5-1-2010

Pond: A Robust, scalable, massively parallel computer architecture

Adam Spirer

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Pond: A Robust, Scalable, Massively Parallel Computer Architecture

by

Adam R. Spirer

A Master’s 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 Phillips, Associate Professor
Committee Member, Department of Electrical and Microelectronic Engineering

Dr. Sohail 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
May, 2010
Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Pond: A Robust, Scalable, Massively Parallel Computer Architecture

I, Adam R. Spirer, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

___________________________________________
Adam R. Spirer

___________________________________________
Date
Acknowledgments

Thank you to my family and my colleagues for supporting me in this research and the writing of this document, specifically to my advisor Dr. Dorin Patru and my committee members for their invaluable feedback.
Abstract

_Pond: A Robust, Scalable, Massively Parallel Computer Architecture_

Adam R. Spirer

_Supervising Professor: Dr. Dorin Patru_

A new computer architecture, intended for implementation in late and post silicon technologies, is proposed. The architecture is a fine-grained, inherently parallel system consisting of a large grid of thousands or millions of simple _atomic processors_ (APs) employing a simple instruction set. Each AP is configured as either a program instruction or data storage element. These elements are organized into logical entities, analogous to traditional programming functions/methods and data structures. Programming work is underway to compile and run programs from traditional sequential code where parallelism is automatically discovered at the high level on both instruction level and function level, and integrated into the object code that is then sent to the processor. The result is a massively parallel architecture that fully exploits instruction and thread-level parallelism. The architecture design is presented, in-progress work involving conversion of existing code is discussed, and examples are shown to indicate the speedup potential that exists in this new architecture when compared to current architectures.
Contents

Acknowledgments ................................................................. iii
Abstract ........................................................................ iv

1 Background and Motivation .................................................. 1

2 Architecture .................................................................... 3
  2.1 Organization ................................................................. 3
  2.2 Instruction Set .............................................................. 6
  2.3 Implementation of Compound Operations ......................... 10

3 Communications ................................................................. 11
  3.1 Handshaking ................................................................. 11
  3.2 Message Passing ........................................................... 15
  3.3 Send and Receive Layers ................................................ 16

4 Operation ........................................................................ 19
  4.1 Entity Movement ........................................................... 19
  4.2 Entity Abutment ............................................................ 21
  4.3 Instruction Execution ..................................................... 22
  4.4 Function Calls ............................................................ 24
  4.5 Input/Output ................................................................. 26
  4.6 Loops ........................................................................ 27
  4.7 Exception Handling ...................................................... 28

5 Performance Evaluation ...................................................... 30
  5.1 Machine Cycle .............................................................. 30
  5.2 Minimum and Maximum Execution Times ......................... 30
  5.3 Results ...................................................................... 35
    5.3.1 Integer Multiplication ............................................... 35
    5.3.2 Integer Division ...................................................... 38
A.3 Functions .......................................................... 93
  A.3.1 wnd_functions.c .............................................. 93
  A.3.2 ap_grid_ui.c ..................................................... 94
  A.3.3 ap_comm.c ....................................................... 95
  A.3.4 sim_engine.c ..................................................... 96
List of Tables

2.1 Configuration and data processing fields for an AP .................................. 7
2.2 Atomic Processor Instruction Set ......................................................... 9
3.1 Handshake codes for atomic processor communications ......................... 13
3.2 Transmit directions for global (G), beamed (B), and entity (E) casts .......... 14
3.3 Message buffer format for result message. All numbers represent bits. ...... 15
3.4 Message buffer format for coded message types; codes are described in Table 3.5. All numbers represent bits. .......................................................... 15
3.5 Message codes and descriptions for non-result message types ............... 17
5.1 Summary of performance metrics, measured in cycles ............................ 35
6.1 Variables table generated for data structure for vector addition code ....... 51
6.2 Variables table generated for data structure for vector addition code ....... 53
6.3 Vector addition machine code for function definition elements ............... 55
6.4 Vector addition code machine code for data structure elements ............. 56
6.5 Fibonacci series machine code for function definition elements ............. 58
6.6 Fibonacci series machine code for data structure elements .................. 59
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A simple representation of the atomic processor sea.</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>An example sea of APs, populated with functions and data.</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>Message and handshake buffers in an atomic processor.</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>Timeline of handshake process; AP $x$ is sending a message to its neighbor, AP $y$.</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>Propagation of communication casts in the sea.</td>
<td>14</td>
</tr>
<tr>
<td>3.4</td>
<td>Send and receive layers of communication buffer circuitry.</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Atomic processor moving around an entity; the dashed line indicates the desired path, and the solid lines represent the redirected path the entity takes.</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Timeline of an instruction cycle, including communication broadcasts.</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>Input/output interface at border of sea.</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>Examples of loop handling in the architecture: program flow for loop iterations created as separate functions, program flow for parallelizable loop iterations executed within a single function, and program flow for sequential loop iterations executed within a single function.</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Examples of best case 100% parallelizable code, worst case 100% parallelizable code, best case 100% sequential code, and worst case 100% sequential code.</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>Example multiplication superentity using left-shift algorithm.</td>
<td>37</td>
</tr>
<tr>
<td>5.3</td>
<td>Example integer division superentity.</td>
<td>39</td>
</tr>
<tr>
<td>5.4</td>
<td>Example floating point multiplication superentity, including use of integer multiplication function for operating on significands.</td>
<td>41</td>
</tr>
<tr>
<td>6.1</td>
<td>Vector addition code as superentity, derived from machine code representation.</td>
<td>57</td>
</tr>
<tr>
<td>6.2</td>
<td>Fibonacci series code as superentity, derived from machine code representation.</td>
<td>60</td>
</tr>
<tr>
<td>6.3</td>
<td>Vector addition code superentity.</td>
<td>62</td>
</tr>
<tr>
<td>6.4</td>
<td>Fibonacci series code superentity.</td>
<td>64</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>7.1</td>
<td>Software screen capture of Simulator window with loaded processor sea.</td>
<td>67</td>
</tr>
<tr>
<td>7.2</td>
<td>Software screen capture of AP Record view.</td>
<td>68</td>
</tr>
<tr>
<td>7.3</td>
<td>Software screen captures of global cast propagation; consecutive machine cycles are shown.</td>
<td>69</td>
</tr>
<tr>
<td>10.1</td>
<td>Example of multiple sea sizes on the same wafer</td>
<td>78</td>
</tr>
</tbody>
</table>
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 [1]. 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 [2–7]. 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 [8–11].

As pointed out by Hennessy and Patterson in [1], 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, [12–24]. Marculescu et al. in [25] 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 [26–52]. Dongarra et al. explore the potential symbiosis between networks on chip and multicore processors in [53].

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 [54], and Wenmei
et al. in [55]. 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 [56]. Lei Zhang et al. address reliability and fault tolerance in networks on chip in [57]. Power dissipation will have to be mitigated starting at the system level. This is already being considered in multicore processors [58, 59], and in networks on chip [60–65]. Nano architectures attempt to specifically address the aforementioned challenges posed by late and post silicon technologies [66–74].

In this thesis, an architecture is proposed, which can efficiently use a few hundred to multi-billion cores. Its implementation is cost effective in late and post silicon technologies, resilient to component failures, massively parallel, and can support a high degree of concurrency without a radical paradigm shift in programming. The architecture shares traits with multicore processor architectures, networks on chip, nano architectures, massively parallel architectures, resilient and fault tolerant architectures, reconfigurable computing, data flow architectures, and neural networks. These similarities and differences will be discussed after the proposed architecture is presented.

In Chapter 2, the organization of the architecture is covered, followed by the communications architecture in Chapter 3, operation in Chapter 4, performance evaluation in Chapter 5, programming in Chapter 6, architectural features and benefits in Chapter 8, related work in Chapter 9, and future work in Chapter 10.
Chapter 2

Architecture

As discussed in Chapter 1, a primary question in developing an effective parallel computing system is the number of cores. Both extremes have been observed; some architectures implement a small number of complex cores, while others implement a large number of simple cores. The proposed architecture takes the latter approach and implements a fine-grained system with a large number of simple cores.

2.1 Organization

Figure 2.1 shows the top-level layout of processing cores, called atomic processors (APs) for this architecture.

As shown, the architecture is composed of a grid-like sea of APs. An AP at a given time will either act as a program instruction (including associated operands), or a storage element containing one data word. Each AP has all the hardware it needs to execute any instruction, theoretically allowing for all available instruction-level parallelism to take place. Programs on the architecture are logically organized into function definitions (FD) consisting of multiple APs containing instructions, which are called to execute when needed. At runtime, function definitions are called to execute as function instances (FI) which are copies of the corresponding function definition, and are capable of operating on data. Groups of data words are also logically organized into data structures (DS). These
Figure 2.1: A simple representation of the atomic processor sea.

logically-organized groups of APs are collectively called entities. These entities are capable of “moving” through the sea of processing elements by way of transferring their individual instructions and data words (entity elements) to adjacent APs, allowing them to interact with other entities. Figure 2.2 shows an example of what a populated small processing sea might look like (actual sea and program/data sizes will be significantly larger to reflect the application).

Each AP is capable of storing both a function definition element, and a function instance or data structure element, at the same time. This breaks down the sea into two logical layers: the definition layer (function definitions) and the execution layer (function instances and data structures). Composed of common high-level programming constructs—function definitions, instances, and data structures—execution of programs takes place in much the same way as in traditional architectures.

Instantiation of the function is equivalent to the function call in high-level programming, and results in the function definition copying its elements into the execution layer of its
APs, creating a function instance. The function instance can then move and interact with data structures, which are also present in the execution layer. Interaction between function instances and data structures is preceded by a move and abut process, in which the function instance locates the data structure of interest and abuts it, creating a *superentity* in which instructions can fetch data as needed.

Each AP is functionally identical and operationally independent. In addition to storing operands and processing circuitry, configuration information is stored—information to identify the entity number, individual entity element number, execution order number (to indicate when the instruction can execute), and other information generated at compile time. Table 2.1 and shows all the independent configuration fields present in a single AP, as well as operand/data fields. These fields are duplicated for the standby layer (for function definitions) as well as the execution layer (for function instances and data structures). The
only exceptions noted are that the entity type field is slightly different between the two layers, and data operand and some other execution-time values are not present in the standby layer. Development is currently being done using a sea of 64 K atomic processors, which is considered to satisfy the needs of an embedded computer system. Field bit widths are also shown in Table 2.1 for 1 Tera atomic processors, which is considered to satisfy the needs of a desktop computer system.

The fields presented in Table 2.1 provide space for all necessary data, instruction operation code, execution conditions, and the information necessary to exploit the parallelism and concurrency discovered at compile time. This also allows for concurrency in instructions within a function instance, allowing for convergence back to sequential operation if needed. In addition, all APs are identified physically and logically. In terms of instruction execution once operands have been loaded (which will arrive via a communications broadcast from another AP in the sea; more on this in Chapter 3), each AP is independent and does not require the resources of any other AP. APs are synchronous blocks, but the entire sea can be asynchronous, as there is no specific need for neighboring APs to operate synchronously. This eliminates the need for global clock synchronization, a common design issue in large circuits. While implementing this architecture via a globally asynchronous, locally synchronous strategy is possible, there are many issues to address in terms of the asynchronous interfaces among blocks [75–78]. In silicon technologies, fields could be implemented with SRAM cells similar to the way FPGA configuration bits are held (this is a topic for future work and is discussed in Chapter 10). Note that the data widths for ID numbers, execution order, and execution repeat values are defined by the sea size; however, a 64K sea size is assumed in this presentation.

### 2.2 Instruction Set

Table 2.2 shows the instruction set for each AP in the sea, and their associated operation
Table 2.1: Configuration and data processing fields for an AP

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits, 64K</th>
<th>Bits, 1T</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity Type (Standby)</td>
<td>1</td>
<td>1</td>
<td>0=Unoccupied; 1=FD element</td>
</tr>
<tr>
<td>Entity Type (Execution)</td>
<td>2</td>
<td>2</td>
<td>0=Unoccupied; 1=FI element; 2=DS Element</td>
</tr>
<tr>
<td>Hardware ID</td>
<td>16</td>
<td>40</td>
<td>X-Y physical location of element</td>
</tr>
<tr>
<td>Entity ID</td>
<td>16</td>
<td>40</td>
<td>Entity logical ID number</td>
</tr>
<tr>
<td>Element ID</td>
<td>16</td>
<td>40</td>
<td>Entity element logical ID number</td>
</tr>
<tr>
<td>Target Entity ID</td>
<td>16</td>
<td>40</td>
<td>(FT): Entity ID of the data structure that the FI entity is associated with</td>
</tr>
<tr>
<td>Primary Operand/Data Word (Execution Only)</td>
<td>32</td>
<td>64</td>
<td>(FD/FT): Primary operand value; (DS): Data word value</td>
</tr>
<tr>
<td>Secondary Operand (Execution Only)</td>
<td>32</td>
<td>64</td>
<td>(FD/FT): Secondary operand value</td>
</tr>
<tr>
<td>Primary Operand ID</td>
<td>16</td>
<td>16</td>
<td>(FD/FT): Primary operand ID, corresponds to an associated DS entity or element ID</td>
</tr>
<tr>
<td>Secondary Operand ID</td>
<td>16</td>
<td>16</td>
<td>(FD/FT): Secondary operand ID, corresponding to associated DS entity ID or element ID</td>
</tr>
<tr>
<td>Primary Operand Type</td>
<td>4</td>
<td>4</td>
<td>(FD/FT): Primary operand type; for future implementation</td>
</tr>
<tr>
<td>Secondary Operand Type</td>
<td>4</td>
<td>14</td>
<td>(FD/FT): Secondary operand type; for future implementation</td>
</tr>
<tr>
<td>Operation Code</td>
<td>8</td>
<td>8</td>
<td>(FD/FT): Instruction/operation code</td>
</tr>
<tr>
<td>Execution Conditions</td>
<td>8</td>
<td>8</td>
<td>(FD/FT): Status bits ((C, N, V, Z, !C, !N, !V, !Z)) required for execution of instruction</td>
</tr>
<tr>
<td>Status Bits ID</td>
<td>16</td>
<td>40</td>
<td>(FD/FT): Element ID whose execution result produces the status bits for this instruction (compared against execution conditions)</td>
</tr>
<tr>
<td>Execution Order</td>
<td>16</td>
<td>40</td>
<td>(FD/FT): Element execution order number</td>
</tr>
<tr>
<td>Prev.-Execution Order</td>
<td>16</td>
<td>40</td>
<td>(FD/FT): Execution order number of element(s) that must execute before this element</td>
</tr>
<tr>
<td>Result Value (Execution Only)</td>
<td>32</td>
<td>64</td>
<td>(FI): Result value of last execution</td>
</tr>
<tr>
<td>Status Bits (Execution Only)</td>
<td>8</td>
<td>8</td>
<td>(FI): Resulting status bits of last execution</td>
</tr>
<tr>
<td>Execution Count (Execution Only)</td>
<td>16</td>
<td>16</td>
<td>(FI): How many times must this FI receive a result broadcast with a particular execution order number (from a previous instruction) before executing itself?</td>
</tr>
<tr>
<td>Execution Count ID</td>
<td>16</td>
<td>40</td>
<td>(FI): Associated DS Element ID that holds the execution count value</td>
</tr>
<tr>
<td>Message Code (Execution Only)</td>
<td>8</td>
<td>8</td>
<td>(FI): Message code for move, abut, and other special communications operations</td>
</tr>
</tbody>
</table>

**Total to Store:**

- To move \(FD/FT\) element: 338
- To move DS element: 84
- To move \(FD/FT\) element: 698
- To move DS element: 188
codes. The instruction set has been chosen to include all arithmetic and logic operations. Each instruction can be conditional or unconditional.

The instruction set qualifies as a reduced instruction set, keeping with the desired goal to have simple processing elements. This does not limit the overall processor/architecture in terms of complexity; any complex operation can almost always be implemented as a set of smaller operations. The given instruction set supports very common RISC instructions that are capable of compounding to the complex operations that may be required by an application. That is, multipliers, floating point units, and other traditional function units can be implemented as entities containing the appropriate sequence of instructions to execute. Note that these functions will be “soft” in that they will be inherently tailored at compile time to the application at hand, making them further efficient.

A given function may have multiple RETURN instructions that can be reached, just as in most functions in high-level languages. However, in this case, multiple RETURN instructions may be executed in the same path to allow more data to be sent from the called function back to the calling function. Regarding execution counts, the default behavior is an execution count of 1, which will be realized if the FI element’s execution count ID matches its own element ID. That is, as soon as an FI element receives a message that another FI element with an execution order number that matches its previous-execution order number, then it will execute. In the case where there are execution conditions, the FI element will also compare the message’s source element ID to its own status bits ID when considering the previous-execution order number, only executing if the element ID and status bits ID values match. All instructions are therefore capable of conditional execution. However, if a FI element’s status bits ID matches its own element ID, then the execution conditions field is ignored, i.e., the instruction will execute unconditionally.
Table 2.2: Atomic Processor Instruction Set

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>0x00</td>
<td>No operation.</td>
</tr>
<tr>
<td>ADDPS</td>
<td>0x01</td>
<td>Add primary and secondary operands.</td>
</tr>
<tr>
<td>ADDPC</td>
<td>0x02</td>
<td>Add carry and primary operand.</td>
</tr>
<tr>
<td>ADDPSC</td>
<td>0x03</td>
<td>Add primary operand, secondary operand, and carry.</td>
</tr>
<tr>
<td>SUBPS</td>
<td>0x04</td>
<td>Subtract secondary operand from primary operand.</td>
</tr>
<tr>
<td>SUBPC</td>
<td>0x05</td>
<td>Subtract carry from primary operand.</td>
</tr>
<tr>
<td>SUBPSC</td>
<td>0x06</td>
<td>Subtract secondary operand and carry from primary operand.</td>
</tr>
<tr>
<td>INC</td>
<td>0x07</td>
<td>Increment primary operand.</td>
</tr>
<tr>
<td>DEC</td>
<td>0x08</td>
<td>Decrement primary operand.</td>
</tr>
<tr>
<td>INV</td>
<td>0x09</td>
<td>Bitwise inversion of primary operand.</td>
</tr>
<tr>
<td>AND</td>
<td>0x0A</td>
<td>Bitwise AND of primary and secondary operands.</td>
</tr>
<tr>
<td>OR</td>
<td>0x0B</td>
<td>Bitwise OR of primary and secondary operands.</td>
</tr>
<tr>
<td>XOR</td>
<td>0x0C</td>
<td>Bitwise XOR of primary and secondary operands.</td>
</tr>
<tr>
<td>SETC</td>
<td>0x0D</td>
<td>Explicitly set ‘carry’ flag.</td>
</tr>
<tr>
<td>SETZ</td>
<td>0x0E</td>
<td>Explicitly set ‘zero’ flag.</td>
</tr>
<tr>
<td>SETN</td>
<td>0x0F</td>
<td>Explicitly set ‘negative’ flag.</td>
</tr>
<tr>
<td>SETV</td>
<td>0x10</td>
<td>Explicitly set ‘overflow’ flag.</td>
</tr>
<tr>
<td>RSTC</td>
<td>0x11</td>
<td>Explicitly reset ‘carry’ flag.</td>
</tr>
<tr>
<td>RSTZ</td>
<td>0x12</td>
<td>Explicitly reset ‘zero’ flag.</td>
</tr>
<tr>
<td>RSTN</td>
<td>0x13</td>
<td>Explicitly reset ‘negative’ flag.</td>
</tr>
<tr>
<td>RSTV</td>
<td>0x14</td>
<td>Explicitly reset ‘overflow’ flag.</td>
</tr>
<tr>
<td>SHL</td>
<td>0x15</td>
<td>Shift left primary operand, pad with zeroes.</td>
</tr>
<tr>
<td>SHR</td>
<td>0x16</td>
<td>Shift right primary operand, pad with MSB.</td>
</tr>
<tr>
<td>SHLC</td>
<td>0x17</td>
<td>Shift left primary operand through carry, pad with zeroes.</td>
</tr>
<tr>
<td>SHRC</td>
<td>0x18</td>
<td>Shift right primary operand through carry, pad with MSB.</td>
</tr>
<tr>
<td>CALL</td>
<td>0x19</td>
<td>Function call instruction; requests instantiation of function definition logical ID# in primary operand to process data structure logical ID# in secondary operand. Execution of this instruction completes when the instruction receives a RETURN result from the called function instance.</td>
</tr>
<tr>
<td>RETURN</td>
<td>0x1A</td>
<td>Function return instruction; broadcasts return value of function instance to the corresponding CALL instruction in the calling function instance. Primary operand holds the data word to return, and secondary operand holds the logical entity ID of the calling function instance.</td>
</tr>
</tbody>
</table>
2.3 Implementation of Compound Operations

As shown in Section 2.2, each AP supports only a small set of simple integer manipulation instructions, holding true to the philosophy of processing via a large number of simple processing elements. More complex operations such as multiplication and floating point operations are very commonly used in many applications, and a successful high-performance architecture should implement these functions efficiently. These *compound instructions* can be implemented as function definitions that are called when required by other function instances currently executing. Concrete examples are shown in Chapter 5.
Chapter 3

Communications

The communication architecture driving these processing APs uses nearest-neighbor communication methodology; each AP can communicate directly with its eight adjacent neighbors. In the case of communication with non-adjacent processors, neighboring APs are capable of acting as conduits for passing messages to other APs in the sea. These communications may take place as broadcasts in all directions to either the entire sea or just a specific function definition, instance, or data structure (an entity), or a directional broadcast to a specific AP inside or outside the entity.

3.1 Handshaking

The APs in the sea communicate via a set of eight pairs of handshake buffers (two for each neighbor; one to send messages and one to receive messages) for handshaking, as well as a common message buffer for passing messages and data once communication initialization has been established via the handshaking protocol. Each AP, as well as its eight neighbors, can write to and read from these buffers. The buffers are shown visually in Figure 3.1. In addition, a dual message buffer (one to store a message to be sent, and one to load a received message).

The handshaking algorithm is a three-step process using the handshake buffer. A handshake buffer set to 0 by its owner indicates that the latter is ready to have its handshake
buffer filled in the next cycle. If and only if the handshake buffer is currently set to 0, AP \( x \) initiates a request to AP \( y \) by filling AP \( y \)’s corresponding handshake buffer with a specific handshake code. The code specifies the particular type of communication that will take place. This completes cycle 1 of the handshake. Next, AP \( y \) indicates its availability to receive a message in the message buffer by resetting the handshake buffer back to 0. This completes cycle 2. Finally, AP \( x \) loads the contents of its message buffer (the message/data it wishes to send) into AP \( y \)’s message buffer. This completes cycle 3, and the communication cycle is complete. Naturally, there will be many cases where requests arrive to a particular AP from multiple neighbors at the same time. The AP will process one request at a time; the remaining APs will wait for their request to be served—as described in cycle 2, AP \( y \) indicates its availability by resetting the appropriate handshake buffer. Until this occurs, AP \( x \) will wait. Priority in communication handling is also implemented; priorities are based on the handshake code, and certain codes will be accepted before others.

Figure 3.2 shows a timeline representation of a communication cycle in which on element (occupying AP \( x \)) sends a message to another element (occupying AP \( y \)).
The handshake codes are 4 bits wide, and are shown in Table 3.1.

<table>
<thead>
<tr>
<th>Code</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (0000)</td>
<td>AP is ready</td>
<td>Indicates that the AP is ready to receive the next handshake code</td>
</tr>
<tr>
<td>1 (0001)</td>
<td>Global cast</td>
<td>Indicates that the communication is to be propagated throughout the entire sea</td>
</tr>
<tr>
<td>2 (0010)</td>
<td>Beamed cast</td>
<td>Indicates that the communication is to be propagated directionally to a specific AP (can be either inside or outside the entity)</td>
</tr>
<tr>
<td>3 (0011)</td>
<td>Entity cast</td>
<td>Indicates that the communication is to be propagated inside an entire entity only, ending propagation at the entity border</td>
</tr>
<tr>
<td>4 (0100)</td>
<td>Local cast</td>
<td>Indicates that the communication is to be sent only to the AP’s eight neighbors</td>
</tr>
<tr>
<td>5 (0101)</td>
<td>P2P cast</td>
<td>Indicates that the communication is broadcast to be sent only to a single neighbor</td>
</tr>
<tr>
<td>6 (0110)</td>
<td>FD Move request</td>
<td>Indicates that the communication is a P2P cast request for a FD to move into a neighboring cell</td>
</tr>
<tr>
<td>7 (0110)</td>
<td>FI Move request</td>
<td>Indicates that the communication is a P2P cast request for a FI to move into a neighboring cell</td>
</tr>
<tr>
<td>8 (0110)</td>
<td>DS Move request</td>
<td>Indicates that the communication is a P2P cast request for a DS to move into a neighboring cell</td>
</tr>
<tr>
<td>9 (0111)</td>
<td>Abut request</td>
<td>Indicates that the communication is a P2P cast request by a FI to abut a neighboring DS</td>
</tr>
<tr>
<td>10-14</td>
<td>Reserved</td>
<td>Reserved for future expansion</td>
</tr>
<tr>
<td>15 (1111)</td>
<td>AP is non-functioning</td>
<td>Used to indicate to neighbors that the AP is currently unable to receive communications. If an AP is deemed non-functional, then the AP’s handshake buffers will be all set to 1111, and neighboring APs will not attempt to handshake with this AP.</td>
</tr>
</tbody>
</table>

For global, beamed, and entity casts listed in Table 3.1, Table 3.2 shows the propagation
directions for global, entity, and beamed casts. Figure 3.3 shows visual examples of the casts’ propagations in the sea.

Table 3.2: Transmit directions for global (G), beamed (B), and entity (E) casts

<table>
<thead>
<tr>
<th>Received From</th>
<th>N</th>
<th>NE</th>
<th>E</th>
<th>SE</th>
<th>S</th>
<th>SW</th>
<th>W</th>
<th>NW</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>G,B,E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NE</td>
<td></td>
<td>G,E</td>
<td>G,B,E</td>
<td></td>
<td></td>
<td></td>
<td>G,E</td>
<td></td>
</tr>
<tr>
<td>SE</td>
<td></td>
<td>G,E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>G,B,E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>G.E</td>
<td>G,B.E</td>
<td></td>
<td>G.E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td>G.B,E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NW</td>
<td></td>
<td>G.E</td>
<td>G.B,E</td>
<td></td>
<td></td>
<td></td>
<td>G.E</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3: Propagation of communication casts in the sea.
3.2 Message Passing

The message buffer contains the actual message to be communicated as well as the required decoding information such as source and destination identification information and locations. Table 3.3 and Table 3.4 describe the two different formats of the message that is transferred through the message buffer. The former is for broadcasting entity cast results from instruction executions; the latter is used for other non-result communication messages.

Table 3.3: Message buffer format for result message. All number represent bits.

<table>
<thead>
<tr>
<th>Sea Size</th>
<th>Message Type</th>
<th>Source ID</th>
<th>Result Value</th>
<th>P/S Operand</th>
<th>E. O. Number</th>
<th>Status Bits</th>
<th>Reserved</th>
<th>Total Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 K</td>
<td>4</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>20</td>
<td>128</td>
</tr>
<tr>
<td>1 T</td>
<td>4</td>
<td>40</td>
<td>64</td>
<td>64</td>
<td>40</td>
<td>8</td>
<td>44</td>
<td>256</td>
</tr>
</tbody>
</table>

Table 3.4: Message buffer format for coded message types; codes are described in Table 3.5. All numbers represent bits.

<table>
<thead>
<tr>
<th>Sea Size</th>
<th>Message Type</th>
<th>Source ID</th>
<th>Dest. ID</th>
<th>Inter. ID</th>
<th>ID Type: S/D/I</th>
<th>Reserved</th>
<th>Total Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 K</td>
<td>4</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>2/2/2</td>
<td>70</td>
<td>128</td>
</tr>
<tr>
<td>1 T</td>
<td>4</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>2/2/2</td>
<td>126</td>
<td>256</td>
</tr>
</tbody>
</table>

The shaded fields are fields in which the bit width depends on the sea size. As shown, the result message type has the capability of storing both the execution result value as well as a primary or secondary operand from the broadcasting element; this allows for future extension where instructions may change operand values and need to broadcast them to keep all other copies of that element updated in the entity. In the coded message type, the ID Type field is broken into S, D, and I fields, corresponding to source, destination, and intermediate ID numbers with which the message is associated. The source and destination IDs are the sending and receiving ID numbers, and the intermediate ID number is used for other entities or elements associated with the message. These numbers can be either entity IDs, element IDs, or hardware IDs; interpretation is determined by the message type. Note
that in both message types, there is reserved space for future extension.

Table 3.5 shows all current message codes utilized by the architecture.

3.3 Send and Receive Layers

One may expect the nature of the communication architecture to present an issue with message backup. With many messages passing in a single entity, it is conceivable that collisions in messages could occur, causing stalls and therefore delay in execution, if the message passing is not handled appropriately. To avoid backup, each AP contains two sets of handshake buffers and two sets of message buffers; one set is dedicated to receiving messages and one set is dedicated to sending messages, essentially creating a bi-directional message passing system that limits the possibility of collisions. Since each AP contains a dedicated buffer for receiving and sending, an AP can both receive and send a message at the same time to any neighboring processor. The only time when a message will need to wait for more than one communication cycle is if it arrives at the same time with another message from another AP. If this does occur, the priority handshaking discussed in Section 3.1 is capable of handling simultaneous messages through priority settings. There is a form of communication back-off inherent in this system; in the case of simultaneous message receipt caused by a number of close entity casts, the priority-based handling of the receiving AP will naturally “stagger” the casts after the first collision, helping to prevent further collisions. Also, collisions will only occur at the “wave fronts” of the casts. Once the communication cycle in which collision occurs is completed, there are no further conflicts.

Figure 3.4 shows the message and handshake buffer connections between two neighboring APs, including both send and receive layers.
Table 3.5: Message codes and descriptions for non-result message types

<table>
<thead>
<tr>
<th>Code</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Result</td>
<td>The result entity cast by a function instance element after it has executed its instruction, or by a data structure element following abutment with a function instance.</td>
</tr>
<tr>
<td>0x1</td>
<td>Instantiate Function</td>
<td>Beamed cast by a function instance element executing a CALL instruction.</td>
</tr>
<tr>
<td>0x2</td>
<td>Instantiate Interrupt Function</td>
<td>Similar to 0x01, except the function is an interrupt service routine.</td>
</tr>
<tr>
<td>0x3</td>
<td>Return Value</td>
<td>Beamed cast by a called function instance to the calling function instance, after it has completed execution.</td>
</tr>
<tr>
<td>0x4</td>
<td>Interrupt Return Value</td>
<td>Similar to 0x03, except the returning function is an interrupt service routine.</td>
</tr>
<tr>
<td>0x5</td>
<td>Move to Abut</td>
<td>Code sent to all elements of a function instance that has just been instantiated from its function definition; triggers the elements to move and abut a given data structure.</td>
</tr>
<tr>
<td>0x6</td>
<td>Abutment Completed</td>
<td>Entity cast by a function instance element once it has abutted an element of the associated data structure entity.</td>
</tr>
<tr>
<td>0x7</td>
<td>Terminate Function Instance Entity</td>
<td>Entity cast by the function instance element that executes the RETURN instruction, after it has beamed cast the return value.</td>
</tr>
<tr>
<td>0x8</td>
<td>Request Location of Function Definition</td>
<td>Global cast by a function instance element executing a CALL instruction; it requests the ((x, y)) coordinates, or HardwareID, of the function definition it wants to call.</td>
</tr>
<tr>
<td>0x9</td>
<td>Acknowledge Location of Function Definition</td>
<td>Beamed cast by a function definition in response to 0x8.</td>
</tr>
<tr>
<td>0xA</td>
<td>Request Location of Data Structure</td>
<td>Global cast by a function instance element executing a CALL instruction; requests ((x, y)) coordinates, or HardwareID, of the data structure the called function wants to process.</td>
</tr>
<tr>
<td>0xB</td>
<td>Acknowledge Location of Data Structure</td>
<td>Beamed cast by a data structure element in response to 0xA</td>
</tr>
</tbody>
</table>
Figure 3.4: Send and receive layers of communication buffer circuitry.

Note that according to Figure 3.4, the message buffers (MB) on both layers are connected; in the case where an AP is only propagating a message and not processing it, the propagating message can be loaded directly into the Send layer, saving a clock cycle and keeping propagation of messages as quick as possible.
Chapter 4

Operation

4.1 Entity Movement

This is a special case of the P2P cast, using a command message broadcast. A move request is sent to a specific AP by a neighbor when that neighbor (an entity element) wants to move into that AP’s cell. This would obviously only take place with that AP were empty, i.e., it does not contain a function definition element (in the case of the standby layer) or a function instance or data structure element (in the case of the execution layer). A single move operation is described in the following steps:

1. In cycle 1, AP $x$ (which contains one element of an entity) requests to move into AP $y$ by setting AP $y$’s handshake buffer to code 6.

2. In cycle 2, AP $y$ will reset its handshake buffer back to 0. In addition, AP $y$ will fill AP $x$’s message buffer with a specific message indicating either an affirmative or negative response to the move request.

3. In cycle 3, if the move request was answered affirmatively, then AP $x$ will transfer into AP $y$’s cell. If the response was negative, then AP $x$ will need to use the same communication process on its other neighbors to attempt an alternative path to move.

The algorithmic development of the movement algorithm to be implemented in simulation as well as implementation is currently in progress. The three described steps are
together referred to as a *communication cycle*. The transfer of AP contents from one cell to another is accomplished via either a parallel interface or the message buffer. In the former case, each field as described in Table 2.1 is multiplexed to those of its neighbors, allowing the simultaneous transfer to take place. In the latter case, two cycles would be used for transmitting required fields over the limited message buffer size. In either case, for simplicity, we will refer to a single move operation as one communication cycle.

Naturally, an AP in the sea may encounter other APs from other entities along its path that impede its movement along that usually direct path. In this case, the AP will attempt to move in an alternative direction into an available cell, and then again try to move in the target direction in the next cycle. Consider the example shown in Figure 4.1.

Figure 4.1: Atomic processor moving around an entity; the dashed line indicates the desired path, and the solid lines represent the redirected path the entity takes.

As shown in Figure 4.1, an entity element that must redirect its path around another entity will do the following:

1. Element moves one step in either a *counterclockwise* (CCW) or *clockwise* (CW) direction as a redirected path (for example, a target southeast movement will redirect to an east movement).
2. Element repeats step 1, continuing its attempt to move around the entity in a CCW or CW direction.

3. Element eventually clears the entity and is able to move unobstructed towards its target.

Note that there is some extra delay when an element must move around an entity. Since the initial move request is rejected, two extra machine cycles occur for the element to move into its next cell, making a communication cycle a total of five machine cycles instead of the usual three machine cycles.

4.2 Entity Abutment

This is also a special case of the P2P cast, using a command message broadcast. This communication is made to a neighboring processor when a function instance wants to associate with a data structure. Once the function instance hits a data structure, it initiates an abutment request to obtain the data structure’s logical ID number at which time it can then decide if it is the correct data structure to process (in the case that it is not, the APs abort the request and attempt to move around the data structure via the method described in Section 4.1). Abutment is described in the following steps:

1. In cycle 1, AP \( x \) (which is an element of a FI) requests identification information from the AP \( y \) (which is an element of a DS) in order to determine if it is the correct DS to abut, by setting AP \( y \)’s handshake buffer to code 6.

2. In cycle 2, AP \( y \) will fill AP \( x \)’s message buffer with its identification information, and also sets its handshake buffer back to 0 to indicate the request has been served.

3. In cycle 3, AP \( x \) will respond by setting AP \( y \)’s handshake buffer to 6, and fill its message buffer with its own identification information. Now that both the FI and DS
contain the other’s identification information, an abutment will occur if the identification information is correct, \textit{i.e.}, that particular FI was looking for that particular DS.

Like the movement algorithm described in Section 4.1, simulation- or implementation-level algorithms for abutment are in current development. Once abutment completes, the FI element(s) that abutted send out a coded message indicating abutment has completed, which is entity cast to the entire \textit{superentity}. Note that multiple command messages may be sent. Following receipt of this broadcast, every AP in the data structure (DS) portion of the superentity will broadcast a result broadcast via an entity cast to distribute its current data word value. This allows instructions to be preloaded with the current state of its operands. Previous instructions that modify any of these operands will entity cast the updated data to the appropriate data structure element as well as any instruction in the sea that requires the use of that particular data storage element. This means that following the initial data structure broadcast after abutment, all instructions in the function instance (FI) constantly have updated copies of their data operands, with no need for explicit “memory access” type procedures.

4.3 Instruction Execution

The execution of an instruction in an AP may be compared to the familiar \textit{fetch-execute-writeback} procedure, but note that each of these steps is not exactly as in typical architectures as there is no specific defined memory interface for programs or data. More specifically, the three steps are most often the following:

1. The needed data operands have been loaded into the AP containing an instruction (via an earlier result broadcast), and the AP has just received a result \textit{entity cast} from a preceding instruction indicating an execution order number that matches the AP’s
previous-execution order number. The conclusion of this step means that the instruction has all the information it needs to execute and is capable of executing based on the compile-time discovered parallelism.

2. The instruction is executed locally on the AP with its stored data operands. This may take one or more machine cycles to execute, but as these are simple atomic operations, they will likely not take any longer than a few clock cycles to complete. Note that the AP can continue to propagate entity casts and global casts because the communication circuitry is independent of the execution circuitry. If a broadcast is received during execution that mandates that the current execution halt (most likely via a global cast; consider interrupts, discussed in Section 4.7), this is the only case where execution will be affected by communication activity. The conclusion of this step means that the instruction execution has completed.

3. The AP sends out a result entity cast addressed to the data structure element ID number that should contain the result of the instruction execution. This broadcast will signal the data structure element as well as any instruction that contains this element as a data operand (i.e., the data structure element ID number matches the primary or secondary operand ID numbers). The conclusion of this step means that the result of the instruction execution has been cast to all locations requiring the data.

Figure 4.2 shows a timeline representation of an entire instruction cycle, including execution as well as related communication broadcasts.

All instructions defined in Table 2.2 follow this process, with the exception of the CALL and RETURN instructions, which are described further in Section 4.4. These instructions utilize the data operands to indicate addresses of data structure and calling/called functions that are involved in the particular function call being done.
4.4 Function Calls

A function call involves three entities: the calling function, the called function, and the associated data structure that contains the arguments for the called function. The steps for this process can be broken into the following, saying function $A$ calls function $B$ to process data structure $C$:

1. Function $A$’s CALL instruction (see Table 2.2) will request the $(x, y)$ coordinate location of the target/called function $B$ via a global cast message.

2. Function $B$ responds back to that CALL instruction with the coordinates, via a beamed cast directed at the $(x, y)$ coordinates of the CALL instruction.

3. Function $A$’s CALL instruction sends a command message beamed cast to function $B$ indicating it should instantiate and process data structure $C$, indicating the data structure’s $(x, y)$ coordinates in the sea.

4. Function definition $B$ receives the message and instantiates each of its elements into the execution layer.

5. Function instance $B$ moves to the location of data structure $C$ and initiates an abut request.

6. Data structure $C$ accepts the abut request; each element sends an entity broadcast to
the rest of the superentity (data structure and abutted function instance) indicating their element ID numbers and data word values.

7. Once execution of the function completes (that is, a RETURN instruction is ready to execute), the RETURN instruction sends a beamed message back to the calling function instance indicating the return value of the called function.

8. The RETURN instruction sends out an entity cast message to its entity indicating that the function has completed execution and elements should terminate (disappear from the sea). Meanwhile, function A’s CALL instruction entity casts the return value it received from the previous step to the rest of its entity.

The actual execution of a function instance always begins with the “first” instruction, which is specified by the following two conditions:

1. The previous-execution order number is 0

2. The execution order number is 1

Execution conditions are also ignored for the first-time execution of the first instruction; subsequent executions however will consider the execution conditions as usually expected.

During the entire function call process, the CALL instruction will not move from its current location, as it relies on beamed cast communications from the called function and this is location-sensitive. The called function definition also will not move between steps 1 and 2, as this is also a location-sensitive communication process. If a function is to be instantiated multiple times for parallel execution (on different data structures), then multiple CALL instructions will be used, using the secondary operand to specify the different data structure entity IDs to use.

Note that “functions” in the sea may or may not be equivalent to the functions or methods defined in the high-level source code. This is to be decided by the compiler. For simple programs, it is likely that the compiler will retain the original function structure of the
program. For more complex programs with many large functions, the compiler will break large functions into smaller blocks that share common data or parallelism.

4.5 Input/Output

Input and output points can be defined at the sea’s periphery, using a border AP to load data into the sea, as shown in Figure 4.3.

Any number of the input/output interfaces as presented in Figure 4.3 can be used along the border of the sea, allowing for as many I/O pins on a chip as needed for a target application. Interfaces can be serial or parallel, and are limited only by the bandwidth requirements of the given protocol and technology.

To communicate with outside elements, interfaces on each output port can be used to emulate the appropriate interfacing protocol, e.g., UART and USART.
4.6 Loops

Loops can be generated in two ways, depending on the complexity of the loop. If a loop containing only a few sequential operations is needed, then the final instruction in an iteration of the loop is used to broadcast the appropriate earlier execution order number, allowing previously-executed instructions to execute again. Using the execution conditions to look for a particular condition, the number of executions is defined. This is similar to current handling of loops in sequential computers, and is the most common way of implementing a loop in assembly programming. For more complex loops that contain a larger number of instructions that present possible concurrency, a function definition is created for the loop contents, and the loop can be called multiple times using multiple CALL instructions.

The handling of loops depends heavily on decisions made by the compiler related to discovery of parallelism, discussed more in Section 6.2.3. There are many cases where loop iterations can be parallelized completely, especially in vector and matrix operations; these loops can be built as a function definition and instantiated multiple times at runtime. Loops can also be implemented as a subset of instructions within a larger function that repeat execution; this occurs by using the execution order number to trigger earlier instructions to execute again if a given condition for running subsequent loop iterations is true. Of course, instruction-level parallelism within a single loop iteration is always possible via the use of the execution order number (i.e., instructions that can execute concurrently have the same execution order number), as it is for any instruction in a loop iteration or not. Figure 4.4 shows three examples of the way loops may run on the architecture.
4.7 Exception Handling

Exception can be handled at the input/output ports through the use of the global cast, utilizing the command message broadcast format, similarly to the way a function call is performed. The exception will propagate through the entire sea, eventually reaching all APs. Since global casts are of highest priority, the propagation will not be halted by any other communications within entities, allowing the exception to be processed as soon as it reaches the exception handling function (routine). Below is a non-exhaustive list of exceptions, as described in [1], that will have to be considered:

- Hardware device interrupts
- Breakpoints/debug interrupts
- Arithmetic overflow
- Memory faults—e.g., a requested FD or DS is not in the sea
- Undefined/unimplemented instruction
- Hardware malfunctions
- Power failure (requires halting of each AP operation once the exception is received)

Only the power failure will affect all morphological entities and all atomic processors in the sea. All other exceptions will affect the entities to which they relate. Exceptions will not stop other communications currently propagating in the sea once the “wave front” of the exception has passed (as the exception will have higher priority than other communications, those other communications will have to wait for a communication cycle to allow the interrupt to pass through). That is, while an exception is propagating in the sea, all other currently propagating communications—result entity cast messages, mostly—will complete their traveling to their destination, and will update their data structure elements and data operand values. This is to maintain consistency in data throughout the sea, as the same data element is referenced locally in more than one AP.
Chapter 5

Performance Evaluation

5.1 Machine Cycle

A machine cycle is assumed to be the standard local instruction cycle on an AP. This includes just the time for execution of the instruction as well as decoding of the message buffer (analogous to instruction fetch) and encoding of the result message (analogous to writeback)—in other words, a full execution cycle. In addition, a machine cycle is assumed to encompass the amount of time for handshaking and message buffer transfer (also known as a full communication cycle).

5.2 Minimum and Maximum Execution Times

The total execution time of a called function, from the moment the calling function instance sends out the CALL message, and until it receives the return message and/or data structure, depends on the duration of the following events: call to instantiate, instantiate, move to abut, abut, effective execution, and RETURN. In turn, the duration of all these events depends on the sizes of the called function instance and its associated data structure, and except for the effective execution time, on the relative location of the calling function instance and the called function definition, and the relative location of the called function instance and its associated data structure. Unless the relative locations are enforced and
therefore known, the times to call, to instantiate, to move, to abut and finally to return, are a function of the instantaneous computational context. Therefore, their exact values are best evaluated using a cycle accurate functional simulator, which is discussed in Chapter 7. However, the effective execution time can be analytically calculated for a few particular cases. To demonstrate these effective times, some example patterns of execution are shown in Figure 5.1.
We consider an entity with a square shape and $n$ atomic processors on a side. If the code is 100% parallelizable, then all elements execute concurrently, except for the one that executes the RETURN instruction. If the latter is in the center of the square entity, the
minimum effective execution time is $\frac{n-1}{2} + 1$ cycles, where the 1 accounts for the execution cycle of the return, and $\frac{n-1}{2}$ for the necessary communication cycles between the elements at the periphery and the center of the entity, as shown in Figure 5.1(a). Alternatively, if the element that executes the RETURN element is located at the periphery, the maximum effective execution time is equal to $(n - 1) + 1$, where the 1 accounts for the execution cycle of the return, and $n - 1$ for the necessary communication cycles between the RETURN element and the element diagonally opposite from it, as shown in Figure 5.1(b). The total number of instructions being $n^2$, the minimum CPI or minimum cycles per instruction is equal to $\frac{n+1}{2n^2}$, or approximately $\frac{1}{2n}$. Similarly, the maximum CPI or maximum cycles per instruction is equal to $\frac{1}{n}$. An alternative metric is the IPC, or instructions per cycle, which is the inverse of the CPI and thus equal to $2n$ and $n$, respectively.

If the code is 100% sequential or non-parallelizable all elements execute in sequence, i.e. no two elements execute concurrently. Then, the minimum effective execution time is equal to $2n^2$, as shown in Figure 5.1(c). This is achieved when the elements executing in sequence are always adjacent, i.e. each instruction cycle is equal to the shortest instruction cycle. The minimum CPI, or minimum cycles per instruction is 2, and the maximum IPC, or the number of instructions per cycle is $\frac{1}{2}$.

For the 100% sequential or non-parallelizable code, the worst-case condition is when each two elements that execute in sequence are located farthest apart, as shown in Figure 5.1(d) for an $8 \times 8$ sized entity. For convenience, let $n$ be even and $m = \frac{n}{2}$. The number of elements that are located $k$ atomic processors from the center is equal to $4(2k - 1)$, and their instruction cycle is $2k$ cycles long, including the execution cycle. Thus the total execution time of all this elements is $4(2k - 1)(2k)$. Notice that moving diagonally, horizontally or vertically takes the same number of cycles. Then, for an entity of size $n^2$ or
The performance metrics discussed so far are summarized in Table 5.1. The very good results for the 100% parallelizable code meet expectations, because of the massively parallel character of the architecture. The results for the 100% sequential code with optimal placement are good. Considered in isolation, the results for the 100% sequential code with worst placement are at best satisfactorily. However, all these results capture only the exploitation of instruction level parallelism, and must therefore be considered in the larger context of the architecture's overall design and implementation.

The CPI is indicated in (5.2).

\[
CPI = \frac{T_{Eff\text{Exec}}}{4m^2} = \frac{4m + 1}{3} - \frac{1}{3m} 
\approx \frac{4m + 1}{3} = \frac{2}{3}n + 1 \quad (5.2)
\]
context of the sea of atomic processors. Intrinsically, the architecture allows thread or functional level parallelism to be fully exploited. This will compensate and further enhance the overall performance of the architecture. Thread and functional level parallelism can only be considered in a specific computational context, and are therefore best evaluated using a cycle accurate functional simulator, which is discussed in Chapter 7.

Table 5.1: Summary of performance metrics, measured in cycles

<table>
<thead>
<tr>
<th>Performance Metric</th>
<th>100% Parallelizable Code</th>
<th>100% Sequential Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum Effective Execution Time</td>
<td>( \frac{n+1}{n} )</td>
<td>( \frac{2n^2}{3n^3 + n^2} )</td>
</tr>
<tr>
<td>Maximum Effective Execution Time</td>
<td>( \frac{n+1}{n} )</td>
<td>( \frac{2}{3n + 1} )</td>
</tr>
<tr>
<td>Minimum CPI</td>
<td>( \frac{n+1}{n^2} )</td>
<td>( \frac{1}{n} )</td>
</tr>
<tr>
<td>Maximum CPI</td>
<td>( \frac{n+1}{n} )</td>
<td>( \frac{2}{n} )</td>
</tr>
</tbody>
</table>

5.3 Results

5.3.1 Integer Multiplication

To showcase the implementation of a compound operation, we consider multiplication. APs do not have dedicated multipliers; multiplication can be done instead by using a left-shift algorithm. For example, multiplying two numbers \( x \) and \( y \), yielding result \( z \):

1. Initialize a counter to the data word width, or 32 bits for this case.
2. Decrement the counter by 1. If counter value is negative, multiplication is complete. Else, continue to step 3.
3. If LSB of \( x \) is 1, then add \( y \) to \( z \) (else, do nothing).
4. Shift \( x \) right by 1 bit, and shift \( y \) left by 1 bit. Repeat steps 2 through 4.

Figure 5.2 shows an example integer multiplication superentity incorporating the algorithm described above. The following abbreviations may be used for the element fields:

- ID: Element ID
• POp: Primary operand
• SOp: Secondary operand
• Cond: Execution conditions
• SBID: Status bits ID
• Count: Execution count
• EO: Execution order number
• PEO: Previous-execution order number
To demonstrate cycle-by-cycle operation, each cycle is enumerated below by execution order number, indicating the behavior of APs in each cycle.

1. Step 1: In cycle 1, execution starts with element ID 8, which bitwise ANDs $x$ with the value 1 as a mask. If the least significant bit of the result is zero, then the Z flag is
set to 1. Else, it is reset to 0.

2. Step 2: In cycle 2, if the Z flag is reset to 0, then $y$ is added to $z$. At the same time, $x$ is shifted right and $y$ is shifted left. Else if the Z flag is set to 1, then do nothing. In either case, the counter is decremented.

3. In cycle 3, if the counter is not zero and not negative, then execution continues again with step 1. Else, if the counter is zero or negative, then the RETURN element executes and the multiplication is complete.

The multiplication algorithm takes the same number of cycles to execute, regardless of the operand values. The algorithm is effectively two execution steps (enumerated steps 1 and 2 as indicated in the algorithm description in Section 5.3.1) repeated for each bit, in addition to the return step (enumerated step 3).

### 5.3.2 Integer Division

To showcase the implementation of another compound operation, we consider integer division. APs do not have dedicated dividers; division is instead done using a shift-and-subtract algorithm. For example, dividing a number $n$ (numerator) by $d$ (denominator), yielding result quotient $q$ and remainder $r$:

1. Initialize $r = n$.

2. Subtract $d$ from $r$.

3. If $r \geq 0$, then add 1 to $q$ and repeat steps 2-3. Else, add $d$ to $r$, and division is complete.

Figure 5.3 shows an example integer division superentity incorporating the algorithm described above.
To demonstrate cycle-by-cycle operation, each cycle is enumerated below by execution order number, indicating the behavior of APs in each cycle.

1. Step 1: In cycle 1, execution starts with element ID 1, which adds 0 to \( n \) and stores in \( r \) (that is, \( r \) is assigned to \( n \)).

2. Step 2: In cycle 2, \( d \) is subtracted from \( r \). If the result is negative, then the N bit is set to 1. Else, the N bit is reset to 0.

3. Step 3: In cycle 3, if the N bit is set to 1, then \( d \) is added back to \( r \). Else, \( q \) is incremented.

4. In cycle 4, if the N bit was set to 1, then the RETURN element executes and the division is complete. Else, continue execution at step 2.
The division algorithm takes a varying number of cycles to execute, as the number of iterations is dependent on the size of the operands. That is, the number of subtractions of the divisor from the dividend is the number of iterations the division algorithm must perform. The more iterations, the longer the execution time.

5.3.3 Floating Point Operations

Floating point arithmetic, multiplication, and other operations are accomplished through the implementation of special function definitions dedicated to unpacking, processing, and repacking these numbers when the program needs it. As long as there are APs available in the sea, as many floating point functions can be instantiated as needed. There are effectively no structural dependencies in providing this functionality, as there are commonly in processors that use a (usually) limited number of dedicated floating point units, whose use must be scheduled.

Consider the floating point multiplication algorithm in Equation 5.3.

\[ s_p \times 2^{e_p} = s_1 \times 2^{e_1} \times s_2 \times 2^{e_2} \]  

(5.3)

The significand is indicated by \( s \) and the exponent by \( e \) for an unpacked floating point number. Figure 5.4 shows an example floating point multiplication superentity, which consists of a multiply operation on the significands (via a function call to the multiplication function shown in Figure 5.2) and an add operation on the exponents.
1. Step 1: In cycle 1, execution begins with element ID 2, where $e_1$ and $e_2$ are added.

2. Step 2: Next, $s_1$ and $s_2$ are multiplied by a call to an integer multiplication function. Floating point multiplication is now complete.

Note that given the latency required for the CALL operation to implement multiplication, the multiplication function can simply be placed in the function along with the additional instructions and data required for floating point. The only trade-off is that the superentity is larger (corresponding to higher communication latency, but this is likely much lower than the latency associated with a function call).

Consider for completeness the floating point addition algorithm in Equation 5.4.

$$s \times 2^e = \left( s_1 + \frac{s_2}{s_1 - e_2} \right) \times b^{e_1}$$  \hspace{1cm} \text{(5.4)}
This assumes $e_1 > e_2$. To compute the significand and exponent of a floating point addition, a division is required of $s_2$ which can be done via a progressive right shift of $e_1 - e_2$ times. Following this shift, the result is added to $s_1$ and this is the sum’s significand. The sums exponent is simply $e_1$, so no operation on the exponent is necessary.
Chapter 6

Programming and Compilation

6.1 Principles

A compiler for this architecture incorporates discovery of parallelism at the high-level (C programs, for example), followed by assembly into a data stream that will then be downloaded to the device, loading the APs with instructions and data. Each instruction and data element is assigned unique identification information which is used to implement the parallelism that is discovered in the compilation process—that is, which specific instructions can execute in parallel (and which cannot), and which higher-level logical functions can execute in parallel (and which cannot). What follows the integration of this parallelism is a program flow unimpeded by resource dependencies or memory access bottlenecks. If a function has the parameters it needs to execute, it will execute. If an instruction within a function has the operands it needs to execute, it will execute.

There are a number of steps that must be taken to first discover parallelism in conventional code, and then to determine an appropriate level of implementation of that parallelism in a given architecture. This architecture is designed to inherently support high-level programming concepts (function calls, structured data, iterative processes, etc.) and to resolve parallelism first at this level as well as the instruction level.
6.2 Code Conversion Steps

Traditional compilation involves the use of several intermediate representations of the code before translation to machine code. Usually these representations resolve memory access routines explicitly, which makes sense in traditional Von Neumann architectures where memory read/write operations are explicit instructions. In our architecture, memory operations are implicit operations, taking place during execution, but not defined at the instruction level. As such, a new representation must be developed that does not explicitly define memory operations and focuses on the high-level programming concepts that the architecture supports.

The C programming language can be used as a basis for these developments. Compilation can be broken into the following steps:

1. Parse the code into a clean high-level format.
2. Identify data structures and their elements, i.e., constants and variables; identify function blocks and instructions within each function, including function calls.
3. Discover instruction-level parallelism within functions, including function calls.
4. Generate programming information for each AP based on preceding steps.

6.2.1 Parsing and CIL Representation

The first step in compilation is to parse the input code. C Intermediate Language (CIL) [79] is an intermediate representation of C that parses and rewrites the code using simple control structures (only if/else and while control structures) while preserving function definitions, providing explicit variable scope by putting functions further into code blocks, and also extracting library functions from headers that may be present. Call graphs for functions may also be generated to indicate dependencies among functions. This cleaner simpler representation of C code can be used to more easily recognize in a given program
which groups of variables should be built into data structures, and where to split up code into functions.

Giving CIL the source code of a C program we want to convert, the following command is run:

cilly --save-temps --dcallgraph --domakeCFG --dooneRet --noPrintLn --noInsertImplicitCasts --printCilAsIs --noWrap source.c > source.c.cg.txt.

where source.c is the source C file. Consider the following simple vector addition code:

```c
int main() {
    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;
}
```

CIL will parse this code into the following:

```c
/* Generated by CIL v. 1.3.6 */
/* print_CIL_Input is true */

int main(void) {
    int vector1[8] ;
    int vector2[8] ;
    int i ;
    int __retres4 ;

```
vector1[0] = 1;
vector1[1] = 2;
vector1[2] = 3;
vector1[3] = 4;
vector1[4] = 5;
vector1[5] = 6;
vector1[6] = 7;
vector1[7] = 8;
vector2[0] = 5;
vector2[1] = 9;
vector2[2] = 1;
vector2[3] = 45;
vector2[4] = 13;
vector2[5] = 52;
vector2[6] = 9;
vector2[7] = 23;
i = 0;
{
while (1) {
    while_0_continue: /* CIL Label */ ;
    if (i < 8) {

    } else {
        goto while_0_break;
    }
    vector1[i] += vector2[i];
i += 1;
}
while_0_break: /* CIL Label */ ;
}
__retres4 = 0;
return (__retres4);
}

As shown, CIL converts all control structures in the original code to a predictable
while loop structure with continue and break labels. In addition, the function return value is given an explicit variable, compared to the direct constant assignment in the original code. Both of these results provide a secure basis for generating data structures and instruction flow for building function definitions.

Consider another example, where we compute the first 12 elements of the Fibonacci series:

```c
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;
}
```

CIL parses the code into the following:

```c
/* Generated by CIL v. 1.3.7 */
/* print_CIL_Input is true */

int fibonacci(void) {
    int i;
    int a;
    int b;
    int c[10];
    int __retres5;
```
{  
c[0] = 0;  
c[1] = 0;  
c[2] = 0;  
c[3] = 0;  
c[4] = 0;  
c[5] = 0;  
c[6] = 0;  
c[7] = 0;  
c[8] = 0;  
c[9] = 0;  
a = 0;  
b = 1;  
i = 0;  
while (i < 10) {  
    c[i] = a + b;  
    a = b;  
    b = c[i];  
    i ++;  
}  
__retres5 = 0;  
return (__retres5);  
}  

6.2.2 Semantics Processing

Once the code has been parsed into CIL format, it can be further processed to identify functions and data structures.

Function definitions indicated in the original code are first given an identification number. Each variable within each function is also identified, forming a data structure associated with the function. In addition, multi-instruction loops within functions may be identified as functions themselves later on in the case that the iterations can be parallelized.
(more in Section 6.2.3). In intermediate representation of this step consists of a list of individual functions and their code, each function supported by a table of variables, logical identification numbers, and initialization values. The table of variables is a representation of the data structure that is associated with its corresponding function. Functions and data structures themselves are also logically identified, allowing cross-referencing for function calls, as well as allowing for multiple functions to operate on the same data structure (consider global variables here). The preprocessed code from the matrix addition CIL-parsed code observed in Section 6.2.1 is shown below, followed by Table 6.1 which shows the table describing the data structure.

```c
/* Generated by CIL v. 1.3.6 */
/* print_CIL_Input is true */

int main(void)
{
    int $L0[8] ;
    int $L8[8] ;
    int $L16 ;
    int $L17 ;
    
    {
        $L0 = 1;
        $L1 = 2;
        $L2 = 3;
        $L3 = 4;
        $L4 = 5;
        $L5 = 6;
        $L6 = 7;
        $L7 = 8;
        $L8 = 5;
        $L9 = 9;
        $L10 = 1;
        $L11 = 45;
        $L12 = 13;
```
$L13 = 52;
$L14 = 9;
$L15 = 23;
$L16 = 0;
{
while (1) {
   @1: ;
   if ($L16 < 8) {

   } else {
      goto @1;
   }
   $L0[$L16] += $L8[$L16];
   $L16 += 1;
}
@2: ;
}
$L17 = 0;
return ($L17);
}
Table 6.1: Variables table generated for data structure for vector addition code

<table>
<thead>
<tr>
<th>Scope</th>
<th>Entity ID</th>
<th>Element ID</th>
<th>Variable</th>
<th>Index</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>1</td>
<td>0</td>
<td>vector1</td>
<td>vector1[0]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>1</td>
<td>vector1</td>
<td>vector1[1]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>2</td>
<td>vector1</td>
<td>vector1[2]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>3</td>
<td>vector1</td>
<td>vector1[3]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>4</td>
<td>vector1</td>
<td>vector1[4]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>5</td>
<td>vector1</td>
<td>vector1[5]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>6</td>
<td>vector1</td>
<td>vector1[6]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>7</td>
<td>vector1</td>
<td>vector1[7]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>8</td>
<td>vector2</td>
<td>vector1[0]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>9</td>
<td>vector2</td>
<td>vector2[1]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>10</td>
<td>vector2</td>
<td>vector2[2]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>11</td>
<td>vector2</td>
<td>vector2[3]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>12</td>
<td>vector2</td>
<td>vector2[4]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>13</td>
<td>vector2</td>
<td>vector2[5]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>14</td>
<td>vector2</td>
<td>vector2[6]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>15</td>
<td>vector2</td>
<td>vector2[7]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>16</td>
<td>i</td>
<td>i</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>17</td>
<td>_retres4</td>
<td>_retres4</td>
<td>int</td>
</tr>
</tbody>
</table>

The Scope column describes the scope of the variable in the original code: *loc* for local variable, *par* for passed parameter (not present here), and *glb* for global variables (not shown here; global variables will get a separate table). Var Name and Index describe the original variable name. These columns are equal except for arrays, in which case the former
gets the variable name and the latter gets the index. Note that in referencing the code above, the replacement for an array that is indexed by another variable uses the *Element ID* of the zero-index element of that array, or the equivalent to the pointer that would be used in C to reference the array.

Consider now the Fibonacci example; semantics processing produces the following code:

```c
/* Generated by CIL v. 1.3.7 */
/* print_CIL_Input is true */

int main(void)
{
  int $L0 ;
  int $L1 ;
  int $L2 ;
  int $L3[10] ;
  int $L13 ;

  {
    $L3 = 0;
    $L4 = 0;
    $L5 = 0;
    $L6 = 0;
    $L7 = 0;
    $L8 = 0;
    $L9 = 0;
    $L10 = 0;
    $L11 = 0;
    $L12 = 0;
    $L1 = 0;
    $L2 = 1;
    $L0 = 0;
    while ($L0 < 10) {
      $L3[$L0] = $L1 + $L2;
      $L1 = $L2;
    }
  }
```
The associated data structure table is shown in Table 6.2.

Table 6.2: Variables table generated for data structure for vector addition code

<table>
<thead>
<tr>
<th>Scope</th>
<th>Entity ID</th>
<th>Element ID</th>
<th>Variable</th>
<th>Index</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>1</td>
<td>0</td>
<td>i</td>
<td>i</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>1</td>
<td>a</td>
<td>a</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>2</td>
<td>b</td>
<td>b</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>3</td>
<td>c</td>
<td>c[0]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>4</td>
<td>c</td>
<td>c[1]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>5</td>
<td>c</td>
<td>c[2]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>6</td>
<td>c</td>
<td>c[3]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>7</td>
<td>c</td>
<td>c[4]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>8</td>
<td>c</td>
<td>c[5]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>9</td>
<td>c</td>
<td>c[6]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>10</td>
<td>c</td>
<td>c[7]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>11</td>
<td>c</td>
<td>c[8]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>12</td>
<td>c</td>
<td>c[9]</td>
<td>int</td>
</tr>
<tr>
<td>loc</td>
<td>1</td>
<td>13</td>
<td>_retres5</td>
<td>_retres5</td>
<td>int</td>
</tr>
</tbody>
</table>
6.2.3 Detection of Parallelism

Detection of parallelism in a given program is often difficult, though there are some fairly mechanical structures in typical code where parallelism can be extracted mechanically. For example, consider the code for vector addition. A clear loop exists in which the individual elements of the vector are added. Specifically, there is no iteration dependence; that is, an iteration of the loop does not depend on a previous iteration of that loop. This is detectable by the fact that the variables indexed using the loop iterator $i$ are not indexed by $i - 1$, $i + 1$, etc. Given that this is effectively the only instruction in the loop (the addition operation), there is also no instruction dependence. For this reason, this loop can be fully unrolled. In the context of our architecture, this means that each iteration (which consists of a single instruction in this case) gets its own AP with the same execution order number. When executing, all iterations will therefore occur concurrently. Note that this is a simple loop example; more complex loops require more analysis to determine iteration dependencies. Discovery of parallelism in high-level code loops has been a subject of research in the context of parallel and concurrent architectures [80].

In the Fibonacci example, there is instruction dependence in the three instructions in the loop iteration. In addition, there is dependence at the iteration level because variable values from previous iterations are used in future iterations, so no unrolling is possible. For this reason, the loop is performed sequentially.

Note that discovering parallelism is not a trivial task. The above discussed vector addition code and Fibonacci code are simple examples in which the programming steps can be justified; however, specifically in the parallelism detection step, more complex code is much more difficult to detect, and is the subject of future work. A variety of work has been done currently that investigates the challenges in detecting both instruction- and thread-level parallelism in code [81–84].
6.2.4 Translation to Machine Code

Once function definitions and data structures are identified, as well as the execution order and concurrency, the fields presented in Table 2.1 are being configured. These fields provide all information needed to guide the execution of the program.

The preprocessing step as described in Section 6.2.2 provides the logical identification numbers for elements as well as data operands, operation codes, status bits, and execution repeat values. The parallelism discovery step described in Section 6.2.3 provides the execution order and previous-execution order values.

Consider the vector addition code; Table 6.3 and Table 6.4 show the machine code in tabular format for the function definition and data structure elements, Figure 6.1 shows the machine code applied to atomic processors in a superentity (note that the \( x, y \) coordinates differ than the machine code, as the function may move during the course of architecture runtime). Numbers are shown in decimal format for readability.

<table>
<thead>
<tr>
<th>Y</th>
<th>X</th>
<th>Type</th>
<th>EntID</th>
<th>ElemID</th>
<th>PriID</th>
<th>SecID</th>
<th>OpCode</th>
<th>EO</th>
<th>PEO</th>
<th>Cond</th>
<th>CountID</th>
<th>SBID</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>13</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>11</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>31</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>12</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>31</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>13</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>31</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>14</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>13</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>15</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>18</td>
<td>0</td>
<td>Calling</td>
<td>FI HID</td>
<td>26</td>
<td>3</td>
<td>2</td>
<td>19</td>
<td>18</td>
</tr>
</tbody>
</table>
Table 6.4: Vector addition code machine code for data structure elements

<table>
<thead>
<tr>
<th>Y</th>
<th>X</th>
<th>EntID</th>
<th>ElemID</th>
<th>PriOp</th>
<th>Primary</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>17</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>2</td>
<td>1</td>
<td>11</td>
<td>45</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>2</td>
<td>1</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>11</td>
<td>15</td>
<td>2</td>
<td>1</td>
<td>13</td>
<td>52</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>2</td>
<td>1</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>17</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>23</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>2</td>
<td>1</td>
<td>19</td>
<td>7</td>
</tr>
<tr>
<td>AP (0, 0)</td>
<td>AP (0, 1)</td>
<td>AP (0, 2)</td>
<td>AP (0, 3)</td>
<td>AP (0, 4)</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>ADDPS</td>
<td>ADDPS</td>
<td>ADDPS</td>
<td>ADDPS</td>
<td>RETURN</td>
<td></td>
</tr>
<tr>
<td>ID: 0</td>
<td>ID: 1</td>
<td>ID: 2</td>
<td>ID: 3</td>
<td>ID: 18</td>
<td></td>
</tr>
<tr>
<td>POP: 1 (ID: 0)</td>
<td>POP: 2 (ID: 1)</td>
<td>POP: 3 (ID: 2)</td>
<td>POP: 4 (ID: 3)</td>
<td>POP: 0 (ID: 18)</td>
<td></td>
</tr>
<tr>
<td>SOP: 5 (ID: 8)</td>
<td>SOP: 9 (ID: 9)</td>
<td>SOP: 1 (ID: 10)</td>
<td>SOP: 45 (ID: 11)</td>
<td>SOP: [Calling FI HID]</td>
<td></td>
</tr>
<tr>
<td>EO: 1</td>
<td>EO: 2</td>
<td>EO: 2</td>
<td>EO: 2</td>
<td>FO: 3</td>
<td></td>
</tr>
<tr>
<td>PEO: 0</td>
<td>PEO: 1</td>
<td>PEO: 1</td>
<td>PEO: 1</td>
<td>PEO: 2</td>
<td></td>
</tr>
<tr>
<td>Count: 7 (ID: 19)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AP (1, 0)</th>
<th>AP (1, 1)</th>
<th>AP (1, 2)</th>
<th>AP (1, 3)</th>
<th>AP (1, 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPS</td>
<td>ADDPS</td>
<td>ADDPS</td>
<td>ADDPS</td>
<td>DS Element</td>
</tr>
<tr>
<td>ID: 4</td>
<td>ID: 5</td>
<td>ID: 6</td>
<td>ID: 7</td>
<td>ID: 18</td>
</tr>
<tr>
<td>POP: 5 (ID: 4)</td>
<td>POP: 6 (ID: 5)</td>
<td>POP: 7 (ID: 6)</td>
<td>POP: 8 (ID: 7)</td>
<td>POP: 0 (ID: 18)</td>
</tr>
<tr>
<td>SOP: 13 (ID: 12)</td>
<td>SOP: 52 (ID: 13)</td>
<td>SOP: 9 (ID: 14)</td>
<td>SOP: 23 (ID: 15)</td>
<td>SOP: [Calling FI HID]</td>
</tr>
<tr>
<td>EO: 2</td>
<td>EO: 2</td>
<td>EO: 2</td>
<td>EO: 2</td>
<td>FO: 3</td>
</tr>
<tr>
<td>PEO: 1</td>
<td>PEO: 1</td>
<td>PEO: 1</td>
<td>PEO: 1</td>
<td>PEO: 2</td>
</tr>
<tr>
<td>Word: 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AP (2, 0)</th>
<th>AP (2, 1)</th>
<th>AP (2, 2)</th>
<th>AP (2, 3)</th>
<th>AP (2, 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS Element</td>
<td>DS Element</td>
<td>DS Element</td>
<td>DS Element</td>
<td>DS Element</td>
</tr>
<tr>
<td>ID: 0</td>
<td>ID: 1</td>
<td>ID: 2</td>
<td>ID: 3</td>
<td>ID: 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AP (3, 0)</th>
<th>AP (3, 1)</th>
<th>AP (3, 2)</th>
<th>AP (3, 3)</th>
<th>AP (3, 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS Element</td>
<td>DS Element</td>
<td>DS Element</td>
<td>DS Element</td>
<td>DS Element</td>
</tr>
<tr>
<td>ID: 5</td>
<td>ID: 6</td>
<td>ID: 7</td>
<td>ID: 8</td>
<td>ID: 9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AP (4, 0)</th>
<th>AP (4, 1)</th>
<th>AP (4, 2)</th>
<th>AP (4, 3)</th>
<th>AP (4, 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS Element</td>
<td>DS Element</td>
<td>DS Element</td>
<td>DS Element</td>
<td>DS Element</td>
</tr>
<tr>
<td>ID: 10</td>
<td>ID: 11</td>
<td>ID: 12</td>
<td>ID: 13</td>
<td>ID: 14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AP (5, 3)</th>
<th>AP (5, 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS Element</td>
<td>DS Element</td>
</tr>
<tr>
<td>ID: 19</td>
<td>ID: 15</td>
</tr>
<tr>
<td>Word: 7</td>
<td>Word: 23</td>
</tr>
</tbody>
</table>

Figure 6.1: Vector addition code as superentity, derived from machine code representation.
Note that AP(0,0) exhibits an ADDPS instruction that executes before the remaining ADDPS instructions; this is to provide the single “entry point” for the function as described in Section 4.4.

Consider the Fibonacci series code; Table 6.5 and Table 6.6 show the machine code, and Figure 6.2 shows a corresponding superentity.

<table>
<thead>
<tr>
<th>Y</th>
<th>X</th>
<th>Type</th>
<th>EntID</th>
<th>ElemID</th>
<th>PriID</th>
<th>SecID</th>
<th>OpCode</th>
<th>EO</th>
<th>PEO</th>
<th>Cond</th>
<th>CountID</th>
<th>SBID</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>10</td>
<td>1 1</td>
<td>3 1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>11</td>
<td>1 1</td>
<td>4 2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>30</td>
<td>12</td>
<td>1 1</td>
<td>5 3</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>30</td>
<td>13</td>
<td>1 1</td>
<td>6 4</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>10</td>
<td>1 1</td>
<td>7 5</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>31</td>
<td>11</td>
<td>1 1</td>
<td>8 6</td>
<td>7</td>
<td>1</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>31</td>
<td>12</td>
<td>1 1</td>
<td>9 7</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>31</td>
<td>13</td>
<td>1 1</td>
<td>10 8</td>
<td>9</td>
<td>1</td>
<td>8</td>
<td>7</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
<td>1 1</td>
<td>11 9</td>
<td>10</td>
<td>1</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>32</td>
<td>11</td>
<td>1 1</td>
<td>12 10</td>
<td>11</td>
<td>1</td>
<td>10</td>
<td>9</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>32</td>
<td>12</td>
<td>1 1</td>
<td>13 0</td>
<td>Calling</td>
<td>26</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
</tbody>
</table>
Table 6.6: Fibonacci series machine code for data structure elements

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>EntID</th>
<th>ElemID</th>
<th>PriOp</th>
<th>Primary</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>17</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>2</td>
<td>1</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>2</td>
<td>1</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>2</td>
<td>1</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>AP (0, 0)</td>
<td>AP (0, 1)</td>
<td>AP (0, 2)</td>
<td>AP (0, 3)</td>
<td>AP (0, 4)</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td><strong>ADDPS</strong></td>
<td><strong>ADDPS</strong></td>
<td><strong>ADDPS</strong></td>
<td><strong>ADDPS</strong></td>
<td><strong>ADDPS</strong></td>
<td></td>
</tr>
<tr>
<td>ID: 3</td>
<td>ID: 4</td>
<td>ID: 5</td>
<td>ID: 6</td>
<td>ID: 7</td>
<td></td>
</tr>
<tr>
<td>EO: 1</td>
<td>EO: 2</td>
<td>EO: 3</td>
<td>EO: 4</td>
<td>EO: 5</td>
<td></td>
</tr>
<tr>
<td>PEO: 0</td>
<td>PEO: 1</td>
<td>PEO: 2</td>
<td>PEO: 3</td>
<td>PEO: 4</td>
<td></td>
</tr>
<tr>
<td>AP (1, 0)</td>
<td>AP (1, 1)</td>
<td>AP (1, 2)</td>
<td>AP (1, 3)</td>
<td>AP (1, 4)</td>
<td></td>
</tr>
<tr>
<td><strong>ADDPS</strong></td>
<td><strong>ADDPS</strong></td>
<td><strong>ADDPS</strong></td>
<td><strong>ADDPS</strong></td>
<td><strong>ADDPS</strong></td>
<td></td>
</tr>
<tr>
<td>ID: 12</td>
<td>ID: 11</td>
<td>ID: 10</td>
<td>ID: 9</td>
<td>ID: 8</td>
<td></td>
</tr>
<tr>
<td>EO: 10</td>
<td>EO: 9</td>
<td>EO: 8</td>
<td>EO: 7</td>
<td>EO: 6</td>
<td></td>
</tr>
<tr>
<td>PEO: 9</td>
<td>PEO: 8</td>
<td>PEO: 7</td>
<td>PEO: 6</td>
<td>PEO: 5</td>
<td></td>
</tr>
<tr>
<td>AP (2, 0)</td>
<td>AP (2, 1)</td>
<td>AP (2, 2)</td>
<td>AP (2, 3)</td>
<td>AP (2, 4)</td>
<td></td>
</tr>
<tr>
<td><strong>RETURN</strong></td>
<td><strong>DS Element</strong></td>
<td><strong>DS Element</strong></td>
<td><strong>DS Element</strong></td>
<td><strong>DS Element</strong></td>
<td></td>
</tr>
<tr>
<td>ID: 13</td>
<td>ID: 1</td>
<td>ID: 2</td>
<td>ID: 3</td>
<td>ID: 4</td>
<td></td>
</tr>
<tr>
<td>SOP: [Calling FI HID]</td>
<td>ID: 0</td>
<td>ID: 0</td>
<td>ID: 0</td>
<td>ID: 0</td>
<td></td>
</tr>
<tr>
<td>EO: 11</td>
<td>PEO: 10</td>
<td>PEO: 10</td>
<td>PEO: 10</td>
<td>PEO: 10</td>
<td></td>
</tr>
<tr>
<td>AP (3, 0)</td>
<td>AP (3, 1)</td>
<td>AP (3, 2)</td>
<td>AP (3, 3)</td>
<td>AP (3, 4)</td>
<td></td>
</tr>
<tr>
<td><strong>DS Element</strong></td>
<td><strong>DS Element</strong></td>
<td><strong>DS Element</strong></td>
<td><strong>DS Element</strong></td>
<td><strong>DS Element</strong></td>
<td></td>
</tr>
<tr>
<td>ID: 9</td>
<td>ID: 8</td>
<td>ID: 7</td>
<td>ID: 6</td>
<td>ID: 5</td>
<td></td>
</tr>
<tr>
<td>AP (4, 0)</td>
<td>AP (4, 1)</td>
<td>AP (4, 2)</td>
<td>AP (4, 3)</td>
<td>AP (4, 4)</td>
<td></td>
</tr>
<tr>
<td><strong>DS Element</strong></td>
<td><strong>DS Element</strong></td>
<td><strong>DS Element</strong></td>
<td><strong>DS Element</strong></td>
<td><strong>DS Element</strong></td>
<td></td>
</tr>
<tr>
<td>ID: 10</td>
<td>ID: 11</td>
<td>ID: 12</td>
<td>ID: 13</td>
<td>ID: 10</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.2: Fibonacci series code as superentity, derived from machine code representation.
6.3 Compiler Development

A full-featured compiler for this architecture is currently in progress to implement all steps in Section 6.2, and is currently under development. Steps 1 and 2 (CIL parsing and semantics processing) are currently completed. Relating to detection of parallelism, the possibility exists for user interaction during the compilation process to assist in detection of parallelism if needed. Development of algorithms for parallelism detection as well as scripts to run semantics processing and machine code translation are currently in progress.

6.4 Example Simulations

Consider the vector addition code developed in Section 6.2. The sea in Figure 6.1 can be manually simulated to show which elements are affected for each execution order step in the program. The sea is reproduced in Figure 6.3.
Figure 6.3: Vector addition code superentity.
Observing the propagation of communication broadcasts from the executing function instance elements, a manual cycle-level simulation can determine the total execution time:

1. In step 1, AP(0,0) executes; its entity cast takes a maximum of 3 cycles to reach AP(0,3) and AP(1,3).

2. In step 2, the maximum entity cast distance is 4 cycles from AP(1,0) to AP(0,4). Note, however, that with multiple entity casts there may be a cycle delay as AP(0,4) must process all the messages. For this reason the message propagation cycle count may increase to 5 or more cycles.

3. In step 3, execution has completed, after 4 cycles in step 1 and 6 cycles in step 2 (execution plus message propagation), or 10 cycles total. Following these 10 cycles, the function must send its return value back to the calling function.

Consider now the Fibonacci series code; the sea in Figure 6.2 can also be simulated. The Fibonacci sea is reproduced in Figure 6.4.
Observing the propagation of communication broadcasts from the executing function instance elements, a manual cycle-level simulation can determine the total execution time:
1. In steps 1 through 10, result message propagation only takes 1 cycle to reach the next element that needs the result. For example, AP(0,1) needs the result from AP(0,0) in step 1; they are neighbors so the result is received in 1 cycle.

2. In step 11, execution completes and the function can return to the calling function. In total there are 2 cycles (execution and message propagation) for each of steps 1 through 10. This results in a total function execution time, including the 2 cycles in step 1, of 22 cycles.
Chapter 7

Functional Simulation

7.1 Goals

Functional simulation software would provide a way of testing the architecture’s cycle-by-cycle functionality for a given code set, to solidify those expectations indicated in performance estimation (Chapter 5). Simulation can begin with the machine code taken from the translation process described in Section 6.2.4, which would provide information on the contents of all APs, and populates an example sea with those APs. Execution can then be taken in steps. The user would have the ability to view the visual contents of the sea at each step, including message propagation and movement of APs. In addition, the contents of each AP could be observed at each step. Execution time would be monitored as well, providing a very accurate basis for benchmarking and comparison with current architectures.

7.2 Software Design

Work has commenced to develop a cycle-accurate simulator. Execution takes place at machine cycle level so each operation can be observed, including instruction execution and communication processes. An interface is provided to view the contents of each atomic processor (AP) at each cycle so expected performance can be confirmed. Software was built for Microsoft Windows in C using the Windows C API functions.
7.3 Current Functionality

The simulation software is designed to remain faithful to machine cycle-specific behavior of the architecture when given appropriate machine code to execute.

7.3.1 Opening Machine Code Files

Machine code is represented in a table-style format that describes the initial contents of each function definition (FD) and data structure (DS) element. These initial contents consist of all values described in Table 2.1 for both execution and standby layers data structure and function definition elements including their ID numbers and operand ID numbers, as well as execution order and execution count numbers. In addition, the machine code file indicates the initial physical \((x, y)\) location in the sea for each entity element. The typical Simulator window viewing a loaded processor sea is shown in Figure 7.1.

![Figure 7.1: Software screen capture of Simulator window with loaded processor sea.](image-url)
7.3.2 Atomic Processor Record View

AP contents at any given simulation step (i.e., any given machine cycle) can be observed via a modeless dialog box that can remain open alongside the main program window which contains the entire Pond processor sea, shown in Figure 7.2.

![Software screen capture of AP Record view.](image)

Each field describing the contents of a single AP is available on this view; following each simulation step, or if the user clicks on a different AP, the record view is updated to reflect the most recent activity.

7.3.3 Illustration of a Global Cast

Each time an atomic processor is updated; that is, it receives a new message or produces a current result, it is highlighted. The wave front of a global cast can be observed in Figure 7.3.

As shown, the global cast is represented by highlighting the APs that have just received
7.4 To Be Implemented

While the majority of the communication architecture has been implemented in the functional simulation software, there is more work to be done on the execution side. Currently the following functions are planned for implementation in the future:
- Beamed, entity, local, P2P, and redirected cast types
- Send and receive layers for handshaking and message passing
- Decoding of received messages via message buffer
- Encoding of messages to send via message buffer
- Local execution of instructions by APs
- Simulation of defective APs
- Additional simulation engine control (run to completion, pause, etc.)
- Saving of current simulation state to machine code files
Chapter 8

Features and Benefits

Having covered the architectures organization, communications, operation, and performance evaluation, we continue in this section to highlight and discuss its most relevant features and benefits. Provided there are sufficient atomic processors available, any number of programs can execute in parallel without being impeded by structural dependencies. These, and implicitly structural hazards are practically eliminated. Any number of instructions can execute concurrently, if there are no data and/or control dependencies. Also for example, if there are no data dependencies between the iterations of a loop, several or all iterations may execute concurrently. Current architectures employ loop unrolling, but the number of unrolled iterations is severely limited by the number of available registers and functional units, and the number of memory accesses. In the proposed architecture these structural bottlenecks are eliminated. As pointed out in the introduction, future technologies will continue to offer an increasing number of components on a chip. However, some of these will fail prematurely or during the lifetime of the system. Therefore, for the design of any architecture the implication is that it must be robust, i.e. it must be able to tolerate component failures while continuing to operate nominally.

Fault tolerance is traditionally addressed through modular redundancy, which in this context can be implemented at either the gate or functional block levels. In either case, the redundant components can only be used in place, i.e., within their related unit. Furthermore, components usually fail due to locally induced conditions. This means that if redundant
components or modules are located in close proximity, the probability for all redundant copies to fail increases.

In the proposed architecture, while each atomic processor can be viewed as a redundant copy of another atomic processor, each copy can be used by different morphological entities at the same time, and can serve different data manipulation needs. The failure of one or more atomic processor does not fatally affect the overall storage, communication, and processing capabilities of the system. By another point of view, the morphological entities in the sea of atomic processors act like multicellular organisms that discard their dead cells, but maintain their functionality. Each atomic processor is envisioned (in future work) to have a self-test and diagnose capability. Such a self-test would be repeated periodically, or at the request of one of its neighbors or a system function.

There is no central control in the sea of atomic processors, nor a unit that keeps track of available resources. Program execution adapts to the resources available at the time of execution. These resources are equal to the total number of atomic processors available in the sea at the time of fabrication, less the number of atomic processors that have died since manufacturing, and the number of atomic processors currently used by other morphological entities. Similarly, data processing is not centralized, *i.e.*, there are no data paths and no shared manipulation units. Each data word is referenced by the element identification number within the data structure to which it belongs.

The size of the sea of atomic processors is scalable, and programs are portable. This means that programs developed in a 64K sea of atomic processors, can run in a 1 Tera sea of atomic processors, and vice versa, as long as the number of available atomic processors satisfies their processing needs.
Chapter 9

Related Work

In the Introduction, work has been cited in the areas of multicore architectures and networks on chip, which addresses problems of late and post silicon technologies. In this section, elaboration is presented on a few other architectural paradigms with which the *Pond* architecture shares traits, but from which it also differs.

Patwardhan *et al.* explore in [85], a DNA based nano architecture. It is massively parallel and uses simple complexity processing nodes. However, in our architecture, functions and data are virtually separated from the physical support, whereas the nodes self-organize into functional blocks. In their follow-up work, they focus on computing for nanoscale sensors [86].

The architecture we propose resembles a systolic array with respect to the homogenous and structured array of simple complexity data processing elements, and the use of next neighbor communications. However, unlike a systolic array which is a data flow architecture, ours is a control flow architecture in which the processing that a group of atomic processors can accomplish is completely reconfigurable through software. Systolic arrays have proven that data driven processing can be a viable alternative to instruction driven processing, but only for linear and uniform data sets, as is shown in [87].

Reconfigurable computing is currently trying to resolve these limitations by employing both paradigms [88]. A recent architecture with orthogonally arranged processing elements and next neighbor communications has been proposed and built by Baas *et al.* in [89]. As
the authors report, the implementation has been a great success. However, it uses only 36 nodes and it is unclear how it could scale up to a higher number of cores to serve other applications than the ones tested.

In their search for alternative computing paradigms, computer architects and scientists have turned for inspiration to biological systems. Neural networks are modeled after the neural system, and have proved useful for specific classes of applications, such as control systems. For computationally precise tasks, neural networks do not yet match traditional computer architectures. However, the former exhibit a feature that is desirable in future architectures on a chip: resiliency to defective components. Following the example of neural networks in learning from biological systems, Abelson et al. have focused on self-assembly and self-organization [90]. Due to the large number of atomic processing elements, our architecture can support the development and implementation of neural networks of sizes and complexities unmatched today. The same physical platform, the sea of atomic processors, can support precise computing algorithms and neural network algorithms.

Other notable architectures which have been designed specifically to exploit instruction and thread level parallelism are: the Connection Machine by Daniel Hillis [91], WaveScalar by Swanson et al. [92], the RAW microprocessor by Agarwal et al. [93], Blue Gene by IBM [94], supercomputing vector machines [95], and the INMOS Transputer [96]. All these are massively parallel, but their processing nodes are more complex than our atomic processors, and long interconnects are not eliminated, but rather mitigated. Architectures have also been proposed that incorporate similar morphological movement and interaction as a basis for execution of programs. Some of these include Self-Reproducing Automata [97] and Invasive Algorithms and Architectures [98]; in contrast with these approaches, the Pond architecture exhibits only physical nearest-neighbor communications as opposed to additional routing structures. In addition, the Plastic Cell Architecture [99] shares similarities in terms of communication infrastructure but is targeted specifically for
reconfigurable hardware devices. In the context of discovery of parallelism as well as dataflow architectures, Macromodules [100] and Micropipelines [101] are similar functionally but do not necessarily incorporate a decentralized or distributed memory storage system.
Chapter 10

Future Work and Conclusions

10.1 Detection of Component Defects

An excellent feature of this architecture design is resiliency against component defects. Conceivably, APs can be intelligent enough to detect if there is a defect in their circuitry, and deactivate at runtime to prevent other APs from attempting communication. Defects are certainly possible and very probable at the time of fabrication, but it is also possible that over time, damage can occur to the sea, especially with the increase in component density and smaller feature sizes of state-of-the-art technologies. Ideally, self-checks could be implemented for periodic diagnostics so the sea is aware of corruption as soon as possible. Since the types of checks that can be done are somewhat ambiguous, and there is nothing preventing the self-check mechanisms themselves from becoming damaged, a good implementation of self-checking could occur at startup from a separate set of circuitry outside the sea (but obviously on the same die; this is the same case with the input/output interfacing which resides outside the grid/sea of APs), which would run a series of test vectors along the path for bit stream programming, and would receive a bit stream back indicating which APs failed a test, and the AP would be forced into an inactive state. Of course, all of this circuitry is subject to defects as well, but assuming the defect detection circuitry is a very small percentage of chip area—a safe assumption—it is much more likely that defects would occur outside the detection circuitry itself.
The fact that the processor sea architecture is functionally able to operate with a significant amount of defects makes the usage of test vectors post-production a sensible approach. If a given circuit has the capability of operating with defects because of high redundancy (as is the case here), a method of running diagnostics is a very valuable way of preserving processor functionality by taking advantage of this redundancy. Since it is highly unlikely that enough defects or runtime damage occur in a given chip to render a significant portion of APs to be inoperable, it is very unlikely that fabrication-time defects as well as field-time corruption would ever mandate that a chip be discarded, except in extreme damage cases. As indicated, a small unlikely exception exists if damage occurs to input/output interfaces or the diagnostic/test circuitry.

10.2 Implementation Considerations

An important aspect of this design to consider is the physical implementation. In the preceding sections we have discussed mainly the design and performance considerations. The arrangement of circuitry in terms of small cells with local communications is not a particularly new approach (though the architecture design, behavior, and programming concepts presented for this architecture are unique), but follows a well-established compute model similar to cellular automata where multiple identical elements communicate via a next-neighbor system [102].

Programming of this architecture may be considered in the form of a bit stream download, where the sea is put into a programming mode, and each AP receives the initialization bits it needs to create data structures, function definitions, and execution order information. Logical identification numbers, initialized operands, operand identification numbers, execution order and preceding execution order numbers, instruction execution repeats (allows instruction concurrency), opcodes, and condition flags may be programmed into each AP via SRAM cells similar to the way FPGAs are configured using bit stream downloads.
Consider also the implementation benefits of this architecture design. The scalable nature of the architecture allows for very efficient wafer use—areas along the perimeter of the wafer usually lost because they do not offer enough area to fit another chip can be used to fabricate smaller seas, as shown in Figure 10.1.

![Figure 10.1: Example of multiple sea sizes on the same wafer](image)

As shown, the edges of the wafer that cannot fit another die of a given sea size can be populated with smaller dies with smaller sea sizes, bringing wafer usage closer to the ideal 100%.

In addition, this architecture offers a design that theoretically will not suffer from “hot spots” as in typical architectures. Hot spots often appear in traditional architectures in high-density manipulation units (multipliers, etc.) and areas of centralized control, where the high density, high usage area generates significantly more heat than surrounding areas [103]. Since there is no centralization of control or areas of the sea that are targeted for specific manipulation functions, theoretically hot spots would be eliminated. However, certain “warm” spots will exist likely at the input/output ports for applications that employ
heavy input or output such as streaming media or bulk data transfers.

The lack of a central bus implementation and no long interconnects means that operation likely could run at a significantly higher speed than traditional architectures. A common issue in system performance in terms of clock speed is parasitics in interconnects. The longer a wire, the higher the series resistance and inductance, and the higher the parallel capacitance. The interconnect can become a low-pass filter, limiting the maximum frequency that can reliably propagate. Therefore, the clock speed and high-frequency communication lines are often limited by what can be supported in these long interconnects because of this low-pass behavior. Since this architecture operates using next neighbor connections only, and does not require global clock distribution, interconnect wires would have significantly less parasitics, allowing for reliable operation at much higher clock speeds. Higher clock speeds on the same processor mean faster execution. In addition, the higher parasitic capacitance in interconnects translates effectively to a larger load capacitance on the wire, increasing power consumption because of the increase in the amount of charge that it takes to drive that wire. Avoiding long interconnects avoids this power loss. That power loss can be “reallocated” to allow for higher clock speeds at the same power draw, or simply allow for a lower-powered chip overall.

10.2.1 Size Requirements

This architecture uses a novel approach to computing by taking multicore processing to its most fine-grained level. In many multicore designs, the number of processing units is substantially smaller than the number of words in storage, requiring the majority of processing to be done sequentially; the amount of parallelism is limited by the small number of processing cores. Since processing functions take significantly more hardware to implement than memory (RAM), it is not surprising that multicore designs have taken this approach. We propose a design that does implement the same number of processing elements as data
storage elements—in fact, they are interchangeable—to eliminate these structural hazards that impede concurrent and parallel execution of programs. With feature sizes decreasing dramatically, and the introduction of post-silicon technologies offering more and more components on a chip, many designs attempt to improve parallelism by further increasing the complexity of processing cores to sometimes substantially improve instruction throughput via runtime scheduling systems and more redundancy in functional units. This approach has its limits; by instead decreasing core complexity and placing more of them on a chip, the inherent parallelism on instruction-level and function-level can be exploited likely on a much higher level.

10.3 Software Development

Areas of current and future development of this architecture include refinements to software simulation, and development of a compiler/assembler as well as a fully-featured IDE (Integrated Development Environment) that allows for program flowchart-style graphical programming with parallelism as well as standard high-level language programming and compilation. Of course, this is an optional course of action and is intended mainly for development purposes; this architecture and its associated compiler, once completed, are intended to work on traditional sequential languages and discover parallelism automatically.

10.4 Contributions to the State-of-the-Art

This thesis has intended to contribute the definition, initial development, and analysis of a massively parallel computer architecture intended for implementation consideration in late and post silicon technologies. Among the notable characteristics are the massively parallel and fine-grained nature, a distributed data storage system providing decentralization of
resources, a theoretically successful next-neighbor message passing system, function or entity-based morphological behavior of cores, and a basis for important features moving into late and post silicon design including scalability and defect/fault tolerance.
Bibliography


[57] 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, Sept 2009.


Appendix A

Functional Simulator Programming Guide

The simulation software is written entirely in C using the Windows Win32 C API.

A.1 File Structure

The simulation software consists of five source files, logically splitting the functional components of the program:

- simulator_win.c: Contains the main program window function and message loop.
- wnd_functions.c: Contains other program window functions and their message loops.
- ap_grid_ui.c: Contains functions related to the processing of the atomic processor visual sea.
- ap_comm.c: Contains state machine for handshaking process and other functions specific to communications.
- sim_engine.c: Contains the main simulator engine loop and associated functions.

In addition, 4 header files are present, containing function prototypes, global variables, and OS-specific resource allocations:

- resource.h: Resource file containing macro definitions for menu items, images, and other UI elements
• resource.rc: Resource file containing more information on images, dialogs, and other UI elements

• globals.h: Contains all global variable definitions, type definitions, function prototypes, and common macros.

• codes.h: Contains macros for message codes for command message format.

A.2 Variables and Structs

A.2.1 record

The record struct represents the storage elements of a single atomic processor. It contains all identification fields, operation code, execution conditions, order, count, status bits, and handshake and message buffers. It also contains other AP-specific information such as handshaking priorities and information on “dead” atomic processor neighbors for handling message rerouting.

A.3 Functions

A.3.1 wnd_functions.c

BOOL CALLBACK DlgProcOpenPond(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)

Function corresponding to modeless AP Record dialog window.

Parameters:

• HWND hWnd: HWND Win32 API window handle for main program window.

• UINT msg: Message passing variable for OS message loop.

• WPARAM wParam: Parameter storing data related to OS message passing.
- **LPARAM lParam**: Parameter storing data related to OS message passing.

**Returns**: OS-specific callback.

### A.3.2 ap_grid_ui.c

```c
void PondBuildGrid(FILE *fi_ds_pnd, FILE *fd_pnd)
```

Builds the grid user interface for viewing the Pond processor sea. Machine code files are whitespace-delimited. A line or row contains the record of one AP.

**Parameters:**

- **FILE *fi_ds_pnd**: Pointer to file containing function instance and data structure machine code.
- **FILE *fd_pnd**: Pointer to file containing function definition machine code.

**Returns**: Nothing.

```c
void RecenterGridOffset(HWND hWnd)
```

Assistive function to re-evaluate locations of APs after dragging and zooming.

**Parameters:**

- **HWND hWnd**: HWND Win32 API window handle for main program window.

**Returns**: Nothing.

```c
void PlaceAPBitmaps(HWND hWnd, HDC windc)
```

Places colors corresponding to function definitions, function instances, and data structures in the sea.

**Parameters:**

- **HWND hWnd**: HWND Win32 API window handle for main program window.
• **HDC windc**: HDC Win32 API device context for handling bitmap placement.

**Returns:** Nothing.

### A.3.3 ap_comm.c

*void* **APNeighborComm(int x, int y)**

Step in neighbor handshaking process where message buffers are loaded.

**Parameters:**

- *int x*: x-coordinate of AP.
- *int y*: y-coordinate of AP.

**Returns:** Nothing.

*void* **APNeighborReq(int x, int y)**

Step in neighbor handshaking process in which the handshake buffer is initially set by the sending AP.

**Parameters:**

- *int x*: x-coordinate of AP.
- *int y*: y-coordinate of AP.

**Returns:** Nothing.

*void* **APBufferProcess(int x, int y)**

Step in neighbor handshaking process where receiving AP acknowledges a handshake request.

**Parameters:**
- **int x**: X-coordinate of AP.
- **int y**: Y-coordinate of AP.

**Returns**: Nothing.

---

```c
void APSetTransDest(record *rec, int origin)
```

Resolves cast types into appropriate directions for APs to send message.

**Parameters**:

- **record *rec**: Pointer to AP Record structure of element sending message.
- **int origin**: Compass direction origin of message that will be sent.

**Returns**: Nothing.

---

### A.3.4 sim_engine.c

```c
void SimInit()
```

Initializes the simulation. Called at the start of each simulation. As of 19 May 2010, this function is empty.

**Parameters**: None

**Returns**: Nothing

```c
void SimEnd()
```

Called at the end of each simulation. As of 19 May 2010, this function is empty.

**Parameters**: None

**Returns**: Nothing.

```c
void SimStep(HWND hWnd)
```
Called to run a single simulation (machine) cycle.

**Parameters:**

- **HWND hWnd**: HWND Win32 API window handle for main program window.

**Returns**: Nothing.

```c
void APExecuteMsgInstr(record *rec)
```

Called when an instruction is ready to execute.

**Parameters:**

- **record *rec**: Pointer to AP Record structure containing element that should execute its instruction.

**Returns**: Nothing.

```c
void LoadMessage(record *rec, int msg0, int msg1, int msg2, int msg3, int msg4, int msg5, int msg6, int msg7)
```

Called when an AP needs to send a message through a communications handshake. Each parameter represents one field of the message vector; indices are described in macros, which are included in the globals.h include file.

**Parameters:**

- **record *rec**: Pointer to AP Record structure containing element whose message buffer should be set up.

- **int msgx**: Individual message bitfields to transfer; not all may be used for all messages.

**Returns**: Nothing.

```c
void APDecodeMsgBuffer(record *rec)
```

Called when an AP receives a message from a communications handshake.

**Parameters:**
- *record *rec: Pointer to AP Record structure containing element who just received the message.

**Returns:** Nothing.