Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

2010

Pond IDE: Machine level program development environment and register transfer level simulator for a massively parallel computer architecture

Jesse Muszynski

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Muszynski, Jesse, "Pond IDE: Machine level program development environment and register transfer level simulator for a massively parallel computer architecture" (2010). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Pond IDE: Machine Level Program Development Environment and Register Transfer Level Simulator for a Massively Parallel Computer Architecture

by

Jesse J. Muszynski

A Thesis Submitted
in
Partial Fulfillment
of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electrical Engineering

Approved by:

Dr. Dorin Patru, Assistant Professor
Thesis Advisor, Department of Electrical and Microelectronic Engineering

---

Dr. Eric Peskin, Assistant Professor
Committee Member, Department of Electrical and Microelectronic Engineering

---

Dr. Daniel B. Phillips, Associate Professor
Committee Member, Department of Electrical and Microelectronic Engineering

---

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING
KATE GLEASON COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
August, 2010

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Pond IDE: Machine Level Program Development Environment and Register Transfer Level Simulator for a Massively Parallel Computer Architecture

I, Jesse J. Muszynski, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Jesse J. Muszynski

Date

© Copyright 2010 by Jesse J. Muszynski
All Rights Reserved

Dedication

This thesis is dedicated to my parents, grandparents, and Carrie Crowley for their unending support throughout my academic career.

Acknowledgments

I would like to acknowledge my advisor, Dr. Patru for his contributions to this work and my committee members for their guidance and feedback in the preparation of this document. Additionally I would like to acknowledge Trevor West, my friend and colleague, for his help consulting me on the proper development of the Computer Science related portions of the IDE, and time spent aiding me with resolution of difficult to solve software bugs.

Abstract

Pond IDE: Machine Level Program Development Environment and Register Transfer Level Simulator for a Massively Parallel Computer Architecture

Jesse J. Muszynski

Supervising Professor: Dr. Dorin Patru

As computing architectures are being implemented in late and post silicon technologies, fault tolerance and concurrent operation are becoming increasingly important. It is already common knowledge that manufacturers are putting two, four or even more cores on a single silicon die to improve computing performance. The proposed architecture far exceeds this number by grouping thousands or even millions of simple *reduced instruction set computing* (RISC) processors, each of which is capable of a single operation at a time, and to communicate with its eight nearest neighbors. In this architecture, if a single core or cluster of cores have defects at the time of manufacture, or later in the life of the system, it is possible to test and disable them as necessary.

A fine-grained architecture of this kind calls for a parallel programming style. One approach to this problem is the use of a parallelizing compiler. Another approach may be to use one of the several *application programming interfaces* (APIs) available for standard text based programming languages, with some built-in features for parallel programming.

This work has generated a solution for creating machine level parallel programs for the massively parallel computer architecture described above using text and graphical means. To support this programming method, an *integrated development environment* (IDE) and a zero communication latency, *register transfer level* (RTL) simulator have been developed. Experimental results include the implementation of fundamental data processing algorithms and complex functions.

Contents

Dedication . . . . **iv**

Acknowledgments . . . . **v**

Abstract . . . . **vi**

1 Background and Motivation **1**

2 Architecture Organization **4**

2.1 Architecture Overview 4
2.2 Instruction Set 8
2.3 Communications 11
2.4 Entity Movement 18
2.5 Input/Output 19
2.6 Entity Abutment 21
2.7 Instruction Execution 21
2.8 Function Calls 22
2.9 Loops 24

3 Programming Environment and Simulation Model **27**

3.1 Integrated Development Environment GUI 27
3.2 Model of a Processor 33
3.3 Specialty Tokens 34
3.4 Simulator Basics 36
3.5 Creating a Function Definition 38

4 Experimental Results **39**

4.1 Sequential Code Example - Fibonacci Series 39
4.2 Concurrent Code Example - Vector Addition 40
4.3 Integer Multiplication using the Left-Shift Algorithm 42

4.4 Floating Point Packing and Unpacking 44
4.4.1 IEEE Floating Point Standard 47
4.4.2 Unpacking the Sign 47
4.4.3 Unpacking the Exponent 48
4.4.4 Unpacking the Significand or Mantissa 48
4.4.5 Packing Floating Point Numbers 50
4.5 Floating Point Multiplication 52
4.5.1 24-bit Fixed-Point Multiplier 52
4.5.2 Function Calls to Perform Floating Point Multiplication 56
4.6 Integer Division 59

5 IDE Development and Programming 65
5.1 Program Structure 65
5.1.1 GUI Related Classes 66
5.1.2 Data Related Classes 67
5.1.3 Simulator Related Classes 69
5.2 Future Development of the IDE 71
5.2.1 Loops 71
5.2.2 Re-annotate IDs and Non-unique IDs 73
5.2.3 Token Quick View and Print 73
5.2.4 Data Structures (Data Element Arrays) 74
5.2.5 Additional Opcodes 74
5.2.6 Partially implemented IDE Features 75

6 Conclusion 79

Bibliography 80

A IDE User Guide 89
A.1 Menus 89
A.1.1 File Menu 89
A.1.2 Edit Menu 90
A.1.3 View Menu 91
A.1.4 Tools Menu 93
A.1.5 Help 93
A.2 Left Hand Tool Bar 95

A.3 Project Files 96
A.4 The Property Grid 96
A.5 Tokens on the Layout Grid 98
A.6 Conditional Tokens 100
A.7 Breakpoints 102
A.8 Function Calls 102

B IDE Development Guide 106

List of Tables

2.1 Information stored by one atomic processor, and its significance. Storage requirements, in bits, for $64K$ and $1T$ seas of atomic processors. 9
2.2 The reduced instruction set. To meet the self-imposed requirement that each atomic processor has to be a low complexity, processing element, we have included in the instruction set only fundamental arithmetic, logic, and control flow instructions. 10
2.3 Communications handshake codes. 13
2.4 Message routing scheme used during a global or entity casts. 15
2.5 Message formats. The numbers of bits in the shaded boxes change with the size of the sea of atomic processors and/or size of operands. The message codes are described in Table 2.6. ID = Identification number; S = Source; D = Destination; I = Intermediate. 18
2.6 Message codes and descriptions for non-result message types. 20

4.1 Definition of bits for the IEEE 754 standard for floating point numbers. . . 47

List of Figures

2.1 The sea of atomic processors. Each atomic processor can communicate with its eight adjacent neighbors. The input / output ports, illustrated in bold lines, can be located at the periphery, or can penetrate the sea. 5

2.2 A snapshot of the sea of atomic processors. Programs are broken down into functions. A function copy is stored in a *function definition* entity. When a function is called, the function definition creates a copy, called a *function instance*. This moves away and abuts with the *data structure* entity it has to process. 5

2.3 The communications hardware is divided into a send and receive layer. Each layer uses a set of 8 handshake buffers and one message buffer. This allows full-duplex message passage. 12

2.4 A complete communications cycle. Atomic processor x is transmitting a message to its neighbor, atomic processor y. 14

2.5 Routing and propagation of a beamed, entity, and global cast messages. In a beamed cast, the maximum number of atomic processors the message has to pass through is equal to $\max\{|x_s - x_r|, |y_s - y_r|\}$, where the sender is at (x_s, y_s) and the receiver is at (x_r, y_r). In an entity cast the propagation stops at the edge of the entity. 16

2.6 A moving entity is forced to move around another entity. Elements of the moving entity move counterclockwise as many times as needed, and correct their course as soon as possible afterwards. 17

2.7 An entity enters the sea of atomic processors, when viewed from $a \to f$, or exits the sea of atomic processors, when viewed from $f \to a$. Large entities may be input or output via multiple ports and associated atomic processors. 20

2.8 Routing and propagation of a global cast message. Using the scheme in Figure 2.9, no atomic processor will receive duplicates of the message. . . . 25

2.9 The execution of a loop without loop carried dependencies (a), and with loop carried dependencies (b). The gray shaded function instance elements are part of the loop iteration. 26

3.1 Text based code for the first four tokens of the 32-bit multiplier shown in Figure 4.7. 29
3.2 IDE Property Grid 32
3.3 Different tokens placed on the layout grid. 35

4.1 C code that calculates the first twelve elements of the Fibonacci series. . . . 40
4.2 Screen Capture of the Fibonacci Program showing the calculation of the first twelve elements of the Fibonacci series. AP 10 returns the twelfth element to the calling function. Call out boxes have been added to show the values within each processor. 41
4.3 C Code that performs vector addition for two eight element vectors. 42
4.4 Screen Capture of the Vector Add Program showing two eight element arrays of data elements, DEs 1 through 7 and 8 through 16, being added by APs 17 through 24. Call out boxes have been added indicating the operands, operation and output of each of the tokens. 43
4.5 C Code that performs integer multiplication using a left shift multiplication algorithm. 44
4.6 Screen Capture of 32-bit Left Shift Integer Multiply Function Definition . . 45
4.7 Screen Capture of 32-bit Left Shift Multiply Simulation, multiplying unsigned integers 5 and 10 to yield a result of 50. Call out boxes have been added to show the values within each processor. 46
4.8 C Code that unpacks the sign bit from the floating point number representation. 48
4.9 The sign bit of a floating point number is unpacked by simply masking the MSB. Call out boxes have been added to show the operation performed by each processor. 49
4.10 C Code that unpacks the exponent portion of the floating point number representation. 50
4.11 The exponent of a 32-bit floating point number is unpacked by masking off the exponent bits, right shifting 23 places, and subtracting the bias of 127. Call out boxes have been added to show the operation performed by each processor. 51

4.12 C Code that unpacks the mantissa or significand portion of the floating point number representation. 52
4.13 The significand of a 32-bit floating point number is unpacked by masking off the least significant 23 bits and adding a 1 in the 23rd bit position, where bit 0 is the LSB. Call out boxes have been added to show the operation performed by each processor. 53
4.14 C Code that packs a sign, exponent, and mantissa into the IEEE Floating Point representation. 54
4.15 Packing operation for a 32-bit floating point number. The function definition takes a sign, exponent and mantissa as inputs and outputs a floating point number. Call out boxes have been added to show the operation performed by each processor. 55
4.16 C Code that performs a 24-bit fixed point multiplication using a right shift multiplication algorithm. 57
4.17 Right Shift Integer multiplier for 24 bit fixed point multiplication used in floating point multiplication algorithm. Call out boxes have been added to show the operation performed by each processor. 58
4.18 C code that implements the top level floating point multiplication algorithm by making calls to the previously mentioned C code. 60
4.19 Floating point multiply program with function calls to floating point unpacking and packing operations and right shift fixed point (24bit integer) multiplier. Call out boxes have been added to show the operation performed by each processor. 61
4.20 C Code to perform division through iterative subtraction. 63
4.21 Example integer division algorithm [1]. 63
4.22 Screen Capture of sample division algorithm with *Next Row to Execute* field and non-unique ID numbers indicated in call out boxes (Simulation non-functional). 64

5.1 Export to Human Readable Code for the first four tokens of the 32-bit multiplier shown in Figure 4.7. 76

A.1 Example Project Properties Dialog from the 32-bit floating point number packing function. 92
A.2 Sample Error and Warning list from 32-bit floating point packing operation. 94

A.3 Toolbar Buttons: A) Atomic Processor, B) Concurrent Array, C) Sequential Array, D) Data Element, E) Function Call, F) Design Mode, G) Simulation Mode. 95
A.4 IDE Property Grid 99
A.5 A) The lines coming into AP1 are highlighted for easier reading, the line exiting AP1 is in its normal unhighlighted state. B) The source token for AP2 has been deleted, a red stub remains. 100
A.6 Conditional Tokens A)Before Simulation tokens are yellow, B)During Simulation tokens change green or red based on if the condition is met. 101
A.7 Example of the 32-bit Left Shift Multiplier stopped at a breakpoint. 103
A.8 Example of the 32-bit Left Shift Multiplier stepped one step forward after a breakpoint. 104
A.9 Example function call property grid for the Floating Point pack operation called in the floating point multiplication program. 105

Chapter 1

Background and Motivation

Parallelism and concurrency are inherent in many computational tasks. Techniques that exploit instruction and thread level parallelism in traditional von Neumann architectures have been successfully applied in single processors, as described by Hennessey and Patterson in [2]. During the past decade researchers and manufacturers have turned to multi-core processors, which at the present time are limited to just a few cores [3–8]. Scaling up the techniques used to exploit instruction and thread level parallelism in single core processors to many core processors is challenging for both hardware and software designers [9–12].

As pointed out by Hennessy and Patterson in [2], multi-core processors are a combination of computer architecture and communications architecture. Computer networks on a chip or cluster computing on a chip are adapting the vast knowledge base of designs and architectures of macro computer networks to the micro scale, [13–26]. Marculescu *et al.* in [27] classify outstanding research problems related to networks on chip into 15 categories. Predominant are problems related to communications infrastructure and communications paradigms, as illustrated by [28–54]. Dongarra *et al.* explore the potential symbiosis between networks on chip and multicore processors in [55].

Late and post silicon era integrated circuit fabrication technologies will continue to increase the number of components on a chip to billions and trillions. The sheer increase in

number will not translate into an increase in performance unless new parallel and concurrent architectures are developed, as pointed out by Rabaey and Malik in [56], and Wen-mei *et al.* in [57]. These new architectures will have to address reliability at the circuit and system levels because some components will experience premature, transient or permanent failures, as highlighted by Austin *et al.* in [58]. Lei Zhang *et al.* address reliability and fault tolerance in networks on chip in [59]. Power dissipation will have to be mitigated starting at the system level. This is already being considered in multicore processors [60, 61], and in networks on chip [62–67]. Nano architectures attempt to specifically address the aforementioned challenges posed by late and post silicon technologies [68–76].

Taking into account the above considerations, a focus has been placed on exploring the feasibility of a hardware design in massively parallel processing. The design itself consists of an undetermined, but large number of simple, interconnected processing elements referred to as a *sea* of processors. Each element has the ability to communicate only with its eight nearest neighbors through a dedicated message passing layer. Messages intended to be passed a distance further than a nearest neighbor must propagate from processor to processor using algorithms designed to optimize message passing using the shortest number of hops to get to the intended recipients. While the design of the underling hardware is important and is discussed in Chapter 2, it is not the primary research focus. The architecture organization and communication algorithms have been developed and described by Adam Spirer in his thesis [1].

Accordingly, the current work focuses on the development of a machine level programming environment, and register-transfer level simulator, for a massively parallel architecture. The register-transfer level simulator assumes zero-latency communications. Initially, the method of programming for the aforementioned architecture was performed in a C like, text based format, but this quickly proved inadequate. The programming environment proposed as a result of these inadequacies, instead uses both text and graphics to layout

program flow and register level information.

The organization of the architecture, including communications and operation, is covered in Chapter 2, followed by the description of the programming environment and simulation model of the architecture in Chapter 3. Experimental results are shown and discussed in Chapter 4, and development of the programming environment and simulator are described in Chapter 5.

Chapter 2

Architecture Organization

An introduction to the architecture has been given in this chapter to provide the reader a basis of what the machine level program development environment and register-transfer level simulator aims to model. The chapters following this overview require this basis for a solid understanding of the implemented model. The work introduced in this chapter describing the development of the architecture is primarily the focus of Adam Spirer's thesis [1].

2.1 Architecture Overview

The proposed architecture is comprised of a sea of atomic processors, or atomic processing elements, arranged in an orthogonal structure, as shown in Figure 2.1. All atomic processors are physically and functionally identical, but operationally independent. Each atomic processor is a low-complexity processing element, capable of:

- Storing and executing one instruction, and storing its associated operands and result, or
- Storing a data structure element comprised of one data word.

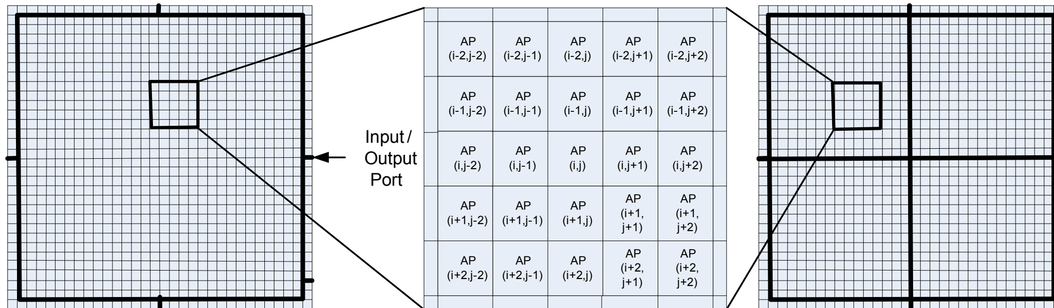


Figure 2.1: The sea of atomic processors. Each atomic processor can communicate with its eight adjacent neighbors. The input / output ports, illustrated in bold lines, can be located at the periphery, or can penetrate the sea.

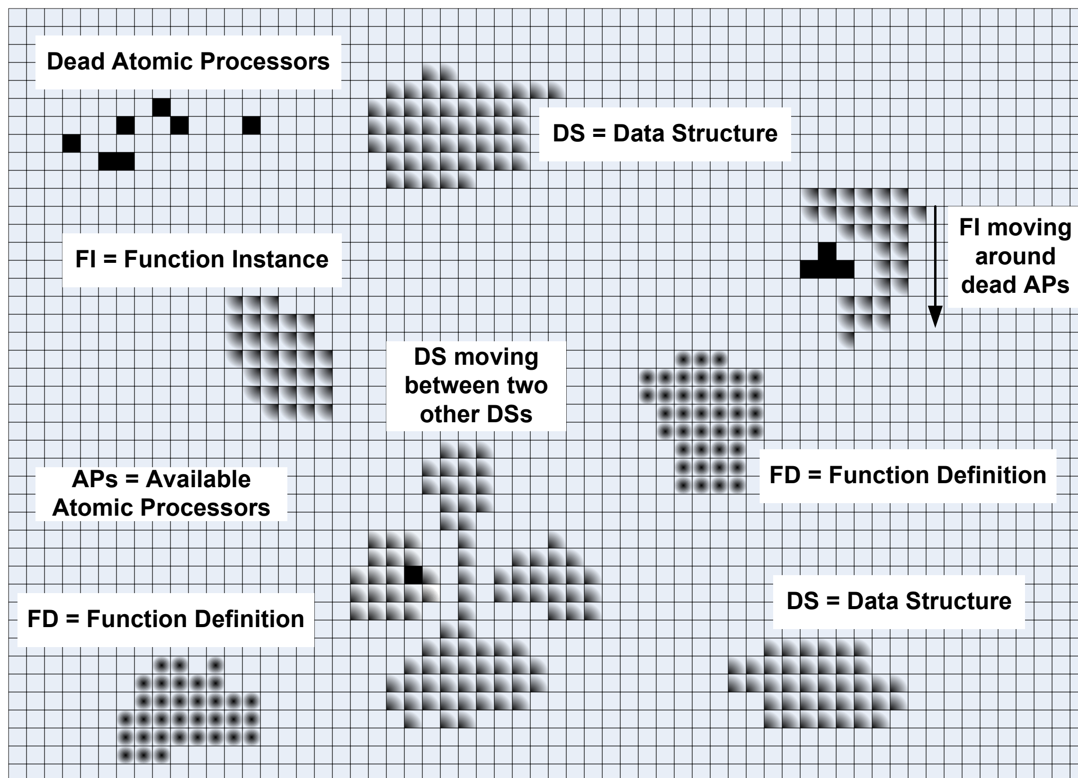


Figure 2.2: A snapshot of the sea of atomic processors. Programs are broken down into functions. A function copy is stored in a *function definition* entity. When a function is called, the function definition creates a copy, called a *function instance*. This moves away and abuts with the *data structure* entity it has to process.

In the sea of atomic processors, programs and data structures are organized as morphological entities, as shown in Figure 2.2. These can move, abut, and dissolve. As in the C programming language, programs are broken down into functions. A function's code is stored in a function definition. When a function is called at runtime, the function definition creates an instance of the function. The function instance moves away, and eventually abuts with the data structure that was passed to it by the calling function instance, before commencing execution. Upon completing execution, the called function instance returns a value or the result of processing in the form of a data structure.

Thus, in the sea of atomic processors as shown in Figure 2.2, we distinguish the following morphological entities:

1. Function Definitions - FD. These are functions / programs not actively associated with any data set or structure. An element of a function definition entity is an instruction word and its associated operand(s). In conventional architectures, these entities are equivalent to copies of programs stored on hard disk or other high capacity storage media. These entities also include functions, which assist with input / output operations, and system level housekeeping. In conventional architectures, these are equivalent to operating system functions.

2. Function Instances - FI. These are runtime instances of function definitions, which can be or are already actively associated with a data set or structure. An element of a function instance is an instruction word and its associated operand(s) and result. In conventional architectures, these entities are equivalent to copies of programs loaded at runtime into main memory.

3. Data Structures - DS. These are collections of associated data elements, which would be processed by the same function instance. An element of a data structure is a data word. Data structures can be as simple as one-dimensional arrays, or as large and

complex as files on a hard disk or other high capacity storage media in conventional architectures.

Not classified as morphological units, *Dead Atomic Processors* are individual or groups of atomic processors, which have been rendered not functional as a result of one or more internal faults.

The architecture is inspired from the field of microbiology. The morphological entities in the sea of atomic processors shown in Figure 2.2, move, change shape, and adapt like microorganisms in a medium. The shape of an entity may change in the course of a normal move, a move around other entities or dead atomic processors, or to group together associated elements.

Each atomic processor can store at the same time a function definition element, and a function instance or data structure element. This breaks down the sea of atomic processors into two functional layers: the definition layer, in which function definition elements are stored, and the execution layer, in which function instance or data structure elements are stored. In addition, in each layer the atomic processor stores configuration information associated with each element.

The storage requirements for an atomic processor are summarized in Table 2.1. We show for reference the size of each field in bits for a $64K$ (2^{16}) and $1T$ (2^{40}) seas of atomic processors. The $64K$ sea of atomic processors is typical for a computer system used in an embedded application, while the $1T$ is typical for a computer system used in desktop applications. The morphological entity type indicates the momentary functional role of the atomic processor. The hardware identification number contains the x and y coordinates of the atomic processor, and does not change during the lifetime of the system. The entity identification number specifies the entity to which the element belongs, and the element

identification number identifies the element within the entity. The target entity identification number and (x, y) coordinates are used during the move and abut processes, which are described in Sections 2.4 and 2.6, respectively. The primary and secondary operand source identification numbers indicate which element within the entity will provide the value of these operands. Either of these can also be initialized. There can be up to 16 data types, of which we currently encode: Boolean, character, character string, unsigned and signed integers of 16, 32, and 64 bits, single and double precision fixed and floating point numbers. The *previous execution order*, *execution order*, *execution count* and *execution count identification number* are used for execution flow control, as described in Sections 2.7 and 2.9. The operation code distinguishes between 256 different possible operations in the instruction set. Each instruction can be executed unconditionally or conditionally on the value of four status bits. Their true and complemented values are stored in the execution conditions field. The values of the status bits are produced by the element whose identification number corresponds to the status bits source identification number.

2.2 Instruction Set

The reduced instruction set is listed in Table 2.2. To meet the self-imposed requirement that each atomic processor has to be a low complexity processing element, we have included in the instruction set only fundamental arithmetic, logic, and control flow instructions. Complex or compound operations like multiplication or division are implemented as functions, as shown in Chapter 4.

Table 2.1: Information stored by one atomic processor, and its significance. Storage requirements, in bits, for $64K$ and $1T$ seas of atomic processors.

Name	**64K**	**1T**	**Value**
METype	2	2	Morphological Entity Type (element) 0=Unoccupied; 1=FD ; 2=FI ; 3=DS
Hardware ID	16	40	(x, y) coordinates of the atomic processor
Entity ID	16	40	Entity element identification number
Element ID	16	40	Entity element identification number
Target Entity ID	16	40	Target entity identification number used during the move and abut processes.
TargetEntityXYCoordinates	16	40	Target entity (x, y) coordinates used during the move and abut processes.
PrimaryOperandSourceID/ DataWordID	16	40	*FD/FI:* Primary operand source identification number; corresponds to an associated DS element data word ID
SecondaryOperandSourceID	16	40	*FD/FI:* Secondary operand source identification number; corresponds to an associated DS Entity ID or Element ID
PrimaryOperandType/ DataWordType (Execution)	4	4	*FD/FI:* Primary operand type; DS: Data word type
SecondaryOperandType (Execution)	4	4	*FD/FI:* Secondary operand type
PrimaryOperandValue/ DataWordValue (Execution)	32	64	*FD/FI:* Primary operand value; DS: Data word value
SecondaryOperandValue (Execution)	32	64	*FD/FI:* Secondary operand value
PrevExecutionOrder	16	32	*FD/FI: ExecutionOrder* number, of element(s) that must execute before this element
ExecutionOrder	16	32	*FD/FI:* Element execution order number
ExecutionCount (Execution)	16	32	*FI:* How many times must this FI receive a result broadcast with a particular execution order number (from a previous instruction) before executing itself?
ExecutionCountID	16	40	*FI:* Associated DS Element ID that also stores the *ExecutionCount* value
StatusBitsSourceID	16	40	*FD/FI:* Element ID whose execution result produces the status bits for this instruction (compared against Execution Conditions)
SourceStatusBitsValues	8	8	*FD/FI:* Status bits values received from *StatusBitsSource*
OperationCode	8	8	*FD/FI:* Instruction/operation code
ExecutionConditions	8	8	*FD/FI:* Status bits (C, N, V, Z, !C, !N, !V, !Z) required for execution of instruction
ResultValue (Execution)	32	64	*FI:* Result value of last execution
ResultStatusBitsValues (Execution)	8	8	*FI:* Result status bits of last execution
Cast Type	8	8	
Total to Store: To move FD/FI element: To move DS element:	386 274 148	754 538 276	

Table 2.2: The reduced instruction set. To meet the self-imposed requirement that each atomic processor has to be a low complexity, processing element, we have included in the instruction set only fundamental arithmetic, logic, and control flow instructions.

Mnemonic	Operation Code	Operation
NOP	0x00	No operation.
ADDPS	0x01	Add primary and secondary operands.
ADDPC	0x02	Add carry and primary operand.
ADDPSC	0x03	Add primary operand, secondary operand, and carry.
SUBPS	0x04	Subtract secondary operand from primary operand.
SUBPC	0x05	Subtract carry from primary operand.
SUBPSC	0x06	Subtract secondary operand and carry from primary operand.
INC	0x07	Increment primary operand.
DEC	0x08	Decrement primary operand.
INV	0x09	Bitwise inversion of primary operand.
AND	0x0A	Bitwise AND of primary and secondary operands.
OR	0x0B	Bitwise OR of primary and secondary operands.
XOR	0x0C	Bitwise XOR of primary and secondary operands.
SETC	0x0D	Explicitly set 'carry' flag.
SETZ	0x0E	Explicitly set 'zero' flag.
SETN	0x0F	Explicitly set 'negative' flag.
SETV	0x10	Explicitly set 'overflow' flag.
RSTC	0x11	Explicitly reset 'carry' flag.
RSTZ	0x12	Explicitly reset 'zero' flag.
RSTN	0x13	Explicitly reset 'negative' flag.
RSTV	0x14	Explicitly reset 'overflow' flag.
SHL	0x15	Shift left primary operand, pad with zeros.
SHR	0x16	Shift right primary operand, pad with MSB.
SHLC	0x17	Shift left primary operand through carry, pad with zeros.
SHRC	0x18	Shift right primary operand through carry, pad with MSB.
CALL	0x19	Function call instruction; requests the function definition whose identification number is stored in the primary operand value to create an instance; the created function instance will process the data structure whose identification number is stored in the secondary operand value; the execution of this instruction completes when it receives a RETURN result from the called function instance.
RETURN	0x1A	Function return instruction; broadcasts return value of function instance to the corresponding CALL instruction in the calling function instance; the primary operand value holds the data word to be returned, and the secondary operand value holds the entity identification number of the calling function instance.

2.3 Communications

Communications in the sea of atomic processors are constrained to the Moore neighborhood, i.e. each atomic processor can only communicate with its eight adjacent neighbors. This self-imposed design constraint is justified by the desire to eliminate the need for long interconnects, which in nanometer technologies scale slower than devices. As a consequence, communication latency becomes a key factor that affects the performance of the architecture. To minimize the communication latency, we have developed a minimum overhead, custom communications protocol.

Communications are used to pass messages, which contain operational information and/or data. In the sea of atomic processors, messages can be broadcast in all directions, i.e. over the entire sea of atomic processors (global cast), or just to a specific entity (beamed cast). Within an entity, messages are sent to all elements of the entity (entity cast), to all eight neighbors (local cast), or to only one neighbor (point-to-point cast).

From a hardware point of view, communications use eight handshake buffer pairs, which are used to exchange handshake codes, as shown in Table 2.3, and two message buffers, which are used to exchange messages, as shown in Figure 2.3. The transfer of a message from atomic processor x to atomic processor y is called hereafter a communication cycle, and is illustrated graphically in Figure 2.4. It proceeds as follows:

1. Atomic processor y resets its receiving handshake buffer once it is ready to receive a handshake code. This does not mean it is ready to receive a message yet.

2. Atomic processor x repeatedly checks the receiving handshake buffer of atomic processor y. When it sees that the latter has been reset, it uploads the handshake code of the message it wants to send.

3. Atomic processor y checks the uploaded handshake code, and compares it to other

handshake codes it may have received from other neighbors. This comparison is used to prioritize transfers, where handshake code 1 in Figure 2.6 has the highest priority. Once it is ready to receive the message from atomic processor x, atomic processor y resets its receiving handshake buffer again.

4. Atomic processor x checks repeatedly the receiving handshake buffer of atomic processor y. When it sees that it has been reset again, it transfers the message into the message buffer of atomic processor y.

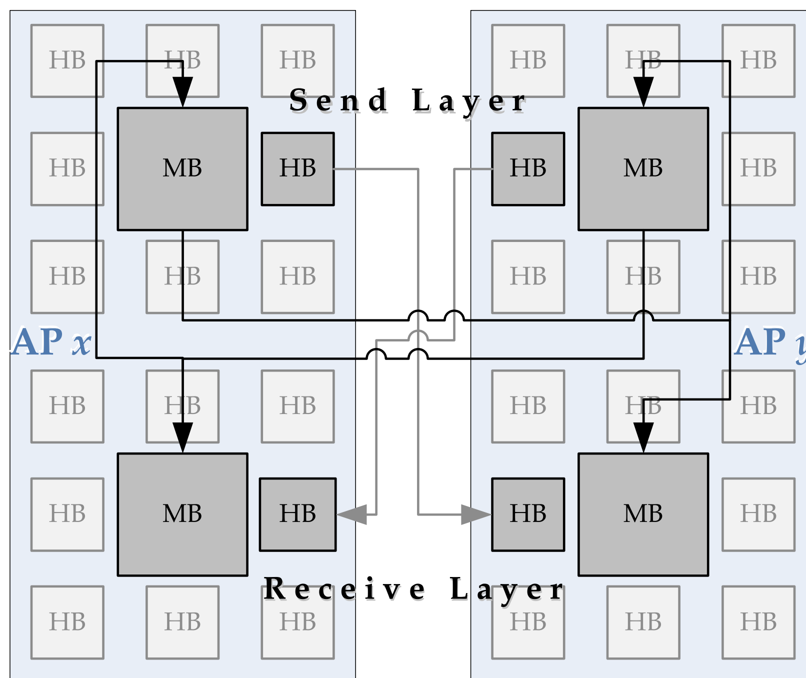


Figure 2.3: The communications hardware is divided into a send and receive layer. Each layer uses a set of 8 handshake buffers and one message buffer. This allows full-duplex message passage.

The handshake code received by an atomic processor is used for two purposes: first, to prioritize transfer requests received in the same cycle from different neighbors, and second, to determine how the message will be processed. There are 16 different handshake codes,

Table 2.3: Communications handshake codes.

Code	Operation	Description
0 (0000)	AP is ready	The atomic processor is ready to receive the next handshake code.
1 (0001)	Global cast	The message has to be propagated throughout the entire sea of atomic processors.
2 (0010)	Beamed cast	The message has to be propagated to a specific location in the sea of atomic processors.
3 (0011)	Entity cast	The message has to be propagated only inside an entire entity; the propagation of the message will end at the edge of the entity.
4 (0100)	Local cast	The message has to be sent only to the atomic processor's eight neighbors.
5 (0101)	P2P cast	The message has to be sent only to a single neighbor.
6 (0110)	Function Definition Move request	The message is a P2P cast, in which a function definition element request to move into a neighboring atomic processor.
7 (0110)	Function Instance Move request	The message is a P2P cast, in which a function instance element request to move into a neighboring atomic processor.
8 (0110)	Data Structure Move request	The message is a P2P cast, in which a data structure element request to move into a neighboring atomic processor.
9 (0111)	Abut request	The message is a P2P cast, in which an abutment between two associated entities is requested.
10-14	Reserved	Reserved for future expansion
15 (1111)	AP is not functional	If an AP is deemed non-functional, then its receiving handshake buffers are all set to 1111; this indicates to its neighbors that it is not functional and not able to communicate.

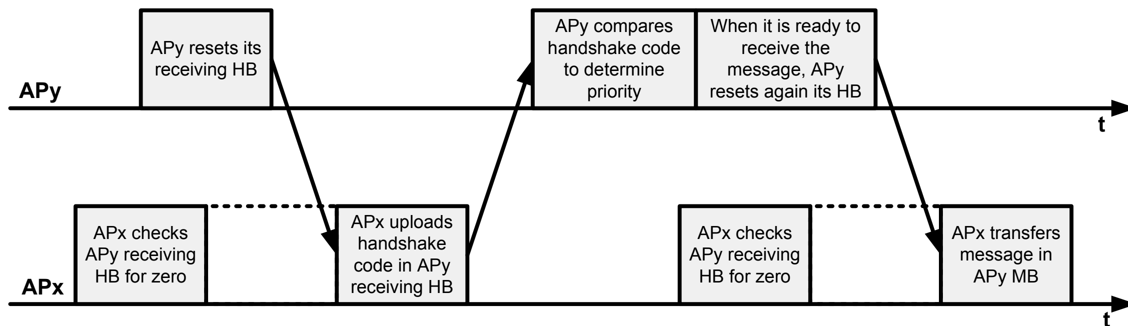


Figure 2.4: A complete communications cycle. Atomic processor x is transmitting a message to its neighbor, atomic processor y.

which are shown in Table 2.3. Code 0 indicates the atomic processor's readiness to accept the next handshake code from its neighbor. Codes 1-5 encode the five different cast types. Codes 6-8 are used during the move process, and code 9 is used during the abut process. Code 15 is used to indicate that the atomic processor is not functional. The remaining codes are currently reserved for future extensions.

Each of the two communication layers shown in Figure 2.3 comprises one set of handshake buffers and one message buffer. Thus, an atomic processor can receive and send a message at the same time. This eliminates the possibility of lockup if both atomic processors try to send a message to each other at the same time.

To avoid duplicate transmissions of the same message to the same atomic processor, during a global, or beamed, or entity cast, a message is routed according to the scheme presented in Table 2.4. For example, if the message is received from the neighbor to the north, it is sent to the south, southeast and southwest neighbors. Messages received from east, south and west are routed in a similar way. Alternatively, if for example the message is received from the neighbor to the northeast, it is sent to the southwest neighbor only. The routing and propagation of a global cast message is shown in Figure 2.5. This is also applicable to an entity cast, except the propagation stops at the edge of the entity. The routing and propagation of a beamed cast, along with other examples of global and entity

casts are shown in Figure 2.6. In a beamed cast, the maximum number of atomic processors the message has to pass through is equal to $\max\{|x_s - x_r|, |y_s - y_r|\}$, where x_s, y_s, x_r, and y_r are the (x, y) coordinates of the sending and receiving atomic processors.

Table 2.4: Message routing scheme used during a global or entity casts.

Receive From	**Transmit To**							
	N	**NE**	**E**	**SE**	**S**	**SW**	**W**	**NW**
N				✔	✔	✔		
NE						✔		
E						✔	✔	✔
SE								✔
S	✔	✔						✔
SW		✔						
W		✔	✔	✔				
NW				✔				

We have identified the need for two message formats, as shown in Table 2.5a and 2.5b. A type 0 or result message is entity cast either by a function instance element or a data structure element. The former entity casts it after it has executed its instruction. The latter entity casts it during the execution initialization phase, which follows the abutment with the associated function instance. Message types are described in Table 2.6, and their usage explained in Sections 2.4 through 2.6.

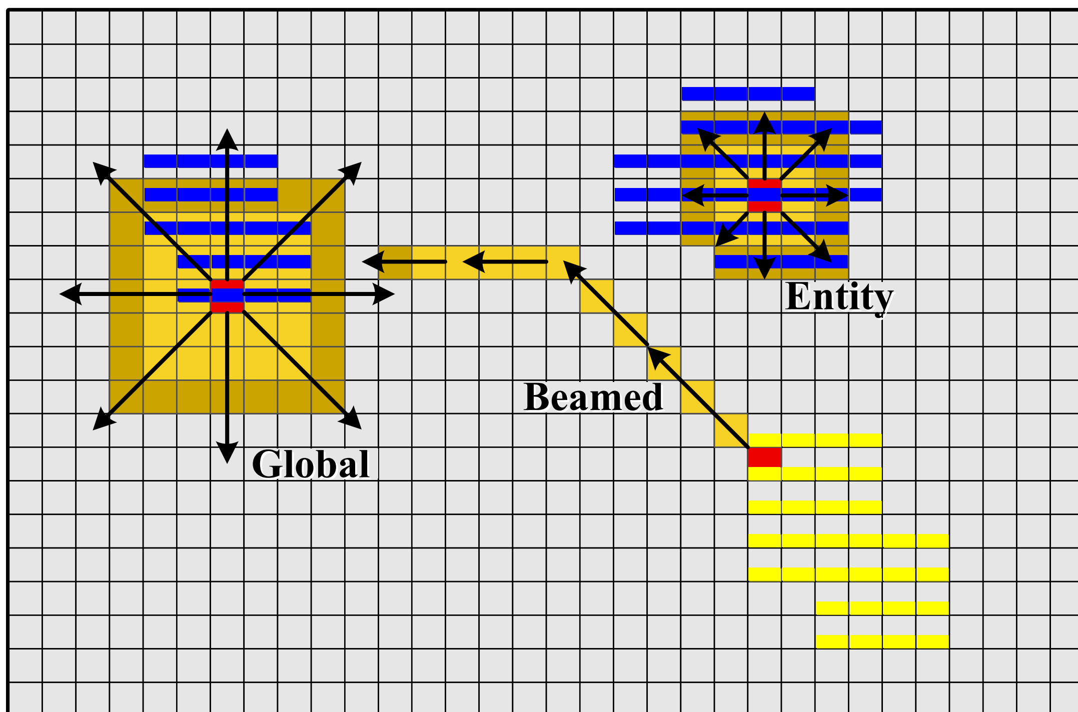


Figure 2.5: Routing and propagation of a beamed, entity, and global cast messages. In a beamed cast, the maximum number of atomic processors the message has to pass through is equal to $\max\{|x_s - x_r|, |y_s - y_r|\}$, where the sender is at (x_s, y_s) and the receiver is at (x_r, y_r). In an entity cast the propagation stops at the edge of the entity.

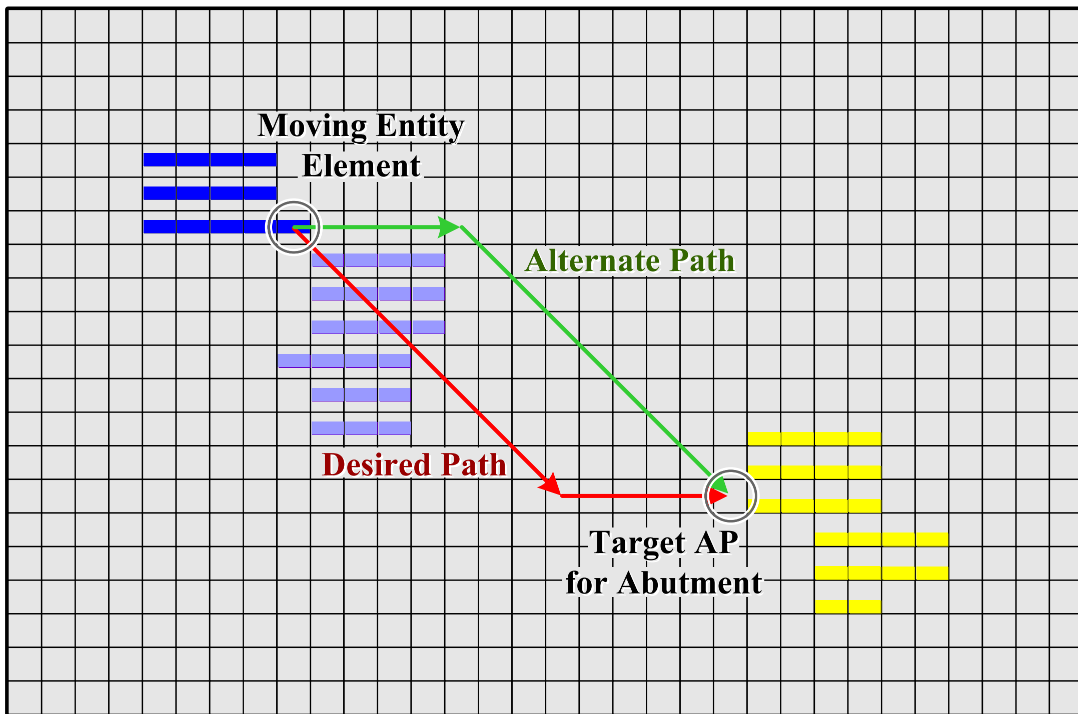


Figure 2.6: A moving entity is forced to move around another entity. Elements of the moving entity move counterclockwise as many times as needed, and correct their course as soon as possible afterwards.

Table 2.5: Message formats. The numbers of bits in the shaded boxes change with the size of the sea of atomic processors and/or size of operands. The message codes are described in Table 2.6. ID = Identification number; S = Source; D = Destination; I = Intermediate.

a) Result Message Format

Sea Size	**Message Type**	**Message Source ID**	**Result Value**	**Primary/ Secondary Operand**	**Execution Order Number**	**Status Bits**	**Reserved**	*Total Bits*
$64K$	4	16	32	32	16	8	20	128
$1T$	4	40	64	64	40	8	44	256

b) Message Formats for types 1-11

Sea Size	**Message Type**	**Message Source ID**	**Message Dest. ID**	**Intermediate ID**	**ID Type: S/D/I**	**Reserved**	*Total Bits*
$64K$	4	16	16	16	2 / 2 / 2	70	128
$1T$	4	40	40	40	2 / 2 / 2	126	256

2.4 Entity Movement

An entity element moves from atomic processor x into atomic processor y as follows:

1. Atomic processor x requests to move an entity element into atomic processor y by setting its handshake buffer to code 6, if the element is a function definition element, or to code 7 if the element is a function instance element, or to code 8 if the element is a data structure element, as described in Table 2.3.

2. Atomic processor y resets its handshake buffer back to 0 if it is ready to receive the new element. It can only do so if it is not storing any other similar entity element.

3. Once the move request is accepted, atomic processor x transfers the entity element information through the message buffer.

The move is further triggered and sustained through the use of the *Move to Abut* message type, shown in Table 2.6. As calculated in Table 2.1, for a function definition or

function instance element in the $64K$ and $1T$ seas of atomic processors, 274 bits and 538 bits, respectively, have to be transferred. The size of the message buffers being 128 bits and 256 bits, respectively, three transfers through the message buffer are necessary, and therefore steps 1 - 3 are repeated three times for these kinds of elements. A data structure element needs to transfer 148 bits and 276 bits, respectively, and therefore steps 1-3 are executed twice.

In the course of a move, an entity element may encounter unavailable atomic processors. In this case, it first moves in an alternative direction, and then corrects its course by moving again towards the target location. For example, in Figure 2.6 the moving entity element encounters an unavailable atomic processor to the southeast. Then it tries the next counterclockwise neighbor. In the example, the neighbor to the east is available, and it moves into it. Then it tries again to move southeast but encounters another unavailable atomic processor. Thus, it tries the next counterclockwise neighbor, and so on. Finally, after five moves due east, it can turn southeast and move towards the target direction. If the front of the moving entity gets stuck, the back is forced to move around and become the new front. Thus, entities will not get stuck during moves.

2.5 Input/Output

Two special kinds of moves are the input and output of entities into the sea of atomic processors. These occur through atomic processors which are connected to the input / output ports, illustrated in bold in Figure 2.1. The input of function definition entities is not time critical. Thus, it can be performed serially, following the steps illustrated in Figure 2.7. As for data, the amount and speed to be input and/or output are application dependent. In these cases, an entity may transfer via several input / output ports and associated atomic processors.

Table 2.6: Message codes and descriptions for non-result message types.

Code	Function	Description
0x0	Result	The result entity cast by a function instance element after it has executed its instruction; or by a data structure element during the initialization phase, following abutment with the associated function instance.
0x1	Instantiate Function	Beamed cast by a function instance element executing a CALL instruction; it instructs the function definition, if it exists, to create a function instance of itself in the execution layer; after it has been created, the latter moves away, abuts with the data structure it has to process, and commences execution.
0x2	Instantiate Function (Interrupt)	Similar to 0x01, except the source of the call is an interrupt source.
0x3	Return Value	Beamed cast by the called function instance to the calling function instance, after it has completed execution.
0x4	Return Value (Interrupt)	Similar to 0x03, except the returning function is an interrupt service routine.
0x5	Move to Abut	Triggers elements of an entity to continue moving to abut.
0x6	Abutment Completed	Entity cast by a function instance element once it has abutted with an element of the associated data structure entity.
0x7	Terminate Function Instance Entity	Entity cast by the function instance element that executes the RETURN instruction, after it has beamed cast the return value.
0x8	Request Location of Function Definition	Global cast by a function instance element executing a CALL instruction; it requests the (x, y) Coordinates or Hardware ID of the function definition it wants to call.
0x9	Acknowledge Location of F.D.	Beamed cast by a function definition element in response to 0x8.
0xA	Request Location of Data Structure	Global cast by a function instance element executing a CALL instruction; it requests the (x, y) Coordinates or Hardware ID of the data structure the called function will have to process.
0xB	Acknowledge Location of D.S.	Beamed cast by a data structure element in response to 0xA

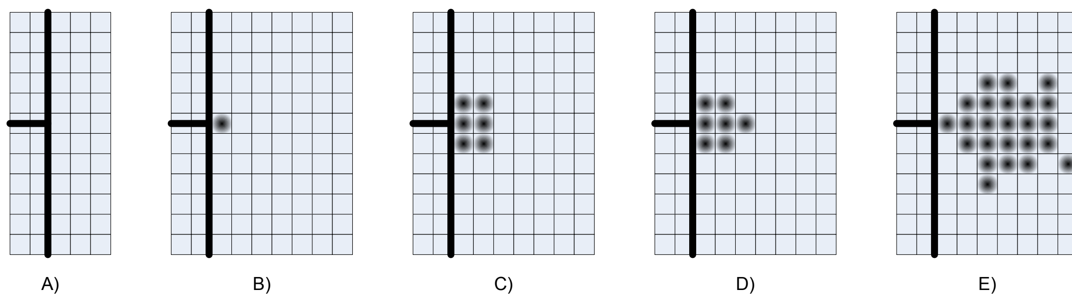


Figure 2.7: An entity enters the sea of atomic processors, when viewed from $a \rightarrow f$, or exits the sea of atomic processors, when viewed from $f \rightarrow a$. Large entities may be input or output via multiple ports and associated atomic processors.

2.6 Entity Abutment

After moving diligently towards the target location, at some point a function instance element encounters a data structure element. The function instance element initiates an abut request, using handshake code 9 and a point-to-point cast, and provides its entity identification number. In return, the data structure element provides its entity identification number. If the identification numbers match, the function instance element, entity casts an abutment complete message, type 6 in Table 2.6. Because the move and abutment processes are asynchronous, multiple function instance and data structure elements may abut at the same time, so multiple abutment complete messages may be entity cast. The function instance entity abutted to its associated data structure entity will hereafter be called a superentity. Once they receive the abutment complete message, all data structure elements entity cast their current data word values over the superentity. These are then received by all function instance elements which have to update their primary and/or secondary operand values. This means that before execution commences, all function instance elements have their operands available, less the values updated at runtime.

2.7 Instruction Execution

Once abutment completes, and all data structure elements entity cast their data word values, the first instruction executes as soon as it has its operands available. At run-time, all function instance elements monitor all entity casted result messages. If the message source identification number matches one of their own source identification numbers, they update their primary operands, and/or secondary operands, and/or source status bits values. At the same time, they check the *ExecutionOrder* number in the result entity cast message. If the received *ExecutionOrder* number matches their own *PrevExecutionOrder* number, and

all operand and source status bits values are available, the function instance element executes the instruction. The *ExecutionOrder* and *PrevExecutionOrder* numbers, which are statically assigned, are the means through which the architecture implements the necessary execution flow control. The operand values, source status bits values and *ExecutionOrder* number may arrive in the same or different result entity cast messages, and in arbitrary order. The instruction cycle completes when the function instance element entity casts the result of its instruction execution.

There are obviously no structural dependencies. Data and control dependencies are explicit, and therefore automatically resolved. Instruction level parallelism is exploited dynamically and it is maximized, because an instruction executes as soon as it has its operands, source status bits, and knowledge of the fact that the previous instruction in the execution flow has executed.

At runtime, all result messages are entity cast over the entire superentity, and so also reach all data structure elements. A data structure element updates its data word value when there is a match between the message source identification number and its element identification number. This means that when the function instance completes execution and is ready to return the data structure, the latter is up to date.

All instructions defined in Table 2.2 execute as described above, except for CALL and RETURN, which are described in the next section.

2.8 Function Calls

A function call involves four entities: the calling function instance, the called function definition and its instance, and the data structure to be processed. A complete function call and return cycle comprises the following steps:

1. An element of function instance X executes the CALL instruction, by asking the

function definition Y of the called function to create an instance of itself, which should move to location (x_Z, y_Z) and process data structure Z. If the calling function instance X knows the location of the function definition Y, it uses a beamed cast. Else, it uses a global cast. In either case, it provides its own (x_X, y_X) coordinates.

2. Function definition Y replies with a beamed cast, in which it provides its current coordinates and a confirmation that the function instance Y_i has been created and moves towards data structure Z. Function definition Y instantiates Y_i by simultaneously creating a copy of its elements from the definition layer into the execution layer.

3. Function instance Y_i moves to, abuts with, and processes data structure Z, as described in the previous four sections.

4. The last instruction executed by function instance Y_i is a RETURN. The element of function instance Y_i that executes the RETURN, beam casts a *return value* message, see Tables 2.5 and 2.6, to the calling function instance X. It also entity casts a *terminate function instance entity* message, if the function instance is no longer needed.

During the entire call-return cycle, the element of function instance X that executes the CALL instruction cannot move, because it has to receive the RETURN beamed cast. Multiple CALL instructions may call for multiple instantiations of the same function definition, to process different, independent data structures.

Input / output port requests or interrupts are handled in the same way, except for the fact that the calling function is actually an input / output port. If and which currently running function instances are halted, or affected by the exception, is decided by the function instance that services the exception.

2.9 Loops

If there are no data and/or control dependencies between the iterations of a loop, they are executed concurrently, as shown in Figure 2.9a. In this case, each loop iteration is implemented by a separate set of elements or function instance entities. These iterations complete out-of-order and the last instructions of all iterations will entity cast the same *ExecutionOrder* number. This is received and counted by the first instruction after the loop. When the count value is equal to the *ExecutionCount* value, it proceeds to execution and the loop is completed. Double counting is eliminated because the same entity cast cannot reach the same element twice, as is illustrated in Figure 2.8. If there are dependencies, loop iterations have to be executed in sequence, as shown in Figure 2.9b. In this case, it is not effective to create or instantiate elements or function instances for each iteration of the loop. The single instance of the iteration is executed repeatedly. The last instruction in the iteration, which can also manipulate the index, executes if the condition to repeat is true. It entity casts the *ExecutionOrder* number of the instruction before the loop, triggering the instructions in the loop to execute again. The first instruction after the loop executes if the condition to repeat is false. This is equivalent to a *for loop* in which the first iteration is always executed. Alternatively, the condition can be tested by the first instruction of the iteration, which creates the possibility to skip the execution of the loop altogether.

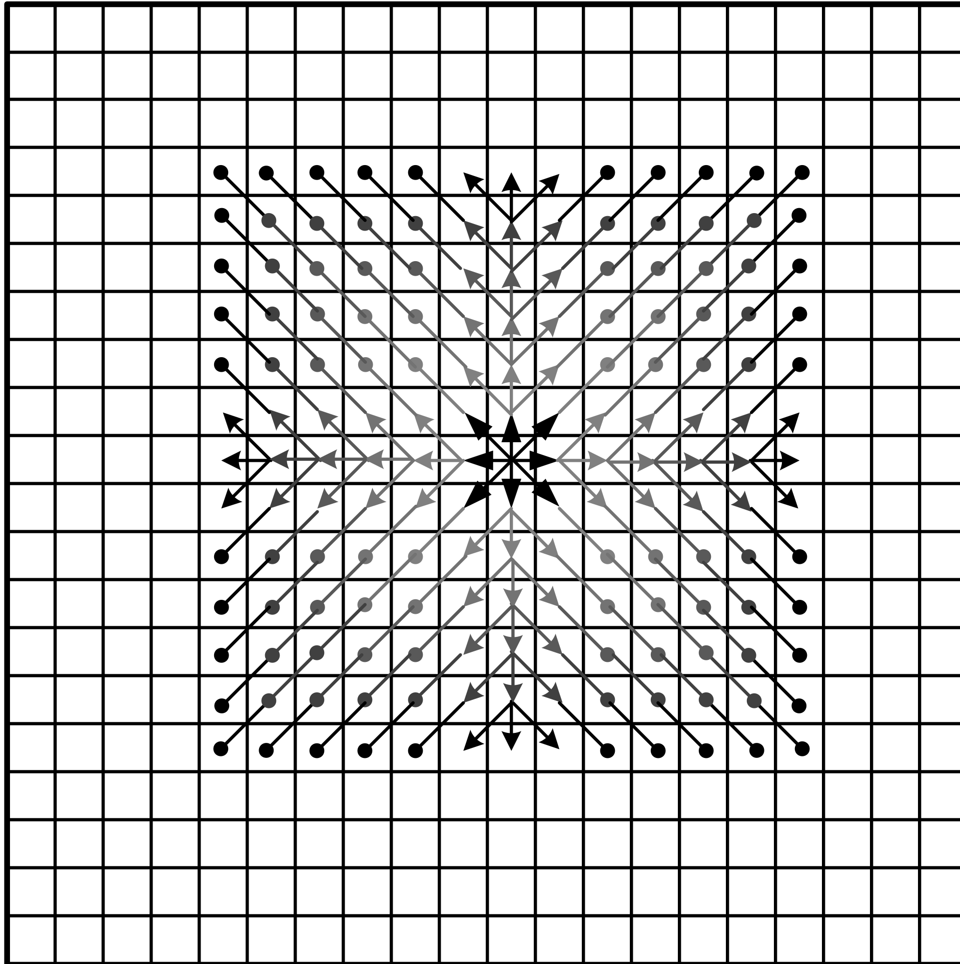


Figure 2.8: Routing and propagation of a global cast message. Using the scheme in Figure 2.9, no atomic processor will receive duplicates of the message.

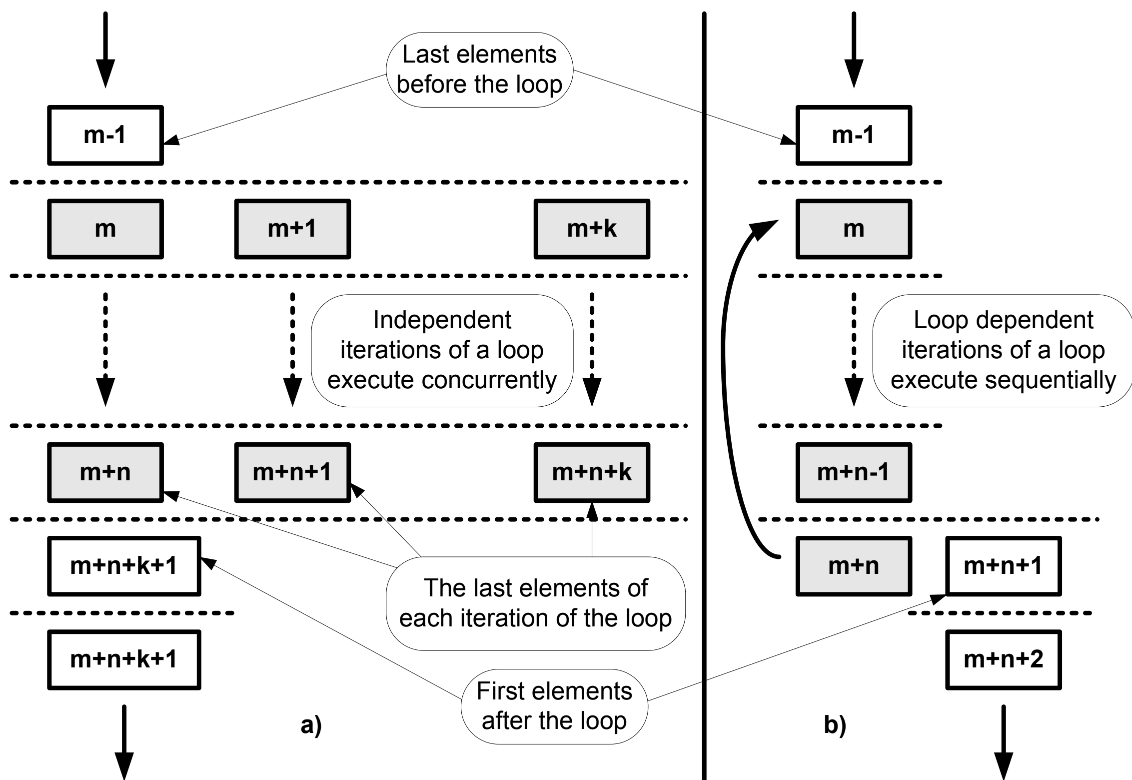


Figure 2.9: The execution of a loop without loop carried dependencies (a), and with loop carried dependencies (b). The gray shaded function instance elements are part of the loop iteration.

Chapter 3

Programming Environment and Simulation Model

This chapter describes the model of the architecture, which was described in the previous chapter. This model is used in the integrated development environment and simulator. Furthermore, justification for several of the software design decisions are introduced here and further discussed in Chapter 5. Finally, a short discussion has been provided describing the creation of a function definition in the IDE.

3.1 Integrated Development Environment GUI

As our computing environments continue to look for ways to exploit more concurrency and focus less on executing a sequential string of commands, programming these upcoming massively parallel architectures is a challenge. Code development for a massively parallel architecture such as the one discussed in Chapter 2, is a difficult task when presented with the choice of the standard text based coding languages such as C, C++ and Visual Basic, just to name a few, which primarily are sequential in nature. Much work has been done to create concurrent programming languages in addition to using APIs such as the *Message Passing Interface* (MPI), OpenMP and POSIX Threads with standard sequential

languages [77–79]. An alternate approach is a compiler designed to automatically extract parallelism from sequential code, but such compilers have very limited success and are typically aimed toward specific applications [80]. Many of these parallel programming techniques are designed for a limited number of processors or processors that are far more complex then the RISC processors in the architecture examined in this research. Because of this, an alternate programming method and environment are required.

Hence, an initial approach at programming the massively parallel architecture was attempted using a C like, text based format. It was quickly discovered that this format was cumbersome and lacked an easy way to convey important information, such as program flow control. Each time a change was desired, even in a simple program, it had to be made in several locations throughout the code. In anything more than a simple algorithm this meant that thousands of lines of code would have to be browsed through. Figure 3.1 shows an example of this text based code for one of the example algorithms discussed later. Ultimately, the limitation of a text only programming language is that text conveys sequential information.

In contrast, a graphical programming environment allows for multiple threads of a program to be displayed simultaneously. An example of this is provided by Mathworks' Simulink, in which several dynamic systems can be graphically diagrammed within the same project. While Simulink offers several libraries for control theory and digital signal processing, its use in implementing a programming environment for a massively parallel computer architecture would require the development of a custom library and significant add-ons. Similarly, one of the several commercially available schematic capture programs would serve adequately for diagramming the model of a massively parallel program. However, these tools would still require an add-on for simulation purposes.

Furthermore, programming tools provided for *field programmable gate arrays* (FPGAs) and *complex programmable logic devices* (CPLDs), by manufacturers such as Altera and

```
                                multiplier32.txt
Begin Concurrent
[ Input.0;
        ExecutionOrder#          = 0;
        ExecutionOrder#Count     = 0;
        ExecuteAfterOrder#       = 0;
        StatusBitsSource         = ;
        StatusBitsValues         = C=False N=False V=False Z=False;
        ConditionalExecution     = Unconditional;
        PrimaryOperandSource     = ;
        PrimaryOperandType       = Unsigned Integer;
        PrimaryOperandValue      = 131072;
        SecondaryOperandSource   = ;
        SecondaryOperandType     = ;
        SecondaryOperandValue    = ;
        Operation                = NOP Unconditional;
        Cast                     = Entity IdentificationNumber ExecutionOrderNumber
StatusBits PrimaryOperand;
]
[ Input.1;
        ExecutionOrder#          = 0;
        ExecutionOrder#Count     = 0;
        ExecuteAfterOrder#       = 0;
        StatusBitsSource         = ;
        StatusBitsValues         = C=False N=False V=False Z=False;
        ConditionalExecution     = Unconditional;
        PrimaryOperandSource     = ;
        PrimaryOperandType       = Unsigned Integer;
        PrimaryOperandValue      = 10000;
        SecondaryOperandSource   = ;
        SecondaryOperandType     = ;
        SecondaryOperandValue    = ;
        Operation                = NOP Unconditional;
        Cast                     = Entity IdentificationNumber ExecutionOrderNumber
StatusBits PrimaryOperand;
]
End Concurrent
[ AtomicProcessor.1;
        ExecutionOrder#          = 0;
        ExecutionOrder#Count     = 0;
        ExecuteAfterOrder#       = 0;
        StatusBitsSource         = ;
        StatusBitsValues         = C=False N=False V=False Z=False;
        ConditionalExecution     = Unconditional;
        PrimaryOperandSource     = i0;
        PrimaryOperandType       = ;
        PrimaryOperandValue      = ;
        SecondaryOperandSource   = ;
        SecondaryOperandType     = Unsigned Integer;
        SecondaryOperandValue    = 1;
        Operation                = AND Unconditional;
        Cast                     = Entity IdentificationNumber ExecutionOrderNumber
StatusBits PrimaryOperand;
]
Begin Concurrent
[ AtomicProcessor.2;
        ExecutionOrder#          = 0;
        ExecutionOrder#Count     = 0;
        ExecuteAfterOrder#       = 0;
        StatusBitsSource         = 1;
        StatusBitsValues         = C=False N=False V=False Z=False;
        ConditionalExecution     = Unconditional;
        PrimaryOperandSource     = ;
        PrimaryOperandType       = Unsigned Integer;
        PrimaryOperandValue      = 0;
        SecondaryOperandSource   = i1;
        SecondaryOperandType     = ;
        SecondaryOperandValue    = ;
        Operation                = ADDPS Z=0;
        Cast                     = Entity IdentificationNumber ExecutionOrderNumber
StatusBits PrimaryOperand;
                                     Page 1
```

Figure 3.1: Text based code for the first four tokens of the 32-bit multiplier shown in Figure 4.7.

Xilinx exhibit similar problems. Logical models of processors in the proposed architecture could be created in these tools and simulated, but computing needs would be excessive. The shortcomings of these tools leads to the conclusion that a custom environment is needed.

Moreover, a common textbook approach to modeling a parallel program is the *dependency graph* [81]. A typical *dependency graph* shows the dependencies between objects and allows one to determine the order or lack of order that the objects must be evaluated in. This method is often used to aid in decisions about the appropriate program flow in concurrent algorithms. The proposed *integrated development environment* (IDE) allows the user to indicate concurrency and dependencies of each processing element for a given function or set of functions in a manner similar to a dependency graph.

Once the programmer considers an algorithm complete, the option is provided to parse the visual display and check for errors as well as provide warnings about potential mistakes. If the program is parsed without any errors, the program is simulated using a simulation engine designed to match the proposed architecture. The simulator does not account for communication delays between processors as would exist in actual hardware because of the large amount of resources required for this type of simulation. Instead, the communication latency is assumed to be zero, allowing for a purely functional simulator used for algorithm development. This is similar to how one may functionally simulate VHDL or Verilog code without hardware timing delays to get an idea of how a given algorithm works. Henceforth, the combined IDE and simulator will be referred to as the IDE for the remainder of the document.

The goals set forth for the prototype IDE are to show proof of concept for the proposed visual programming method and to functionally simulate simple algorithms, intended primarily for embedded applications, on the proposed architecture. A typical embedded application may differ from a desktop or server system in that it often has real time constraints for the processing of data and may be used in signal processing applications. Additionally,

these applications also quite frequently require that system memory and power consumption are kept to a minimum. In many cases, the only system memory is part of the processor chip itself [2]. One example of an embedded application is a cell phone; it requires real time processing of radio signals in a small, low-power package.

Accordingly, the user interface is intended to mimic a dependency graph in some regards and is based on a layout grid that the user can place tokens onto. The grid is arranged such that when the user places a token, it will snap to that grid location automatically. The grid is divided into rows and columns. A token is assigned an (x, y) location based on the row and column into which it has been placed. Each row signifies a concurrent step to be simulated so that all tokens on a given row execute in parallel. Columns provide no significance in the user interface beyond allowing each token to be assigned a unique location on the grid. There are several types of tokens that can be placed on to the layout grid. The token types include *atomic processors* (APs), *data elements* (DEs), *function calls* (Fns), *inputs*, *sequential arrays* (SAs) and *concurrent arrays* (CAs). The different types of tokens will be discussed in more detail later in this chapter, but for the purpose of this description, each token represents a single atomic processor, with the exception of the array tokens.

Once tokens are placed onto the design grid, properties for each token can be defined in a table, called the property grid as shown in Figure 3.2, by selecting the desired token. If a user indicates that a token should take one of its operands from another token, a line is drawn connecting the two tokens to indicate the data dependency. Additionally, information defining a processor's operation code, status bits, execution order and broadcast type are editable from the property grid. Currently, the execution order and broadcast information have no effect on the simulator, due to the exclusion of communication delays from the functional simulator and the implied program flow of the IDE's *graphical user interface* (GUI). If the IDE was further developed to compile code for a communication accurate simulator or actual hardware, these fields would likely be generated at compile time to

Atomic Processor 1 Hide

Property	Value
Broadcast	
Bcast_1stOp	True
Bcast_2ndOp	False
BcastStatus	True
CastType	Entity
Execute Order	
ExecuteAfter	0
ExecuteOrderCount	0
ExecuteOrderID	1
NextRowToExecute	2
Execution	
ConditionalExecution	Unconditional
ConditionalOperation	Unconditional
OperationCode	ADDPS
Operands	
ElementID	1
FirstOpSource	
FirstOpType	Unsigned Integer
FirstOpValue	10
SecondOpSource	
SecondOpType	Unsigned Integer
SecondOpValue	10
Status Bits	
CarryBit	False
NegativeBit	False
OverflowBit	False
StatusBitsSource	
ZeroBit	False

OperationCode
The op code for this processor.

Figure 3.2: IDE Property Grid

avoid the tedious and error prone process of populating them by hand.

3.2 Model of a Processor

Each processor is modeled by a single token placed on the layout grid. The basic building block of the architecture, an atomic processor, is modeled by an AP token. Each AP token is capable of performing a single arithmetic, logic, status bit or function return operation as discussed in Chapter 2. Function calls, while represented by a standard operation code are handled by a separate token in the IDE. An AP token provides the user with fields for primary and secondary operands which can be defined within the token by providing a type and value or received from another token on the layout grid. Status bit value and source fields are also provided to allow for conditionals within the token. As with operands, if another token is provided as a source for the status bits its values will override the internal values.

Additionally, conditional tokens can be used to make program flow decisions based on status bits passed from other processors or as defined within a processor. When a token is made conditional, by defining a condition in its property grid, it will change to yellow and display a conditional symbol on the face of the token. During a simulation conditional tokens change color based on the program flow. If a conditional token executes, it becomes green. If it does not execute, it becomes red.

Furthermore, there are two types of conditionals in the simulator: *conditional execution* and *conditional operation*. *Conditional execution* blocks the broadcast of any information from the token in the event the condition is not met. *Conditional operation* is non-blocking and allows a broadcast of information from the token, but does not perform the indicated operation on the operands when the condition is not met. A false *conditional operation* is analogous to a NOP, which broadcasts its primary operand unaltered as the result.

Finally, other fields included for an AP fall into categories of broadcast information and execute order fields. These fields are provided to the user solely for the export to human readable text function of the IDE. They have no effect on the simulator which handles execution order and message passing internally.

3.3 Specialty Tokens

In addition to the standard AP tokens, five other token types are available to the user, *Input Tokens*, *Function Calls*, *Data Elements*, *Concurrent Arrays* and *Sequential Arrays*. The different token symbols are shown in Figure 3.3. Input tokens are added to the IDE through the view menu by selecting the project properties dialog, currently the number of input tokens is limited to ten. Input tokens do not represent physical processors in hardware, but are a means for debugging a function definition during its development. Default values are provided by an input token so a function definition can be simulated without instantiating it from another Pond project. When a function definition is instantiated from another project the default values provided by the input token are instead overridden by values being passed into the function instance. Input tokens only provide fields for the default operand value and type, and input description and label fields to aid the user in selecting the proper inputs during a function call.

The remaining token types are added to the IDE the same way AP tokens are added, through the left hand side tool bar, shown to the left of Figure 3.3. Function call tokens allow the user to instantiate a function definition from within a project. Each function call token has a field for the .pnd file that the function definition is stored in. Upon populating this field, the file is read and the appropriate number of input source fields show up in the property grid interface for the user to define. Each input source field provides a label and description to the user in the help box at the bottom the property grid. While this

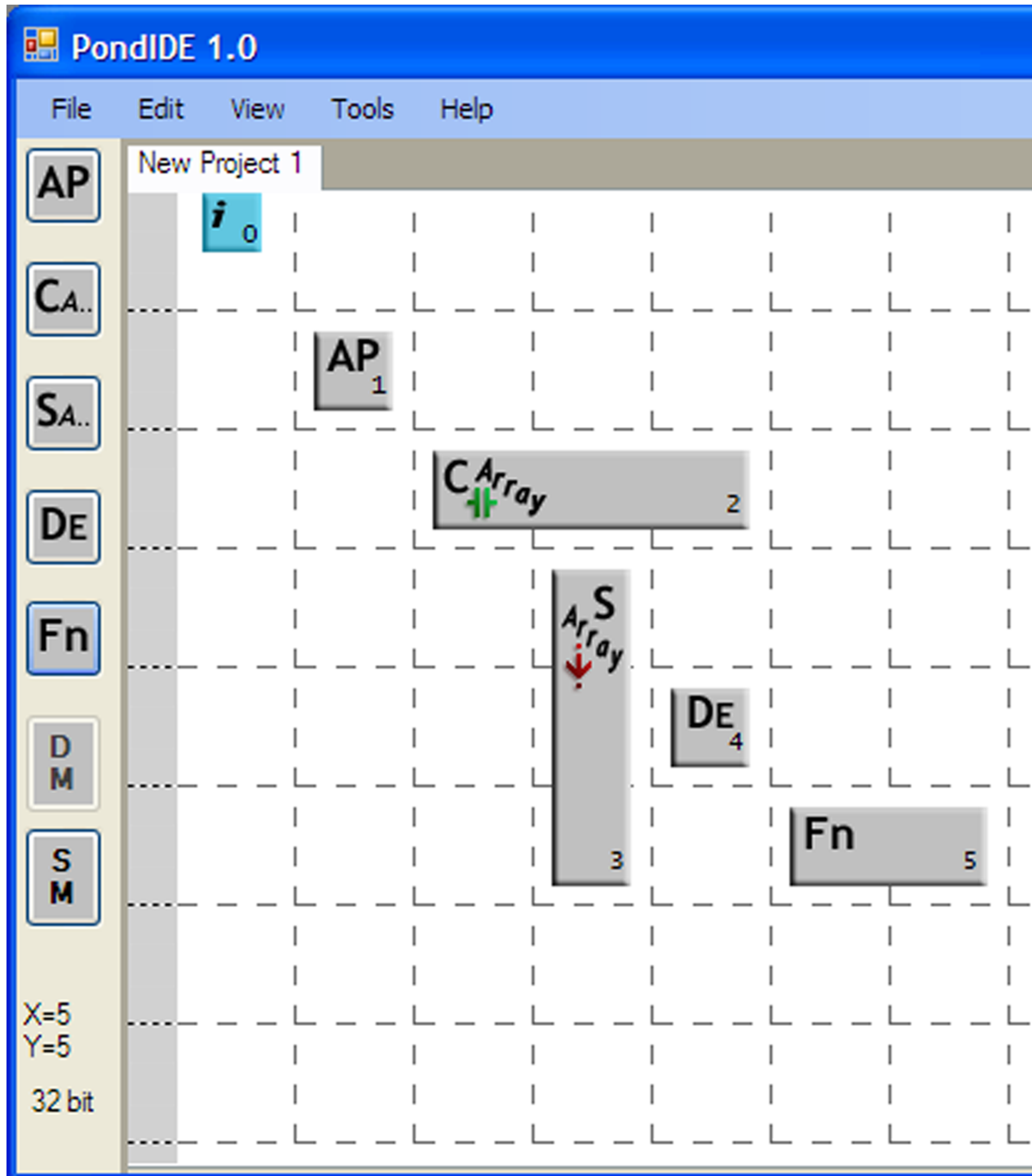


Figure 3.3: Different tokens placed on the layout grid.

method of calling functions differs from the actual description of how a function is called in hardware, which requires a normal atomic processor with the function to instantiate in the primary operand field and a reference to the data structure of data elements with the function inputs as the second operand, this seemed impractical in the IDE. The primary reason for this decision was the lack of a way to use a file name as a primary operand in an atomic processor, and a still-to-be-determined way to define or reference a data structure in the IDE. Overall the *function call* token still models what happens in hardware from the underlying simulation engine by instantiating a function definition and providing it with data to process. When a function instance reaches its return token, the function call token is provided with the returned operand to pass as an output to the rest of the project that called the function.

Data element tokens represent entries within a data structure as defined by the architecture in Chapter 2. Each data element has fields for first operand source, value, and type. Each is capable of providing data to other tokens or receiving data from a source token. Currently, the *sequential* and *concurrent array* tokens have no function in the simulator, they have been left for a future implementation of the IDE. The original thought behind the array tokens was to allow the user to represent several tokens performing the same operation with only a single token, thus reducing the amount of time spent placing replica tokens on the layout grid. Each array token has an array size field and operand source index fields in addition to the standard AP fields. These fields would allow the user to indicate the number of tokens in the array and how to process source inputs to the array, respectively.

3.4 Simulator Basics

When a simulation is started by the user by selecting simulate under the tools menu or by clicking on the *simulate mode* (SM) button on the left hand side tool bar, the design

is parsed. During the parsing process, each token's properties are read in and checked for errors. If an error is found during parsing, an error list is returned to the user indicating the problem. Warnings can also be returned to the user indicating a potential problem, but do not stop simulator execution. If a token's information is successfully read in without errors, a simulation operation for that token is created. Once an entire row is parsed successfully without errors, a simulation step that contains each of the simulation operations for that row is created. An overall simulation path is assembled from steps that represent each row of the layout grid, as further discussed in Section 5.1.3.

Likewise, during simulation, the simulator calls the step for each row, which in turn calls the operation for each token. Operations are split into *math*, *logic*, *status*, *no*, *function* and *call* operations. The *math* and *logic* operations are called if an operation code defined in a token is an arithmetic or logic type operation. *Status* operations are called for manipulations to status bits and *no* operations are called if a NOP is desired. *Function* operations handle returns from a function instance and *call* operations handle function instantiation calls.

Furthermore, the simulator can run all the steps at once or an option is provided to step through them one at a time. Breakpoints are provided for the *simulate all* option to stop simulator execution at a desired location. Additionally, during the *simulate step* option, or at a breakpoint, a yellow bar highlights the current row. Once the user has finished simulating their design they can return the IDE to design mode by selecting the *DM* button on the right hand side tool bar. Further details about the simulator internals are provided in Section 5.1.

3.5 Creating a Function Definition

Now that a basic understanding of the IDE exists, it is possible to create simple function definitions. A function definition consists of an appropriate number of input tokens to pass data into the function instance and tokens on the layout grid to process data. The user may set the number of inputs for the function definition and define name and description fields for the function from within the project properties dialog. In addition the user can select the bit size representation for the project; options of 16, 32, or 64 bit operands are available. The function definition ends in an AP token with the “RETURN” opcode as its operation. Once a function definition is completed and saved, it can be instantiated from other functions created in the IDE.

Furthermore, after the number of inputs for the function has been defined, default values for each of the inputs should be defined for testing and debug purposes. It should be noted that the default values are only used when debugging a function definition. They are ignored during a function instantiation, regardless of whether the user defined a source in the function call token. While the user is defining default values for the inputs, they can also enter labels and descriptions for the inputs. When a function is instantiated from a different project these labels and descriptions will be shown to the user in the help box to keep track of what each input is for.

Chapter 4

Experimental Results

The following chapter discusses several sample applications that have been implemented within the IDE and the various challenges associated with each of these algorithms. Screen captures, with additional call out boxes added, have been included for further clarity of each example. Details regarding the simulation results of each algorithm are explained, along with a discussion on any special cases.

4.1 Sequential Code Example - Fibonacci Series

An example of an algorithm that exhibits a low amount of parallelism is calculating the Fibonacci series, which is an entirely sequential process. To calculate the first twelve numbers of the Fibonacci series and return the last one, eleven steps are required. The first processor is initialized with 0 and 1 as its operands to add together. The processors that follow the first add the outputs of the previous processors as defined by the algorithm to calculate the Fibonacci series. The final and eleventh processor in the function returns the twelfth element in the Fibonacci series. A detailed screen capture of the program used to calculate the Fibonacci series is shown in Figure 4.2. The algorithm implemented in the IDE is modeled after the C code provided in Figure 4.1.

```
int fibonacci()
{
  int i;
  int a, b;
  int c[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
  a = 0;
  b = 1;
  for(i=0; i<10; i++) {
    c[i] = a + b;
      a = b;
      b = c[i];
    }
  return 0;
}
```

Figure 4.1: C code that calculates the first twelve elements of the Fibonacci series.

4.2 Concurrent Code Example - Vector Addition

A simple vector addition example shows how the architecture operates when a given algorithm exhibits a high level of parallelism. This example uses two eight-element arrays of data elements for the vectors and eight concurrent atomic processors. The function executes in three cycles, one cycle is used for the initial broadcast of data to the atomic processors and another cycle is used to add each element of the first vector to its respective element in the second vector. The final cycle would be used to return an array of data elements to the calling function. Currently, the simulator supports passing data element arrays into a function, but can only return a single value. Because of this, the return instruction has been omitted from the example. A screen capture of the example is shown in Figure 4.4. C code for simple vector addition is provided in Figure 4.3 to show the model the IDE implementation is based on.

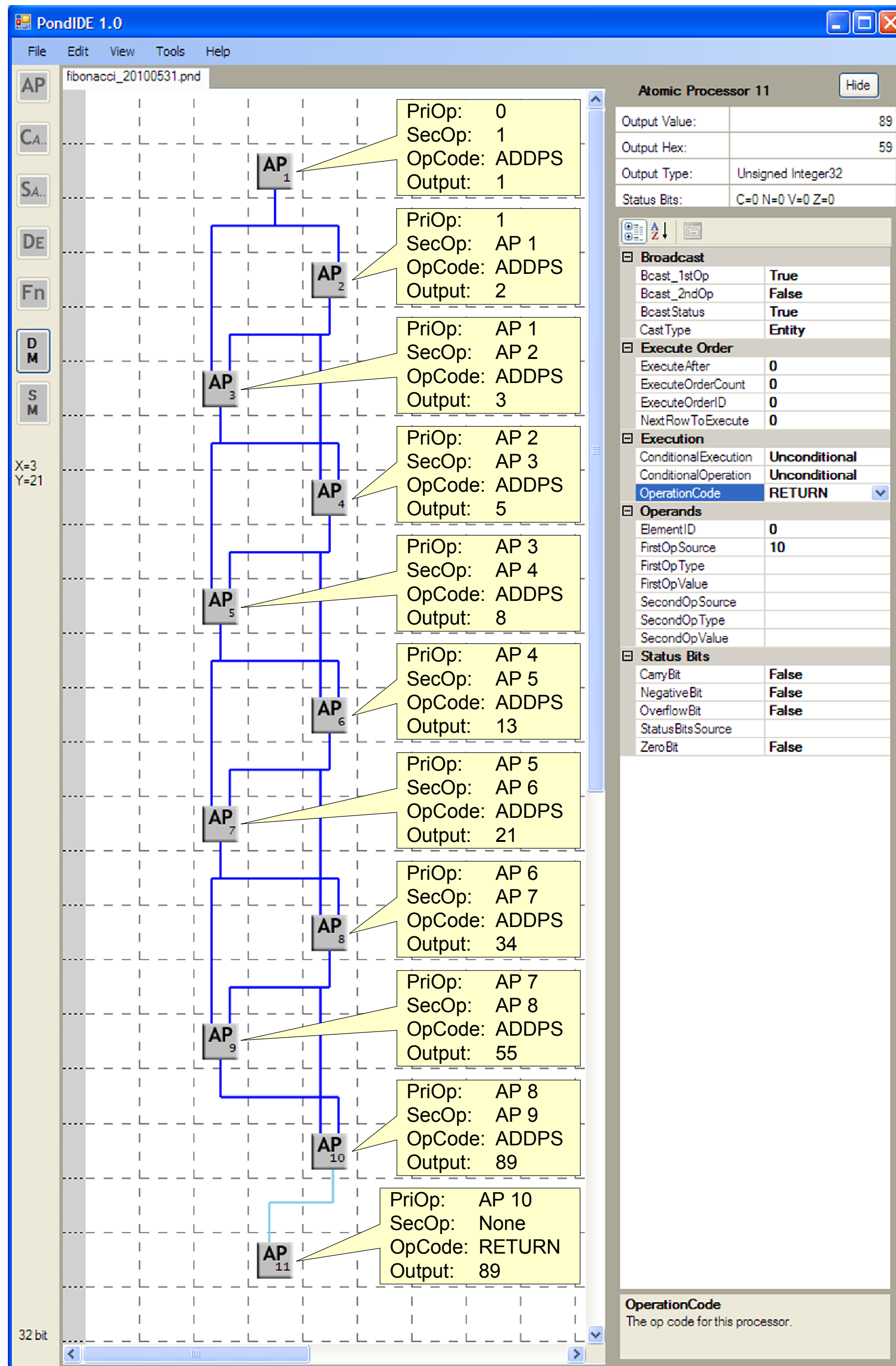


Figure 4.2: Screen Capture of the Fibonacci Program showing the calculation of the first twelve elements of the Fibonacci series. AP 10 returns the twelfth element to the calling function. Call out boxes have been added to show the values within each processor.

```
int vector_add()
{
  int vector1[8] = {1, 2, 3, 4, 5, 6, 7, 8};
  int vector2[8] = {5, 9, 1, 45, 13, 52, 9, 23};
  int i;
  for(i=0; i<8; i++) {
    vector1[i] = vector1[i] + vector2[i];
  }
  return 0;
}
```

Figure 4.3: C Code that performs vector addition for two eight element vectors.

4.3 Integer Multiplication using the Left-Shift Algorithm

Integer multiplication has been implemented in the IDE by using an unrolled version of the left-shift multiplication algorithm. In this algorithm the multiplicand is added to the product if the multiplier has a binary value of 1 in its *least significant bit* (LSB). For each iteration of the algorithm, the multiplier is shifted right one bit, and the multiplicand is shifted left one bit. The example shown in Figure 4.6 is a 32-bit left shift multiplier. Input 0 is the multiplicand and Input 1 is the multiplier passed into the function instance. After 32 iterations of the left shift algorithm complete, the function instance returns. Figure 4.7 shows a simulation of the unsigned integers 5 and 10 being left-shift multiplied to yield a result of 50. A single iteration of the left shift integer multiply algorithm is detailed by the following steps; References to AP IDs are the numbers shown on the faces of the tokens in Figure 4.7. A C code implementation of the algorithm is provided in Figure 4.5 to show the model the IDE implementation is based on.

1. Bitwise AND the value of the appropriately shifted multiplicand, a, with the binary mask 1 as shown in AP5. The Z flag is set if the least significant bit of the result is zero.

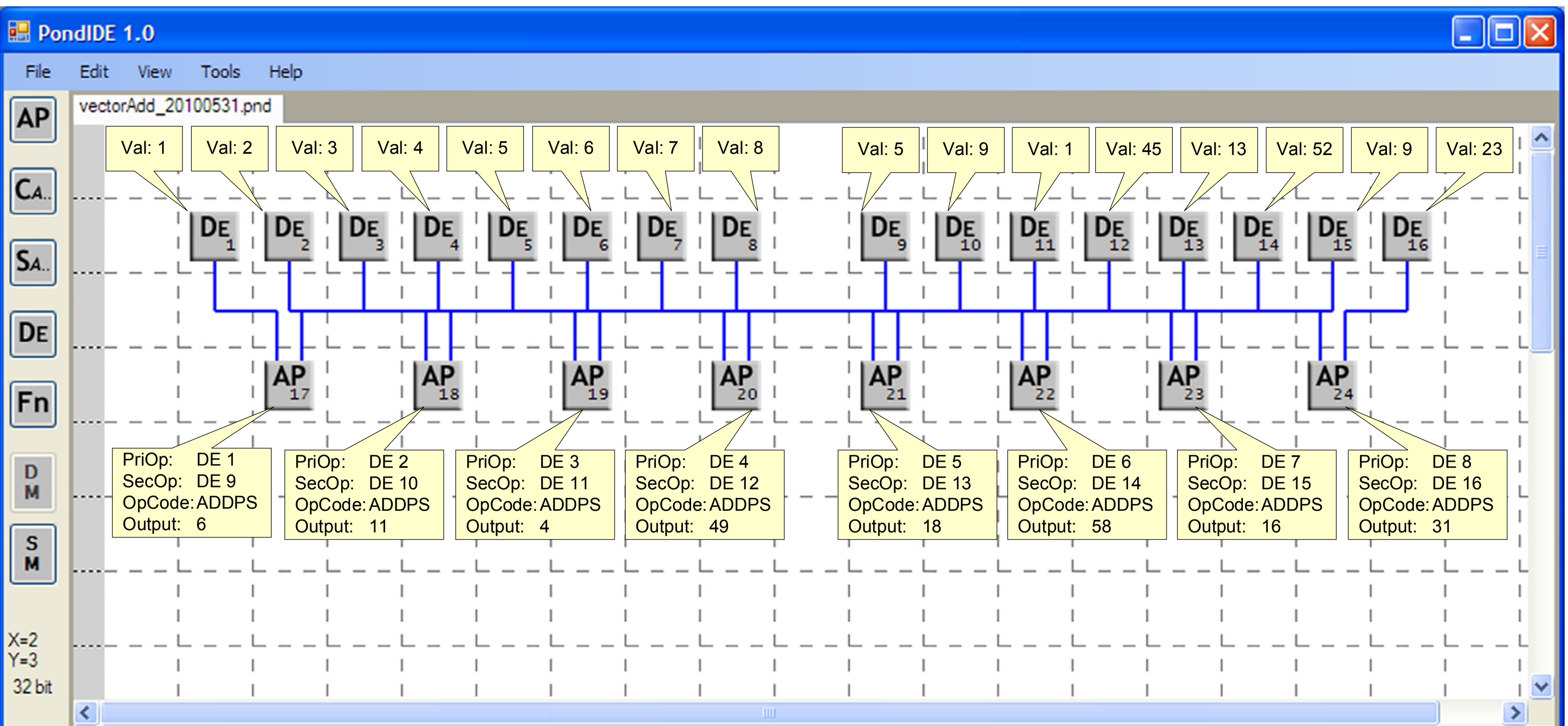


Figure 4.4: Screen Capture of the Vector Add Program showing two eight element arrays of data elements, DEs 1 through 7 and 8 through 16, being added by APs 17 through 24. Call out boxes have been added indicating the operands, operation and output of each of the tokens.

2. If the Z flag is 0, then the appropriately shifted multiplier, x, is added to the partial product, as shown in AP6. During this step, a is shifted right and x is shifted left unconditionally in preparation for the next iteration, as shown in AP3 and AP4 respectively. In the looped version of this algorithm, a counter would be decremented here to track the number of iterations.

3. After the appropriate number of iterations, tracked by the counter or explicitly defined in the unrolled version of the algorithm shown here, the multiplier function instance returns the product accumulated from each adder AP, as shown in AP127.

```
// a is the multiplicand
// x is the multiplier
int left_multiply(int a, int x)
{
  int i;
  int product;                    // partial product
  product = 0;
  for( i=0; i<32; i++) {
    if( a & 1 ) {
      product = product + x;
    }
    a = a >> 1;
    x = x << 1;
  }
  return product;
}
```

Figure 4.5: C Code that performs integer multiplication using a left shift multiplication algorithm.

4.4 Floating Point Packing and Unpacking

The IDE uses four custom-defined function definitions for the following floating point operations: Unpacking the sign bit, unpacking the exponent, unpacking the mantissa, and

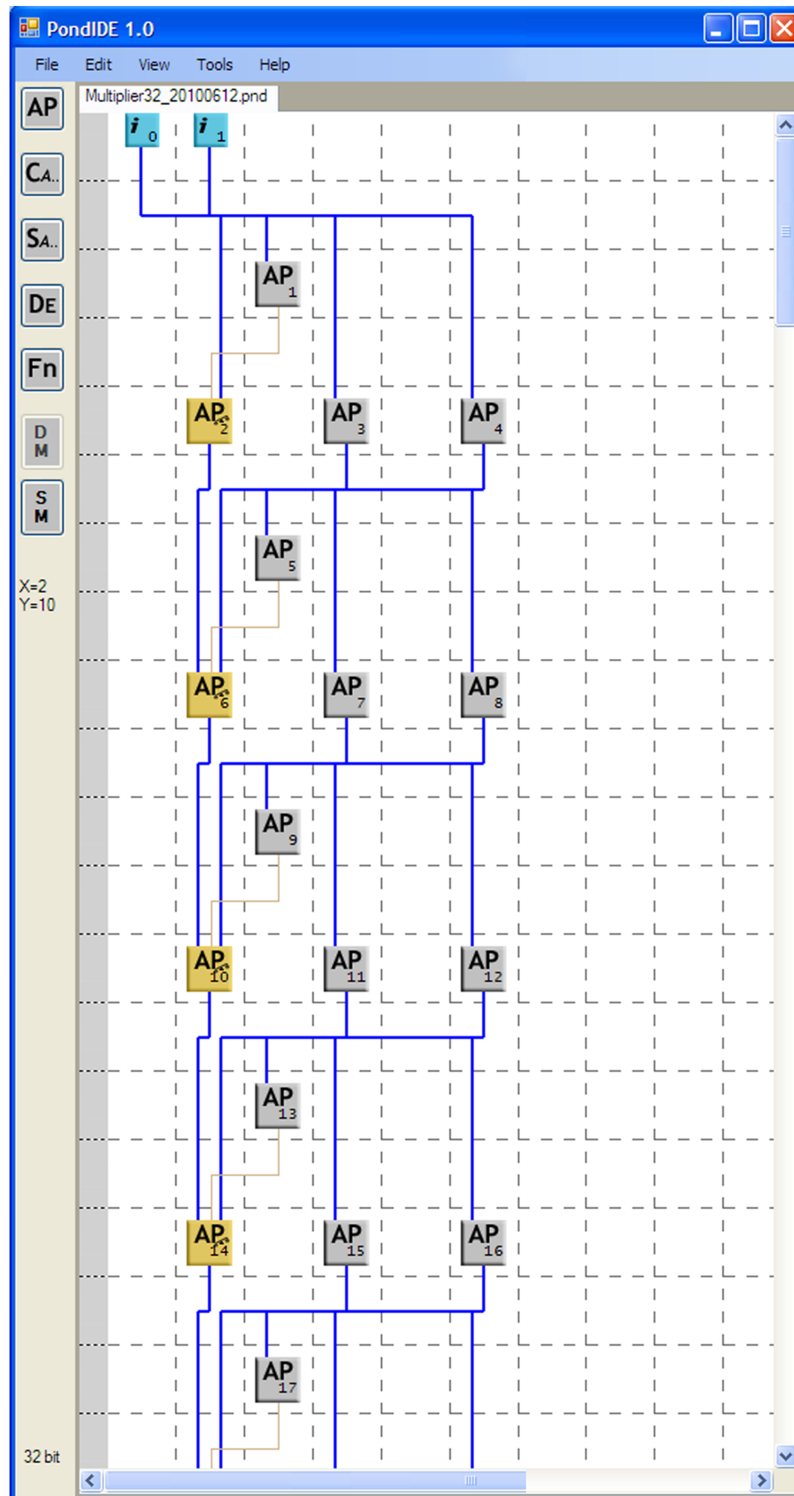


Figure 4.6: Screen Capture of 32-bit Left Shift Integer Multiply Function Definition

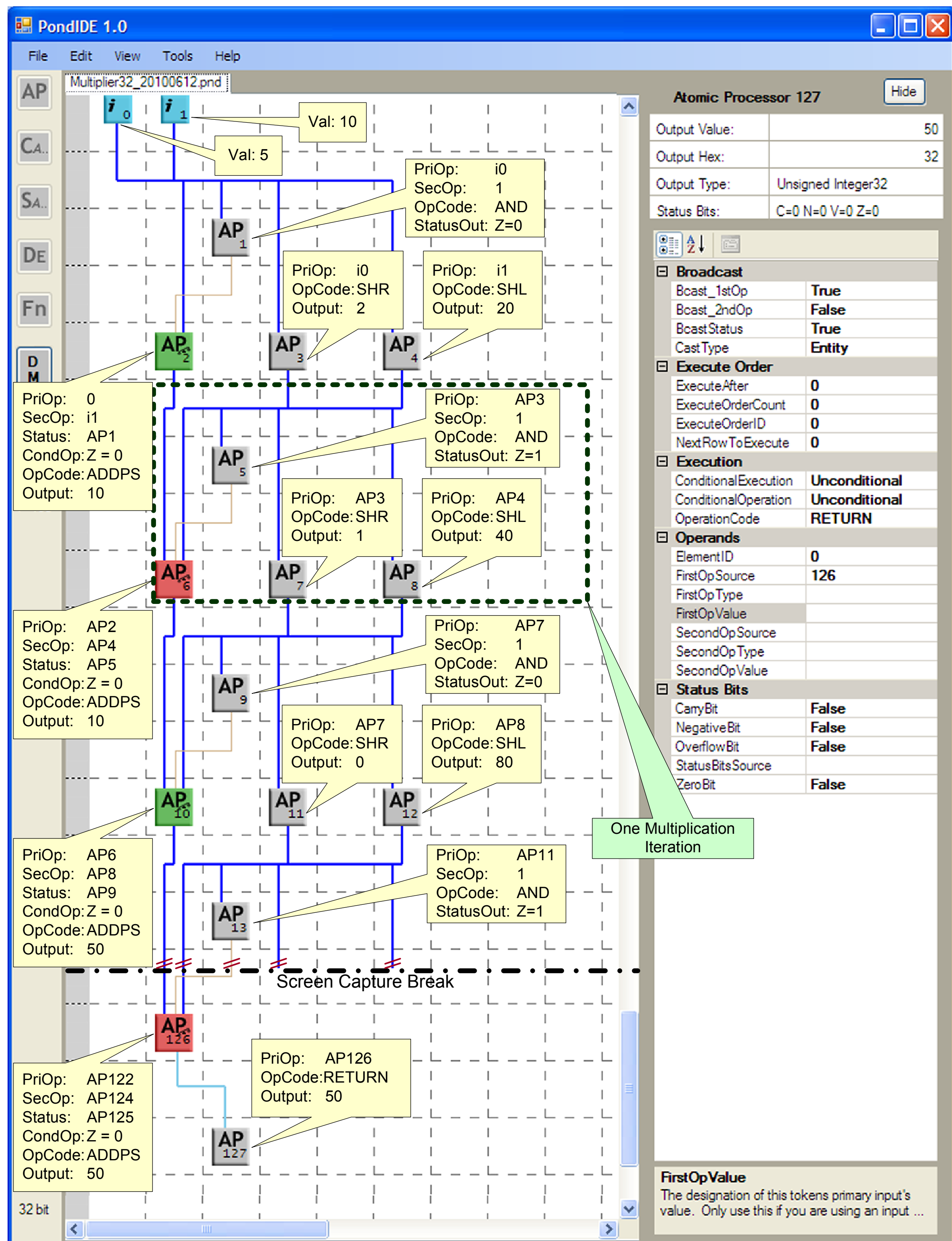


Figure 4.7: Screen Capture of 32-bit Left Shift Multiply Simulation, multiplying unsigned integers 5 and 10 to yield a result of 50. Call out boxes have been added to show the values within each processor.

packing sign bit, exponent and mantissa into a floating point number. Because each of these operations is defined as a function definition it can be instantiated from other function instances as shown in Section 4.5. Each of the unpacking and packing operations is discussed in detail in the sections that follow.

4.4.1 IEEE Floating Point Standard

The IEEE 754 standard [82] for floating point numbers defines both 32-bit and 64-bit representations; details of the bit usage in the standard have been given in Table 4.1 [83].

Table 4.1: Definition of bits for the IEEE 754 standard for floating point numbers.

	Sign	Biased Exponent	Significand $s = 1.f$ (the 1 is omitted)
	$\pm$	e + bias	f
32-bit:	1 bit	8 bits + 127	23 + 1 bits, single-precision
64-bit:	1 bit	11 bits + 1023	52 + 1 bits, double-precision

4.4.2 Unpacking the Sign

The sign bit for a floating point number is stored in the *most significant bit* (MSB). To unpack the sign bit, the binary representation of the floating point number is simply masked with a 1 in the MSB position using an AND operation. Once the result is in this form, the operation is complete. It is not necessary to shift the sign bit to an alternate location, as it can be manipulated in this position with simple bitwise logic. C code that unpacks a floating point number's sign is shown in Figure 4.8. Figure 4.9 shows details of how a floating point number's sign is unpacked in the IDE.

```c
int FPunpack_sign(float f)
{
  int s;
  // create a pointer to memory location of f
  unsigned int* pI = (unsigned int*)&f;
  s = *pI & 0x80000000;          // mask off sign bit
  return s;
}
```

Figure 4.8: C Code that unpacks the sign bit from the floating point number representation.

4.4.3 Unpacking the Exponent

The exponent for a floating point number is stored as an unsigned integer with a bias added to it, allowing representation of both positive and negative exponents without the use of a sign bit. To unpack the exponent for a 32-bit floating point number, bits 23 through 30 (assuming the LSB is bit 0) are masked with ones in an AND operation. Next, the bits can be shifted right 23 places to allow them to reside in the eight least significant bit locations. Finally, the bias of 127 is subtracted from the exponent. Depending on the math operations to be performed on the exponent, not all of these operations are always required, but in this case, the most generic approach has been taken. C code that unpacks a floating point number's exponent is shown in Figure 4.10. Figure 4.11 shows details of how a floating point number's exponent is unpacked in the IDE.

4.4.4 Unpacking the Significand or Mantissa

The *significand*, also often referred to as the *mantissa*, is the fractional portion of the floating point number representation. In the single-precision representation it resides in the least significant 23 bits. In order to unpack this portion of the number, the lowest 23 bits are first masked off using an AND operation with the hexadecimal number 0x7FFFFF. These 23 bits represent the numbers to the right of the decimal place in the fractional portion of the

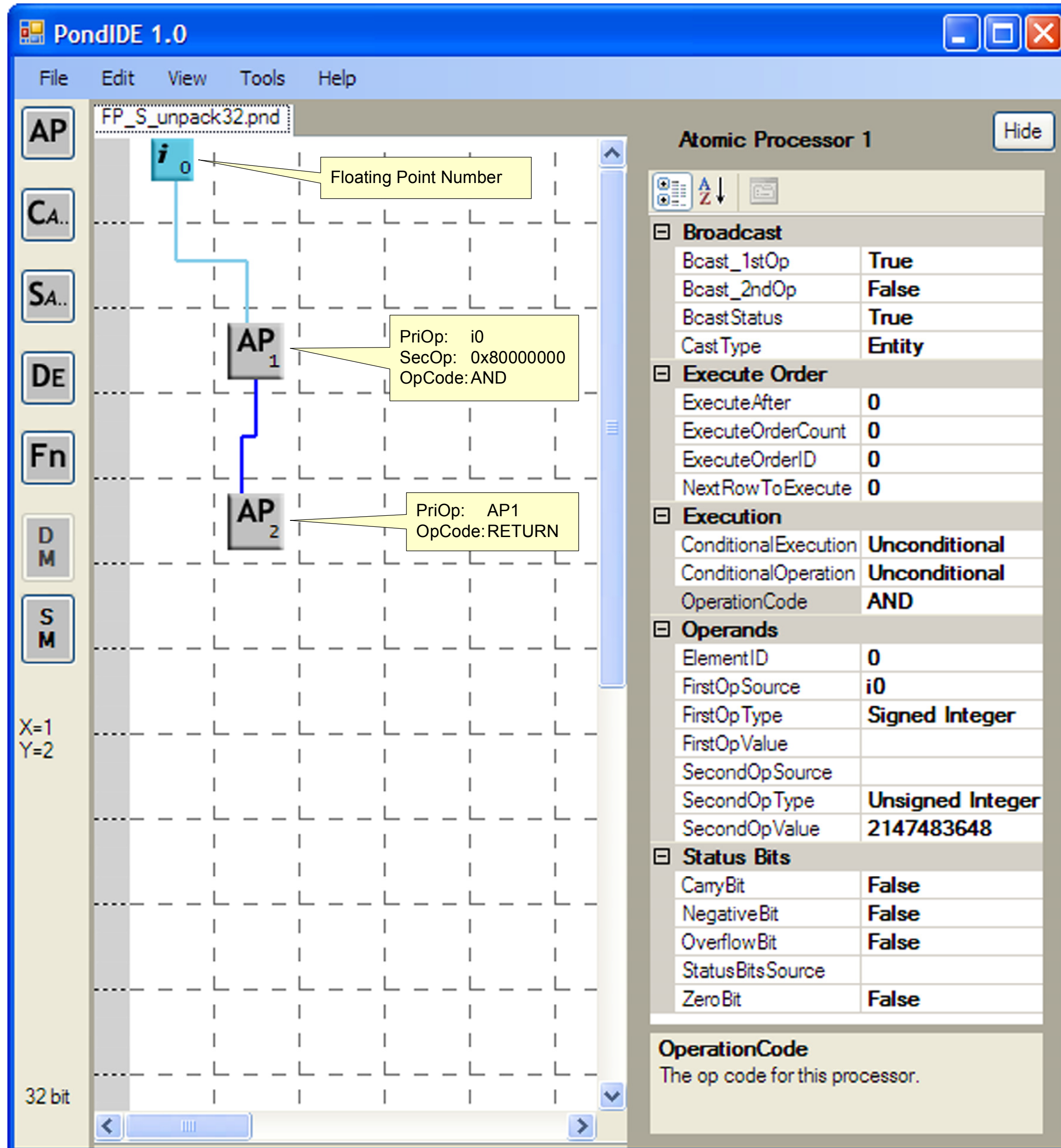


Figure 4.9: The sign bit of a floating point number is unpacked by simply masking the MSB. Call out boxes have been added to show the operation performed by each processor.

```
int FPunpack_exponent(float f)
{
  int s;
  // create a pointer to memory location of f
  unsigned int* pI = (unsigned int*)&f;
  s = *pI & 0x7F800000;          // mask off exponent bits
  s = s >> 23;                   // shift right 23 places
  s = s - 127;                   // subtract bias of 127
  return s;
}
```

Figure 4.10: C Code that unpacks the exponent portion of the floating point number representation.

floating point number. In order to complete the unpack operation for the mantissa, a 1 must be added to the left of the decimal place. This single whole bit is always removed from the mantissa when the floating point number is packed in the IEEE representation. To add in this 1, an OR operation is used with a 1 set as the 23rd bit, where bit 0 is the LSB, also represented as the hexadecimal number 0x800000. C code that unpacks a floating point number's mantissa is shown in Figure 4.12. Figure 4.13 shows the details of this operation performed in the IDE.

4.4.5 Packing Floating Point Numbers

The process of packing a floating point number from its sign, exponent, and mantissa reverses all the processes performed when the number was unpacked. The sign bit is allowed to remain in its native representation throughout floating point operations, so it requires no logic operations to format it. The exponent bit requires that the bias be added back in, thus the value of 127 is added to the exponent. The mantissa is checked for a second digit to the left of the decimal place, if this has occurred the number is shifted one place to the right and a 1 is added to the exponent. Next, before the exponent is shifted left 23 places, it is checked for overflow. If bit 8 of the exponent is set to 1, where bit 0 is the LSB, then

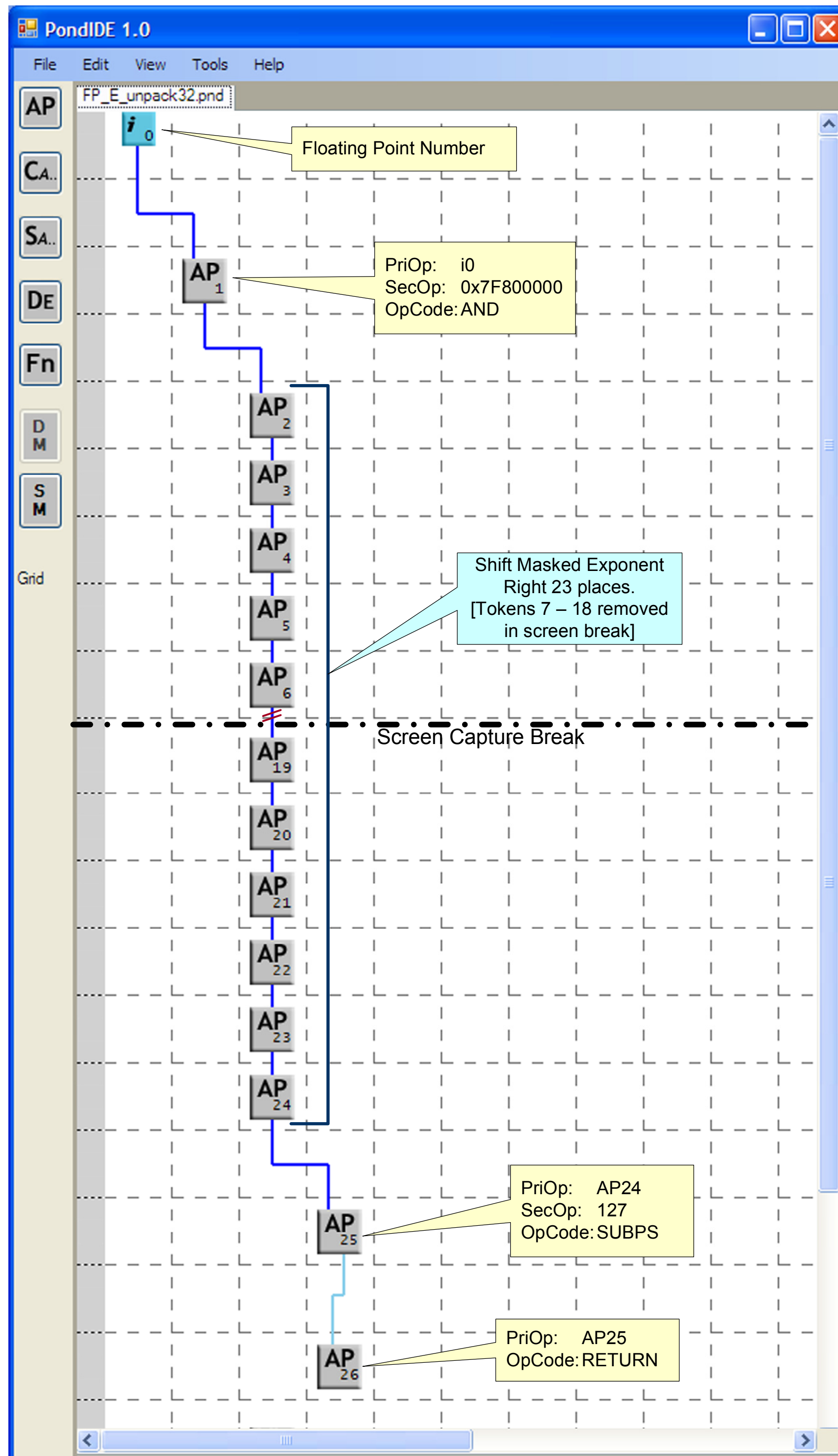


Figure 4.11: The exponent of a 32-bit floating point number is unpacked by masking off the exponent bits, right shifting 23 places, and subtracting the bias of 127. Call out boxes have been added to show the operation performed by each processor.

```c
int FPunpack_mantissa(float f)
{
  int s;
  // create a pointer to memory location of f
  unsigned int* pI = (unsigned int*)&f;
  s = *pI & 0x7FFFFF;           // mask off mantissa bits
  s = s | 0x800000;             // add one to left of decimal
  return s;
}
```

Figure 4.12: C Code that unpacks the mantissa or significand portion of the floating point number representation.

the exponent has overflowed, and the packing operation returns the floating point representation of positive infinity. Other special cases are defined in the IEEE standard, but they have been omitted from this simple example. If the exponent has not overflowed, then it is shifted 23 places to the left and a logic OR with the sign bit performed. Finally, the extra 1 to the left of the decimal in the mantissa is masked off and a logic OR is performed between the combined sign/exponent result and the mantissa before returning. C code that the algorithm is modeled after is provided in Figure 4.14. This operation in the IDE is detailed in Figure 4.15.

4.5 Floating Point Multiplication

4.5.1 24-bit Fixed-Point Multiplier

The floating point multiplication algorithm discussed in the next section requires a fixed-point multiplier to multiply the two mantissas together. As discussed in Section 4.4.4, the mantissa of a single-precision floating point number has 23 bits to the right of the decimal place and a single one to the left of the decimal place for a total of 24 bits. Because these fractional representations will be truncated to 24 bits in length to fit within the packed

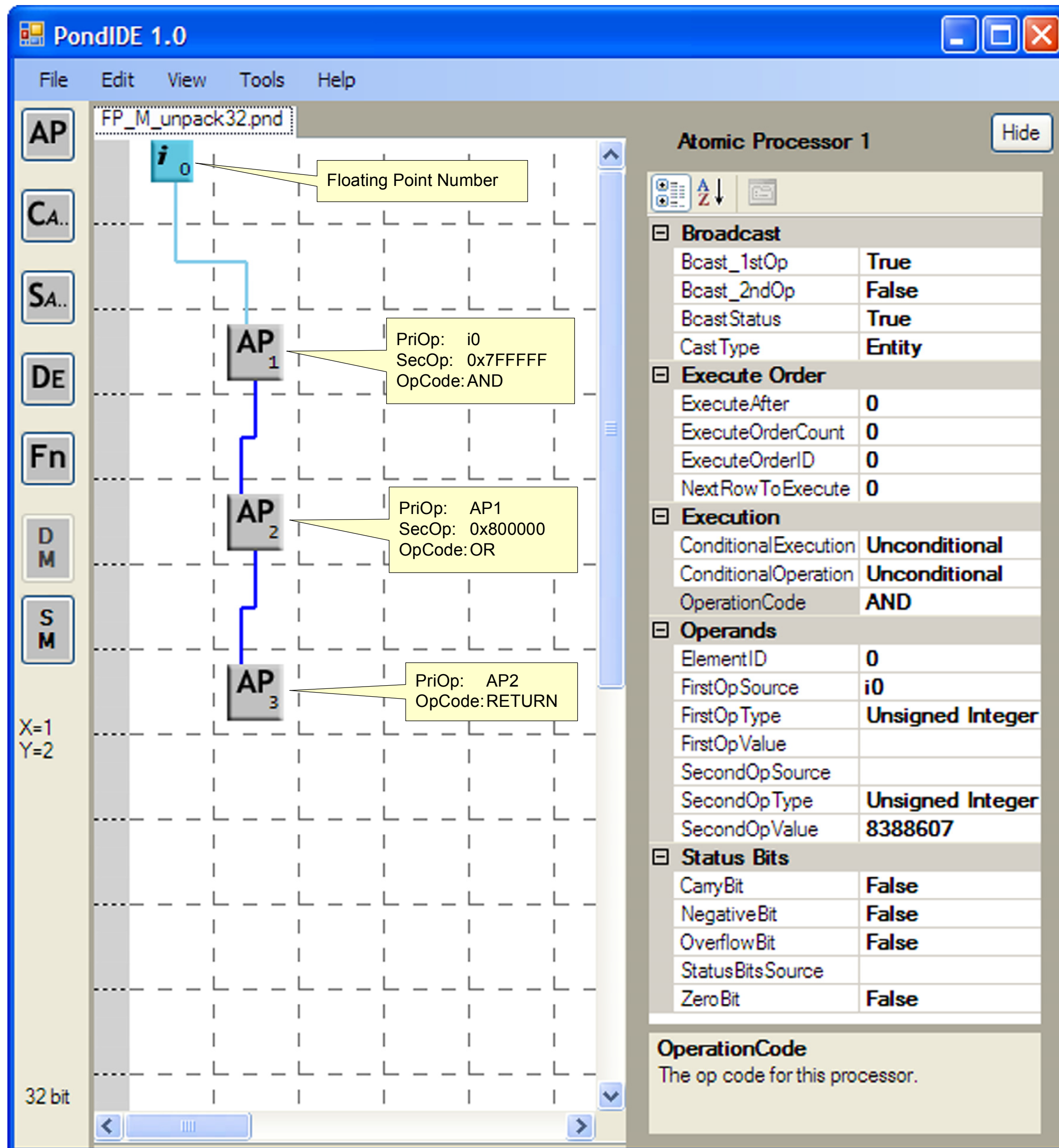


Figure 4.13: The significand of a 32-bit floating point number is unpacked by masking off the least significant 23 bits and adding a 1 in the 23rd bit position, where bit 0 is the LSB. Call out boxes have been added to show the operation performed by each processor.

```
float FPpack( int sign, int exponent, int mantissa )
{
  int FPbits;
  // create a pointer to memory location of FPbits
  float* F = (float*)&FPbits;
  exponent = exponent + 127;
  if( mantissa & 0x1000000 ) {
    exponent += exponent;
    mantissa = mantissa >> 1;
  }
  // check for overflow of exponent
  if( exponent & 0x100 ) {
    // return infinity
    FPbits = 0x7F800000;
  } else {
    exponent = exponent << 23;
    mantissa = mantissa & 0x7FFFFF;
    FPbits = sign | exponent | mantissa;
  }
  return *F;
}
```

Figure 4.14: C Code that packs a sign, exponent, and mantissa into the IEEE Floating Point representation.

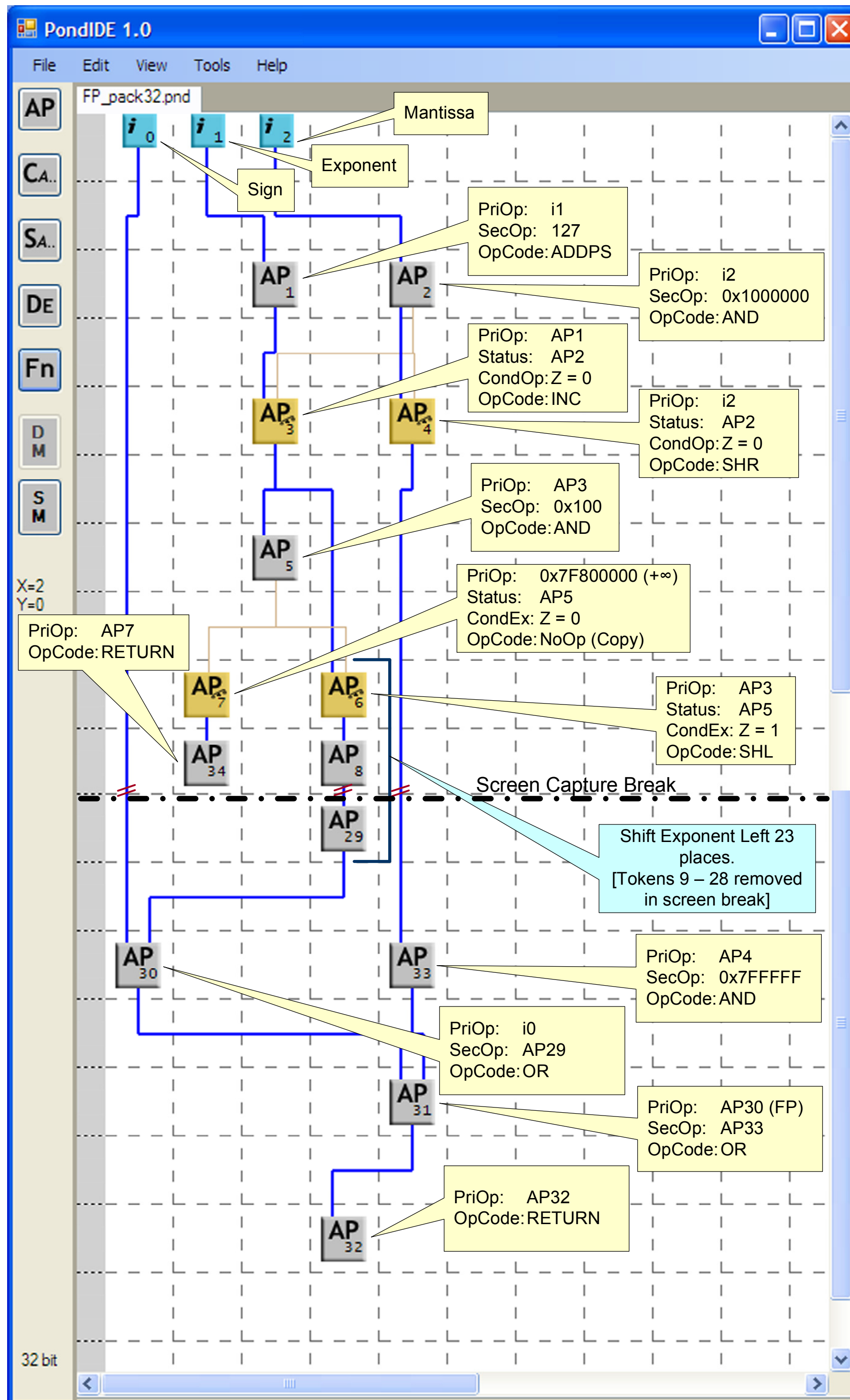


Figure 4.15: Packing operation for a 32-bit floating point number. The function definition takes a sign, exponent and mantissa as inputs and outputs a floating point number. Call out boxes have been added to show the operation performed by each processor.

floating point number, it is advantageous to discard the extra bits produced during multiplication to avoid overflowing the 32-bit word length. This is easily done with a right shift integer multiplication algorithm, purposefully shortened to 24 iterations instead of the normal 32. It is noted that the final iteration of this algorithm does not right shift the resulting product as normally done in a right shift multiplier. This is because 23 bits are required to the right of the decimal place. A final right shift would cause there to be only 22. The algorithm as shown in Figure 4.17 is executed with the following steps, C code is provided in Figure 4.16 for easier understanding:

1. Bitwise AND the value of the appropriately shifted multiplier, x, with the binary mask 0x1 as shown in AP1. The Z flag is set if the least significant bit of the result is zero.

2. If the Z flag is 0 then the multiplicand, a, is added to the partial product, as shown in AP2. During this step, x is shifted right unconditionally in preparation for the next iteration, as shown in AP3. In the looped version of this algorithm, a counter would be decremented here to track the number of iterations.

3. The partial product is shifted right as shown in AP4. Step one for the next iteration begins concurrently, as shown in AP5.

4. After the appropriate number of iterations, tracked by the counter or explicitly defined in the unrolled version of the algorithm shown here, the multiplier function instance returns the product accumulated from each adder AP, as shown in AP95.

4.5.2 Function Calls to Perform Floating Point Multiplication

Multiplication of two floating point numbers requires a few simple logic and arithmetic operations once the parts of the floating point number have been unpacked. Equation 4.1

```
// a is the multiplicand
// x is the multiplier
int Fixed_right_multiply(int a, int x)
{
  int i;
  int product;                    // partial product
  product = 0;
  for( i=0; i<23; i++) {
    if( x & 1 ) {
      product = product + a;
    }
    x = x >> 1;
    product = product >> 1;
  }
  if( x & 1 ) {
    product = product + a;
  }
  return product;
}
```

Figure 4.16: C Code that performs a 24-bit fixed point multiplication using a right shift multiplication algorithm.

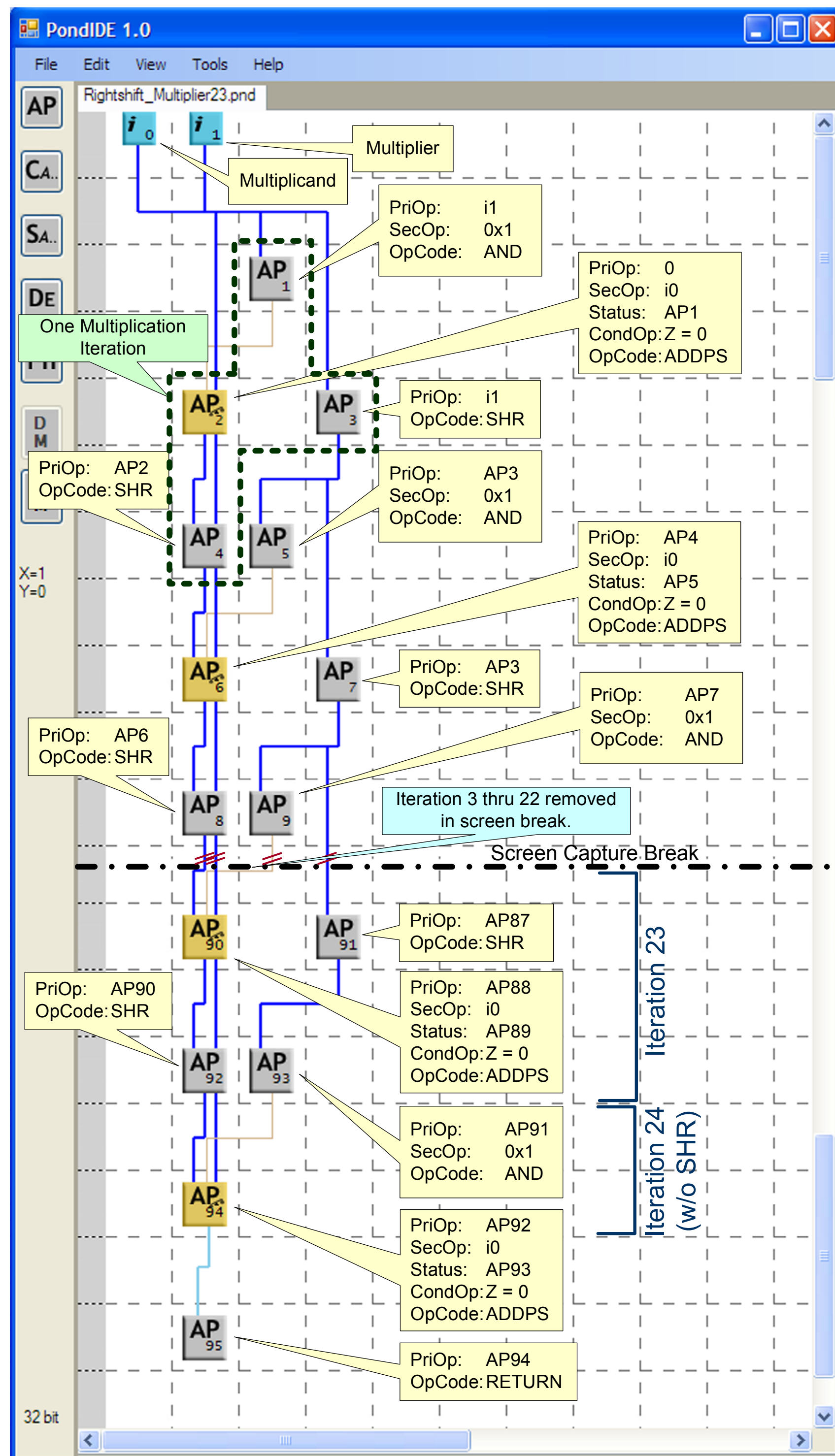


Figure 4.17: Right Shift Integer multiplier for 24 bit fixed point multiplication used in floating point multiplication algorithm. Call out boxes have been added to show the operation performed by each processor.

shows the process for floating point multiplication where the two exponents are added and the significands are multiplied [83].

$$(\pm s_1 \times b^{e_1}) \times (\pm s_2 \times b^{e_2}) = \pm (s_1 \times s_2) \times b^{e_1+e_2} \quad (4.1)$$

An example of floating point multiplication in the IDE is shown in Figure 4.19. Unpacking operations occur in function calls 1 through 6. Next, the sign bit of the result is calculated using a simple XOR function between the two input sign bits, as shown in AP7. Concurrently, the exponent bit for the result is calculated by adding together the two input exponents, shown in AP8. Additionally, the resultant mantissa is also calculated concurrently through a function call to the 24-bit fixed point multiplier discussed in the previous section, as shown in function call 9. Finally, the resulting parts can be packed into a 32-bit floating point number through the packing operation discussed in Section 4.4.5, as shown in function call 10. AP11 returns the resulting product of the floating point multiplication. The C code in Figure 4.18 shows how each of the previously mentioned C code implementations would be called to perform the floating point multiplication algorithm. It should be noted that the C code is sequential while the IDE performs several of these operations concurrently.

4.6 Integer Division

A complete example of integer division with the currently configured IDE is not possible due to the unimplemented loop functionality, as discussed in Section 5.2.1. The provided example shows how a division function would be implemented once loop functionality is added to the IDE, as discussed in Section 5.2.1. The division algorithm that is shown in the example is analogous to the one shown in Figure 4.21 taken from [1]. The ID numbers

```c
float FPmultiply( float a, float b )
{
  int sign_a, sign_b, exp_a, exp_b, mantissa_a, mantissa_b;

  // unpack floating point numbers with unpack commands
  sign_a = FPunpack_sign(a);
  sign_b = FPunpack_sign(b);
  exp_a = FPunpack_exponent(a);
  exp_b = FPunpack_exponent(b);
  mantissa_a = FPunpack_mantissa(a);
  mantissa_b = FPunpack_mantissa(b);

  // perform floating point multiplication on unpacked parts
  sign_a = sign_a ^ sign_b; // XOR signs
  exp_a = exp_a + exp_b;     // Add exponents
  mantissa_a = Fixed_right_multiply(mantissa_a, mantissa_b);

  // repack result as a floating point number and return
  return FPpack(sign_a, exp_a, mantissa_a);
}
```

Figure 4.18: C code that implements the top level floating point multiplication algorithm by making calls to the previously mentioned C code.

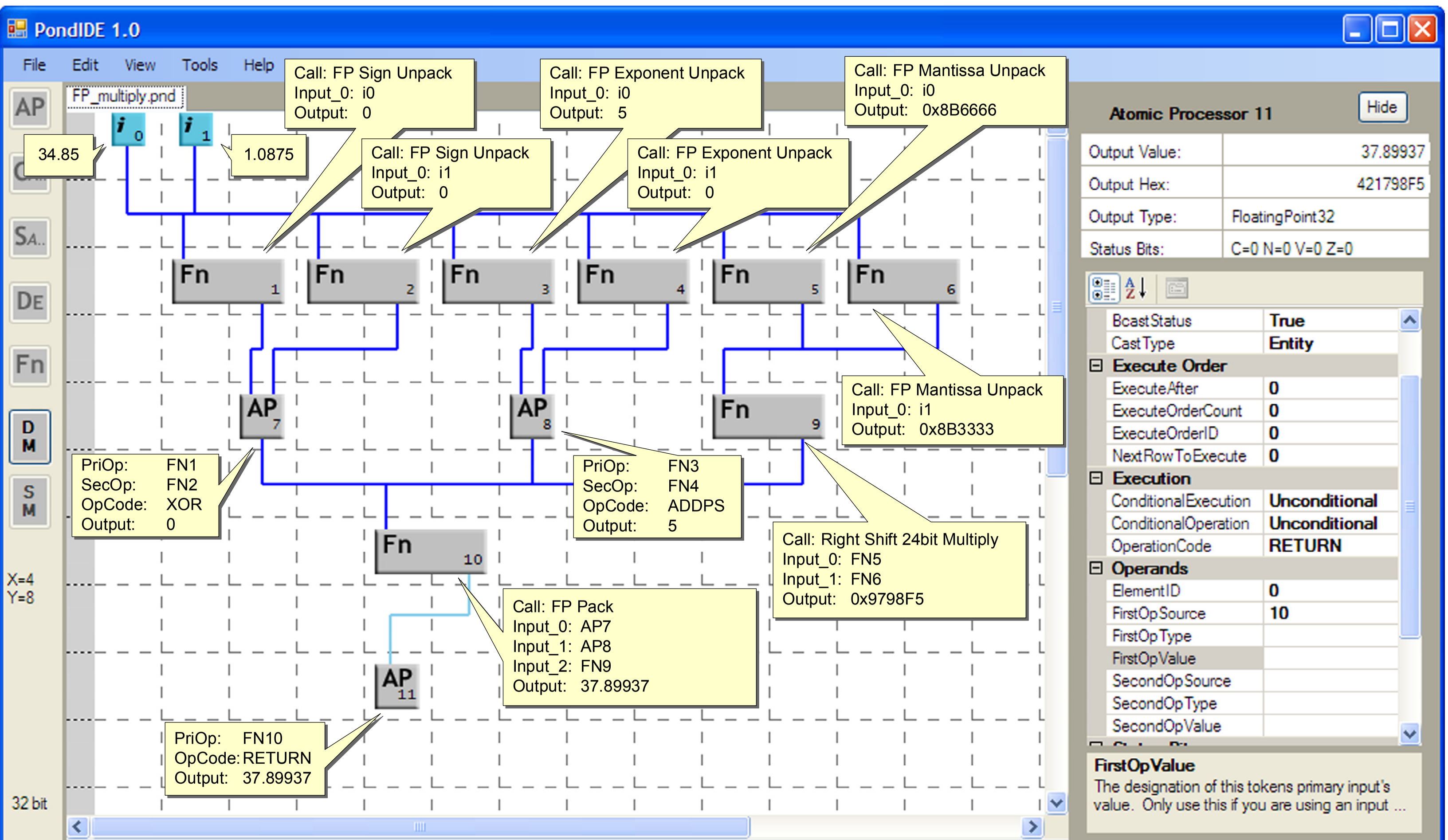


Figure 4.19: Floating point multiply program with function calls to floating point unpacking and packing operations and right shift fixed point (24bit integer) multiplier. Call out boxes have been added to show the operation performed by each processor.

used in Figure 4.21 coincide with the ID numbers shown in parenthesis in Figure 4.22. The steps of the algorithm are shown below with references to AP ID numbers shown on the token faces in Figure 4.22. Figure 4.20 shows a C code implementation of the algorithm.

1. Assign the dividend or numerator, n, to the remainder, r, as shown in AP1.

2. Subtract the divisor, d from r as shown in AP2.

3. If $r \geq 0$, then increment the quotient, q, as shown in AP4 and repeat steps 2 and 3. Otherwise the division is complete, add d to r and return, as shown in AP3 and AP6 respectively.

It is recognized that this algorithm lacks efficiency because of the undetermined and often excessively large number of iterations required. In this division algorithm, the number of iterations relates directly to the numerical size of the numerator in regards to the denominator; For example, if you divide one million by one, it would take one million iterations. This simple and inefficient algorithm was chosen to show how a loop of an undetermined number of iterations would be implemented in the IDE. More efficient algorithms, such as a shift and subtract division, significantly reduce the number of operations required. The number of iterations in these algorithms are related to the bit size width in the binary representation. Because of this, the number of iterations are significantly less and predictable.

```
int iterativeDivision( int n, int d )
{
  int r = n;
  int q = 0;
  while( 1 ) {
    r = r - d;
    if ( r >= 0 ) {
      q = ++q;
      } else {
        r = r + d;
        return q;
      }
  }
}
```

Figure 4.20: C Code to perform division through iterative subtraction.

AP (0, 0)	AP (0, 1)	AP (0, 2)	AP (0, 3)	AP (0, 4)
ADD ID: 1 POp: *n* (ID: 5) SOp: 0 (ID: 4) EO: 1 PEO: 0	**SUB** ID: 1 POp: *r* (ID: 1) SOp: *d* (ID: 2) EO: 2 PEO: 1	**ADD** ID: 1 POp: *r* (ID: 1) SOp: *d* (ID: 2) EO: 3 PEO: 2 Cond: N SBID: 1	**RETURN** ID: 3 POp: *q* (ID: 3) SOp: [Calling FI HID] EO: 4 PEO: 3	**[DS Element]** ID: 5 Word: *n*
AP (1, 0)	AP (1, 1)	AP (1, 2)	AP (1, 3)	AP (1, 4)
INC ID: 3 POp: *q* (ID=3) SOp: - EO: 1 PEO: 2 Cond: !N SBID: 1	**[DS Element]** ID: 4 Word: 0	**[DS Element]** ID: 2 Word: *d*	**[DS Element]** ID: 1 Word: *r*	**[DS Element]** ID: 3 Word: *q*

Figure 4.21: Example integer division algorithm [1].

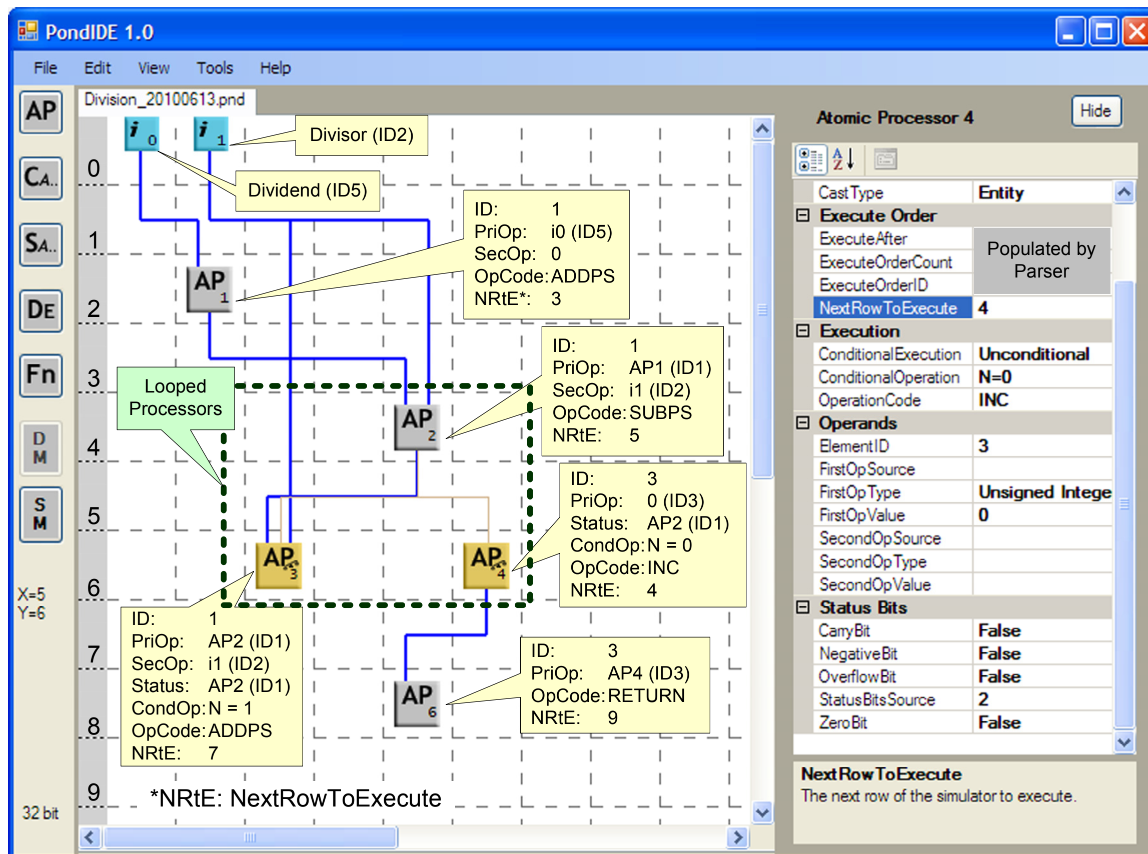


Figure 4.22: Screen Capture of sample division algorithm with *Next Row to Execute* field and non-unique ID numbers indicated in call out boxes (Simulation non-functional).

Chapter 5

IDE Development and Programming

This chapter discusses relevant details about development of the IDE. The IDE was developed in Visual Basic using Microsoft Visual Studio 2008. This development environment was chosen because of its ability to provide quick GUI development within an object oriented programming environment.

5.1 Program Structure

The IDE is composed of thirty Visual Basic classes which will be described briefly in this section. The thirty classes can be classified into three different types: Classes related to the GUI, classes that hold and interface with the data for each atomic processor, and classes related to the simulator. The main class related to all other classes is the "Form1.vb" class. This class contains all functions and variables related to the main GUI, and interfaces with all other classes in the program, directly or indirectly.

5.1.1 GUI Related Classes

Form1.vb

As mentioned above, the *Form1* class is the main class for the project. The Form1 class includes Windows Form Designer code which contains the structure of the GUI including details such as color, size and control locations. Additionally the Form1 class includes code that allows tokens to be placed and snapped to the layout grid, keeps track of the current tab and provides a layer of abstraction between the user and all other functions of the IDE program.

PG_data.vb

The PG_data class, short for property grid data, is a base class from which the following classes inherit:

- **PGD_AP.vb**: Defines property grid for Atomic Processors.
- **PGD_CA.vb**: Defines property grid for Concurrent Arrays.
- **PGD_DS.vb**: Defines property grid for Data Elements.
- **PGD_FN.vb**: Defines property grid for Function Calls.
- **PGD_ID.vb**: Defines property grid for Inputs.
- **PGD_SA.vb**: Defines property grid for Sequential Arrays.

Each of these classes define the properties displayed in the right hand side property grid which appears when a token is selected. A different child class is defined for each of the token types, as indicated by the abbreviation following the underscore in the name of each child class. When a token is created the appropriate child class is instantiated and stored with its respective token to hold all data defined for that token.

Line.vb

Handles how lines are drawn between tokens and defines all the properties associated with a line including the output token to start the line at, the input token to end the line at, the number of the input for the line being drawn (primary, secondary or status bits) and information about the status of the line.

ErrorList.vb

Includes Windows Form Designer code defining the form displayed with errors and warnings after a design is parsed.

Properties.vb

Includes Windows Form Designer code defining the form displayed when the user selects *Project Properties* under the view menu. Additionally, the class contains code to define the number of inputs, the project bit size representation and a name and description for the project/function definition, as described in Section 3.5.

About.vb

Includes Windows Form Designer code defining the form displayed when the user selects *About* under the help menu.

5.1.2 Data Related Classes

Operation.vb

Handles the undo and redo functionality in the IDE. Currently, the class handles undo / redo functionality for placement of new tokens and deletion of existing tokens. Future versions of the IDE should further develop the functionality of this class.

Frame.vb

Acts as a container that provides access to all the information in a given project. While nothing from the frame class is saved with a project, it provides all the access functions to search for tokens within their specified *data* class. It tracks the currently selected token, zoom level, undo and redo stacks, and the simulator.

Data.vb

Contains the data for a given project. Included within the data class is an array list of all the tokens for a project, an array of booleans indicating if a breakpoint is set for a given row, the width and height of the layout grid for the project, the project bit size representation, number of inputs, function name and description, and background color for the project, among other important items. When a project is saved the data class for that project is serialized into a binary file using the Visual Basic serialize command. When a project is opened, the file is deserialized and placed into a frame to provide access to the data.

Token.vb

Stores all data associated with each specific token. Each time a token is placed on the grid a token class is instantiated and added to the array list of tokens in the respective *data* class. Tokens store their type, size, absolute position, grid location, ID, properties, inputs, and color. The appropriate picture for the token is retrieved based on type, size, and color each time the tokens on the layout grid are redrawn.

5.1.3 Simulator Related Classes

Parser.vb

Allows a given frame to be parsed for simulation. When a frame is parsed, each token is checked to make sure it has at least one input, its operands are valid, and that each token has the appropriate number of inputs. Additionally warnings are provided to the user for tokens that do not output to other tokens or tokens that are not connected to any other tokens. A return token will not be marked as lacking an output as this is intentional. Once a frame is successfully parsed without errors, the parser class will instantiate the appropriate *SimOps*, *SimSteps*, and *Simulator* classes. Further details about these classes are provided below. A sub-parser may be instantiated during simulator runtime to handle a function call token.

Simulator.vb

The simulator contains a step for each row of the frame and each step is broken into one operation per token in the given row. The simulator handles execution of each step and stops if a breakpoint or the end of the file is reached. An option is also provided to the user to simulate the frame one step at a time.

SimStep.vb

The parser creates *SimSteps* when a row is successfully parsed. A *SimStep* consists of all the token *Sim* operations on a given row and the breakpoint status of the given row. The *SimStep* is assigned an ID based on the row the step is for. When the *ExecuteOp* function in a *SimStep* is called each *SimOp* in that step will be called and executed.

SimOp.vb

The Simulator Operations Class is a base class from which the following classes inherit:

- **MathOp.vb**: Responsible for: ADDPS, ADDPC, ADDPSC, SUBPS, SUBPC, SUBPSC, INC, DEC.
- **LogicOp.vb**: Responsible for: INV, AND, OR, XOR, SHL, SHR, SHLC, SHRC
- **StatusOp.vb**: Responsible for: SETC, SETN, SETV, SETZ, RSTC, RSTN, RSTV, RSTZ
- **NoOp.vb**: Responsible for: NOP
- **CallOp.vb**: Used to perform Function Calls in function call tokens.
- **FunctionOp.vb**: Responsible for: RETURN
- **DataOp.vb**: Used to pass data out of data elements.
- **InputOp.vb**: Used to pass data out of inputs.

A *SimOp* contains the operands for a single token's operation and the opcode to be performed by the operation. Each type of opcode has a child class that contains specialized operations for that type of operation. For example the *MathOp* class contains operations for addition and subtraction, the *LogicOp* class contains operations that require bitwise manipulation such as logic AND, OR, XOR, NOT and shifts.

Operand.vb

Operands represent any value used by the simulator. Types of operands used in the simulator are:

- Primary Input Operands
- Secondary Input Operands

- Status Bit Input Operands
- Output Operands
- Status Bit Output Operands

Each operand consists of a value and type. The value is converted to and stored as a byte array, the type is stored as a string defined by the public constants in the operand class. When a value is required in its specified form it is converted from a byte array to the appropriate type using its defined type.

OperandException.vb

Can be used to throw a program exception related to an operand, such as in the case that operand types are mismatched.

5.2 Future Development of the IDE

While every effort has been made to create a fully functional IDE and simulator, several future developments and additions can be made to the current IDE. These additions will be detailed in this section.

5.2.1 Loops

The addition of loop capability to the simulator is an important feature defined in the original design. Because the organization and operation of the architecture was developed concurrently with this work, the loop capability will be implemented in the next version. Currently, the IDE only supports loops in their completely unrolled form; in order to include loop functionality in the IDE it will be necessary to modify the functionality of the parser

significantly. A summary of the required changes includes: Implementing non-unique IDs as discussed in the next section, and modifying the way simulator steps are created. The current version of the parser creates all steps for a simulation before the simulation runs, which would be analogous to static or compile time step creation. Iterative loops without a predefined number of iterations will require dynamic or runtime step creation during a simulation.

Additionally, it should be noted that while loops are an important feature in the architecture, they are not always desired. An example of a situation where loops would not be desired is in a real time data intense operation where pipelining of operations is required. Performing real time signal processing in an embedded application, such as live audio or video filtering, would likely require this type of pipelined data processing.

Furthermore, it is proposed that instead of using the *Execute Order ID*, *Execute Order Count* and *Execute After* fields to create loops, which would be time consuming for a user to populate, the *Next Row to Execute* field could be used; it is speculated that this will increase the IDE's usability. In actual hardware or a timing accurate simulator the other fields would still need to be populated. When an export feature for a timing accurate simulator is implemented, as discussed in Section 5.2.6, the compiler (parser) could populate these fields based off of the *Next Row to Execute* field and the implied program flow of the current IDE, discussed in Section 3.4. It is important to emphasize that these fields are not being removed or replaced in the architecture, but simply automatically populated based off of the *Next Row to Execute* field. Upon implementation of this functionality the *Execute Order ID*, *Execute Order Count* and *Execute After* fields will become read-only to minimize confusion in the user interface. By default the *Next Row to Execute* field will be populated for the user with the next row in the layout grid following the current token. The only time the user will need to modify this field is in the event they would like to step back to a previous row for a loop.

5.2.2 Re-annotate IDs and Non-unique IDs

The ability to re-annotate the ID number on tokens, either on a case by case basis or for the entire design is not presently implemented. The IDE currently assigns a unique ID to each token at the time it is placed on the grid. Once an ID has been used it can never be used again, even if the original token with that ID number has been deleted from the design. This was done to allow undo functionality without creating duplicate ID numbers. It was later decided during development of the IDE that it is desired to allow tokens to share ID numbers and keep only their (X, Y) location as a unique identifier, unfortunately due to time constraints this feature was not implemented. An *ElementID* field was added to the property grid for AP tokens to show the concept, but no further development action was taken. The current access functions for tokens rely on unique ID numbers, if non-unique ID numbers were implemented, a major restructuring of both these functions and the way the parser works would be necessary. Non-unique IDs is one way that would allow data to be treated as a variable that can be manipulated cumulatively with an iterative loop by using one value to initialize the loop and another during loop iterations. Currently, the IDE does not support this style of data manipulation, but instead requires a piece of data be explicitly passed from token to token with no option to take input data from multiple locations.

5.2.3 Token Quick View and Print

A token quick-view feature would allow the user to view data and simulation results for a token without selecting the token and viewing the information from the property grid on the right hand side of the IDE. This feature would allow the user to take screen captures similar to the ones shown earlier in this document with call out boxes placed next to each token. Additionally this feature would allow for faster debugging of a given design. Furthermore, the addition of a *Print* option to the File menu allowing a design to be printed,

would eliminate the need for taking screen captures of the IDE and capture an entire design regardless of size. Options should be provided to the user about what to include with the printed design such as call out boxes and simulation data similar to how layers can be enabled and disabled in other computer aided drafting tools.

5.2.4 Data Structures (Data Element Arrays)

Data Elements exist in the current IDE as stand alone entities. The original design indicates the intent to allow data elements to be grouped as arrays into a data structure which can be passed at will back and forth from function instances. At the time of development no detailed description was provided as to how this happens in the underlying hardware. Future research should clearly define how a grouping of data elements into a data structure is performed and how such an entity is referenced in the hardware as well as the IDE.

5.2.5 Additional Opcodes

Implementation of a floating point packing and unpacking opcode would be useful for performing math operations on floating point numbers. Currently, to unpack or pack a floating point number the IDE needs to perform several simple logic operations. These operations result in an operand mismatch, requiring the user to type cast the floating point number to a signed integer in order to properly view the pack or unpack operations. A more eloquent solution would be the addition of pack and unpack opcodes to the architecture. The unpack opcode would need to account for the project bit size representation, and whether the exponent, significand, or sign is desired to be returned as the result. The pack opcode would require the exponent and significand, which could be provided as a signed integer to account for the sign, to pack the floating point number. Furthermore the opcodes should

automatically convert the operand type from a floating point number to an integer and vice-versa.

Implementation of a rotate by N places opcode would be useful for simple math operations where more than a single shift is required. Considering that the current shift opcodes only require a single operand, the number to be shifted, the second operand could be used to indicate the number of places to rotate or left blank for the standard single place bitwise shift.

5.2.6 Partially implemented IDE Features

Export Features

In addition to the currently supported option to export to human readable code, other export features will need to be supported by the IDE: namely the ability for a design to be exported to a future timing accurate simulator and for a given data structure to be exported as a delimited list of numbers. With these features, a user could program other simulators or hardware and plot simulations results in a program such as Microsoft Excel or Matlab. Figure 5.1 shows an example of the current human readable export for a portion of the 32-bit multiplier shown in Figure 4.7.

Copy, Cut, Paste, Select and Find

The current edit menu shows the standard copy, cut, paste, select and find features as disabled. Currently, the user can only select one token at a time and only has the option to view, move or delete a token. Future versions of the IDE should include the ability to cut or copy and paste tokens, in addition to being able to select multiple tokens at a time when using this functionality. A find feature that searches for a specific token in a large design could also prove to be a useful feature.

```
                                  multiplier32.txt
Begin Concurrent
[ Input.0;
        ExecutionOrder#         = 0;
        ExecutionOrder#Count    = 0;
        ExecuteAfterOrder#      = 0;
        StatusBitsSource        = ;
        StatusBitsValues        = C=False N=False V=False Z=False;
        ConditionalExecution    = Unconditional;
        PrimaryOperandSource    = ;
        PrimaryOperandType      = Unsigned Integer;
        PrimaryOperandValue     = 131072;
        SecondaryOperandSource  = ;
        SecondaryOperandType    = ;
        SecondaryOperandValue   = ;
        Operation               = NOP Unconditional;
        Cast                    = Entity IdentificationNumber ExecutionOrderNumber
StatusBits PrimaryOperand;
]
[ Input.1;
        ExecutionOrder#         = 0;
        ExecutionOrder#Count    = 0;
        ExecuteAfterOrder#      = 0;
        StatusBitsSource        = ;
        StatusBitsValues        = C=False N=False V=False Z=False;
        ConditionalExecution    = Unconditional;
        PrimaryOperandSource    = ;
        PrimaryOperandType      = Unsigned Integer;
        PrimaryOperandValue     = 10000;
        SecondaryOperandSource  = ;
        SecondaryOperandType    = ;
        SecondaryOperandValue   = ;
        Operation               = NOP Unconditional;
        Cast                    = Entity IdentificationNumber ExecutionOrderNumber
StatusBits PrimaryOperand;
]
End Concurrent
[ AtomicProcessor.1;
        ExecutionOrder#         = 0;
        ExecutionOrder#Count    = 0;
        ExecuteAfterOrder#      = 0;
        StatusBitsSource        = ;
        StatusBitsValues        = C=False N=False V=False Z=False;
        ConditionalExecution    = Unconditional;
        PrimaryOperandSource    = i0;
        PrimaryOperandType      = ;
        PrimaryOperandValue     = ;
        SecondaryOperandSource  = ;
        SecondaryOperandType    = Unsigned Integer;
        SecondaryOperandValue   = 1;
        Operation               = AND Unconditional;
        Cast                    = Entity IdentificationNumber ExecutionOrderNumber
StatusBits PrimaryOperand;
]
Begin Concurrent
[ AtomicProcessor.2;
        ExecutionOrder#         = 0;
        ExecutionOrder#Count    = 0;
        ExecuteAfterOrder#      = 0;
        StatusBitsSource        = 1;
        StatusBitsValues        = C=False N=False V=False Z=False;
        ConditionalExecution    = Unconditional;
        PrimaryOperandSource    = ;
        PrimaryOperandType      = Unsigned Integer;
        PrimaryOperandValue     = 0;
        SecondaryOperandSource  = i1;
        SecondaryOperandType    = ;
        SecondaryOperandValue   = ;
        Operation               = ADDPS Z=0;
        Cast                    = Entity IdentificationNumber ExecutionOrderNumber
StatusBits PrimaryOperand;
                                       Page 1
```

Figure 5.1: Export to Human Readable Code for the first four tokens of the 32-bit multiplier shown in Figure 4.7.

Undo/Redo

The IDE supports the ability to undo or redo the deletion of a token, or the placement of a new token. Future versions of the IDE should expand on this functionality so that actions such as moving a token and manipulating the data within a token are also tracked and fully reversible with the undo and redo menu options.

Concurrent/Sequential Arrays

Graphical support exists within the IDE for Sequential and Concurrent Arrays, but no means is provided to the user to parse and simulate these types of tokens. Future versions of the IDE should allow for a user to create a concurrent or sequential array of tokens intended for performing repetitive tasks without the need to place multiple tokens. Further information about the planned functionality of concurrent and sequential arrays was provided in Section 3.3.

Save on Close

Prompt the user if they would like to save the currently open projects before the IDE program closes.

Line Numbers

The IDE tracks rows internally for simulation and token placement purposes. The display of these row numbers in the GUI on or next to the breakpoint margin could assist the user once loop functionality is added to the IDE using the *Next Row To Execute* field.

Fixed Point Numbers

Visual Basic does not natively support fixed-point numbers. In order to convert fixed-point numbers between byte arrays and their native format, custom functions would need to be written. While options for fixed-point numbers are presently included in the GUI portion of the IDE, the underlying functionality is not implemented.

Hexadecimal Numbers

The IDE supports displaying an output value as a hexadecimal number, in addition to this feature the IDE needs to support hexadecimal numbers as an input type to simplify creation of bitwise masks for logic operations.

Chapter 6

Conclusion

This work has focused on the development of an IDE and zero-latency timing simulator for the proposed architecture. The IDE has allowed for implementation of sample algorithms on the fine-grained, massively parallel *Pond* architecture. It has been shown that floating point operations, such as multiplication, can be performed on the architecture without the use of dedicated floating point hardware. The concept of each processor performing a single operation before passing a message to its neighbors has been shown to be feasible. With this proof of concept, it can be speculated that further development of the combined IDE and simulator will allow for exploration of more complex algorithms, an example being the Fast Fourier Transform. Future research paths will need not only to focus on algorithm development and use of real world timing delays, but also simulation of defective processing cores, scalability of the architecture, and ultimately a hardware implementation of the design.

Bibliography

[1] Adam Spirer. Pond: A robust, scalable, massively parallel computer architecture. Master's thesis, Rochester Institute of Technology, May 2010.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Amsterdam: Elsevier, 4 edition, 2007.

[3] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005.

[4] David Yeh, Li-Shiuan Peh, Shekhar Borkar, John Darringer, Anant Agarwal, and Wen-mei Hwu. Thousand-core chips [roundtable]. *Design Test of Computers, IEEE*, 25(3):272–278, May / June 2008.

[5] Markus Levy and Thomas M. Conte. Embedded multicore processors and systems. *Micro, IEEE*, 29(3):7–9, May / June 2009.

[6] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, November 2009.

[7] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core opteron processor. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 102–103, 2007.

[8] U.M. Nawathe, M. Hassan, L. Warriner, K. Yen, B. Upputuri, D. Greenhill, A. KuMar, and H. Park. An 8-core 64-thread 64b power-efficient SPARC SoC. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 108–590, 2007.

[9] M. Mehrara, T. Jablin, D. Upton, D. August, K. Hazelwood, and S. Mahlke. Multicore compilation strategies and challenges. *Signal Processing Magazine, IEEE*, 26(6):55–63, November 2009.

[10] M.D. McCool. Scalable programming models for massively multicore processors. *Proceedings of the IEEE*, 96(5):816–831, May 2008.

[11] S. Gal-On and M. Levy. Measuring multicore performance. *Computer*, 41(11):99–102, November 2008.

[12] James Donald and Margaret Martonosi. An efficient, practical parallelization methodology for multicore architecture simulation. *Computer Architecture Letters*, 5(2):14, July / December 2006.

[13] S. KuMar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 105–112, 2002.

[14] Jiang Xu, W. Wolf, J. Henkel, and S. Chakradhar. A methodology for design, modeling, and analysis of networks-on-chip. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 1778–1781 Vol. 2, 2005.

[15] L. Benini and D. Bertozzi. Network-on-chip architectures and design methods. *Computers and Digital Techniques, IEEE Proceedings on*, 152(2):261–272, March 2005.

[16] M. Amde, T. FeliciJan, A. Efthymiou, D. Edwards, and L. Lavagno. Asynchronous on-chip networks. *Computers and Digital Techniques, IEEE Proceedings on*, 152(2):273–283, March 2005.

[17] K. Goossens, J. Dielissen, and A. Radulescu. AEthereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 22(5):414–421, September / October 2005.

[18] D. Bertozzi and L. Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *Circuits and Systems Magazine, IEEE*, 4(2):18–31, 2004.

[19] G. Leary, K. Srinivasan, K. Mehta, and K.S. Chatha. Design of network-on-chip architectures with a genetic algorithm-based technique. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(5):674–687, May 2009.

[20] M. Forsell. A scalable high-performance computing solution for networks on chips. *Micro, IEEE*, 22(5):46–55, September / October 2002.

[21] D.A. llitzky, J.D. Hoffman, A. Chun, and B.P. Esparza. Architecture of the scalable communications core's network on chip. *Micro, IEEE*, 27(5):62–74, September / October 2007.

[22] P.P. Pande, C. Grecu, A. IvaNov, R. Saleh, and G. De Micheli. Design, synthesis, and test of networks on chips. *Design Test of Computers, IEEE*, 22(5):404–413, September / October 2005.

[23] R. Saleh. An approach that will NoC your SoCs off! *Design Test of Computers, IEEE*, 22(5):488, September / October 2005.

[24] H.C. Freitas, F.L. Madruga, M. Alves, and P. Navaux. Design of interleaved multi-threading for network processors on chip. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 2213–2216, 2009.

[25] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv.*, 38(1):1, 2006.

[26] K. Nagami, K. Oguri, T. Shiozawa, H. Ito, and R. Konishi. Plastic Cell Architecture: Towards Reconfigurable Computing for General-Purpose. In *IEEE Symposium on FPGAs for Custom Computing Machines, 1998. Proceedings*, pages 68–77, 1998.

[27] R. Marculescu, U.Y. Ogras, Li-Shiuan Peh, N.E. Jerger, and Y. Hoskote. Outstanding research problems in NoC design: System, microarchitecture, and circuit perspectives. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(1):3–21, January 2009.

[28] Kuei-Chung Chang, Jih-Sheng Shen, and Tien-Fu Chen. Evaluation and design trade-offs between circuit-switched and packet-switched NOCs for application-specific SOCs. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 143–148, 2006.

[29] Gaoming Du, Duoli Zhang, Yukun Song, Minglun Gao, Luofeng Geng, and Ning Hou. Scalability study on mesh based network on chip. In *Computational Intelligence and Industrial Application, 2008. PACIIA '08. Pacific-Asia Workshop on*, volume 2, pages 681–685, 2008.

[30] Huy-Nam Nguyen, Vu-Duc Ngo, and Hae-Wook Choi. Assessing routing behavior on on-chip-network. In *Computer Engineering and Systems, The 2006 International Conference on*, pages 62–65, 2006.

[31] Xinming Duan, Dakun Zhang, and Xuemei Sun. Routing schemes of an irregular mesh-based NoC. In *Networks Security, Wireless Communications and Trusted Computing, 2009. NSWCTC '09. International Conference on*, volume 2, pages 572–575, 2009.

[32] ShiJun Lin, Li Su, Haibo Su, Depeng Jin, and Lieguang Zeng. Design trade-offs in packetizing mechanism for network-on-chip. In *Digital Society, 2009. ICDS '09. Third International Conference on*, pages 316–321, 2009.

[33] J. Hu and R. Marculescu. Communication and task scheduling of application-specific networks-on-chip. *Computers and Digital Techniques, IEEE Proceedings on*, 152(5):643–651, 2005.

[34] A. Leroy, D. Milojevic, D. Verkest, F. Robert, and F. Catthoor. Concepts and implementation of spatial division multiplexing for guaranteed throughput in networks-on-chip. *Computers, IEEE Transactions on*, 57(9):1182–1195, September 2008.

[35] A. Shacham, K. Bergman, and L.P. Carloni. Photonic networks-on-chip for future generations of chip multiprocessors. *Computers, IEEE Transactions on*, 57(9):1246–1260, September 2008.

[36] D. Schinkel, E. Mensink, E. Klumperink, E. van Tuijl, and B. Nauta. Low-power, high-speed transceivers for network-on-chip communication. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(1):12–21, January 2009.

[37] G. Schelle, J. Fifield, and D. Griinwald. A software defined radio application utilizing modern FPGAs and NoC interconnects. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 177–182, 2007.

[38] J. Chan and S. Parameswaran. NoCOUT: NoC topology generation with mixed packet-switched and point-to-point networks. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pages 265–270, 2008.

[39] V.F. Pavlidis and E.G. Friedman. Interconnect-based design methodologies for three-dimensional integrated circuits. *Proceedings of the IEEE*, 97(1):123–140, January 2009.

[40] A. Mejia, M. Palesi, J. Flich, S. KuMar, P. Lopez, R. HolsMark, and J. Duato. Region-based routing: A mechanism to support efficient routing algorithms in NoCs.

Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 17(3):356–369, March 2009.

[41] J.H. Bahn and N. Bagherzadeh. Design of simulation and analytical models for a 2d-meshed asymmetric adaptive router. *Computers Digital Techniques, IET*, 2(1):63–73, January 2008.

[42] M. Palesi, R. HolsMark, S. KuMar, and V. Catania. Application specific routing algorithms for networks on chip. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):316–330, March 2009.

[43] A. Ganguly, P.P. Pande, and B. Belzer. Crosstalk-aware channel coding schemes for energy efficient and reliable NOC interconnects. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(11):1626–1639, November 2009.

[44] Xin Wang, Tapani Ahonen, and Jari Nurmi. Applying CDMA technique to network-on-chip. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(10):1091–1100, October 2007.

[45] S. Murali, D. Atienza, P. Meloni, S. Carta, L. Benini, G. De Micheli, and L. Raffo. Synthesis of predictable networks-on-chip-based interconnect architectures for chip multiprocessors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(8):869–880, August 2007.

[46] H.C. Freitas, T.G.S. Santos, and P.O.A. Navaux. Design of programmable NoC router architecture on FPGA for multi-cluster NoCs. *Electronics Letters*, 44(16):969–971, July 2008.

[47] N. Bagherzadeh and M. Matsuura. Performance impact of task-to-task communication protocol in network-on-chip. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, pages 1101–1106, 2008.

[48] F. Karim, A. Nguyen, and S. Dey. An interconnect architecture for networking systems on chips. *Micro, IEEE*, 22(5):36–45, September / October 2002.

[49] S. Rodrigo, S. Medardoni, J. Flich, D. Bertozzi, and J. Duato. Efficient implementation of distributed routing algorithms for NoCs. *Computers Digital Techniques, IET*, 3(5):460–475, September 2009.

[50] S. Yan and B. Lin. Joint multicast routing and network design optimisation for networks-on-chip. *Computers Digital Techniques, IET*, 3(5):443–459, September 2009.

[51] M. Daneshtalab, M. Ebrahimi, S. Mohammadi, and A. Afzali-Kusha. Low-distance path-based multicast routing algorithm for network-on-chips. *Computers Digital Techniques, IET*, 3(5):430–442, September 2009.

[52] M. Palesi, S. KuMar, and V. Catania. Bandwidth-aware routing algorithms for networks-on-chip platforms. *Computers Digital Techniques, IET*, 3(5):413–429, September 2009.

[53] Se-Joong Lee, Kangmin Lee, Seong-Jun Song, and Hoi-Jun Yoo. Packet-switched on-chip interconnection network for system-on-chip applications. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 52(6):308–312, June 2005.

[54] F. Jafari, M.S. Talebi, A. Khonsari, and M.H. Yaghmaee. A novel congestion control scheme in network-on-chip based on best effort delay-sum optimization. In *Parallel Architectures, Algorithms, and Networks, 2008. I-SPAN 2008. International Symposium on*, pages 191–196, 2008.

[55] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier. High-performance computing: clusters, constellations, MPPs, and future directions. *Computing in Science Engineering*, 7(2):51–59, March / April 2005.

[56] J.M. Rabaey and S. Malik. Challenges and solutions for late- and post-silicon design. *Design Test of Computers, IEEE*, 25(4):296–302, July / August 2008.

[57] Wen-mei Hwu, K. Keutzer, and T.G. Mattson. The concurrency challenge. *Design Test of Computers, IEEE*, 25(4):312–320, July / August 2008.

[58] T. Austin, V. Bertacco, S. Mahlke, and Yu Cao. Reliable systems on unreliable fabrics. *Design Test of Computers, IEEE*, 25(4):322–332, July / August 2008.

[59] Lei Zhang, Yinhe Han, Qiang Xu, Xiao wei Li, and Huawei Li. On topology reconfiguration for defect-tolerant NoC-based homogeneous manycore systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(9):1173–1186, September 2009.

[60] M. Lammie, P. Brenner, and D. Thain. Scheduling grid workloads on multicore clusters to minimize energy and maximize performance. In *Grid Computing, 2009 10th IEEE/ACM International Conference on*, pages 145–152, 2009.

[61] P. Chaparro, J. Gonzalez, G. Magklis, Cai Qiong, and A. Gonzalez. Understanding the thermal implications of multi-core architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 18(8):1055–1065, August 2007.

[62] Jingcao Hu and R. Marculescu. Energy- and performance-aware mapping for regular NoC architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(4):551–562, April 2005.

[63] N. Banerjee, P. Vellanki, and K.S. Chatha. A power and performance model for network-on-chip architectures. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 1250–1255, 2004.

[64] A. Sarathy, A. Louri, and A.K. Kodi. Low-power low-area network-on-chip architecture using adaptive electronic link buffers. *Electronics Letters*, 44(8):512–513, April 2008.

[65] Kangmin Lee, Se-Joong Lee, and Hoi-Jun Yoo. Low-power network-on-chip for high-performance SoC design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(2):148–160, February 2006.

[66] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *Computers Digital Techniques, IET*, 3(5):398–412, September 2009.

[67] T. Simunic, S.P. Boyd, and P. Glynn. Managing power consumption in networks on chips. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(1):96–107, January 2004.

[68] R. Iris Bahar, Dan Hammerstrom, Justin Harlow, William H. Joyner Jr., Clifford Lau, Diana Marculescu, Alex Orailoglu, and Massoud Pedram. Architectures for silicon nanoelectronics and beyond. *Computer*, 40(1):25–33, January 2007.

[69] Shuo Wang, Lei Wang, and F. Jain. Dynamic redundancy allocation for reliable and high-performance nanocomputing. In *Nanoscale Architectures, 2007. NANOSARCH 2007. IEEE International Symposium on*, pages 1–6, 2007.

[70] F. Martorell and A. Rubio. Defect and fault tolerant cell architecture for feasible nano-electronic designs. In *Design and Test of Integrated Systems in Nanoscale Technology, 2006. DTIS 2006. International Conference on*, pages 244–249, 2006.

[71] N.Z. Haron and S. Hamdioui. Emerging crossbar-based hybrid nanoarchitectures for future computing systems. In *Signals, Circuits and Systems, 2008. SCS 2008. 2nd International Conference on*, pages 1–6, 2008.

[72] Shuo Wang and Lei Wang. A defect-tolerant memory nanoarchitecture exploiting hybrid redundancy. In *Nanotechnology, 2008. NANO '08. 8th IEEE Conference on*, pages 707–710, 2008.

[73] Shanrui Zhang, Minsu Choi, and Nohpill Park. Defect characterization and yield analysis of array-based nanoarchitecture. In *Nanotechnology, 2004. 4th IEEE Conference on*, pages 50–52, 2004.

[74] Trong Tu Bui and T. Shibata. A scalable architecture of associative processors employing nano functional devices. In *Ultimate Integration of Silicon, 2009. ULIS 2009. 10th International Conference on*, pages 213–216, 2009.

[75] Shanrui Zhang, Minsu Choi, and N. Park. Modeling yield of carbon-nanotube/silicon-nanowire FET-based nanoarray architecture with h-hot addressing scheme. In *Defect and Fault Tolerance in VLSI Systems, 2004. DFT 2004. Proceedings. 19th IEEE International Symposium on*, pages 356–364, 2004.

[76] J.A. Casas, J.M. Moreno, J. Madrenas, and J. Cabestany. A novel hardware architecture for self-adaptive systems. In *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, pages 592–599, 2007.

[77] Ian Foster. *Designing and Building Parallel Programs*. Addison Wesley, 1995. Available Online: `http://www.mcs.anl.gov/˜itf/dbpp/`.

[78] Open MP: The OpenMP API Specification for Parallel Programming, June 2010. `http://openmp.org/wp/`.

[79] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, 7 edition, 2005.

[80] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Professional, 2005.

[81] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware / Software Approach*. Morgan Kaufmann, San Francisco, 1999.

[82] IEEE Task P754. *IEEE 754-2008, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, NY, USA, August 2008.

[83] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford, New York, 2000.

Appendix A

IDE User Guide

A.1 Menus

A.1.1 File Menu

New

Opens a new tab and creates a new blank project within the tab.

Open

Opens a file chooser window for the user to select a previously saved *.pnd file. Once a valid file is selected, it is opened in a new tab for the user to edit or simulate. Files saved with the export to human readable format (*.prd) cannot be opened in the IDE.

Save, Save As, Save All

Opens a file browser window for the user to select a filename for a previously unsaved file or when the *Save As* option is selected. Saves the file in a *.pnd binary file, with the new or existing file name. *Save All* is not implemented in this version of the IDE.

Export

Exports a project to another format. Currently, only *Human Readable* format is available, *Comm Sim* is to be implemented in a later version of the IDE.

Close, Close All

Closes the selected tab or in the case of *Close All*, closes all open tabs. The user is prompted if they would like to save the project before each tab is closed.

Exit

Exits the program. At this time, the user is not prompted about saving the currently open projects; any unsaved work will be lost.

A.1.2 Edit Menu

Undo, Redo

Allows the user to undo or redo a previous operation. Currently, placement of new tokens and deletion of existing tokens are undo / redo compatible operations. Other operations such as moving a token and editing the information within a token's property grid will be implemented in future versions of the IDE.

Cut, Copy, Paste

Cut, Copy or Paste the selected tokens. Currently, this functionality is not implemented in the IDE.

Select All, Find

Select all tokens or Find a specific token based on location or ID. Currently, this functionality is not implemented in the IDE.

Delete

Delete the currently selected token. Any lines previously attached to the deleted token will become red stubs, indicating a disruption in data flow. The ID's of deleted tokens are not reused within the project. The delete operation is tracked and can be undone.

Design Width and Design Height

Opens a prompt for the user to enter a new width or height for the project, the new value must be an integer number representing the desired number of grid spaces. The layout grid can be made larger or smaller with these options, but it is not possible to make the design smaller then the currently placed tokens, thus preventing tokens from falling off of the grid.

A.1.3 View Menu

Zoom In, Zoom Out

The IDE supports two zoom levels, a close up view with numbered tokens and a zoomed out view with small unnumbered tokens. The two modes can be toggled with the *Zoom In* and *Zoom Out* menu options, which automatically enable or disable based on which view is currently provided.

Background Color

The IDE provides the option of a black background with a white grid or a white background with a black grid. The black background is intended for use when laying out a design and

the white background is useful for screen captures.

Project Properties

The project properties dialog is opened when this menu item is selected. Once within the project properties dialog, options are provided to change the number of input tokens displayed for the project, edit the function name, which can be different then the file name, and to edit the function description. Additionally the bit size representation of the project can be changed from the project properties dialog; options are provided for 16, 32 or 64 bit wide operands. An example project properties dialog is shown in Figure A.1.

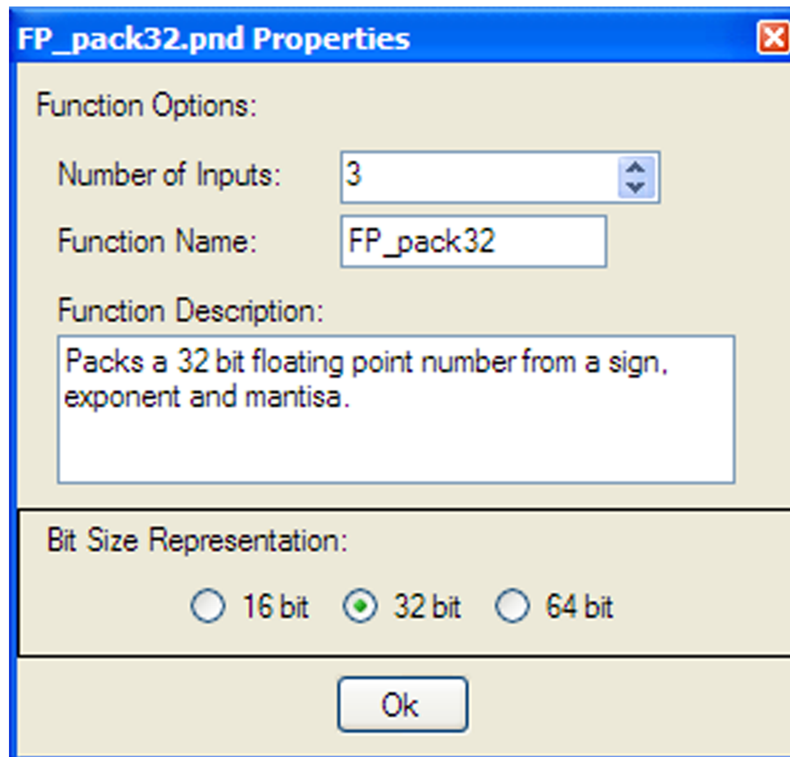


Figure A.1: Example Project Properties Dialog from the 32-bit floating point number packing function.

A.1.4 Tools Menu

Simulate

Parses the project on the currently selected tab and displays an error and warning list to the user, as shown in Figure A.2. If no errors were found in the project then a simulation is started and run until a breakpoint or the end of the project is encountered. By selecting this option the project is put into simulation mode, which allows simulation data to be displayed for each token when selected.

Simulate Step

If the project is not yet in simulation mode then the project is parsed and the standard error/warning list is provided to the user. If no errors are found the project is simulated as normal until a breakpoint or the end of the project is encountered. If a simulator is already active, but paused at a breakpoint, the project is stepped forward one step (row) at a time each time this option is selected. A yellow highlight bar indicates the current row that the simulator is paused on. An example of this is shown in Figure A.8

A.1.5 Help

Pond Help

The current version of the IDE does not open a help file when this option is selected. This Appendix is the usage and help manual for the user.

About...

Displays an *About this Program* box to the user.

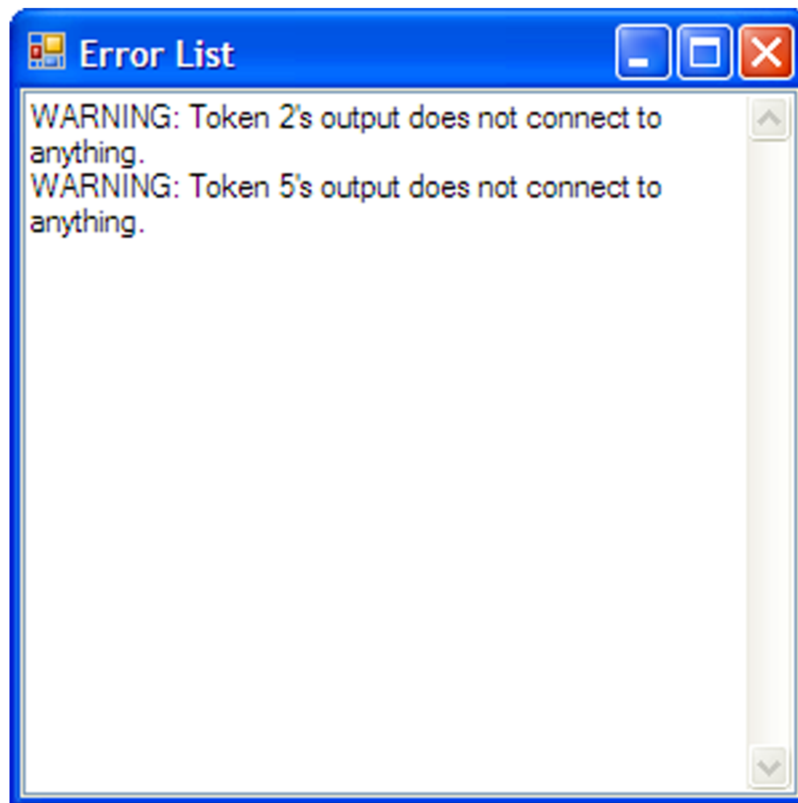
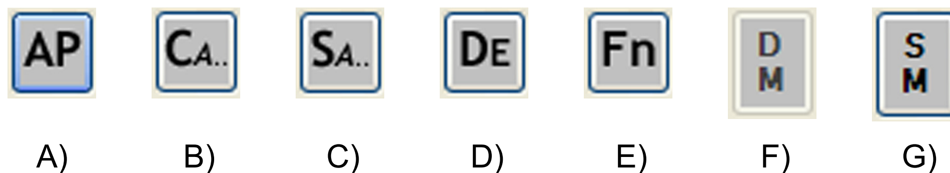


Figure A.2: Sample Error and Warning list from 32-bit floating point packing operation.

A.2 Left Hand Tool Bar

The left hand side tool bar is composed of seven buttons, five of which allow the user to place tokens on the layout grid and two to indicate the IDE's mode. These buttons are shown in Figure A.3. The AP button allows the user to place a single atomic processor on to the layout grid; each atomic processor is capable of a single operation. Tokens placed horizontally adjacent to each other, more simply referred to as in the same row, execute concurrently during simulation. The CA and SA buttons allow for the placement of concurrent and sequential arrays; neither of these tokens have simulation functionality in the current IDE. The DE token allows for placement of data elements and the Fn button allows for placement of function call tokens.

The additional two buttons on the left hand side tool bar allow for toggling between design mode, indicated by the DM button, and simulation mode, indicated by the SM button. While in design mode, the user is free to place tokens on the layout grid, move tokens around, delete tokens and manipulate the properties defined for each token. Once in simulation mode, the project is placed into a read-only state which allows the user to see the properties and result values for each token, but not edit the layout grid. The left hand side tool bar also provides the grid location of the selected token, as well as the project bit size representation, as displayed by the two labels on the mid and lower left respectively.



A) B) C) D) E) F) G)

Figure A.3: Toolbar Buttons: A) Atomic Processor, B) Concurrent Array, C) Sequential Array, D) Data Element, E) Function Call, F) Design Mode, G) Simulation Mode.

A.3 Project Files

When a project is created with the IDE it is saved to as *.pnd file. It is recommended for easy use of the function call ability of the simulator, to place all project files in the folder “PondEnv” on your C drive. This is important due to the way the file paths are stored when a function call token is used. Failure to follow this suggestion may result in an inability to open project files on another workstation. Relative paths within the IDE are properly supported in Windows 7, but are incorrect in Windows XP SP3. Use of the above suggestion will circumvent any of these issues within the IDE.

A.4 The Property Grid

The property grid allows for manipulation of all properties associated with a token other than: location on the layout grid and the unique ID number assigned to a token upon its creation. To open the property grid, left click on any token on the layout grid. An example of the property grid is shown in Figure A.4. In the lower portion of the property grid a description is provided for the currently selected field. During the use of a function call token the function description and input descriptions are displayed in this location when the appropriate fields are selected.

Within the property grid, drop down fields may be populated by selecting the appropriate value with the mouse cursor or typing the first letter of the desired value; if multiple values exist with the same first letter, repeatedly pressing the first letter of the desired value will toggle through all possible choices. Clicking the hide button in the upper right hand corner of the window will minimize the property grid. When viewing the property grid for a newly created token, several of the properties are populated with default values to minimize the number of fields the user is required to fill out.

The standard properties defined for a token are broken down based on type within the property grid. Five categories exist: *Broadcast*, *Execute Order*, *Execution*, *Operands* and *Status Bits*. Currently, the fields defined within the *Broadcast* and *Execute Order* categories have no implementation with the simulator portion of the IDE; their purposes have been discussed in Chapter 2 and some of them are used for the export to Human Readable feature.

The *Execution* category contains three fields. Settings for *Conditional Execution* and *Conditional Operation* are described in Section A.6. The *Operation Code* field sets the operation that the processor is to perform. A description of each of the possible instructions is available in Table 2.2.

Moreover, the *Operands* category contains seven fields. The first field, *Element ID*, has been added for demonstration purposes only and does not have any function in the IDE. The other fields allow the user to define inputs for the given token. The words *First* and *Second* have been omitted in the following labels, as their purpose only serves to differentiate between the two inputs. If it is desired to have an input take the value of another processor's output, the **OpSource* field can be used by entering the unique ID number shown on the face of the source processor. If using an input token as the source the number must be preceded by an "i". If the source value type is as desired the **OpType* field may be left blank, otherwise it may be set to the desired type; this does not affect the actual data in any way, but simply changes the format it is displayed in within the results table during simulation. Furthermore, if the user would like to use a predefined value for an input to the token, the **OpType* and **OpValue* fields must be populated with the desired type and value for the input. The **OpSource* field should be left blank in this case, otherwise it will take precedence.

Finally, the *Status Bits* category shows the predefined values for each of the status bits

and the *StatusBitsSource* to override them with if desired. The *StatusBitsSource* field behaves in a similar manner to the *FirstOpSource* and *SecondOpSource* fields, where the unique ID shown on the face of the desired source processor is used as the value. The *StatusBitsSource* field should always be used to indicate the status bits used for a conditional execution or operation, otherwise the condition will always execute the same way based off of the predefined values.

A.5 Tokens on the Layout Grid

Tokens are placed on the layout grid by selecting the appropriate button from the left hand side tool bar. Once tokens have been placed on the layout grid, they are assigned a unique ID number, which can not be changed or reused during the life of the project. If the user desires to move a token to a different location, they can right click the token and it will unsnap from the layout grid and on to the mouse cursor. Once the token is dragged to the desired new location, a single right or left click will snap the token back down to the grid. To delete a token, select it by left clicking on it and then select the *Delete* option from the *Edit* menu.

Resizing the layout grid can be done with the *Design Height* and *Design Width* options provided under the *Edit* menu. By default a design is twenty grid spaces wide and twenty grid spaces high. These sizes can be altered at anytime while in design mode, but the design may not be made smaller than the extents of the already placed tokens.

Lines are drawn on the layout grid to connect two tokens that pass information between each other. Lines are automatically generated when the operand source or status bits source fields are populated in the property grid for a token. When a token is moved on the layout grid, the lines will automatically update their positions based on the new token location.

Furthermore, the lines for the currently selected token become highlighted and slightly

Atomic Processor 1

Hide

Property	Value
Broadcast	
Bcast_1stOp	True
Bcast_2ndOp	False
BcastStatus	True
CastType	Entity
Execute Order	
ExecuteAfter	0
ExecuteOrderCount	0
ExecuteOrderID	1
NextRowToExecute	2
Execution	
ConditionalExecution	Unconditional
ConditionalOperation	Unconditional
OperationCode	ADDPS
Operands	
ElementID	1
FirstOpSource	
FirstOpType	Unsigned Integer
FirstOpValue	10
SecondOpSource	
SecondOpType	Unsigned Integer
SecondOpValue	10
Status Bits	
CarryBit	False
NegativeBit	False
OverflowBit	False
StatusBitsSource	
ZeroBit	False

OperationCode
The op code for this processor.

Figure A.4: IDE Property Grid

shifted to the left or right, this is for easier viewing when several lines are drawn on the grid. When deleting a token, if a receiving token is deleted the respective lines are deleted with the token. If a source token is deleted, then the lines on the receiving tokens become red stubs to indicate the interrupted data flow. Figure A.5 provides examples of highlighting and the red stub remaining when a source token is deleted.

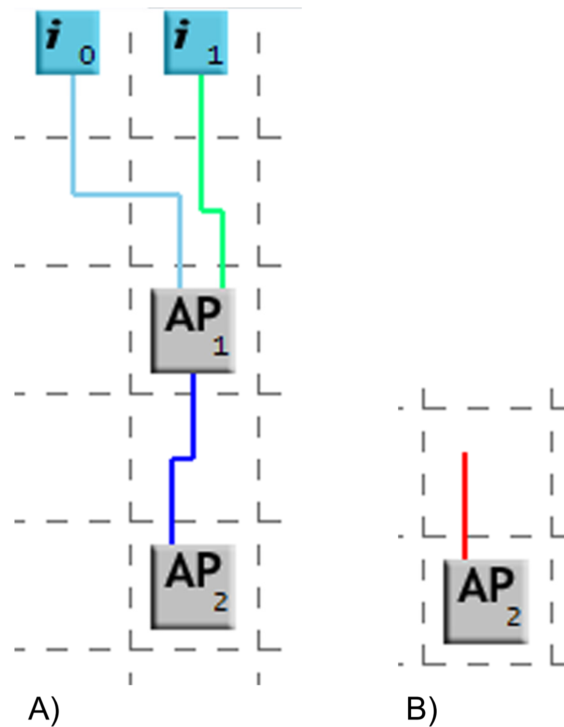


Figure A.5: A) The lines coming into AP1 are highlighted for easier reading, the line exiting AP1 is in its normal unhighlighted state. B) The source token for AP2 has been deleted, a red stub remains.

A.6 Conditional Tokens

Conditional tokens are created in the IDE by setting either the *ConditionalExecution* or *ConditionalOperation* fields in the property grid for a token. The difference between each is the way a token passes messages during simulation. A conditional execution that fails to

meet its condition will not pass a message, which blocks all tokens connected to it further down the layout grid from executing. A conditional operation that fails to meet its condition will pass its primary operand, unaltered, to the tokens connected to it further down the grid; this allows for the proceeding tokens to still execute.

When a token is set to be conditional its color on the layout grid changes to yellow and in the zoomed in mode a conditional symbol appears on the face of the token. During simulation a conditional token will change color based on if its condition is met. When a token's condition is met the color changes to green, conversely the token changes to red when the condition is not met. Figure A.6 shows an example of two atomic processors which have conditions associated with them.

Conditional execution or operation tokens base their condition on the value of one of the four status bits defined for the processor. Status bits are provided for: *Carry* (C), *Negative* (N), *Overflow* (V) and *Zero* (Z). Conditions can check for a value of 0 or 1 for each of these. When a status bit source is provided to the processor, the received status bits are checked for the condition, otherwise the status bits predefined within the processor are used.

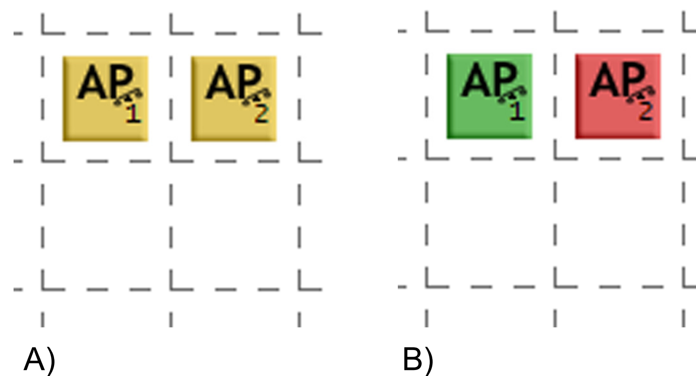


Figure A.6: Conditional Tokens A)Before Simulation tokens are yellow, B)During Simulation tokens change green or red based on if the condition is met.

A.7 Breakpoints

Breakpoints are indicated in the IDE by a red dot shown in the left hand margin. A breakpoint is set or unset by the user by simply clicking in the gray margin for the row they would like to toggle the breakpoint on. Figure A.7 shows an example of a project stopped at a breakpoint during simulation. A yellow bar highlights the row that the simulator has stopped on. It can be seen that conditional token AP2 has not yet changed color based on its execution. Additionally, simulation results are not available for the row highlighted in yellow or any of the rows bellow the highlighted row. Figure A.8 shows the same simulation after it has been stepped forward by one step using the *Simulate Step* option under the *Tools* menu. It can now be seen that AP2 has turned red, indicating its condition for execution has not been met.

A.8 Function Calls

Function call tokens have special fields defined within the property grid which are dynamically updated based on the function being called. When a user creates a function call token, the first property that needs to be defined is the *FuncFileName*. Once the user has browsed to the appropriate file in the "C:/PondEnv" folder, the property grid will update to reflect the requirements of the specified function. Figure A.9 shows an example function call property grid. In this case the called function has three inputs, which were displayed after the file name field was populated. Each of the inputs shows a unique description in the help bar based on what was defined within the input tokens of the function definition. Function calls may have a maximum of ten inputs in the current IDE.

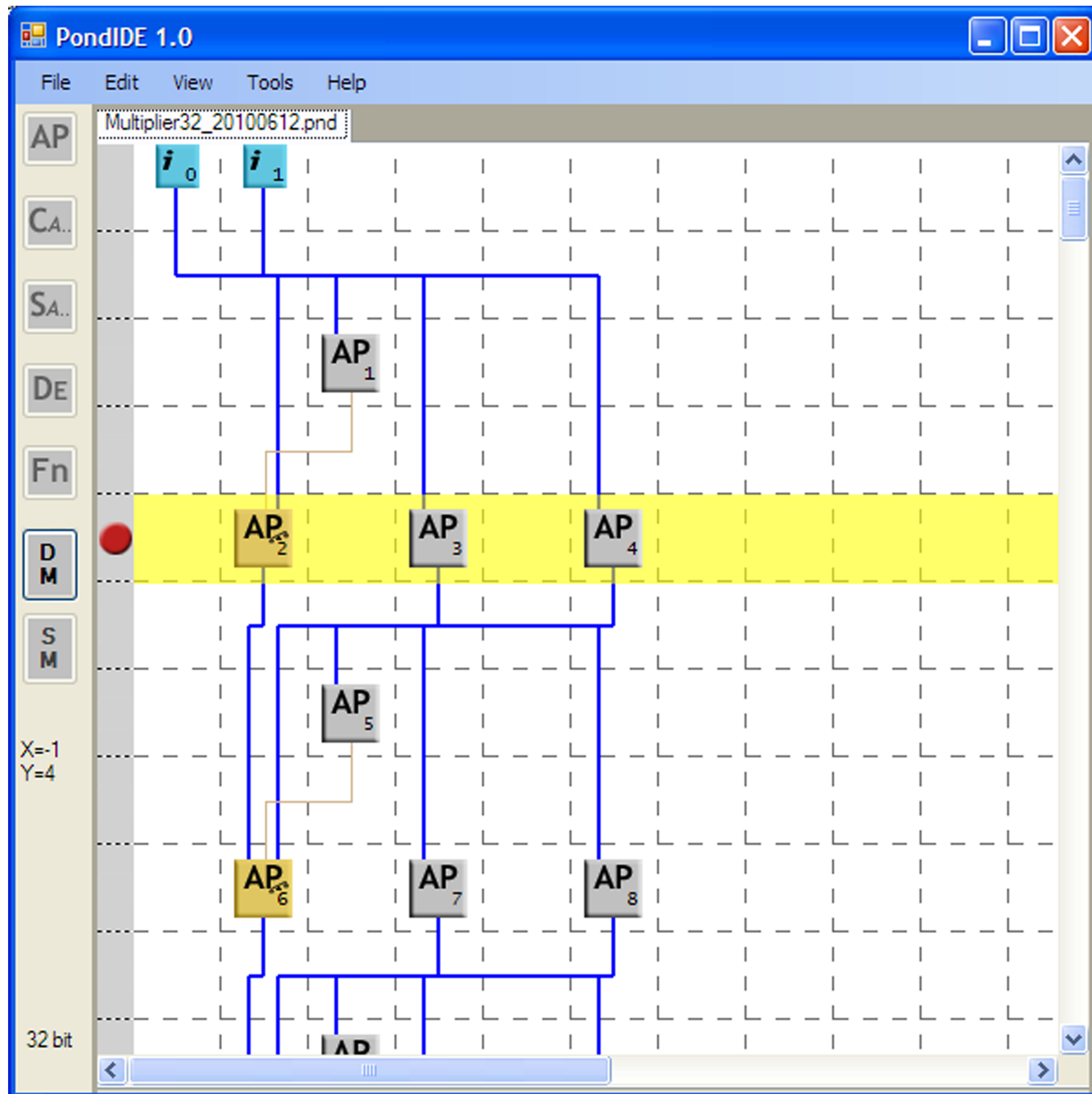


Figure A.7: Example of the 32-bit Left Shift Multiplier stopped at a breakpoint.

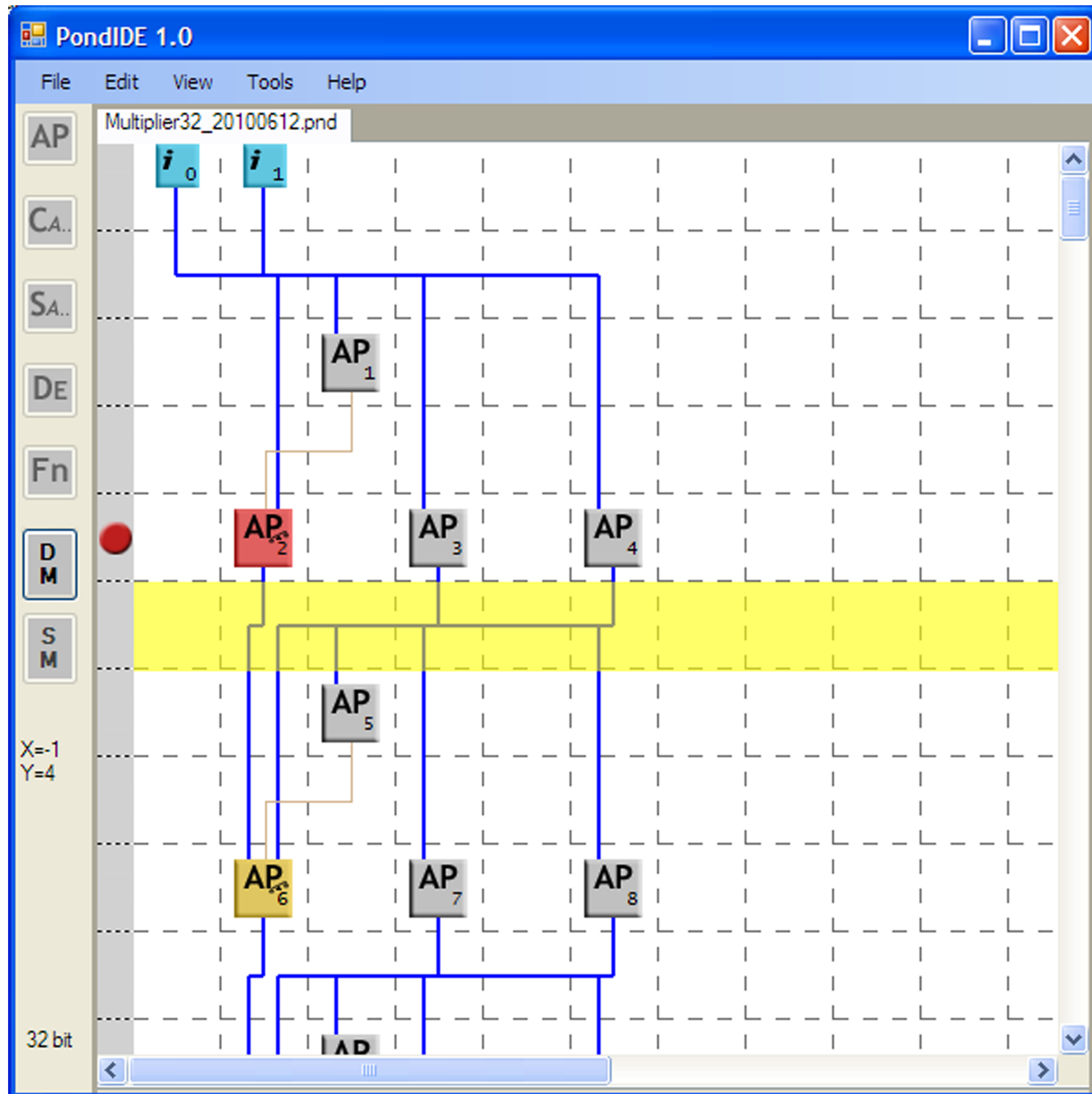


Figure A.8: Example of the 32-bit Left Shift Multiplier stepped one step forward after a breakpoint.

Function Call 10

Hide

Broadcast	
Bcast_1stOp	True
Bcast_2ndOp	False
BcastStatus	True
CastType	Entity
Execute Order	
ExecuteAfter	0
ExecuteOrderCount	0
ExecuteOrderID	0
Execution	
ConditionalExecution	Unconditional
StatusBitsSource	
Filename	
FuncFileName	C:\PondEnv\FP_pa
FunctionName	FP_pack32
Inputs	
Input_0	7
Input_1	8
Input_2	9

Input_2
FP Mantisa: Significand or Mantisa for the floating point number.

Figure A.9: Example function call property grid for the Floating Point pack operation called in the floating point multiplication program.

Appendix B

IDE Development Guide

The IDE was developed using Microsoft Visual Basic 2008, Version 9.0.21022.8 RTM, with Microsoft .NET Framework Version 3.5 SP1. Development was performed under Microsoft Windows XP SP3 and Microsoft Windows 7.

Each of the Visual Basic classes are described in detail in Chapter 5. Additionally, recommended improvements to the IDE are covered in Section 5.2. Further information regarding development and inner workings of the IDE can be found in the comments provided within each visual basic class for the project. These comments include a function header for each sub or function defined within the classes. Each function header comment provides a short description of what the function does, along with descriptions of each input argument and return.