A Stack-Based Concatenative Approach to Co-processor Simulation and Code Generation

Conrad Jakob 1
Paul Beckett 2

1 Crunch DSP Pty. Ltd.
151 Elgin St., Carlton, Melbourne
Email: conrad@crunchdsp.com.au

2 Royal Melbourne Institute of Technology,
PO Box 2476, Melbourne, Victoria 3000,
Email: pbeckett@rmit.edu.au

Abstract
As we run to the end of the silicon roadmap, the rapidly escalating cost of design, fabrication and test in future systems-on-chip may justify a re-evaluation of homogeneous reconfigurable mesh architectures. Ultimately, the geometric constraints of extreme nanoscale device layout may support only simple functional arrays with predominately nearest neighbour connectivity. However, these have proved to be difficult to configure and program effectively as even small load imbalances, unnecessary synchronization overheads or delayed accesses to remote data can prevent typical applications from running efficiently on large distributed arrays of processors. We describe an approach to the problem of configuring a system-on-chip comprising an array of small interconnected processors that imposes a common structure on distributed programs and trades some code efficiency for ease of programming and ease of verification. Zeta is a stack-based, concatenative language that bears some similarity to Forth and has been developed as an aid for managing some of the complexities of implementing distributed code on regular array platforms. Zeta programs implement the computation and connections of directed acyclic graphs (DAGs) by manipulating a stack of state variables. State information can be transferred ‘forward’ or ‘backwards’ (in time) across the directed acyclic graph as needed using a virtual bus construct. This paper describes the key features of the language and illustrates its use via an example implementation of an LDPC decoder.

Keywords: concatenative language, stack-based, multi-coprocessor programming

1 Introduction

Because of its implications to petascale and ultimately exascale computing (Kogge et al. 2008), concurrent operation on multiple processing elements is an intensive area of research. The objective is to efficiently and effectively exploit parallelism in the face of languages and operating systems that seem to be actively seeking to hide it. At best this involves significant software modifications targeting a specific multi-processor platform. More often than not it means completely rethinking the design, debug and optimization methodologies. It is already clear that even small load imbalances, unnecessary synchronization overheads or delayed accesses to remote data can prevent typical applications from running efficiently on large distributed arrays of processors.

In this paper, we describe Zeta, a stack-based, “concatenative” language used for simulation and code generation targeting a homogeneous multi-processor array platform. The language bears some similarity to Forth and has been developed as an aid for managing some of the complexities of implementing distributed code on regular array platforms. Zeta programs implement the computation and connections of directed acyclic graphs (DAGs) by manipulating a stack of state variables. State information can be transferred ‘forward’ or ‘backwards’ (in time) across the directed acyclic graph as needed using a virtual bus construct.

This work is part of a wider program established to directly address the issues of performance, power and scalability in homogeneous multi-coprocessor platforms. Multi-processor programming tends to be characterised by excessive degrees of freedom i.e., there are many different ways to achieve a required outcome. Zeta restricts these degrees of freedom by imposing a common structure on programs and trades some code efficiency for ease of programming and ease of verification. The Zeta language has been used for fast prototyping and educational demonstration of these architectures, and allows designs to be implemented and simulated without complex EDA tools such as re-targetable compilers.

The remainder of this paper continues as follows. Section 2 briefly outlines the organization of the homogeneous coprocessor array for which the Zeta language was developed. Section 3 outlines the syntax and key characteristics of this stack-based language. In Section 4 we illustrate the application of the language with a relatively complex example of an LDPC algorithm. Finally, Section 5 summarizes and concludes the paper.

2 The Co-processor Fabric

In this section, we briefly describe our simulation platform that has been set up to explore the issues of power and performance in homogeneous reconfig-
urable arrays. The Homogenous Extensible Large Grained Array (HELGA) is a regular mesh-connected multi-processor architecture aimed at high throughput media-related workloads. The coprocessor fabric has been implemented as a C++ simulation, and some parts of it transformed to a synthesizable Verilog hardware description. Together, the C++ and Verilog simulations form a co-verification fabric in which tightly coupled hardware and software (e.g., instruction set) optimizations can be made.

The focus on high-throughput multimedia applications implies that the operations that need to be supported most efficiently include simple addition, subtraction, accumulate, multiplication and multiply-accumulate, with variably sized operands in the range 8 to 32 bits. Further, the workloads of interest are predominately single-instruction multiple-data (SIMD). As such, HELGA is one in a long line of homogenous reconfigurable arrays (see (Theodoridis et al. 2008) for a survey of coarse grained reconfigurable architectures and (Hartenstein 2001) for a good historical survey). For example, the MorphoSys architecture (Singh et al. 2000) was extensively studied in the late 1990’s, while more recent coarse grained architectures include QUKU (Shukla et al. 2007), RICA (Khwam et al. 2008), MORA (Lamuzza et al. 2007), ADRES (Vander Aa et al. 2011), EGRA (Ansaloni et al. 2011), BiIRC (Atak & Atalar 2013) and the Real-time baseband processor of (Zhang 2014), all of which use various architectural techniques to speed dataflow performance and maintain streaming throughput.

**Figure 1: Basic Array Architecture**

The coarse-grained array architecture we are using as an example platform here comprises a uniform array of hierarchically connected concurrent processing elements (CPEs) (Figure 1) in which groups of 16 processing elements are organized into four groups (‘clusters’) of four CPEs (‘quads’) with a small amount of local shared data memory. Each PE can be considered to be equivalent to a simple 8-bit logic unit and is connected horizontally, vertically and diagonally to its nearest neighbours via shared busses. In turn, each quad is interconnected with its three neighbors via 32-bit data paths. As a result, arithmetical operations can be performed on a pair of 8-bit operands, or the units can be ganged together into 16/32 and 64-bit within a cluster.

In this way, HELGA draws on work such as the MorphoSys architecture (Singh et al. 2000), the primary differences here being the inclusion of local diagonal interconnect and the use of a small conventional instruction RAM (e.g., 128 or 256 entries) in place of their row/column context memory. Instruction mem-

ory fetches operate synchronously across all CPEs in a cluster i.e., during each cycle, every processor executes the instruction at the same program counter.

While these types of array architectures support fast, high-efficiency computing, transforming useful algorithms into usable program code is non-trivial and requires the developer to simultaneously map the time and place of all state information. This becomes prohibitive for anything more than the simplest of algorithms. Typically, development tools do not exist for such novel architectures, and so developers must either hand-code their efforts, or “roll-their-own” development solutions.

Hand-coding is the simplest approach, but leads to highly brittle code. Because state information is valid only at specified times and places, any changes to a hand-coded program will usually invalidate code at other locations. This effect is also problematic for architectures under active design, where assumptions about instruction and data timing are often completely invalidated by design changes. To counter this, there is a need for “thin” development tools that can easily adapt to any architectural changes yet allow programs to be specified in a time/place independent manner. The following section will develop this theme.

### 3 Zeta

As mentioned above, Zeta is a stack-based, concatenative language that has been developed to simplify the process of simulation and code generation on the HELGA platform. It is able to directly describe the computation and connectivity of directed acyclic graphs (DAGs) by manipulating a stack of state variables. Programs targeting the hierarchical platform can be connected using the concept of a “virtual bus” that passes state information ‘forward’ through the directed acyclic graph as required.

Coprocessor code generation and mapping is handled automatically so that programs become easier to write and verify. Like most stack languages, the syntax is terse and low-level. The stack represents a primary data store, with instructions popping the top elements as inputs and pushing their results back to the stack. Functions are composed merely by concatenating functions together, so the parsing of concatenative languages becomes trivial.

The task of programming large arrays of processors is typically characterised by an excessively broad design space, meaning there are always many different ways to achieve a desired outcome. Zeta restricts these degrees of freedom somewhat by imposing a common structure on coprocessor programs, effectively trading efficiency for ease of programming and verification. For this reason, a Zeta program imposes a programming model that includes:

- a stack of eight 32-bit variables, addressable in 8-, 16- or 32-bit slices
- stack persistence across a sequence of adjacent CPE groups, allowing arbitrary-length computation not restricted to a single group
- parallel threads with inter-thread communication operations
- easy I/O operations through ports positioned at the boundary of each group of CPEs
- memory accessible across multiple clusters
• simulation with a self-contained virtual machine, allowing verification of the results of coprocessor programs against simulated results

The price paid for this convenience is a loss of control over the selection and mapping of HELGA code. There are few code-generation decisions to be made when using Zeta, because code generation follows a simple strategy of aggregating code in the order in which instructions are encountered. In this sense, the language superimposes “traditional” computing resources (such as buses, threading and I/O) onto a non-traditional computing fabric.

Zeta programs are simulated using a simple virtual machine that allows the flow of HELGA code to be verified against a simulated version, so any errors are highlighted in the simulator output, greatly increasing confidence in the generated code. However, one side-effect of this generation technique is that, because code is generated based on the location of stack elements stored across CPEs, the stack cannot “evolve” in a data-dependent way. This means that, once coded, the HELGA machine organization is static for that implementation i.e., there are no Zeta instructions that will modify the number of elements used in the stack in response to any data.

3.1 Syntax

In addition to a number of conventional “meta instructions”, which provide high-level control over the code (such as comments, compiler definitions, conditional compile-time switches, etc.), a complete Zeta program includes:

• a main namespace
• an execute function
• one or more Zeta instructions

Namespaces (e.g., main::) are used to organize code into libraries while Functions (e.g., execute::) contain all of the instructions encountered up to the next namespace or function statement. Concatenation is achieved by ‘calling’ functions using a namespace::function call, e.g. main.execute. All instructions in the called function are then copied into the definition of the current function. This can be illustrated using the simple example program shown in Figure 2 which consumes three constants a, b and c and places a+b and c+b on top of the stack. The main.execute function loads the three immediate values and calls a library function to perform element-wise addition. String substitution is performed using the define instruction to define a key and value, and then using $key$ to perform the substitution.

Everything between a hash ‘#’ and the end of a line are ignored. In a similar manner to Forth, these comments are often used for documenting the stack evolution by using simple stack diagrams. For example, the comment

`# ( a b c -- (a+b) (c+b) )

documents a function that takes three operands a, b and c (where a is on the top of the stack), and returns two results (a+b) and (c+b) with the first element on the top of the stack. The “#” character is used to start the comment.

3.2 Threads

Zeta programs are written for a single thread and simple parallelism is achieved by requiring that multiple threads use a common set of instructions (‘single instruction, multiple data’, or SIMD). Threads are implemented by considering the coprocessor grid as one or a series of CPE sequences, defined as arrangements of CPEs where program state is communicated between neighbours.

Processing sequences are derived by partitioning the grid into linear chains of CPE groups that can communicate through their shared buses. Two specific arrangements are supported for the current coprocessor: ‘quad’ ordering of PEs as a repeating grid of 2x2 tiles, and ‘hex’ as a grid of 4x4 tiles. The cluster arrangements are sliced into sequences as shown in an example in Figure 3.

While all threads execute concurrently, groups within a thread’s CPE sequence also execute concurrently. PE groups later in the sequence will execute before the data wave front of causality traverses along each sequence. This can lead to problems when synchronizing HELGA state evolution with simulated results.

Threads in a multi-threaded design can communicate directly with their upper and lower neighbours through the N and S buses. Zeta commands: parallel, ->>>, ++>>-, ->>> and ->>> are used for defining threads and communicating stack information between threads.

3.3 State

Zeta provides a persistent program state consisting of eight 32-bit variables accessed as a stack. Each variable is accessible as 8-, 16-, or 32-bit slices, and slices may be accessed and used in parallel in 1x32-bit, 2x16-bit and 4x8-bit combinations.
3.4 Memory

Memory use in multi-processor programs is particularly complex because RAM is associated with a particular group of CPEs. However memory read and write operations could possibly be separated by several groups of CPEs in a thread sequence. Several memory structures have been developed to automate the use of RAM. Two of these structures are suitable for read/write operations within a single CPE group, and two are suitable where read operations occur before write operations and these occur across different groups. For these last two, RAM is allocated at the read operation and a ‘reverse bus’, so-called because it operates in the reverse direction from the flow of computation (Figure 5), is used to communicate information back from the write operation to the cluster containing the read operation.

Four memory types are defined by Zeta: Read-only (ROM), single cluster scratchpad RAM (SPM), multiple cluster RAM (RAM) and multiple cluster FIFO (SAM, sequential-access memory). SPM - scratchpad memory is a random-access memory type that allows reading/writing to a block of memory within a single cluster using an address. It is used mostly for simple variables that are read, updated and written locally (loop indices, for example). ROM - read-only memory can also be read with an address, but cannot be written and is typically used for lookup tables. SAM - sequential access memory is a FIFO that maintains an internal ‘current’ index to read a value and then overwrite that value with a new value from the reverse bus.

Distributed memory types, i.e., that can be read and written across groups of CPEs need data values (and addresses for RAM) to be communicated between the read source and write destinations. The reverse bus is used to move four 32-bit variables from later sequences to earlier sequences via a multiple-sequence zig-zag pattern (length 4 is shown in Figure 5 but this is arbitrary). Three of the four sequences merely pass data from an input port to an output port, and every fourth sequence writes the data to the relevant group of CPEs and two are suitable where read operations occur before write operations and these occur across different groups.

Memory use in multi-processor programs is particularly complex because RAM is associated with a particular group of CPEs. However memory read and write operations could possibly be separated by several groups of CPEs in a thread sequence. Several memory structures have been developed to automate the use of RAM. Two of these structures are suitable for read/write operations within a single CPE group, and two are suitable where read operations occur before write operations and these occur across different groups. For these last two, RAM is allocated at the read operation and a ‘reverse bus’, so-called because it operates in the reverse direction from the flow of computation (Figure 5), is used to communicate information back from the write operation to the cluster containing the read operation.

Four memory types are defined by Zeta: Read-only (ROM), single cluster scratchpad RAM (SPM), multiple cluster RAM (RAM) and multiple cluster FIFO (SAM, sequential-access memory). SPM - scratchpad memory is a random-access memory type that allows reading/writing to a block of memory within a single cluster using an address. It is used mostly for simple variables that are read, updated and written locally (loop indices, for example). ROM - read-only memory can also be read with an address, but cannot be written and is typically used for lookup tables. SAM - sequential access memory is a FIFO that maintains an internal ‘current’ index to read a value and then overwrite that value with a new value from the reverse bus.

Distributed memory types, i.e., that can be read and written across groups of CPEs need data values (and addresses for RAM) to be communicated between the read source and write destinations. The reverse bus is used to move four 32-bit variables from later sequences to earlier sequences via a multiple-sequence zig-zag pattern (length 4 is shown in Figure 5 but this is arbitrary). Three of the four sequences merely pass data from an input port to an output port, and every fourth sequence writes the data to the relevant group of CPEs and two are suitable where read operations occur before write operations and these occur across different groups.

Memory use in multi-processor programs is particularly complex because RAM is associated with a particular group of CPEs. However memory read and write operations could possibly be separated by several groups of CPEs in a thread sequence. Several memory structures have been developed to automate the use of RAM. Two of these structures are suitable for read/write operations within a single CPE group, and two are suitable where read operations occur before write operations and these occur across different groups. For these last two, RAM is allocated at the read operation and a ‘reverse bus’, so-called because it operates in the reverse direction from the flow of computation (Figure 5), is used to communicate information back from the write operation to the cluster containing the read operation.

Four memory types are defined by Zeta: Read-only (ROM), single cluster scratchpad RAM (SPM), multiple cluster RAM (RAM) and multiple cluster FIFO (SAM, sequential-access memory). SPM - scratchpad memory is a random-access memory type that allows reading/writing to a block of memory within a single cluster using an address. It is used mostly for simple variables that are read, updated and written locally (loop indices, for example). ROM - read-only memory can also be read with an address, but cannot be written and is typically used for lookup tables. SAM - sequential access memory is a FIFO that maintains an internal ‘current’ index to read a value and then overwrite that value with a new value from the reverse bus.

Distributed memory types, i.e., that can be read and written across groups of CPEs need data values (and addresses for RAM) to be communicated between the read source and write destinations. The reverse bus is used to move four 32-bit variables from later sequences to earlier sequences via a multiple-sequence zig-zag pattern (length 4 is shown in Figure 5 but this is arbitrary). Three of the four sequences merely pass data from an input port to an output port, and every fourth sequence writes the data to the relevant group of CPEs and two are suitable where read operations occur before write operations and these occur across different groups.

Memory use in multi-processor programs is particularly complex because RAM is associated with a particular group of CPEs. However memory read and write operations could possibly be separated by several groups of CPEs in a thread sequence. Several memory structures have been developed to automate the use of RAM. Two of these structures are suitable for read/write operations within a single CPE group, and two are suitable where read operations occur before write operations and these occur across different groups. For these last two, RAM is allocated at the read operation and a ‘reverse bus’, so-called because it operates in the reverse direction from the flow of computation (Figure 5), is used to communicate information back from the write operation to the cluster containing the read operation.

Four memory types are defined by Zeta: Read-only (ROM), single cluster scratchpad RAM (SPM), multiple cluster RAM (RAM) and multiple cluster FIFO (SAM, sequential-access memory). SPM - scratchpad memory is a random-access memory type that allows reading/writing to a block of memory within a single cluster using an address. It is used mostly for simple variables that are read, updated and written locally (loop indices, for example). ROM - read-only memory can also be read with an address, but cannot be written and is typically used for lookup tables. SAM - sequential access memory is a FIFO that maintains an internal ‘current’ index to read a value and then overwrite that value with a new value from the reverse bus.

Distributed memory types, i.e., that can be read and written across groups of CPEs need data values (and addresses for RAM) to be communicated between the read source and write destinations. The reverse bus is used to move four 32-bit variables from later sequences to earlier sequences via a multiple-sequence zig-zag pattern (length 4 is shown in Figure 5 but this is arbitrary). Three of the four sequences merely pass data from an input port to an output port, and every fourth sequence writes the data to the relevant group of CPEs and two are suitable where read operations occur before write operations and these occur across different groups.

Memory use in multi-processor programs is particularly complex because RAM is associated with a particular group of CPEs. However memory read and write operations could possibly be separated by several groups of CPEs in a thread sequence. Several memory structures have been developed to automate the use of RAM. Two of these structures are suitable for read/write operations within a single CPE group, and two are suitable where read operations occur before write operations and these occur across different groups. For these last two, RAM is allocated at the read operation and a ‘reverse bus’, so-called because it operates in the reverse direction from the flow of computation (Figure 5), is used to communicate information back from the write operation to the cluster containing the read operation.

Four memory types are defined by Zeta: Read-only (ROM), single cluster scratchpad RAM (SPM), multiple cluster RAM (RAM) and multiple cluster FIFO (SAM, sequential-access memory). SPM - scratchpad memory is a random-access memory type that allows reading/writing to a block of memory within a single cluster using an address. It is used mostly for simple variables that are read, updated and written locally (loop indices, for example). ROM - read-only memory can also be read with an address, but cannot be written and is typically used for lookup tables. SAM - sequential access memory is a FIFO that maintains an internal ‘current’ index to read a value and then overwrite that value with a new value from the reverse bus.

Distributed memory types, i.e., that can be read and written across groups of CPEs need data values (and addresses for RAM) to be communicated between the read source and write destinations. The reverse bus is used to move four 32-bit variables from later sequences to earlier sequences via a multiple-sequence zig-zag pattern (length 4 is shown in Figure 5 but this is arbitrary). Three of the four sequences merely pass data from an input port to an output port, and every fourth sequence writes the data to the relevant group of CPEs and two are suitable where read operations occur before write operations and these occur across different groups.

Memory use in multi-processor programs is particularly complex because RAM is associated with a particular group of CPEs. However memory read and write operations could possibly be separated by several groups of CPEs in a thread sequence. Several memory structures have been developed to automate the use of RAM. Two of these structures are suitable for read/write operations within a single CPE group, and two are suitable where read operations occur before write operations and these occur across different groups. For these last two, RAM is allocated at the read operation and a ‘reverse bus’, so-called because it operates in the reverse direction from the flow of computation (Figure 5), is used to communicate information back from the write operation to the cluster containing the read operation.

Four memory types are defined by Zeta: Read-only (ROM), single cluster scratchpad RAM (SPM), multiple cluster RAM (RAM) and multiple cluster FIFO (SAM, sequential-access memory). SPM - scratchpad memory is a random-access memory type that allows reading/writing to a block of memory within a single cluster using an address. It is used mostly for simple variables that are read, updated and written locally (loop indices, for example). ROM - read-only memory can also be read with an address, but cannot be written and is typically used for lookup tables. SAM - sequential access memory is a FIFO that maintains an internal ‘current’ index to read a value and then overwrite that value with a new value from the reverse bus.

Distributed memory types, i.e., that can be read and written across groups of CPEs need data values (and addresses for RAM) to be communicated between the read source and write destinations. The reverse bus is used to move four 32-bit variables from later sequences to earlier sequences via a multiple-sequence zig-zag pattern (length 4 is shown in Figure 5 but this is arbitrary). Three of the four sequences merely pass data from an input port to an output port, and every fourth sequence writes the data to the relevant group of CPEs and two are suitable where read operations occur before write operations and these occur across different groups.
Table 1: Zeta Instruction Types

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Stack before</th>
<th>Stack after</th>
</tr>
</thead>
<tbody>
<tr>
<td>drop</td>
<td>drop the top of stack</td>
<td>a</td>
<td>-</td>
</tr>
<tr>
<td>dup</td>
<td>duplicate the top of stack</td>
<td>a</td>
<td>a a</td>
</tr>
<tr>
<td>pop</td>
<td>pop from top of stack, push onto stack</td>
<td>a</td>
<td>-</td>
</tr>
<tr>
<td>m&gt;</td>
<td>pop from stack n, push onto top of primary stack for n=1..9</td>
<td>-</td>
<td>a</td>
</tr>
<tr>
<td>over</td>
<td>copy the top of stack</td>
<td>a b</td>
<td>a b a</td>
</tr>
<tr>
<td>rot,rot3</td>
<td>rotate the stack</td>
<td>a b c d</td>
<td>b c a</td>
</tr>
<tr>
<td>-rot, -rot3</td>
<td>reverse rotate the stack</td>
<td>a b c d</td>
<td>c a b</td>
</tr>
<tr>
<td>rot8</td>
<td>rotate the stack (4 elements)</td>
<td>a b c d</td>
<td>b c d a</td>
</tr>
<tr>
<td>-rot4</td>
<td>reverse rotate the stack (4 elements)</td>
<td>a b c d</td>
<td>d a b c</td>
</tr>
<tr>
<td>swap</td>
<td>swap the top two elements</td>
<td>a b</td>
<td>b a</td>
</tr>
<tr>
<td>reset</td>
<td>reset the stack</td>
<td>a b c ...</td>
<td>-</td>
</tr>
<tr>
<td>&lt;,const&gt;</td>
<td>load immediate</td>
<td>-</td>
<td>&lt;const&gt;</td>
</tr>
<tr>
<td>random</td>
<td>load random immediate</td>
<td>-</td>
<td>&lt;const&gt;</td>
</tr>
</tbody>
</table>

The coprocessor code can therefore be generated in the order of a concatenated Zeta program. This is achieved by ‘locating’ each input stack element and possibly moving these input values to an appropriate quad, performing the calculation, and moving the results back to the appropriate stack element location. In HELGA, two registers (r1 & r0) contain the state information for the algorithm code. Therefore all generated code must avoid overwriting these registers unless calculation results are being written, forcing the compiler to select either the accumulator or other registers to maintain program state.

Communication between parallel threads is only possible because Zeta threads are single-instruction parallel. Zeta programs can be considered as dataflow programs with a static schedule of execution as determined by the execution order of Zeta instructions. The schedule is developed implicitly by the developer by arranging the calculations to effectively use the stack as each instruction ‘consumes’ data from previous instructions.

The coprocessor code can therefore be generated in the order of a concatenated Zeta program. This is achieved by ‘locating’ each input stack element and possibly moving these input values to an appropriate quad, performing the calculation, and moving the results back to the appropriate stack element location. In HELGA, two registers (r1 & r0) contain the state information for the algorithm code. Therefore all generated code must avoid overwriting these registers unless calculation results are being written, forcing the compiler to select either the accumulator or other registers to maintain program state.

The coprocessor code can therefore be generated in the order of a concatenated Zeta program. This is achieved by ‘locating’ each input stack element and possibly moving these input values to an appropriate quad, performing the calculation, and moving the results back to the appropriate stack element location. In HELGA, two registers (r1 & r0) contain the state information for the algorithm code. Therefore all generated code must avoid overwriting these registers unless calculation results are being written, forcing the compiler to select either the accumulator or other registers to maintain program state.

Code is mapped to the array in the order in which it is encountered and is mapped as ‘early’ as possible.
to maximise efficiency. In the case of data dependencies, code generation will first ‘synchronize’ to the earliest possible location such that all input operands are available. (It could be said that the code generated by Zeta resembles an upside-down Tetris game, where irregularly-shaped sections of code are placed as high (early) as possible in the instruction memory while satisfying any data dependencies.)

A simple example of Zeta code is shown in Figure 9, where three 32-bit immediate values are first stored on the stack using the r0 or r1 registers of quads Q1, Q2 and Q3. The first addition is performed by moving from the Q1 stack element to the accumulator of Q0, while also moving the Q2 stack element to the Q2 accumulator, and then adding with the ALU. The second addition is performed in a similar manner, before the final results is read from an ADD register and stored in a Q3 stack element.

Figure 9: Simple code generation example

Many factors affect the efficiency of the code generation. Some of these have been identified empirically:

- the mapping of stack elements to quads/registers should cause code to ‘cover’ all quads as uniformly as possible
- the slicing of operations into heterogeneous 8- and 16-bit operations should be used sparingly
- the stack should be kept as full as possible to allow data to be ‘preloaded’ into the corresponding stack variables
- generated code blocks should be small and ‘atomic’, and higher-level code generation should be composed of many smaller units

3.7 Execution

Executing a Zeta program will cause coprocessor code to be generated, and a set of test data to be created. However, there are some subtleties associated with synchronised execution of a Zeta program on a virtual machine and as coprocessor code. These have no bearing on the performance of the code, and are merely artifacts of the interplay between this dataflow approach and the more traditional thread-based processing of the stack-based virtual machine. These potential issues are identified below.

3.7.1 Pipelining

Coprocessor programs execute continuously and so each PE in the array repeats the same instructions in a loop. When the execution of all PE groups is considered together, the coprocessor program comprises one stage in an infinite set of pipelines. Figure 10 shows an example of this. A set of eight clusters (0,1,2,3,16,17,18,19, all E-W neighbours) is arranged in a sequence. Pipeline 0 begins execution with cluster 0 in user tick 0, and execution continues in cluster 1 in user tick 1, cluster 2 in user tick 2, and so on. In parallel, pipeline 1 begins execution with cluster 1 in user tick 1, and continues similarly.

Figure 10: Pipelining of cluster sequences

Because RAM is persistent, pipelines can communicate with each other by storing/loading state between stages, allowing the execution of directed cyclic graphs (DCGs). This is illustrated in Figure 10 with the W0 and W1 operations. Operation W0 (memory write origin) first occurs in the 0th pipeline in sequence 3 (cluster 3), and the address/data is communicated to sequence 1 (cluster 1) where operation W1 (memory write destination) occurs. The latency between write operations is therefore dependent on the sequence distance between W0 and W1 operations. Because read operations are performed immediately before the W1 operation in the code generation, the total latency between read to write operations is approximately double the sequence distance.

3.7.2 Simulation

Zeta programs are self-contained and are simulated using a simple virtual machine based on a set of eight 4x8-bit stacks. The evolution of these stacks is logged to stdout.

The results of the simulation are then used to verify the generated code. The Zeta compiler will automatically generate a test vector file as an input for the simulator. Multiple pipelines are simulated by rerunning the simulation from the start, but without changing any RAM settings. The RAM banks can then be used to create feedback for DCGs by creating threads where initial values are loaded from RAM, processed, and the written back to RAM using an appropriate memory type.

3.7.3 Synchronization

The pipelining illustrated in Figure 10 makes obvious the problem of synchronizing the coprocessor code.
with the simulation. The shaded positions in the pipeline map represent clusters that do not participate in the 0th pipeline until a number of user clock ticks have passed. In general, the delay before a cluster participates in the 0th pipeline is equal to its position in the sequence.

This presents no problems if the coprocessor code is implementing a DAG, because any unwanted results due to processing before the 0th pipeline can be safely ignored. The same is not true, however, for DCGs where an unwanted change in state can perpetuate and cause inaccurate results.

The solution to this is to ‘guard’ the RAM write operations to prevent RAM state from being changed until the 0th pipeline. When accessing RAM with an address, this is achieved using a simple reference counting scheme. For sequential-access memory, it is achieved by adding a number of ‘dummy’ locations to the start of the FIFO and then skipping these when the FIFO pointer is reset. In this way, simulation results can be used to verify the code generation by ensuring that virtual machine results match the results from simulation.

3.7.4 Verification

Results from the virtual machine can be used to verify code generation by writing intermediate VM outputs to the vector file as expected simulation outputs. The simulator will then automatically compare these against the simulation outputs and flag any errors.

Thus program correctness automatically splits into two sub-problems: algorithmic correctness, which can be verified by comparing VM outputs against an external reference model and code correctness, which can be verified automatically by the simulator.

In the next section, we look at a typical example to illustrate the application of the Zeta language to coprocessor design and implementation.

4 A Case Study: LDPC Decoder Design and Implementation

Low-density parity check codes (LDPC) are a class of error checking and correcting codes originally proposed by Gallager (Gallager 1963) that have been adopted for 10 Gb/S Ethernet (IEEE 803.3an), WiFi (IEEE 802.11n) and WiMAX (IEEE 802.16e) standards, digital storage, and deep-space communications. LDPC codes are characterised by large, sparse parity-check matrices (denoted by \( H \)) that can be used to generate syndromes. A syndrome \( s \) is a check vector calculated from a received codeword \( c \):

\[
Hc^T = s.
\]

Codewords are encoded from a sequence of message bits by adding redundancy in the form of parity bits, and are created in such a way that they are distinct, allowing higher decoding accuracy if they are corrupted in transmission. A null syndrome \( s = 0 \) indicates that the received codeword is a valid output codeword of the encoder, and, because the Hamming distance between codewords is generally maximal, it is assumed that the codeword has been received with no bit errors.

All operations are assumed to be on \( GF(q) \), the Galois (finite) field of order \( q \), which constitutes a set of \( q \) symbols and addition and multiplication operators that are commutative, jointly distributive, and have defined identities and inverses. Field \( GF(2) \) is commonly used and has obvious advantages for two-value logic circuits, however there is a large body of work that studies LDPC codes with higher radices \( q > 2 \). All operations described in this report will be on \( GF(2) \), comprising the symbols 0 and 1 and modulo-two addition and multiplication operators. Accordingly, the modulus operator ‘mod 2’ is often omitted from arithmetic equations.

The role of an LDPC encoder is to augment the bits of a message with a set of parity bits, thus increasing the redundancy of the bit stream. The encoder can be implemented as a matrix multiplication, derived from the parity check matrix \( H \), or using linear algebra to derive a set of encoding equations from \( H \) that require fewer calculations. Similarly, the role of an LDPC decoder is to interpret a set of bit likelihoods, generally as the result of a preceding demodulation step, to find the most likely message that could have been sent.

To demonstrate the application of the Zeta language, an LDPC decoder was built and simulated. MATLAB and C versions of the decoder were adapted to generate input and verification data for this implementation. The design focused on one of the emerging international standards, IEEE 802.11n. As this standard contains several different LDPC codes, one of the smallest (Table R.1(d)) was selected, as summarised in Table 2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>648</td>
<td>Number of bits per decode</td>
</tr>
<tr>
<td>R</td>
<td>5/6</td>
<td>Code rate</td>
</tr>
<tr>
<td>L</td>
<td>27</td>
<td>Number of bits per LDPC column</td>
</tr>
<tr>
<td>J</td>
<td>4</td>
<td>Number of LDPC rows</td>
</tr>
<tr>
<td>I</td>
<td>24</td>
<td>Number of LDPC columns</td>
</tr>
</tbody>
</table>

Table 2: LDPC design goal (from Table R.1(d) of (IEEE WG802.11 Standards Group 2009))
Figure 12: LDPC data flow (one of 24 threads)

- bit node update: updates the bit node estimates
- statistics (not shown): calculates statistics for the bit node values across all threads
  - \(\min_0/\min_1\): the minimum and second minimum of the bit node magnitudes \(|M_i|\)
  - \(S\): the sign-product of the bit node values \(M_i\)
- check node update: updates the check node bits and bit likelihoods
- feedback: reads/writes the check node and bit likelihood values using RAM

There is a natural parallelism in the algorithm based on the QC column, and this is used to develop the code in a “quasi-SIMD” form, where all threads use (almost) common code to operate on the multiple data across the threads. Thus the decoder requires 24 threads. The design is based on a layered approach in which the LDPC matrix is considered as a set of layers, and each layer in turn is used for updating a subset of the estimates (Hocevar 2004)(Sharon et al. 2004). As a result, memory usage is optimized to:

- \(J\): 1-byte,
- \(z\): 1-byte,
- \(C\): 24 x 4 bytes,
- \(L\): 24 x 27 bytes,
- \(E\): 24 x 4 x 27 bytes,

- total: 3338 bytes/decoder

which is distributed across the 24 threads.

All calculations use 8-bit saturated arithmetic. This was previously verified using the C model, and allows a simple optimization to be used where the decoder is written as 4 parallel decoders by using the ALUs in 8-bit non-carry modes. Thus the LDPC code is common across the four decoders, however they can operate on separate data.

The Zeta stack-based language was used to develop the coprocessor code because it automatically creates the necessary infrastructure (buses, memory types, continuous calculation), removes the burden of code generation, and allows simulation of the decoder within the language’s virtual machine and generation of verification data. Even so, the two most difficult modules to implement were the minima calculations and the sign-product calculation, as these both involve a high degree of interconnection between threads.

All inter-thread communication is through the nearest-neighbour by using the appropriate parallel communication function within Zeta. For example, the minima calculation has been implemented as a 24-way sort, followed by the selection and distribution of the two lowest values, as shown in Figure 13. Here, the arrows represent a two-element sorting operation. In the first stage, threads \(2i\) and \(2i+1\) are compared and swapped if needed. In the second stages \(2i+1\) and \(2i+2\) are similarly sorted, and the stages repeat until, in the worst case, a smallest value in thread 23 has the opportunity to traverse across to the 0-th thread. Once the thread’s values have been sorted, the minimum and second minimum is merely distributed across the clusters.

Figure 13: Minima calculation using the “sort and distribute” method

The sign-product is more complicated than the minima calculation owing to the nature of the sign-product operation itself. The sorting operation described above is idempotent, and subsequent sorting operations after the first do not affect the result. The two-element sign-product operation is not idempotent, and so care must be taken to ensure the correct results is calculated.

For the sign-product calculation, a shuffle network is used similar to the minima calculation, except after four iterations three partial sign products \(S_{0..7}, S_{8..15}, \) and \(S_{16..23}\) are available equi-spaced across the threads. These are combined to form a single sign product for all threads, and the result is distributed.
4.2 Results

The current version of the example coprocessor implementation exhibits the following performance:

- Throughput \( \approx 26.0 \text{Mb/s} \)
  - Message length = 2160 bits
  - 4 independent decoders
  - 540 bits/message
  - Decode time \( \approx 83000 \text{clocks/decode} \)
  - SNR = 1dB
  - Number of iterations = 2

Each module was profiled by determining the number of clock ticks required for that module. The results are summarized in Figure 15. Clearly, this implementation requires further profiling and optimization and clearly demonstrates the mutual relationship between the compilation process and the underlying hardware structure. About 5000 of the 6100 clocks required for a single iteration of the LDPC decoder are used by the minima and sign product modules in the statistics module. This module performs a large number of “shuffles” between adjacent groups of CPEs, and the minima calculation generates in the order of 80% of these.

5 Conclusions

In this paper, we have described the main characteristics of Zeta, a stack-based, concatenative language that has been developed to manage the complexities of simulation and code generation in a multi-coprocessor hardware environment that comprises an array of small interconnected processors. The language imposes a common structure on distributed programs and trades off some code efficiency for ease of programming and ease of verification. Zeta programs implement the computation and connections of directed acyclic graphs (DAGs) by manipulating a stack of state variables.

An example implementation of an LDPC decoder has illustrated the effectiveness of the approach. It was found that more than 80% of the clock cycles required for a single iteration of the LDPC decoder are used by two of the statistics modules and these modules generate the majority of the bus traffic between them. It is exactly this sort of analysis that Zeta was originally set up to support. From these results we can clearly see that the behavior of the interprocessor communication channels has the greatest effect on the overall performance of the machine. Using Zeta to generate stimulus vectors for the eventual hardware platform will support the future optimization of these channels for not only LDPC but many other algorithms.

References


Figure 14: Sign-product implementation using the partial + combine + distribute approach

Figure 15: Profiling results


