THE UNIVERSITY OF CALGARY

CASE STUDIES IN ASYNCHRONOUS SYSTEM DESIGN

by

WANZHEN YU

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
DECEMBER, 1994

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Case Studies in Asynchronous System Design" submitted by Wanzhen Yu in partial fulfillment of the requirements for the degree of Master of Science.

G Birtwistle

Supervisor, Dr. G. Birtwistle,

Department of Computer Science

Paul Kwok

Dr. P. Kwok,

Department of Computer Science

L E Turner

Dr. L. E. Turner,

Dept of Computer&Electrical Engineering

Date 24 December 1994

Abstract

As VLSI improves, the feature size of devices is getting smaller and the speed of these devices is getting faster. These improvements allow much more complicated systems to be packed into a single die. Along with these improvements, however, synchronous systems are hitting limits associated with the distribution of the global clock signal. This has rekindled the examination of unclocked asynchronous design. A major problem in asynchronous design is deadlock, and it is very important to check this and kindred properties such as livelock, safety and liveness, etc, before an asynchronous system is sent for manufacture. Recently the mathematics and mechanized support tools have been developed to check these properties. This thesis presents two case studies in designing, specifying and verifying asynchronous systems, which help bridge the gap between the formal method and engineering approach and clarify the hierarchical methodology for developing asynchronous systems.

Acknowledgements

I would like to express my profound appreciation to my supervisor Dr. Graham Birtwistle for his valuable guidance, help and financial support. Graham always had time for me. He read and returned drafts quickly, always making insightful suggestions. Finally, I would like to thank him for expressing confidence in my research and encouraging me when I really needed it.

Thanks to Dr. Saul Greenberg, Dave Spooner and Barry Yee, who carefully read the draft of my thesis. Their insights and suggestions helped me greatly in improving the quality of this thesis. Thanks also to Dr. Robin Cockett for his insightful advice, and to my thesis committee, Drs. Paul Kwok and Laurence Turner, for their careful reading and constructive criticisms.

Thanks to fellow graduate students: Tom Fukushima, Fabian Gomes, Ying Liu, Ken Stevens, Charles Tuckey who were always helpful and supportive; to my roommate Winnie Ho; and to Mr & Mrs Kroeker whose kindness helped me through my first year in Canada.

Finally, special thanks to my parents and my husband for their long lasting support and encouragement.

Contents

List of Tables

List of Figures

CHAPTER 1
Introduction

This thesis culminates in the specification and verification of a small pipelined asynchronous microprocessor. In this chapter, we first motivate asynchronous design. Then we introduce the terminology and methodology used throughout this thesis. Finally, we give the structure of the thesis and summarize its contributions.

1.1. Motivation for Asynchronous Design

1.1.1. Synchronous Design and Its Limitations. Synchronous design has been with us for 30 years. A wealth of experience is now supported by many case studies and well-understood methodologies.

However, VLSI technology is still developing rapidly. The feature size of devices is getting smaller and the speed of these devices is getting faster. These improvements allow much more complicated systems to be packed into a single die. Along with these improvements, however, synchronous systems are hitting limits because they perform computations based on the successive pulses of the global clock. Major problems are listed below:

(1) Clock Distribution: All devices in a synchronous system are supposed to fire at the same time. The delay time of the global clock signal should be within a small window from the clock source to every device in the system. This gives rise to two distinct difficulties:

(a) Area used: A significant part in a chip is used for the clock distribution, e.g., approximately one third of the area [Fur93] in the DEC ALPHA is occupied by the clock distribution circuitry.

(b) Tricky algorithms are required for layout: As feature size decreases by n, the number of devices increases by n^2. With dense chips, the clock distribution network imposes strict conditions on where devices can be placed.

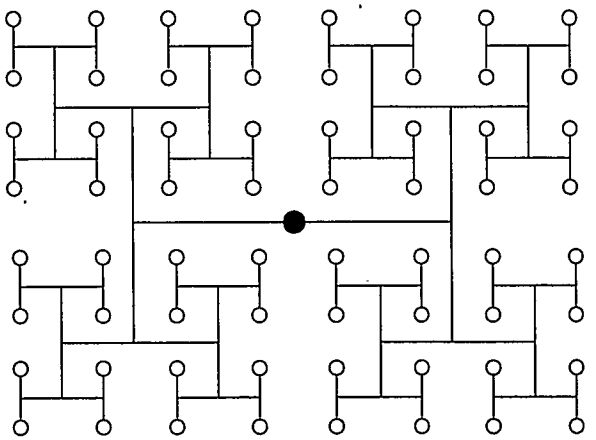


FIGURE 1.1. A Possible Space Filling

For example, a possible space filling is given by Figure 1.1. It guarantees the same delay time by connecting all devices to the clock source with the same wire length. This means that devices can be placed only at locations marked by white circles. The clock source shown as a black circle is in the middle.

(2) Power Consumption: In CMOS, the power dissipated is proportional to the frequency of the clock. The reduction of power consumption offered by the decrease of circuit feature size is offset by the increase of the number of circuits in a single chip. So as chips get faster and denser, removing the generated heat becomes a real problem. The 2nd generation DEC ALPHA uses 60w.

Furber[Fur93] has estimated the dissipated power of a 0.1 μm 5v CMOS processor might reach 2000w by the year 2000!

(3) Performance: The correct operation of a synchronous circuit is established by making the clock period larger than the worst case delay of any possible subcomputation (e.g, the nth stage of a ripple carry ADDER) even though the probability of the occurrence of the worst case may be very small.

(4) Metastability: Any computer circuit that has a number of stable states also has metastable states. When a circuit gets into a metastable state, it can remain there for an indefinite period of time before it resolves into a stable state. Consequently, when a metastable phenomenon occurs in a synchronous system, erroneous data may be sampled at the time of the clock pulse.

These problems motivate the re-examination of asynchronous design.

1.1.2. Asynchronous Design. In an asynchronous system, every component works mostly independent and occasionally cooperating with others by communication. Asynchronous systems have the following potential advantages:

(1) No clock distribution problem: There is no global clock. The tradeoff is that local communication primitives are now needed.

(2) Power consumption: There is no power consumption related to clock distribution. Potential power reduction may be offered by asynchronous design because the devices work only when needed. As yet, there are no good power estimation tools for asynchronous design and an optimal solution is unlikely at present.

(3) Performance: This should reflect the average case of performance. In asynchronous design, as soon as a circuit finishes its computation, it informs the requester that the result is available. The time used only depends on the specific case the circuit is working on. Hence there is no need to optimize a circuit for speed if the likelihood of its worst case behavior is small.

(4) Metastability: This is not a problem for an asynchronous system. When an asynchronous circuit gets into a metastable state, it won't send an acknowledge signal to its user until it settles. This means the result will be sampled only after it settles. No matter how long it stays in a metastable state, it does not affect the correctness of the system.

(5) Uniform interfaces between subsystems: Each subsystem may use 2-phase or 4-phase for signalling, and dual rail or bundled data protocols for data passing. It is easy to switch from one to another, as shown in Section 1.3.2.

(6) The correctness can be established by two separate steps: The correctness of the behaviour of the basic elements is proved by means of physical principles only, and the correctness of the behavior of connections of the basic elements is proved by means of mathematical principles only (i.e., composability).

While the advantages are clear, asynchronous design as a technique is immature. Unlike synchronous design for which many mature techniques have been developed, asynchronous design has been neglected for a long time. There are two main reasons why asynchronous design has been neglected for decades. The first is that basic elements are much more complicated and cost much more than their synchronous counterparts. The second is that there were no nice mathematical tools to tackle the complications of asynchronous designs, such as deadlock, livelock, liveness and safety. The first problem is no longer so overwhelming due to the increasing capability of VLSI fabrication technology. In academic circles asynchronous techniques have always retained a niche since they provide a good test bed for mathematical techniques for proving the correctness of asynchronous systems. Quite recently the mathematics has matured and mechanized testing tools are now in place, e.g., CCS[Mil89] for specifying asynchronous design, the modal μ-calculus [Sti91],[JC90], [Sti92b] and [Sti92a] for property checking as mechanized in the CWB [Mol91] which we shall use in our design, specification and verification. All these are sufficient to make the renewed interest in asynchronous design viable.

1.2. Methodology

Three methodologies will be used in this research: the CCS specification language, the CCS methodology, and hierarchical methodology. Each is discussed below.

1.2.1. CCS Specification Language. CCS is a specification language for asynchronous systems. It builds from the agent (process or object) 0 which can do nothing. From this basic agent, there are four ways of building more interesting agents. Before we introduce them, we first explain the essential "actions" part of an agent. We distinguish two types of actions: input action a and output action $\overline{a}$ which are a pair of actions on the same channel.

Prefixing .

For a given agent P and an action "a", a.P is an agent which first does "a" then evolves into the agent P.

E.g., MATCH $\stackrel{def}{=}$ strike.$\overline{burn}$.0, this means that a match can be struck, respond with a burn and then die.

Non-deterministic Choice +

For two agents A and B, and two actions a and b, a.A + b.B is an agent which evolves into A if it receives a transition on a or into B when it receives a transition on b.

E.g., VEND $\stackrel{def}{=}$ 1p.$\overline{smallcandy}$.VEND1 + 2p.$\overline{bigcandy}$.VEND2, this means that a vending machine pops a small candy and evolves into VEND1 if it accepts one penny, or pops a big candy and evolves into VEND2 if it receives a two penny coin. Here we suppose VEND1 and VEND2 are two other predefined agents.

Parallel Composition |

For two agents A and B, A|B is an agent which allows concurrent behaviours and synchronizations between A and B. E.g.,

PROG $\stackrel{def}{=}$ ((comp1.result2.PROG1) | (comp2.$\overline{result2}$.PROG2)) \ {result2}

this means that PROG consists of two concurrent programs which do their computations concurrently and synchronize on the internal line result2. Here $\backslash\{result2\}$ means that result2 is an internal line in PROG which can not be accessed by the outside. This composition allows us to build a large system only by connecting basic modules.

Recursive Definition

By the above three constructors, we can only build agents which carry out finite actions and then can do nothing. Hardware circuits are reusable — they function forever. With these three basic constructors plus recursive definitions, we are ready to specify the basic modules of asynchronous design.

Example 1: the *C element* is a rendezvous element which produces an output after both inputs arrive regardless of the order they arrive. In CCS this is written: $C \stackrel{def}{=} a.b.\overline{z}.C + b.a.\overline{z}.C$

If the C element receives a transition on a first, it will evolve into $b.\overline{z}.C$ which waits for a transition on b, then causes a transition on z and finally evolves back to C, and can repeat the actions again. Similarly, if b appears first, it will evolve into $a.\overline{z}.C$ which waits for a transition on a, then causes a transition on z and finally evolves back to C, and can repeat the actions again.

Example 2: the *bubbled C element* is a rendezvous element with one transition already fired. In CCS this is written: $C' \stackrel{def}{=} a.\overline{z}.C$

Here b is the input on which it is assumed a transition has happened at the

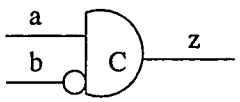


FIGURE 1.2. Bubbled C Element

beginning, see Figure 1.2. When C' receives a transition on a, it will produce a transition on z and then behave exactly the same as C.

Example 3: the *Merge element* produces an output after it receives a transition on any one of its two inputs. In CCS this is written:
$M \stackrel{def}{=} a.\overline{z}.M + b.\overline{z}.M$
If M receives a transition on a or b, it will produce a transition on z and then evolve back to M which can repeat forever.

Example 4: the *Toggle element* alternatively produces a transition on one of two outputs after every transition arrives on its input, with the first transition on the output marked by a dot as in Figure 1.3. In CCS, this is written:
$T \stackrel{def}{=} a.\overline{b}.a.\overline{c}.T$

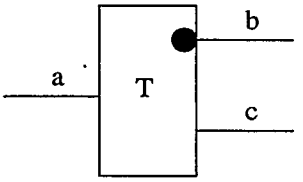


FIGURE 1.3. Toggle Element

Here b is the output marked by a dot. When T receives a transition on a, it will produce a transition on b. When the second transition on a arrives, it will produce a transition on c and evolve back to T which repeats forever.

1.2.2. CCS Methodology. Modal μ-calculus (Hennessy Milner logic plus fixed point) is a companion logic for CCS specifications. It can express all the temporal operators of temporal logic. Thus its expressiveness is strong enough to describe desirable properties for CCS specifications, such as liveness, safety, deadlock and livelock, etc (see chapter 3). This is the logic we used to express properties for an asynchronous

system.

The concurrency workbench CWB is a tool supporting CCS. It provides a very powerful model checker to verify whether a system specified in CCS enjoys properties expressed in the modal μ-calculus, and if two different abstract specifications are equivalent. We checked properties and equivalences in the CWB as explained in chapters 5 and 6.

1.2.3. Hierarchical Methodology. In the CWB, by using CCS and μ-calculus, asynchronous design allows us to develop complicated systems in a hierarchical way from the very abstract level down to a Register Transfer Level implementation which is a composition of basic modules.

We can prove whether an implementation conforms to its specification by checking whether any two adjacent levels in the development towards the lowest level are equivalent to each other on the CWB, and build the correctness of the whole system by checking the correctness of its subcomponents and communications between these subcomponents. When checking properties and correctness, we can use different levels of abstraction for any component to avoid unnecessary details of a big system in any particular design stage. The hierarchical methodology will help us to sort out deadlock and other mistakes. Based on this idea, we can develop hardware designs in the same way as we write software.

In this thesis, by using the tools CCS, the modal μ-calculus and the CWB, we specify and test 2-phase and 4-phase asynchronous designs — two variants of Sutherland's move machine:

(1) AMM (asynchronous move machine), Chapter 5

(2) PAMM (pipelined AMM), Chapter 6

based on the hierarchical methodology.

CCS, the modal μ-calculus and the CWB proved to be adequate for the following reasons: CCS is a simple notation with a clean semantics for specifying asynchronous systems. CCS naturally reflects the behavior of asynchronous systems, E.g., the agent a.b.R, this means that b will occur any time after a happens (without timing restriction). So it is delay insensitive. Finally CCS is expressive enough to give coarse specifications of a large system (synchronizations but not data). It has been found detailed enough to serve as a blue print for AMM, and PAMM. The modal μ-calculus is certainly powerful enough to describe all interesting properties of asynchronous systems, e.g., deadlock, livelock, safety and liveness, etc. When we use general purpose macros, modal μ-calculus formulae can be quite readable. The CWB is an automated model checker with its automated equational checking capability and automated property testing ability. The hierarchical methodology allows us to abstract away some unnecessary detail at any particular design stage, which helps us spot mistakes and property failures in our design as it unfolds level by level.

1.3. Terminology

In this section, we introduce the terminology in our asynchronous design. There are several different timing models for asynchronous design. We introduce the three most common models: speed independent model, delay-insensitive model and bounded delay model. In our design the bounded delay model is used because it is much cheaper than the delay-insensitive model and its delay requirements are not too hard to satisfy. In asynchronous design, every component works independently and cooperates with others by communication. As mentioned previously, communications are the essential part in asynchronous systems. Commonly used communication protocols are combinations of the following: 2-phase or 4-phase styles for signalling; and dual rail or bundled data communication for data passing. In the following subsections, we will define these communication protocols and give the circuits which convert from one communication style to another one.

1.3.1. Timing Model. Asynchronous design is based on one of several timing models. The three most common are described next:

Speed Independent Model A system can function correctly regardless of any delay within its components. This model assumes that no delay is associated with wiring.

Delay Insensitive Timing Model A system can function properly regardless of any delay within the components and in the wires which connect components as well.

Bounded Delay Timing Model A system can function properly if delays within this system satisfy some predefined limits.

A delay-insensitive asynchronous system is guaranteed to work properly in the presense of arbitrary delays in circuits and wires. However it will be very expensive due to the prohibitive number of connection wires needed for dual rail data communication. Some trade-offs need to be considered. Usually a bounded delay timing model is used if the delay requirement is not hard to establish. For example, the bundled data communication will save a significant number of wires. The delay requirement is that the bundled data should arrive at all its destinations before raising the control signal to inform these destinations to sample the data.

1.3.2. Communication Protocols. In asynchronous systems, a signal transition (a rising or falling edge) is used as a basic event. Commonly used communication protocols are one combination of the following: 2-phase and 4-phase styles for signalling; dual rail and bundled data communication for data passing. In the descriptions below, we assume all communication signals commence low.

Signalling Communication Protocol

The interface for asynchronous request-acknowledge communication is shown in Figure 1.4.

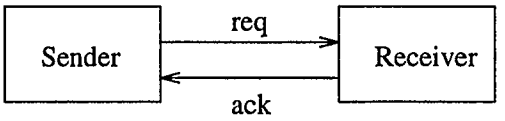


FIGURE 1.4. Asynchronous Communication Interface

2-phase communication as shown in Figure 1.5 is a form of communication which adheres to the following sequence of transitions:

(1) One communication is initiated by the sender making a transition on the request wire,

(2) The communication is ended by the receiver making a transition on the acknowledge wire.

4-phase communication as shown in Figure 1.6 is a form of communication which adheres to the following sequence of transitions:

(1) The communication is initiated by the sender raising the request wire.

(2) The receiver responds to the request by raising the acknowledge wire.

(3) When the sender senses the change on the acknowledge wire, it lowers the request wire to indicate there is no request.

(4) When the receiver detects the change on the request wire, it lowers the acknowledge wire and indicates the current communication has been finished.

Conversion between 2-phase and 4-phase Protocols: 2-phase and 4-phase styles can be used within one system. 2-phase usually is used for control components

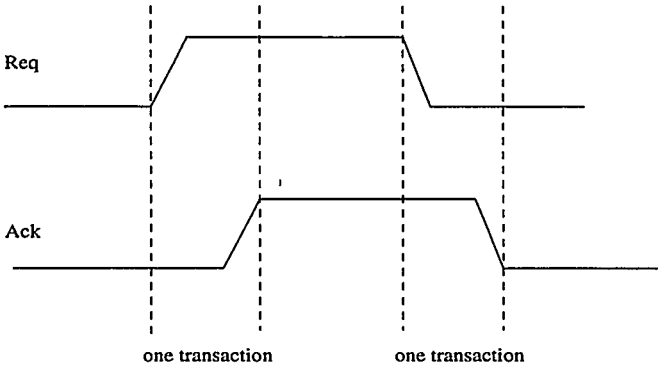


FIGURE 1.5. 2-phase Communication Protocol

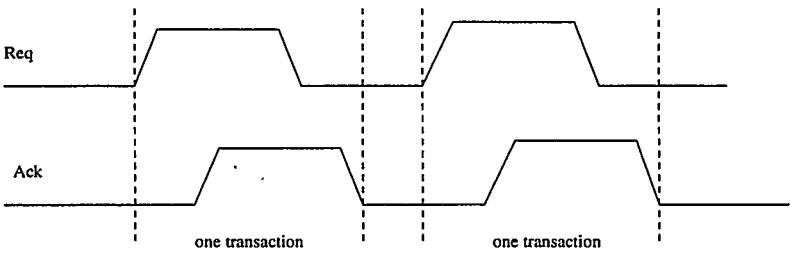


FIGURE 1.6. 4-phase Communication Protocol

due to its simplicity, and 4-phase is used for computation components because a computational circuit needs to be restored to its predefined state before its next use. We can easily convert from one regime to another as typified by:

(1) Conversion from 4-phase to 2-phase: Two versions of converters from 4-phase to 2-phase are illustrated in Figure 1.7 where T is a toggle element and M is a merge element, r1 and a1 is a pair of request and acknowledgement signals of

the 4-phase circuit, r2 and a2 is a pair of request and acknowledgement signals of the 2-phase circuit.

In version 1 the sequence of signallings is — r1 r2 a2 a1 r1 a1. In version 2 the sequence of signallings is — r1 -≺r2 a2 | a1 r1-≺ a1. As the two subsequence r2 a2 and a1 r1 can go in parallel, it is obviously faster. However version 1 is perhaps safer because the called circuit has finished its work when the first acknowledge signal is brought back to the 4-phase caller.

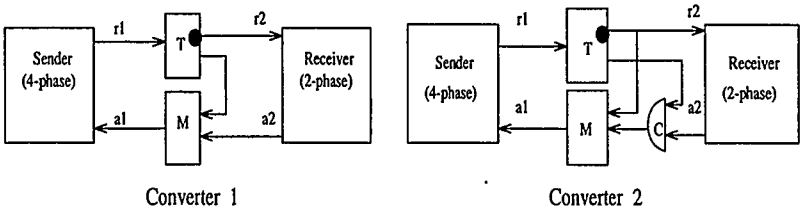


FIGURE 1.7. Converters from 4-phase to 2-phase

For example, a 2-phase FIFO gets data from a 4-phase source as in Figure 1.8. We need a converter from 4-phase to 2-phase between the data source

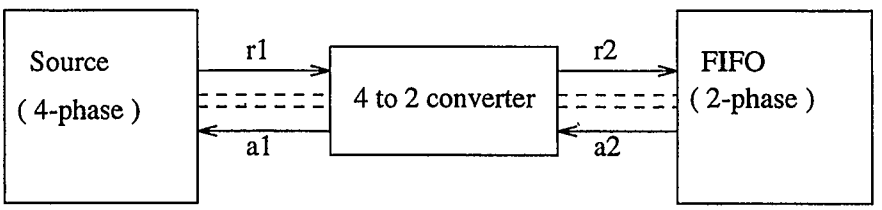


FIGURE 1.8. A 4-2 Example

and the FIFO. In order for the data to be safely stored into FIFO, the data has to remain valid before the FIFO sends its acknowledge signal. In this case, version 2 has some problems. Consider the following scenario:

(a) the source sends request signal to the converter.

(b) the converter will send a request r2 to the FIFO and an acknowledge a1 back to the source.

(c) FIFO can start sampling the data on the bus; at the same time the source is tristating the bus. So the FIFO will not sample the correct data.

Therefore version 2 can not be used when data passing is involved. But if no data passing is involved, version 2 is better because it is faster.

(2) Conversion from 2-phase to 4-phase: This converter as shown in Figure 1.9

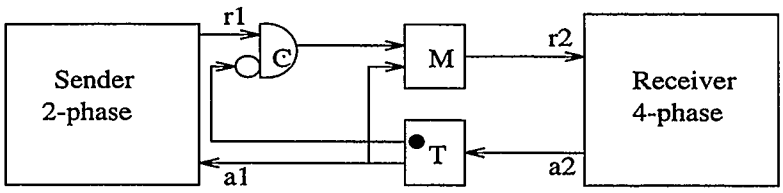


FIGURE 1.9. Converter From 2-phase to 4-phase

passes the request signal from the 2-phase sender to the 4-phase receiver, and then brings the first acknowledge signal back to both the sender as its acknowledge and the receiver for lowering the request line, and finally takes the transition of the acknowledge lowering signal to C element for its next use.

Data Communication Protocol

In dual rail data communication, two request wires (denoted as rF and rT)

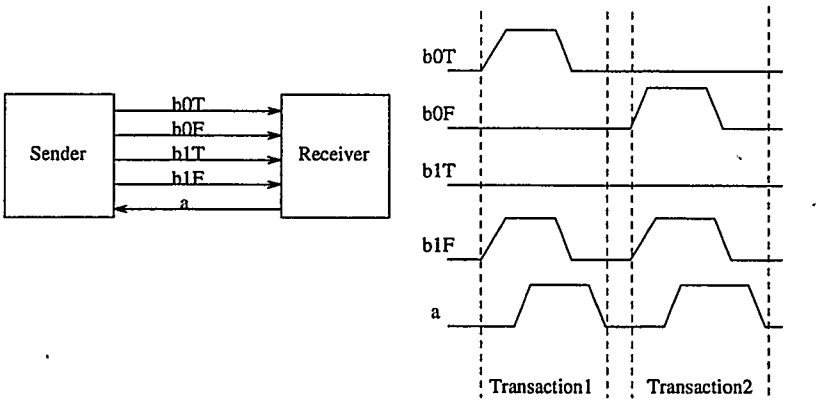


FIGURE 1.10. Dual Rail Data Communication

are needed for passing each bit (rF is for the value 0 of the bit, and rT is for the value 1 of the bit). An additional wire is required for sending the acknowledge back. One 4-phase dual rail data passing transaction involves:

(1) the sender makes transitions in either rF or rT (but not both) for every bit,

(2) the receiver makes a transition in the acknowledge wire to indicate the data has been accepted,

(3) the sender lowers all those wires it raised,

(4) the receiver lowers the acknowledge signal to finish off one transaction.

Without loss of generality, we illustrate a two-bit dual rail data communication in Figure 1.10.

In a bundled data communication, as shown in Figure 1.11, a set of standard data wires (one wire per bit) and a pair of request-acknowledge wires are used. In this protocol a delay requirement must be guaranteed by the data bundle and control wires in order for the system to behave correctly. The requirement is that the delay of the control wire (the request signal) should be longer than the longest delay

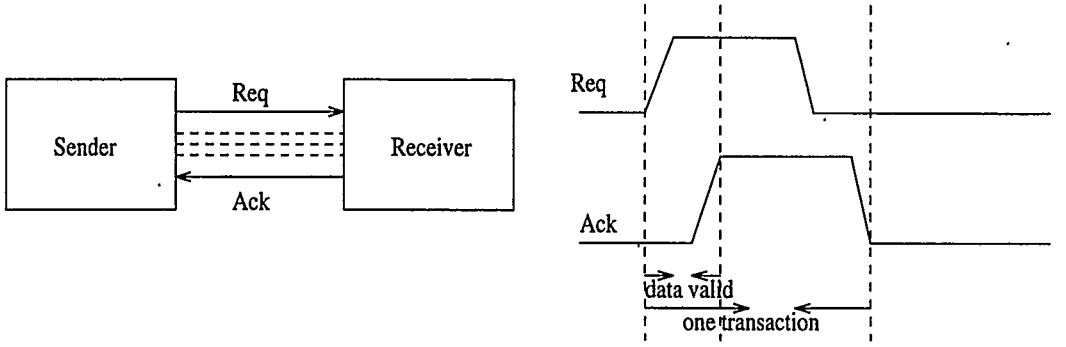


FIGURE 1.11. Bundled Data Communication

for the data bundle to arrive at all its receivers.

Conversion between dual rail and bundled data communication: Here we give the conversion between 4-phase dual rail and 4-phase bundled data communication.

(1) Conversion from dual rail to bundled data communication as typified in Figure 1.12. EN is an element which usually shut off the data flow, let the data through only when requested. Every bit bi on the dual rail bus is connected to biT since the actual level of bi is the same as biT. This converter raises the request signal r when each bit is raised on dual rail wires. This can be done by ANDing all bits to the ENable unit which enables the data to flow from one side to the other side. After the acknowledgement signal is raised, all raised dual rail wires are lowered and therefore the output from the big AND gate becomes low, which causes the EN to be shut off and then lower r. Finally the acknowledgement signal is lowered and this transaction is finished off.

(2) Conversion from bundled data communication to dual rail as typified in Figure

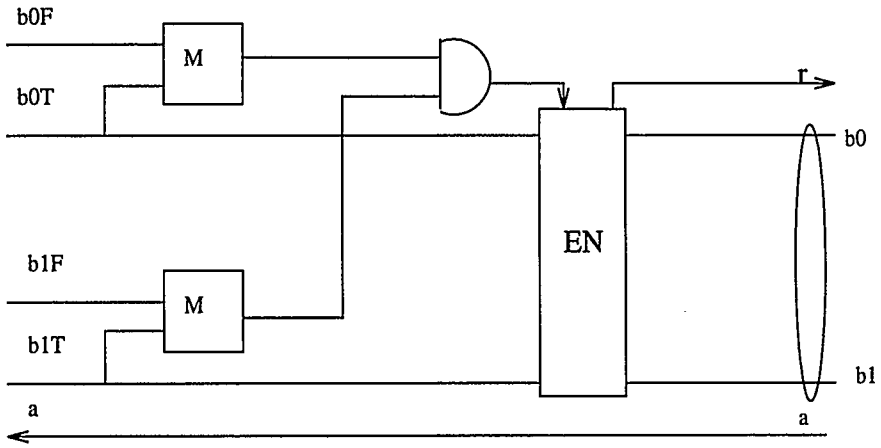


FIGURE 1.12. Conversion from Dual Rail to Bundled

1.13. This converter outputs r$\wedge\overline{bi}$ as biF and r$\wedge$bi as biT. So when r is raised, biT will be raised if b0 is 1 and biF will be raised otherwise. When a is raised, r will be lowered and therefore all dual rail wires become low again. Finally the acknowledgement signal a will be lowered to finish off this transaction.

Therefore we can use the protocol most suited to each subsystem in a design and convert between them easily when needed.

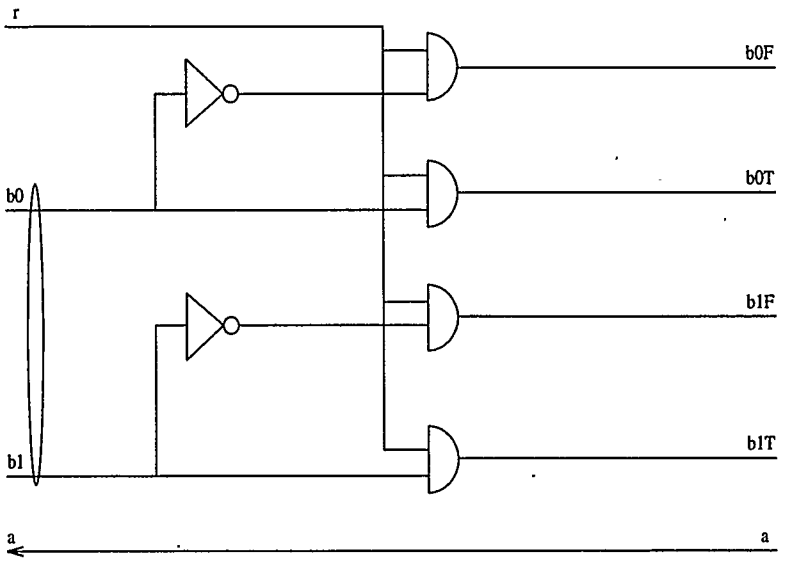


FIGURE 1.13. Conversion from Bundled to Dual Rail

1.4. Structure of the thesis

Chapter 2 surveys three major approaches of asynchronous design. First, silicon compilation typically develops a high-level programming language for specifying asynchronous circuits and builds a silicon compiler to automatically translate programs to circuits that are correct by construction rather than verified. Second, the formal approach focuses on finding suitable models for describing the behavior of asynchronous systems, hierarchical specification techniques, and mathematical tools to check correctness and important properties. Third, the engineering approach builds large scale asynchronous designs using standard tools (for synchronous design) and relies on insight and experience to "get it right". We illustrate these three approaches by introducing one typical related work for each. Our main interest is in the formal specification and verification of large asynchronous system.

Chapter 3 describes the CCS notations, the modal μ-calculus (HML plus fixedpoint operator), and CWB. Based upon transition systems, CCS semantics is given, the interpretation of any formula of the modal μ-calculus and the satisfaction relation between a formula and an agent in a given transition system are defined, and various equivalence relations are introduced. It is argued that the observational congruence best fits our concerns for asynchronous system. Interesting temporal logic operators are defined in terms of the modal μ-calculus. The CWB commands we used in our specification and verification are listed.

Chapter 4 introduces Sutherland's move machine and our variants. Sutherland's move machine is a slave machine which helps the CPU to get rid of the work needed for moving data between CPU and memory, and between two locations in the memory. Our variant is a slightly expanded version of Dave Spooner's original version [BLS+94a] with more consistent register usage and with the addition of a LDI instruction to facilitate the handling of exceptions. Our account is closely based upon

[BLS+94b].

Chapter 5 systematically presents a 4-phase AMM RTL (register transfer level) implementation through three different abstract levels of CCS specification. Our account is closely based upon [BLS+94a] and is included for contrast and comparison with the PAMM model of chapter 6. At the highest abstract level, we specify what AMM does at the instruction level. At the next level, we give the RTL definitions of the datapath and its control signals, but we keep the control unit quite abstract and do not fix precisely where the control signals come from. At the lowest level, we specify the control unit using basic control modules, and wire these modules and datapath together to detail the control signal flow. This refined specification has served as a blueprint for engineers. Property checks and equivalence checks are carried out on these specifications. We show that AMM enjoys such desirable properties as deadlock and livelock freedom, liveness, no bus contention, etc. Three different abstract levels are proved to be consistent.

Chapter 6 covers PAMM design which includes a fetch unit and an execute unit. These two units work independently of each other and communicate with each other in the following cases: (i) interrupt raised by the execute unit when a JCC is met with the condition code true, (ii) stop raised by the execute unit when a HALT instruction is executed, and (iii) new instruction output raised by the fetch unit. In order to improve the speed, we try to make this asynchronous machine maximally parallel. It is very tricky to avoid deadlock in a design when too many components work in parallel and communicate with each other. But we have verified that the pipelined asynchronous move machine is deadlock free.

Finally, chapter 7 summarizes the thesis work and possible future work.

1.5. Contributions of the thesis

This thesis presents two case studies in asynchronous design. In these case studies, we present a systematic development of AMM through three abstract levels which are shown to be consistent and down to a level which has served as an implementation blueprint. This presentation is heavily based upon two technical reports [BLS+94b] and [BLS+94a] and is included for comparison and completeness. We also have designed, specified and verified PAMM by carefully abstracting away regular structures. At the time of writing, these case studies are some of the largest specifications given and specified. These case studies are valuable in their own right and help clarify methodology and bridge the gap between the formal method and engineering approaches.

CHAPTER 2
Approaches to Asynchronous Design

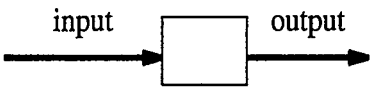
In this chapter we highlight three main approaches to asynchronous design. The first approach is silicon compilation which typically develops a high-level programming language suitable for specifying asynchronous circuits and then generates circuits directly from programs written in that language. Silicon compilers produce designs that are correct by construction rather than verification. The second approach is formal and focuses on finding suitable models for describing the behavior of asynchronous systems, hierarchical specification techniques (e.g., composition) for large systems, and mathematical tools to check important properties (e.g., liveness, safety, deadlock and livelock and different equivalence relationships, etc). The third approach is the engineering approach in which one goes ahead and builds large scale asynchronous designs using standard tools and relying on insight and experience. We will illustrate these three approaches by examples.

2.1. Silicon Compilation

Perhaps the main interest of researchers in computer science is to develop high-level programming languages suitable for specifying asynchronous circuits and then to build silicon compilers for them which generate circuits from programs automatically.

Brunvand [Bru91] chose a LISP-like variant of OCCAM as the source language for specifying asynchronous systems and built a silicon compiler for translating an OC-

CAM program into a 2-phase circuit. As an example of this methodology, Brunvand specifies a single place buffer by:




```
(Block ((char input<8> output<8>));a process with two 8-bit channels
   ( While True                    ;repeat forever
      (Seq (( Var temp<8>))        ;sequential composition with 8-bit
                                   ;local variable temp
        ( input? temp)             ;get value from channel input and
                                   ;store in temp
        ( output! temp))           ;send value in temp to channel output
```

and the silicon compiler translates such a description into a circuit with 2-phase and bundled data passing protocol (the control part of which is delay-insensitive) in the following steps.

(1) Initial replacement: Every OCCAM primitive and OCCAM construct will be replaced by a corresponding circuit. This replacement is guaranteed to be correct by construction. The initial circuit of the single place buffer generated by the compiler is as shown in Figure 2.1, in which the circuits labelled "M" are Merge elements.

(2) Optimization: A peephole-like technique is used to optimize the initial resulting circuits. The basic idea is to locate in a circuit a subcircuit matching some template and replace it with a simpler part which conforms to the original one. Replacement is repeated until the circuit can not be further modified.

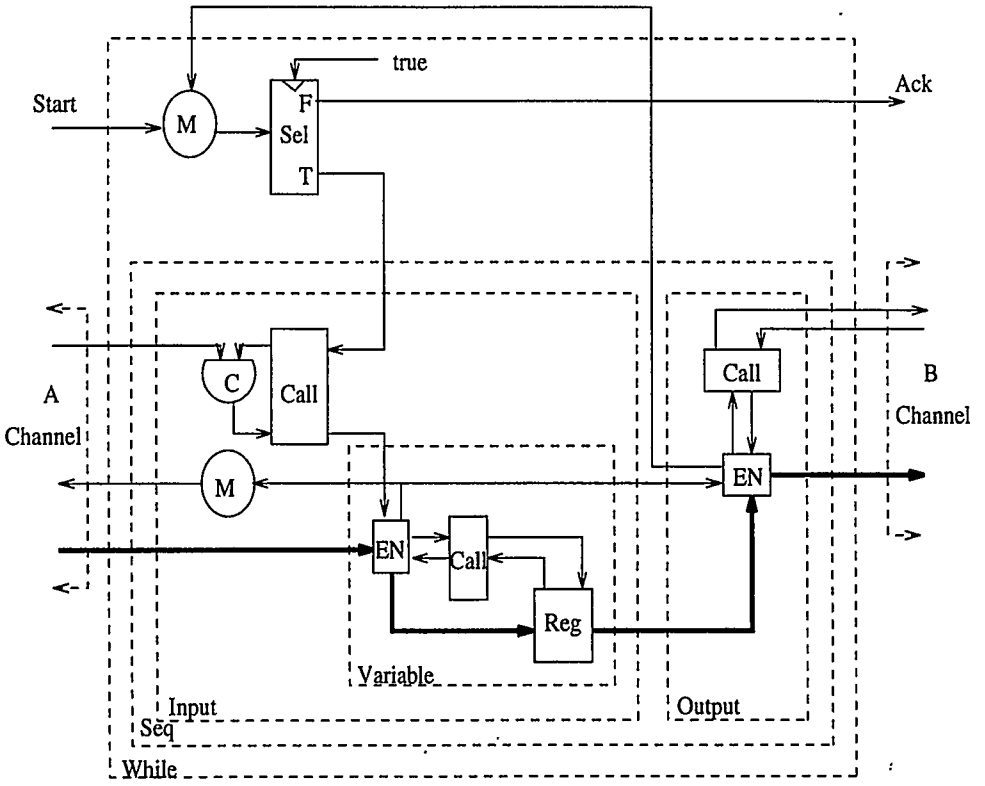


FIGURE 2.1. The initial circuit of single place buffer

The initial circuit of the single-place buffer is modified by the following replacement:

- Every call module with a single client will be replaced by a pair of wires connecting R1 (the request signal from the caller) to Rs (the request signal issued by the call element), and As (the acknowledge from the called circuit to the call element) to A1 (the acknowledge issued by the call element to the caller) because the trace structures are identical when the environmental constraints present in the call module are applied to this pair of wires. Therefore all the call modules in the initial circuit are replaced by pairs of wires.
- A Select module (asynchronous multiplexer) instantiated with constant True or False conditions on their sel inputs will be replaced by a wire connecting A and B (or C) because the trace structures are identical when the environmental

constraints present in the Select module are applied to this wire. The Select module in the circuit generated by step 1 is replaced by a single wire.

- Enable modules are designed to gate different signals onto a common bus. In particular, they will drive signals onto a shared bus upon request and then report that the values on the bus are correct. Given a request to disable, the module will stop driving the bus and make sure the module outputs are in a high impedance state before reporting the bus is available. The purpose is to avoid bus contention. When there is only one source connecting to a bus, the EN unit is superfluous and can be replaced by wires connecting Ren to Aen, Rdis to Adis, and data-in to data-out. The circuit is changed by replacing two EN modules with wires.
- The Start signal is a transition that is issued by the environment after intialization to initiate action in the circuit. If Start is wired to a Merge module, the Merge module produces an output transition upon the receipt of a Start transition. In this case it is possible to remove the Start signal and replace the Merge module with an inverter. This transformation depends on the fact that modules respond to the master-clear signal by setting their control signal outputs low, so the inverter will produce the effect of a Start transition issued immediately after master-clear is removed. The Start signal in this circuit is removed and the Merge element is replaced by an inverter.
- M-element with a single input is trace-equivalent to a wire connecting the input to the output. This means that it can be replaced by a wire. One M-element is replaced by a single wire.

After the above sequence of transformations, the final circuit is given in Figure 2.2. The final circuit is internally presented as a graph with the basic modules as nodes and the connection wires as arcs. Brunvand also built output drivers to print the final circuit in the format of Netlist for simulation, and to print the final circuit in the format of Fusion [Bru91] output for routing and placement.

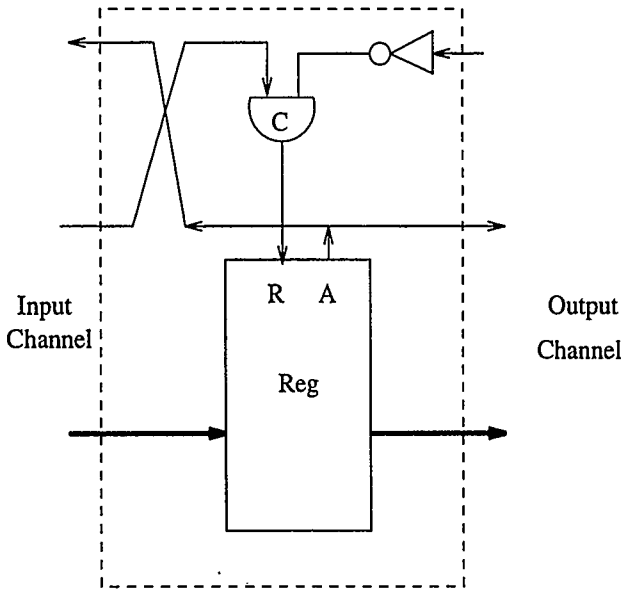


FIGURE 2.2. The final circuit of single place buffer

The VLSI circuits constructed using this technique have been mapped into CMOS, FPGA and GaAs technologies. A simple RISC processor has been implemented on multiple FPGA elements to demonstrate the practicality of the approach.

However, there is no support for verification, and the specification which could be written in this frame is not really abstract since one only can design programs by using operators predefined over data types. Although the rules of transformation are well argued, they are not formally proved.

Similar work has been carried out at Eindhoven, Caltech, and Philips. In Eindhoven University of Technology, Michiel Van der Korst[vdK92] has built a Silicon Compiler VOICE which translates a specification written in a CSP-like language VOKEL into a handshake circuit, an intermediate representation. Prototype tools for optimization, simulation and visualization of handshake circuits are developed.

In California Institute of Technology, Martin and Burns[BM88] have described a technique for automatically translating a concurrent program based on CSP and Dijkstra's guarded commands to a 4-phase delay-insensitive circuit. An asynchronous RISC style microprocessor has been developed that demonstrated the feasibility of this approach.

Van Berkel at Philips Research [vBNRS88], [vBKR+91] also built a compiler which translates programs written in CSP-like language Tangram in two steps.

(1) Produce a handshake circuit which is a network of components connected together by point-to-point channels.
(2) Change this intermediate form into a netlist of standard-cell VLSI modules for final silicon layout.

The resulting VLSI circuits use a delay-insensitive, 4-phase, dual-rail protocol for communication between components. The system has been used to generate a number of VLSI circuits[vB92b].

2.2. Formal Approach

The formal approach focuses on finding suitable models for describing the behaviour of asynchronous systems, hierarchical specification techniques (e.g., composition) for large systems, and mathematical tools to check some important properties (e.g., deadlock, livelock and different equivalence relationships, etc).

For example, Jo Ebergen has developed a language ([Ebe89], [Ebe91b] and [Ebe91a]), based on trace structure, for specifying asynchronous systems. This language includes variables, channels and guarded selections and can deal with data processing as well as communication synchronizations.

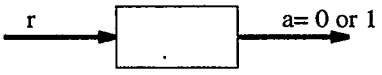


FIGURE 2.3. A modulo-N counter

For example, here is a modulo-N counter, which outputs 1 after each of the first N-1 inputs and 0 after the Nth input. It can be specified by the following description.

```
ModC(N:int, r?:un, a!: bin)

= { by definition }

|[var n:int ::
  initially n=0 ::
  pref*[r?;
```

```
        n:=(n+1) mod N;
        if n>0 then a:=1
        |  n=0 then a:=0
        fi;
        a!;
       ]
]|
```

The formal semantics of a command is its corresponding trace structure. For example, the above specification has the trace structure:

E = < iE, oE, tE> where,

$iE = \{r\}$;

$oE = \{< a, 0 >, < a, 1 >\}$;

$tE = pref * [(r?; < a!, 1 >)^{N-1}; r?; < a!, 0 >]$

The definition of decomposition formalizes the idea of "implementing a specification by a network of components". The substitution theorem formalizes the modular design method. A network, which is a decomposition of a specification, represents a speed-independent circuit. If all constituent components are delay-insensitive, then a speed-independent decomposition is also a delay-insensitive decomposition.

We can decompose this specification by using a divide and conquer approach: decompose the modulo-2N counter into a modulo-N counter and a "small" subcomponent which has a small number of states (we just show the even number case of N) . One possible decomposition of this specification is:

CELL(r?:un,a!:bin,sr!:un,sa?:bin)

={definition}

|[var k:bin::
initially sa=0,k=0::
pref*[r?;k:=(k+1) mod 2;
if k=0 then sr!;sa?|k $\neq$ 0 then skip fi;
if sa $\neq$ 0 or k $\neq$ 0 then skip
| sa=0 and k=0 then a:=0 fi; a!;]
]|

ModC(2N,r!,a!)

={ def. of weave}

|[chan sr:un,sa:bin::CELL(r?,a!,sr!,sa?) || ModC(N,sr?,sa!)]|
$\longrightarrow$ { def. of decompostion}
(CELL(r?,a!,sr!,sa?), ModC(N,sr?,sa!))

Figure 2.4 shows two implementations of the CELL. When N = 2^k, k CELLs are connected to implement the modulo-N counter.

Ebergen has given the analyses for response time, area complexity and power consumption within the first order approximations with no reference to any specific physical implementation provided that certain conditions are satisfied by the network. The measures of these three performances are given below.

(1) The area of a system is measured by the number of "basic" components in the decomposition, where a basic component can be any component whose number of states is bounded by a predetermined constant.

(2) The energy is measured by the number of communication actions in a behaviour. The power consumption is measured by the total number of communication actions amortized over the external communication actions for a

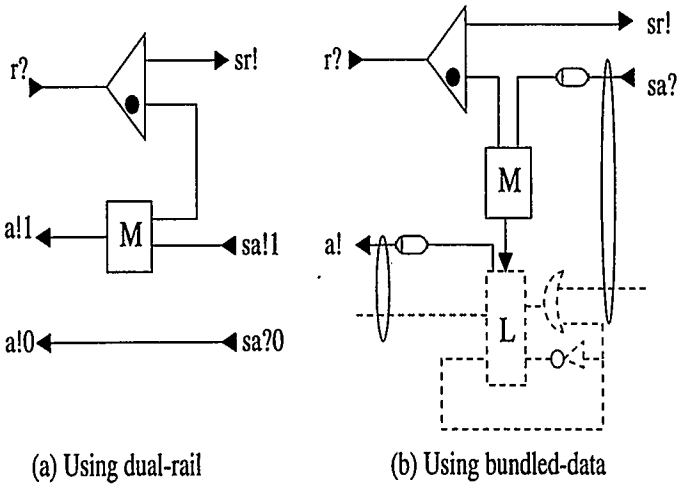


FIGURE 2.4. Implementation for CELL

worst-case environment which communicates with the implementation in such a way that the total number of communication actions is maximized over the long term.

(3) The response time is measured from the time the last input arrives that enables the production of that output to the actual production of that output. In calculating the response time, it is assumed that the response time of basic components are bounded from above and below by fixed constants.

The above implementation for the modulo-N counter has an amortized constant bound of response time and power consumption, and O(log(N)) area complexity. This is optimal for a modulo-N counter.

However, the decomposition of specification is conducted by hand, and the implementation correctness criteria deduced by the definition of decomposition are not strong enough since they are based on trace equivalence, e.g., an implementation which deadlocks will not be detected.

There is some other work based on transitional semantics. Various equivalent relationships have been given for different applications. We are most interested in observational congruence which well characterizes the faithfulness of an implementation to its specification. A group of researchers in University of Edinburgh have done extensive research work for formal verification [Mol91], [Sti92b], and [Sti92a]. Their workbench CWB has automated the equivalence (various equivalence relationships) checking and property (expressed in the modal μ-calculus) checking of asynchronous systems specified in CCS.

Other formal work can be found in [DNS92], [Kal86], [Sch85], [Udd86], and [UV88], [BS87] and [BE92].

2.3. Engineering Approach

The engineering approach demonstrates the feasibility of asynchronous design by constructing real asynchronous designs. The AMULET group in Manchester University has developed an asynchronous implementation [Fur93] and [PDF+92] of the ARM microprocessor as part of a broad investigation into lower power techniques. Their first commercially realistic asynchronous product is named AMULET1. Their methodology is based on Sutherland's "Micropipelines" [Sut89], a transition signalling bundled data model.

The Manchester AMULET group has implemented the basic library of event control elements proposed by Sutherland, and extra event control elements: a transparent latch for blocking events, a decision-wait element for performing a rendezvous between one control line and either of a pair of event lines, an event control transparent latch, and a capture-pass latch.

The datapath of AMULET1 [Fur95] is decomposed into 4 blocks: a data interface, an address interface, an execution unit and a register bank.

- Data interface forwards instruction and data from, and returns data to the memory.
- Address interface addresses to the memory. It autonomously increments PC while new addresses (for branch or data access) can arrive asynchronously.
- Execution unit carries out arithmetic and logic operations. It takes register operands from the register bank. Immediate operands are forwarded from the execute unit. The result is usually returned to the register bank. It is implemented as three pipelined stages to improve performance.
- Register bank holds current values of all the registers except the PC. It locks the left hand side registers until the corresponding right hand side operation

has been completed and assigned. It includes an arbiter-free locking mechanism which enables efficient read operations in the presence of multiple pending write operations.

First silicon returned from fabrications arrived in April 1994 (AMULET1 has now been fabricated on two CMOS processes: a 1 μm process at ES2 and a 0.7 μm process at GEC Plessey Semiconductors). Both prototype devices are functional and execute programs produced by standard ARM development tools such as the assembler and C compiler. The comparison of AMULET1 and ARM6 [PDF+92] is shown in the following table:

	AMULET1/ES2	AMULET1/GPS	ARM6
Process	1μ	0.7 μ	1μ
Area(mm^2)	5.5 x 4.1	3.9 x 2.9	4.1 x 2.7
Transistors	58,374	58,374	33,494
Performance	20.5kDhry	40kDhry	31kDhry
Multiplier	5.3ns/bit	3ns/bit	25ns/bit
Conditions	5V,20°C	5V,20°C	5V,20MHz
Power	152mW	?	148mW
MIPS/W	77	?	120

TABLE 2.1. Comparison of AMULET and ARM6

AMULET1 has demontrated the feasibility of designing a full functionality commercial RISC architecture in asynchronous logic. While this design doesn't outperform its synchronous counterpart, its performance is within a factor of two in all areas. As this is a first attempt by this group at producing an asynchronous design of this complexity, it is quite encouraging. Preliminary indications are that AMULET 2 will be both faster and use less power than ARM6.

At HP Laboratory, Bill Coates, Al Davis and Ken Stevens [Dav95],[DN95] developed an asynchronous, 300,000 transistor, full custom CMOS chip designed as the communication coprocessor (named as Post Office) for the Mayfly scalable parallel processor.

2.4. Summary

Asynchronous design has always retained a niche in academic circles because it provides a good framework for mathematical techniques for proving the circuit correctness. By now there are some adequate mathematical tools available for specifying and verifying asynchronous circuits. In turn these have awakened renewed interest from industry. The VLSI group at Calgary specializes in the formal specification and verification of large asynchronous hardware systems by using a coarse-grain model. CCS is an appropriate tool, supported by the modal μ-calculus for expressing desirable properties and CWB for property checks and equivalence checks. This thesis contains two case studies which demonstrate the application of CCS and the μ-calculus (as mechanized in the CWB) to the specification and verification of a small microprocessor (variants of Sutherland's move machine). The goal is to bridge the gap between formal and engineering schools, and show that coarse grain formal specifications are pitched at the right abstract level to both establish the major deadlock, livelock, and safety properties *and* to serve as an implementation blueprint for the engineers. Time is too short to write a silicon compiler supporting our approach.

CHAPTER 3
Tools for Specification and Verification

This chapter introduces the notation and tools used to design and test our variants of Sutherland's move machine. CCS is a specification language for asynchronous systems. Its semantics is defined in terms of labelled transitional systems. It is argued that observational congruence best fits our concern for asynchronous circuits. The modal μ-calculus is a companion logic for CCS specification. It can be used to express and test that certain desirable properties hold for our asynchronous systems. Some important commands in the Concurrency Workbench (CWB) are listed and used to check the equivalence between CCS specifications at different abstract levels and the satisfaction of those properties we wish to hold. For complete accounts, see [Mil89] for CCS, [Sti91] for the modal μ-calculus and [Mol91] for the workbench CWB. For the necessary intuition and application, see also the theses of Liu [Liu92] and Stevens [Ste94].

3.1. CCS - Calculus of Communicating Systems

In this section, we present the syntax of CCS in BNF and its operational semantics. Both are illustrated by examples.

3.1.1. Syntax of CCS. CCS has a simple and clean syntax, as described below.

E ::= 0	Nil agent
$\mid A$	constant
$\mid a.E$	a prefix($\alpha \in Act$)

$\|E1 + E2 + ... + En$	summation
$\|E1\|E2\|...\|En$	composition
$\|E\backslash L$	restriction (L$\subseteq \mathcal{L}$)
$\|E[f]$	relabelling

where $\mathcal{L}$ is a fixed set of labels, τ is an internal action, Act is $\mathcal{L} \cup \{ \tau \}$ and f is a relabelling fuction.

The expressions in this language are called agent expressions, or agents for short. As illustrations, we now give three example definitions of common asynchronous circuits.

Example 1: A *fork* element, which routes a transition at its input to its two outputs, is specified by:

```
F = in.('out1.'out2.F + 'out2.'out1.F)
```

Example 2: A *call* element, which serves a circuit with two users who never need the circuit at the same time, is specified in CCS as:

```
call = r1.'r.a.'a1.call + r2.'r.a.'a2.call
```

Example 3: An *arbiter* unit, which also serves a circuit with two user who may need the circuit at the same time, is specified in CCS as:

```
Sem = 'g.p.Sem
U1 = r1.g.'g1.'d1.'p.U1
U2 = r2.g.'g2.'d2.'p.U2
Arbiter = (U1 | U2 | Sem) \ {g,p}
```

3.1.2. Operational Semantics of CCS. We give the CCS semantics as a labelled transition system: ($\mathcal{E}$, Act, $\rightarrow$) where $\mathcal{E}$ is the set of all agent expressions, $\rightarrow$ is a triple relation over $\mathcal{E} \times$ Act $\times \mathcal{E}$ (when (E1, a, E2) $\in \rightarrow$, we write $E1 \xrightarrow{a} E2$). The transition relation is given by induction on the structure of agent expressions:

$Act \frac{}{a.E \xrightarrow{a} E}$

$Sum1 \frac{E1 \xrightarrow{a} E1'}{E1+E2 \xrightarrow{a} E1'}$ $\quad$ $Sum2 \frac{E2 \xrightarrow{a} E2'}{E1+E2 \xrightarrow{a} E2'}$

$Com1 \frac{E1 \xrightarrow{a} E1'}{E1|E2 \xrightarrow{a} E1'|E2}$ $\quad$ $Com2 \frac{E2 \xrightarrow{a} E2'}{E1|E2 \xrightarrow{a} E1|E2'}$ $\quad$ $Com3 \frac{E1 \xrightarrow{a} E1', E2 \xrightarrow{\bar{a}} E2'}{E1|E2 \xrightarrow{\tau} E1'|E2'}$

$Res \frac{E \xrightarrow{a} E'}{E\backslash L \xrightarrow{a} E'\backslash L}$ (a,'a $\notin$ L)

$Rel \frac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]}$ $\quad$ $Con \frac{P \xrightarrow{a} P'}{A \xrightarrow{a} P'} (A \stackrel{def}{=} P)$

In example 3, we specified the arbiter in CCS. When both users request before

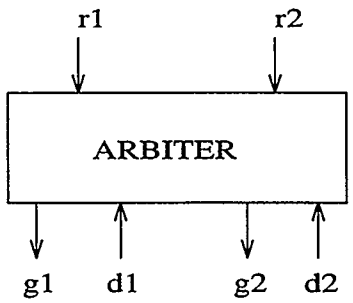


FIGURE 3.1. Arbiter

any grant has been made, one request will be granted and the other user has to wait until the grantee is done. Let us prove this is what the arbiter can do according to the above semantics.

Ecample 4: A verification of mutual exclusion via arbitration.

r1.g.'g1.d1.'p.U1 $\xrightarrow{r1}$ g.'g1.d1.'p.U1 (Act)

r2.g.'g2.d2.'p.U2 $\xrightarrow{r2}$ g.'g2.d2.'p.U2 (Act)

$U1 \xrightarrow{r1}$ g.'g1.d1.'p.U1 ($A \stackrel{def}{=} P$)

$U2 \xrightarrow{r2}$ g.'g2.d2.'p.U2 ($A \stackrel{def}{=} P$)

(U1|U2|Sem) $\xrightarrow{r1}$ (g.'g1.d1.'p.U1|U2|Sem) (Com1)

(U1|U2|Sem)$\backslash\{g,p\} \xrightarrow{r1}$ (g.'g1.d1.'p.U1|U2|Sem)$\backslash\{g,p\}$ (Res)

Arbiter $\backslash\{g,p\} \xrightarrow{r1}$ (g.'g1.d1.'p.U1|U2|Sem)$\backslash\{g,p\}$ ($A \stackrel{def}{=} P$)

(g.'g1.d1.'p.U1|U2|Sem) $\backslash\{g,p\} \xrightarrow{r2}$

(g.'g1.d1.'p.U1|g.'g2.d2.'p.U2 |Sem) $\backslash\{g,p\}$ (Com2)

g.'g2.d2.'p.U2 $\xrightarrow{g}$ 'g2.d2.'p.U2 (Act)

'g.p.Sem $\xrightarrow{'g}$ p.Sem (Act)

Sem $\xrightarrow{'g}$ p.Sem ($A \stackrel{def}{=} P$)

(g.'g1.d1.'p.U1|g.'g2.d2.'p.U2 |Sem) $\backslash\{g,p\} \xrightarrow{\tau}$

(g.'g1.d1.'p.U1| 'g2.d2.'p.U2 |p.Sem) $\backslash\{g,p\}$ (Com3)

(by now g.'g1.d1.'p.U1 cannot do anything since there is no 'g in Sem to synchronize with g.)

'g2.d2.'p.U2 |p.Sem $\xrightarrow{'g2}$ d2.'p.U2 |p.Sem (Act)

d2.'p.U2 |p.Sem $\xrightarrow{d2}$ 'p.U2 |p.Sem (Act)

(g.'g1.d1.'p.U1| 'g2.d2.'p.U2 |p.Sem) $\backslash\{g,p\} \xrightarrow{'g2}$

(g.'g1.d1.'p.U1| d2.'p.U2 |p.Sem) $\backslash\{g,p\}$ (Com2)

(g.'g1.d1.'p.U1| d2.'p.U2 |p.Sem) $\backslash\{g,p\} \xrightarrow{d2}$

(g.'g1.d1.'p.U1| 'p.U2 |p.Sem) $\backslash\{g,p\}$ (Com2)

'p.U2 $\xrightarrow{'p}$ U2 (Act)

p.Sem $\xrightarrow{p}$ Sem (Act)

('p.U2 | p.Sem) $\xrightarrow{\tau}$ (U2|Sem) (Com3)

(g.'g1.d1.'p.U1|'p.U2 | p.Sem) $\xrightarrow{\tau}$ (g.'g1.d1.'p.U1|U2|Sem) (Com2)

(g.'g1.d1.'p.U1|'p.U2 | p.Sem)$\backslash\{g,p\} \xrightarrow{\tau}$

(g.'g1.d1.'p.U1|U2|Sem)$\backslash\{g,p\}$ (Com2)

So the arbiter can evolve into (g.'g1.d1.'p.U1|U2|Sem)$\backslash\{g,p\}$ by a sequence of actions r1, r2, τ(g and 'g), 'g2, d2, and τ(p and 'p). In this sequence r2 has been granted

first and r1 cannot be granted until user 2 is done and releases the Sem by doing 'p which synchronizes with p.

3.1.3. Some Equivalence Relationships. Various equivalence relationships have been proposed in terms of the transition relation over the set of agent expressions. This section introduces four equivalence relations (trace equivalence, strong bisimulation, weak bisimulation and observational congruence). Observational congruence is the one that best addresses our concerns about the equivalence between asynchronous system specifications at different abstract levels.

Trace equivalence

Definition 1: The trace set of an agent E is { t ∈ Act* | for some E', $E \stackrel{t}{\Rightarrow} E'$}. We use tE to stand for the trace set of E.

Definition 2: Two agents E1 and E2 are trace equivalent if and only if tE1 = tE2.

This equivalence relation is too weak. For instance,
E1 = a.0 + a.E1 and E2 = a.E2 are trace equivalent. E1 includes deadlock while E2 is deadlock free. Apparently we want to distinguish these two agents.

Strong bisimulation

Definition 3: A binary relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ over agents is a strong bisimulation if (P,Q) ∈ $\mathcal{S}$ implies, for all $\alpha \in Act$,

(1) Whenever $P \xrightarrow{a} P'$ then, for some Q', $Q \xrightarrow{a} Q'$ and $(P', Q') \in S$

(2) Whenever $Q \xrightarrow{a} Q'$ then, for some P', $P \xrightarrow{a} P'$ and $(P', Q') \in S$

Definition 4: P and Q are strongly equivalent (or strongly bisimular), written P ∼ Q, if (P,Q) ∈ $\mathcal{S}$ for some strong bisimulation $\mathcal{S}$. This may be equivalently expressed as follows: ∼ = ⋃{$\mathcal{S}$: $\mathcal{S}$ is a strong bisimulation}

$\sim$ is also a congruence, which means strong equivalence is substitutive under all combinators. However, this equivalence relation is too strong because every τ action has to be matched. For example, the two agents E1 = a.τ.0 and E2 = a.0 are not strongly equivalent. Since the internal action τ is unobservable, both agents have an a action and then evolve into deadlock. These two agents don't make any difference to us in this sense and they should not be distinguished.

Weak bisimulation

Definition 5: A binary relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ over agents is a (weak) bisimulation if (P,Q) $\in \mathcal{S}$ implies, for all $a \in L$,

(1) Whenever $P \xrightarrow{a} P'$ then, for some Q', $Q(\xrightarrow{\tau})* \xrightarrow{a} (\xrightarrow{\tau}) * Q'$ and $(P', Q') \in S$

(2) Whenever $Q \xrightarrow{a} Q'$ then, for some P', $P(\xrightarrow{\tau})* \xrightarrow{a} (\xrightarrow{\tau}) * P'$ and $(P', Q') \in S$

(3) Whenever $P \xrightarrow{\tau} P'$ then, for some Q', $Q(\xrightarrow{\tau}) * Q'$ and $(P', Q') \in S$

(4) Whenever $Q \xrightarrow{\tau} Q'$ then, for some P', $P(\xrightarrow{\tau}) * P'$ and $(P', Q') \in S$

Definition 6: P and Q are bisimular, written P $\approx$ Q, if (P,Q) $\in \mathcal{S}$ for some bisimulation $\mathcal{S}$. This may be equivalently expressed as follows:
$\approx = \bigcup\{\mathcal{S}$: $\mathcal{S}$ is a bisimulation$\}$

From this definition, we can see the internal action τ is totally ignored. Unfortunately $\approx$ is not a congruence. For example, b.0 $\approx \tau$.b.0 while a.0 + b.0 $\not\approx$ a.0 + τ.b.0.

Observational congruence

Definition 7: P and Q are observationally congruent, written P = Q, if for all a$\in$ Act,

(1) Whenever $P \xrightarrow{a} P'$ then, for some Q', $Q(\xrightarrow{\tau})* \xrightarrow{a} (\xrightarrow{\tau}) * Q'$ and P'$\approx$Q',

(2) Whenever $Q \xrightarrow{a} Q'$ then, for some P', $P(\xrightarrow{\tau})* \xrightarrow{a} (\xrightarrow{\tau}) * P'$ and P'$\approx$Q'.

From this definition, we can see every τ action has to be matched by at least one τ action between two observationally congruent agents. = is substitutive under all

combinators. If $P \approx Q$ and both are stable (a stable agent is one that cannot have any immediate τ action), $P = Q$. So the observational congruence check can be reduced to bisimularity check if the two compared agents are stable. Also we can easily see that two agents are not observationally congruent if one is deadlock and the other one is not.

3.2. Process Logics

In this section, we introduce the process logics: Hennessy Milner Logic (HML) and modal μ-calculus. We use agents as their models. Thus we interpret formulae by agents. The satisfaction relationship between an agent and a formula are defined. Some examples are given to show how to express properties.

3.2.1. Hennessy Milner logic. This subsection gives HML syntax in BNF, its interpretation by any given agent and the satisfaction relationship between an agent and a formula by structural induction.

Syntax of HML in BNF:

$A ::= T \mid \neg A \mid A \wedge B \mid [K]A$

Where

T is the constant true,

K is a subset of Act.

Models:

A labelled transition system is $(\mathcal{P}, \mathcal{A}, \rightarrow)$ where

$\mathcal{P}$ is a nonempty set of agents,

$\mathcal{A}$ is an action set,

$\rightarrow$ is a relation over $\mathcal{P} \times \mathcal{A} \times \mathcal{P}$ for each $a \in \mathcal{A}$,

$P1 \xrightarrow{a} P2$ stands for $(P1, a, P2) \in \rightarrow$.

Interpretation: Every formula in HML is interpreted with an agent of a given labelled transitional system as its model, by structural induction:

(1) T is true under all agents,

(2) ¬A is true under E iff A is false under E,

(3) A ∧ B is true under an agent E if both A and B are true under E,

(4) $[K]A$ is true under an agent E iff for all E' and all a∈$\mathcal{K}$, if E $\xrightarrow{a}$ E' then A is true under E'

Satisfaction: An agent E satisfies a formula A, written as E⊨A iff A is interpreted as true under E. So the satisfaction relation is as follows:

(1) E ⊨ T for all models E,

(2) E ⊨ ¬A iff not E ⊨ A,

(3) E ⊨ A ∧ B iff E ⊨ A and E ⊨ B,

(4) E ⊨[K]A iff for all E' and all a∈$\mathcal{K}$, if E $\xrightarrow{a}$ E' then $E' \models A$

HML logic only can express properties about finite action sequences of agents. However the properties (e.g., deadlock and livelock) about infinite sequences of behaviors are the most important part of asynchronous systems. For example, a simple clock: Clock = tick.Clock ticks forever, but HML cannot express such a property. What we need to cope are fixpoints, and these are supplied by the modal μ-calculus.

3.2.2. Modal μ-Calculus. The modal μ-Calculus is HML plus a fixpoint operator.

Theorem: There will always be at least one solution to the fixpoint equation X = FX provided that each fixed point variable is within the scope of an even number of negations.

This is an easy syntactic check. From now on we assume all our modal μ expressions pass it. There may be several fixpoints, but the minimum fixpoint and maximum

fixpoint are unique. Fortunately these are the two fixpoints we are most interested in. To interpret the minimum and maximum fixed points, we first need to associate with a property expressed in HML the set of states (agents) satisfying it within a given labelled transition system ($\mathcal{P}$, $\mathcal{A}$, $\rightarrow$).

$|| \mathrm{T} || \stackrel{def}{=} \mathcal{P}$

$|| \mathrm{F} || \stackrel{def}{=} \Phi$

$|| \neg A || \stackrel{def}{=} \mathcal{P}$ - $|| \mathrm{A} ||$

$|| \mathrm{A} \wedge \mathrm{B} || \stackrel{def}{=} || \mathrm{A} || \cap || \mathrm{B} ||$

$|| [a]\mathrm{A} || \stackrel{def}{=} \{ \mathrm{P} \in \mathcal{P} \mid \forall P' \in \mathcal{P}, \mathrm{P} \stackrel{a}{\rightarrow} \mathrm{P'} \text{ and } \mathrm{P'} \in ||\mathrm{A}|| \}$

Now we can calculate the sets of agents which satisfy minimum and maximum fixpoints as described below.

$|| \mu\mathbf{X.FX} ||$: The set of agents satisfying the minimum fixpoint μX.FX are computed by:

(1) $X_0 = \Phi$ and i = 0

(2) Repeat i = i+1 and compute $X_i = \mathrm{F}X_{i-1}$
until $X_i = X_{i-1}$

(3) $|| \mu\mathrm{X.FX} || = X_i$

$|| \nu\mathbf{X.FX} ||$: The set of agents satisfying the maximum fixpoint νX.FX are computed by:

(1) $X_0 = \mathcal{P}$ and i = 0

(2) Repeat i = i+1 and compute $X_i = \mathrm{F}X_{i-1}$
until $X_i = X_{i-1}$

(3) $|| \nu\mathrm{X.FX} || = X_i$

Satisfaction relation between an agent and a formula in the modal μ-calculus can be fully defined by adding two statements for the minimum and maximum fixpoints:

$\mathrm{E} \models \mu\mathrm{X.FX}$ iff $\mathrm{E} \in || \mu\mathrm{X.FX} ||$

$\mathrm{E} \models \nu\mathrm{X.FX}$ iff $\mathrm{E} \in || \nu\mathrm{X.FX} ||$

3.2.3. Derived Operators of Modal μ-calculus. Some important derived operators are given below:

$||\mathrm{F}|| = ||\neg \mathrm{T}||$

$||\mathrm{A} \vee \mathrm{B}|| = || \neg(\neg \mathrm{A} \wedge \neg \mathrm{B})||$

$||\langle K \rangle||\mathrm{A} = ||\neg[K]\neg A||$

$||\nu X.FX|| = ||\neg(\mu X.\neg(FX))||$

3.2.4. Definable Interesting Modalities. Some interesting modalities are defined in terms of HML and the minimum and maximum fixed points.

BOX P $\stackrel{def}{=} \nu$Z.P $\wedge$ [$-$]Z

CAN P $\stackrel{def}{=} \nu$Z.P $\wedge$ ($\langle - \rangle$Z $\vee$ [$-$]F)

EVENT P $\stackrel{def}{=} \mu$Z.P $\vee$ ([$-$]Z $\wedge$ $\langle - \rangle$T)

POSS P $\stackrel{def}{=} \mu$Z.P $\vee$ $\langle - \rangle$Z

where

S $\models$ BOX P iff P holds in any state which S can evolve into.

S $\models$ CAN P iff there is at least one path along which S can evolve and all states satisfy P. The path can be finite and end with deadlock.

S $\models$ EVENT P iff there exists one state satisfying P on every path along which S can evolve. The pathes may not deadlock.

S $\models$ POSS P iff there exists one state satisfying P which S can possibly evolve into.

BOX P and POSS P are dual: $\neg$ (BOX P) = POSS $\neg$P,

CAN P and EVENT P are dual: $\neg$ (CAN P) = EVENT $\neg$P

Examples of using these modalities

Safety means that something bad would never happen. This can be expressed as: BOX $\neg$P, where P stands for the bad thing.

Example 1 a deadlock free system S:

S $\models$ BOX $\neg$[-]F, where [-]F means that no action is possible (i.e., deadlock).

Liveness means good thing may happen.

Example 2 In the arbiter (see the definition in section 1.1.1), one desirable property is that every request will eventually be able to get granted. For r1, this can be expressed in the modal μ-calculus as:

Arbiter $\models$ BOX([r1](EVENT($\langle$g1$\rangle$T)))

But we cannot express fairness in the modal μ-calculus. E.g., does the arbiter fairly grant to user 1 and to user 2? This just reflects CCS, since we cannot specify a fair arbiter in CCS.

3.3. The Workbench CWB

The Edinburgh Concurrency Workbench (CWB) is an automated tool which caters for the manipulation and analysis of concurrent systems. Here are some functions we used in our specification and verification.

Agent definition: to define behaviors given in the syntax of CCS.

bi: this command binds a given identifier to a given agent.

State space analyses and equivalence checking: to perform various analyses on these behaviors such as analysing the state space of a given agent, or to check various semantic equivalences and preorders.

sim: this command allows for interactive simulation of a given agent;

vs: this command takes an integer and an agent, and lists all possible observations of the given agent of the given length;

size: this command prints the number of states of a given agent;

min: this command takes an agent and an identifier, and binds to that identifier the agent with the smallest state space which is bisimular to the given agent;

fd: This command takes an agent and tell you if there is any deadlock. If there is a deadlock, it gives a sequence of actions which make the agent evolve into a deadlock.

eq: this command takes two agents and return a boolean value indicating whether or not these two agents are weak bisimular; we only use this to do equivalence check because all our agents are stable and the most interesting observational congruence checking can be reduced to bisimularity checking.

Property definition: to define properties or propositions in the modal μ-calculus.
bpi: this command binds a given identifier to a given proposition;
bmi: this command binds a given identifier to a given propositional macro;

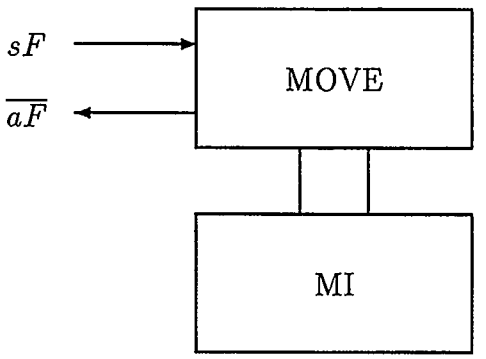
Model checking: to check whether a given agent satisfies a certain property.
cp: this command takes an agent and a property and return a boolean value representing whether or not the agent satisfies the property.

3.4. Summary

In this chapter, we have introduced CCS syntax and semantics. Examples have been given for illustrating how to specify basic circuit modules in CCS and how to look at the meaning of a specification. We also have introduced its companion modal μ-calculus. and their mechanized model checker CWB. Basic temporal operators (macros) are defined in this modal μ-calculus, therefore we can define desirable

properties using these basic macroes instead of directly using the minimum and maximum operators. This makes the specification of properties much easier and more understandable. Examples have been given for showing how to express properties such as deadlock and liveness. Some equivalence relationships have been discussed and it is argued that the observational congruence is the best one for characterizing asynchronous systems. We also have listed commands in the CWB which are used for analyzing and checking our CCS specifications. In the chapter 5 and 6, we will use CCS to specify AMM and PAMM, and the modal μ-calculus to express desirable properties of AMM and PAMM, and then CWB to do model-checking.

CHAPTER 4
The Move Machine



The MOVE machine was first suggested by Sutherland in a 1970's CalTech Report, now presumed lost. Sutherland observed that CPUs spend much of their time moving data back and forth between themselves and memory. Why not have a slave processor to do just that? When the CPU wants a block of data shifted from A to B, it passes details of the request to the MOVE machine and fires it up. When the MOVE machine has completed the move, it sends an acknowledgement back to the CPU and awaits the next request.

It turns out that this small processor has sufficient variety that experimenting with the various design styles is very instructive. This work in this thesis is based upon it. In the later chapters, we shall give the specifications and verifications of variants

of an asynchronous move machine in CCS and the modal μ-calculus, testing them on the CWB. In this chapter, we follow the technical report [BLS+94b] closely, and display its instruction set by giving three example programs.

4.1. Typical Data Movement

We wanted our MOVE machine able to cope with three types of data movement:

(1) Clear: set a block of data starting at address **a** to zero. The program is as follows.

```
L: while k ne n do
   { M[a] := 0;
     a    := a + 1;
     k    := k + 1;
   }
X: halt
```

(2) Copy: copy the data block starting at address **a** to address **b**. The program is as follows.

```
L: while k ne n do
   { M[b] := M[a];
     a    := a + 1;
     b    := b + 1;
     k    := k + 1;
   }
X: halt
```

(3) Compact: copy (and compact) a list starting at address **a** to address **b**. We assume that list items are consecutive words with the data in word 2 and the

next pointer in word 1. Let a point to the head of the list and b point to the head of the compaction area:

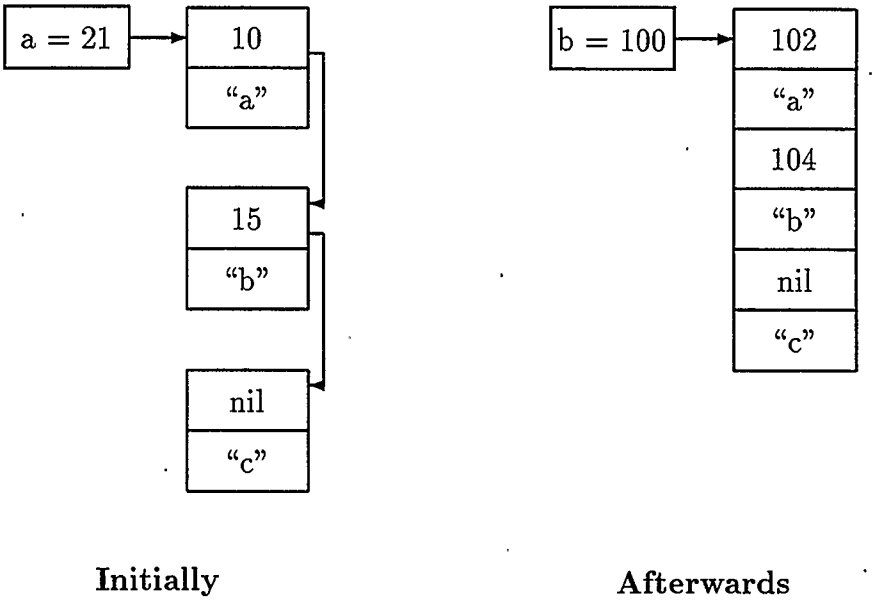


FIGURE 4.1. Move and compact

The program is as follows:

```
L: p      := M[a];
   M[b+1] := M[a+1];
   if p = nil then goto X;
   nb     := b+2;
   M[b]   := nb
   b      := nb;
   a      := p;
   goto L;
X: M[b]   := nil;
   halt;
```

4.2. Instruction Set

David Spooner came up with a simple instruction set able to handle these programs. We have this extended by the addition of LDI so that later we are able to record exceptions such as memory failure and arithmetic overflow. We also change the instruction format from two operators to three operators. This makes it closer in spirit to specifying AMULET [BLP94a, BLP94b, BLGP94, BL94], another major project within the Calgary VLSI group.

code	name	writeback (addr)	reg_1 (data)	reg_2 (data)	action (semantics)
000	LOD	w	r1		w := M[r1]
001	STO		r1	r2	M[r1] := r2
010	MOV	w		r2	w := r2
011	SCC		r1	r2	cc := (r1=r2)
100	INC	w	r1		w := r1+1
101	JCC			r2	if cc then ip := r2
110	LDI	w		constant	w=:constant
111	HLT				

With this instruction set, the above three data movement can be coded as following:

(1) **Program Clear** — Assume the following register initialization and aliasing:

r0 = 0 aliased to constant 0
r1 = 0 aliased to counter K
r2 = N length of the block
r3 = A start address to be zeroed
r4 = L code start
r5 = X code end

and our program reads

```
L : SCC     r1 r2          % cc := k = n ?
    JCC        r5          % if cc then goto X
    STO     r3 r0          % M[a] := 0
    INC r3  r3             % a    := a + 1
    INC r1  r1             % k    := k + 1
    SCC     r1 r1          %
    JCC        r4          % goto L
X : HLT
```

(2) **Program Copy** — Assume the following register initialization and aliasing:

r0 = 0 aliased to counter K
r1 = N length of the block
r2 = A start address of block to be moved
r3 = B start address of receiving area
r4 = L code start
r5 = X code end

and our program reads

```
L : SCC     r0 r1          % cc := k = n ?
    JCC        r5          % if cc then goto X
    LOD  r6 r2             % r6   := M[a]
    STO     r3 r6          % M[b] := r6
    INC  r0 r0             % k    := k + 1
    INC  r2 r2             % a    := a + 1
    INC  r3 r3             % b    := b + 1
    SCC     r1 r1          %
    JCC        r4          % goto L
X : HLT
```

(3) **Program Compact** — Assume the following register initialization and aliasing:

r0	=	nil	
r1	=	A	start of the list to be compacted
r2	=	P	the pointer in this word
r3	=	D	the data in this word
r4	=	L	code start
r5	=	X	code end
r6	=	B	start address of this receiving pair of words
r7	=	NB	r6 +1

The program is as follows.

```
L : LOD r2 r1          %  p := M[a]
    INC r1 r1          %  a := a + 1
    LOD r3 r1          %  d := M[a+1]
    STO    r7 r3       %  M[b+1] := d
    SCC    r0 r2       %  if p = nil then
    JCC       r5       %     goto X
    INC r7 r7          %  nb := b+2
    STO    r6 r7       %  M[b] := b+2
    MOV r6    r7       %  b := b+2
    INC r7 r7          %  nb := b+3
    MOV r1    r2       %  a := p
    SCC    r0 r0
    JCC       r4       %  goto L
X : STO    r6 r0       %  M[b] := nil
    HLT
```

4.3. Abstract Specification of AMM

At this level of abstraction, we specify what AMM is supposed to do but not how to implement it. In 4 phase style, we expect AMM to work as the following:

(1) AMM is initiated by CPU which initializes AMM as required by a certain data movement and then starts AMM by raising sF.

(2) AMM carries out the required data movement, and raises aF when it finishes the current data movement.

(3) CPU checks registers to see if data movement was okay and then lowers sF.

(4) AMM lowers aF to be ready for the next data movement.

A very abstract specification of a 4 phase AMM is:

$$\text{AMM4} = \text{initialize'}.\text{sF}.\text{move'}.\overline{aF}.\text{sF}.\overline{aF}.\text{AMM4}$$

Similarly an abstract specification of the 2 phase AMM is:

$$\text{AMM2} = \text{initialize'}.\text{sF}.\text{move'}.\overline{aF}.\text{AMM2}$$

4.4. Summary

This chapter presents the move machine, a small processor capable of dealing with simple data movements such as setting an area of memory to zero, copying the data from one location to another location, and copying and compacting list linked data into a contiguous area. A set of instructions for carrying out data movements is listed. With this instruction set, the possible code segments for these three data movements are given. We specify AMM at a very abstract level in which we only "black box" what AMM does. We look at register transfer level descriptions of AMM in the following chapters.

CHAPTER 5

AMM — An Asynchronous Move Machine

This chapter systematically presents a 4-phase AMM RTL (register transfer level) implementation through three different abstract levels of CCS specification. Our account is closely based upon [BLS+94b] and [BLS+94a] and is included for contrast and comparison with the PAMM model of chapter 6. At the highest abstract level, we specify what AMM does at the instruction level. At the next level, we give the RTL definitions of the datapath and its control signals, but we keep the control unit quite abstract and do not fix precisely where the control signals come from. At the lowest level, we specify the control unit using basic control modules, and wire these modules and datapath together to detail the control signal flow. This refined specification would serve as a blueprint for engineers. Property checks and equivalence checks are carried out on these specifications. We show that AMM enjoys such desirable properties as deadlock and livelock freedom, liveness, no bus contention, etc. Three different abstract levels are proved to be consistent.

5.1. Abstract Level of Specification

In chapter 4, we gave a top level 4 phase specification of AMM, namely

$$\text{AMM4} = \text{initialize'.sF.move'.'aF.sF.'aF.AMM4.}$$

We now move down one level of abstraction, and detail the semantics of every instruction available for data movement. Looking at the instruction set introduced in

chapter 4, we see that the semantics of the JCC instruction depends upon the current condition code. We distinguish these two cases at two stages in our specification:

- stage 0 (ready for a new data movement), SPEC0 is the state with the condition code false, and SPEC1 is the state with the condition code true.
- stage 1 (starting the cycle of instruction fetch and execution), NEXT0 is the state with the condition code false, and NEXT1 is the state with the condition code true.

We structure the specification by casing over the instructions. We assume the condition code is false when AMM is powered up (i.e., AMM is initiated in state SPEC0). Once started by CPU raising sF, it enters the state NEXT0 for starting the cycle of fetch instruction and execution. At this level of abstraction, Fetch includes two steps: ir' (get the current instruction from the memory) and ip' (increment IP). LOD modifies register file (denoted as rf') and then continues its NEXTi (i=0 or 1). SCC enters NEXT1 if its two operators are the same (denoted as ccT) or enters NEXT0 otherwise. JCC modifies IP (again denoted as ip') if the state is NEXT1 before this cycle is started. Otherwise it does nothing (the PC is automatically incremented as part of fetch). HLT turns the AMM into a state in which AMM raises the acknowledgement signal (aF) to CPU, and gets ready for the next data movement by lowering aF after CPU lowers the sF.

Specification of AMM. The specification of AMM is structured as follows:

```
bi AMM
SPEC0

bi SPEC0
sF.NEXT0

bi SPEC1
```

```
sF.NEXT1

bi NEXT0                                  \ ** cc = F
ir'.ip'.( lod.rf'.NEXT0                   \ ** register file is updated.
        + sto.mem'.NEXT0                  \ ** mem' is memory-writing operation.
        + mov.rf'.NEXT0                   \ ** register file is updated.
        + scc.(ccT.NEXT1 + ccF.NEXT0)     \ ** case split on value in CC.
        + inc.rf'.NEXT0                   \ ** register file is updated.
        + jcc.NEXT0                       \ ** no jump happens.
        + ldi.rf'.NEXT0                   \ ** register file is updated.
        + hlt.'aF.sF.'aF.SPEC0 )

bi NEXT1                                  \ ** cc = T
ir'.ip'.( lod.rf'.NEXT1                   \ ** register file is updated.
        + sto.mem'.NEXT1                  \ ** mem' is memory-writing operation.
        + mov.rf'.NEXT1                   \ ** register file is updated.
        + scc.(ccT.NEXT1 + ccF.NEXT0)     \ ** case split on value in CC.
        + inc.rf'.NEXT1                   \ ** register file is updated.
        + jcc.ip'.NEXT1                   \ ** jump occurs and IP is updated.
        + ldi.rf'.NEXT1                   \ ** register file is updated.
        + hlt.'aF.sF.'aF.SPEC1 )
```

Property Checking: Given a specification of AMM, the next step is to check its behavior on the CWB.

Observable actions of AMM: the sort command lists all observable actions in the tested system.

```
Command: sort AMM
```

```
{ccF,ccT,hlt,inc,ip',ir',jcc,ldi,lod,mem',mov,rf',sF,scc,sto,'aF}
```

Size of AMM state space: the size command gives us a measure of the complexity of a specification.

```
Command: size AMM
AMM has 20 states.
```

Deadlock freedom: the fd command can check to see if a system can deadlock.

```
Command: fd AMM
No such agents.
```

5.2. Middle Level of Specification

We now unfold the design through one level and decompose it into a datapath and a control unit. We further decompose the control unit into two parts: one responsible for the fetch instruction, and the other one responsible for instruction execution.

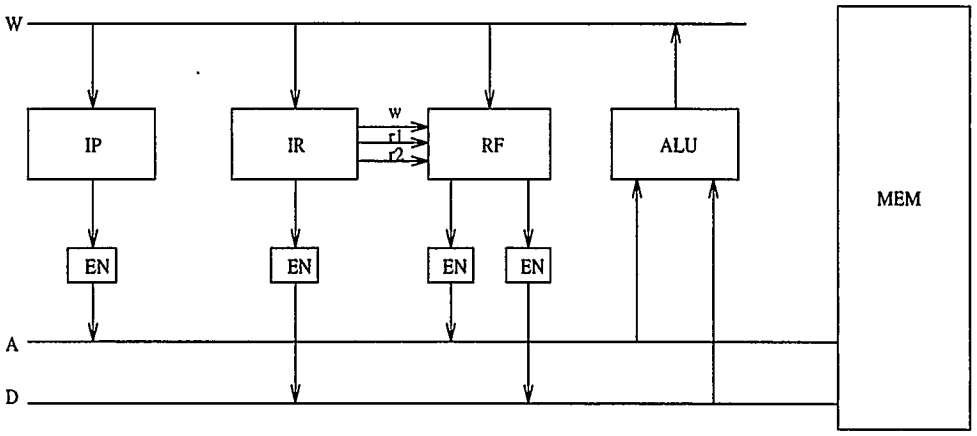


FIGURE 5.1. AMM Datapath

5.2.1. Datapath. There are three buses in the AMM datapath given in Figure 5.1 — the write back bus W, the data bus D and the address bus A. The memory

is shared by instructions and data. The register file has two output (read) ports and one input (write) port. The three control wires indicate the two registers to be read and one register to be written. The ALU carries out arithmetic and logic operations, and passes the data D to W. An instruction pointer register IP maintains the address of the next instruction. An instruction register IR keeps the current instruction. Registers show their outputs strongly except when being written. The EN latches shut off data flow to buses and prevent bus contention.

Our model departs slightly from the original [BLS+94a] in that (i) data is moved from busses A and D to W through the ALU rather than through separate enable elements, and (ii) IR is wired to bus D via an enable element to faciliate the LDI instruction.

5.2.2. Basic Data Path Modules. In this section we introduce the basic data path modules. For the detailed CCS specification and explanations with diagrams and usages, refer to appendix A. In our library of basic data path modules we have the wire, register, register file, boolean register, enable unit, memory and ALU.

WIREs are used to model control signals. The level (high or low) of a WIRE (or WIREs) will determine which function should be carried out. The level of a WIRE is changed when its controller makes a transition. Its current level can be sensed by a stream down circuit.

REG is a register whose output is always strong. Its contents can be written upon request, its value will then be replaced by the current input. It issues an acknowledge signal when the new value is latched and shows strongly.

ENables are placed between registers and busses. In their quiescent state, they block their input. In the enabled state, input passes through and shows strongly. They are normally in their quiescent state. The enabled state is entered upon request and cut off again upon acknowledgement.

BOOL_REG can be set to 1 or cleared to 0, and be tested for its current value.

RF is a register file which has a block of registers. It has two output ports which are

chosen by two addresses r1 and r2, and one input port which is chosen by the address w. Two outputs always have strong data corresponding to RF[r1] and RF[r2]. The register w can be written when requested.

MEM has two connected buses: address bus and data bus, and has a control wire mrw connected to it. When mrw is high, MEM will put the content at the memory location from the address bus on the data bus upon request. When mrw is low, the content at the memory location from the address bus will be replaced by the data on the data bus.

ALU is an algorithmic and logic unit which has two inputs (din1 and din2) and an output. It can carry out the following functions: compare two inputs and set the condition code boolean register, increase din1 and pass to output, and pass din2 to its output. It has two connected control wires alu1 and alu2 to determine which function is actually carried out upon request.

5.2.3. Datapath composition. With all the basic modules introduced in the last section and specified in Appendix A, we are ready to specify the AMM datapath which is the collection of all its components connected by the three busses. We follow the object oriented design style in the sense that every component is an independent object and interacts with others through communication on busses and wires.

```
bi DATAPATH
( REG[rwIP/rwR,awIP/awR,ip'/reg']          \ ** IP
| EN[reIP/reE,aeIP/aeE,sIP/sEN,zIP/zEN]    \ ** ENABLE for IP
| REG[rwIR/rwR,awIR/awR,ir'/reg']          \ ** IR
| EN[reIR/reE,aeIR/aeE,sIR/sEN,zIR/zEN]    \ ** ENABLE for IR
| RF                                       \ ** Register File
| EN[reRF1/reE,aeRF1/aeE,sRF1/sEN,zRF1/zEN \ ** ENABLE for RF[r1]
| EN[reRF2/reE,aeRF2/aeE,sRF2/sEN,zRF2/zEN]\ ** ENABLE for RF[r2]
| ALU                                      \ ** Arithmetic Unit
| BOOL_REG[testcc/test,noj/zero,jmp/one]   \ ** Condition Code Boolean Register
```

```
| WIRE[alu1/in,alu1F/lo,alu1T/hi]          \ ** ALU's
| WIRE[alu2/in,alu2F/lo,alu2T/hi]          \ ** Control Wires
| MEM                                      \ ** Memory
| WIRE[mrw/in,mrwT/hi,mrwF/lo]             \ ** MEM read/write Wire
) \ DATAPATHlines

basi DATAPATHlines
testcc noj jmp alu1T alu1F alu2T alu2F mrwT mrwF
```

Some properties of the datapath:

Observable actions:

```
Command: sort DATAPATH
{alu1,alu2,rA,sA,zA,'aA,
ccT,ccF,testcc,'jmp,'noj, ** related to ALU
mrw,rM,sD,zD,mem',aM,     ** related to memory
rwIP,ip',awIP,            ** related to IP
reIP,sIP,zIP,aeIP,        ** related to EN_IP
rwIR,ir',awIR,            ** related to IR
reIR,sIR,zIR,aeIR,        ** related to EN_IR
rwRF,rf',awRF,            ** related to RF
reRF1,sRF1,zRF1,aeRF1,    ** related to EN_RF1
reRF2,sRF2,zRF2,aeRF2}    ** related to EN_RF2
```

The number of states:

```
Command: size DATAPATH
DATAPATH has 449280000
```

5.2.4. Instructions. The datapath of AMM is composed of the following resources: IP, EN_IP, IR, EN_IR, RF, EN_RF1, EN_RF2, CC, ALU and MEM. All

instructions carried out use some of these resources. In a 4-phase design, these resources are requested for a computation phase (r↑ a↑). The signals are lowered later. There are two control wires alu1 and alu2 for the ALU and one mrw for MEM. These wires are normally low and the actual function of their data module depends upon their levels. ALU carries out three possible functions. One is to pass the value on DBUS to WBUS when both alu1 and alu2 are low. The caller does not raise any of these wires for requesting this function. The second one is to increase the value on DBUS and passes the increased value to WBUS when alu1 is high and alu2 is lower. The caller raises alu1 for requesting this function. The last one is to compare two operators on ABUS and DBUS and set the conditional code register CC when alu2 is high and alu1 is low. The caller raises alu2 for requesting this function.
MEM carries out two possible functions read and write. Read operation when mrw is high. The caller raises mrw for requesting this function. Write operation when mrw is low. The caller does not raise mrw for requesting this function. For convenience, we tabulate the signal names in Table 5.1.

Micro-operations of Each Instruction

Now we go through AMM instructions one by one. Each is a sequence of the above micro-operations and each micro-operation is simply one use of a particular resource. Once we have explained the AMM instructions, generating a description of the controller is quite mechanical. Sequences of micro-operations corresponding to instructions are described as the following.

Fetching instruction (IR'): IR := MEM[IP] is carried out in the following steps:

(1) put the current instruction address on ABUS by enabling EN_IP,

(2) read the instruction from the memory at the address on ABUS to DBUS,

(3) pass it from DBUS to WBUS by requesting ALU pass,

(4) modify IR with the instruction on WBUS.

Micro-operation	Resource Used	Req	Ack	Control
IP:=WBUS	IP	rwIP	awIP	
ABUS:=IP	EN_IP	reIP	aeIP	
IR:=WBUS	IR	rwIR	awIR	
DBUS:=IR	EN_IR	reIR	aeIR	
RF[w]:=WBUS	RF	rwRF	awRF	
ABUS:=RF[r1]	EN_RF1	reRF1	aeRF1	
ABUS:=RF[r2]	EN_RF2	reRF2	aeRF2	
WBUS:=DBUS	ALU	rA	aA	
WBUS:=ABUS+1	ALU	rA	aA	alu1
cc:=(ABUS=DBUS)	ALU	rA	aA	alu2
DBUS:=MEM[ABUS]	MEM	rM	aM	mrw
MEM[ABUS]:=DBUS	MEM	rM	aM	

TABLE 5.1. Micro-operation, Resource and Control Signal

Micro-operations	Signals
ABUS := IP	'reIP.aeIP
DBUS := M[ABUS]	mrw.'rM.aM
WBUS := ABUS	'rA.aA
IR := W	'rwIR.awIR

TABLE 5.2. Micro-operations of IR'

Incrementing IP (IP'): IP := IP + 1 is carried out in the following steps:

(1) put the address in IP on ABUS by enabling EN_IP,

(2) increase the address on ABUS and pass it to WBUS by requesting an ALU increment,

(3) modify IP with the increased value on WBUS.

Micro-operations	Signals
ABUS := IP	'reIP.aeIP
WBUS := ABUS + 1	alu1.'rA.aA
IP := W	'rwIP.awIP

TABLE 5.3. Micro-operations of IP'

LOD w r1: RF[w] := MEM[RF[r1]] is carried out in the following steps:

(1) put the memory address on ABUS by enabling EN_RF1,

(2) get the value from the memory at ABUS to DBUS by requesting memory read,

(3) pass the data on DBUS to WBUS by requesting ALU pass,

(4) store the data on WBUS into the register file by requesting a register file write.

Micro-operations	Signals
ABUS := RF[r1]	'reRF1.aeRF1
DBUS := MEM[ABUS]	mrw.'rM.aM
WBUS := DBUS	'rA.aA
RF[w] := WBUS	'rwRF.awRF

TABLE 5.4. Micro-operations of LOD

STO r1 r2: MEM[RF[r1]] := RF[r2] is carried out in the following steps:

(1) put the memory address on ABUS by enabling EN_RF1,

(2) put the data on DBUS by enabling EN_RF2,

(3) store the data on DBUS into the memory at the address on ABUS by requesting memory write.

MOV w r2: RF[w] := RF[r2] is carried out in the following steps:

(1) put RF[r2] on DBUS by enabling EN_RF2,

(2) pass the data on DBUS to WBUS by requesting ALU pass,

Micro-operations	Signals
ABUS := RF[r1]	'reRF1.aeRF1
DBUS := RF[r2]	'reRF2.aeRF2
M[ABUS] := DBUS	'rM.aM

TABLE 5.5. Micro-operations of STO

(3) store the data on WBUS into the register file.

Micro-operations	Signals
DBUS := RF[r2]	'reRF2.aeRF2
WBUS := DBUS	'rA.aA
RF[w] := WBUS	'rwRF.awRF

TABLE 5.6. Micro-operations of MOV

SCC r1 r2: CC := (RF[r1]==RF[r2]) is carried out in the following step:

(1) put the first operator on ABUS by enabling EN_RF1,

(2) put the second operator on DBUS by enabling EN_RF2,

(3) compare two operators and set the condition code register by requesting ALU compare.

Micro-operations	Signals
ABUS := RF[r1]	'reRF1.aeRF1
DBUS := RF[r2]	'reRF2.aeRF2
CC := (RF[r1]==RF[r2])	alu2.'rA.aA

TABLE 5.7. Micro-operations of SCC

INC w r1: RF[w] := RF[r1] + 1 is carried out in the following steps:

(1) put RF[r1] on the ABUS by enabling EN_RF1,

(2) increase the value on ABUS and pass to WBUS by requesting an ALU increment,

(3) store the value on WBUS into the register file by requesting register file write.

Micro-operations	Signals
ABUS := RF[r1]	'reRF1.aeRF1
WBUS := ABUS + 1	alu1.'rA.aA
RF[w] := WBUS	'rwRF.awRF

TABLE 5.8. Micro-operations of INC

JCC r2: if cc is true then IP := RF[r2], and if cc is false it does nothing. This is carried out in the following steps:

(1) check the condition code by testcc, if it is true it continues to do the following steps, otherwise they are omitted.

(2) put the new address RF[r2] on DBUS by enabling EN_RF2,

(3) pass it from DBUS to WBUS by requesting ALU pass,

(4) modify IP with the data on WBUS by requesting IP write operation.

Micro-operations	Signals
check condition code	'testcc.(jmp + noj)
DBUS := RF[r2]	'reRF2.aeRF2
WBUS := DBUS	'rA.aA
RF[w] := WBUS	'rwRF.awRF

TABLE 5.9. Micro-operations of JCC

LDI w i: RF[w] := i is carried out in the following steps:

(1) put the immediate value on DBUS by enabling EN_IR,

(2) pass it from DBUS to WBUS by requesting ALU pass,

(3) store the value on WBUS into the register file by requesting register file write.

Micro-operations	Signals
DBUS := ir	'reIR.aeIR
WBUS := DBUS	'rA.aA
RF[w] := WBUS	'rwRF.awRF

TABLE 5.10. Micro-operations of LDI

5.2.5. Fetch Unit. The fetch unit organizes control signals for guaranteeing the correct micro-operation sequence of fetching an instruction (IR'), incrementing IP (IP'), activating the execute unit to execute the current instruction and finally deciding to continue this cycle or terminate this data movement depending on the signal from the execute unit which indicates whether a HALT instruction has been executed. The specifications of IR' and IP' are simply sequences of the signals listed in the tables in the last subsection:

```
bi IR'
rIR'.'reIP.aeIP.mrw.'rM.aM.'rA.aA.'rwIR.awIR.'aIR'.IR'

bi IP'
rIP'.'reIP.aeIP.alu1.'rA.aA. 'rwIP.awIP.'aIP'.IP'
```

The fetch unit behaves as follows:

When the fetch unit is started by CPU raising sF, it first computes and flattens IR', and then compute and flatten IP' for instruction fetch and IP increment as specified in the following code:

```
bi FET
```

```
sF.READ_INST

bi READ_INST
'rIR'.aIR'.'rIR'.aIR'.'rIP'.aIP'.'rIP'.aIP'.PRE_DEC
```

The fetch unit drives RF[r1] onto WBUS before decoding instruction during the instruction execution because no other source needs to use WBUS during this period. This makes instruction specifications simpler since we do not need to enable it later. Then it activates the execute unit by sending sE.

```
bi PRE_DEC
'reRF1.aeRF1.'sE.FET_END
```

Now the fetch unit waits for a signal from the execute unit. If the signal cF is raised, this means the fetch unit will continue to read a new instruction. In this case it brings down all raised signals and repeats READ_INST. If eP is raised, this means that this data movement has been finished. Then it should send acknowledge signal aF back to CPU, brings down all raised signals and get ready for the next data movement.

```
bi FET_END
   cF.'reRF1.aeRF1.'sE.cF.READ_INST   \
+  eP.'reRF1.aeRF1.'sE.eP.'aF.sF.'aF.FET
```

The fetch unit consists of IR', IP' and FET.

```
bi FETCH
( FET | IP' | IR' )\{rIP',aIP',rIR',aIR'}
```

5.2.6. Execute Unit. The execute unit is used to issue all control signals for ensuring the correct micro-operation sequence of instruction execution. It is further decomposed into a decode unit and all instruction bodies. When the decode part is started by the fetch unit raising sE, it will decode the current instruction. If the current instruction is not HLT, it computes this instruction and informs the fetch unit by raising cF when the computation is done, and then flattens this instruction

when sE is lowered. If the current instruction is HLT, it computes this instruction and informs the fetch unit by raising eP when the computation is done, and then flattens this instruction when sE is lowered. Its CCS specification is given as:

```
bi DEC_EXEC
sE.EXEC1

bi EXEC1
   lod.'rLOD.aLOD.'cF.sE.'rLOD.aLOD.'cF.DEC_EXEC     \
+  sto.'rSTO.aSTO.'cF.sE.'rSTO.aSTO.'cF.DEC_EXEC     \
+  mov.'rMOV.aMOV.'cF.sE.'rMOV.aMOV.'cF.DEC_EXEC     \
+  scc.'rSCC.aSCC.'cF.sE.'rSCC.aSCC.'cF.DEC_EXEC     \
+  inc.'rINC.aINC.'cF.sE.'rINC.aINC.'cF.DEC_EXEC     \
+  jcc.'rJCC.aJCC.'cF.sE.'rJCC.aJCC.'cF.DEC_EXEC     \
+  ldi.'rLDI.aLDI.'cF.sE.'rLDI.aLDI.'cF.DEC_EXEC     \
+  hlt.'rHLT.aHLT.'eP.sE.'rHLT.aHLT.'eP.DEC_EXEC
```

Each instruction is responsible for carrying out its associated sequence of micro-operations. We specify each instruction simply by sequencing all control signals in its corresponding table in subsection 5.3.4. Here we should remember that ABUS =: RF[r1] has been carried out before decoding. So we do not need to enable EN_RF1 for individual instructions any more.

For example,

```
bi LOD
rLOD.'mrw.'rM.aM.'rA.aA.'rwRF.awRF.'aLOD.LOD
```

when rLOD arrives from the decode unit, the execution of LOD w r1 starts with mrw.'rM.aM to read the value from the memory at the address ABUS (which has been driven by RF[r1] in advance) to DBUS, then 'rA.aA to pass the value on DBUS to WBUS, and finally 'rwRF.awRF to store the value on WBUS into RF[w]. The

computation stage is finished off by 'aLOD. The decode unit will flatten this instruction by a second pass through the same signals.

The other instructions are specified in the same way:

```
bi STO
rSTO.'reRF2.aeRF2.'rM.aM.'aSTO.STO
bi MOV
rMOV.'reRF2.aeRF2.'rA.aA.'rwRF.awRF.'aMOV.MOV
bi SCC
rSCC.'reRF2.aeRF2.'alu2.'rA.aA.'aSCC.SCC
bi INC
rINC.'alu1.'rA.aA.'rwRF.awRF.'aINC.INC
bi JCC
rJCC.'testcc.JCC1
bi JCC1
jmp.'reRF2.aeRF2.'rA.aA.'rwIP.awIP.'aJCC.JCC2 + noj.'aJCC.JCC3
bi JCC2
rJCC.'testcc.jmp.'reRF2.aeRF2.'rA.aA.'rwIP.awIP.'aJCC.JCC
bi JCC3
rJCC.'testcc.noj.'aJCC.JCC
bi LDI
rLDI.'reIR.aeIR.'rA.aA.'rwRF.awRF.'aLDI.LDI
bi HLT
rHLT.'aHLT.HLT
```

Finally the execute unit is given by wiring the decode unit and all instructions, and hiding internal connections in Elines.

```
basi Elines
rLOD rSTO rMOV rSCC rINC rJCC rLDI rHLT    \
aLOD aSTO aMOV aSCC aINC aJCC aLDI aHLT
```

```
bi EXEC
(DEC_EXEC|LOD|STO|MOV|SCC|INC|JCC|LDI|HLT) \ Elines
```

5.2.7. AMM. We have now specified the datapath, the fetch unit and the execute unit. AMM is merely their composition, hiding all internal signals.

```
bi AMM
(DATAPATH|FETCH|EXEC) \ Mlines
basi Mlines
  rwIP rwIR rwRF rM rA reIP reIR reRF1 reRF2  \
  awIP awIR awRF aM aA aeIP aeIR aeRF1 aeRF2  \
  alu1 alu2 mrw testcc ccT ccF sE cF eP
```

5.2.8. Property Checking. Any asynchronous system is prone to deadlock, progress and safety problems. We can express and check such properties using the modal μ-calculus, as mechanized in the CWB. This subsection provides checks for some typical desirable properties.

Minimization: An asynchronous system is built by connecting subcomponents. To check properties of a large system, we first generate an equivalent machine with the minimized number of states. In general, an agent and its minimized version are weakly bisimular. In our case, they are observationally congruent since all our agents are stable.

We minimize AMM by first minimizing its components: the datapath to DATAPATH', and the fetch unit to FETCH' and the execute unit to EXEC'.

```
Command: bi AMM
Agent: (DATAPATH' | FETCH' | EXEC') \ Mlines
Command: min AMM
```

```
Save result in identifier: AMM'
AMM' has 90 states.
```

Deadlock Freedom: If a system can always make a move, it is deadlock free. In the modal μ-calculus, deadlock free can be expressed by: BOX ⟨-⟩T.

```
Command: fd AMM'
No such agents.
```

Livelock Free: If a system never could get into a state from which it can do internal actions forever, it is livelock free.

```
Command: cp AMM
proposition: BOX (max(X.<t>X))
false
```

Safety: Something bad never happens.

Safety 1: every driving source drives buses in the following way: once it drives a bus, it won't drive the bus again before tristating the bus. We use macro CYCLE2 to express this property. For example, CYCLE2 sIP zIP means that zIP has to occur after one sIP has happened and before the next sIP will happen.

```
Command: cp AMM'
Proposition: CYCLE2 sIP zIP
true
```

Similarly, we have successfully tested this property for the following pairs: sIR and zIR, sRF1 and zRF1, sRF2 and zRF2, sA and zA, sD and zD.

Safety 2: one cannot reach a state in which different sources may drive the same bus.

Test bus A: There are two sources EN_IP and EN_RF1 which can drive bus A via requests on sIP and sRF1.

```
Command: cp AMM'
Proposition: BOX (~(<sIP> T & <sRF1>T))
```

```
true
```

Test bus D: There are three sources EN_IR, EN_RF2 and MEM which can drive bus D via requests on sIR, sRF2 and sD.

```
Command: cp AMM'
Proposition: BOX (~(<sIR>T & <sRF2>T & <sD>T))
true
Command: cp AMM'
Proposition: BOX (~(<sIR>T & <sRF2>T))
true
Command: cp AMM'
Proposition: BOX (~(<sIR>T & <sD>T))
true
Command: cp AMM'
Proposition: BOX (~(<sD>T & <sRF2>T))
true
```

Test bus W: only ALU drives this bus (via sA). So no bus contention is possible.

Safety 3: Once a bus is driven, it must be tristated before it will be driven again. We can use macro NEC_FOR a P to express this property. This macro means that a action is necessary for P to hold.

Test bus A: Once it is driven by sIP, it is necessary to be tristated by zIP before sIP or sRF1 is possible. Similar test has been successfully carried out when it is driven by sRF1.

```
Command: cp AMM'
Proposition: BOX ([sIP] (NEC_FOR zIP <sIP,sRF1>T))
true
```

Test bus D: Once it is driven by sIR, it is necessary to be tristated by zIR before sIR, sRF2 or sD is possible. Similar tests are successful for sRF2 and sD.

```
Command: cp AMM'
Proposition: BOX ([sIR] NEC_FOR zIR <sIR,sRF2,sD>T)
true
```

Safety 4: AMM never acknowledges back to CPU before it executes a halt instruction.

```
Command: cp AMM'
Proposition: NEC_FOR hlt 'aF
true
```

Liveness:

Strong liveness means that something eventually happens. This can be expressed as BOX EVENT P.

Weak liveness means that it is always possible for something to happen. This can be expressed as BOX POSS P.

Liveness 1: the actions related to Fetch are strong live transitions. They always eventually happen (every instruction requires to fetch IR and increase IP).

```
Command: cp AMM'
Proposition: BOX EVENT <ir'>T
true
```

```
Command: cp AMM'
Proposition: BOX EVENT <ip'>T
true
```

Liveness 2: the 8 instructions are all weak live transitions. For example, a LOD instruction is a weak live transition and can be tested below (a program may not contain a LOD). Similar tests have been successfully carried out for all other instructions.

```
Command: cp AMM'
Proposition: BOX POSS <lod>T
```

```
true
```

5.2.9. Equivalence Checking. We have given two different abstract levels of specification. When we systematically develop a system hierachically, we have to make sure that a lower concrete level faithfully represents its higher more abstract level. To check this consistency, we need to hide actions specific to the lower level because these actions are not observable at the abstract level. We accomplish this below by composing AMM with the agent R and hiding Rlines.

```
bi R
'sA.R + 'sD.R + 'sIP.R + 'sIR.R + 'sRF1.R + 'sRF2.R +  \
'zA.R + 'zD.R + 'zIP.R + 'zIR.R + 'zRF1.R + 'zRF2.R
basi Rlines
sA zA sD zD sIP zIP sIR zIR sRF1 zRF1 sRF2 zRF2
Command: bi AMM_R
Agent: (AMM'|R) \ Rlines
Command: min AMM_R
Save result in identifier: AMM_R'
AMM_R' has 20 states.
Command: fd AMM_R'
No such agents.
```

The consistency between two levels means that they are weak bisimular. In our case they are also observationally congruent since all agents are stable.

```
Command: eq
Agent: SPEC0
Agent: AMM_R'
true
```

5.3. Lowest Level of Specification

At the middle level of AMM specification, we specified a control unit by giving, for each instruction, the order in which the control signals activate the datapath resources. In this section, we go down one further level of abstraction and detail where the control signals come from by wiring basic control modules and the datapath together. The control unit consists of a fetch unit, an execute unit and a call box which routes the datapath access signals between the fetch or execute unit and datapath and enables datapath elements to be shared. We also check properties and prove that it is consistent with the middle level of specifications and hence with the abstract level.

5.3.1. Basic Control Modules. In this subsection we introduce the basic control modules. For their detailed specification and explanation, see chapters 1 and 3.

C is a rendezvous element which generates a transition after both of its inputs arrive regardless of their order.

C' is a bubble C element which assumes one input transition has already arrived at the beginning and will have the same behavior as C.

M is a merge element which generates a transition whenever it receives a transition at either of its two inputs.

FORK is an element which produces transitions at both outputs once it receives a transition at its input. The order of transitions at two outputs is arbitrary.

FASTFK is an element which is similar to FORK except that the order of transitions at two outputs is fixed. This is used to reduce the number of states in CCS specification.

CALL element is used to solve multi-user problem. When a resource is shared by two or more users who never contend at the same time, CALL element is put between the resource and its users. This ensures the acknowledge signal goes back to the calling

user.

S is van Berkel's S element. It is used to call a sub-circuit twice (once to compute, once to flatten).

5.3.2. Fetch Unit. The fetch unit is implemented in Figure 5.2. IR' and IP' in the fetch unit access datapath modules to carry out instruction fetch and IP increment. Some modules are shared by IP', IR' and instruction executions. Every datapath module, once activated by one certain user, should send the acknowledge signal back to the calling user. We delay the analysis of IR' and IP' datapath accesses until subsection 5.3.4 where we will examine all datapath accesses by IR', IP' and all instruction executions, and solve the resource sharing problem by using call modules. Here the fetch unit specification doesn't include the real access to data path.

As shown in Figure 5.3, the fetch unit sequentially organizes the activities: (i) IR', (ii) IP', and (iii) instruction execution. These activities will be carried out through CALL elements.

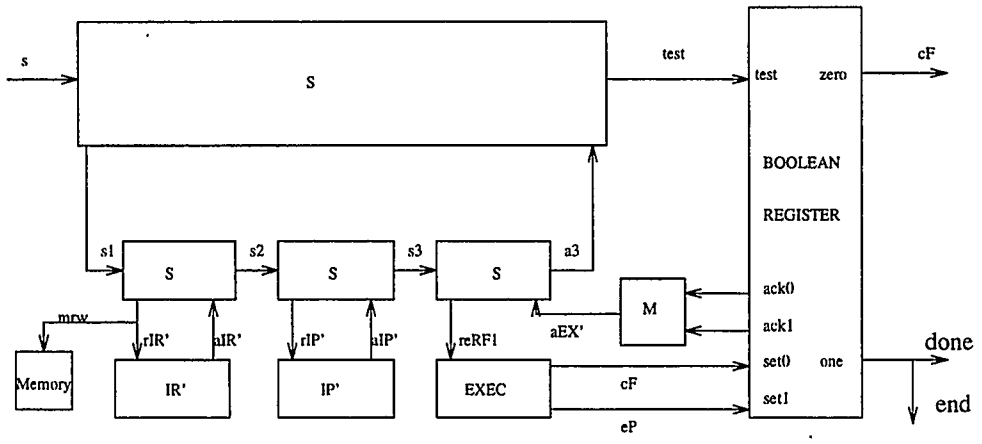


FIGURE 5.2. Main Part of Fetch Unit

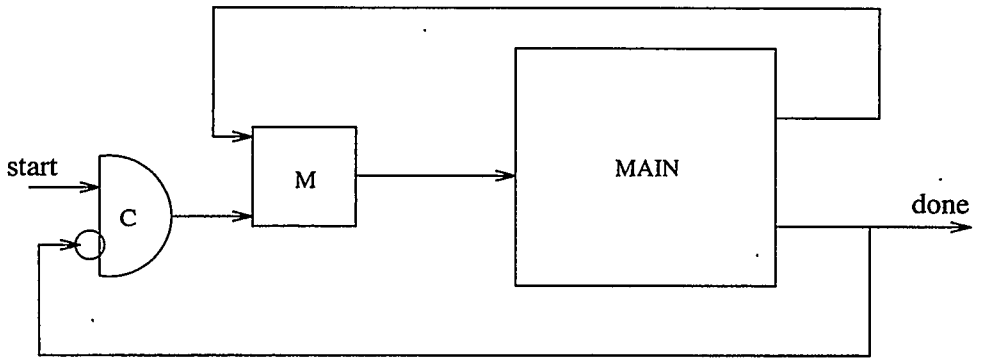


FIGURE 5.3. Implementation of Fetch Unit

```
bi FETCHimp
( C'         [ sF/a, end/b, in/z ]          \
| M2         [ zero/a, in/b, s/z   ]        \
| FORK       [ one/a, aF/b, end/c ]         \
| S          [ s/s, s1/r, a3/a, test/d  ]   \
| S          [ s1/s, setM/r, aIR'/a, s2/d ] \
| FASTFK     [ setM/a, mrw/b, rIR'/c ]      \
| S          [ s2/s, rIP'/r, aIP'/a, s3/d ] \
| S          [ s3/s, reRF1/r, aEX'/a, a3/d] \
| M2         [ ack0/a, ack1/b, aEX'/z ]     \
| BOOL_REG   [ cF/set0, eP/set1 ]           \
) \ Fetchlines

basi Fetchlines
in zero s end s1 s2 s3 a3 ack0 ack1 aEX' test one  setM
```

5.3.3. Execute Unit. The execute unit is to organize instruction execution, as given in Figure 5.4. Instruction executions are carried out through CALL modules. All datapath accesses will be dealt in the next subsection.

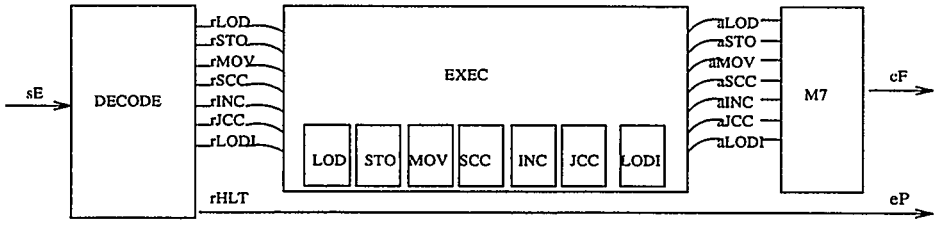


FIGURE 5.4. Implementation of Execute Unit

```
bi M7
a1.'z.M7 + a2.'z.M7 + a3.'z.M7 + a4.'z.M7 + a5.'z.M7 + a6.'z.M7 + a7.'z.M7

bi EXECimp
   sE.EXECimp1

bi EXECimp1
  lod. 'rLOD.sE.'rLOD.EXECimp      \
+ sto. 'rSTO.sE.'rSTO.EXECimp      \
+ mov. 'rMOV.sE.'rMOV.EXECimp      \
+ scc. 'rSCC.sE.'rSCC.EXECimp      \
+ inc. 'rINC.sE.'rINC.EXECimp      \
+ jcc. 'rJCC.sE.'rJCC.EXECimp      \
+ ldi. 'rLDI.sE.'rLDI.EXECimp      \
+ hlt. 'rHLT.sE.'rHLT.EXECimp
```

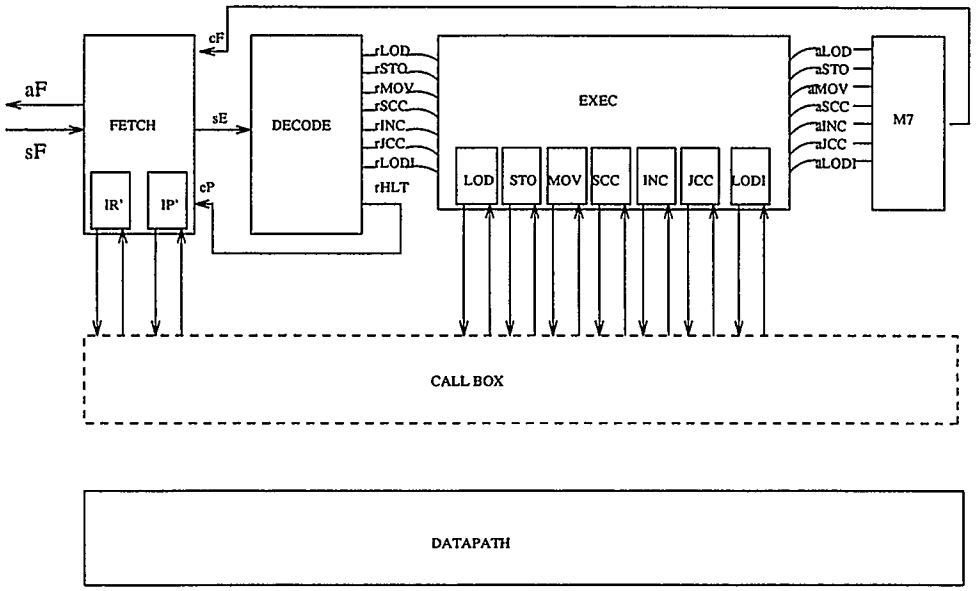


FIGURE 5.5. AMM Architecture

5.3.4. Call Box. AMM architecture is given in Figure 5.5. Now we examine the datapath accesses of the fetch unit (through IR' and IP'), and the execute unit (through all instruction executions). Each instruction involves a sequence of micro-operations. Each micro-operation accesses exactly one basic datapath module. According to the micro-operation tables of instructions in subsection 5.2.4, we can give the data modules accessed by a particular instruction in Table 5.11. According to this table, we can calculate how many users share a particular data module, as shown in its reverse table 5.12. Having this resource use table, we can easily construct a call element for each data module. For example, MEM is shared by three instructions IR', LOD and STO, and CALL3 element will be put in front of the MEM for three users to share. Therefore the call box is specified as:

```
bi CALLS
   ( C2IP  |  C4RF  |  C2EN_IP  |  C4EN_RF2  |  C8ALU  |  C3MEM  )
```

Instruction	Resource
IR'	EN_IP, MEM,ALU,IR
IP'	EN_IP, ALU, IP
LOD	EN_RF1, MEM, ALU, RF
STO	EN_RF1, EN_RF2, MEM
MOV	EN_RF2, ALU, RF
SCC	EN_RF1, EN_RF2, ALU
INC	EN_RF1, ALU, RF
JCC	if cc=T then EN_RF2, ALU, IP
LDI	EN_IR, ALU, RF
HLT	

TABLE 5.11. Datapath Accesses by Instructions

Where C2IP is a CALL2 element for IP, C4RF is a CALL4 element for RF, C2EN_IP is a CALL2 element for EN_IP, C4EN_RF2 is a CALL4 element for EN_RF2, C8ALU is a CALL8 element for ALU, and C3MEM is a CALL3 element for MEM.

5.3.5. Control Unit. The control unit is a unit by connecting fetch unit, execute unit and their CALL box.

```
bi SETMRW
   FASTFK [ rLOD/a, mrw/b, rLODf/c ]
bi SETALU1IP
   FASTFK [ ip1/a, alu1/b, ip1f/c ]
bi SETALU1INC
   FASTFK [ rINC/a, alu1/b, rINCf/c ]
bi SETALU2
   FASTFK [ scc1/a, alu2/b, scc1f/c ]
bi WHLT
```

Resource	Instruction	Calls
IP	IP', JCC (cc =T)	2
IR	IR'	1
RF	LOD, MOV, INC, LDI	4
EN_IP	IR', IP'	2
EN_IR	LDI	1
EN_RF1	LOD, STO, SCC, INC	4
EN_RF2	STO, MOV, SCC, JCC	4
ALU	IR',IP', LOD, MOV, SCC, INC,JCC, LDI	8
CC	JCC	1
MEM	IR', LOD, STO	3

TABLE 5.12. Resource Uses

```
   rHLT.'eP.WHLT
bi MAJCC
   M2  [ jcc4/a,  noj/b, aJCC/z ]
bi McF
   M7 [ aLOD/a1, aSTO/a2, aMOV/a3, aINC/a4, aSCC/a5, aJCC/a6, aLDI/a7, cF/z]
bi CONTROLimp
   ( FETCHimp | EXECimp | SETMRW | SETALU1INC | SETALU1IP | SETALU2  \
   | CALLS | WHLT | McF | MAJCC) \ Contrlines
basi Contrlines
   rLOD rSTO rMOV rSCC rINC  rHLT                \
   aLOD aSTO aMOV aSCC aINC aJCC aLDI            \
   jcc2 jcc3 jcc4 ip1 ip1f ip2 ldi2 rINCf inc1   \
   mov1 mov2  rLODf lod1 lod2 ir1 ir2 sto1 scc1f \
   scc1 rIR' aIR' aIR' rIP' aIP'
```

5.3.6. AMM. The AMM is the composition of the datapath and the control unit. Its CCS specification is given as the following:

```
bi AMMimp1
   ( CONTROLimp                                                    \
   | DATAPATH[ rLDI/reIR, aIR'/awIR, sE/aeRF1,rJCC/testcc]         \
   ) \ MOVElines

basi MOVElines
   rA rM reIP rLDI rwIP rwIR reRF1 reRF2 rwRF     \
   aA aM aeIP aeIR awIP  sE  aeRF1 aeRF2 awRF     \
   jmp noj rJCC mrw alu1 alu2 cF eP rwIR
```

AMMimp1 is started by the signal *start* and finish off a data movement by issuing the signal *done*. If we use *start* as sF and *done* as aF, a second sF can be accepted before an aF is issued. Obviously it is not equivalent to our more abstract levels of specification, which guarantee the order sF $\rightarrow$ 'aF $\rightarrow$ sF $\rightarrow$ 'aF. So we can not use *start* as sF and *done* as aF. Some restrictions should be put on the order of sF and aF, as below:

```
bi RES
sF.'start.done.'aF.RES

bi AMMimp
( RES | AMMimp1 ) \ {start, done}
```

5.3.7. Property Checking. Here we minimize AMMimp and check it for deadlock.

Minimization:

```
Command: min FETCHimp
Save result in identifier: FETCHimp'
```

```
FETCHimp' has 716 states.

Command: min AMMimp
Save result in identifier: AMMimp'
AMMimp' has 90 states.
```

Deadlock Freedom:

```
Command: fd AMMimp'
No such agents.
```

5.3.8. Equivalence Checking. The equivalence of this level and the middle level are checked in the following. Since we have proved the most abstract level and the middle level are equivalent in the last section, three levels of AMM specification are consistent.

```
Command: eq
Agent: AMM'
Agent: AMMimp'
true
```

5.4. Summary

This chapter systematically develops AMM through three abstract levels which are proven consistent. It is a reworking [BLS+94a] with a slightly modified instruction set and datapath. The most abstract level only addresses what AMM is supposed to do at the instruction level. The next level spells out its datapath and points out all control signals and when they should be sent to the datapath by the control unit. The lowest level details all the control signal flows by wiring basic control modules and the datapath together. This level of specification is clean enough and clear enough to serve as an implementation guide. T.Borsodi, a graduate student in ECE, took a CCS description of AMM [BLS+94b] and [BLS+94a] and implemented it in Xilinx FPGA technology in a matter of 2 or 3 weeks in 1993. In the manner of [BLS+94a],

we have also proved that AMM enjoys desirable properties such as deadlock freedom, livelock freedom, no bus contention, liveness and safety.

CHAPTER 6

PAMM — A Pipelined Asynchronous Move Machine

This chapter gives the specification of a 2 phase pipelined asynchronous move machine. In the asynchronous move machine we designed in chapter 5, the operations were sequential and did not overlap. One of the advantages of an asynchronous system is that components usually work in parallel and only occasionally need to cooperate with others by communication. We now consider a 2 phase pipelined move machine which implements this inherent parallelism. We decompose PAMM to two modules: a fetch stage and an execute stage. We also prove it to be deadlock free and possess the required liveness properties.

6.1. PAMM Architecture

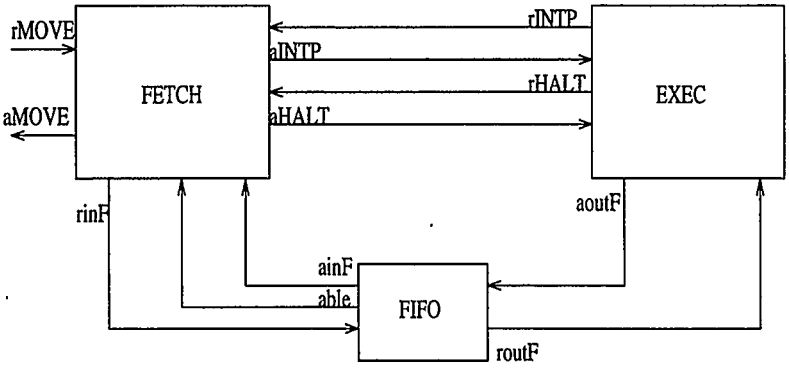


FIGURE 6.1. PAMM

As shown in Figure 6.1, PAMM consists of two major modules: a fetch module and an execute module separated by a FIFO. The fetch module expects code to be sequential. It speculatively generates sequential PC values, and pushes the corresponding instruction into the FIFO buffer. The execute module takes instructions one by one from the FIFO and executes them. We posit a Harvard architecture with separate instruction and data memories.

What happens when the execute module meets a jump instruction which is taken? In this case the sequential anticipation is wrong and none of the prefetched instructions following the jump instruction should be executed. The halt instruction has similar problems. These can be solved with a global arbiter and a colour register in each module. The arbiter allows three contenders to arrive asynchronously: sequential anticipation, instruction flow change, and stop. The colour registers indicate the parity of the current instruction sequence. Each fetched instruction carries the current colour when it is pushed into the FIFO. Once a jump or halt is taken, the execute module flips its colour register and requests the arbiter. When this request gains control, the sequential generation of PC values is blocked in the fetch module. The colour register is flipped and the new value from the execute unit becomes the current PC. The fetch unit then generates instructions from the new PC base and with the flipped colour. The execute unit automatically discards all unwanted instructions by comparing the colour of the next instruction with its own current colour. We assume that both colour registers (the one in the fetch stage and the one in the execute stage) have the same colour after PAMM is powered up.

6.2. 2-Phase Basic Modules

This section introduces basic control modules and datapath modules.

6.2.1. Control Modules. In our library, we have FORK and FASTFK element, C element and C' the bubbled C element, WAIT element, CALL and ARBITER.

FORK, FASTFK, C and C' elements are the same as introduced in chapter 5. CALL is basically the same as the 4-phase except that a 2-phase CALL element does not have the flatten stage.

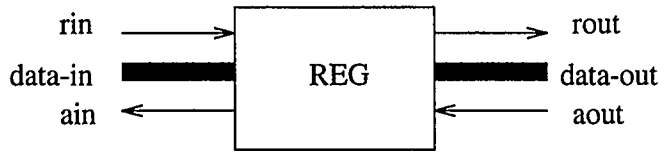
WAIT element is for holding an transition until a certain condition is satisfied.

The ARBITER element is put between a resource and its users who share this resource. The ARBITER allows exactly one user and blocks the others. Only after the allowed user is done, the ARBITER will accept another waiting user.

6.2.2. Datapath Modules. In our library, we have register REG, incrementor INC, memories IMEM and DMEM, a first-in-first-out queue FIFO, a counter WC and ALU.

Register

A register is a one position buffer which can accept a new value when it is empty



and can be read when it has a value.

Specification:

```
LR = rin.nf.inc.'ain.LR

C0 = 'inc.C1 + 'nf.C0

C1 = 'dec.C0 + 'ne.C1

RR = ne.'rout.aout.dec.RR

REG = ( LR | C0 | RR ) \ {inc,dec,ne,nf}
```

Its writer should follow the pattern:

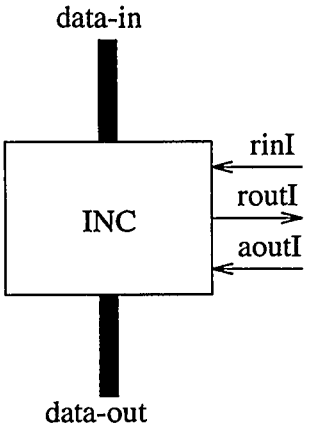
WRITER = ...prepare data.'rin.ain...

Its reader follows the pattern:

READER = ...rout.process data.'aout...

Incrementer

An incrementer is a very trivial element which accepts an input d, produces d+1



and gets ready for repeating this procedure when d+1 has been used.

Specification:

INC = rinI.'routI.aoutI.INC

The caller follows the pattern:

caller = ...prepare data.'rinI.routI.latch new data.'aoutI...

IMEM

IMEM is a memory with reduced functionality. PAMM only reads instructions from it. So we hide its write operation to reduce the complexity of PAMM.

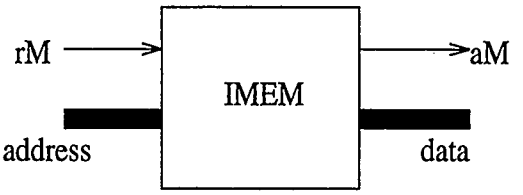
Specification:

IMEM = rIM.'aIM.IMEM

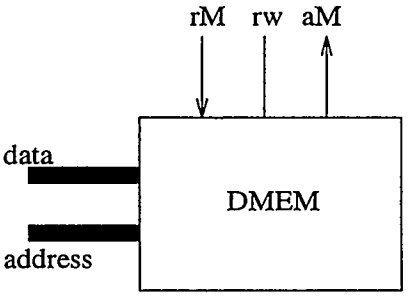
The caller follows the pattern:

caller = ... prepare address.'rM.aM ...

DMEM



DMEM is a traditional memory with both write and read operations. We consider



the control wire as a 1-bit local bus driven by its controller.

Specification:

DMEM = rM.(rw=1.'aM.MEM + rw=0.mem'.'aM.MEM)

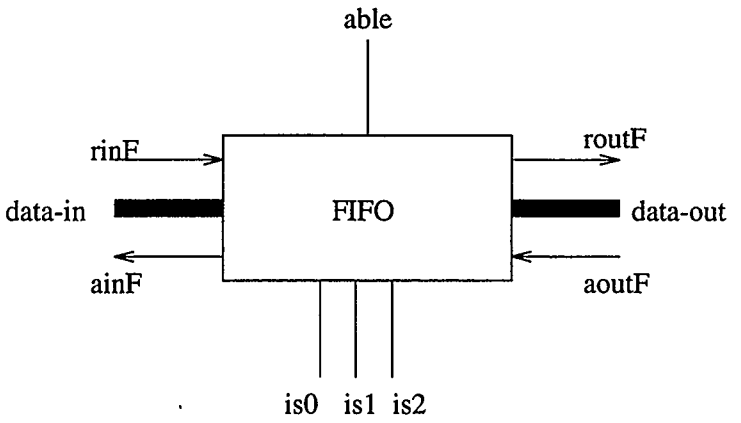
Its caller follows the pattern:

Write = ... prepare data and address.drive rw with 0.'rM.aM ...

Read = ... prepare address.drive rw with 1.'rM.aM ...

FIFO

FIFO is a first in and first out queue which can accept a new value when it is not full and can output a data when it is not empty. The user can check how many positions are occupied. Here FIFO have a signal "able" which indicates that there is two space left. In our PAMM we need to leave a space for dealing with a jump instruction which arrives asynchronously. Every time we need to check "able" before a new request for the next sequential instruction is issued. This means that FIFO always leaves one space dealing with the jump instruction.



Specification:

```
LF = rinF.nf.'inc.'ainF.LF

RF = ne.'routF.aoutF.'dec.RF

F0 = 'able.F0 + inc.F1 +          'nf.F0 +          'is0.F0

F1 = 'able.F1 + inc.F2 + dec.F0 + 'nf.F1 + 'ne.F1 + 'is1.F1

F2 =            inc.F3 + dec.F1 + 'nf.F2 + 'ne.F2 + 'is2.F2

F3 =                     dec.F2          + 'ne.F3

FIFO = ( LF | F0 | RF ) \ {ne, nf, inc, dec}
```

Writer = ... prepare data.'rinF.ainF ...

Reader = ... routF.process data.'aoutF ...

WC counter

WC is a counter which can be set to a certain value, tested for its value and decreased by one. This element is used for modeling the colour in our design. A counter with 1 as its maximal value is specified below.

```
bi WC

WC0


bi WC0
```

```
setWC0.WC0 + setWC1.WC1 + 'wc0.WC0

bi WC1
'wc1.WC1 + minus.WC0
```

Boolean Register

2 phase boolean register is basically the same as the 4 phase one except that it does not have the flatten stage.

Specification:

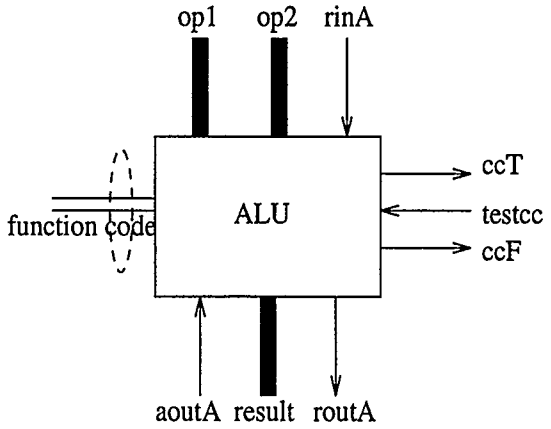
```
bi BOOL_REG
BOOL_REG0

bi BOOL_REG0
set0.'ack0.BOOL_REG0 +
set1.'ack1.BOOL_REG1 +
test.'zero.BOOL_REG0

bi BOOL_REG1
set0.'ack0.BOOL_REG0   +
set1.'ack1.BOOL_REG1   +
test.'one.BOOL_REG1
```

ALU

ALU is an arithmetic and logic unit. It has two inputs and one output. It also includes a conditional code boolean register CC. We use a two-bit local traditional bus as its control wires which is driven by its controller. This ALU does one of three basic functions depending on the current function code (fc).

(1) Passes op1 through, if the function code is 00.

(2) Increases op1 and output the increased value, if the function code is 01.



(3) Compares op1 and op2 and sets CC to true (op1=op2) or to false (op1≠op2), if the function code is 10.

Specification:

```
CC = BOOL_REG[setT/set1,setF/set0,
              ackT/ack1,ackF/ack0,
              ccT/one,ccF/zero,testcc/test]
ALU = rinA.(fc=00.ALU1 + fc=01.ALU1 + fc=10.ALU2)
ALU1 = 'routA.aoutA.ALU
ALU2 = eq.'setT.ackT.ALU1 + neq.'setF.ackF.ALU1
ALU_CC = ( ALU | CC ) \{setT, ackT, setF, ackF}
```

The caller follows the following pattern:

Pass = ... drive op1 and fc with 00.'rinA.routA.process data.'aoutA ...

Increase = ... drive op1 and fc with 01.'rinA.routA.process data.'aoutA ...

Compare = ... drive op1, op2 and fc with 10.'rinA.routA.'aoutA ...

Tester = ... (ccT.code for ccT + ccF.code for ccF) ...

With all these 2-phase basic modules, we are able to specify the PAMM.

6.3. Fetch Unit

The fetch unit in PAMM is responsible for fetching instructions. Once a new data movement is requested, the fetch stage gets the next instruction, pushes it with the current colour into a FIFO. It repeats this procedure (called the main loop) unless the following situations are met:

(1) An interrupt signal arrives when the execute stage executes a JCC instruction with the condition code true. Both the fetch stage and the execute stage will change their current instruction colours, and the fetch stage throws away the address in PC and gets the current instruction address from execute stage. Then the fetch stage will return to the main loop.

(2) A halt signal arrives when a halt instruction is executed. Both the fetch stage and the execute stage will change their current instruction colours. The fetch stage will acknowledge this current data movement to the CPU and halt to the execute stage, and prevent the fetch unit from working until a new data request comes.

The architecture of the pipelined asynchronous fetch unit is given in Figure 6.2. The fetch unit consists of two parts: The top part is responsible for choosing and passing a fetch request from different sources down to the bottom part. The bottom part carries out the actual fetch operation, increases the current instruction address and stores it into the program counter PC, which are in parallel. Detailed explanations will be given as the specification unfolds.

TARB is an arbiter with three contenders: **rLOOP** is an internal request which is released by WAIT when the FIFO is "able" after rLOOP1 is issued. rLOOP1 is issued in two cases: (i) after the move machine has been initialized, (ii) after the current sequential fetch has been finished. **rINTP** is a request from the execute unit when a jump occurs. This means that the current instruction flow is changing. The current instruction address will be from the execute unit not from PC.

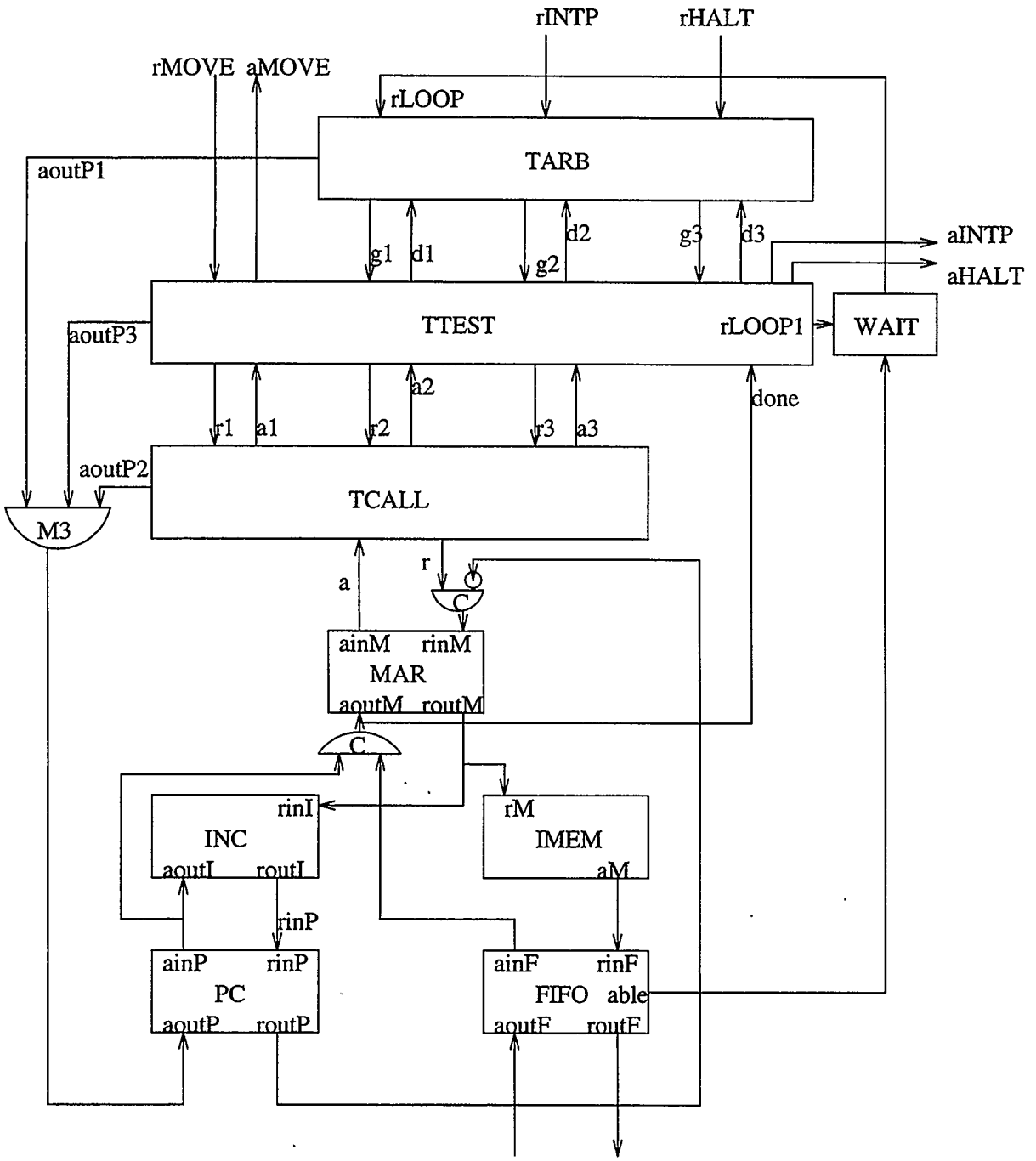


FIGURE 6.2. PAMM Fetch Unit

Here we have to void the address in PC to avoid a deadlock which would otherwise happen when no one takes the old PC out and a new PC value tries to get in. Even though it is obviously necessary to do this, it took quite a few days to locate and fix this deadlock when the CWB showed its existence in our design. The most straightforward solution is to send an acknowledge signal to the PC. The FASTFK is used to route this request to PC. rHALT is a request from the execute unit when a halt instruction is executed. This means the current data movement has been finished. Its CCS specification is:

```
bi TARB
( ARB3 [ rLOOP/r1, rINTP/r2, rHALT/r3,g2'/g2 ] \
| FASTFK [g2'/a,aoutP1/b, g2/c]                \
) \{g2'}
```

TEST is a unit which carries out a number of trivial functions. Four sources use this component. It governs both a colour register and a halt register for recording whether a rHALT has been handled. Its functions include:

When a **rMOVE** arrives, it passes down the request, the initial address from CPU. When both the acknowledgement signal from the CALL unit and the done signal from the bottom part arrive, it raises rLOOP1 for fetching a new sequential intruction. This request is released as rLOOP whenever the FIFO becomes "able". Here the acknowledgement signal from the CALL unit means that the current address is in the MAR (memory address register). The done signal means that the current instruction has been placed in the FIFO and the PC is changed to MAR + 1. The "able" means that there are two spaces left in the FIFO and the sequential access can continue.

```
bi TEST1
rMOVE.'r1.a1.done.'rLOOP1.TEST1
```

When **g1** arrives (sequential PC generation passes through the arbiter), it checks the halt register. If a rHALT has not been handled, it passes on this request, the

address in PC. When both the acknowledgement signal from the CALL unit and the done signal from the bottom part arrive, it signals on d1 for releasing the arbiter and rLOOP1 to the WAIT element. rLOOP will be raised whenever the FIFO becomes "able". Otherwise it clears the halt register for the next use, voids the PC value by acknowledging the PC, releases the arbiter and then sends aMOVE back to the CPU to indicate that the current data movement has been finished.

```
bi TEST2
g1.'testhalt.TEST21

bi TEST21
  halt.'clearhalt.ackclear.'aoutP3.'d1.'aMOVE.TEST2      \
+ nohalt.'r2.a2.done.'d1.'rLOOP1.TEST2
```

When **g2** arrives (the rINTP gains control), it changes the current colour. This is modeled by a counter which records how many instructions carry wrong colour and done by checking how many instructions following the jump and setting the counter. Here we should notice the number of wrong colour instructions is equal to n − 1 (the number of instructions in the FIFO is n) since the jump instruction is still in the FIFO. Then it passes down this request and the new address. When both the acknowledgement signal from the CALL unit and the done signal from the bottom arrive, it sends aINTP to the execute unit and d2 to release the arbiter.

```
bi TEST3
g2.(n1.'setWC0.TEST31 + n2.'setWC1.TEST31)

bi TEST31
'r3.a3.done.'aINTP.'d2.TEST2
```

When **g3** arrives, it sets the halt register, changes the colour register (in the same way as for the jump instruction), sends aHALT to the execute unit and releases the arbiter.

```
bi TEST4
g3.(n1.'setWC0.TEST41 + n2.'setWC1.TEST41)

bi TEST41
'sethalt.ackset.'aHALT.'d3.TEST4
```

Here is the CCS specification for TEST:

```
bi HALT
BOOL_REG[sethalt/set1,clearhalt/set0,          \
         ackset/ack1,ackclear/ack0,            \
         halt/one,nohalt/zero,testhalt/test]

bi WAIT
rLOOP1.able.'rLOOP.WAIT

bi TEST
TEST1 + TEST2 + TEST3 + TEST4

bi TTEST
( HALT | TEST | WAIT ) \ TTESTlines

basi Hlines
sethalt clearhalt ackset ackclear testhalt halt nohalt rLOOP1
```

TCALL is a CALL3 element. It stores the current memory address for three different sources into MAR. When it gets an acknowledgement signal from the MAR, this means the current memory address is safely in the MAR. If this address is from PC, the old address is no longer useful and taken out by sending an acknowledgement to PC. The FASTFK element routes this signal to PC. Its CCS specification is:

```
bi TCALL
( CALL3 [a2'/a2] | FASTFK  [ a2'/a, aoutP2/b, a2/c ] ) \ {a2'}
```

Now we can specify the top part by connecting the above specifications and hiding internal actions.

```
bi TOP
( TARB | TEST | TCALL                          \
| M3[aoutP1/a,aoutP2/b,aoutP3/c,aoutP/z] \
) \ TOPlines

basi TOPlines
d1, d2, d3, g1, g2, g3,                        \
r1, r2, r3, a1, a2, a3,                        \
aoutP1, aoutP2, aoutP3, rLOOP
```

The bottom part is a component by wiring MAR, INC, IMEM, PC, FIFO. Its specification is:

```
bi BOTTOM
( MAR                                          \
| C [ aoutM1/a, ainF/b, aoutM2/z ]             \
| FASTFK [aoutM2/a,done/b,aoutM/c]             \
| FASTFK [ routM/a, rinI/b, rM/c]               \
| INC [ rinP/routI]                             \
| PC                                           \
| IMEM [ rinF/aM]                              \
| FIFO                                         \
| FASTFK [ ainP/a, aoutM1/b, ainI/c ]          \
| C'[routP/a, r/b, rinM/z]                      \
) \ BOTTOMlines
```

```
basi BOTTOMlines
rinM aoutM routM aoutM1 aoutM2 \
rM                             \
rinF ainF                      \
rinP ainP routP                \
rinI ainI
```

Finally the fetch unit can be specified by composing the top component and the bottom component, and hiding internal communications.

```
basi FETCHlines
r a able done aoutP n1 n2

bi FETCH
( TOP |  BOTTOM ) \ FETCHlines
```

Property Check: Observable actions at the FETCH interface:

```
Command: sort FETCH
rMOVE 'aMOVE 'routF aoutF rINTP 'aINTP rHALT 'aHALT 'setWC0 'setWC1
```

The complexity of FETCH unit::

```
Command: min TOP
Save result in identifier: TOP'
TOP' has 2362 states.

Command: min BOTTOM
Save result in identifier: BOTTOM'
BOTTOM' has 292 states.

Command: min FETCH
```

```
Save result in identifier: FETCH'
FETCH' has 1217 states.
```

6.4. Execute Unit

The architecture of the execute unit is shown in Figure 6.3. The execute stage always gets an instruction from the FIFO as long as the FIFO is not empty. If the instruction doesn't have the same colour as the current colour of the execute stage, it means that this instruction is obsolete (it is prefetched following a JCC which changes the current instruction flow or HALT) and should be thrown away. Otherwise the execute stage will execute the current instruction and then repeats this procedure. There are two instructions which could change the colour register. One is JCC. If the conditional code is true, the execute stage will change its colour and then send an interrupt request to the fetch stage. It can return to the normal procedure after the interrupt acknowledge signal arrives. The other one is HALT. The execute stage will change its colour and then send a halt request to the fetch stage. It can return to the normal procedure after the halt acknowledge signal arrives.

Here we model this colouring by a counter WC which records the number of the wrong colour instructions in the FIFO. The execute unit decides whether or not the current instruction should be executed by checking the WC. If WC indicates there is at least one wrong colour instruction, it just discards this one and decreases WC by one. Otherwise it executes this instruction. Since this execute unit is quite trivial to implement, we keep it abstract for reducing PAMM complexity. This unit is specified below:

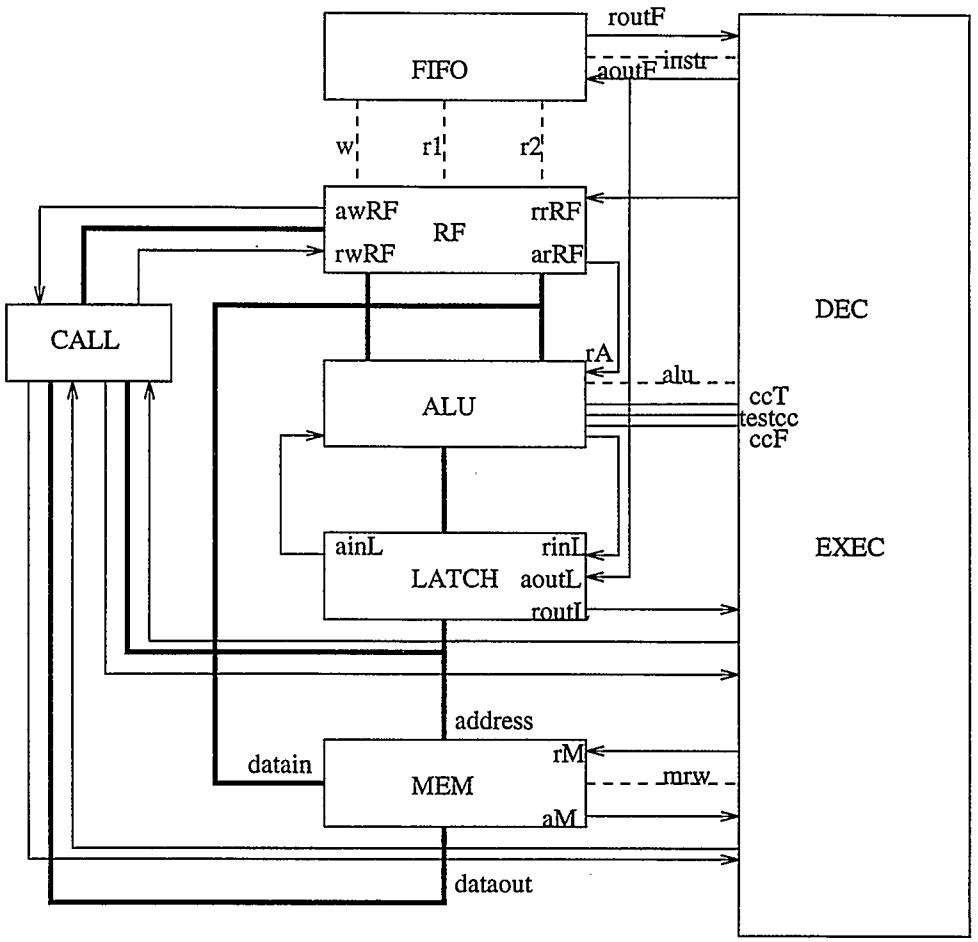


FIGURE 6.3. PAMM EXECUTE UNIT

```
bi XC

XC0


bi XC0                                          ** CC is false

routF.XC01


bi XC01
```

```
 wc0.take.                                  \ ** right colour
        ( jcc.'aoutF.XC0                    \ ** no jump occurs
     +    hlt.'rHALT.aHALT.'aoutF.XC0       \ ** halt
     +    lod.'aoutF.XC0                    \
     +    scc.XC2                           \
     +    nop.'aoutF.XC0                    \
     +    mov.'aoutF.XC0                    \
     +    inc.'aoutF.XC0                    \
     +    sto.'aoutF.XC0 )                  \
+ wc1.discard.'minus.'aoutF.XC0               ** wrong colour instruction

bi XC2
eq.'aoutF.XC1 + neq.'aoutF.XC0               ** CC = (r1==r2)

bi XC1                                       ** CC is true
routF.XC11

bi XC11
 wc0.take.                                  \** right colour
     (jcc.'rINTP.aINTP.'aoutF.XC1           \** jump occurs
    + hlt.'rHALT.aHALT.'aoutF.XC1           \
    + lod.'aoutF.XC1                        \
    + scc.XC2                               \
    + nop.'aoutF.XC1                        \
    + mov.'aoutF.XC1                        \
    + inc.'aoutF.XC1                        \
    + sto.'aoutF.XC1 )                      \
+ wc1.discard.'minus.'aoutF.XC1              ** wrong colour instruction
```

Property Check:

```
Command: sort XC
routF 'aoutF.'rINTP aINTP 'rHALT aHALT wc0 wc1
take discard eq neq lod sto mov inc scc jcc nop hlt
```

The complexity of the execute unit:

```
Command: min XC
Save result in identifier: XC
XC' has 22 states.
```

6.5. PAMM

PAMM is the composition of the fetch unit and the execute unit, as specified below:

```
bi PAMM
( FETCH | XC | WC) \ PAMMlines

basi PAMMlines
rINTP aINTP rHALT aHALT      \
routF aoutF setWC0 setWC1 wc0 wc1 minus
```

Observable actions in PAMM:

```
Command: sort PAMM
eq,hlt,inc,jcc,lod,mov,neq,nop,scc,sto,rMOVE,'aMOVE
```

PAMM complexity:

```
Command: min PAMM
Save result in identifier: PAMM'
PAMM' has 104 states.
```

6.6. Environment

PAMM is a slave processor for the CPU. CPU has to obey some rules in order to use it, as specified below:

```
bi ENV
initialize.'rMOVE.aMOVE.ENV
```

6.7. Property Check

Now we can look at the PAMM when it operates in its environment. PAMM is proved to be deadlock free and possess required liveness.

```
bi PAMM_ENV
(PAMM | ENV)\{rMOVE, aMOVE}
Command: min PAMM_ENV
Save result in identifier: PAMM_ENV'
PAMM_ENV' has 20 states.

Command: fd PAMM_ENV'
No such agents
Command: cp AMM'
Proposition: BOX POSS <lod>T
true
```

Similar tests for other instructions have been successfully carried out.

6.8. Summary

This chapter gives a PAMM specification which has been shown deadlock free. PAMM is a highly parallel machine which takes advantage of asynchronous design style. But very tricky reasoning is necessary to avoid deadlock. During our design, we met three major deadlocks and fixed them after they were tested by the CWB.

(1) Requesting the sequential fetch:

   (a) The first natural thought is that we request a sequential fetch once the current instruction has been fetched. When we did this, a deadlock happened since we sent too many requests which could not be consumed

in the following case: a rLOOP and rINTP arrive at the same time and the aribter allows the rINTP to go through. When this request rINTP has been processed, a new rLOOP is issued before the previous one can be processed.

This deadlock is fixed by finding rLOOP can be issued only after the initialization or a sequential fetch has been carried out.

(b) The second natural thought is that rLOOP is issued whenever FIFO is not full. A deadlock happened when the FIFO is full and no rLOOP can be issued and in the meantime a rINTP comes. The new instruction cannot be put into the FIFO and the execute unit is not taking any instruction out of FIFO since the jump instruction is not yet completed. This deadlock can be removed by keeping a space in the FIFO for dealing with this case.

(2) PC value: When a jump is taken, the old PC value should be discarded. In asynchronous design, we have to explicitly remove the value and free the PC for its next use. A deadlock happened until we noticed this.

CHAPTER 7
Conclusions

7.1. Summary

The contribution of this thesis has been to present two case studies which:

(1) help bridge the gap between the formal method and engineering approaches by focussing on block level descriptions which map exactly into CCS and yet serve as blueprints for implementations (see also [BLS+94a, BLP94a, BLP94b, BLGP94, BL94]);

(2) help clarify a hierarchical methodology for systematically developing and testing asynchronous systems (see also [Ste94, Liu92]);

(3) are valuable in their own right.

They are amongst largest verifications yet done of asynchronous systems.

In chapter 1, we explained the terminologies used in asynchronous design. In chapter 2, we surveyed three disparate approaches of asynchronous design: Silicon Compilation, Formal Methods, and the Engineering Approach. We also explained these approaches giving one typical example for each approach. In chapter 3, we covered the specification language CCS, its companion logic modal μ-calculus and the mechanized workbench CWB which are used throughout the rest of this thesis to specify and verify our abstract designs. The next two chapters closely followed previously published work [BLS+94b, BLS+94a]. In chapter 4, we described the move

machine upon which this thesis research is based. In chapter 5, we systematically followed AMM through three different abstract levels of specification, showed these levels to be consistent, and showed that they were deadlock free, livelock free and in possession of certain safety and liveness properties. The heart of the thesis was chapter 6, in which we specified and verified PAMM, a pipelined 2-phase machine, whose implementation is inherently parallel in operation. As case studies, the models of chapter 5 and chapter 6 explicate the 4 phase and 2 phase design styles. They are also vehicles for expanding the specification driven design methodology which systematically takes one down from the top level to a provenly equivalent description which serves as an implementation blueprint, thus helping to bridge the gap between the Formal and the Engineering approaches.

Through these case studies, CCS has been demonstrated to be an appropriate and usable tool for describing and developing asynchronous hardware. At each abstract level, an asynchronous circuit has a finite number of distinguished states. This finiteness allows us to use the CWB workbench. The companion logic to CCS (the modal μ-calculus) copes with the complicated properties, such as deadlock, livelock, liveness and safety, etc., inherent in asynchronous systems. The CWB also allows us to test the equivalence of descriptions at different levels of abstraction. We believe that observational congruence best defines the equivalence between two circuits. Real circuits are always stable, which means that the first action is not an internal action (this matches event-driven circuits). So the equivalence check is reduced to a check for weak bisimularity, which is easy to carry out.

These case studies have exhibited some shortcomings of the methodology.

(1) State explosion: A very succinct CCS specification can be a very complicated model which has millions of states.

(2) Tricky reasoning to locate and fix deadlock: Asynchronous systems are very prone to deadlock. Although the CWB can test whether or not a design includes deadlock, locating and fixing a deadlock requires very tricky reasoning.

(3) Slow algorithms in the CWB: the algorithms for agent minimization and equivalence checking are very slow. Even in the SUN/SPARC 20, it takes 3 to 4 hours to minimize an agent which has more than 1000 states after minimization.

7.2. Future Work

7.2.1. Silicon Compiler. CCS is a specification language with succinct syntax and clean semantics. However, since a CCS specification can be very abstract, it might take an experienced engineer a long time to generate a real design from this specification. It is not a mechanical step. Building a silicon compiler based upon CCS would be a great help but also a difficult topic. As noted by Stevens[Ste94], some restrictions should be put on the CCS specification to be translated by its silicon compiler.

7.2.2. Faster Algorithms. The algorithms in the CWB for agent minimization and equivalence check are very slow. Improving their efficiencies will greatly shorten the development cycle of asynchronous design. Analyzing the complexity for this problem and finding an approximate optimal algorithm will be worthwhile.

Bibliography

[BE92] Janusz A. Brzozowski and Jo C. Ebergen. On the Delay-Sensitivity of Gate Networks. *IEEE Transactions on Computers*, 41(11):1349–1360, November 1992.

[BL94] G. Birtwistle and Y. Liu. Manchester Amulet Specification: Top Level. Computer Science Technical Report, Computer Science Department, University of Calgary, 1994.

[BLGP94] G. Birtwistle, Y. Liu, J. Garside, and N. Paver. Manchester Amulet Specification: The Execution Pipeline. Computer Science Technical Report, Computer Science Department, University of Calgary, 1994.

[BLP94a] G. Birtwistle, Y. Liu, and N. Paver. Manchester Amulet Specification: The Address Interface. Computer Science Technical Report, Computer Science Department, University of Calgary, 1994.

[BLP94b] G. Birtwistle, Y. Liu, and N. Paver. Manchester Amulet Specification: The Register Bank. Computer Science Technical Report, Computer Science Department, University of Calgary, 1994.

[BLS+94a] Graham Birtwistle, Y. Liu, D. Spooner, J. Aldwinckle, K. Stevens, and W.Yu. Case Studies in Asynchronous Design. Part II: a 4-stroke AMM. Computer Science Technical Report, Computer Science Department, University of Calgary, 1994.

[BLS+94b] Graham Birtwistle, Y. Liu, D. Spooner, J. Aldwinckle, K. Stevens, and W.Yu. Case Studies in Asynchronous Design. Part I: AMM Architecture. Computer Science Technical Report, Computer Science Department, University of Calgary, 1994.

[BM88] Steven M. Burns and Alain J. Martin. Syntax-directed Translation of Concurrent Programs into Self-timed Circuits. In J. Allen and F. Leighton, editors, *Proceedings of the Fifth MIT Conference on Advanced Research in VLSI*, pages 35–50. MIT Press, 1988.

[Bru91] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits.* PhD thesis, Carnegie Mellon University, 1991.

[BS87] J. A. Brzozowski and C.-J. Seger. A Unified Theory of Asynchronous Networks. Report CS-87-24, Computer Science Dept., Univ. of Waterloo, Cananda, March 1987.

[Dav95] A. Davis. Burst Mode Controllers: Synthesis and Experience. In G. Birtwistle and A. Davis, editors, *Proceedings VII Banff Workshop: Asynchronous Digital Circuit Design.* Springer Verlag, Workshops in Computing Series, 1995.

[DN95] A. Davis and S. Nowick. Introduction. In G. Birtwistle and A. Davis, editors, *Proceedings VII Banff Workshop: Asynchronous Digital Circuit Design.* Springer Verlag, Workshops in Computing Series, 1995.

[DNS92] David L. Dill, Steven M. Nowick, and Robert F. Sproull. Specification and automatic verification of self-timed queues. *Formal Methods in System Design*, 1(1):29–60, July 1992.

[Ebe89] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tract.* Centre for Mathematics and Computer Science, 1989.

[Ebe91a] Jo C. Ebergen. A Formal Approach to Designing Delay-Insensitive Circuits. *Distributed Computing*, 5(3):107–119, 1991.

[Ebe91b] Jo C. Ebergen. Parallel Computations and Delay-Insensitive Circuits. In Graham Birtwistle, editor, *IV Higher Order Workshop, Banff 1990*, pages 85–104. Springer-Verlag, 1991.

[Fur93] S. B. Furber. Computing without clocks: Micropipelining the arm processor. In Graham Birtwistle, editor, *Proceedings VII Banff Workshop: Asynchronous Digital Circuit Design*, 1993.

[Fur95] S. Furber. Computing without Clocks: Micropipelining the ARM Processor. In G. Birtwistle and A. Davis, editors, *Proceedings VII Banff Workshop: Asynchronous Digital Circuit Design.* Springer Verlag, Workshops in Computing Series, 1995.

[JC90] J.Bradfield and C.Stirling. Verifying temporal properties of processes. In J.Sifakis, editor, *Proceedings of CONCUR '90*, number 458, pages 115–125. Springer-Verlag, 1990.

[Kal86] Anne Kaldewaij. *A Formalism for Concurrent Processes.* PhD thesis, Dept. of Math. ans C.S., Eindhoven Univ. of Technology, 1986.

[Liu92] Ying Liu. *Reasoning about asynchronous designs.* MScEE thesis, Electrical and Computer Engineering, The University of Calgary, 1992.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mol91] Faron Moller. The Edinburgh Concurrency Workbench. Technical Report, Computer Science Department, University of Edinburgh, 1991.

[PDF+92] N. C. Paver, P. Day, S. B. Furber, J. D. Garside, and J. V. Woods. Register locking in an asynchronous microprocessor. In *Proc. Int'l. Conf. Computer Design*, pages 351–355. IEEE Computer Society Press, October 1992..

[Sch85] Huub M. J. L. Schols. A formalisation of the foam rubber wrapper principle. Master's thesis, Dept. of Math. ans C.S., Eindhoven Univ. of Technology, 1985.

[Ste94] Ken Stevens. *Burst Mode Asynchronous Design.* PhD thesis, Computer Science, The University of Calgary, 1994.

[Sti91] Colin Stirling. An introduction to modal and temporal logics for ccs. In A.Yonezawa and T.Ito, editors, *Concurrency: Theory, Language, and Architecture*, number 491, pages 2–20. Springer-Verlag, 1991.

[Sti92a] Colin Stirling. Modal and Temporal Logics. Technical Report ECS-LFCS-91-157, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1992.

[Sti92b] Colin Stirling. Modal and Temporal Logics for Processes. Technical Report ECS-LFCS-91-221, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1992.

[Sut89] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, January 1989.

[Udd86] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986.

[UV88] Jan Tijmen Udding and Tom Verhoeff. The mathematics of directed specifications. Technical Report WUCS-88-20, Dept. of C.S., Washington Univ., St. Louis, MO, June 1988.

[vB92a] Kees van Berkel. Beware the Isochronic Fork. *Integration, the VLSI journal*, 13(2):103–128, June 1992.

[vB92b] Kees van Berkel. *Handshake Circuits: An Intermediary between Communicating Processes and VLSI.* PhD thesis, Eindhoven University of Technology, 1992.

[vBKR+91] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-Programming Language Tangram and its Translation into Handshake Circuits. In *Proc. European Design Automation Conf.*, pages 384–389, 1991.

[vBNRS88] C. H. (Kees) van Berkel, Cees Niessen, Martin Rem, and Ronald W. J. J. Saeijs. VLSI Programming and Silicon Compilation. In *Proc. Int'l. Conf. Computer Design*, pages 150–166, Rye Brook, New York, 1988. IEEE Computer Society Press.

[vdK92] Michiel van der Korst. VOICE, a silicon compiler for asynchronous circuits. Technical report, IVO, Eindhoven University of Technology, August 1992.

APPENDIX A
4 Phase Basic Data Path Modules

In the following, we list all 4 phase basic data modules used in our design. All these descriptions are from [BLS+94b] and [BLS+94a] with permission from Dr. G. Birtwistle.

A.1. Wire

Wires may be either high or low; and may be sensed. The description is:

$$WIRE \stackrel{def}{=} WIRE0$$

$$WIRE0 \stackrel{def}{=} in.WIRE1 + \overline{lo}.WIRE0$$
$$WIRE1 \stackrel{def}{=} in.WIRE0 + \overline{hi}.WIRE1$$

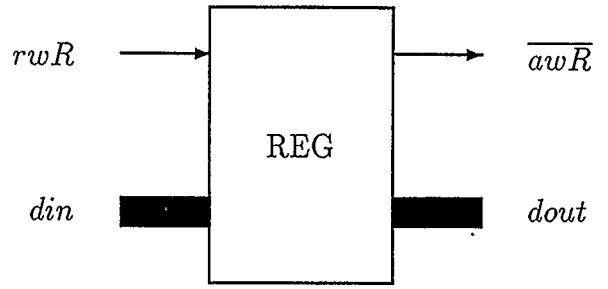
The controller of the wire makes it high with the first *in* and low with the second *in*.

Sensor code typically follows the pattern:

$$...(hi.\text{code for high} + lo.\text{code for low})$$

A.2. Register

- the output is always "strong" and may be read several times
- the input may "wobble". It is ignored until there is a write request at which time the output changes at once.



Protocol. The interaction sequence is:

Writer: sDin $\overline{rwR}$ awR zDin $\overline{rwR}$ awR ...

$\Downarrow$ $\Uparrow$ $\Downarrow$ $\Uparrow$

REG: rwR reg' $\overline{awR}$ rwR $\overline{awR}$ REG

Specification. From the register's point of view, after rwR is raised, the (strong) value on *din* is written into the register (indicated by reg'). The fresh value also drives the output bus. When the register raises awR, the caller is safe to assume that the new value is in the register and is strong. The caller then lowers rwR and the register lowers awR.

$$REG \stackrel{def}{=} rwR.\mathsf{reg}'.\overline{awR}.rwR.\overline{awR}.REG$$

Usage. The caller puts new value on the input bus (sDin) then raises rwR. When the register raises awR, the caller will tristate the input bus (zDin) before lowering rwR and then waits for awR to be lowered. The onus is on the caller to make the input strong before raising rwR.

$$Writer \stackrel{def}{=} \mathsf{sDin}.\overline{rwR}.awR.\mathsf{zDin}.\overline{rwR}.awR....$$

A.3. Enable

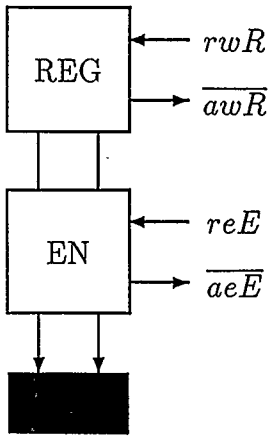
- the input to the ENABLE is normally strong,
- its output is normally tristated, it is driven only on request

Protocol. The interaction sequence is:

Caller: $\overline{reE}$ aeE read $\overline{reE}$ aeE ...

$\Downarrow$ $\Uparrow$ $\Downarrow$ $\Uparrow$

EN: reE sEN $\overline{aeE}$ reE zEN $\overline{aeE}$ EN



Enable registers are placed between busses and "ordinary" registers whose outputs are normally strong so that the latter only drive the bus on request. When reE is raised, the strong data value of the register is passed, indicated by sEN. aeE is then raised, a signal to the caller that the bus is safe to read. When the bus has been read, the caller lowers reE. The enable register then cuts off the register value thus tristating the bus (zEN) and then lowers aeE.

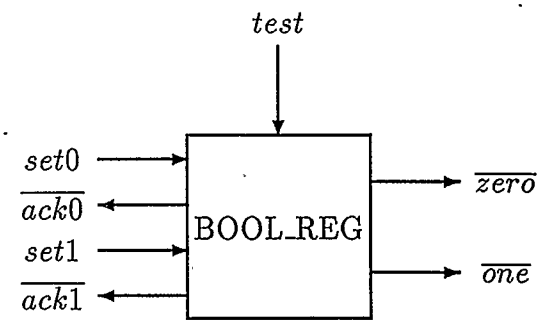
Specification.

$$EN \stackrel{def}{=} reE.\mathsf{sEN}.\overline{aeE}.reE.\mathsf{zEN}.\overline{aeE}.EN$$

Usage.

$$\text{Caller} \stackrel{def}{=} \overline{reE}.aeE.\mathsf{read}.\overline{reE}.aeE....$$

A.4. Boolean register



- we assume that read, write, and test requests do not overlap
- write requests set the register true or false. (There is no wobbling — sets are always accepted.)
- the current value may be tested and read several times.

Protocol. The interaction sequences are:

Set/reset register.

Setter: $\overline{set0}$ $ack0$ $\overline{set0}$ $ack0$ ...

$\Downarrow$ $\Uparrow$ $\Downarrow$ $\Uparrow$

B_REG: $set0$ reg=F $\overline{ack0}$ $set0$ $\overline{ack0}$ B_REG

Read value in register. Suppose the boolean register is in state 1.

Read: $\overline{test}$ one $\overline{test}$ one ...

$\Downarrow$ $\Uparrow$ $\Downarrow$ $\Uparrow$

B_REG1: $test$ $\overline{one}$ $test$ $\overline{one}$ B_REG1

Specification.

$$BOOL\_REG \stackrel{def}{=} B\_REG0$$

$$\begin{aligned} B\_REG0 \stackrel{def}{=}\ & set0.\overline{ack0}.set0.\overline{ack0}.B\_REG0 \\ +\ & set1.\overline{ack1}.set1.\overline{ack1}.B\_REG1 \\ +\ & test.\overline{zero}.test.\overline{zero}.B\_REG0 \end{aligned}$$

$$\begin{aligned} B\_REG1 \stackrel{def}{=}\ & set0.\overline{ack0}.set0.\overline{ack0}.B\_REG0 \\ +\ & set1.\overline{ack1}.set1.\overline{ack1}.B\_REG1 \\ +\ & test.\overline{one}.test.\overline{one}.B\_REG1 \end{aligned}$$

A boolean register may be set to 0 or 1. Each setting is acknowledged. To read a register, a caller raises *test*. The register will respond by raising either *zero* or *one*.

Usage. The calling tactic is:

$$\overline{test}.(zero.\overline{test}.zero.\text{code for 0} + one.\overline{test}.one.\text{code for 1})$$

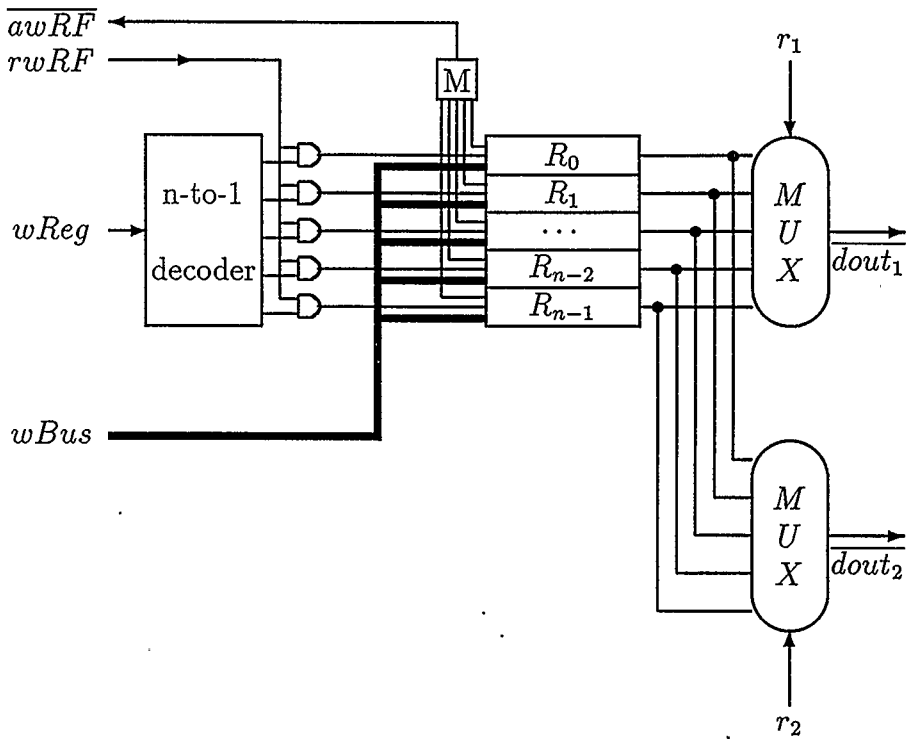
A.5. Register file

A register file is a block of (in our case 8) registers each following the model of §2.1.2. The outputs from the registers (which are always strong) are filtered through two multiplexers — one selected by the control bits $r1$, the other by the control bits $r2$. Thus

$dout1 \stackrel{def}{=} RF[r1]$

$dout2 \stackrel{def}{=} RF[r2]$

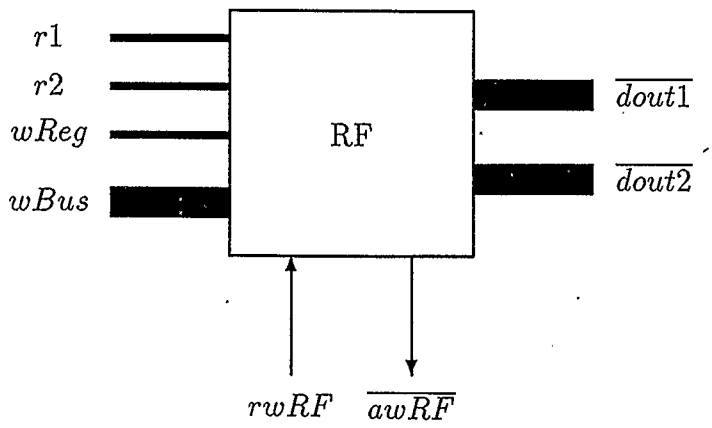
The input bus is copied to all 8 registers. The write signal from $rwRF$ is and'ed with the decoded control bits $wReg$ so that only one register will be "invited" to write



when we go through the sequence $rwRF/awRF$ up and down. The 8 register awR signals are or'ed together to produce the external $awRF$ signal.

Here is our abstract view of the register file:

Protocol. The interaction sequence is:



Specification. At our level of abstraction, its the same as a single register.

$$RF \stackrel{def}{=} rwRF.\mathrm{rf}'.\overline{awRF}.rwRF.\overline{awRF}.RF$$

NB we have abstracted away $r1$, $r2$, $wReg$, $wBus$, $dout1$ and $dout2$ from our specification.

Usage. See a single register.

As far as AMM is concerned, the

$r1$ is wired to bits 3..5 of IR

$r2$ is wired to bits 6..8 of IR

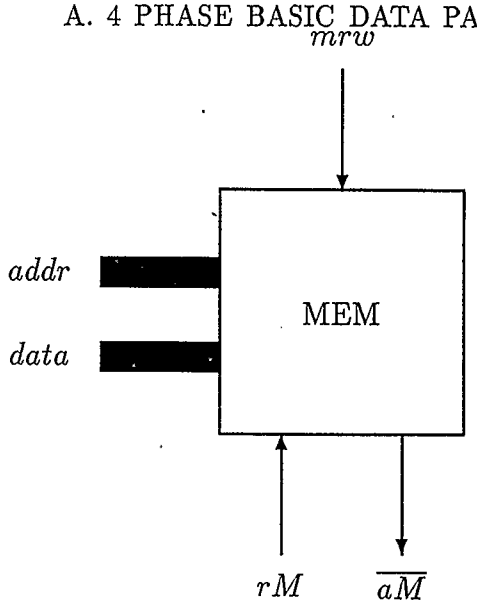
$wReg$ is wired to bits 6..8 of IR

When IR changes, so do they, and so do the values on $dout1$ and $dout2$.

A.6. Memory

The address bus is unidirectional; the data bus is bidirectional. The caller raises the read/write line mrw for a read and leaves it down for a write. The read/write line is lowered after the read has taken place.

Protocol. The two interaction sequences are:



	mrw↑				mrw↓			
Read:	sA	$\overline{rM}$	aM	read; zA	$\overline{rM}$	aM	...	
		⇓	⇑		⇓	⇑		
MEM:		rM	sD $\overline{aM}$		rM	zD $\overline{aM}$	MEM	

For a read, the caller raises the line mrw and drives the address bus (sA). It then raises rM and awaits the raising of aM at which time the memory will have put the read value strongly onto the data bus. When the data has been read, the caller lowers mrw and tristates the address bus (zA) before lowering rM. The memory unit will tristate the data bus (zD) and then lower aM.

	sA			zA			
Write:	sD	$\overline{rM}$	aM	zD	$\overline{rM}$	aM	...
		⇓	⇑		⇓	⇑	
MEM:		rM	mem' $\overline{aM}$		rM	$\overline{aM}$	MEM

For a write, the caller leaves the line mrw low, but drives the address bus (sA) and the data bus (sD). It then raises rM and awaits the raising of aM at which time the memory will have been updated (mem'). The caller then tristates both busses (zA.zD) and then lowers rM. The memory unit then lowers aM.

Specification.

$$MEM \stackrel{def}{=} rM.(mrwT.READ + mrwF.WRITE)$$

$$READ \stackrel{def}{=} \mathsf{sD}.\overline{aM}.rM.\mathsf{zD}.\overline{aM}.MEM$$

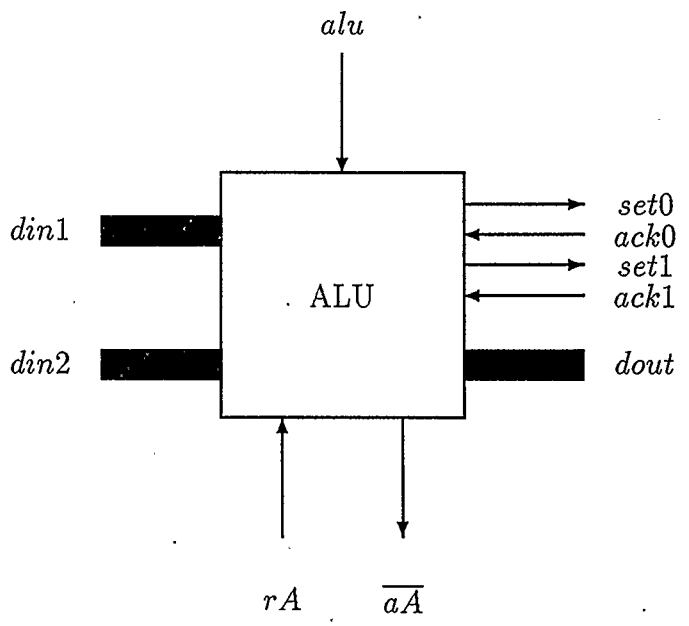
$$WRITE \stackrel{def}{=} \mathsf{mem'}.\overline{aM}.rM.\overline{aM}.MEM$$

Usage. The caller follows one or other of the protocols:

$$Read \stackrel{def}{=} \overline{mrw}.\mathsf{sA}.\overline{rM}.aM.\mathsf{read\ data}.\overline{mrw}.\mathsf{zA}.\overline{rM}.aM$$

$$Write \stackrel{def}{=} \mathsf{sA.sD}.\overline{rM}.aM.\mathsf{zA.zD}.\overline{rM}.aM$$

A.7. ALU



The ALU can either compare (in which case it sets a boolean register) or increment. The alu line is raised then lowered for compare and left low throughout an increment.

Protocol. The two interaction sequences are:

Compare

	alu↑					alu↓			
	sDin1					zDin1			
COMP:	sDin2	$\overline{rA}$			aA	zDin2	$\overline{rA}$	aA	...
		⇓			⇑		⇓	⇑	
ALU:		rA	din1=din2⇒SET1		$\overline{aA}$		rA	$\overline{aA}$	ALU
			din1≠din2⇒SET0		$\overline{aA}$		rA	$\overline{aA}$	ALU

where SET1 = $\overline{set1}.ack1.\overline{set1}.ack1$

SET0 = $\overline{set0}.ack0.\overline{set0}.ack0$

The caller sets the compare operation by raising alu, drives the data on the input busses, and then raises rA. The circuit fires by setting a condition code register to F (via $set0/ack0$) or T (via $set1/ack1$). It then raises aA. When the caller is ready, it will lower flatten the line *alu* and the busses and then lower rA. The ALU responds by lowering aA.

Increment

INC:	sDin1	$\overline{rA}$		aA	read	zDin1	$\overline{rA}$		aA	...
		⇓		⇑			⇓		⇑	
ALU:		rA	sA	$\overline{aA}$			rA	zA1	$\overline{aA}$	ALU

The caller drives $din1$ and then raises then raises rA. The circuit fires, putting the result on the output bus *dout.* and then raising aA. When the caller has read the value on *dout* it lowers rA. The ALU then tristates the *dout* bus and lowers aA.

Specification.

$$ALU \stackrel{def}{=} rA.(aluT.COMP + aluF.INC)$$

$$COMP \stackrel{def}{=} \overline{set1}.ack1.\overline{set1}.ack1.\overline{aA}.rA.\overline{aA}.ALU + \overline{set0}.ack0.\overline{set0}.ack0.\overline{aA}.rA.\overline{aA}.ALU$$

$$INC \stackrel{def}{=} \mathsf{sA}.\overline{aA}.rA.\mathsf{zA}.\overline{aA}.ALU$$

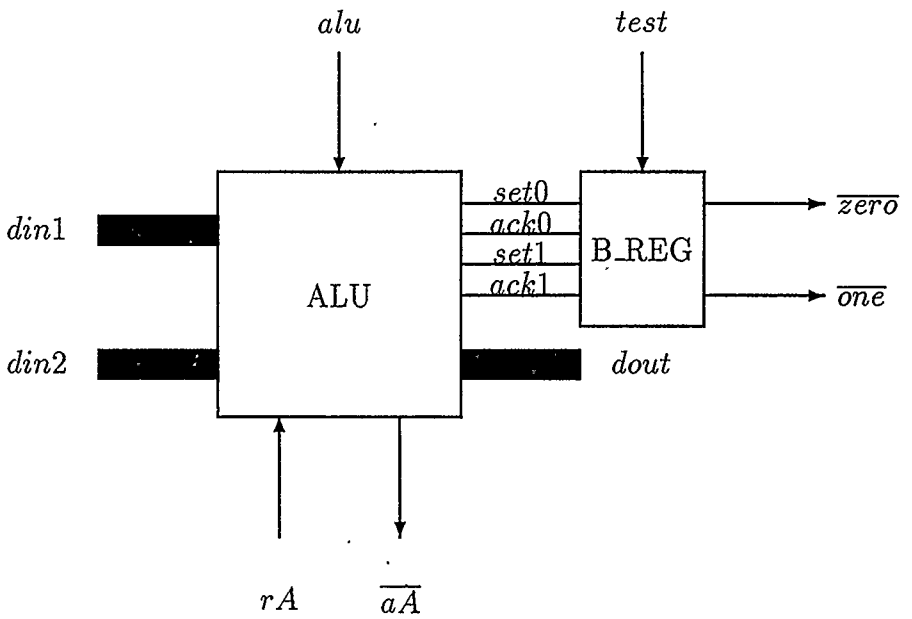
Usage. The caller follows the protocols:

$$COMP \stackrel{def}{=} alu.\mathsf{sDIN1}.\mathsf{sDIN2}.\overline{rA}.aA.$$
$$alu.\mathsf{zDIN1}.\mathsf{zDIN2}.\overline{rA}.aA$$

$$INC \stackrel{def}{=} \mathsf{sDIN1}.\overline{rA}.aA.$$
$$\mathsf{zDIN1}.\overline{rA}.aA$$

ALU'

Since our ALU will always be associated with a "condition code" register, we might as well give it define it. Here is the implementation:

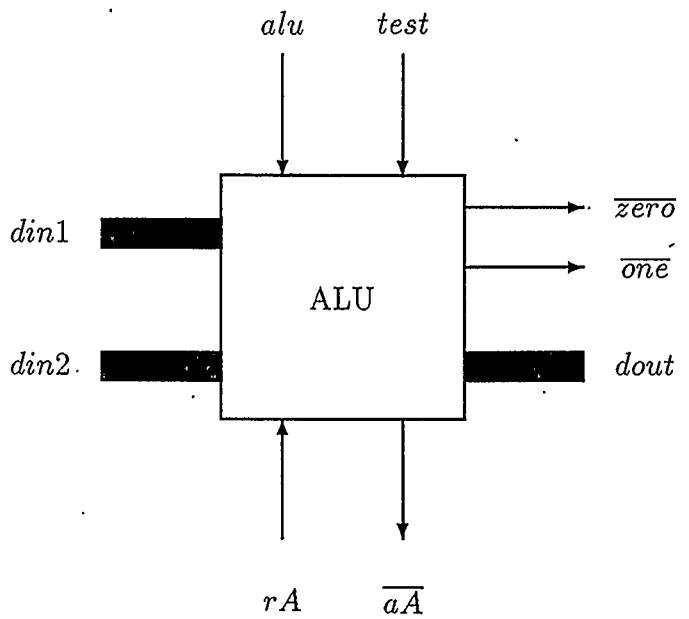


with CCS definition:

$$ALU' \stackrel{def}{=} (ALU \mid BOOL\_REG) \setminus \{set0, ack0, set1, ack1\}$$

We picture this composition by:

and its specification is (roughly):



$$ALU' \stackrel{def}{=} rA.(COMP + INC) + test.TEST$$

$$COMP \stackrel{def}{=} \mathsf{cc}'.\overline{aA}.rA.\overline{aA}.ALU'$$

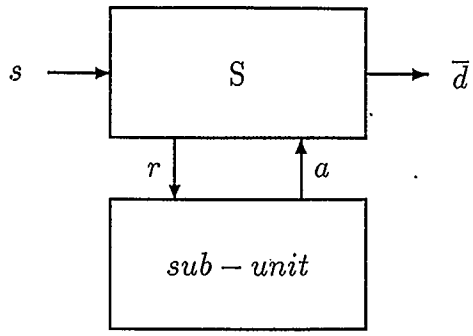
$$INC \stackrel{def}{=} \mathsf{sA}.\overline{aA}.rA.\mathsf{zA}.\overline{aA}.ALU'$$

$$TEST \stackrel{def}{=} \mathsf{ccT}.\overline{one}.test.\overline{one}.ALU' + \mathsf{ccF}.\overline{zero}.test.\overline{zero}.ALU'$$

A.8. van Berkel's S element

Protocol. The interaction sequence for van Berkel's S circuit ([vB92a]) is:

Caller:	$\overline{s}$						d	...	$\overline{s}$	d	Caller
	$\Downarrow$						$\Uparrow$		$\Downarrow$	$\Uparrow$	
S:	s	$\overline{r}$		a	$\overline{r}$	a	$\overline{d}$		s	$\overline{d}$	S
		$\Downarrow$		$\Uparrow$	$\Downarrow$	$\Uparrow$					
sub-unit:	r	compute	$\overline{a}$	r	flatten	$\overline{a}$					sub-unit



$$S \stackrel{def}{=} s.\overline{r}.a.\overline{r}.a.\overline{d}.s.\overline{d}.S$$

Specification. When s is raised, it calls an associated sub-unit twice (once to compute, once to flatten). Raising r fires the subunit. Once the sub-unit has computed it raises a, whereupon the S element lowers r. The sub-unit now flattens its local circuits and then lowers r. The S element now signals it is done by raising d. Then s and d are lowered in turn.

$$S \stackrel{def}{=} s.r.\text{let subunit compute}.a.\overline{r}.\text{let sub-unit flatten}.a.\overline{d}.s.\overline{d}.S$$

Usage.

$$Caller \stackrel{def}{=} \overline{s}.d.\overline{s}.d....$$