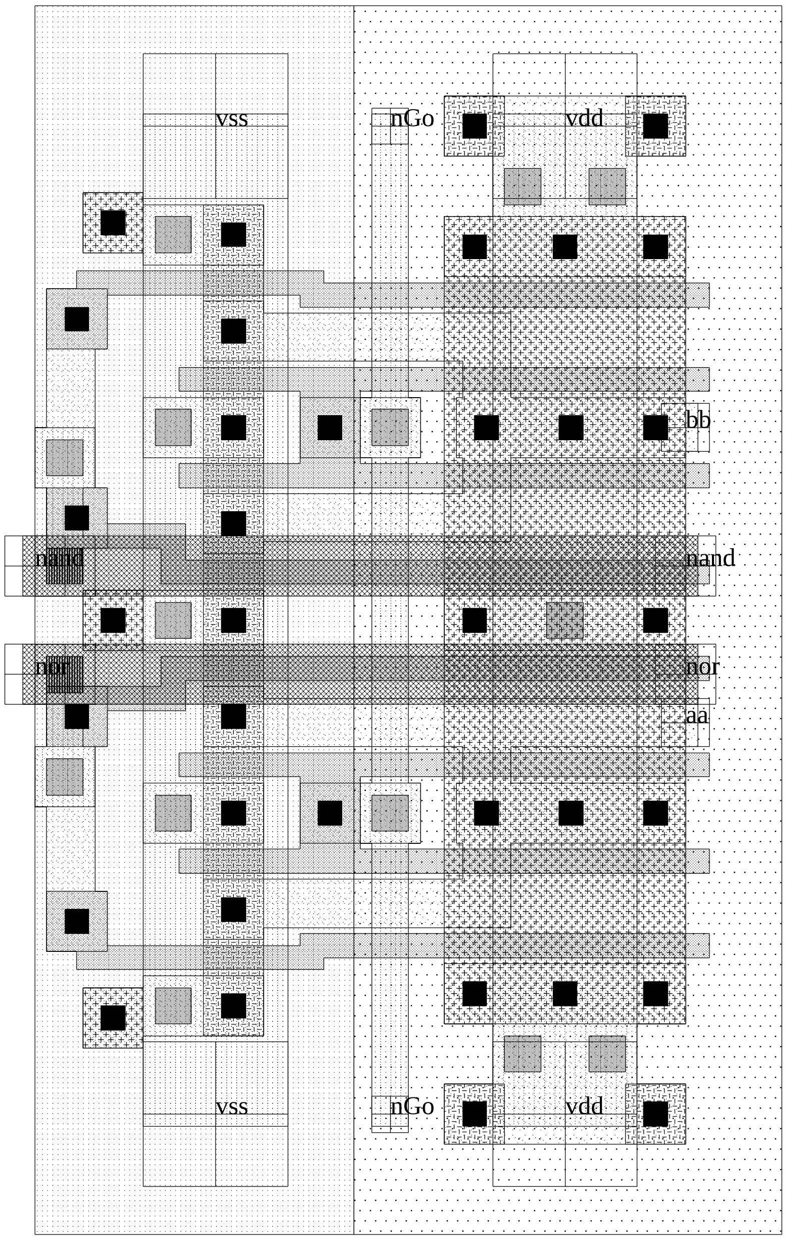
Below is a list:

- **adder_arbiter3**
- **adder_nor2**
- **adder_xor2**
- **adder_select**

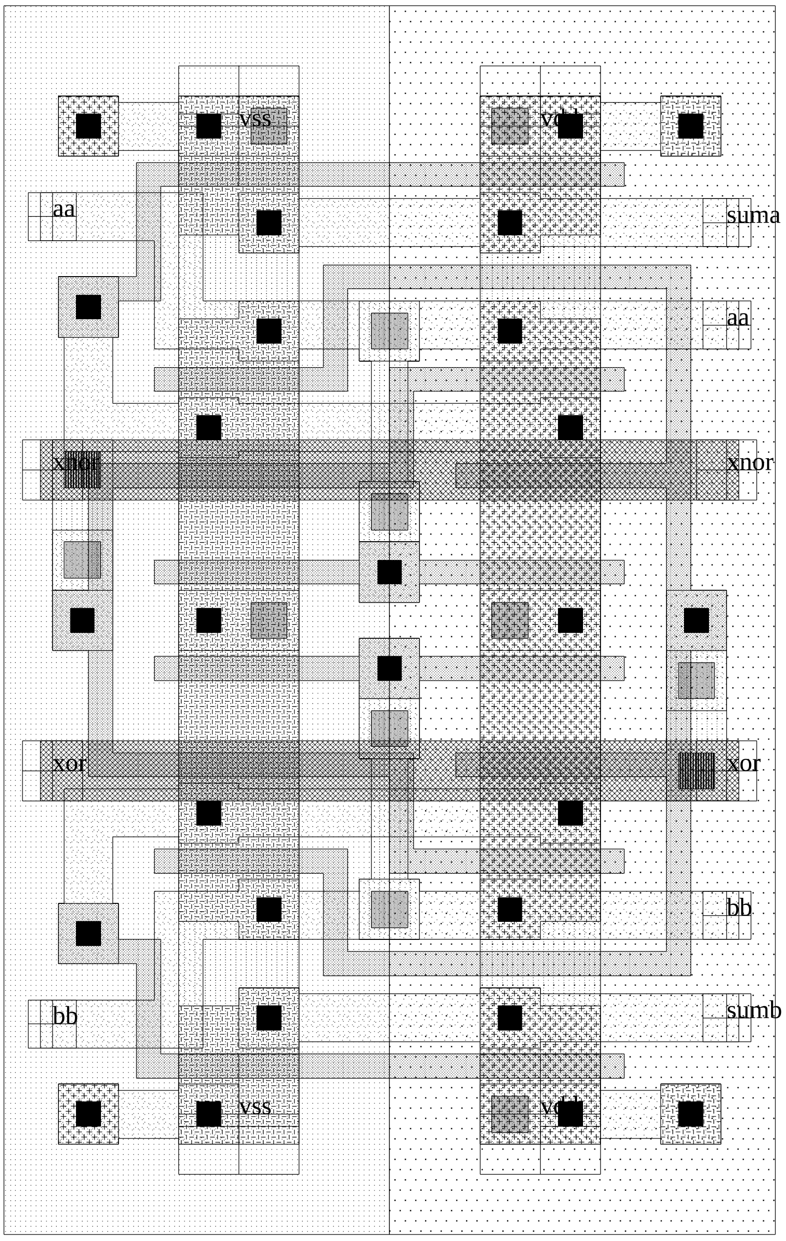
B.1 adder_arbiter3



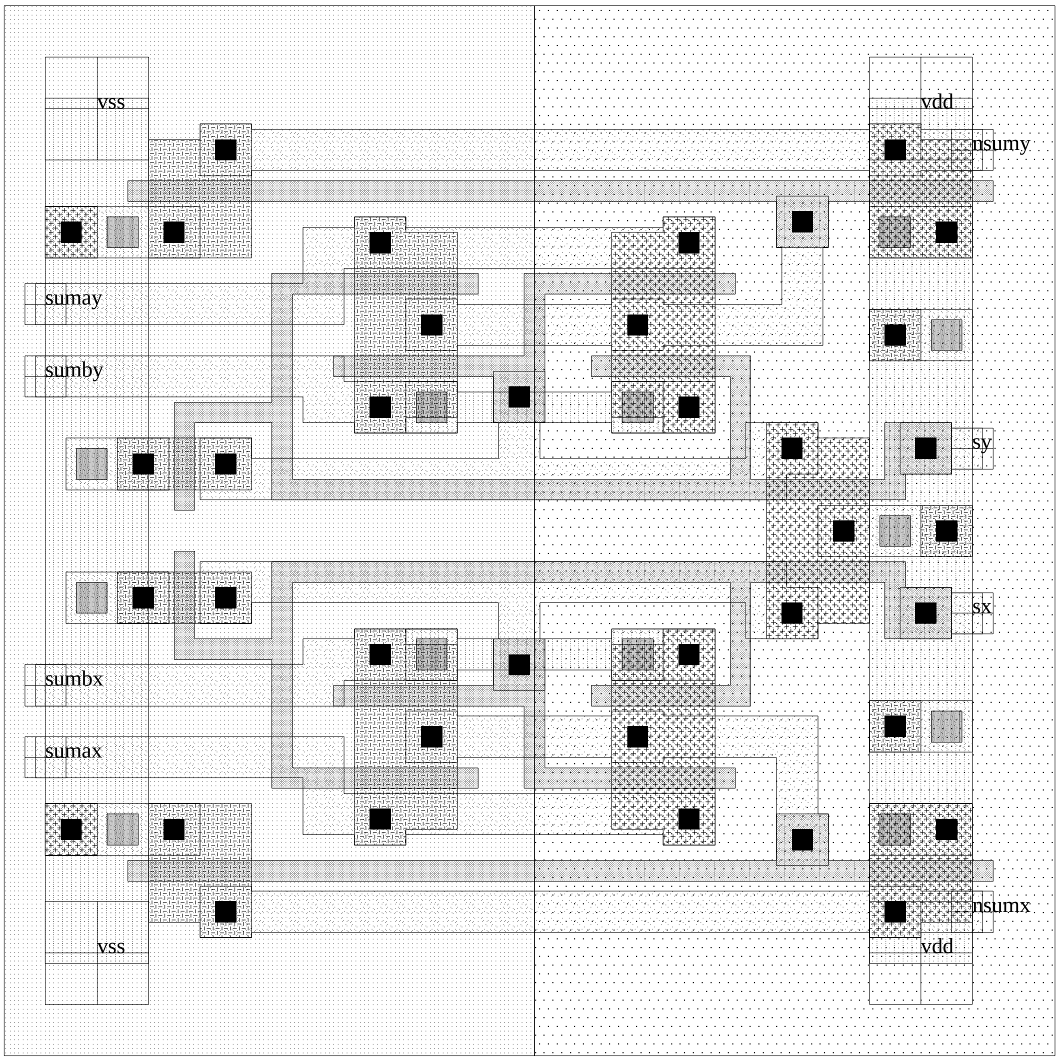
B.2 adder_nor2



B.3 adder_xor2



B.4 adder_select

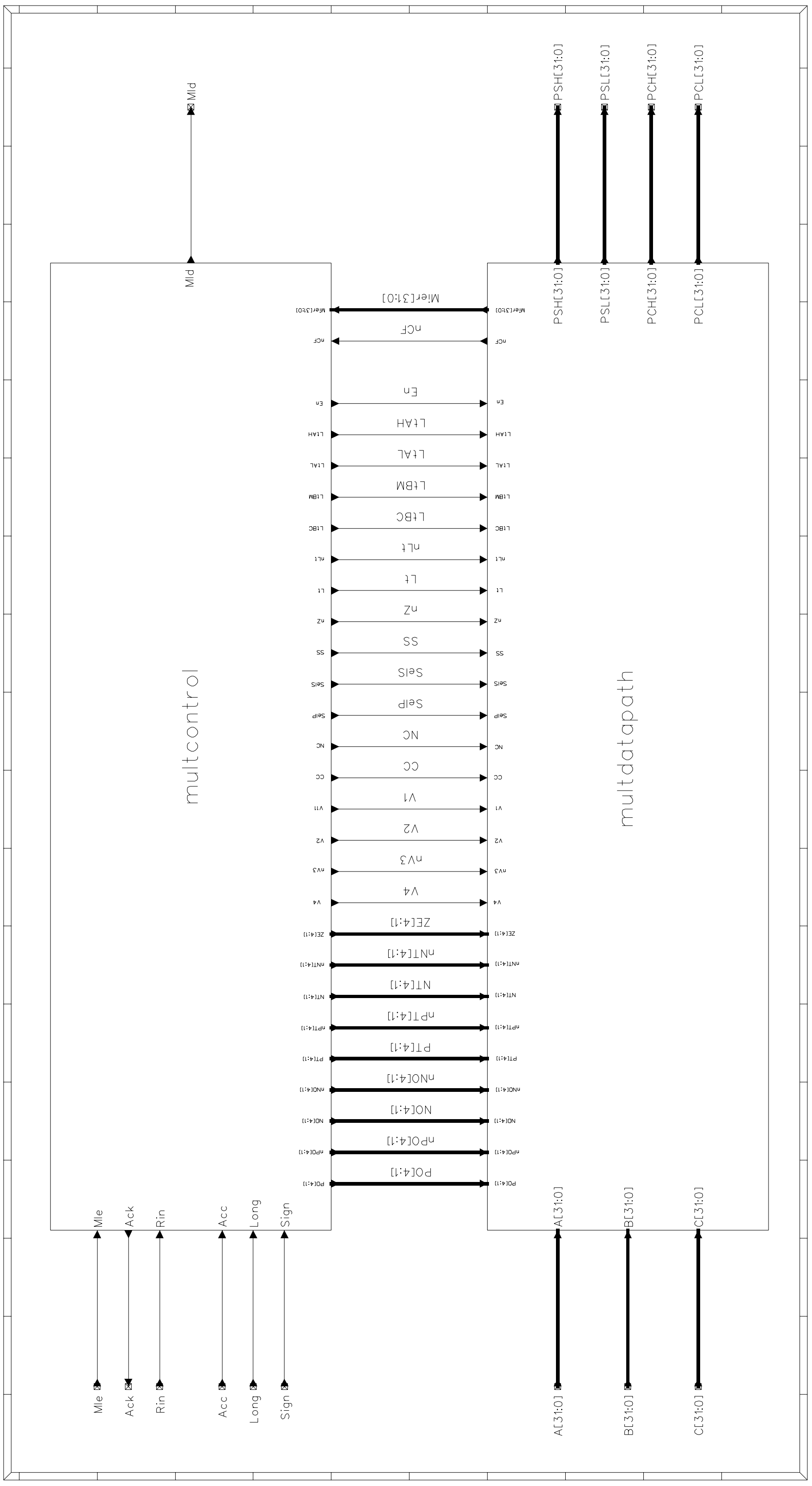


Multiplier schematics C

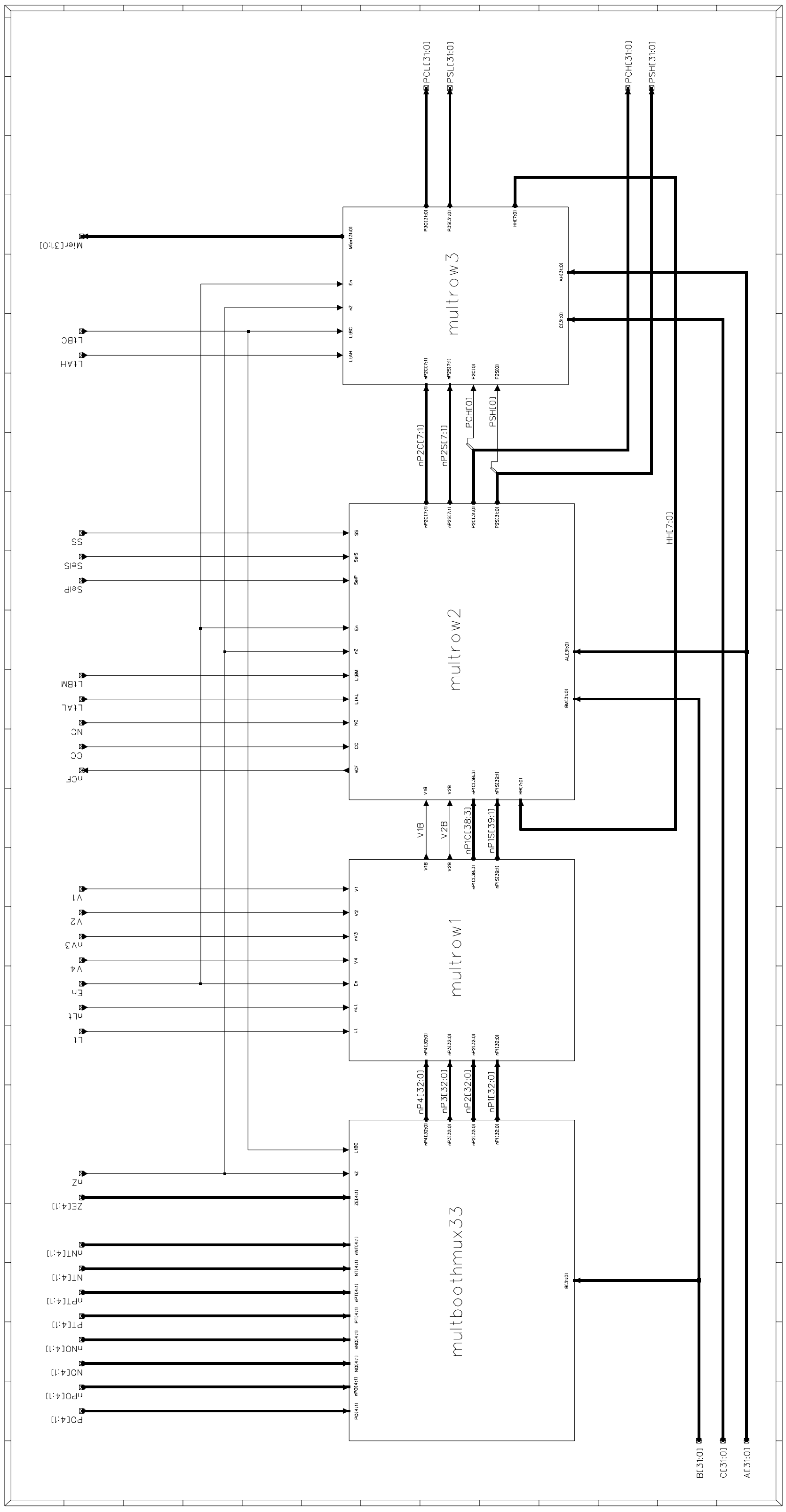
This appendix contains the schematics of some of the cell library for the AMULET3 multiplier. Below is a list of the following appendix sections:

- ❏ **AMULET3_Multiplier**
- ❏ **multdatapath**
- ❏ **multboothmux33**
- ❏ **multrow1**
- ❏ **multrow2**
- ❏ **multrow3**
- ❏ **multboothmux**
- ❏ **multcnt42e**
- ❏ **multcnt42c**
- ❏ **multmuxe**
- ❏ **multlatch**
- ❏ **multdffa**
- ❏ **multdffb**
- ❏ **multdffc**

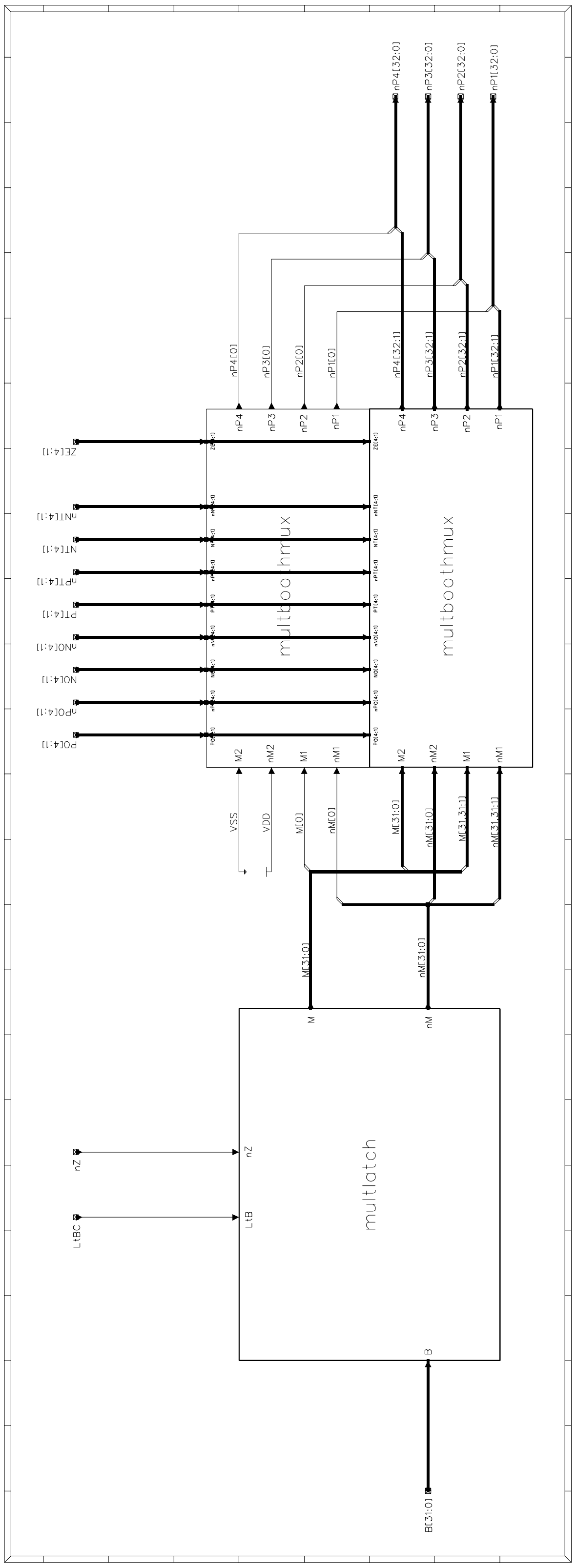
C.1 AMULET3 Multiplier



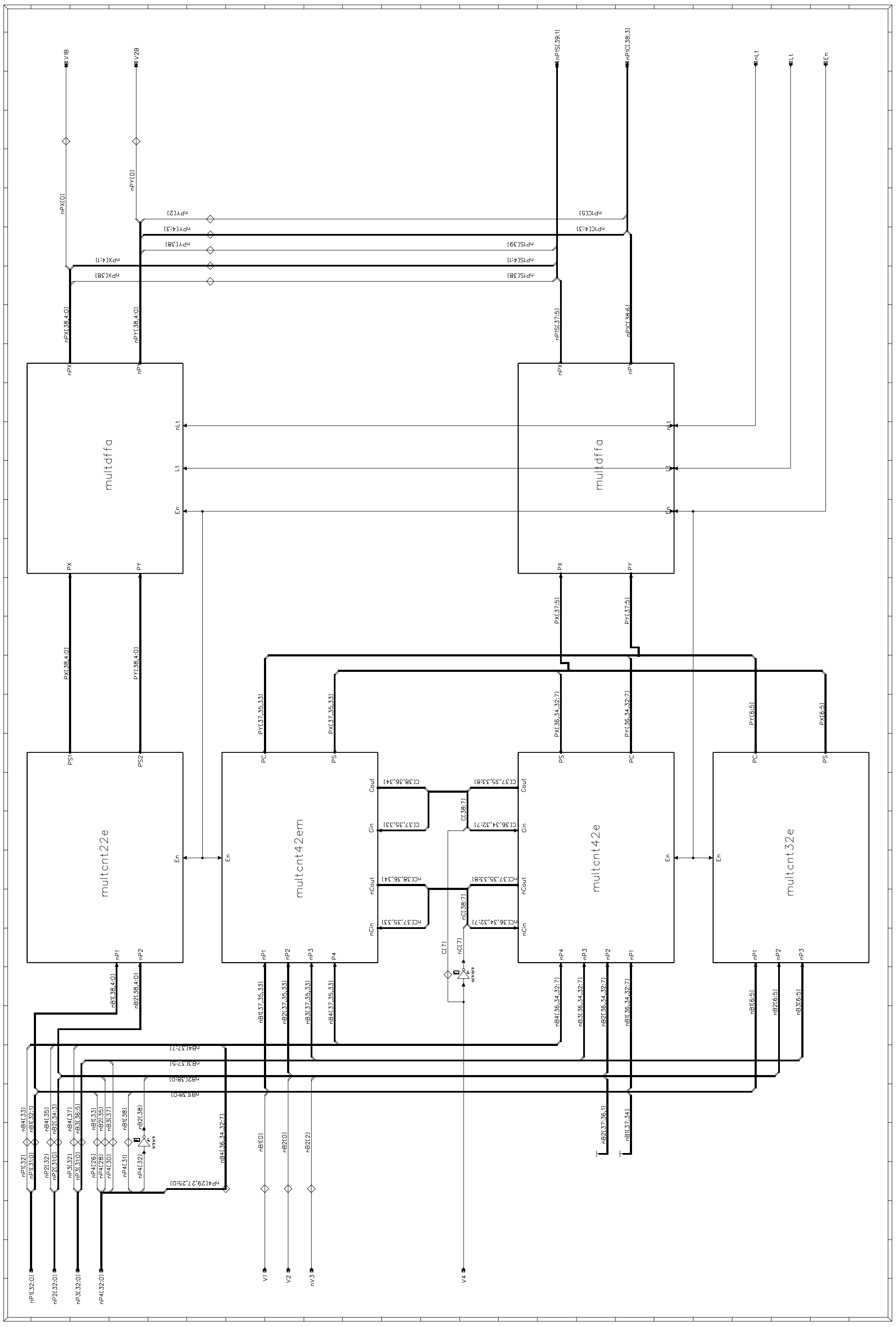
C.2 multdatapath



C.3 multboothmux33

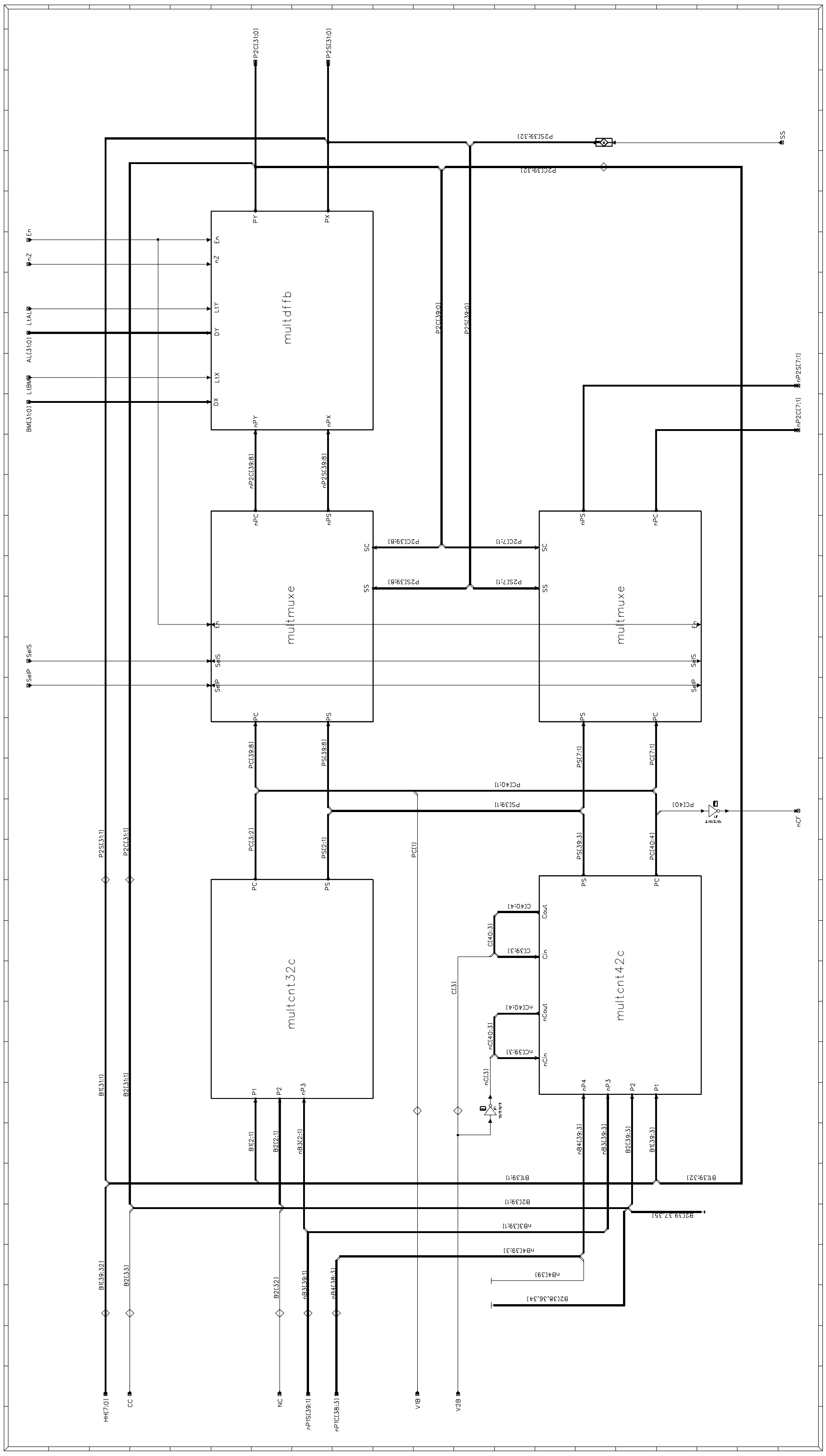


C.4 multrow1

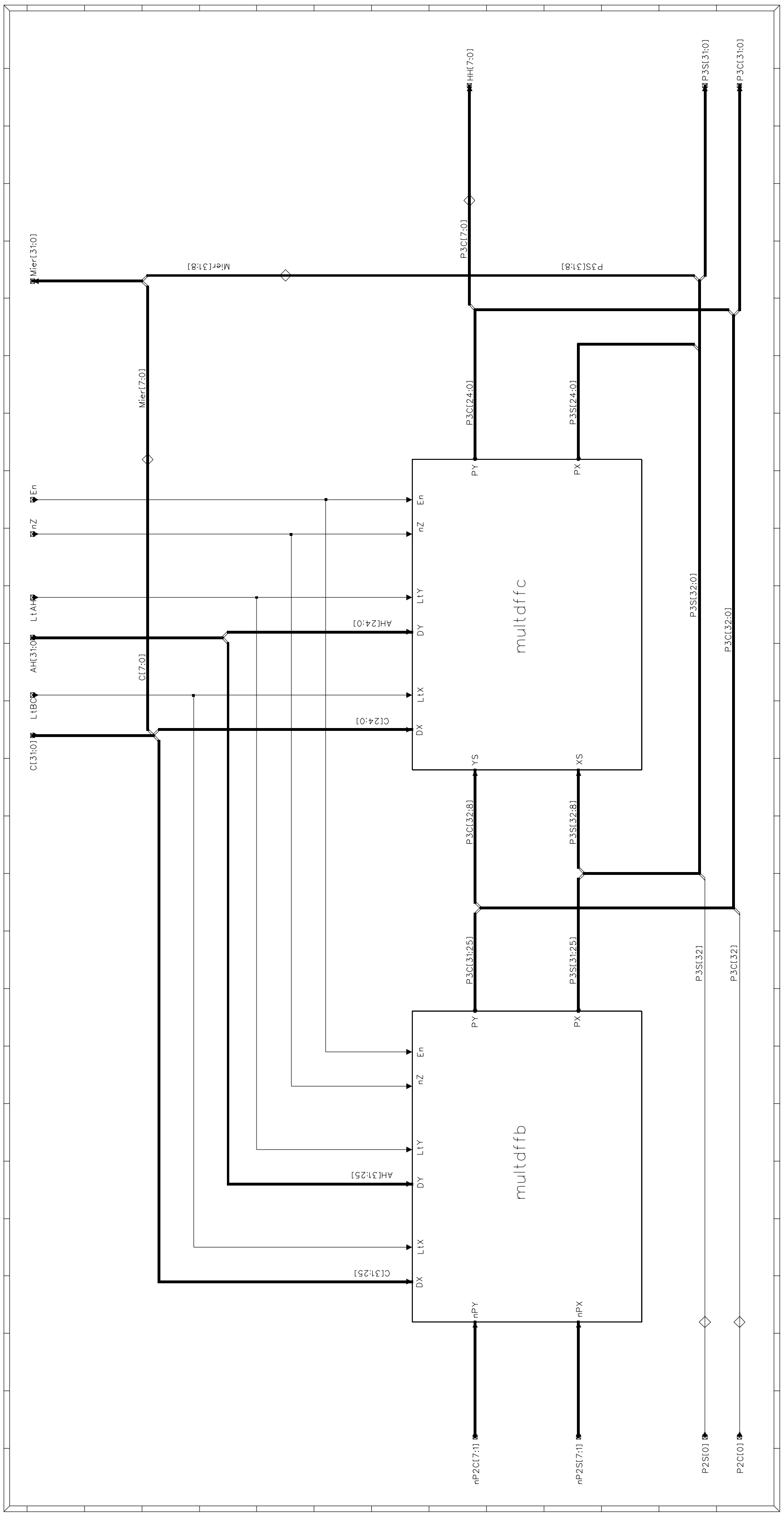


multdffa
multdffa
multcnt22e
multcnt42em
multcnt42e
multcnt32e

C.5 multrow2

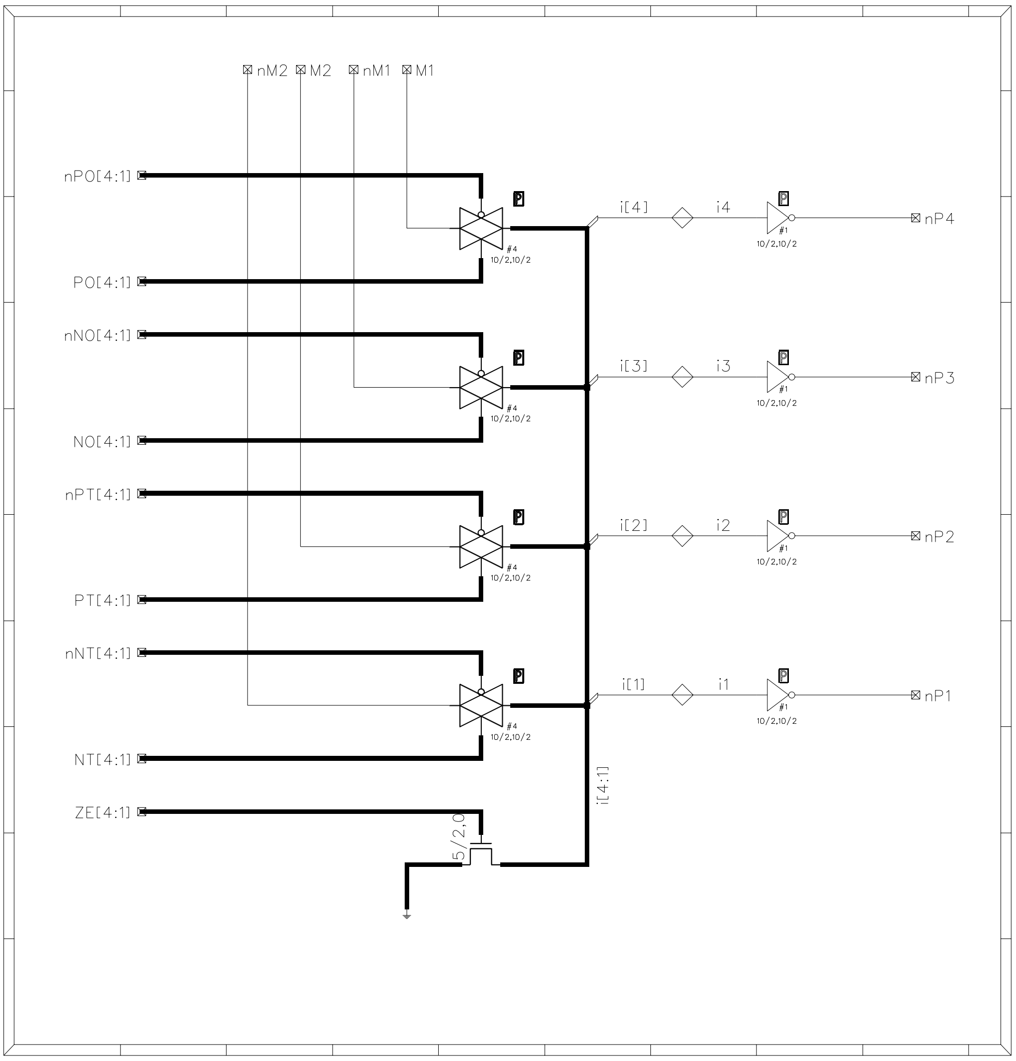


C.6 multrow3

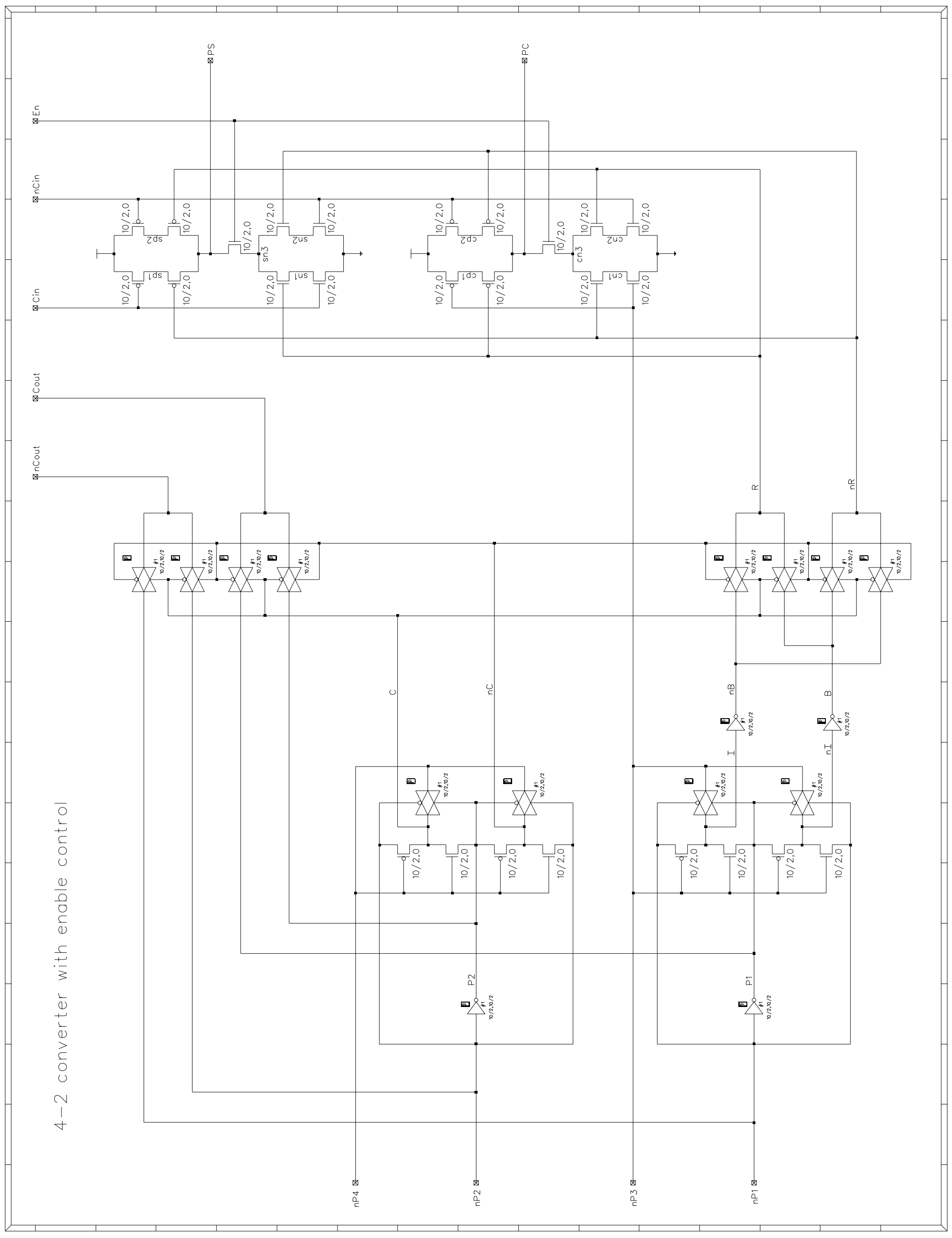


multdffc
multdffb
Mier[31:0]
Mier[31:8]
Mier[7:0]
En
nZ
LtAH
AH[31:0]
LtBC
C[31:0]
C[7:0]
AH[24:0]
C[24:0]
AH[31:25]
C[31:25]
HH[7:0]
P3C[7:0]
P3S[31:8]
P3S[31:0]
P3C[31:0]
P3C[24:0]
P3S[24:0]
P3S[32:0]
P3C[32:0]
P3C[32:8]
P3S[32:8]
P3C[31:25]
P3S[31:25]
P3S[32]
P3C[32]
nP2C[7:1]
nP2S[7:1]
P2S[0]
P2C[0]
PY
PX
LtY
DY
LtX
DX
YS
XS
nPY
nPX

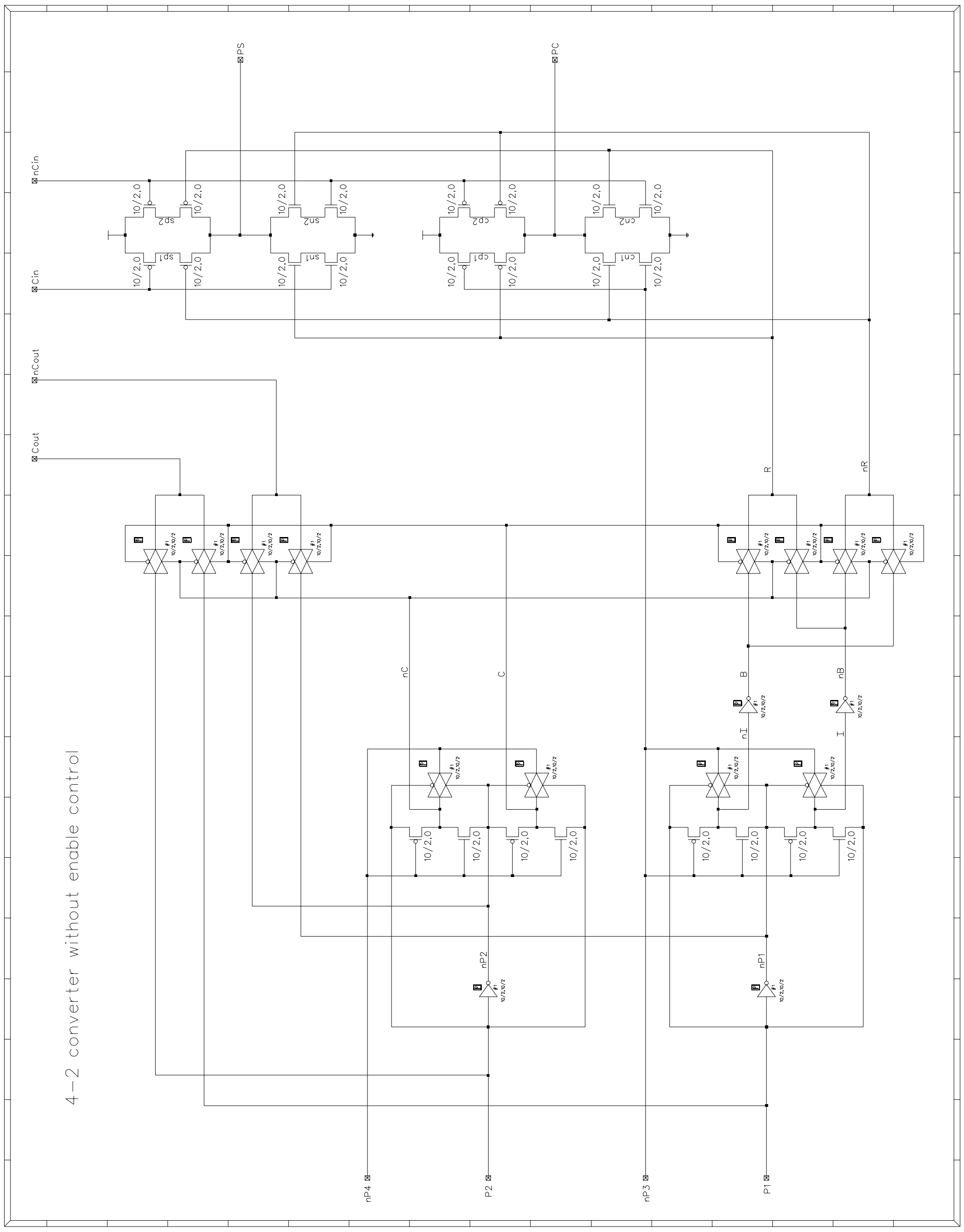
C.7 multboothmux



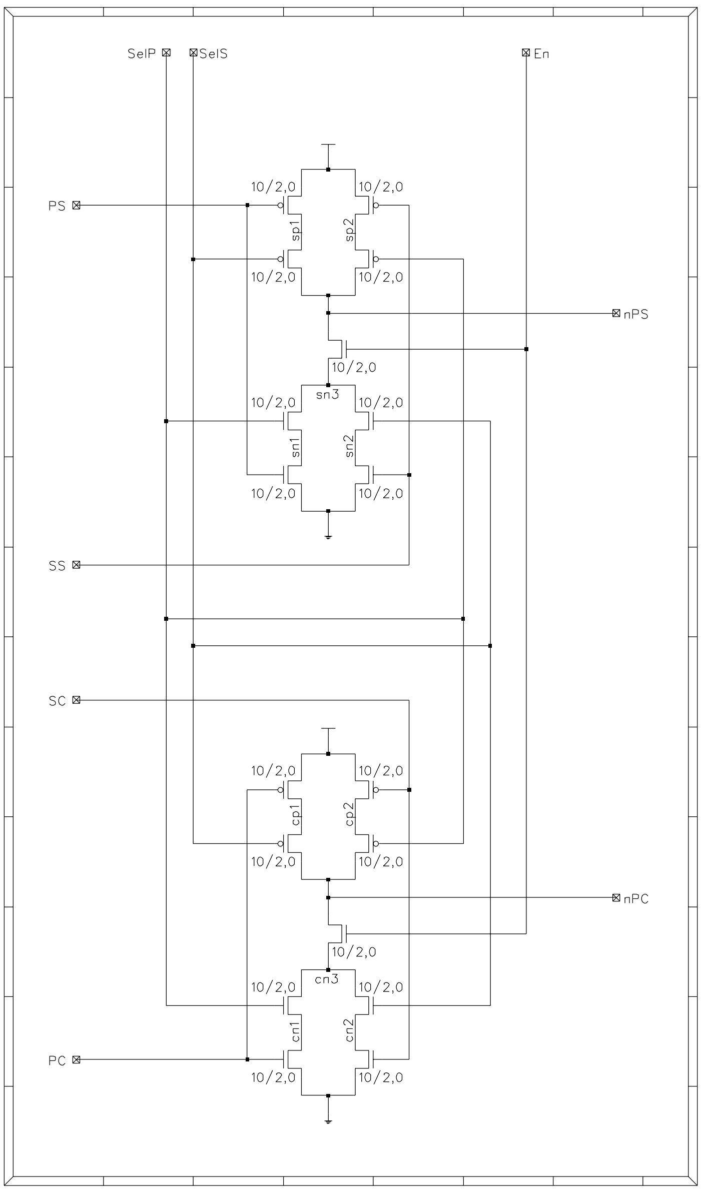
C.8 multcnt42e



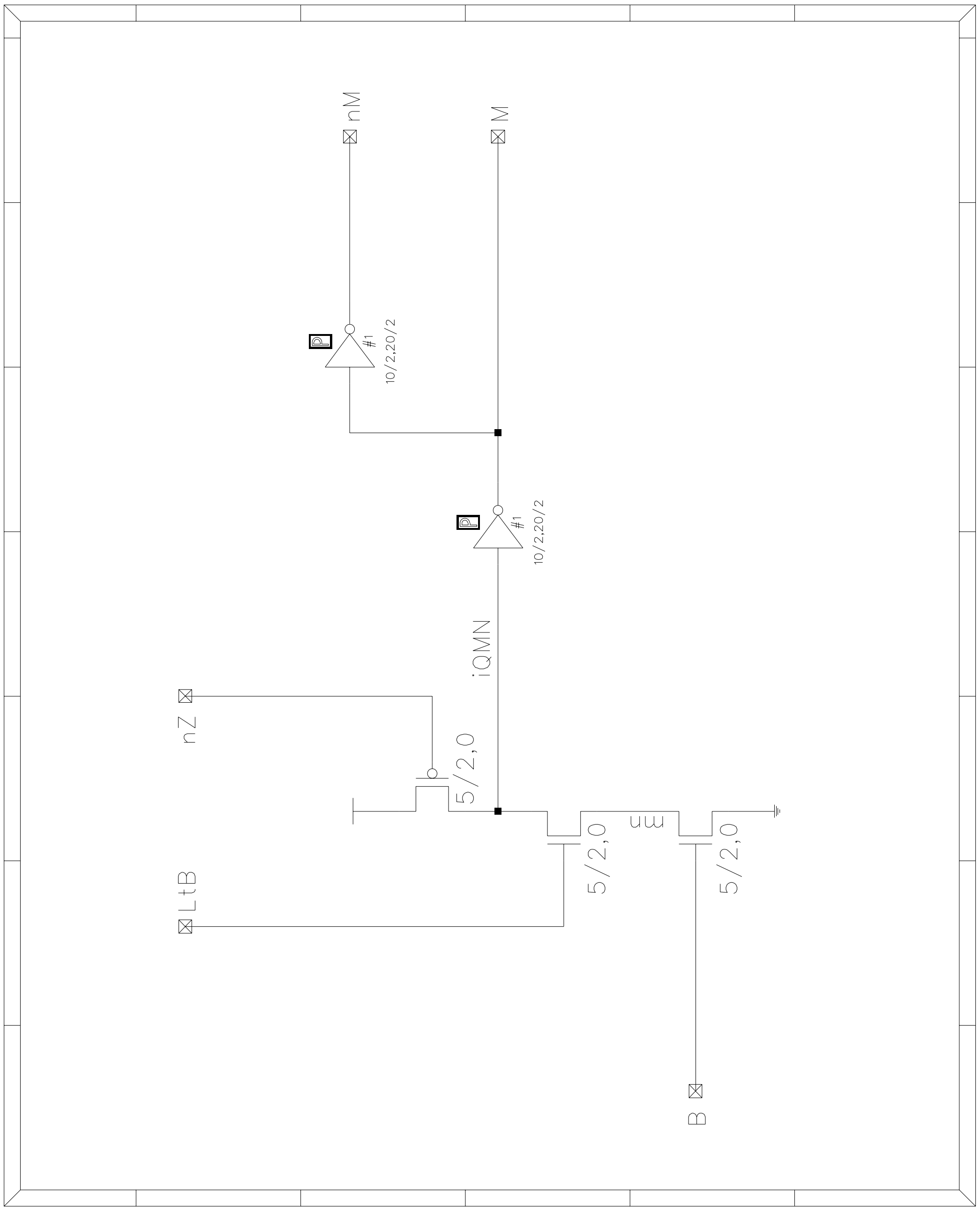
C.9 multcnt42c



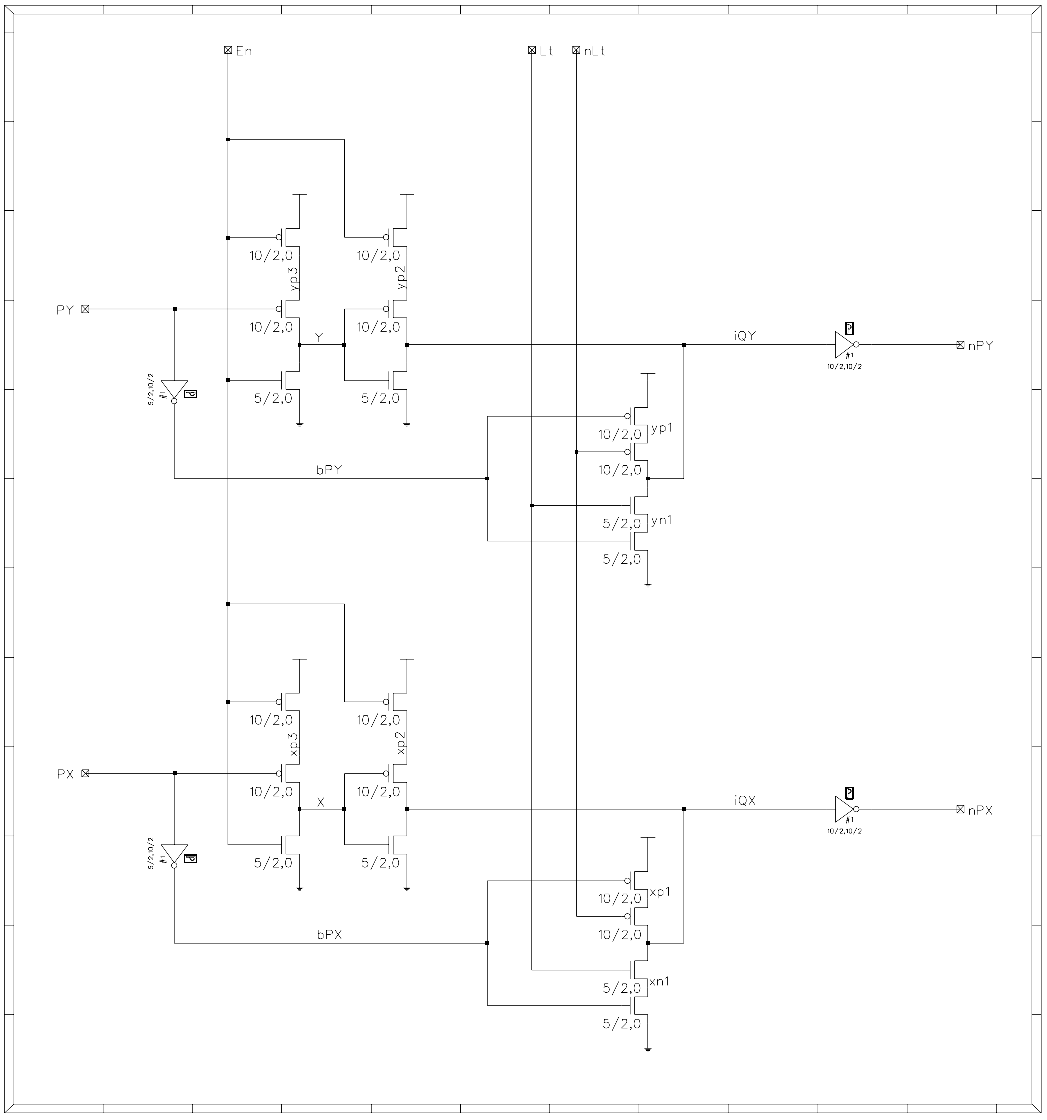
C.10 multmuxe



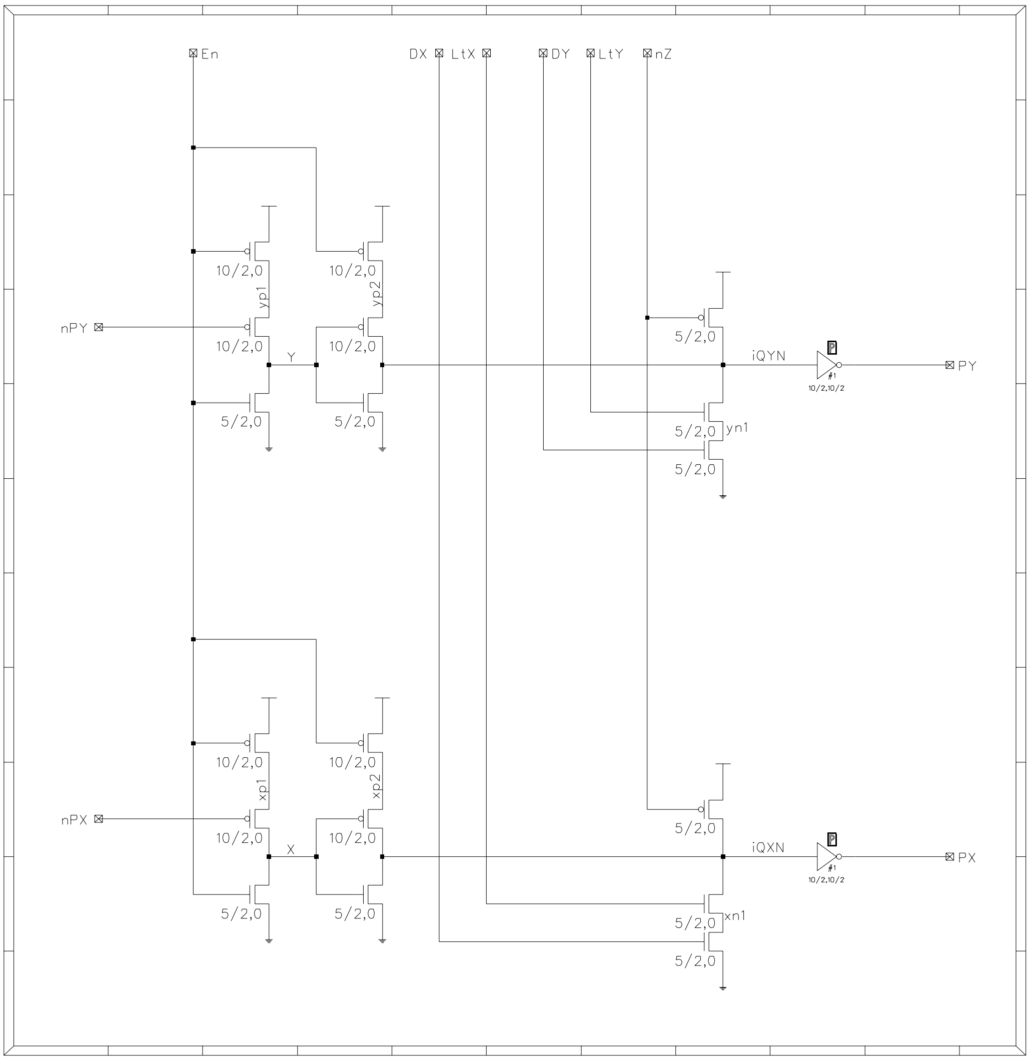
C.11 multlatch



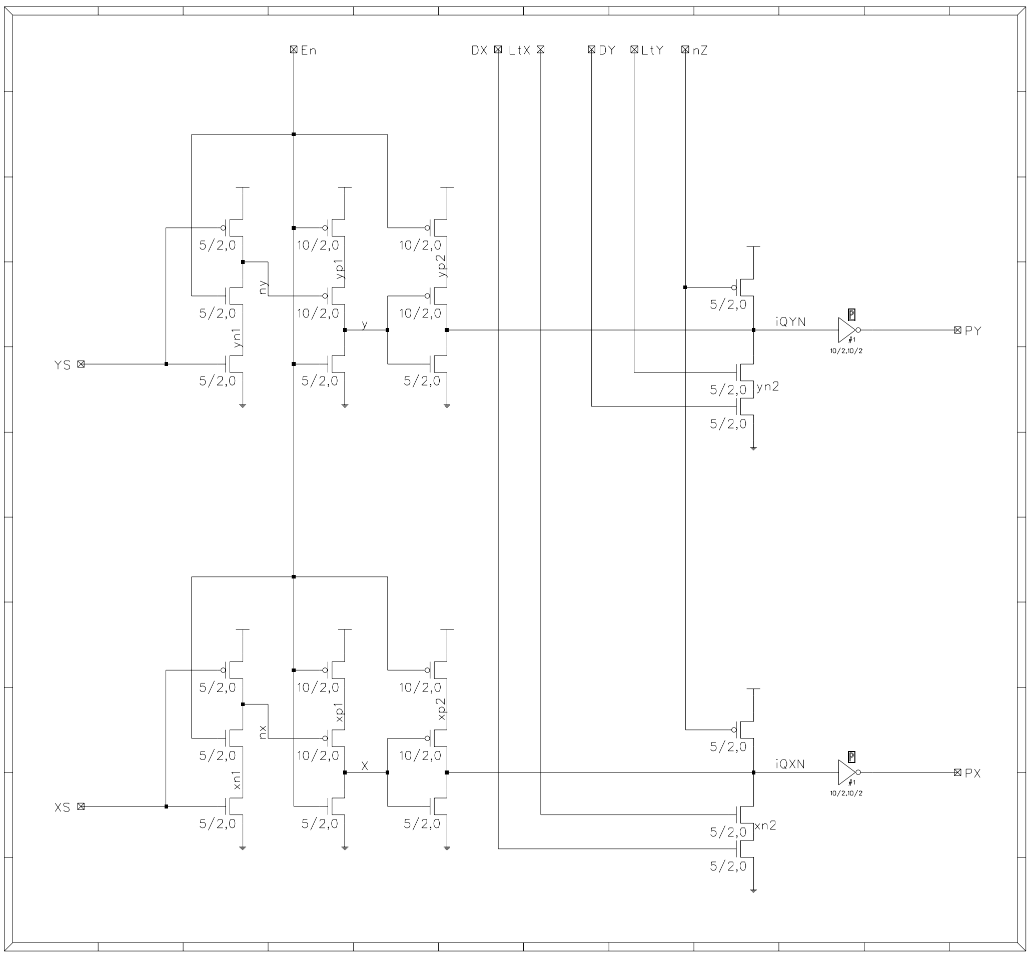
C.12 multdffa



C.13 multdffb



C.14 multdffc

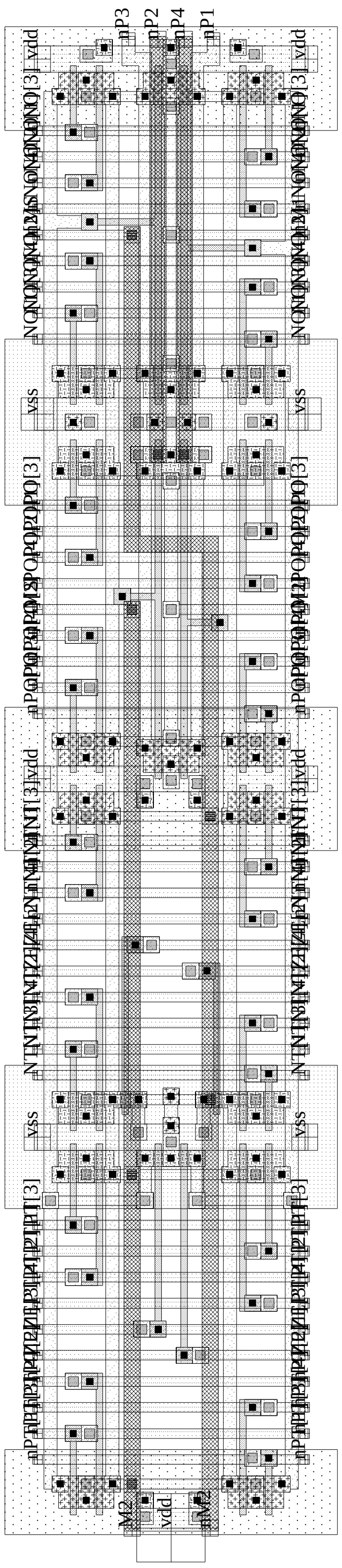


Multiplier layouts D

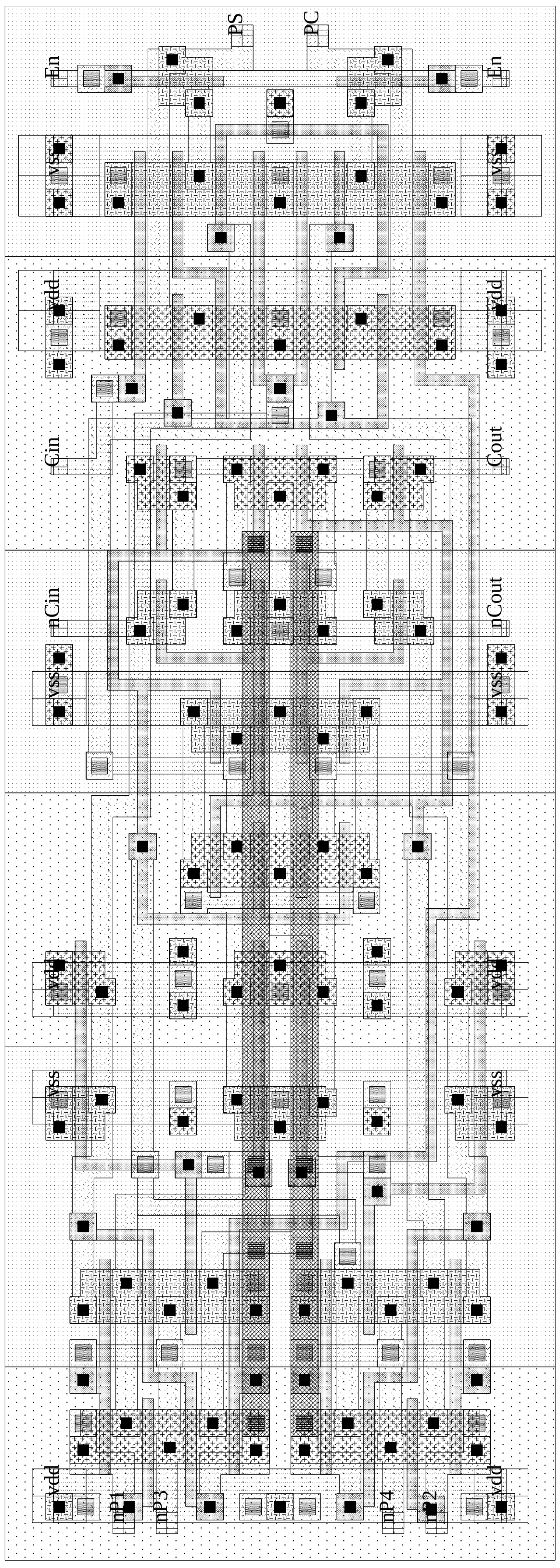
This appendix contains the layouts of some of the cell library for the AMULET3 multiplier. Below is a list:

- ❑ **multboothmux**
- ❑ **multcnt42e**
- ❑ **multcnt42c**
- ❑ **multmuxe**
- ❑ **multlatch**
- ❑ **multdffa**
- ❑ **multdffb**
- ❑ **multdffc**

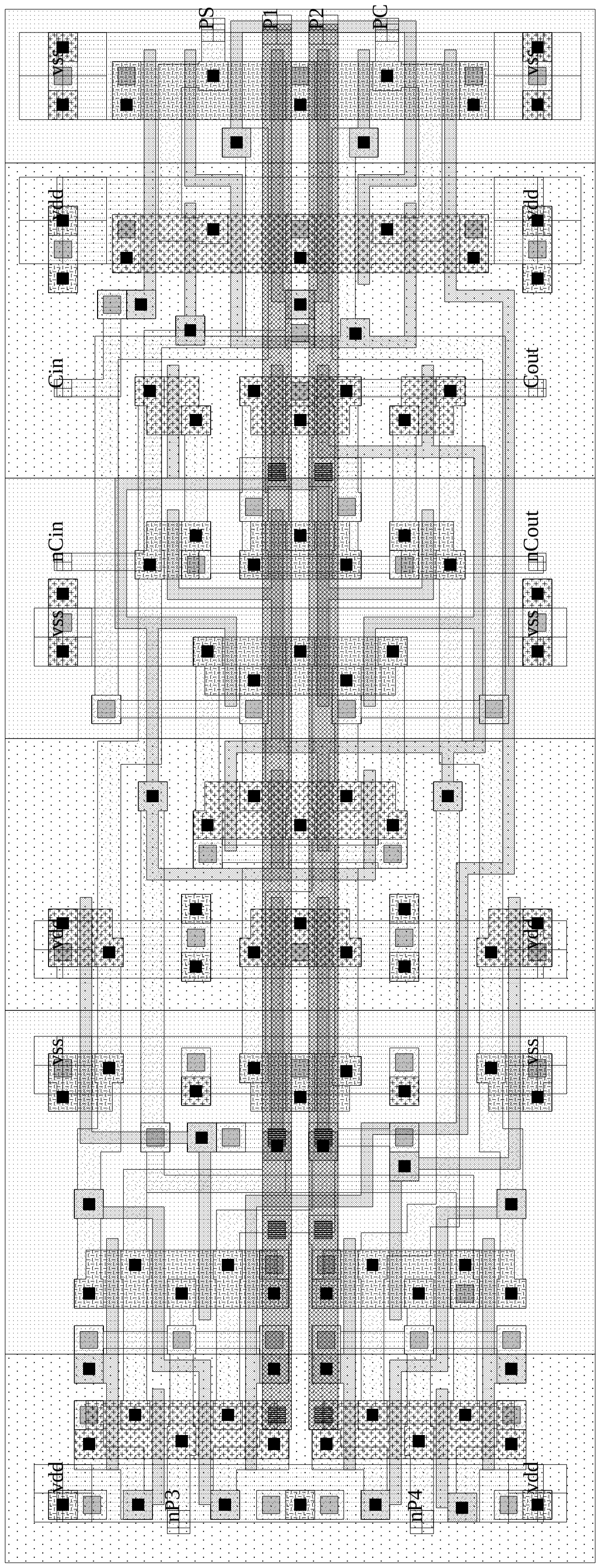
D.1 multboothmux



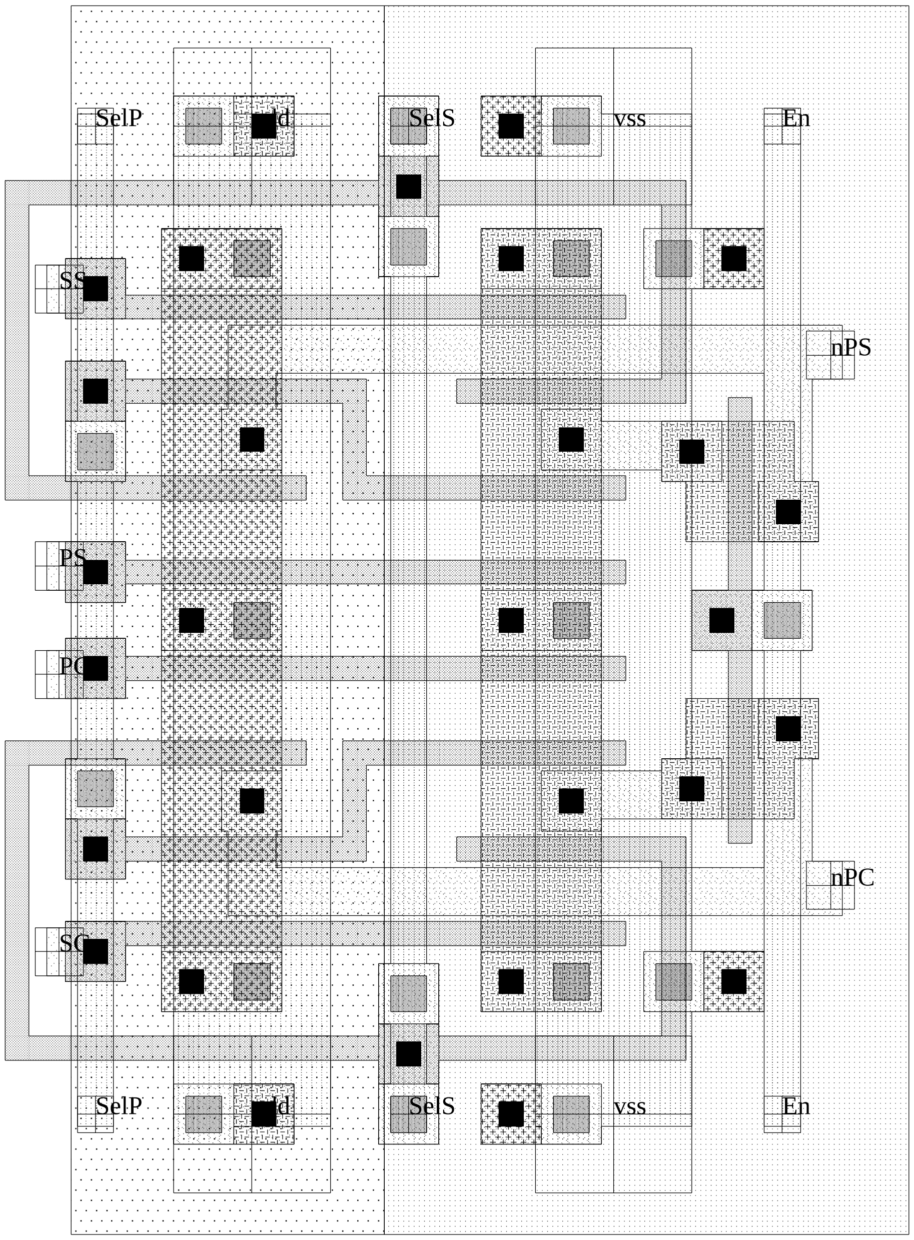
D.2 multcnt42e



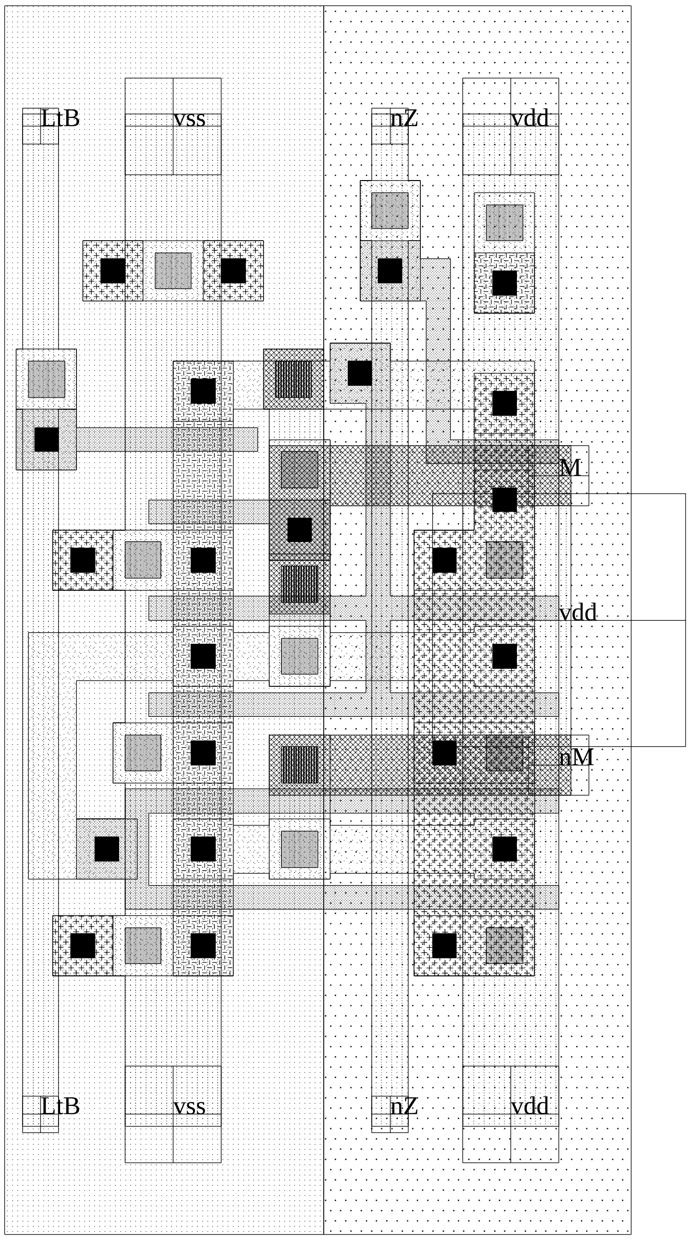
D.3 multcnt42c



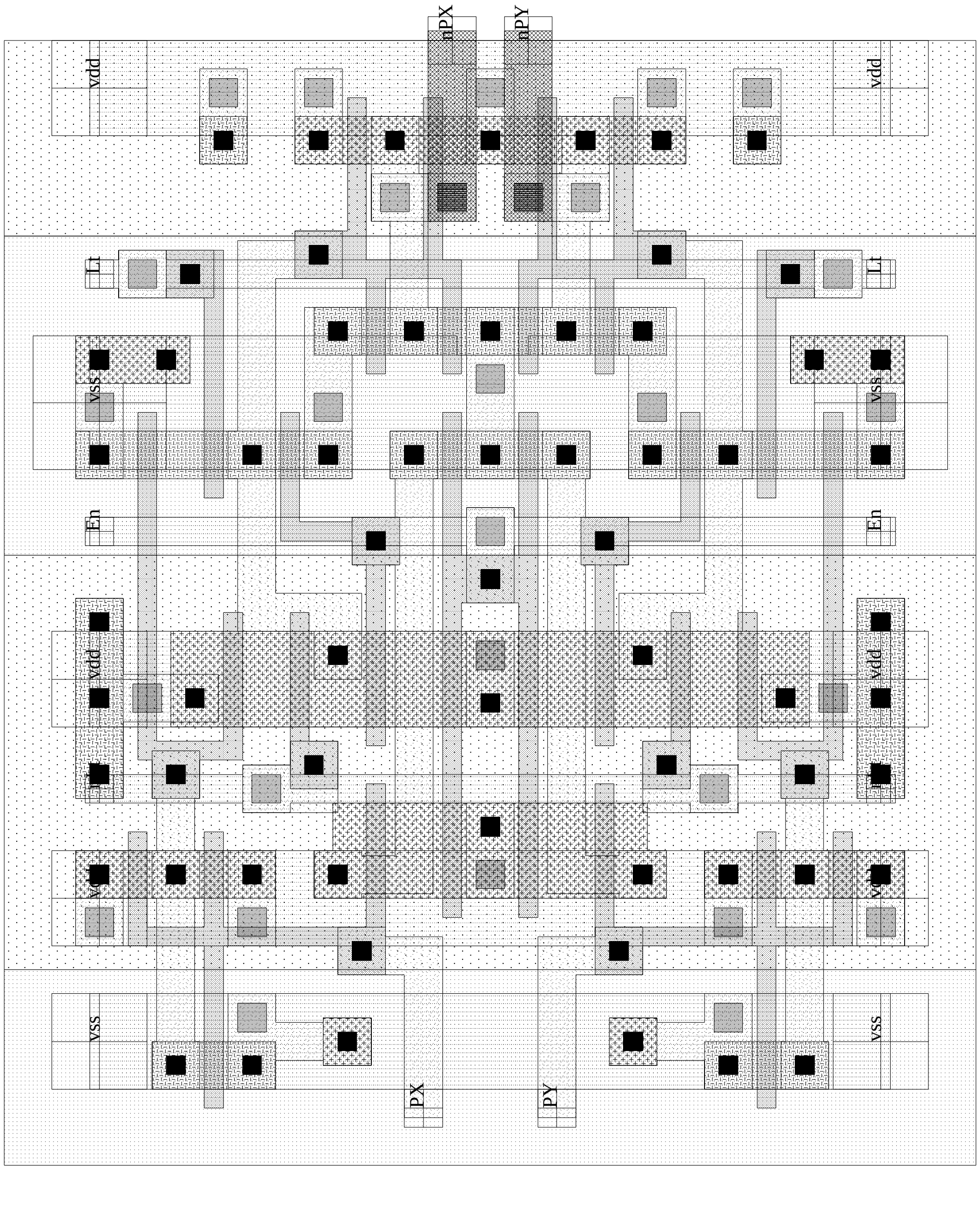
D.4 multmuxe



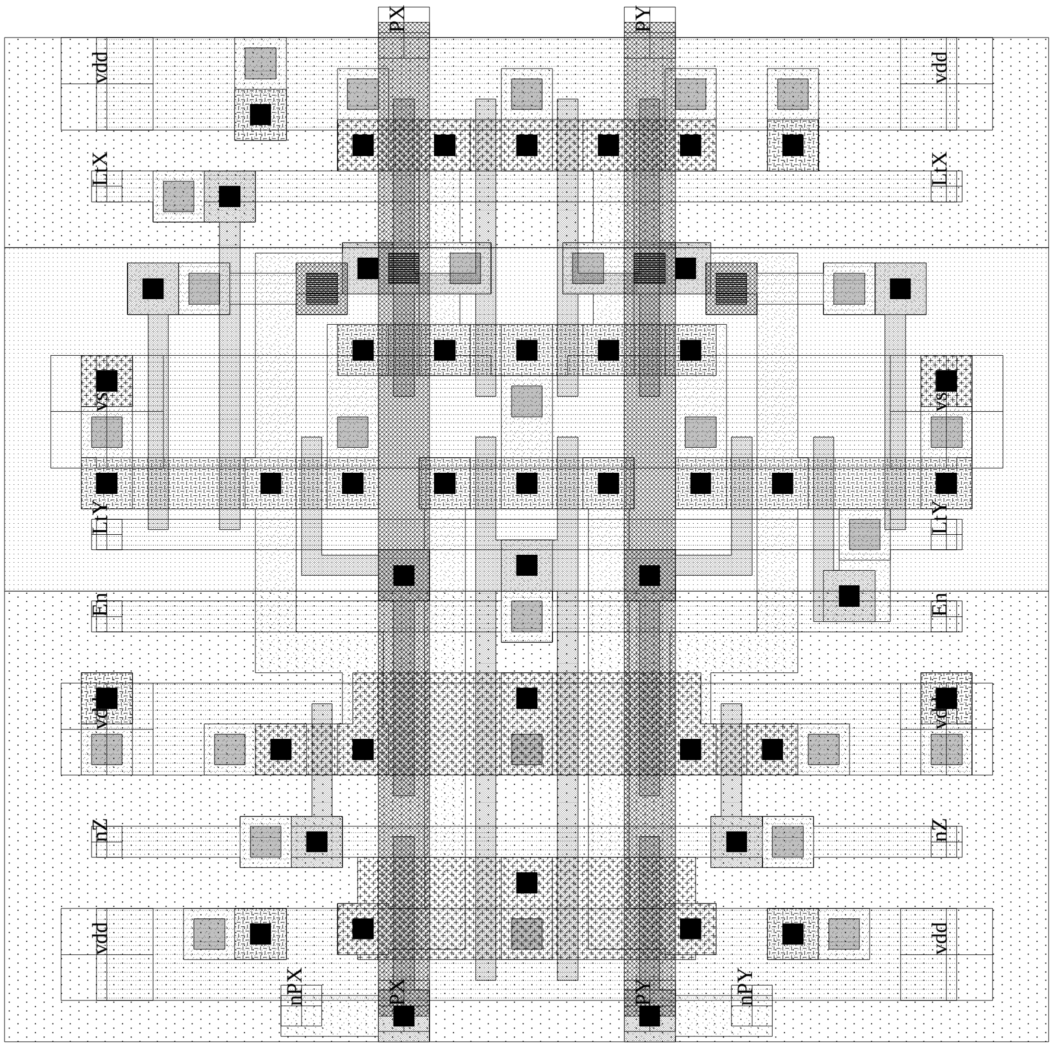
D.5 multlatch



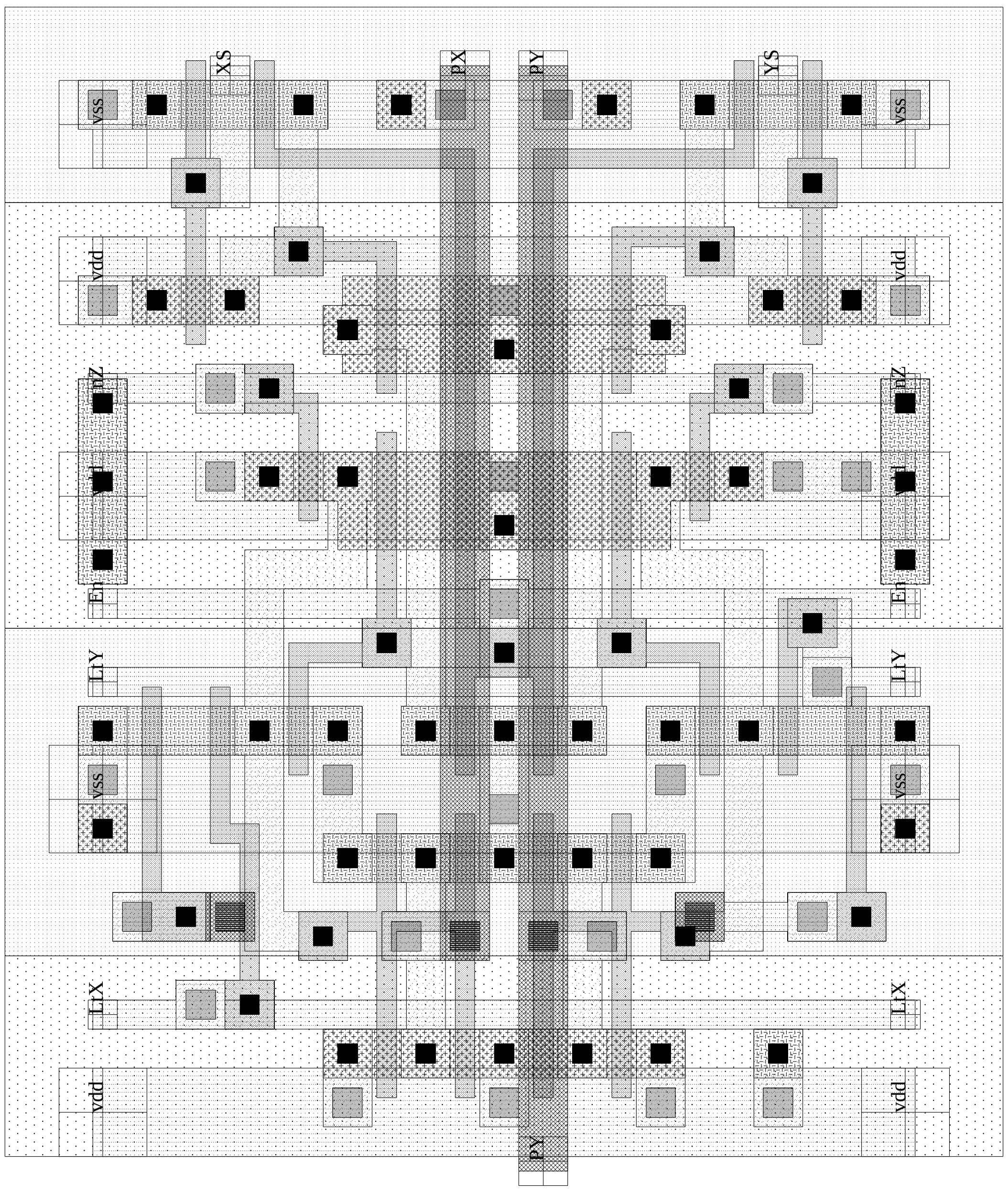
D.6 multdffa



D.7 multdffb



D.8 multdffc


vss
XS
PX
PY
YS
vss
vdd
vdd
nZ
nZ
vdd
vdd
En
En
LtY
LtY
vss
vss
LtX
LtX
vdd
vdd
PY