Design of a Multibus Data-Flow Processor Architecture

by

Sarosh Malayattil

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Mark T. Jones, Chair

Thomas L. Martin

Paul E. Plassmann

February 17, 2012

Blacksburg, Virginia

Keywords: Electronic Textiles, Multibus, Automation Framework

Design of a Multibus Data-Flow Processor Architecture

Sarosh Malayattil

(ABSTRACT)

General purpose microcontrollers have been used as computational elements in various spheres of technology. Because of the distinct requirements of specific application areas, however, general purpose microcontrollers are not always the best solution. There is a need for specialized processor architectures for specific application areas. This thesis discusses the design of such a specialized processor architecture targeted towards event driven sensor applications. This thesis presents an augmented multibus dataflow processor architecture and an automation framework suitable for executing a range of event driven applications in an energy efficient manner. The energy efficiency of the multibus processor architecture is demonstrated by comparing the energy usage of the architecture with that of a PIC12F675 microcontroller.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0834490.

Acknowledgments

I am indebted to many people for the completion of this work.

I would like to thank my advisor Dr. Mark Jones for his guidance throughout the project work. I would like to especially thank him for proof reading my thesis patiently and coming up with valuable suggestions.

I would like to thank Dr. Thomas Martin for his support and encouragement throughout the research.

I would like to extend my gratitude to Dr. Paul Plassmann for serving in my thesis defense committee.

The work demonstrated in this thesis would not have been possible without the cooperation and contribution from Mr. Anup Mandlekar, my project partner. My special thanks goes out to him.

Finally, I would like to thank my parents Mr UP Aravindakshan and Mrs Sarala Aravind for their unflinching support through all my endeavors.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Electronic textiles (e-textiles) are fabrics that have electronics and interconnections woven into them in an unobtrusive manner. The field of e-textiles is becoming an increasingly important part of wearable computing, helping to make pervasive devices truly wearable [1]. E-textiles are used in application areas in which sensing and computing are closely related to the physical shape and movement of the textile [2]. These application areas include health monitoring, location awareness, and motion analysis. Moreover, e-textiles have strict limitations in power and size [3].

General purpose processors such as PIC® [4] microcontrollers have been traditionally used as computation elements in e-textiles. E-textiles applications are inherently event driven in nature, acting only in response to received signals [5]. In the event driven domain, a node will remain idle unless it is instigated by an event [6]. The low power requirements of an e-textile [3] necessitate that nodes not be unnecessarily active. General purpose processors have a control flow architecture [7]. Mapping an event driven application on a general purpose processor is inefficient in terms of energy and performance. E-textiles have strict energy constraints, hence, general purpose microcontrollers are not ideal to be used in e-textile ap-

plications. An event-based, power-aware, and configurable dataflow processor architecture with no instructions has been developed to address this problem [8]. The architecture inherently handles multiple events by design. In addition, an automation frame work has been developed, which aims to speed up programming of applications, aiding fast prototyping and validation of the target architecture [9]. However, the preliminary version of this architecture is limited in configurability and the range of applications it can be used in. Thus, there is a need for an improved architecture and automation framework.

This thesis presents the design of such a multibus dataflow processor architecture with an automation framework. The architecture allowed execution of a range of event driven applications in an energy efficient manner.

1.1 Motivation

The non-ISA dataflow processor architecture [8] is limited in scope with respect to the class of applications that can execute on it. Thus, the preliminary version of the architecture cannot execute most of the real world sensor applications it was designed for.

The non-ISA dataflow architecture [8] uses a single event-bus. Event-buses are used to transfer data between modules. A functional unit starts execution only after it receives all inputs, performs the required functionality, and generates output data. The output data is transferred to the destination module on predefined cycles through the event-bus. Thus, if the event-bus is in use by a module, all other modules with data ready to be transferred will have to wait until the event-bus has completed transfer of the data. This leads to a schedule overhead, which in turn leads to energy inefficiency.

The work presented in this thesis has developed an automation framework with a scheduler

capable of scheduling realistic sensor applications. Additionally, this thesis presents an architecture that supports multiple event-buses and a new set of modules and applications to test the architecture.

1.2 Contributions

This thesis describes two primary contributions leading to the design of an energy efficient processor capable of executing a range of event driven applications.

The first contribution is the design of an automation framework with an object oriented, stack based, backtracking scheduler. The scheduler is designed to schedule a range of applications on the dataflow architecture. The scheduler is also designed to find all the possible schedules for a given application and implement the schedule with minimum schedule length on the architecture. This improves the energy efficiency of the processor.

The second contribution is the development of a multiple event-bus processor architecture. The module architecture is augmented to support multiple output registers. Two new modules are created to enable the use of multiple buses.

The effectiveness of the scheduler is demonstrated by the successful execution of a range of applications on a wide range of architectures using the scheduler. The effectiveness of the multibus architecture is shown by the comparison of the energy usage accrued by running an application on the single bus and multiple bus versions of the architecture.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2 discusses the related research. Chapter 3 discusses the design of the multiple event-bus dataflow processor architecture. Chapter 4 explains the improved design of the automation framework, including details of the components of the framework. Chapter 5 discusses the analysis of effectiveness of the scheduler and the multibus architecture. The chapter also discusses the effectiveness of the multibus architecture for specified applications in terms of energy efficiency. Chapter 6 summarizes the contributions and identifies areas for future work.

Chapter 2

Background

This chapter discusses the background information related to this thesis. A brief introduction of electronic textiles is given in Section 2.1, demonstrating the need for a specialized processor architecture. Section 2.2 provides an overview of the research in electronic textiles architecture, dataflow architecture, and event driven architecture that helps in understanding the motivation behind this thesis. Section 2.3 discusses some scheduling algorithms, differentiating the scheduler developed in this thesis from them.

2.1 Introduction to Electronic Textiles

Electronic textiles (e-textiles) are intelligent fabrics that have sensors and computational elements embedded in them in an unobtrusive manner. As seen in the examples below, e-textile applications consist of an array of sensors connected to computation elements over a wired network. Most of the sensors and computation elements are battery powered. Hence, the architecture of the computation elements should be suited to handle such sensor applications and should also complement the low power requirements of e-textile applications.

Thus, there is a need for a specialized processor architecture. This thesis describes the development of such a processor architecture.

Though e-textiles is a relatively nascent field, it has found applications in different spheres of life. The "Wearable Motherboard project" at Georgia Tech created an e-textile capable of monitoring human vital signs [10]. The wearable motherboard consisted of sensors to monitor respiration rate, heart rate, and temperature connected to monitoring devices over a network of fiber optic and specialty fibers. In addition, a microphone had been integrated to the motherboard to transmit the user's voice to monitoring stations. A "health shirt" for round the clock blood pressure monitoring was described in [11]. The blood pressure was estimated by monitoring the ECG and PPG signals. A wrist watch was used to process the signals and display the blood pressure level of the patient. The signals from the ECG and PPG sensors were directed to the watch through e-textiles fabric enclosed within the cloth. The "body-worn e-textile antennas" described in [12] is an example of the use of e-textiles in the field of communication. An eight element e-textile antenna array was constructed and gave a performance similar to regular antennas. This e-textile application was intended for use in space missions by astronauts or for improvement in tracking and location applications such as the global positioning system (GPS). "OFSETH" used optical fibers to keep a track of vital signs of a patient under medical resonance imaging (MRI) and preventing sudden deaths in infants [13]. Fiber optic sensors were used for monitoring ECG, blood pressure, carbon dioxide waveform, respiratory rate and other vital signs. The optical signals were transmitted through on fabric optical fibers to an opto-electronic module, where the signals were conditioned and digitalized. The signals were then transmitted to an acquisition module. The "MagIC System"consisted of a vest and a portable electronic board. The vest consisted of sensors for measuring ECG and respiration rate. The signals from the sensors were fed to the electronic board through conductive fibers. The electronic

board also consisted of a two axis accelerometer and transmitted all the signals to a remote computer [14].

The e-textiles group at Virginia Tech has worked on many sensor applications over the past few years. An acoustic beam forming array was developed to determine the location of passing vehicles [15] [16]. The array consisted of a cluster of sensors with several microphones in each cluster. The system then used an algorithm, which used the data collected from the cluster of sensors to come up with the location of the vehicle. An "E-Textile Based Automatic Activity Diary" was developed in [17]. The system consisted of various piezoelectric sensors, heart rate sensors, temperature sensors, and communication devices woven into a body suit. This body suit kept a track of various ambulatory and context related data required for medical applications.

2.2 Related Research in Architectures

This section describes some of the research performed in the field of e-textiles architecture, event driven architecture, and dataflow architecture.

A two-tier architecture for an e-textile application was proposed in [18]. The advantages of using a wired network embedded in the fabric, compared to a wireless sensor network was also discussed. In the two-tier architecture, tier-1 nodes collect data from a cluster of sensors. The nodes perform simple operations on the data and pass it over the on fabric network to the tier-2 nodes. Tier-2 nodes contain the actual processing units that perform the computation needed by the application. The architecture described in this thesis is for the tier-1 nodes.

Sensor applications are inherently event driven in nature. General purpose microprocessors

use an event driven operating system to mask the deficiencies of hardware that have not been designed for event driven sensor applications. TinyOS [19], and Contiki [20] are examples of such operating systems. Hardware architectures have also been developed to handle event driven applications. A Multi-log processor is an example of such a hardware architecture [21]. The processor is divided into a number of processor cores, with each core having an ALU, an event handler, a speculative local cache, a hardware event queue, and a register file. The event handler in each core fetches the event at the head of the local event queue, decodes it, and sets the correct control signals in the ALU. The event handler and the hardware event queue handle event driven applications successfully without the need for an event driven operating system. The software overhead associated with an event driven operating system is removed. This architecture has an instruction set architecture (ISA) and requires hardware to fetch and decode instructions. In contrast, the architecture proposed in this thesis does not have an ISA and eliminates the instruction fetch and decode hardware.

A low power event driven architecture for sensor networks was proposed in [22]. This architecture consisted of a microcontroller, a bus arbiter, and an event processor as master devices and hardware acceleration units such as an ADC, a radio, a data filter, a timer, and memory as slave units. Events were sent as interrupts to the hardware acceleration units, which then compete for the interrupt signal in the bus interface. The event processor and the microcontroller act as master devices, servicing the interrupt requests generated by the slave devices and transferring data between the slave devices. The architecture is energy efficient as the microcontroller is powered off most of the time and cycle efficient hardware accelerators perform the required functionality. There is still power usage associated with servicing interrupts and bus arbitration. The architecture described in this thesis uses similar hardware units for implementing the functionality required by the applications. However,

the architecture does not need bus arbitration or interrupt servicing as the execution is made deterministic through configurations sent to the specific modules.

The Sensor Network Asynchronous Processor (SNAP) is another example of an event driven hardware architecture [23]. The SNAP has a processor core connected to the network through incoming and outgoing message buffers. The architecture has a hardware execution queue which contains tokens representing messages or executable events. New messages are inserted by the incoming message buffer. The SNAP processor uses a co-processor to manage the execution queue and insert executable events into the queue. The SNAP does not have a cache, virtual memory, and exceptions. Therefore, the non determinism caused by these elements is removed. However, the SNAP does have an ISA and instruction fetch and decode units, making the processor and the co-processor essentially sequential. In contrast, the processor architecture described in this thesis does not have an ISA which allows parallel execution of its modules.

A dataflow architecture is a computer architecture in which instructions execute when all of the input arguments are available. The dataflow architecture does not use a program counter, which causes sequential execution in the von Neumann architecture. Thus, dataflow architectures allow programmers to explicitly express parallelism [24]. An elementary dataflow architecture is described in [25]. In this architectural model the memory is split into instruction cells, where each instruction cell corresponds to an operator. Each instruction cell consists of three registers. The first register stores the operation and the address of the target register. The second and third registers store the operands. When all the registers are filled, the instruction is ready for execution. An arbitration unit transfers the data in the form of an operation packet to the operation unit. After the operation has been performed, the distribution unit transfers the data packets to the correct memory addresses. The operation units are pipelined for increased throughput. This architecture is successful in achieving a

degree of parallelism. The architecture requires an arbitration unit and a distribution unit to avoid conflicts. These units consume power throughout the execution cycle. In contrast, the architecture described in this thesis does not require an arbitration unit because of its design which allows deterministic execution of applications.

Another example of a data flow architecture is WaveScalar [26]. Wavescalar is a dynamic dataflow architecture where multiple values are allowed to wait at the input arc of an instruction. Wavescalar differs from other dataflow architectures as it has a memory ordering scheme, which allows programs written in traditional languages like C to execute on it effectively. Wavescalar still uses an ISA, where the nodes of the dataflow graph correspond to the instructions and the arcs represent the dependencies between them. In contrast, the architecture described in this thesis does not use an ISA. The architecture uses configurations to map an application on the architecture.

A non-ISA based dataflow architecture was developed in [8]. The work done in this thesis is part of the same project in which the architecture in [8] was developed. The architecture consisted of independently operating coarse grained modules. The coarse grained modules communicated with each other through an event-bus. An input FIFO and an output FIFO were used for communication with the high speed network. An automation framework for this architecture was designed in [9]. The framework automatically generated configuration values for the hardware modules based on the application. These configuration values ensured that there were no run time resource conflicts. Thus, the architecture ensured deterministic execution and did not require arbitration units to resolve conflicts. However, this architecture could not execute large applications and was constrained in the type of applications that could execute on it. The work described in this thesis was built on the architecture developed in [8] and [9] and made improvements to it so that a wide variety of applications could be executed on the architecture.

2.3 Related Research in Scheduling Algorithms

The architecture described in this thesis uses a scheduling algorithm to come up with delay values associated with output registers of modules. This makes the execution of an application on the architecture deterministic and ensures the absence of an ISA and a bus arbiter. This section reviews the related research that has been carried out in the past in developing scheduling algorithms.

Scheduling algorithms have traditionally been developed for multiprocessor systems. The scheduling algorithms can be classified as static and dynamic depending on whether the scheduling is performed at compile time or run time. Static scheduling algorithms use a Directed Acyclic Graph (DAG) model. The program to be scheduled is represented as a graph, where the nodes of the graph represent tasks and the edges represent the dependencies between the tasks. The aim of the scheduling algorithm is to map the task onto the processors such that the task dependencies are honored and the overall program finishes as early as possible.

The Highest Level First with Estimated Times (HLFET) described in [27] is a static scheduling algorithm for multiple processors. The longest distance of each node from the exit node is calculated statically. A ready list is prepared, which initially contains only the entry nodes. The first node in the ready list is scheduled to a processor that would allow its earliest execution. The ready list is then updated to contain all the nodes that are ready to be executed. The ready list always contains nodes in descending order of their distances from the exit node. The time complexity of the HLFET algorithm is found to be O(v^2), where v is the number of nodes in the DAG. This algorithm does not take into account the condition when two nodes can start at the same time. This is because this algorithm is targeted towards real time systems, where deadline satisfaction is the primary objective.

The algorithm does not need to go through all possible combinations of execution of nodes. In contrast, the scheduler described in this thesis takes this condition into consideration and all combinations of executing edges are tried.

A dynamic scheduling algorithm for parallel video processing was described in [28]. In this algorithm the ready tasks are inserted in deadline order into a ready queue. The feasibility of a task is checked early and the partial solution is extended only if it is found to be feasible. If the partial solution is not found to be feasible, the algorithm forces the search tree to backtrack and try the next task in the ready queue. The algorithm in [28] has been developed for a real time system, where the primary concern is deadline satisfaction. Also, there are strict constraints on the speed of execution of the algorithm. Thus, the algorithm is not concerned about finding the minimum possible schedule. The scheduler described in this thesis uses a similar backtracking approach to come up with a valid schedule. If a partial solution is not found to be feasible due to resource conflicts, the current scheduler backtracks to a edge which caused the conflict. However, the current scheduler is not restricted to find a valid solution. It finds the minimum possible schedule by going through all the possible edge execution orders.

The myopic scheduling algorithm described in [29] is another example of a dynamic scheduling algorithm. The tasks are always arranged in ascending earliest deadline order. The algorithm starts with an empty list of tasks. At each stage it selects the task with the earliest deadline from among a set of k tasks, which is a subset of the remaining tasks. The algorithm initially checks for strong feasibility of the partial solution. A partial solution is said to have strong feasibility if the schedule found by extending the partial solution by any of the remaining tasks is feasible. If the partial solution is not found to be strongly feasible, the algorithm discards the partial solution and backtracks to the previous partial solution. It then selects the task with the second lowest value of earliest deadline and continues. How-

ever, the backtracking is restricted to overcome the overhead of backtracking. Hence, the best schedule is not always found. This algorithm is found to have O(n) time complexity. The scheduler developed in this thesis also uses a similar backtracking approach. However, the backtracking in the scheduler is not restricted. Backtracking is used extensively to come up with the shortest schedule.

A backtracking scheduling algorithm for clustered Very Large Instruction Word (VLIW) architectures is described in [30]. The algorithm schedules instructions based on a Data Dependence Graph (DDG). The algorithm is divided into two phases. In the preliminary scheduling phase, an instruction is selected from a ready list based on its height in the DDG. After this step, a valid schedule point is found for the instruction. In the next phase called the backtracking optimization phase, the algorithm examines the schedule generated in the preliminary phase and finds out where optimizations can be made. In contrast, the scheduler developed in this thesis does not explicitly have a backtracking optimization phase. It improves on the valid schedule generated in the first try by going through all possible execution orders of the edges.

A backtracking scheduling algorithm for project selection and task scheduling was proposed in [31]. The algorithm determines the scheduling of a list of projects so as to maximize profit. The algorithm goes through a tree containing all the permutations of the projects available, backtracking when the cost of a partial schedule is larger than the minimum cost of a schedule on record. The algorithm developed in this thesis follows a similar approach to backtracking, going through all possible combinations of schedulable edges and aborting a partial schedule when it is found to be larger than the minimum schedule on record.

Chapter 3

Multibus Architecture

This chapter describes the design of the multibus dataflow processor architecture. Section 3.1 gives an overview of the features of the multibus architecture. Section 3.2 describes each component of the architecture in detail, focusing primarily on the use of multiple output registers and the use of multiple event-buses.

3.1 Multibus Architecture Overview

This section discusses the salient features of the multibus architecture. The main features of the architecture are a deterministic timing behavior, a native support for event driven applications, and an inherent support for concurrency. Figure 3.1 shows an example instance of the multibus architecture. The multibus architecture was built using the preliminary architecture developed in [8]. The multibus architecture uses the same input register module and FIFO as the preliminary architecture in [8]. The parameterizing of the packet structure has been implemented in [32]. The work described in this thesis augmented the preliminary

architecture developed in [8] towards a functional module structure supporting multiple output registers and multiple event-buses.

The multibus architecture consists of independently operating functional units. The functional units are coarse grained modules that perform a specific function. The execution of an application on the architecture can be demonstrated through a sample application shown in Figure 3.2. The application consists of three modules: a timer, an ADC, and a delay unit. The timer generates periodic events. The ADC starts sampling analog data upon reception of an event from the timer. The converted data is sent to a delay unit, which counts down a specified number of cycles before putting data on the output bus. As seen in the timing diagram in Figure 3.3, the configuration values are sent initially in reverse order of data flow. These configuration values include wrapper configuration packets and internal configuration packets. When the timer generates an event and puts data on the bus, the input register of the ADC starts reading data at the same time. The ADC takes ten cycles to sample the data and generates an output data packet. This packet is read by the delay unit, which waits five cycles before putting data on the bus. This is followed by a number of free cycles. The duration of the free cycles is controlled by the input FIFO. It is during this period that an application can be programmed through the network. The timer fires again after the free cycle period. The time period of the timer is the sum of the schedule period and the free cycle period.

The functional units communicate with one another through the event-buses. The implementation of the multibus architecture described in this thesis has two event-buses, but other arrangements are possible. A functional module starts execution after it receives all the inputs, performs the necessary function, and generates output data. The output data is transferred to the destination functional modules on predefined cycles through the corresponding event-bus. Data is passed between modules through data packets. The packet

structure used in the architecture is parameterized, with control over the bit size of the module address, the wrapper configuration register address, the configuration, and the data in a packet. The parameterized packet structure is described in detail in Section 3.2.1. The processor architecture is designed with a FIFO for communicating with the network. The FIFO is used to synchronize the transfer of events between the low speed processor architecture and the high speed network.

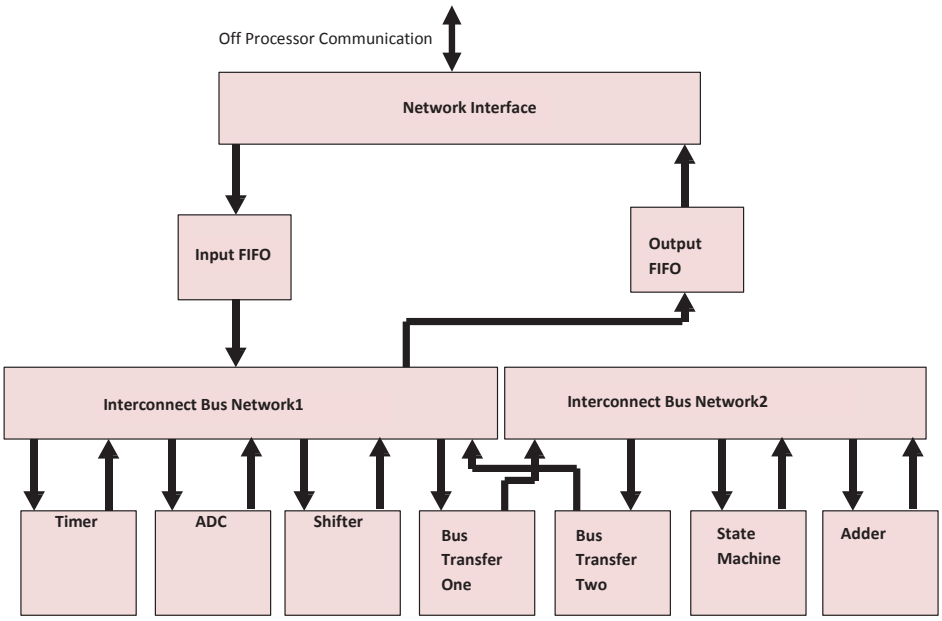

Figure 3.1: A High Level Representation of an Instance of the Multibus Architecture

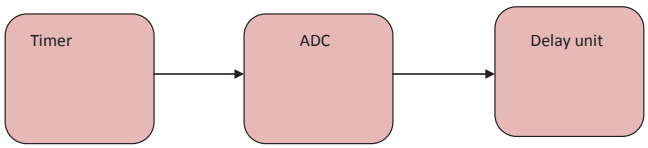

Figure 3.2: Sample Application Graph

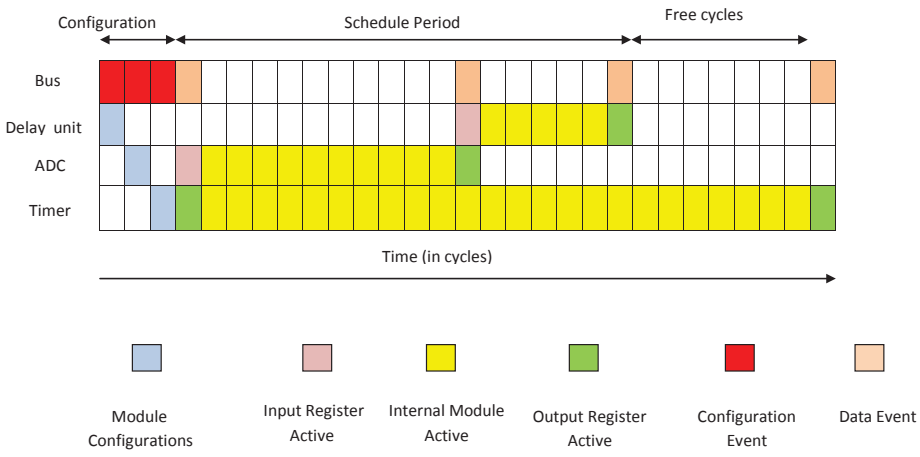

Figure 3.3: Timing Diagram of a Sample Application

3.2 Multibus Architecture Components

This section describes the components of the multibus dataflow processor architecture. The components of the architecture including the event-buses, the packet structure, and the coarse grained modules are described in this section.

3.2.1 Interconnection Bus Network and the Packet Structure

The functional modules communicate with each other and with the network using an interconnection bus network. The time sharing of the bus by the functional modules is deterministic. Thus, there is no need for an arbitration logic to resolve collisions. A bus takes one cycle to transfer a data or a configuration event. The buses do not differentiate between the transfer of data and configuration, that is the responsibility of the modules.

This section illustrates the packet structure used in the multibus architecture. The execution of an application on the architecture is controlled by the transfer of configuration and data

events. The transfer of events is through fixed length packets. The packet structure in the architecture is parameterized. The bit size of the module address, the wrapper configuration register address, the data, and the configuration can be changed. This feature is useful because the architectural requirements vary between applications. Some applications require an architecture with a large number of hardware modules, which in turn requires more module address bits, while other applications use a large number of output registers, which requires more wrapper configuration registers. The packet structure for data and configuration events is shown in Figure A.1 and Figure A.2 of the Appendix respectively. The individual bits of the packet structure have also been explained in detail in Section A.1 of the Appendix.

3.2.2 Module Design

This section describes the design of the individual functional modules. The design of these modules has a parameterized template structure as given in Figure 3.4, though each individual module can vary in functionality.

A functional module has *MOD_ID, ADDRESS_BITS, WR_CONFIG_BITS, MAX_REUSE, INT_CONFIG_BITS* as parameters, where *MOD_ID* indicates the hardware address of the module, *MAX_REUSE* indicates the maximum number of times a module can be used in a given application, *WR_CONFIG_BITS* indicates the number of bits used to address the wrapper configuration registers, and *INT_CONFIG_BITS* indicates the number of bits used to address the internal configuration registers. Each of the modules can be connected to any event-bus.

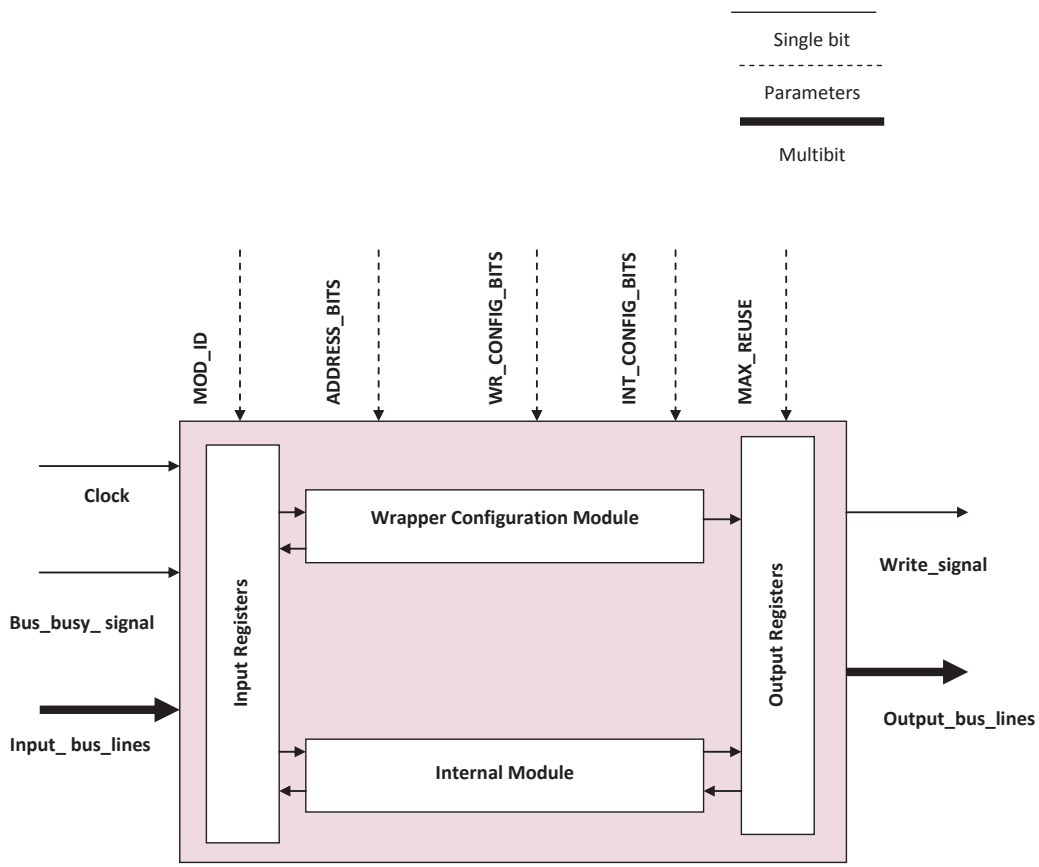

Figure 3.4: A Generic Representation of the Parameterized Template Structure

3.2.2.1 Input Register Module and Wrapper Configuration Module

The multibus architecture uses the same input register module used in [8]. A description of the input register module can be found in Section A.2 of the Appendix.

Figure 3.5 shows a high level design of the wrapper configuration module. This module plays the central role in the use of multiple output registers. Hence, it has been described in detail in this section. This module configures the the wrapper registers that help in the creation of event packets. The wrapper registers ensure that each instance of a module passes on data to the correct destination and the required output registers of each instance of the module have

the correct delay values associated with them. The wrapper configuration module shown in Figure 3.5 has three output registers. Each of the output registers has corresponding output and handshake signals in the wrapper configuration module. The *wr_config_pkt_used* signal is asserted after the wrapper module receives a configuration packet from the input register module and after it has set the corresponding configuration registers. The *wr_config_pkt_used* is used to reset the *wr_config_pkt_rdy* signal of the input register module. The *out_reg1_active, out_reg2_active*, and *out_reg3_active* handshake signals indicate when the internal module can put data to the corresponding output register module. These signals are asserted and reset based on the instance of a module in use at that time

Table 3.1 shows the address of the wrapper configuration registers and the configurations associated with them. The module active indicates if a particular functional unit is ready to operate. The num used configuration indicates the number of times the corresponding functional unit is reused in an application. The destination address indicates the address of

Table 3.1: Wrapper Configuration Details

Address	Configuration Registers
000	module active
001	num used
010	destination/delay for output register 1
011	destination/delay for output register 2
100	destination/delay for output register 3
101	destination/delay for output register 4
110	destination/delay for output register 5
111	new instance of a module

the destination functional unit corresponding to the given instance of a functional module. The delay value indicates the number of cycles that the output bus interface of the given instance of the functional module has to wait before putting the data on the bus. The wrapper configuration module differentiates between an address and delay based on the

order in which the packets arrive. An address configuration packet is always sent before a delay configuration packet. The configuration register with the maximum address, register with address 111 in this case is used to indicate a new instance of a module. This helps in resetting or asserting the *out_reg_active* signal for the output register being used by the corresponding instance of the module. Table 3.1 shows that there are different configuration registers for each output register used. Also, this set of configuration registers exist for each instance of the module. As seen, a maximum of five output registers can be used by a module when three bits are used to address the configuration registers.

3.2.2.2 Internal Module

Figure 3.6 shows a high level design of the internal module. The internal module plays an important role in the selection of the output register that a module uses. Hence, it has been described here. The internal module implements the functionality associated with a module. A generic structure of the internal module of a module that uses 3 output registers is shown in Figure 3.6. The internal module reads the data packet from the input register module, performs the required operation on the data, and asserts the *inp_reg_file_used* to reset the *data_pkt_rdy* signal in the input register module. The internal module then checks if it has received all the inputs and if the results of the previous execution have been transferred to the output register module. If true, the internal module sends the output data to the corresponding output registers based on the *out_reg_active* signal that is high at that time.

The internal module has internal configuration registers associated with it. The internal configuration registers help in controlling the functionality of the internal module. The internal configurations are specified in the nodes of the application graph. The automation framework generates the internal configuration packets for each instance of a module based on the values specified in the application graph.

3.2.2.3 Output Register Module

Figure 3.7 shows a high level design of the output register module. The output register module is connected to any one of the buses by the output_bus_lines. The output data packet is formed by combining the *output_data* from the internal module and the *dest_add* from the wrapper configuration module. The output register then checks if the *out_reg_active* signal is high. If so, the output register module asserts the *out_data_used* signal to reset the *out_data_rdy* signal of the internal module and begins counting down from the *delay_value*. After the specified number of wait cycles the data packet is put on the output bus associated with the module.

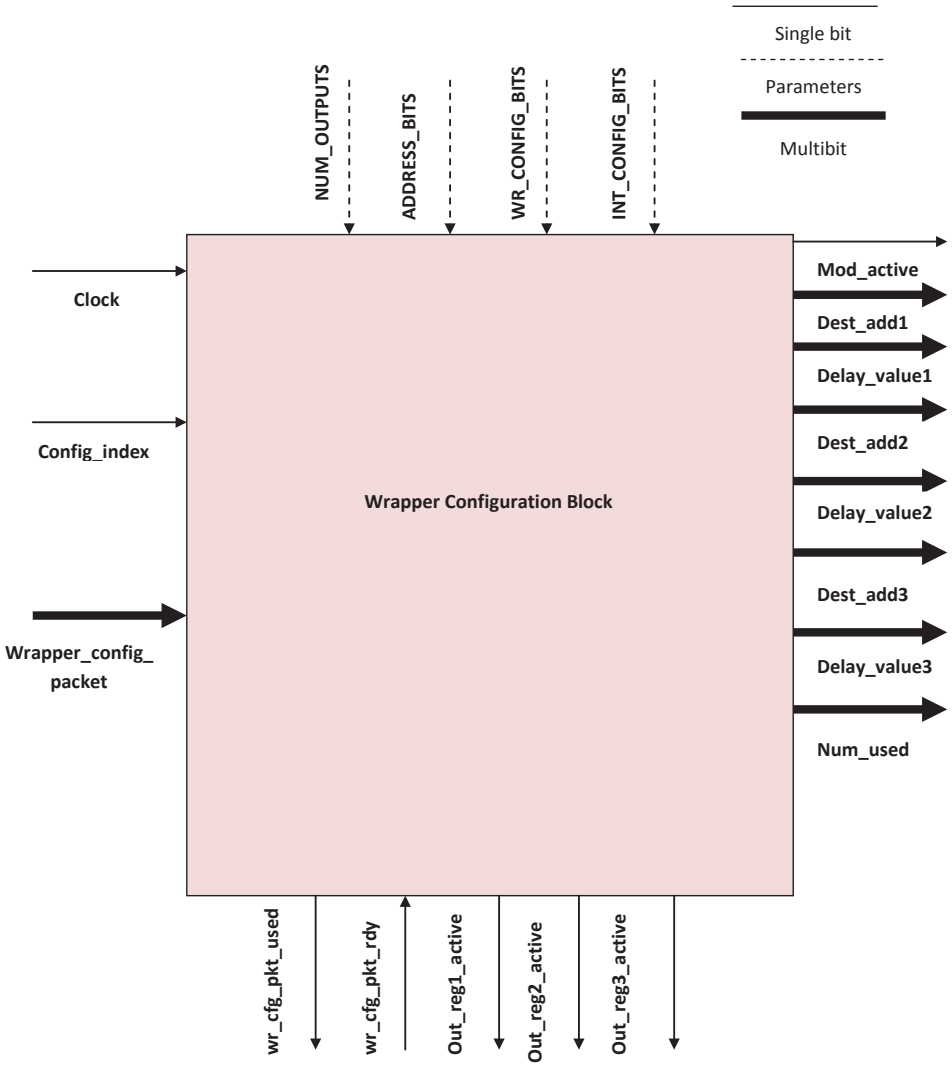

Figure 3.5: A High Level Representation of the Wrapper Configuration Block

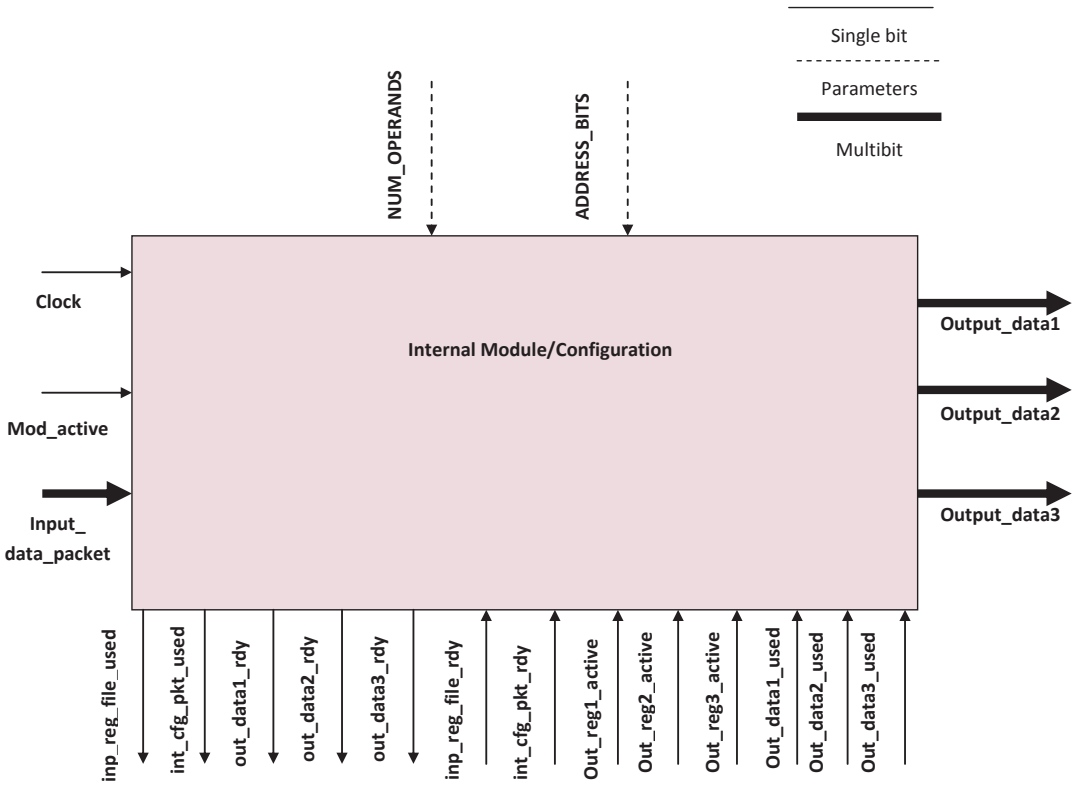

Figure 3.6: A High Level Representation of the Internal Module

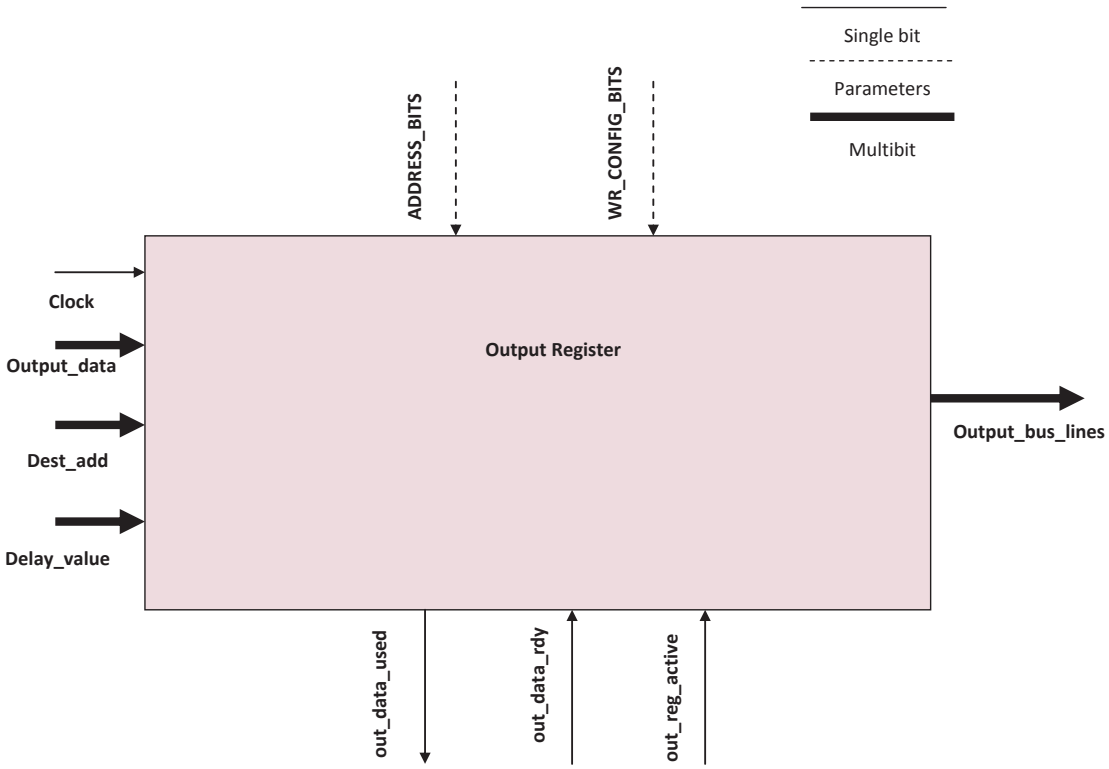

Figure 3.7: A High Level Representation of the Output Register Module

Chapter 4

Automation Framework Design

This chapter discusses the automation framework of the multibus architecture. The multibus architecture is programmed through configuration values. The internal configuration values are generated from the application. The wrapper configuration values on the other hand are generated automatically by the automation framework. The wrapper configuration values include the destination address values for each instance of all modules used by the application and the delay values for the output register of each instance of every module used by the application. Section 4.1 gives an outline of the automation framework. Section 4.2 gives a detailed explanation of the object-oriented design of the scheduler.

4.1 Automation Framework Outline

This section gives an outline of the automation framework including the contribution of this thesis towards the automation framework.

An outline of the automation framework can be seen in Figure 4.1. The automation frame-

work consists of three parts, a prototyping framework, a validation framework, and a configuration framework. The prototyping framework is responsible for generation of an architecture instance. This architecture instance is given as an input to the mapper in the configuration framework. The prototyping framework is also responsible for generating the synthesized,

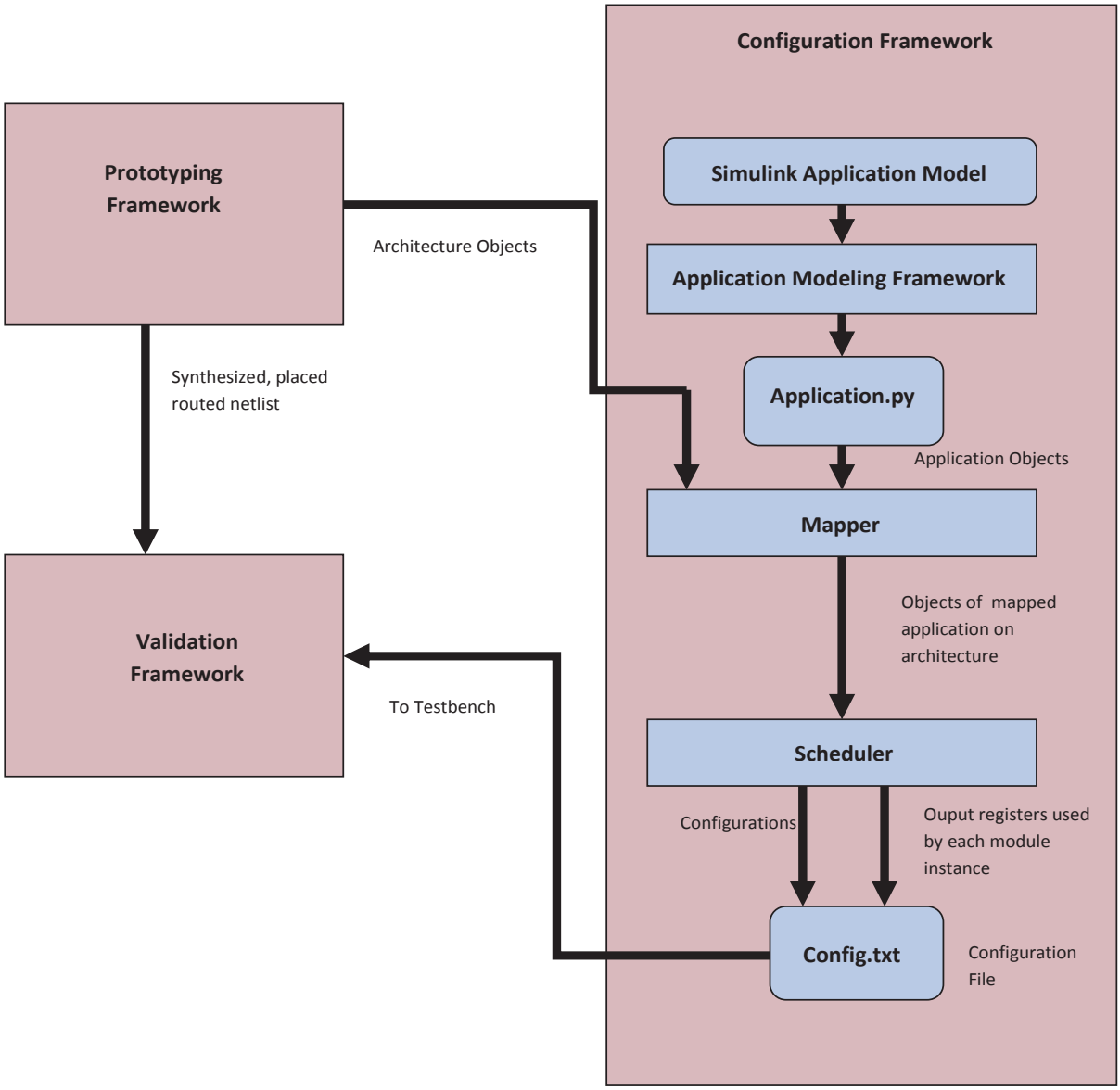

Figure 4.1: Automation Framework Outline

placed, and routed netlist from the architecture files using ASIC tools. The multibus architecture discussed in this thesis used the prototyping framework developed in [9].

The validation framework is responsible for generating the result files after simulation. The synthesized, placed, and routed netlist is given as an input to the ASIC simulation block of the validation framework. The ASIC simulation block receives the configuration values from a test bench generation block of the validation framework. The results of the simulation are passed on to the result extraction block, which generates the power and area results. The multibus architecture presented in this thesis used the the validation framework developed in [8].

The configuration framework is responsible for generating the configuration values for each instance of the hardware modules. These configuration values ensure deterministic timing behavior of the architecture. The configuration framework flow has been shown in Figure 4.1. The application modeling framework takes as input a Simulink application model. The application modeling framework parses the Simulink model and generates an application graph in the form of a python file. This application graph along with the architecture object, generated by the prototyping framework are given as inputs to the mapper. The mapper maps the application graph on to the architecture and passes it on to the scheduler. The scheduler generates the configuration values and the output registers to be used by each instance of the modules in the architecture. In the configuration framework, the application modeling framework has been developed in [32]. The work presented in this thesis used the mapper developed in [9]. This thesis primarily discusses the work done in developing the scheduler in the configuration framework. The scheduler has been described in detail in Section 4.2.

4.2 Scheduler Design

Table 4.1: Scheduler Functions Properties

Functions	Arguments	Explanation
SCHEDULE	G(V,E)	The top level function which initializes elements and calls BACKTRACK with the graph G(V,E) and edge 0 as arguments.
BACKTRACK	G(V,E), back_edge	Backtracks to edge back_edge in graph G(V,E)
UPDATE	current_schedule	If current_schedule is less than minimum_schedule, update min_schedule to current_schedule.
SOLUTION	-	Keeps a track of permutations of current set of ready edges. Also keeps a track of the permutations that have been tried so far. Changes order of execution of edges to the next untried combination.
SIMULATE	edge	Simulates the execution of the edge. On failure due to resource conflict, returns edge to backtrack to.
UPDATE_DELAY	edge	Computes the delay that a edge should have in order to avoid resource conflicts. The edge delay of the edge is then updated to the predicted value.
UPDATE_OUTPUT_REG	node, edge	Changes the output register used by the node associated with the edge to avoid resource conflicts.

This section gives a detailed description of the scheduler which is a part of the configuration framework of the architecture. As explained in Section 3.2.2.3, resource conflicts in the multibus architecture are avoided by the delay configuration values. This lends determinism to the execution of an application on the architecture and allows the absence of any bus arbitration logic. The scheduler is responsible for determining these delay configurations.

```
SCHEDULE( G(V,E) )                          //Outer scheduler function
    for each edge u ∈ G(E)
        u.visited = 0                       //Initialize all edges in the application graph as unvisited
    BACKTRACK( G,0 )//Call backtracking function with the application graph, and edge0 as arguments

BACKTRACK( G( V,E),u )                      //Backtracking function
    If  u.index ≥ |E|           //Check if all edges for a schedule instance have been placed
        if all_schedules_found == false     //Check if all schedules have not been found
            UPDATE(current_schedule ) //Update minimum schedule
            ret2 = SOLUTION()//Change order of execution of edges and return to an edge
            return  ret2
        else
            return 0 //All schedules found
    If  u.delay ≤ max_delay
        ret = SIMULATE( u )                 //Call SIMULATE function to simulate execution
        if ret == success                   //No conflicts, and edge u successfully placed
            ret1 = BACKTRACK( G, u.next)//Call BACTRACK with next edge
            if ret1 != u.index
                return ret1                 //Return to a particular edge
        else
            return ret
    else
        exit()                              //No Schedules found

SIMULATE( u )
    While stack.top.index  ≥ u.index
        stack.pop()                 //Pop the scheduler objects corresponding to edge u from stack
    current_state = stack.top
    u.visited  =  1                         //Mark the edge as visited
    if resource conflicts exist due to vertex v ∈ G(V)
        UPDATE_OUTPUT_REG(v.input_edge.source_node )//Update output register of node
        UPDATE_DELAY(v.input_edge )//Update delay of input edge of conflicting node
        stack.push( u.index , scheduler_objects)
        return v.input_edge.index       //Return to input edge of conflicting node
    else
        stack.push( u.index , scheduler_objects)
        return success                      //Edge successfully placed
```

Figure 4.2: Scheduling Algorithm Outline

Thus, the scheduler plays a central role in the functioning of the multibus architecture described in Chapter 3.

A top level scheduler script takes as an input the application mapped on the architecture from the mapper. The script then generates a text file containing details of the application, architecture, and mapping of the application on the architecture. The work presented in this thesis used the top level scheduler script developed in [9].

The text file is read by the object oriented scheduler. An outline algorithm of the object oriented scheduler presented in this thesis is shown in Figure 4.2. Table 4.1 and Table 4.2 give a detailed description of the functions and variables used in the scheduling algorithm respectively.

The scheduler constructs a graph G(V,E) from the text file, where V and E represent the vertices and edges of the graph respectively. This graph encompasses all the details of the application graph and the mapping of the application on the architecture. The main scheduler function SCHEDULE(G(V,E)) initializes all edges of the graph G(V,E) as unvisited. This function then calls the BACKTRACK function with G(V,E) and edge 0 as arguments.

The BACKTRACK function provides the outer framework needed for recursive backtracking. The function initially checks if the edge delay is less than or equal to the maximum permissible delay on the architecture. If not, the scheduler exits as a schedule could not be found. If the edge delay is within the permissible limit, the function calls the SIMULATE function which simulates the execution of the application edge on the architecture. The SIMULATE function returns success on successful execution of the edge and the index of the input edge of the conflict causing vertex in case of any conflicts. If the return value is not success, the edge delay value of the conflicting edge is incremented and SIMULATE is

called with the conflicting edge as an argument. If not, the next edge of the graph is passed on to the SIMULATE function.

The SIMULATE function makes use of a stack structure for backtracking. When the BACKTRACK function calls the SIMULATE function with a conflicting edge as an argument, SIMULATE does not start execution from edge zero. Instead it pops the stack until the saved state with an index of the conflicting edge is reached. The scheduler objects from the stack are then copied into the current state. This saves a lot of overhead. Similarly when an edge has completed execution completely or when there is a conflict, the scheduler objects are pushed on to the stack with the corresponding edge index as the stack index.

The scheduler uses useful functions like UPDATE_DELAY and UPDATE_OUTPUT_REG. The UPDATE_DELAY function is used to calculate the delay value required by the conflicting edge to avoid the conflict. The UPDATE_OUTPUT_REG function updates the output register used by the source node of the conflicting edge if needed. Both these functions help in avoiding excessive backtracking and hence, save a lot of overhead due to recursion. The edge delay values found using the scheduler are the delay units the output register of the source node of the edge should wait before putting data on to the bus.

The scheduler is designed to find all the possible solutions for a particular graph G(V,E), and implement the least schedule on the architecture. Hence, when a valid schedule has been found, the BACKTRACK function checks if the global variable all_schedules_is false. If so, the function SOLUTION() is called. This function reorders the edge execution order and keeps a track of the order in which the edges have been already executed. The reordering is done by changing the edge.next component of an edge. After all the combinations of the edges have been tried, the function sets the global variable all_schedules_found to be true. Thus, the algorithm goes through all the combination of paths of edges possible and finds the least schedule.

Table 4.2: Scheduler Variables Properties

Variables	Explanation
edge.visited	Indicates if an edge has been visited.
edge.delay	Delay associated with edge.
edge.index	Edge number of the edge.
edge.next	Points to the next edge in the sequence, if an edge has been successfully placed. This pointer is modified by SOLUTION to change the order of execution of the edges. In the normal order of execution, next points to edge.index+1.
edge.source_node	Source node of the edge.
vertex.input_edge	input edge of the vertex.
current_state	An object encompassing all the objects of the graph G(V,E), used to store the current state of the graph after backtracking and popping from the stack.
max_delay	The maximum delay allowable for any edge in the graph G(V,E).
max_edge	The number of edges in the graph G(V,E).
scheduler_objects	An object encompassing all the objects of the graph G(V,E), used to store the current state of the graph. This is pushed on to the stack if an edge was successfully placed.
all_schedules_found	Indicates if all the schedules have been found. This is set to true in the SOLUTION function after all the combinations of edges have been tried.
stack	A data structure used to store the state of the graph G(V,E) at different points of time. Each element of the stack corresponds to a state when a particular edge of the graph has been placed.
stack.top.index	The sequence number of the top of the stack. The elements of the stack are stored with an index corresponding to an edge.index.

Chapter 5

Results

This chapter discusses the results of this thesis. The first section lists the experimental architectures and applications used to obtain the results in this thesis. The second section analyses the efficiency of the current scheduler. The third section discusses the advantages of using multiple output registers in a hardware module as compared to multiple hardware modules on the architecture. The fourth section demonstrates the energy efficiency and effectiveness of a multibus architecture compared to the single bus architecture. The fifth section shows the energy efficiency of the multibus architecture compared to the commercial PIC microcontroller.

5.1 Applications and Architectures

This section describes the different architectures and applications that are used in the thesis and states the reason behind choosing them. The configuration framework is used to map an application on the architecture and to configure the architecture to execute the applica-

tion. The general flow of the configuration framework can be seen in Figure 4.1. All the architectures used to test applications in the thesis have a bus width of 16.

5.1.1 Applications

This section describes the six sets of applications that are used to obtain the results. The application sets cover a wide range of areas that commercial microcontrollers like the PIC are used for. These areas include sensor applications, control applications, signal processing, and applications for mathematical computation. The range of applications demonstrate the flexibility and capability of the architecture. The applications are described in detail below.

Table 5.1: Characteristics of the Applications

Application	Number of Nodes	Number of Edges
Thermostat	9	11
Two bit multiplier	17	22
Four bit multiplier	35	48
Six bit multiplier	53	74
Eight bit multiplier	71	100
Two bit multiply and sum	34	46
Two coefficient filter	11	13
Three coefficient filter	15	19
Eight coefficient filter	35	49
Square root solution	28	39
Free fall detector	33	47
Robot path follower	23	29

Application-I set[1] [9] consists of a thermostat application shown in Figure C.1. This consists of an ADC, which samples a sensor value periodically based on a timer event. The output

[1]The Application-I set was developed in [9].

of the ADC is compared with a set temperature threshold. The difference is fed to a state-machine which controls an actuator.

Application-II set consists of a two bit multiplier[2] [9] shown in Figure C.2, a four bit multiplier shown in Figure C.3, a six bit multiplier shown in Figure C.4, an eight bit multiplier shown in Figure C.5, and a two bit multiply and sum shown in Figure C.10. All of the multiplier applications use a shift and add approach to multiply the operands.

Application-III set[3] [32] consists of a two coefficient FIR filter shown in Figure C.6, a three coefficient FIR filter shown in Figure C.7 and an eight coefficient FIR filter shown in Figure C.8. The FIR filter is a type of a signal processing filter which performs a convolution of the input values with the filter coefficients and produces a sequence of output values [33].

Application-IV set consists of a square root solution application shown in Figure C.9. While a common computation, the square root function is not available as part of the instruction set in most microcontrollers. The square root of an eight bit number is calculated in this application [34]. The computation starts by comparing the square of a constant with the input number. The constant is then manipulated based on the comparison. This comparison and manipulation goes on for four stages as the square root will be a four bit number. The final manipulated constant is the required square root.

Application-V set[4] [32] consists of a free fall detector shown in Figure C.11. The application takes as an input the three axes readings from an accelerometer. The application then uses a set of equations [35] to compute whether the acceleration readings point to a device in free fall. This reading is used to turn off a hard disk for protection.

Application-VI set consists of a robot path follower shown in Figure C.12. The application

[2]The 2 bit multiplier application was developed in [9].
[3]The Application-III set was developed in [32].
[4]The Application-V set was developed in [32].

takes inputs from two IR sensors located on the two wheels of a robot. Based on the readings from the two sensors, the application determines whether to turn the robot right, left, or continue in a forward direction [36].

5.1.2 Architectures

The eight base architectures used for the experiments are discussed in this section. These architectures are used as they cover all the modules required by the six set of applications described in Section 5.1.1. These set of architectures also cover all the variations needed to test the flexibility of the architecture. Table 5.9 lists the architectures used in this thesis, the modules used by them, and the main feature of the multibus architecture they test. Table 5.2 and Table 5.9 demonstrate the feasibility of running each application on each of the eight architectures. Each of the cells in the tables represent whether it is feasible to run the application in the corresponding column on the architecture in the corresponding row.

The power and area are obtained by running the automation framework shown in Figure 4.1. The architecture design files are given as input to an automated ASIC tool-flow. The ASIC simulation tool takes the synthesized, placed, and routed netlist as an input from the automated ASIC tool-flow and the configurations from the Test Bench Generation to simulate the execution of the application on the architecture. The output of this stage is sent to the Result Extraction block, which generates the power and area numbers. The standard cell library used for the simulation is the TSMC 45nm library. Section 5.3, and Section 5.4 use the power numbers obtained from this simulation directly. This library does not support a power aware design. Hence, Section 5.5 uses the power numbers obtained through a flash based design done in [32].

Architecture-I, Architecture-II, Architecture-III, Architecture-IV, Architecture-V, Architecture-

Table 5.2: Feasibility of Executing the Applications on the Architectures (Part-1)

Architecture	Thermostat	Two bit multiplier	Four bit multiplier	Six bit multiplier	Eight bit multiplier	Two bit multiply and sum
Architecture-I	yes	no	no	no	no	no
Architecture-II	no	yes	yes	yes	yes	no
Architecture-III	no	no	no	no	no	no
Architecture-IV	no	no	no	no	no	no
Architecture-V	no	no	no	no	no	no
Architecture-VI	no	no	no	no	no	no
Architecture-VII	no	no	no	no	no	no
Architecture-VIII	no	no	no	no	no	yes

VI, Architecture-VII, and Architecture-VIII are demonstrated in Figure D.1, Figure D.2, Figure D.3, Figure D.4, Figure D.6, Figure D.5, Figure D.7, and Figure D.8 respectively.

5.2 Scheduler Analysis

This section concentrates on the analysis of the efficiency of the scheduler developed in this thesis. It demonstrates the effectiveness of the current scheduler and analyzes the various factors effecting scheduler execution time.

The time the scheduler takes to find a valid schedule of an application depends on size (number of nodes) of the application graph, the schedule length, and the number and types of hardware modules used. This is analyzed in Table 5.4. The time and schedule shown in the table is for the first valid schedule that the scheduler finds. As seen in the table, the time to find a schedule generally increases with an increase in the size of the application. This is evident in Table 5.4, as the size of the multiplier application increases from a two bit to an eight bit multiplier, the scheduler execution time increases rapidly. This trend

Table 5.3: Feasibility of Executing the Applications on the Architectures (Part-2)

Architecture	Two coefficient filter	Three coefficient filter	Eight coefficient filter	Square root solution	Free fall detector	Robot path follower
Architecture-I	no	no	no	no	no	no
Architecture-II	no	no	no	no	no	no
Architecture-III	yes	yes	yes	no	no	no
Architecture-IV	no	no	no	yes	no	no
Architecture-V	no	no	no	no	yes	no
Architecture-VI	no	no	no	no	no	yes
Architecture-VII	yes	yes	yes	no	no	no
Architecture-VIII	no	no	no	no	no	no

is reversed when the scheduler execution time for an eight coefficient filter is compared to the scheduler execution time of a square root finder. The scheduler takes more time on the square root finder as compared to the eight coefficient filter, even though the eight coefficient filter has more nodes than the square root finder. This is because of the types of hardware modules used in the eight coefficient filter. The eight coefficient filter does not have any modules where the order in which data is available on the input edges matter. For example, the square root finder application uses a state machine as seen in Architecture-IV shown in Figure D.4. This state machine requires its input edges to be in order. The square root finder also uses a comparator module which also requires its input edges in order. These modules create an overhead in the scheduler in terms of number of times back tracking has to be performed. Hence, the scheduler execution times for applications using such modules increases.

The scheduler execution time also depends on the number of hardware modules used. This is because as module reuse increases the scheduler has to backtrack more because of increased resource usage conflicts. When additional hardware modules are used, resources are available more readily and hence there is no need for extensive backtracking. This can be seen in

Table 5.4. The two bit multiply and add has almost the same number of nodes as a four bit multiplier but the scheduler takes much lesser time for a two bit multiply and add as compared to a four bit multiplier. This is because the two bit multiply and add uses fourteen modules as compared to the four bit multiplier which uses eight modules.

The scheduler execution time also depends on the schedule length. It generally increases with an increase in schedule length. This is seen in Table 5.4, as the schedule length doubles from a two bit multiplier to a six bit multiplier and the scheduler execution time is nearly multiplied four times. This is because as size of applications increase due to the effect of increased resource sharing, backtracking increases drastically due to resource conflicts. This in turn increases the schedule length and scheduler execution time. However, in some cases the scheduler execution time decreases with an increase in schedule length. This is seen in Table 5.4 where the schedule length increases from a free fall detector to an eight coefficient filter but the schedule scheduler execution time decreases. This is because the other factors which effect scheduler execution time compensate for an increase in schedule length. Here, the schedule length decreases because of a decrease in the number of nodes in the application. However, the free fall detector uses Architecture-V shown in Figure C.11 which uses the subtracter and comparator module which requires their edges to be in order. This overshadows the effect of increase in number of nodes on scheduler execution time. Hence, scheduler execution time decreases.

The current scheduler finds the best schedule out of all possible legal schedules. As the size of the applications increase there are more choices available and hence, the gap between the worst and best schedule increases. This is evident in Table 5.5. The gap between the worst and best schedules increases from a two bit multiplier to an eight bit multiplier. In the same way the gap between the worst and best schedule increases from a two coefficient filter to an eight coefficient filter.

Table 5.4: Run Time vs Complexity of Application Graphs

Application	Number Of Nodes	Number Of Hardware Modules Used	Execution time (seconds)	Schedule length (cycles)
Thermostat	9	7	5.42	26
Two bit multiplier	17	8	13.63	48
Four bit multiplier	35	8	32.46	69
Six bit multiplier	53	8	58.45	96
Eight bit multiplier	71	8	80.78	130
Two bit multiply and sum	34	16	21.12	62
Two coefficient filter	11	8	5.60	28
Three coefficient filter	15	8	7.12	34
Eight coefficient filter	35	8	16.46	55
Square root calculator	28	8	18.15	57
Free fall detector	33	9	17.35	53
Robot path follower	23	8	15.13	52

The scheduler has also been improved to support multiple event-buses. As can be seen from the above analysis, all the six sets of applications can be scheduled using the current scheduler. With the hardware simulation obtained using Synopsys VCS, there are no bus or resource usage conflicts for all the six sets of applications running on all the eight base architectures. Moreover, the data transfers between modules match the framework-generated schedule. This validates the working of the scheduler.

5.3 Multiple Output Register Usage Analysis

This section demonstrates the advantages of using multiple output registers in a hardware module on the architecture. The use of multiple output registers is compared to a solution in which multiple output registers are not used but rather multiple hardware modules are

Table 5.5: Scheduler Optimization Results

Application	Worst Schedule (cycles)	Best Schedule (cycles)
Thermostat	27	26
Two bit multiplier	49	46
Four bit multiplier	73	65
Six bit multiplier	104	93
Eight bit multiplier	142	126
Two coefficient filter	29	28
Three coefficient filter	37	34
Eight coefficient filter	61	52
Square root calculator	59	54
Free fall detector	58	51
Robot path follower	55	49

used. The comparison is performed in terms of power usage, total area, and schedule length. Application-III set is used to obtain results in this section. Architecture-III shown in Figure D.3 is used to test the approach when multiple output registers are used and Architecture-VII shown in Figure D.7 is used for the approach when multiple hardware modules are used instead.

Table 5.6 shows that both the total area and power are better for the approach in which multiple output registers are used. The benefit on the power usage and area when multiple output registers are used increases with the size of application. This is evident as the difference total area when multiple output registers are used and when they are not used increases from 4654 μm^2 for a 3 coefficient filter to 13444 μm^2 for an eight coefficient filter. The power usage follows a similar trend and the difference increases from 220 μW for a 3 coefficient filter to 1156 μW for an eight coefficient filter.

On the other hand, the use of multiple output registers results in slightly longer schedules as shown in Table 5.6. This is because when multiple output registers are used the sharing

of the modules may result in delays. This is not necessary when multiple hardware modules are used. But the power lost due to increased schedule length is overshadowed by the savings from smaller hardware as shown in Table 5.6.

Table 5.6: Advantage of Using Multiple Output Registers

Application	Multiple Output Registers Used			Multiple Output Registers Not Used		
	Schedule Length (cycles)	Total Area (μm^2)	Total Power (μW)	Schedule Length (cycles)	Total Area (μm^2)	Total Power (μW)
Three coefficient filter	34	112308	10472	33	116962	10692
Five coefficient filter	45	146993	17730	44	156491	18568
Eight coefficient filter	52	209925	29160	50	223369	30316

The increase in the area and power with an increase in use of output registers is due to the increase in area of the wrapper configuration module. This trend is shown in Table B.1 in the Appendix. A detailed split up of the area details of an eight coefficient filter when multiple output registers are used and when multiple hardware modules are used is provided in Table B.2 in the Appendix.

As seen in the above analysis, it is more efficient and effective in terms of area and power usage to use multiple output registers than use multiple hardware modules. This advantage increases with an increase in size and complexity of applications.

5.4 Multiple Event-Bus Architecture Analysis

This section demonstrates the effectiveness of a multiple event-bus architecture compared to a single event-bus architecture based on the schedule length, energy usage, and area results. In table 5.7 the results are tabulated for a part of Application-II set (a two bit multiply and

sum) executed on Architecture-VIII. The 2 bit multiply and sum application is used to test the multiple event-bus architecture because for the multiple buses to be useful there should be a situation in which both the event-buses are used in parallel.

A multiple event-bus architecture also uses new modules to transfer data between buses. These new modules create an overhead in the schedule, energy usage, and area. This is seen in Table 5.7, when only one module uses bus two the modules used to transfer data between buses is reused eight times. This adds a schedule overhead and the schedule length increases to 66 from 62 when only a single bus was in use. The extensive reuse of the bus transfer module also adds to the total area and energy usage (due to increase in wrapper configuration block area). Thus, using the second bus proves inefficient in this case. As the reuse of the bus transfer module decreases the area, energy usage, and schedule length decrease (this happens as the output register of a module and the input register of the destination module are connected to the same bus, hence, the need for a bus transfer module decreases). This is seen in table, when the number of modules using bus two is two and reuse of the bus transfer modules drop to one, the schedule length, total area, and energy usage also decrease from the situation in which number of modules using bus two is three and the reuse of bus transfer modules is four.

In the multiple event-bus architecture there is a correspondence between number of modules using bus two and the schedule, energy usage, and total area. As seen in the table, as the number of modules using bus two increases from one to four, the schedule length drops from 66 to 58. The energy usage and area also decrease. But this is provided that the reuse of the bus transfer modules decrease (as more modules use bus two, there is lesser requirement for use of modules to transfer data between buses). Thus, modules that use bus two must be selected carefully so as to decrease the use of additional bus transfer modules. The ideal situation is when the maximum number of modules possible use bus two, and bus one and

Table 5.7: Energy Efficiency of a Multiple Event-Bus Architecture

Architecture	Schedule Length (cycles)	Number Of Times Bus Connector Module Reused	Number Of Modules Using Bus 2	Total Area (μm^2)	Energy Usage For A Schedule Period(nJ)
Single Event-Bus	62	-	-	229466	72
Multiple Event-Bus[5]	66	8	2	252271	84
	65	6	1	249585	82.1
	61	4	3	246735	76.4
	58	2	4	244407	71.6
	63	3	1	245482	78.3
	59	1	2	236939	70.8
	58	3	4	245482	72.2
	54	2	6	244407	66.7
	52	1	6	236939	62.4

bus two can be used in parallel. Also, ideally the minimum number of extra modules are used to transfer data between buses. This situation occurs when the schedule length on multi bus architecture is 52. In this situation the schedule decreases to its least value. The energy usage for a schedule period reduces significantly. As seen in Table 5.7, the energy usage for a multibus architecture is approximately 10nJ less than the single bus architecture.

As seen in the above analysis, the multiple event-bus architecture helps in reducing the schedule of an application drastically. Also, the energy usage in a schedule period for a multibus architecture is much less compared to a single bus architecture because of the drastic decrease in schedule length.

[5]The rows in the table corresponding to the multiple event-bus architecture use different versions of Architecture VIII. The different versions of Architecture VIII are obtained by using different combinations of modules connected to the second event-bus. The different combinations of module connections to the second event-bus are shown in Table B.3 of the Appendix.

5.5 Energy Usage Analysis

This section demonstrates that the designed multibus architecture is more energy efficient as compared to the PIC microcontroller [37]. The magnitude of the energy efficiency is analyzed using different applications. The energy usage of the multibus architecture is calculated as:

$$E = P * (1/f) * N \tag{5.1}$$

where, E is the energy usage for a single schedule period in Joules, P is the average power consumption for a schedule period in Watts, f is the processor clock frequency in Hertz, and N is the clock cycles taken for the execution of a single schedule period.

The energy usage of the processor architecture is compared with that of a general purpose PIC12F675 microcontroller. The implementation of the applications are carried out in MPLAB Integrated Development Environment using C and cross compiled to the PIC12F675 platform using the HI-TECH C compiler [38]. The implementation is simulated on PIC12F675 using the MPLAB SIM. Thus, the number of instruction cycles taken by each application is obtained in this way. The energy usage for PIC is estimated as:

$$Eusage = V * I * T * N \tag{5.2}$$

where, Eusage is the energy usage for the execution of the application in Joules, V is the typical value of the supply voltage in Volts, I is the typical value of the supply current in Amperes, T is the processor clock period in seconds, and N is the number of clock cycles for the execution of an application.

In Equation 5.2, N depends on the application. The other parameters are fixed for a particular processor architecture and are obtained from the data sheet. From [37], typical values

of V, I, and T for PIC12F675 are found to be 3 V, 190 μA, and 1 μs respectively. The PIC microcontoller uses a two stage instruction pipeline which allows all instructions to execute in a single cycle, except for program branches which require two cycles.

Table 5.8 shows the energy usage comparison between the multibus architecture and the PIC for different applications. From the energy usage values shown in Table 5.8, it is evident that the designed dataflow processor architecture is more energy efficient as compared to the PIC microcontroller. The magnitude of the energy efficiency depends on the size, type, and module reuse of the application executing on the architectures. Thus, for the small 2 bit multiplier, the multibus architecture is approximately 26 times more energy efficient as compared to the PIC microcontroller. For a large application like an 8 bit multiplier, the multibus architecture is approximately 3 times more efficient as compared to the PIC microcontroller. This is because, as the size of the applications increase, modules have to be reused more heavily on the multibus architecture. The use of multiple output registers combined with module reuse adds a huge overhead to the wrapper configuration block. This limits the energy efficiency that can be achieved using the multibus architecture for large applications.

The filter applications and the free fall detector application below show a drastic difference in energy usage between the dataflow architecture and the PIC. For the eight coefficient filter, the multibus architecture is approximately 70 times more efficient than the PIC and for the free fall detector the multibus architecture is approximately 50 times more efficient than the PIC. This is because, the PIC does not have a multiply instruction in its instruction set. It has to implement multiply using shift and add. Thus, the PIC takes a number of instruction cycles to execute the filter application. However, the multiplier module is a part of Architecture-III.

However, this energy usage comparison between the multibus architecture, and the PIC

microcontroller is not completely fair. The PIC12F675 is an eight bit microcontroller. The features of this microcontroller are very similar to the Atmel AT89C51RD2 microcontroller. From [39], the total die area of the AT89C51RD2 using 0.35 μm technology is found out to be 18 mm^2. The total area of the multibus architecture as seen in Table 5.7 is approximately 0.25 mm^2. The area of the AT89C2051 and by relation the area of the PIC12F675 is approximately 70 times the area of the multibus architecture. Thus, the PIC is bound to consume more energy.

Table 5.8: Energy Efficiency Comparison of Data Flow Architecture and PIC

Application	Multibus Architecture			PIC Microcontroller		
	Execution Cycles	Execution Time (μs)	Energy Usage (nJ)	Execution Cycles	Execution Time (μs)	Energy Usage (nJ)
Two bit multiplier	46	92	8.7	392	392	223.4
Four bit multiplier	65	130	14.87	448	448	255.4
Six bit multiplier	93	186	47.06	504	504	287.2
Eight bit multiplier	126	252	99.74	564	564	321.4
Two coefficient filter	28	56	3.8	532	532	303.2
Three coefficient filter	34	68	5.16	620	620	353.4
Eight coefficient filter	52	104	11.42	1252	1252	713.6
Square root finder	54	108	12.44	460	460	262.2
Free fall detector	51	102	11.26	960	960	547.2
Robot path follower	49	98	10.09	432	432	246.2

Thus, the energy efficiency of the multibus architecture as compared to the PIC depends on the type and size of applications executing on it. For large applications, having large resource reuse and using multiple output registers the energy efficiency is very small. For applications without extensive reuse and applications using specific modules such as a multiplier, the energy efficiency of the multibus architecture compared to the PIC is significant.

Table 5.9: Characteristics of the Architectures

Architecture	Number of Modules	Types of Modules	Features
Architecture-I	7	Timer, ADC, ALU, Subtracter, Replicator, State Machine, FIFO	Used to execute simple applications.
Architecture-II	8	Timer, ADC, Replicator, State Machine, FIFO, Shifter, Adder, Constant Generator	Demonstrates capability to execute large applications.
Architecture-III	8	Timer, ADC, Replicator, Multiplier, FIFO, Delay Unit, Adder, Constant Generator	Demonstrates capability of architecture to handle modules with more than two outputs.
Architecture-IV	8	Timer, ADC, Replicator, Comparator, FIFO, State Machine, Multiplier, Constant Generator	Demonstrates capability of packet structure to handle additional internal configuration bits.
Architecture-V	9	Timer, ADC, Replicator, Comparator, FIFO, Subtracter, Multiplier, Constant Generator, Adder	Demonstrates capability of the architecture to execute mathematical formulae.
Architecture-VI	8	Timer, ADC, Replicator, Comparator, FIFO, State Machine, Delay Unit, Constant Generator	Demonstrates the use of the delay unit.
Architecture-VII	14	Timer, ADC, Replicator, Multiplier, FIFO, Delay Unit, Adder, Constant Generator, six replications of multiplier unit	Demonstrates capability of architecture to handle more than eight modules.
Architecture-VIII	16	Timer, ADC, Replicator, State Machine, FIFO, Shifter, Adder, Constant Generator, two Bus Transfer modules, replications of ADC, Replicator, State Machine, Shifter, Adder, and Constant Generator	Demonstrates the multibus architecture.

Chapter 6

Conclusion

This thesis demonstrates the design of a multibus dataflow architecture. This architecture was used to successfully execute a range of applications. The advantages of using multiple output registers compared to using multiple hardware modules in terms of area and power was demonstrated. The energy efficiency of a multiple event-bus architecture compared to a single event-bus architecture was also demonstrated. The designed architecture was shown to be more energy efficient than a PIC microcontroller. The work done in this thesis developed an object oriented, stack based scheduler capable of finding a valid schedule for a range of applications.

The work done in this thesis combined with the work done in [8] and [9] yields a multibus, dataflow, non-ISA processor architecture targeted towards event driven applications and capable of inherently managing concurrency. A robust automation framework capable of generation of configurations and automatic validation of a range of applications was also developed to reduce the design complexity.

6.1 Future Work

The architecture developed in this thesis has a single input FIFO and a single output FIFO for communication with the network. The architecture can be expanded to encompass multiple FIFOs.

In the multibus architecture, the modules used to transfer data between the modules use a wrapper configuration block. Thus, the area and power usage of these modules increase with reuse. This can be avoided by designing the bus transfer modules without a wrapper configuration block. In this architecture, the bus transfer modules would be responsible to keep a track of the modules that require their use. The bus transfer modules would transfer a data packet from one bus to another based on the information stored in them. This can be explored in the future.

Bibliography

[1] Marculescu, D. and Marculescu, R. and Zamora, N.H. and Stanley-Marbell, P. and Khosla, P.K. and Park, S. and Jayaraman, S. and Jung, S. and Lauterbach, C. and Weber, W. and Kirstein, T. and Cottet, D. and Grzyb, J. and Troster, G. and Jones, M. and Martin, T. and Nakad, Z., "Electronic textiles: A platform for pervasive computing," in *Proceedings of the IEEE*, vol. 91, pp. 1995–2018, December 2003.

[2] Edmison, Josh and Jones, Mark and Lockhart, Thurmon and Martin, Thomas, *Wearable eHealth Systems for Personalised Health Management: State of the Art and Future Challenges*, ch. An e-Textile System for Motion Analysis, pp. 292 – 301. IOS Press, 2005.

[3] Kao, Jung-Chun and Marculescu, Radu, "Energy-Aware Routing for E-Textile Applications," in *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition*, pp. 1530 – 1591, 2005.

[4] Microchip Technologies. 8 bit PIC Microcontrollers. `http://www.microchip.com/en_us/family/8bit`.

[5] Zeh, Christopher, "Softwear: A Flexible Design Framework for Electronic Textile Systems," Master's thesis, Virginia Polytechnic Institute and State University, 2006. April.

[6] Petitpierre, C., "Feedback Scheduling: An Event-Driven Paradigm," *ACM SIGPLAN Notices*, vol. 42, pp. 7 – 14, December 2007.

[7] Shen, John Paul, *Modern processor design : fundamentals of superscalar processors.* McGraw-Hill Higher Education, 2005.

[8] Narayanswamy, Ramya, "Design of a Power-aware Dataflow Processor Architecture," Master's thesis, Virginia Polytechnic Institute and State University, July 2010.

[9] Lakshmanan, Karthick, "Design of an Automation Framework for a Novel Data-Flow Processor Architecture," Master's thesis, Virginia Polytechnic Institute and State University, 2010. July.

[10] Park, Sungmee and Mackenzie, Kenneth and Jayaraman, Sundaresan, "The wearable motherboard: a framework for personalized mobile information processing (PMIP)," in *Proceedings of the 39th annual Design Automation Conference*, (New York, NY, USA), pp. 170 – 174, ACM, 2002.

[11] Yuan-ting Zhang and Poon, C.C.Y. and Chun-hung Chan and Tsang, M.W.W. and Kin-fai Wu, "A Health-Shirt using e-Textile Materials for the Continuous and Cuffless Monitoring of Arterial Blood Pressure ," in *3rd IEEE/EMBS International Summer School on Medical Devices and Biosensors, 2006*, (Cambridge, MA), pp. 86 – 89, September 2006.

[12] Kennedy, T.F. and Fink, P.W. and Chu, A.W. and Champagne, N.J. and Lin, G.Y. and Khayat, M.A., "Body-Worn E-Textile Antennas: The Good, the Low-Mass, and the Conformal," *IEEE Transactions on Antennas and Propagation*, vol. 57, pp. 455 – 462, April 2009.

[13] De jonckheere, J. and Jeanne, M. and Grillet, A. and Weber, S. and Chaud, P. and Logier, R. and Weber, J., "OFSETH: Optical Fibre Embedded into technical Textile for Healthcare, an efficient way to monitor patient under magnetic resonance imaging ," *29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, 2007. EMBS 2007.*, pp. 3950 – 3953, October 2007.

[14] M. Di Rienzo and F. Rizzo and G. Parati and G. Brambilla and M. Ferratini and P. Castiglioni, "MagIC System: a New Textile-Based Wearable Device for Biological Signal Monitoring. Applicability in Daily Life and Clinical Setting ," in *27th Annual International Conference of the Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005.*, (Shanghai), pp. 7167 – 7169, April 2005.

[15] Nakad, Z. S., *Architectures for e-textiles.* PhD thesis, Virginia Polytechnic Institute and State University, 2003.

[16] Jones, M. and Martin, T. and Nakad, Z., "A service backplane for e-textile applications," *Workshop on Modeling, Analysis, and Middleware Support for Electronic Textiles 2002*, pp. 15 – 22, October 2002.

[17] Edmison, J. and Lehn, D. and Jones, M. and Martin, T., "E-textile based automatic activity diary for medical annotation and analysis," in *Wearable and Implantable Body Sensor Networks, 2006. BSN 2006. International Workshop on*, pp. 4 – 134, April 2006.

[18] Jones, Mark and Martin, Thomas and Sawyer, Braden, "An Architecture for Electronic Textiles," in *Proceedings of the ICST 3rd international conference on Body area networks, BodyNets 08, ICST, Brussels, Belgium: ICST*, pp. 1 – 4, 2008.

[19] Gay, David and Levis, Phil and Culler, David, "Software design patterns for TinyOS," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers,*

and tools for embedded systems, LCTES '05, (New York, NY, USA), pp. 40 – 49, ACM, 2005.

[20] Dunkels, A. and Gronvall, B. and Voigt, T., "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors," in *29th Annual IEEE International Conference on Local Computer Networks, 2004.*, pp. 455 – 462, December 2004.

[21] Viswanath, V., "Multi-log Processor Towards Scalable Event-Driven Multiprocessors," in *Proceedings of the EUROMICRO Systems on Digital System Design (DSD04)*, pp. 279 – 286, September 2004.

[22] Hempstead, Mark and Tripathi, Nikhil and Mauro, Patrick and Wei, Gu-Yeon and Brooks, David, "An Ultra Low Power System Architecture for Sensor Network Applications," in *The 32nd Annual International Symposium on Computer Architecture, 2005*, (Madison, WI, USA), pp. 208 – 219, May 2005.

[23] Ekanayake, V.N. and Kelly, C., IV and Manohar, R., "BitSNAP: Dynamic Significance Compression for a Low-Energy Sensor Network Asynchronous Processor," in *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC 05)*, (Washington, DC, USA: IEEE Computer Society), pp. 144 – 154, March 2005.

[24] Arvind and Culler, D. E., "Dataflow Architectures," *Annual review of computer science*, vol. 1, pp. 225 – 253, 1986.

[25] Dennis, Jack B. and Misunas, David P., "A preliminary architecture for a basic dataflow processor," in *Proceedings of the 2nd annual symposium on Computer architecture, ISCA '75*, (New York, NY, USA), pp. 126 – 132, December 1975.

[26] Swanson, Steven and Schwerin, Andrew and Mercaldi, Martha and Petersen, Andrew and Putnam, Andrew and Michelson, Ken and Oskin, Mark and Eggers, Susan J., "The WaveScalar Architecture," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, pp. 1 – 54, May 2007.

[27] Adam, Thomas L. and Chandy, K. M. and Dickson, J. R., "A Comparison of List Schedules for Parallel Processing Systems," *Commun. ACM*, vol. 17, pp. 685 – 690, December 1974.

[28] Da Li and Yibin Hou and Zhangqin Huang and Chunhua Xiao, "A Framework of Multi-Characteristics Fuzzy Dynamic Scheduling for Parallel Video Processing on MPSoC Architecture," in *IEEE International Conference on Fuzzy Systems (FUZZ), 2011*, pp. 627 – 634, June 2011.

[29] Ramamritham, K. and Stankovic, J.A. and Shiah, P.F., "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 184 – 194, April 1990.

[30] Yang Xu and Tang Zhizhong and Guo Deyuan and He Hu, "Backtracking Optimized DDG Directed Scheduling Algorithm for Clustered VLIW Architectures," in *International Conference on Future Computer Sciences and Application (ICFCSA), 2011*, pp. 82 – 85, June 2011.

[31] Jiaqiong Chen and Ronald G. Askin, "Project selection, scheduling and resource allocation with time dependent returns," *European Journal of Operational Research*, vol. 193, no. 1, pp. 23 – 34, 2009.

[32] Mandlekar, Anup, "An Application Framework for a Power Aware Processor Architecture," 2012. Private Communication.

[33] Richard G. Lyons, *Understanding Digital Signal Processing, Second Edition.* Prentice Hall PIR, 2004.

[34] Microchip Technologies, "Microchip Application Notes." `ww1.microchip.com/downloads/en/AppNotes/91040a.pdf`, 2000.

[35] "Accelerometer and Gyro Tutorial." `http://www.instructables.com/id/Accelerometer-Gyro-Tutorial/`.

[36] "Line Following Robot." `www.kmitl.ac.th/~kswichit/ROBOT/Follower.pdf`.

[37] Microchip Technologies, "PIC12F675 Data Sheet." `http://ww1.microchip.com/downloads/en/DeviceDoc/41190G.pdf`.

[38] Microchip Technologies, "HI-TECH C for PIC, Users Guide." `ww1.microchip.com/downloads/en/devicedoc/htc_pic_manual.pdf`, 2010.

[39] Atmel Corporation, "AT89C51RD2 Data Sheet." `www.atmel.com/Images/doc4235.pdf`.

Appendices

Appendix A

Multibus Architecture Details

A.1 Packet Structure Details

A3	A2	A1	A0	C/D'	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

Figure A.1: Packet Structure for Data

A3	A2	A1	A0	C/D'	WR/INT'	CA2	CA1	CA0	CD6	CD5	CD4	CD3	CD2	CD1	CD0

Figure A.2: Packet Structure for Configuration

The packet structure shown in Section A.1 of Appendix uses 4 module address bits, 3 bits to address internal configuration registers, 11 bits for data, and 7 bits for configuration. This packet configuration allows 16 hardware modules on the architecture and 5 output registers for each module. The details of the packet structure are as below:

$A_{3..0}$

The bits represent the address of the destination functional unit for the event.

C/D'

A value of one denotes a configuration packet, and zero denotes a data packet

W R/INT'

A value of one denotes a configuration for the wrapper, and zero denotes a configuration for the internal module

$CA_{2..0}$

The bits represent the address of wrapper configuration registers, to which configuration data is addressed. This is explained in detail in section 3.2.2.1

$CD_{6..0}$

The bits represent the configuration in a configuration event

$D_{10..0}$

The bits represent the data in a data event

A.2 Input Register Module

Figure A.3 shows the top-level design of an input register module. The input register can be connected to any of the buses through the input_bus_lines. The input register is responsible for creating the 16 bit packets shown in figure A.1 and A.2. The input interfaces of all the modules read the address bits, and only the module whose MOD_ID matches the address bits in the event, creates the packet. The start of packet transmission is identified by the bus busy signal from the corresponding bus. On receipt of the packet, the input register identifies the type of the packet and asserts the data_pkt_ready or int_cfg_pkt_rdy or wr_cfg_pkt_rdy signal based on the type of packet. These signals are reset based on the

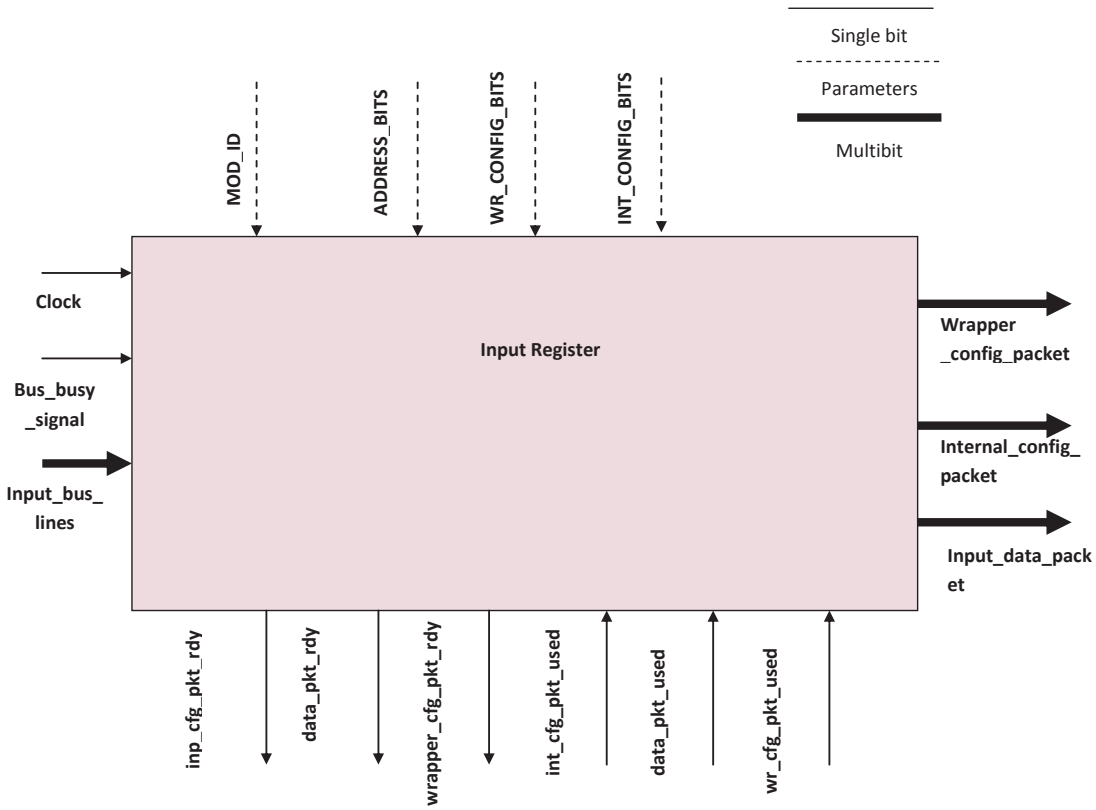

Figure A.3: A High Level Representation of the Input Register Module

reception of acknowledgment from the corresponding module, indicating that the packet has been read. This acknowledgment is in the form of signals data_pkt_used, wr_cfg_pkt _used and int_cfg_pkt_used.

Appendix B

Result Section Data

B.1 Multiple Output Register Usage Analysis

Consider table B.1, as is evident in the table as multiplier reuse and the number of output registers used increases the multiplier wrapper configuration block area increases drastically. Thus for a 2 coefficient filter in which the mutiplier reuse is two and only one output register is used, the total wrapper configuration area is small at 2517 μm^2. When multiplier module reuse increases to eight and the number of output registers used increases to seven the wrapper configuration area drastically increases to 30262 μm^2. This is because the use of multiple output registers adds an overhead in terms of extra registers, to the wrapper configuration block(as additional registers need to be defined to store the delay and destination address values of the target instance of the module, for each output register). The size of these registers also increase with increase in reuse of a module (as the delay and destination address values of each instance of a hardware needs to be stored in the wrapper configuration block). These two effects combine to inflate the size of the wrapper configuration block in applications such as an eight coefficient filter.

Table B.1: Effect of Multiple Output Registers on Wrapper Area

Application	Number Of times Multiplier reused	Number Of Output Registers Used By Multiplier	Wrapper Configuration Block Area (μm^2)
Two coefficient filter	2	1	2517
Three coefficient filter	3	2	3639
Five coefficient filter	5	4	11046
Eight coefficient filter	8	7	30262

Table B.2 shows the detailed analysis of area of the multiplier module of a part of an eight coefficient FIR filter application. As seen in Table B.1 the wrapper configuration block area is the major contributor to the total area of the multiplier. This is the only part of the module whose area and power consumption increases with increase in reuse. The area and power usage of the internal module and output registers also increase with an increase in the number of output registers. But the addition of additional hardware increases the total area as all the parts of the module including the input register, wrapper configuration register, the internal module, wrapper control module and the output registers are replicated again. This total increase when multiple hardware modules are used overshadows the large increase in wrapper area when multiple output registers are used.

B.2 Multibus Architecture Energy Efficiency Analysis

Table B.2: Eight Coefficient Filter Area Details

	Multiple Output Registers Used	Multiple Output Registers Not Used
Number of times multiplier is reused	8	2
Additional hardware modules to replicate multiplier	0	6
Number of output registers used by multiplier	7	1
Multiplier wrapper configuration block area(μm^2)	30262	2517
Total area of output registers(μm^2)	16233	2202
Total internal module area(μm^2)	10135	2275
Total area of multiplier(μm^2)	60478	10737
Total area of additional hardware modules used to replicate multiplier(μm^2)	0	64422

Table B.3: Combinations of Module Connections to the Second Event-Bus

Schedule Length	Modules Connected to second Event-Bus	Number Of Bus Connector Modules used
66	Input register of Repcp, and output register of Statecp	8
65	Input register 1 of Shiftcp, and output register of Shiftcp	6
61	Output register of Constcp, input register 1 of Shiftcp, input register 1 of Addercp, and output register of Addercp	4
58	Input register of Adccp, input register 1 of Shiftcp, output register of Constcp, and output register of Statecp	2
63	Input register 1 of Addercp, input register 2 of Addercp, and output register of Addercp	3
59	Input register 1 of Shiftcp, and output register of Statecp	1
58	Output register of Constcp, input register 1 of Shiftcp, input register 1 and 2 of Addercp, output register of Addercp, and output register of Statecp	3
54	Output register of Constcp, input and output registers of Shiftcp, input registers of Addercp, input and output registers of Statecp, input and output registers of ADCcp, and input and output registers of Repcp	2
52	Input and Output registers of Constcp, input and output registers of Shiftcp, input registers of Addercp, input and output registers of Statecp, input and output registers of ADCcp, and input and output registers of Repcp	1

Appendix C

Application Graphs

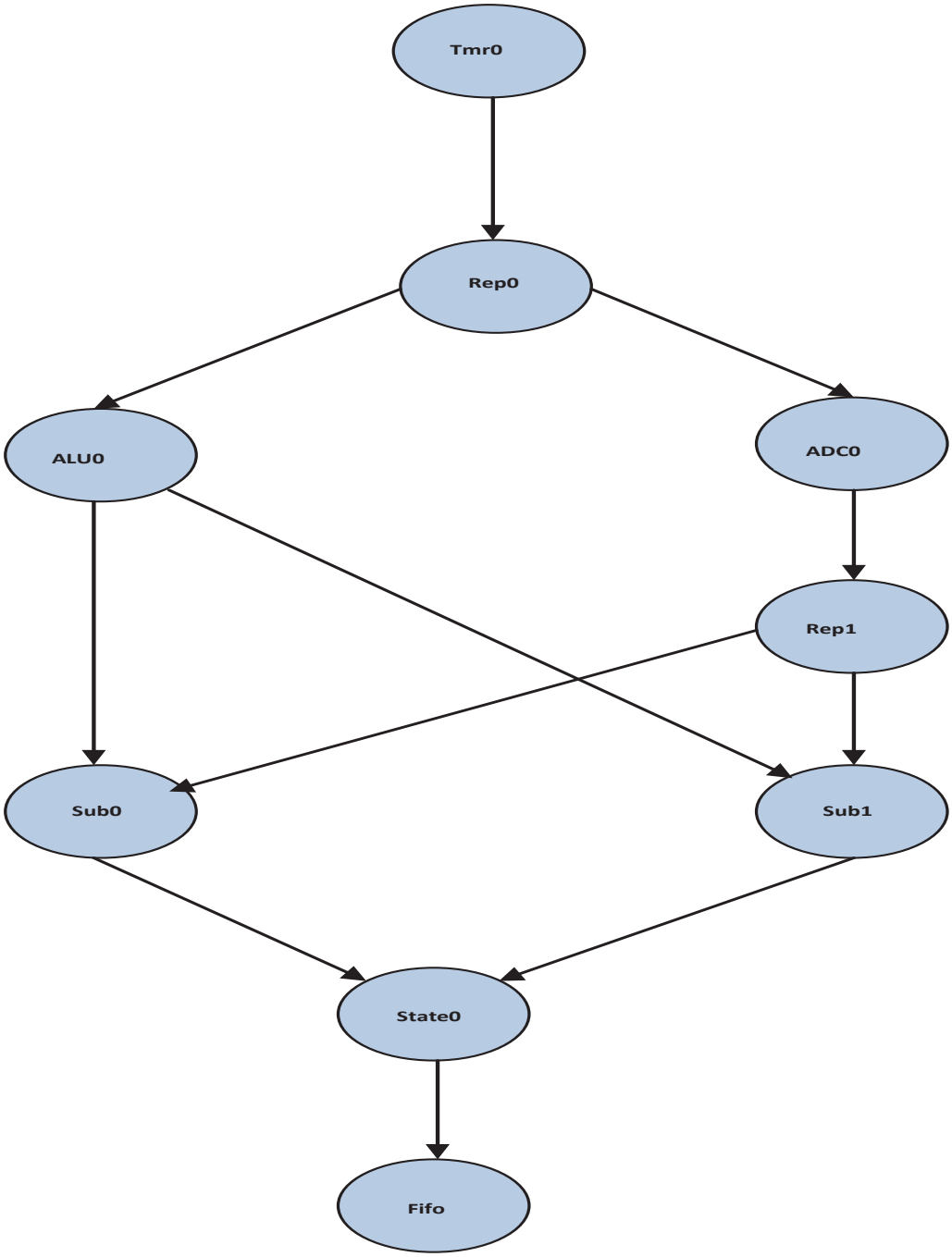

Figure C.1: Application Graph for Thermostat

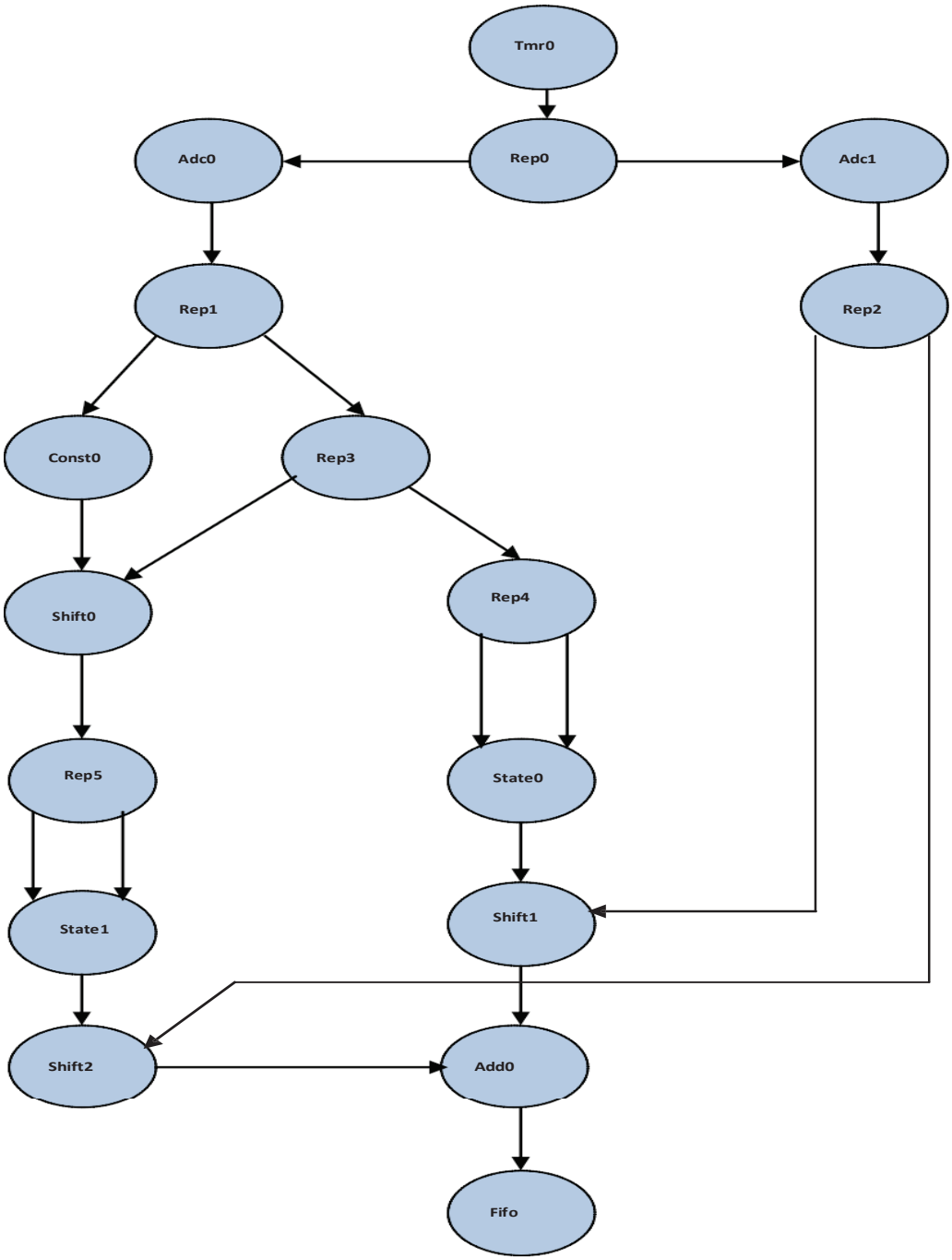

Figure C.2: Application Graph for 2 Bit Multiplier

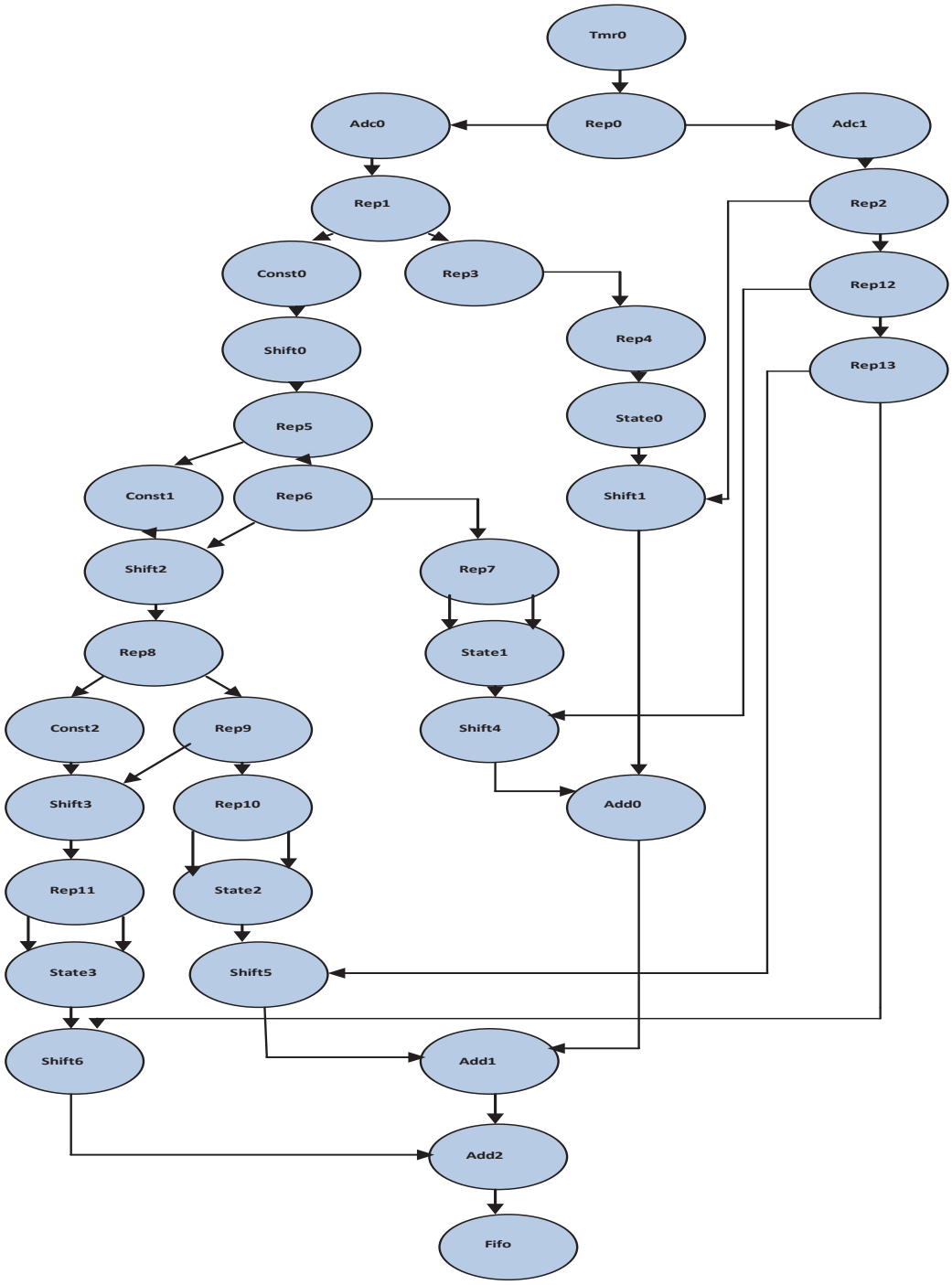

Figure C.3: Application Graph for 4 Bit Multiplier

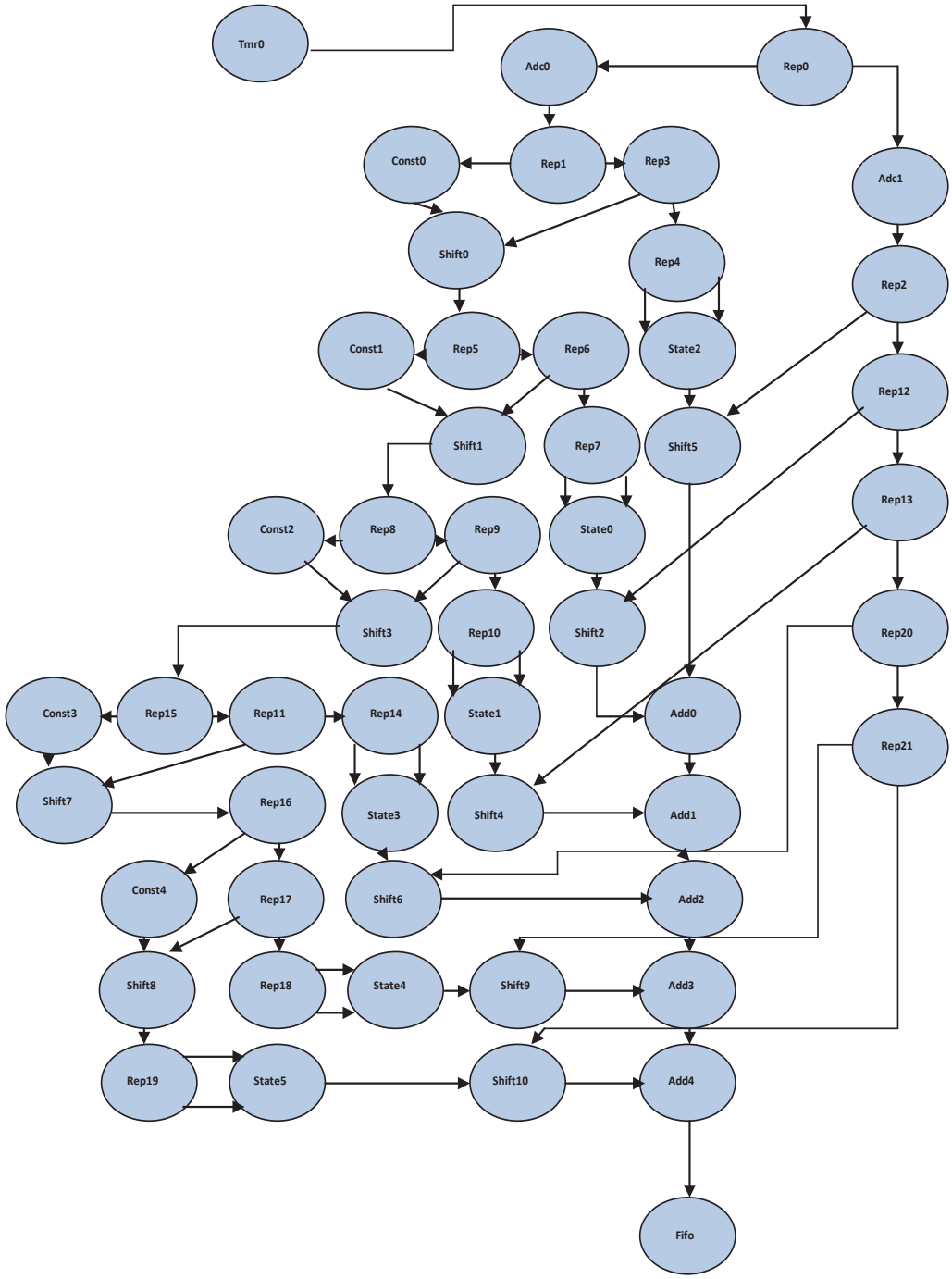

Figure C.4: Application Graph for 6 Bit Multiplier

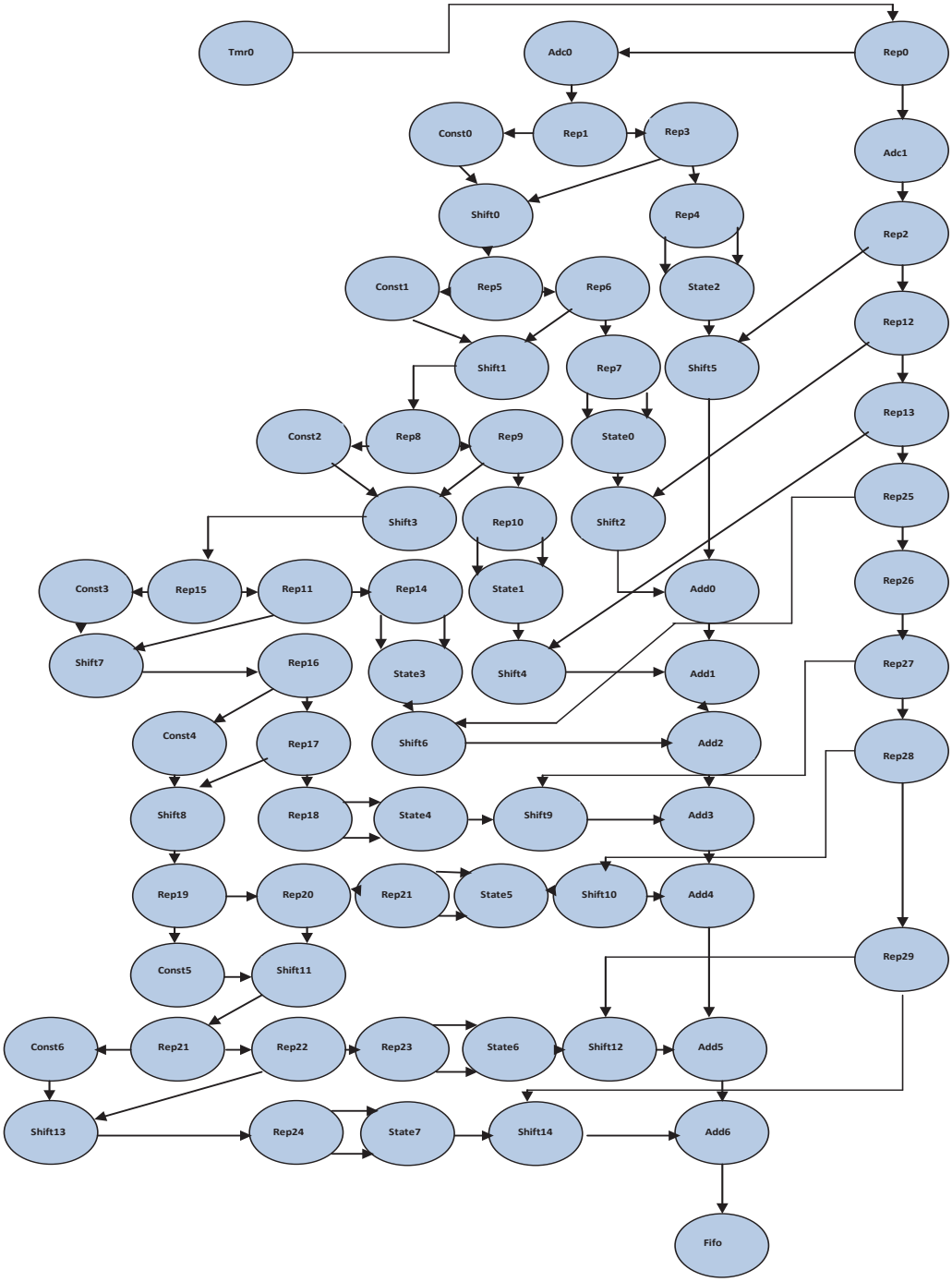

Figure C.5: Application Graph for 8 Bit Multiplier

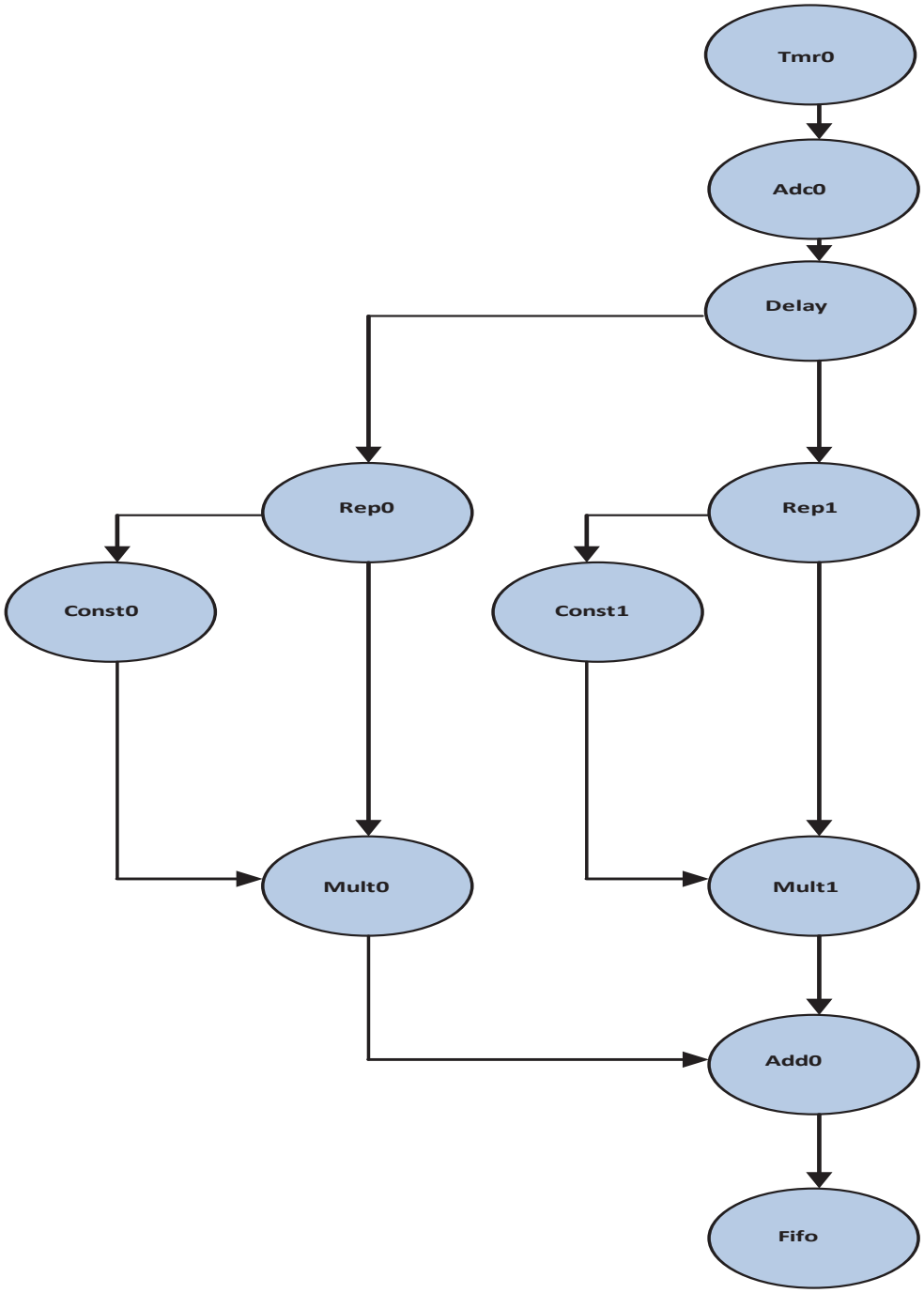

Figure C.6: Application Graph for 2 Coefficient Filter

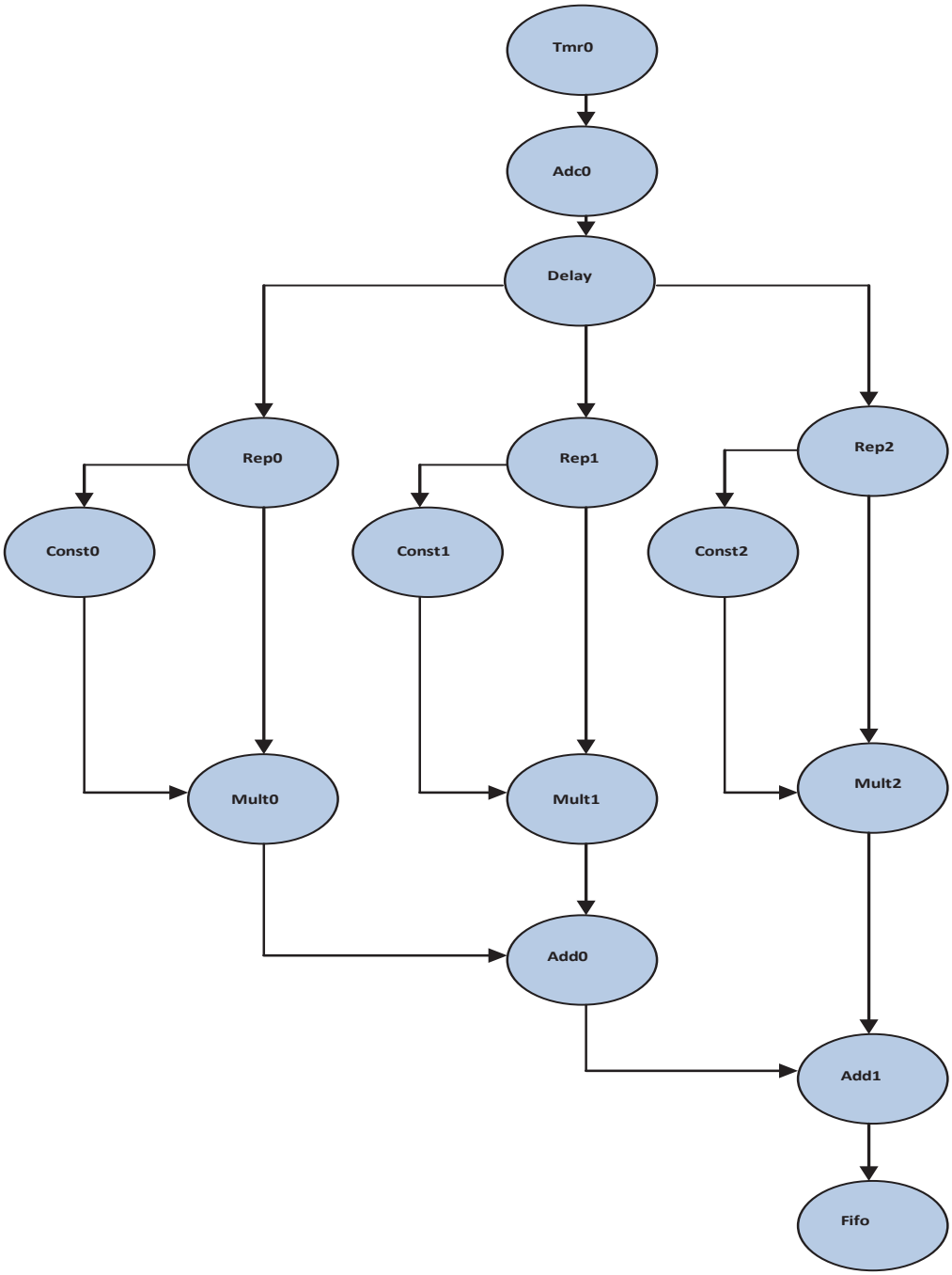

Figure C.7: Application Graph for 3 Coefficient Filter

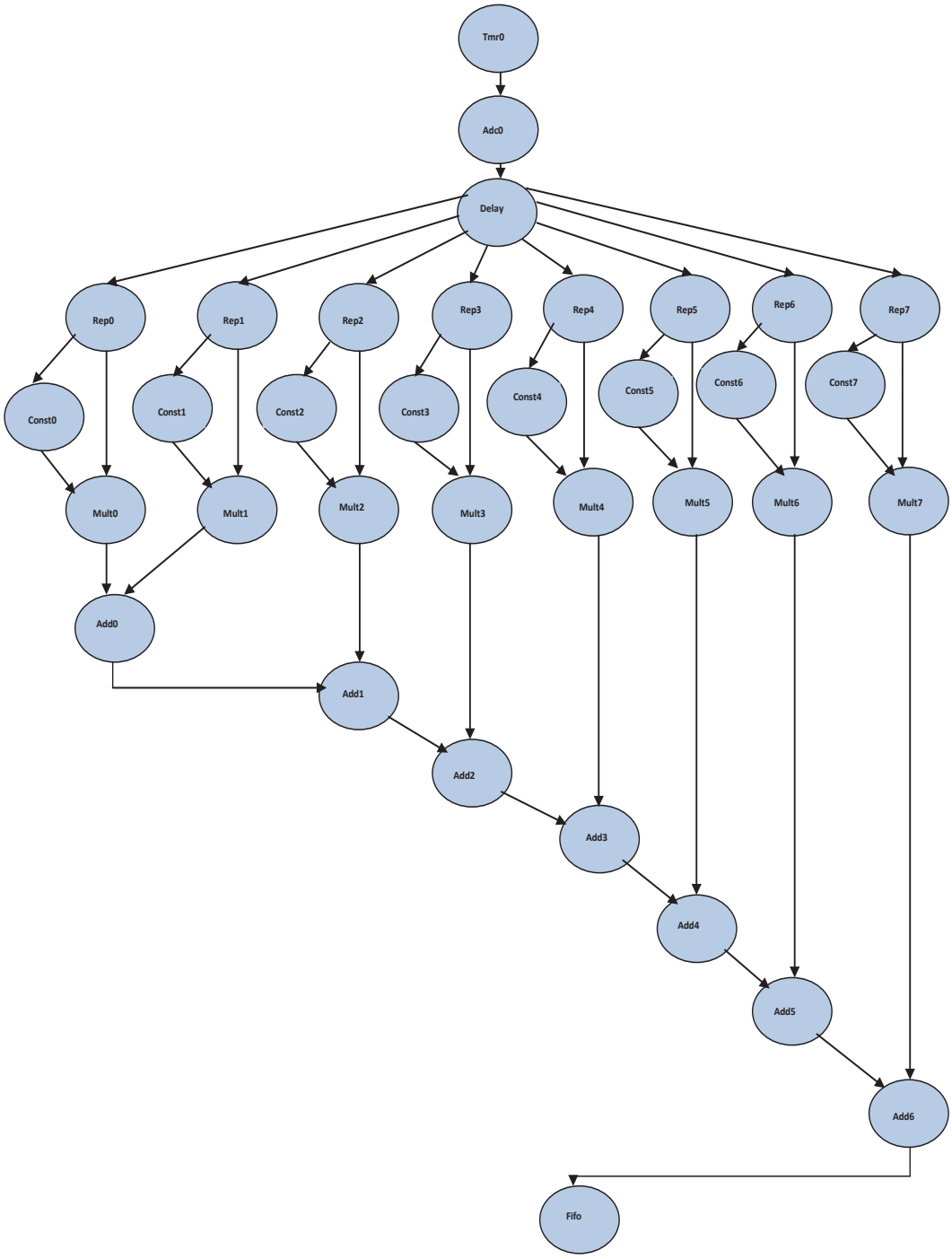

Figure C.8: Application Graph for 8 Coefficient Filter

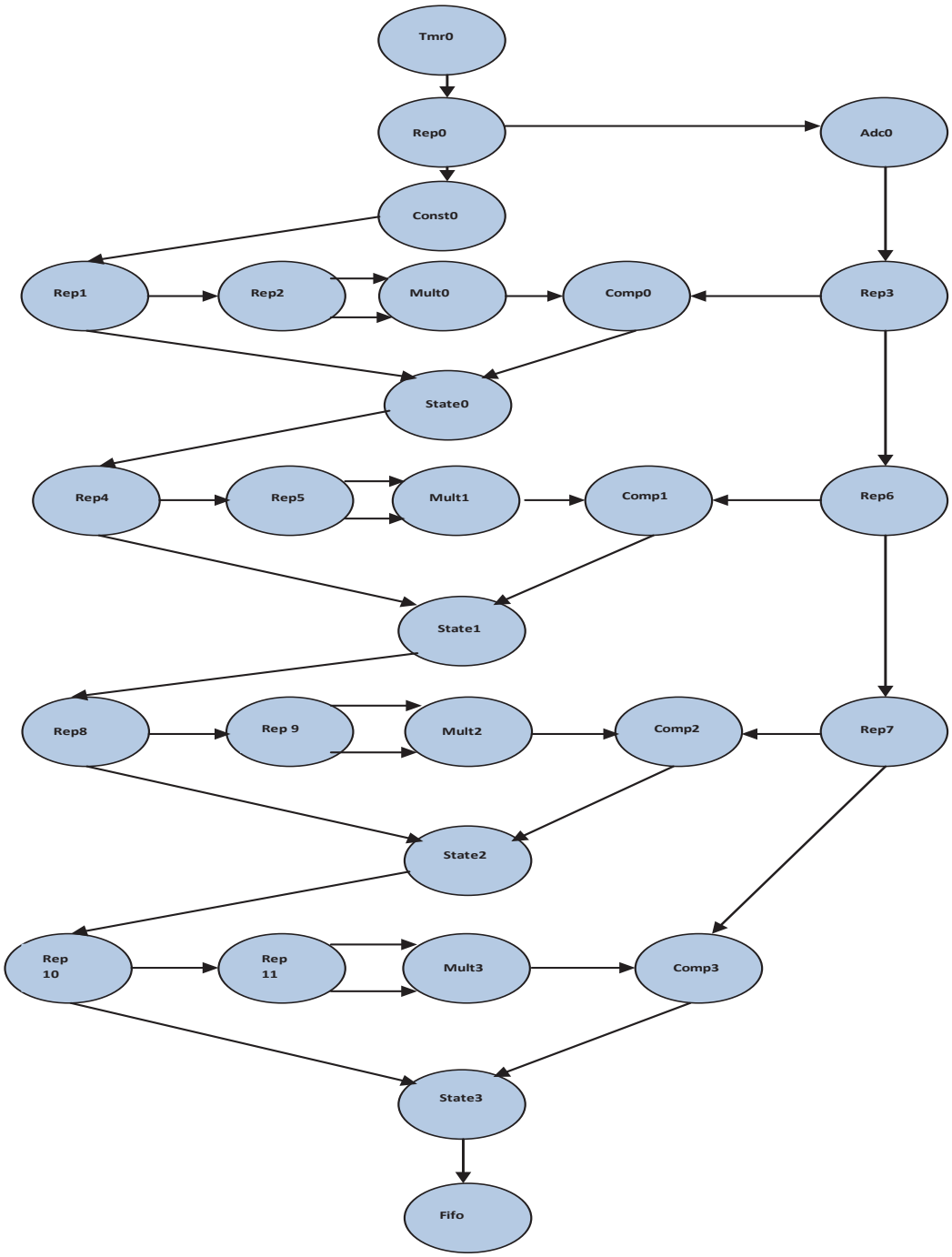

Figure C.9: Application Graph for Square Root Finder

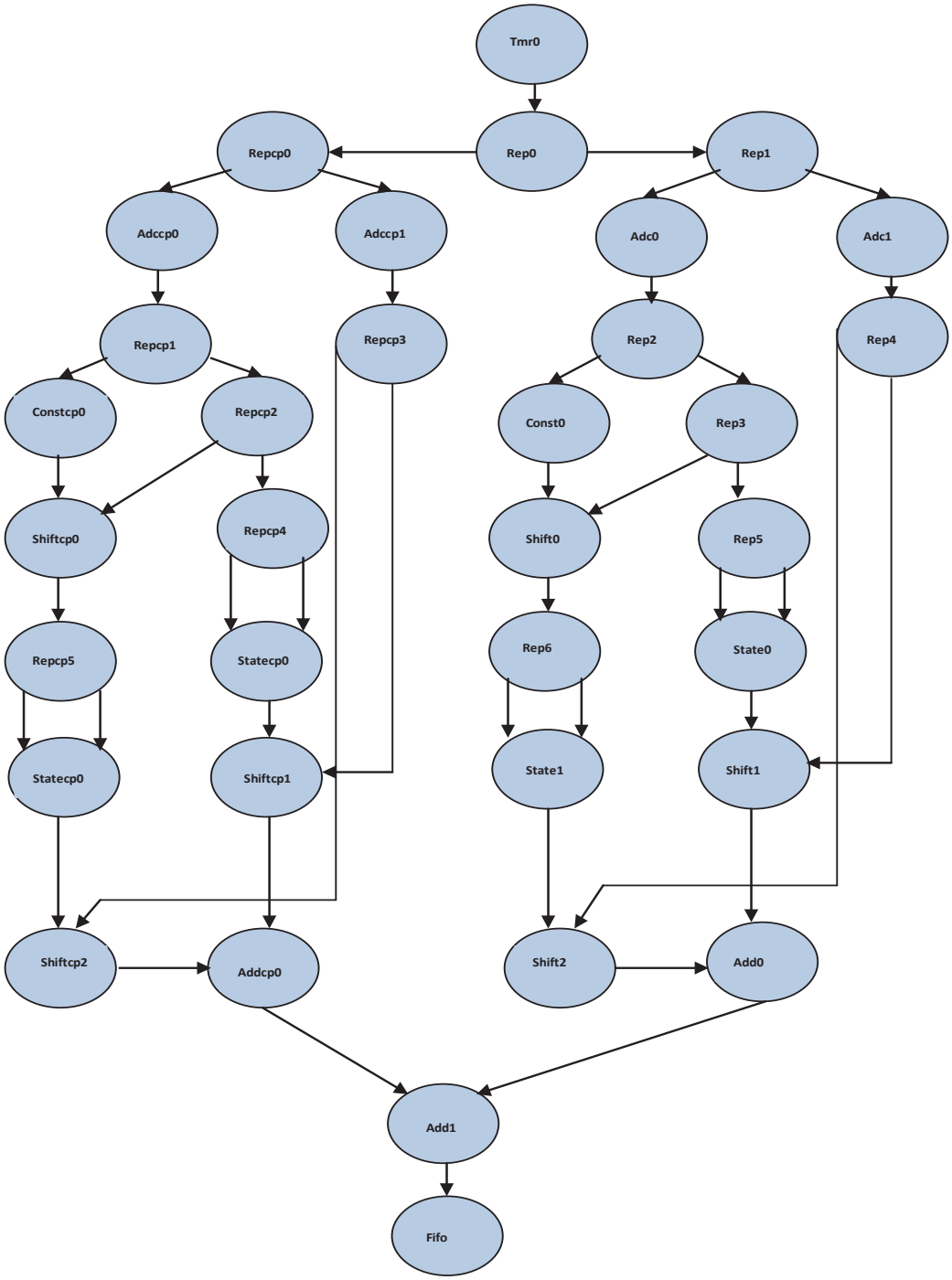

Figure C.10: Application Graph for Two Bit Multiply and Sum

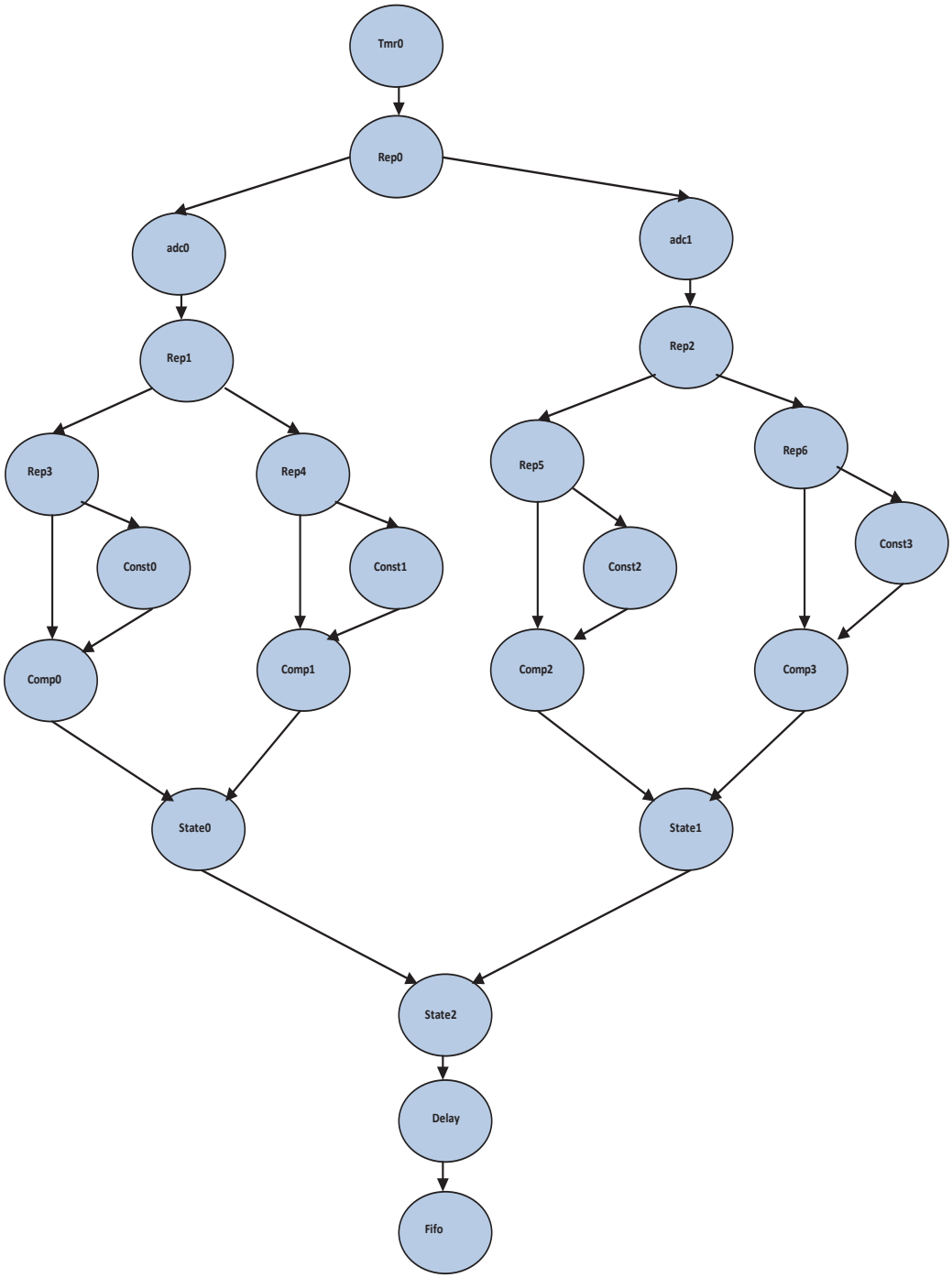

Figure C.11: Application Graph for Robot Path Follower

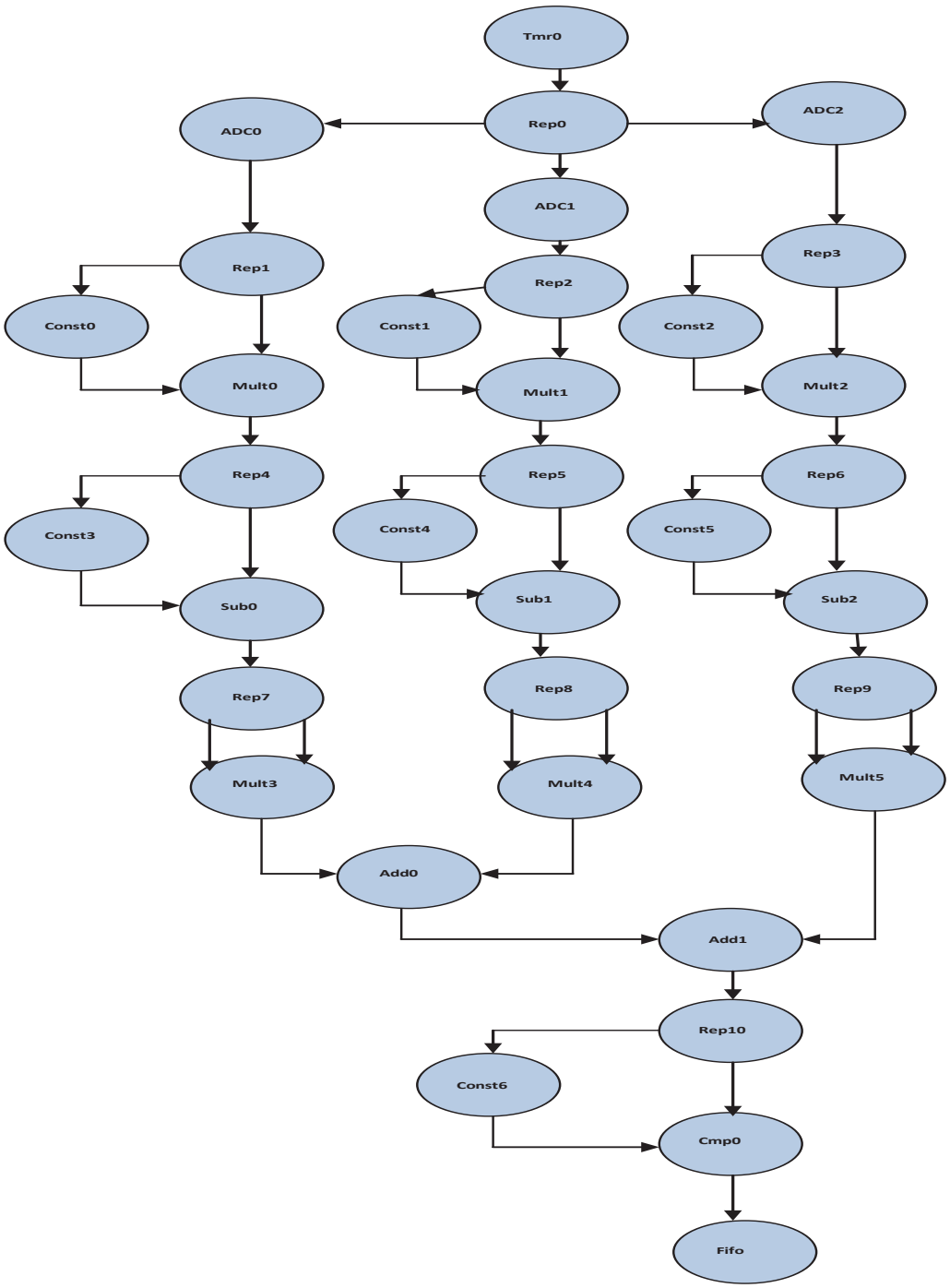

Figure C.12: Application Graph for Free Fall Detector

Appendix D

Architectures

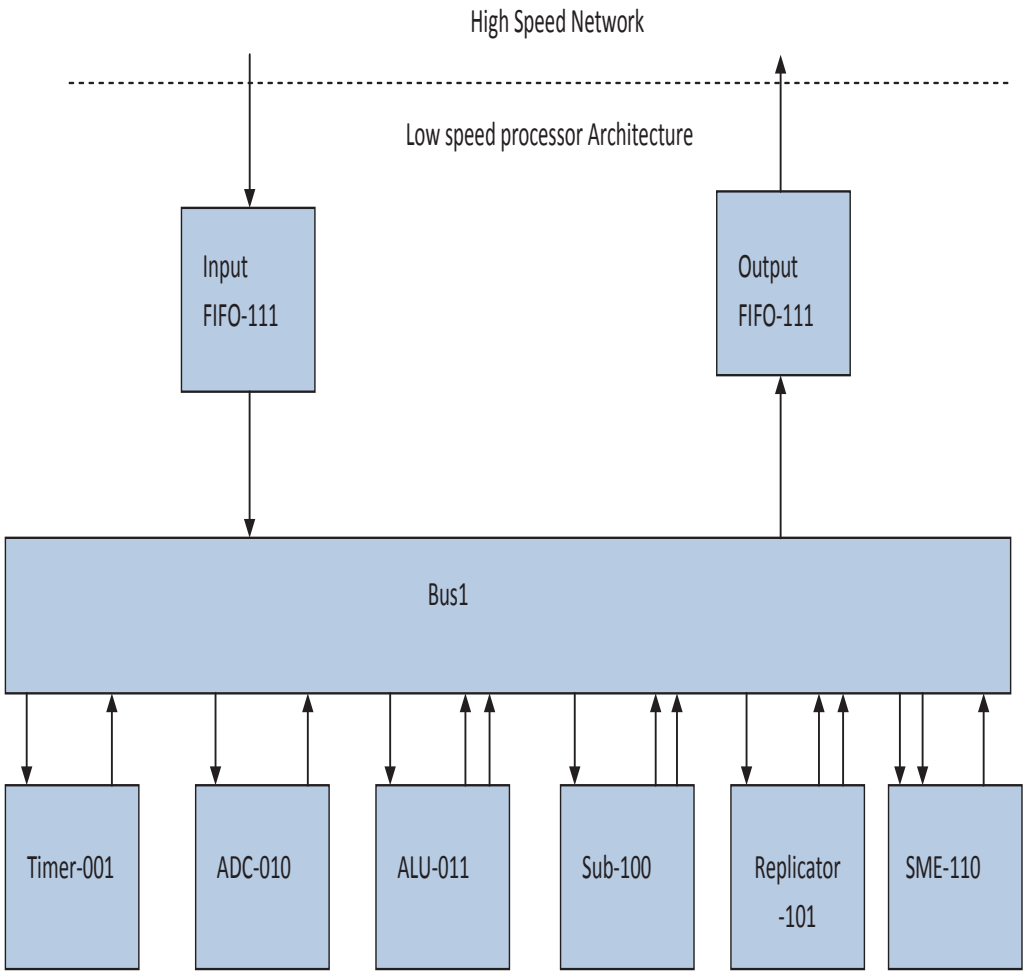

Figure D.1: Architecture Model for Architecture-I

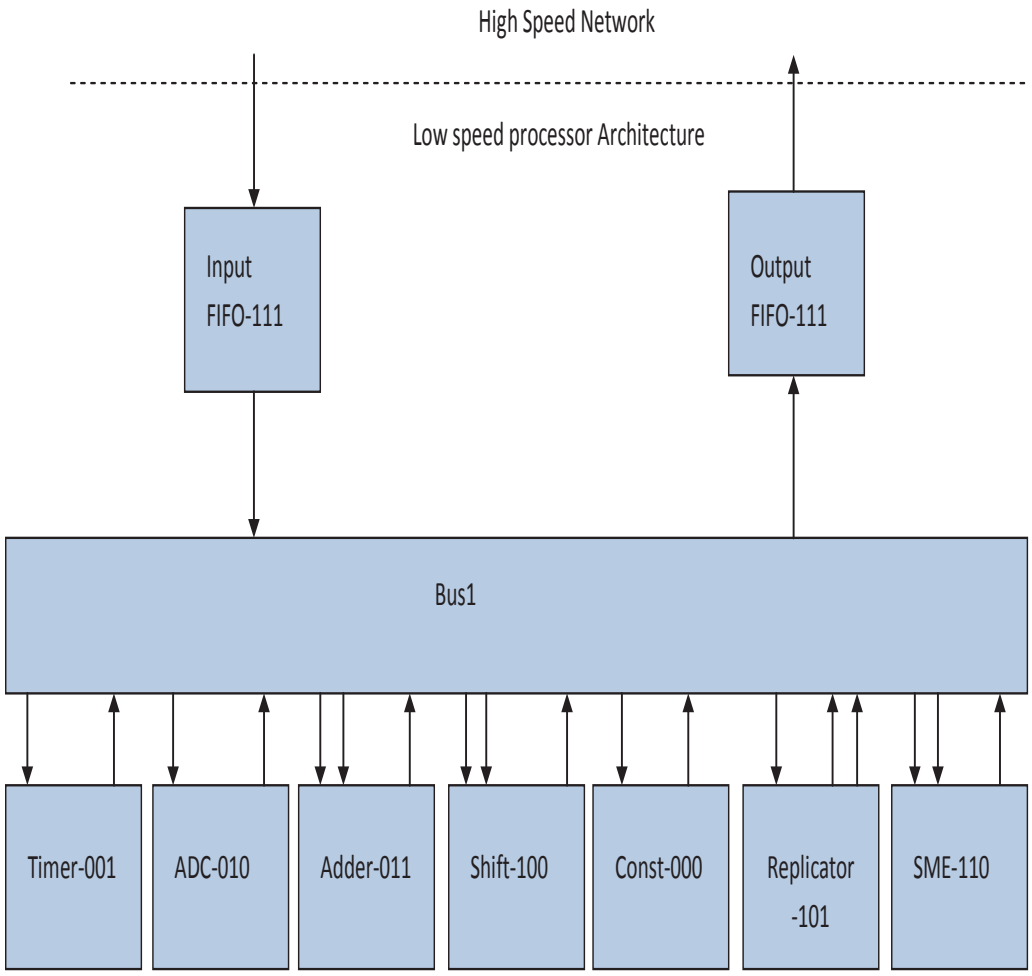

Figure D.2: Architecture Model for Architecture-II

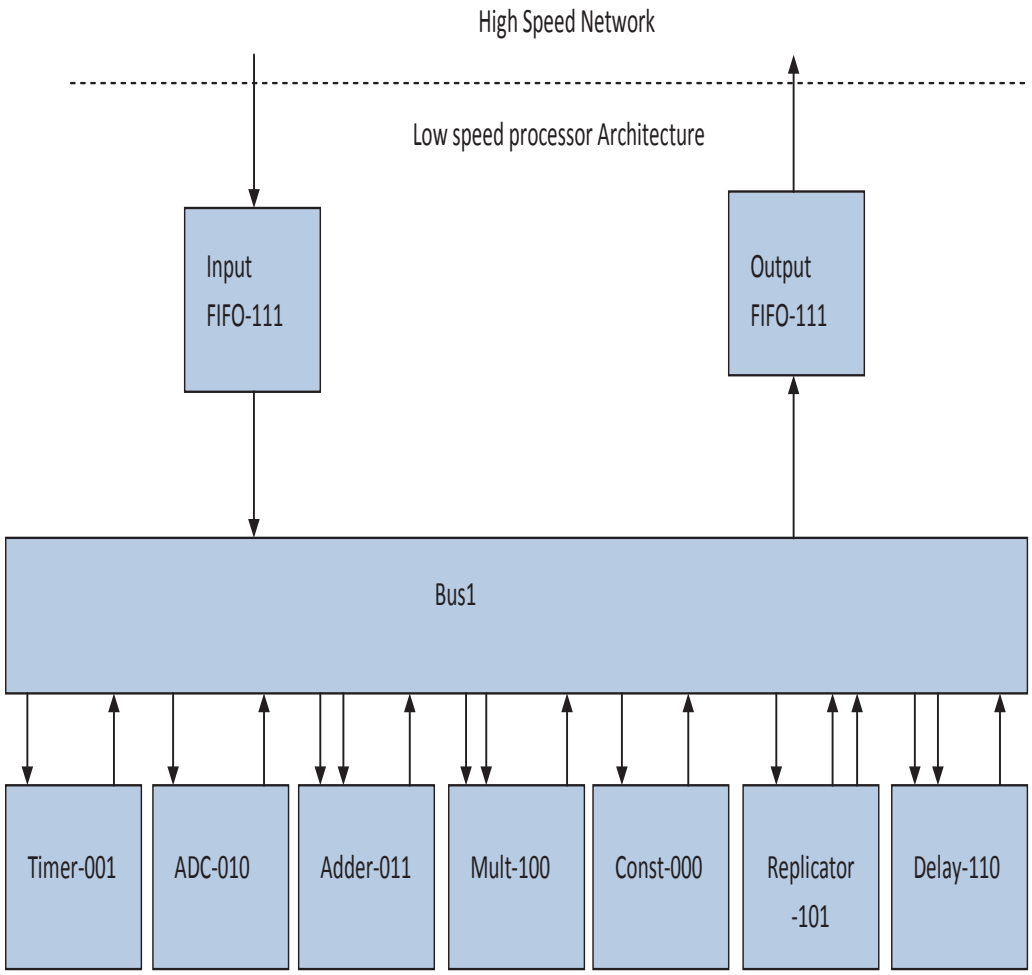

Figure D.3: Architecture Model for Architecture-III

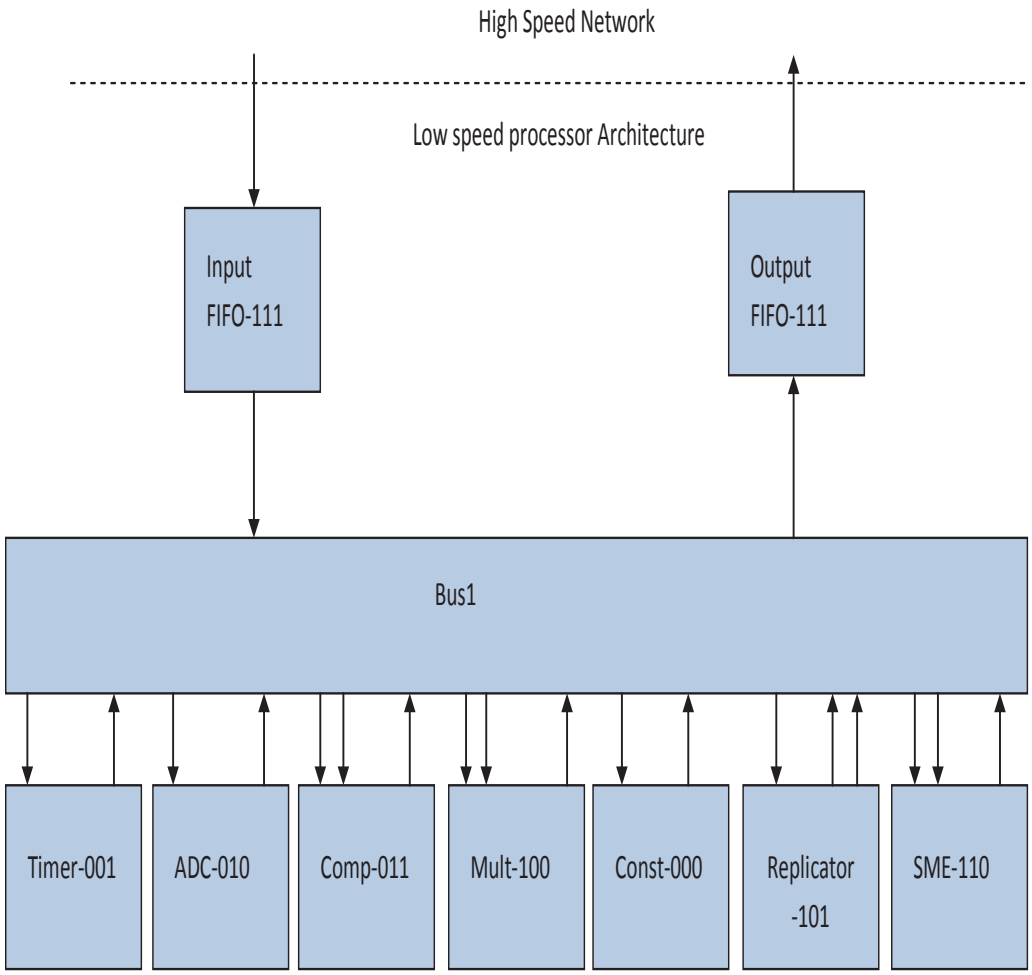

Figure D.4: Architecture Model for Architecture-IV

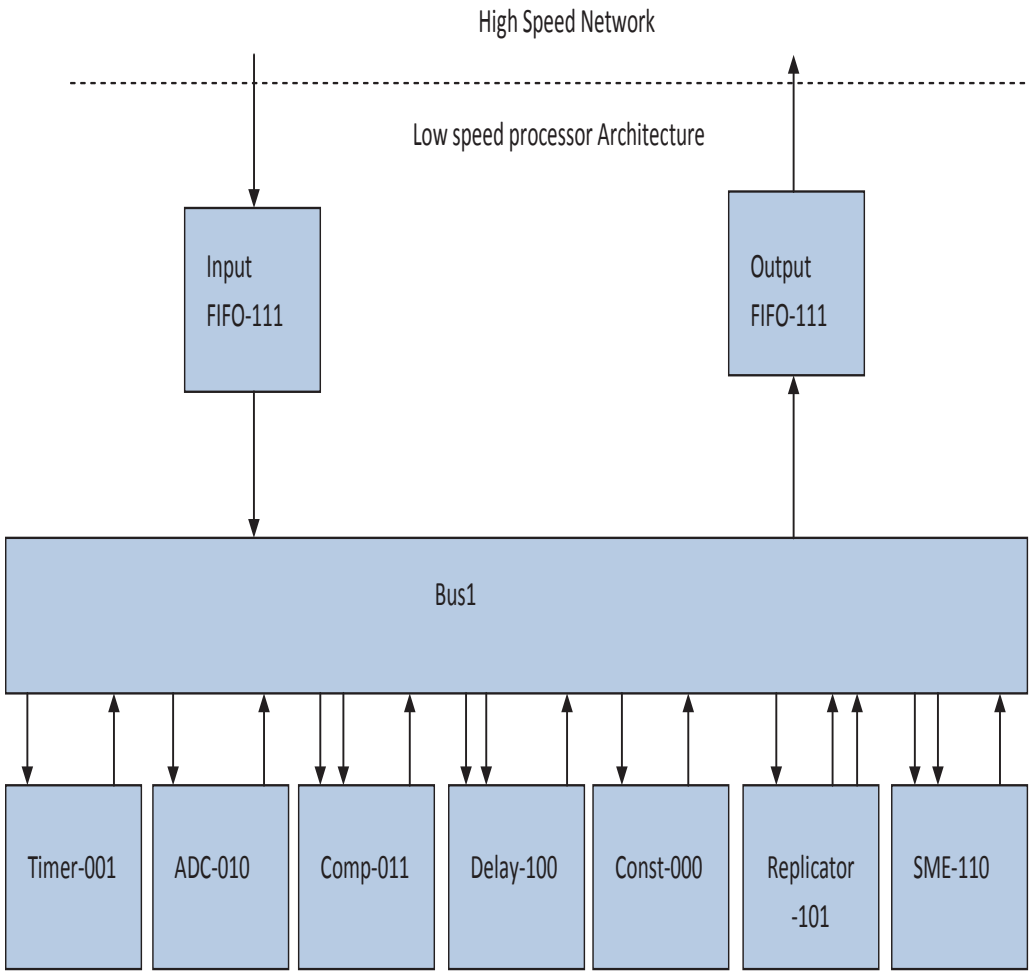

Figure D.5: Architecture Model for Architecture-VI

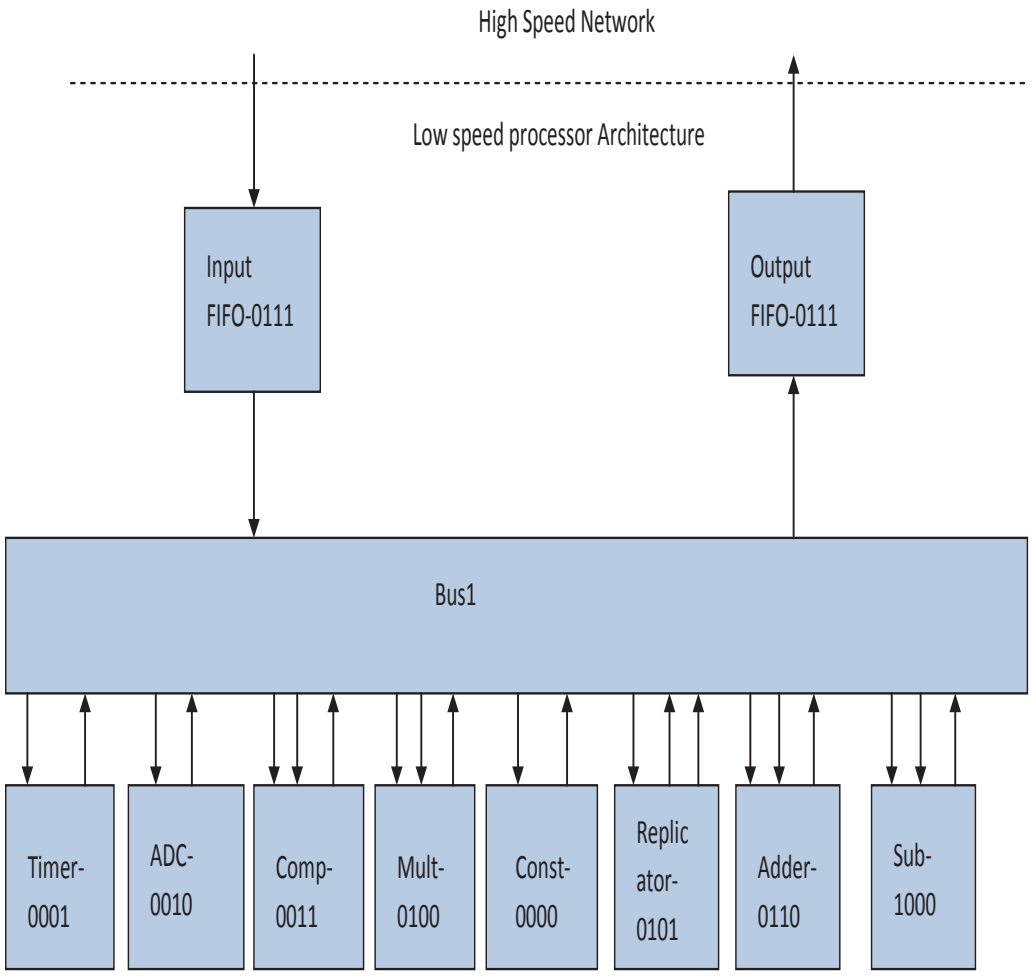

Figure D.6: Architecture Model for Architecture-V

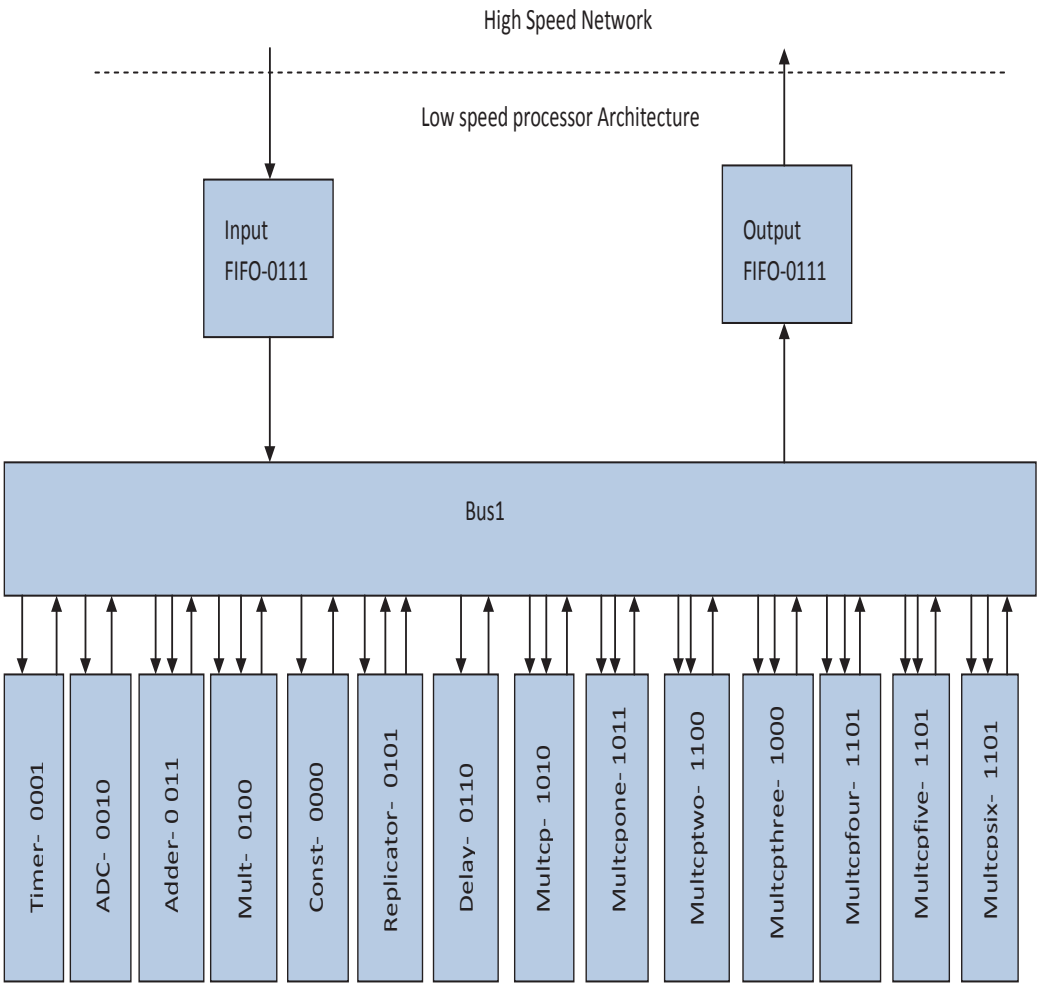

Figure D.7: Architecture Model for Architecture-VII

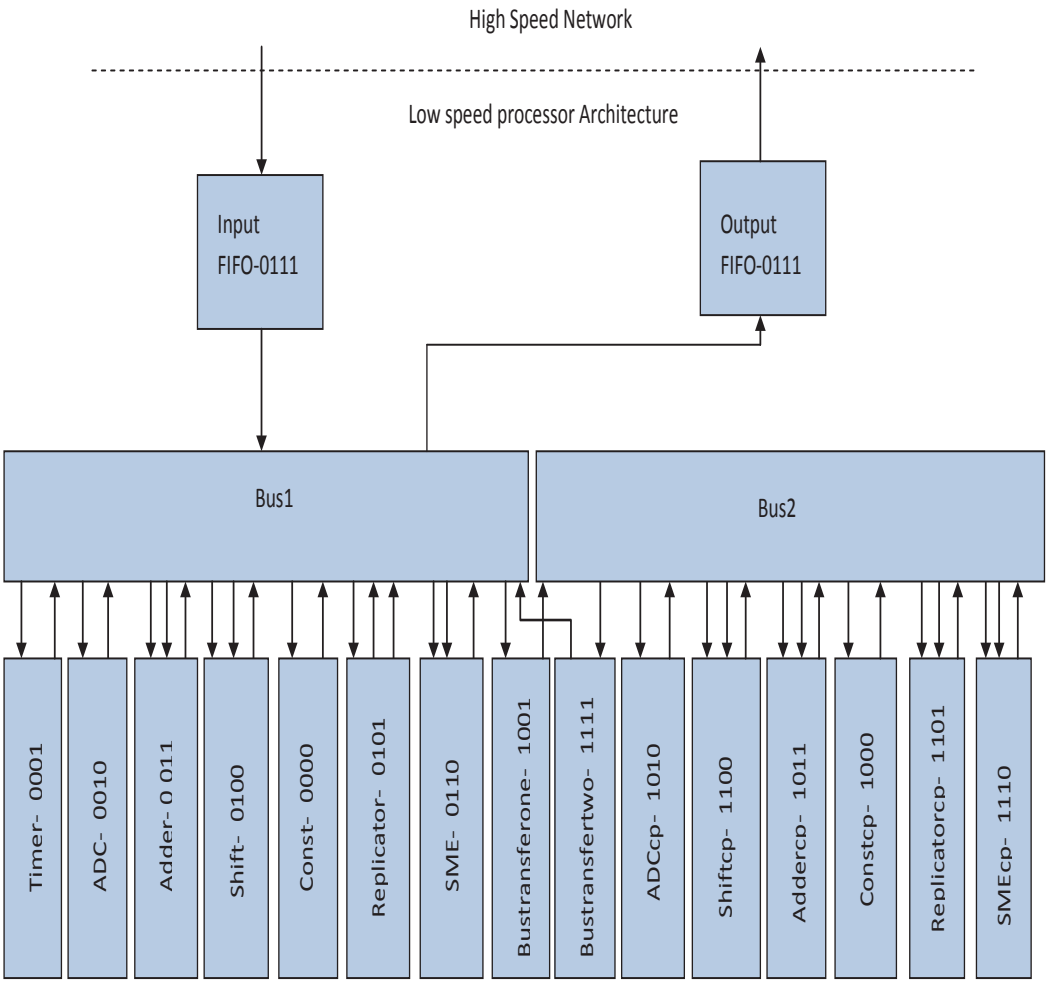

Figure D.8: Architecture Model for Architecture-VIII

Appendix E

Timing Diagram

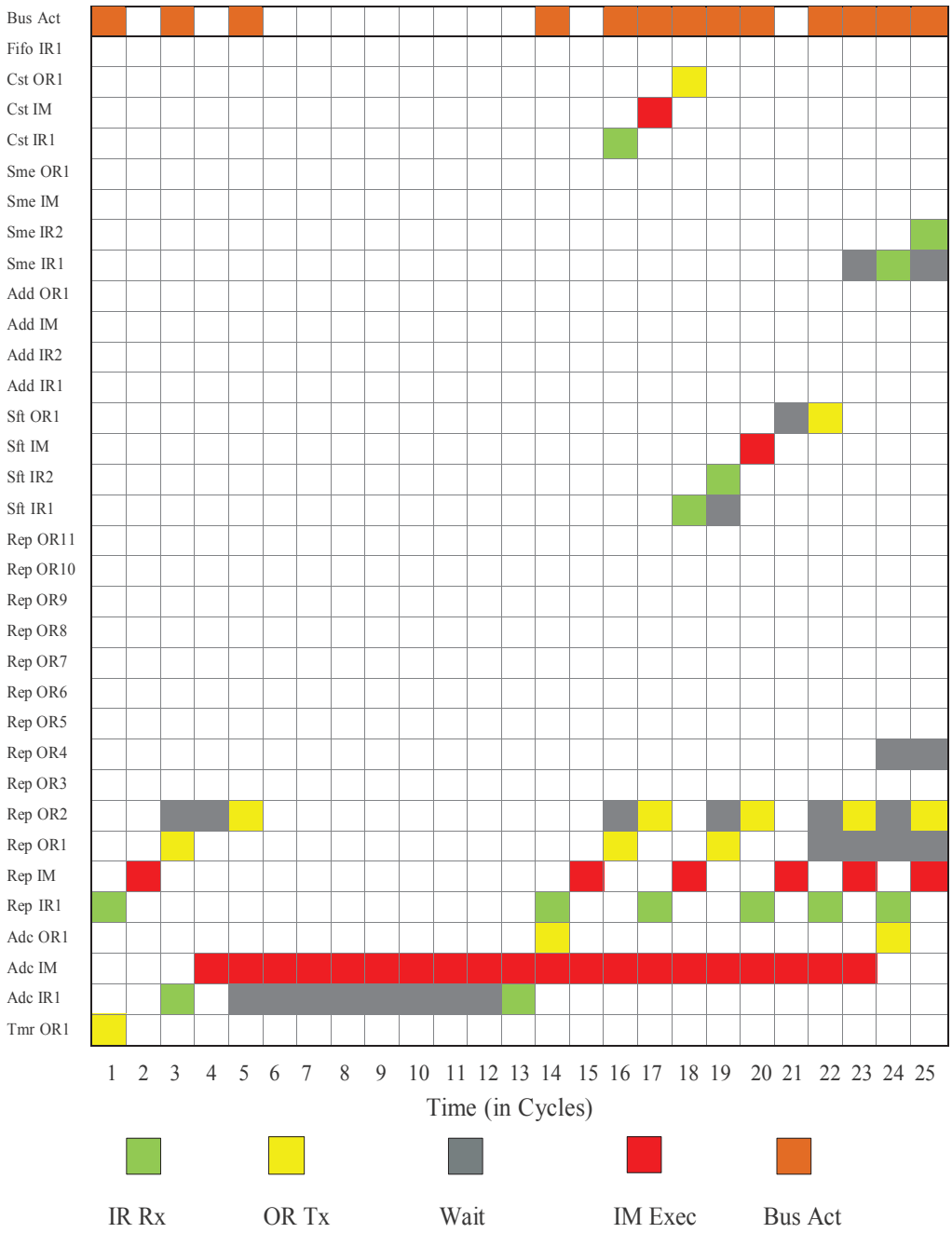

Figure E.1: Schedule Diagram for 8 Bit Multiplier(Part 1)

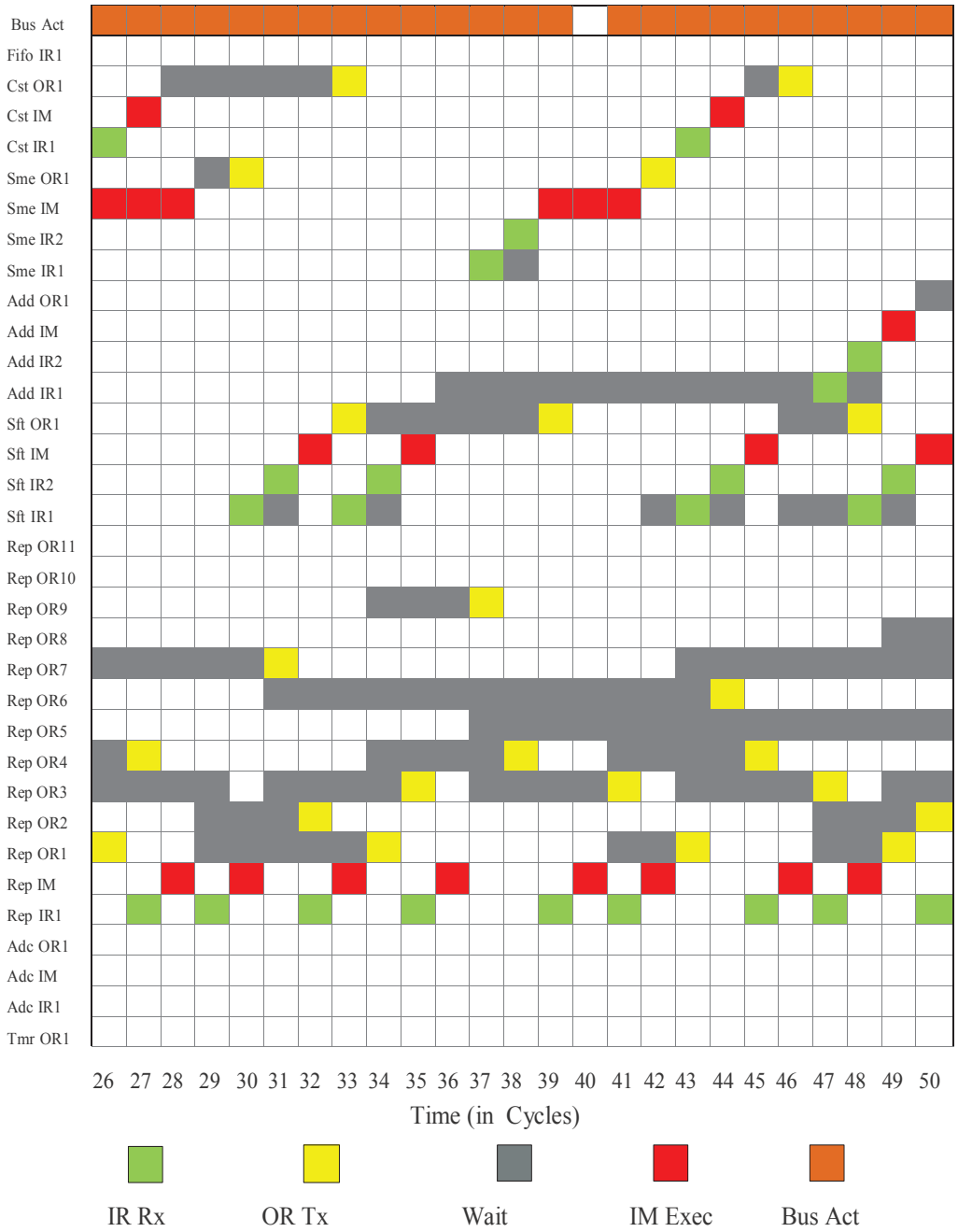

Figure E.2: Schedule Diagram for 8 Bit Multiplier(Part 2)

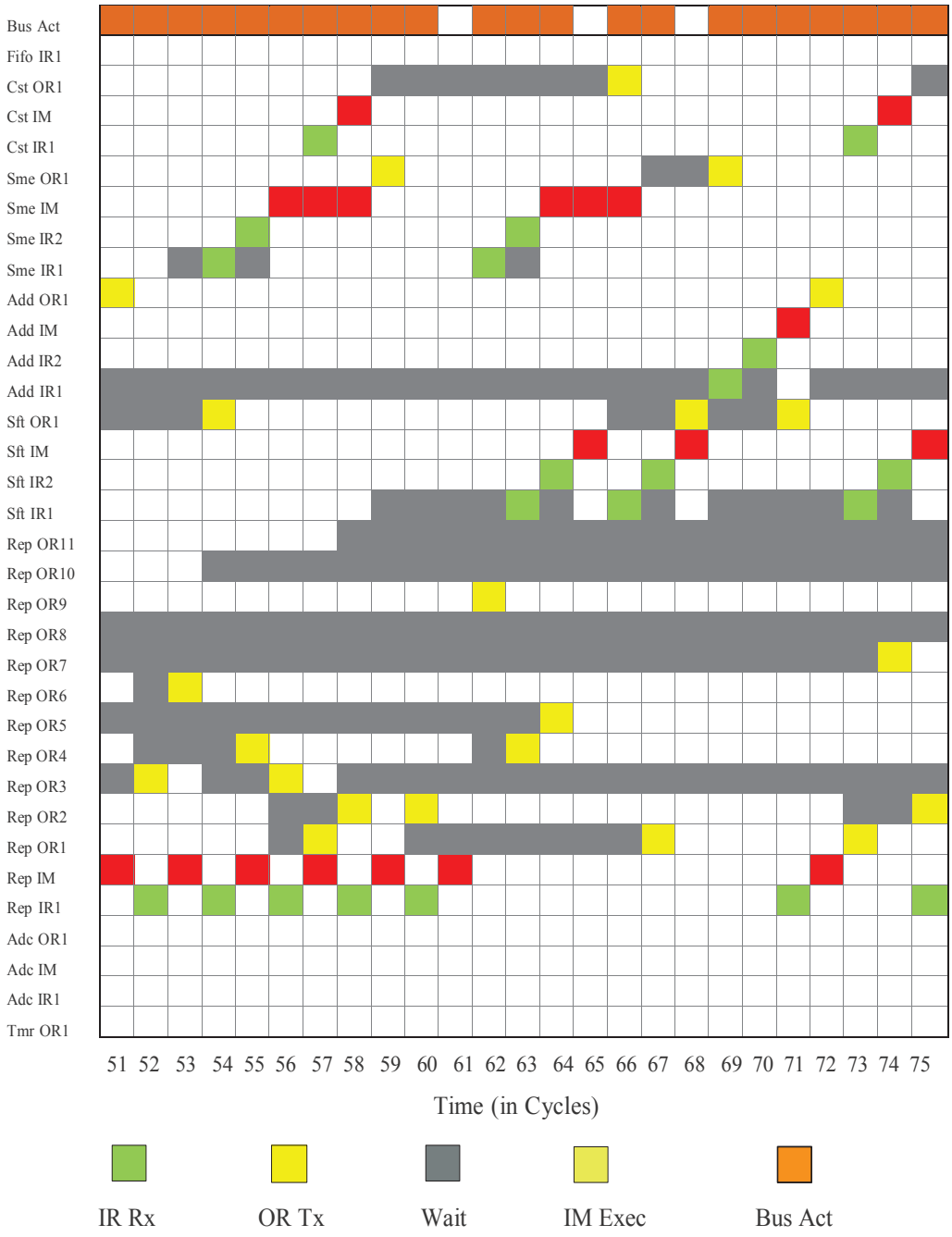

Figure E.3: Schedule Diagram for 8 Bit Multiplier(Part 3)

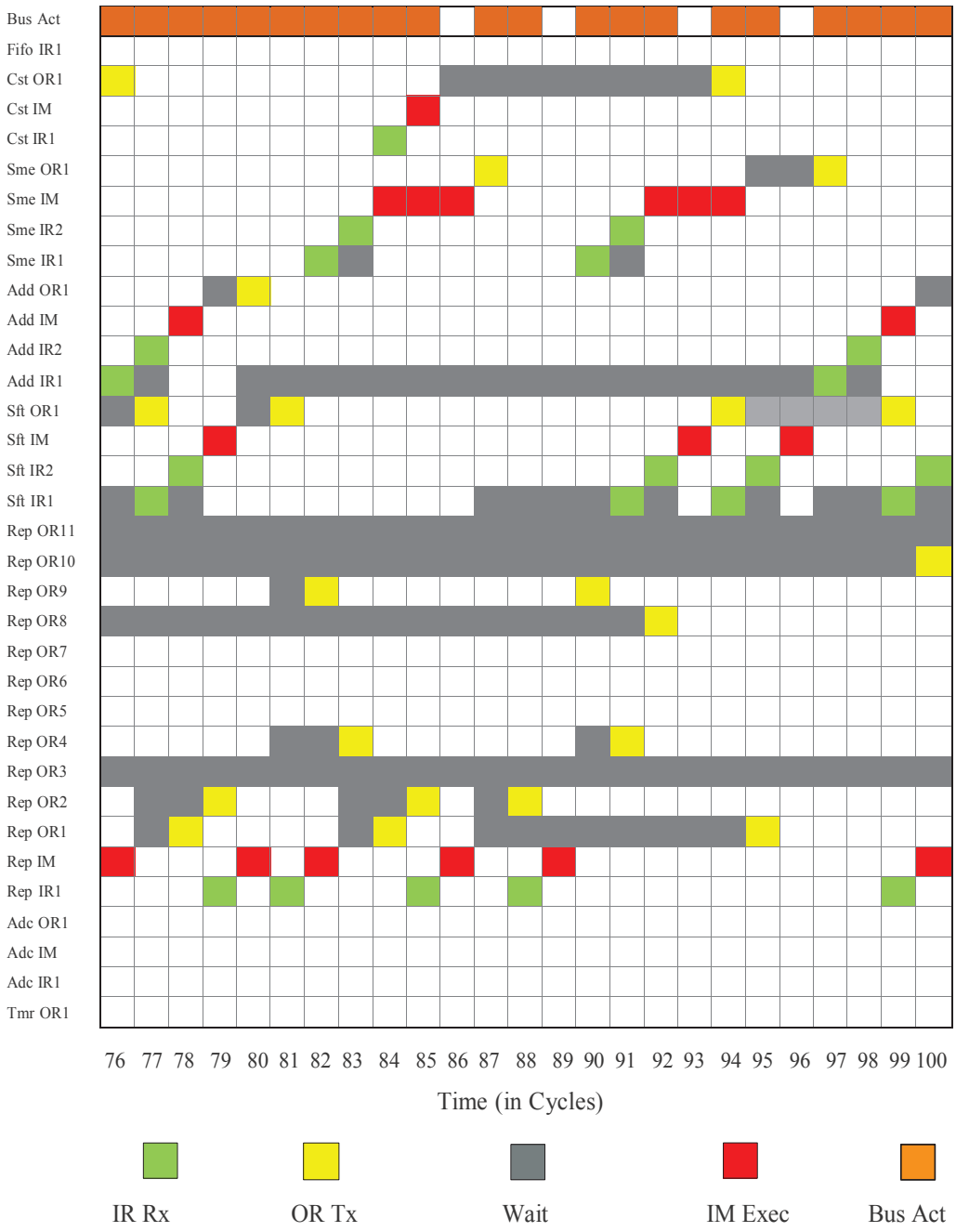

Figure E.4: Schedule Diagram for 8 Bit Multiplier(Part 4)

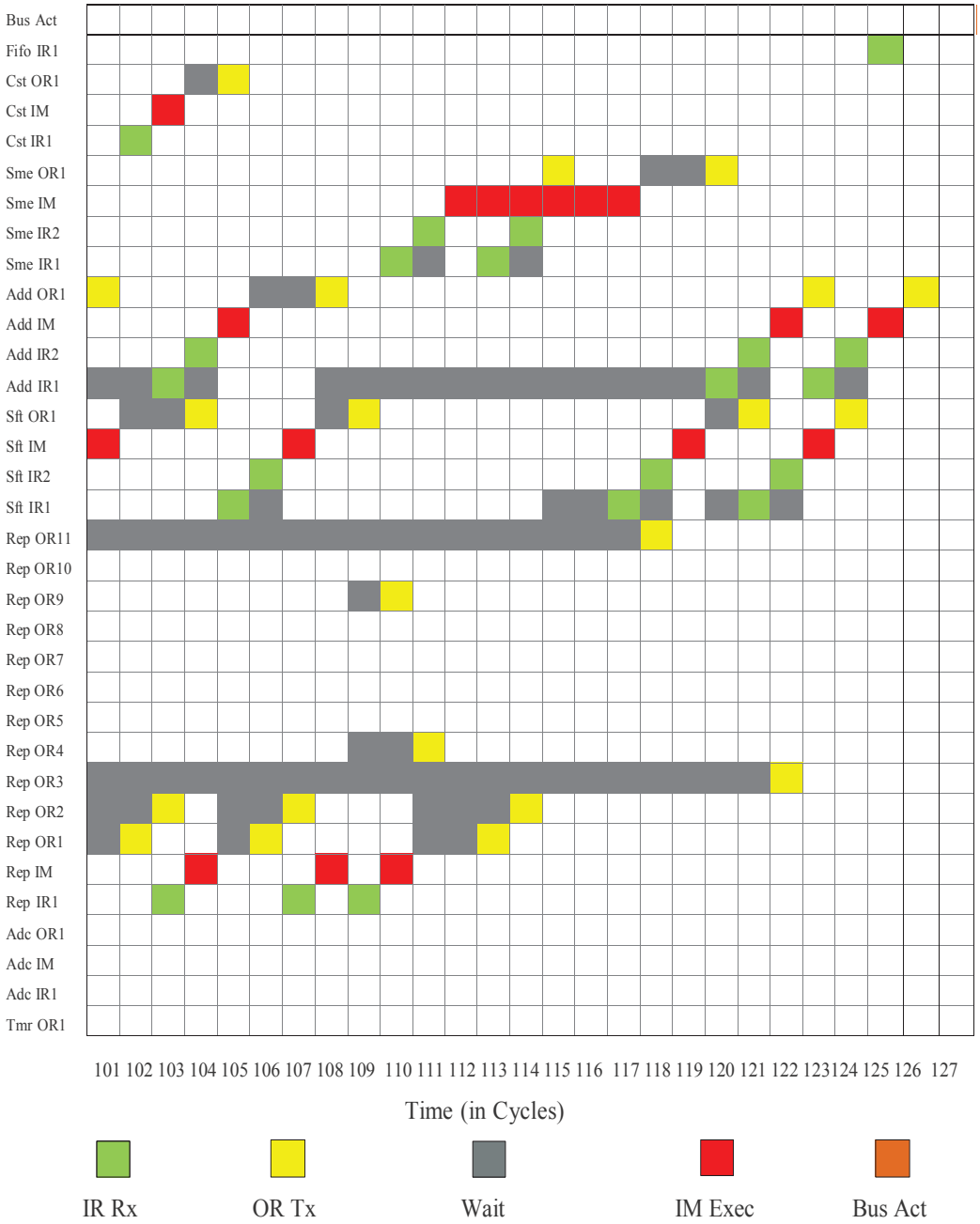

Figure E.5: Schedule Diagram for 8 Bit Multiplier(Part 5)

Appendix F

PIC Implementations

Code listing 1 Four Bit Multiplier Implementation On PIC

```
#include <htc.h>
#define _XTAL_FREQ 1000000// oscillator frequency for __delay_us()
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTDIS & INTIO);
unsigned char ones = 0,twos = 0;
void main()
{
        int a=0;
        TRISIO = 0;// Configure all IO as outputs
        ANSEL = 1<<0;// make only AN0 analog
        CMCON = 7;// disable comparators (CM = 7)
        OPTION = 0b11000010;// configure Timer0:
        ADCON0 = 0b10000001; // configure ADC
        T0IE = 1;// enable Timer0 interrupt
        ei();//and global interrupts
        for (;;)
        { // Main loop
                a =a+1;// set breakpoint
        }
}
void interrupt isr(void)
{
        T0IF = 0;//Breakpoint set here// clear interrupt flag
        unsigned char ans;
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        ones = ADRESL & 0x0f;// get last 4-bits of ADRESL
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        twos = ADRESL & 0x0f;// get last 4-bits of ADRESL
        ans = ones * twos;//Multiply the two 4-bit variables
        ans = ans+1;// set breakpoint
```

```
}
```

Code listing 2 Six Bit Multiplier Implementation on PIC

```
#include <htc.h>
#define _XTAL_FREQ 1000000// oscillator frequency for __delay_us()
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTDIS & INTIO);
unsigned char ones = 0,twos = 0;
void main()
{
        int a=0;
        TRISIO = 0;// Configure all IO as outputs
        ANSEL = 1<<0;// make only AN0 analog
        CMCON = 7;// disable comparators (CM = 7)
        OPTION = 0b11000010;// configure Timer0:
        ADCON0 = 0b10000001; // configure ADC
        T0IE = 1;// enable Timer0 interrupt
        ei();//and global interrupts
        for (;;)
        {       // Main loop
                a =a+1;// set breakpoint
        }
}
void interrupt isr(void)
{
        T0IF = 0;//Breakpoint set here// clear interrupt flag
        unsigned short ans;
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        ones = ADRESL & 0x3f;// get last 6-bits of ADRESL
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        twos = ADRESL & 0x3f;// get last 6-bits of ADRESL
        ans = ones * twos;//Multiply the two 6-bit variables
        ans = ans+1;// set breakpoint
}
```

Code listing 3 Eight Bit Multiplier Implementation on PIC

```
#include <htc.h>
#define _XTAL_FREQ 1000000// oscillator frequency for __delay_us()
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTDIS & INTIO);
unsigned char ones = 0,twos = 0;
void main()
{
        int a=0;
        TRISIO = 0;// Configure all IO as outputs
        ANSEL = 1<<0;// make only AN0 analog
        CMCON = 7;// disable comparators (CM = 7)
        OPTION = 0b11000010;// configure Timer0:
        ADCON0 = 0b10000001; // configure ADC
        T0IE = 1;// enable Timer0 interrupt
        ei();//and global interrupts
        for (;;)
        { // Main loop
                a =a+1;//set breakpoint
        }
}
void interrupt isr(void)
{
        T0IF = 0;//Breakpoint set here// clear interrupt flag
        unsigned short ans;
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        ones = ADRESL & 0xff;// get 8-bits of ADRESL
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        twos = ADRESL & 0xff;// get 8-bits of ADRESL
        ans = ones * twos;//Multiply the two 8-bit variables
        ans = ans+1;//set breakpoint
}
```

Code listing 4 Two Coefficient Filter Implementation on PIC

```
#include <htc.h>
#define _XTAL_FREQ 1000000// oscillator frequency for __delay_us()
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTDIS & INTIO);
unsigned char ones = 0,twos = 0;
char a0=1, a1=2;
void main()
{
        int a=0;
        TRISIO = 0;// Configure all IO as outputs
        ANSEL = 1<<0;// make only AN0 analog
        CMCON = 7;// disable comparators (CM = 7)
        OPTION = 0b11000010;// configure Timer0:
        ADCON0 = 0b10000001; // configure ADC
        T0IE = 1;// enable Timer0 interrupt
        ei();//and global interrupts
        for (;;)
        {       // Main loop
                a =a+1;// set breakpoint
        }
}
void interrupt isr(void)
{
        T0IF = 0;//Breakpoint set here// clear interrupt flag
        unsigned short ans;
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        ones = ADRESL & 0x1f;// get last 5-bits of ADRESL
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        twos = ADRESL & 0x1f;// get last 5-bits of ADRESL
        ans = ones * a1 + twos * a0;//Find result
        ans = ans+1;//set breakpoint
}
```

Code listing 5 Three Coefficient Filter Implementation on PIC

```
#include <htc.h>
#define _XTAL_FREQ 1000000// oscillator frequency for __delay_us()
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTDIS & INTIO);
unsigned char ones = 0,twos = 0, threes=0;
char a0=1, a1=2, a2=3 ;
void main()
{
        int a=0;
        TRISIO = 0;// Configure all IO as outputs
        ANSEL = 1<<0;// make only AN0 analog
        CMCON = 7;// disable comparators (CM = 7)
        OPTION = 0b11000010;// configure Timer0:
        ADCON0 = 0b10000001; // configure ADC
        T0IE = 1;// enable Timer0 interrupt
        ei();//and global interrupts
        for (;;)
        {       // Main loop
                a =a+1;//set breakpoint
        }
}
void interrupt isr(void)
{
        T0IF = 0;//Breakpoint set here// clear interrupt flag
        unsigned short ans;
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        ones = ADRESL & 0x1f;// get last 5-bits of ADRESL
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        twos = ADRESL & 0x1f;// get last 5-bits of ADRESL
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        threes = ADRESL & 0x1f;// get last 5-bits of ADRESL
        ans = ones * a2 + twos * a1+ threes * a0;//Find result
        ans = ans+1;//set breakpoint
}
```

Code listing 6 Eight Coefficient Filter Implementation on PIC

```
#include <htc.h>
#define _XTAL_FREQ 1000000// oscillator frequency for __delay_us()
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTDIS & INTIO);
unsigned char ones = 0,twos = 0, threes=0, fours=0, fives=0, sixes=0, sevenes=0, eights=0;
char a0=1, a1=2, a2=3, a3=4, a4=5, a5=6, a6=7, a7=8 ;
void main()
{
        int a=0;
        TRISIO = 0;// Configure all IO as outputs
        ANSEL = 1<<0;// make only AN0 analog
        CMCON = 7;// disable comparators (CM = 7)
        OPTION = 0b11000010;// configure Timer0:
        ADCON0 = 0b10000001; // configure ADC
        T0IE = 1;// enable Timer0 interrupt
        ei();//and global interrupts
        for (;;)
        {       // Main loop
                a =a+1;// set breakpoint
        }
}
void interrupt isr(void)
{
        T0IF = 0;//Breakpoint set here// clear interrupt flag
        unsigned short ans;
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        ones = ADRESL & 0x1f;// get last 5-bits of ADRESL
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        twos = ADRESL & 0x1f;// get last 5-bits of ADRESL
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        threes = ADRESL & 0x1f;// get last 5-bits of ADRESL
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        fours = ADRESL & 0x1f;// get last 5-bits of ADRESL
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        fives = ADRESL & 0x1f;// get last 5-bits of ADRESL
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        sixes = ADRESL & 0x1f;// get last 5-bits of ADRESL
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        sevens = ADRESL & 0x1f;// get last 5-bits of ADRESL
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        eights = ADRESL & 0x1f;// get last 5-bits of ADRESL

        ans = ones * a7+twos * a6+threes*a5 +fours * a4 +fives * a3
        +sixes * a2 + sevens * a1+ eights * a0;//Find result
        ans = ans+1;// set breakpoint
}
```

Code listing 7 Square Root finder Implementation on PIC

```
#include <htc.h>
#define _XTAL_FREQ 1000000// oscillator frequency for __delay_us()
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTDIS & INTIO);
unsigned char ones = 0;
char a0=8;//stores the square root
void main()
{
        int a=0;
        TRISIO = 0;// Configure all IO as outputs
        ANSEL = 1<<0;// make only AN0 analog
        CMCON = 7;// disable comparators (CM = 7)
        OPTION = 0b11000010;// configure Timer0:
        ADCON0 = 0b10000001; // configure ADC
        T0IE = 1;// enable Timer0 interrupt
        ei();//and global interrupts
        for (;;)
        {       // Main loop
                a =a+1;// set breakpoint
        }
}
void interrupt isr(void)
{
        T0IF = 0;//Breakpoint set here// clear interrupt flag
        unsigned short ans,temp;
        int i=0,j=4;
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        ones = ADRESL & 0x0f;// get 8-bits of ADRESL

        while(i<4)
        {
                if(j==0)
                        j=1;
                temp= a0*a0;
                //State machine function
                if(temp<ones)
                {
                        a0=a0|j;// set the bit

                }
                else if(temp>ones)
                {
                        a0=a0& ~(j*2)
                        a0=a0|j;        //shift the 1 to the right
                }
                j=j/2;
                i=i-1;
        }

        ans = ans+1;//set breakpoint
}
```

Code listing 8 Robot Path Follower Implementation on PIC

```
#include <htc.h>
#define _XTAL_FREQ 1000000// oscillator frequency for __delay_us()
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTDIS & INTIO);
unsigned char ones = 0;
char a0=8;
void main()
{
        int a=0;
        TRISIO = 0b11111100;// Configure GP1 and GP0 as output;
        ANSEL = 1<<0;// make only AN0 analog
        CMCON = 7;// disable comparators (CM = 7)
        OPTION = 0b11000010;// configure Timer0:
        ADCON0 = 0b10000001; // configure ADC
        T0IE = 1;// enable Timer0 interrupt
        ei();//and global interrupts
        for (;;)
        { // Main loop
                a =a+1;//set breakpoint
        }
}
void interrupt isr(void)
{
        T0IF = 0;//Breakpoint set here// clear interrupt flag
        unsigned short p1_0, p1_1;// left sensor and right sensor value respectively
        unsigned short ans;
        unsigned short range1=1, range2=5;// defines the range in which
                                // the inputs should be turn on and of the motors
        int i=0;
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        p1_0 = ADRESL & 0x0f;// get 8-bits of ADRESL

        while (GODONE)// wait until done
        ;
        p1_1 = ADRESL & 0x0f;// get 8-bits of ADRESL

        //state machine function
        if((p1_0 <range1)&&(p1_1 > range2))
        { //Turn right
                GPIO0=1; //Left motor running in forward direction.
                 GPIO1=0; //Right motor not running
                for(i=0;i<1275;i++)//delay
                {
                }
                GPIO0=0; //Left motor not running.
                GPIO1=0; //Right motor not running
        }

        else if((p1_0 > range2)&&(p1_1 < range1))
        { //turn left
                GPIO0=0; //Left motor not running.
                 GPIO1=1; //Right motorrunning in forward direction
                for(i=0;i<1275;i++)//delay
                {
                }
                GPIO0=0; //Left motor not running.
                GPIO1=0; //Right motor not running
        }
        else
        { //move forward
```

```
                GPIO0=1; //Left motor running in forward direction.
                GPIO1=1; //Right motorrunning in forward direction
        }

        ans = ans+1;//set breakpoint
}
```

Code listing 9 Free Fall Detector Implementation on PIC

```
#include <htc.h>
#define _XTAL_FREQ 1000000// oscillator frequency for __delay_us()
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTDIS & INTIO);
unsigned char ones = 0;
char a0=8;
void main()
{
        int a=0;
        TRISIO = 0b11111100;// Configure GP1 and GP0 as output;
        ANSEL = 1<<0;// make only AN0 analog
        CMCON = 7;// disable comparators (CM = 7)
        OPTION = 0b11000010;// configure Timer0:
        ADCON0 = 0b10000001; // configure ADC
        T0IE = 1;// enable Timer0 interrupt
        ei();//and global interrupts
        for (;;)
        { // Main loop
                a =a+1;//set breakpoint
        }
}
void interrupt isr(void)
{
        T0IF = 0;//Breakpoint set here// clear interrupt flag
        unsigned short p1_0, p1_1, p1_2;// Accelerometer digital values of the three axes.
        unsigned short v_ref=3.3; //reference voltage of adc.
        unsigned short res=1023; //resolution of the adc
        unsigned short VoltsRx, VoltsRy, VoltsRz; //adc output converted to voltage
        unsigned short DeltaVoltsRx, DeltaVoltsRy, DeltaVoltsRz; // voltage shifts from 0g voltage
        unsigned short zerog_v=1.6; // zero g voltage level
        unsigned short R_square;//resultsnt vector square
        unsigned short ans;
        int i=0;
        GODONE = 1;// start conversion
        while (GODONE)// wait until done
        ;
        p1_0 = ADRESL & 0x0f;// get 8-bits of ADRESL

        while (GODONE)// wait until done
        ;
        p1_1 = ADRESL & 0x0f;// get 8-bits of ADRESL
        while (GODONE)// wait until done
        ;
        p1_2 = ADRESL & 0x0f;// get 8-bits of ADRESL

        //state machine function
        VoltsRx = p1_0 * v_ref / res ;
        VoltsRy = p1_1 * v_ref / res ;
        VoltsRz = p1_2 * v_ref / res ;

        DeltaVoltsRx = VoltsRx - zerog_v ;
        DeltaVoltsRy = VoltsRx - zerog_v ;
        DeltaVoltsRz = VoltsRx - zerog_v ;


        R_square =  DeltaVoltsRx*DeltaVoltsRx + DeltaVoltsRy*DeltaVoltsRy
            + DeltaVoltsRz*DeltaVoltsRz;
        if(R_square==0)// under free fall
        {
                GPIO0=1; //signal to turn off hard disk
        }
        else
```

```
        {
                GPIO0=0;
        }
        ans = ans+1;//set breakpoint
}
```