A Hardware Generator for Factor Graph Applications

James Daniel Demma

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Engineering

May 6, 2014
Blacksburg, VA

Keywords: Factor Graph, Probabilistic Graphical Model,
LDPC, Sudoku, Belief Propagation

A Hardware Generator for Factor Graph Applications

James Daniel Demma

(ABSTRACT)

A Factor Graph (FG -- http://en.wikipedia.org/wiki/Factor_graph) is a structure used to find solutions to problems that can be represented as a Probabilistic Graphical Model (PGM). They consist of interconnected variable nodes and factor nodes, which iteratively compute and pass messages to each other. FG's can be applied to solve decoding of forward error correcting codes, Markov chains and Markov Random Fields, Kalman Filtering, Fourier Transforms, and even some games such as Sudoku. In this paper, a framework is presented for rapid prototyping of hardware implementations of FG-based applications. The FG developer specifies aspects of the application, such as graphical structure, factor computation, and message passing algorithm, and the framework returns a design. A system of Python scripts and Verilog Hardware Description Language templates together are used to generate the HDL source code for the application. The generated designs are vendor/platform agnostic, but currently target the Xilinx Virtex-6-based ML605. The framework has so far been primarily applied to construct Low Density Parity Check (LDPC) decoders. The characteristics of a large basket of generated LDPC decoders, including contemporary 802.11n decoders, have been examined as a verification of the system and as a demonstration of its capabilities. As a further demonstration, the framework has been applied to construct a Sudoku solver.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Probabilistic Graphical Models (PGM) can be used to describe a broad scope of problems. They may be applied to speech recognition, computer vision, information coding, machine learning, and protein modeling, to name a few. At their most basic, PGMs are sets of random variables with conditional interdependencies. They are graphical, in that they can be modeled as connected graphs, with the random variables as vertices and the conditional dependencies as edges. A Factor Graph (FG) is a variation of a PGM that executes an algorithm to find a solution to a set of constraints, given some initial a priori information. In terms of the PGM examples listed, a factor graph might be used to map recorded voice audio to a known words, or to discern objects from image data, or to fix errors in an encoded set of bits.

Factor graphs work in the domain of probability. A FG solver may be thought of as a probability processor. Elements within a FG communicate with one another by passing messages about what they believe they are, what they believe their neighbors to be, and how strongly they believe it. They don't work deductively, in serial fashion. They process iteratively and in parallel, converging their beliefs on a solution. Some problems that are difficult to solve deterministically, such as those described above, can be solved probabilistically, using factor graphs.

1.1 Motivation and Contribution

This work presents a system for generating factor graph solvers. It was inspired by the current state of the art of hardware FG applications, and attempts to further it with two of its components. The first is abstract; it is a generalized model for hardware FG solvers. Analysis of hardware FG applications led to the realization that their structures and hierarchies could be represented by a single model. This model is the foundation upon which the FG-Solver modules of this work are constructed. The second component is more concrete. It is the Generator Framework itself, a software system for automatically generating hardware FG solver designs. Its intended benefits are ease of creation and rapid prototyping. The system has been applied to real problems, and the generated designs have withstood thorough testing and verification.

1.2 Document Structure

This document begins with an explanation of FGs and their structure and operation. Following this, the factor graph solver generator system is explained. The individual component generators will be discussed in detail. Generated component micro- and macro-level architectures will be presented, as well those of the supporting subsystems. Three FG applications will be presented. Each application discussion begins with the motivation for its selection. It is followed by an analysis of the results of testing the application, and a conclusion regarding how the application fared. This document concludes with an analysis of the capabilities and limitations of the presented system, and the implications of certain modifications and additions to it. Lastly, value of such a system with regard to the present state of factor graph applications is theorized upon.

Chapter 2
Factor Graphs Explained

As the name implies, a FG's structure is that of a graph, with vertices and edges. It is usually *bipartite*, meaning it's vertices can be partitioned into two sets such that a vertex of one set does not share an edge with another vertex in the same set. These two sets of vertices are commonly called *variable nodes* and *factor nodes*. Edges in the graph define connections between variable and factor nodes. These connections represent channels through which the nodes communicate with one another via messages. Through iterations of this message passing, a solution is converged upon.

The process begins by presenting the variable nodes with *a priori* data, which are an initial set of probabilities of values for the variable nodes. Then variable nodes compute messages to pass to factor nodes based on the messages they receive from those same nodes. Once they've received their messages, factor nodes compute theirs as well and pass them on. Variable nodes go again, and then factor nodes again, and so on. One round of message passing by both variable and factor nodes comprises an *iteration*. This continues until an iteration limit is reached, or a condition is met that represents a solution. In addition to calculating their messages, variable nodes also calculate *a* *posteriori* data, which are a post-computation set of probabilities of values for the variable nodes. These are more commonly referred to as *beliefs*. The strongest belief (that which has the highest probability) is a variable node's *hard belief.* When the hard beliefs satisfy a termination condition, then they are a solution of the factor graph.

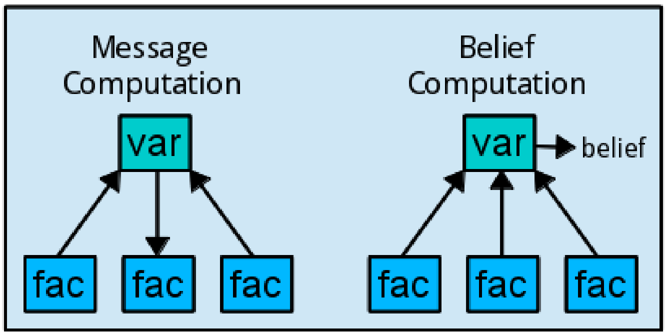


Fig 1. *Message and Belief Computation*

A variable node's beliefs are calculated the same as *joint probability*. They are the product of the incoming messages from factor nodes and the a priori data. A visual representation of the belief computation is shown in the right of Fig. 1. Variable node messages are *marginal probabilities*. This means that they are calculated the same as the beliefs, except that computing a message to a factor node disregards the message received from that factor node. In Fig. 2.1, this marginalization is shown on the left. Actual variable node message calculation will be introduced Section 4. Factor node messages are calculated as *marginal summaries*. Such a message is a function of the incoming messages from variable nodes and an application specific factor function. It uses a marginalization computation similar to that portrayed in the left of Fig. 2.1. Calculation of factor messages is often very computationally complex, as it requires evaluating the factor function for all possible permutations of connected variable node domain values [1]. For this reason, factor graph applications often compute an approximation function for factor node messages [2]. Actual factor node message calculation is included in Section 4.

The methods by which the variable and factor nodes compute their messages constitute the algorithm executed upon a factor graph. The two algorithms that are implemented in this work are the Sum-Product algorithm and the Min-Sum algorithm.

Chapter 3
Related Work

Computing resources do currently exist for modeling and solving FGs. Several software packages are available that take the form of programming libraries. They tend to be object oriented and run the gamut of languages, with implementations in C++, Java, Python, and others. The procedures for programming FG solvers with these libraries each follow similar lines: 1. Define variables and their domains; 2. Define factors (constraints); 3. Define a graph or variable interdependency structure; 4. Specify an algorithm and number of iterations; 5. Specify a priori information; 6. Initiate the solver. Though the APIs look different and implementation details vary, this coding flow is followed almost without deviation in each software. Of the many FG/PGM libraries that exist, a few notable ones will be subsequently discussed. Available FG hardware is considerably more sparse. What hardware does exist is nearly completely application specific, either as FPGA or ASIC implementations. A single exception, so far as the author is aware, is a device called GP5, and is capable of generic FG computation. It was developed by Lyric Semiconductor (now Lyric Labs of Analog Devices), and the name is an acronym for General Purpose Programmable Probability Processing Platform.

3.1 Factor Graph Software

OpenGM is a prominent FG/PGM tools library developed at Universität Heidelberg in Germany [3]. It is implemented in C++. It is capable of executing a wide variety of algorithms on arbitrary graph structures. It is templated, modular, and extensible. The majority of the currently published applications using OpenGM are directed towards 2-D and 3-D image and structure modelling. One such application is image segmentation, which attempts to partition an image into sets of like colors [4]. Fig. 2 shows an example [3]. Another application is scene decomposition, whose goal is to divide an image into partitions that represent discrete objects [5]. An example is portrayed in Fig. 3 [3].

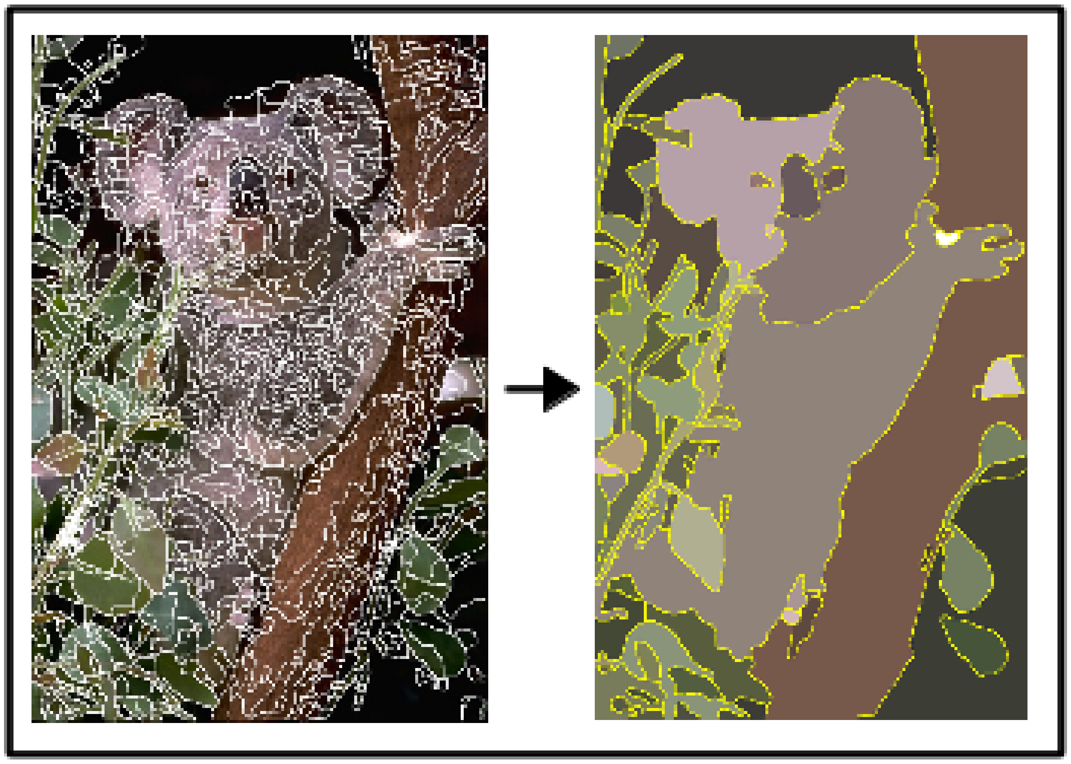


Fig 2. *Image Segmentation App using OpenGM.*
http://hci.iwr.uni-heidelberg.de/opengm2/, 2014
and used under fair use, 2014

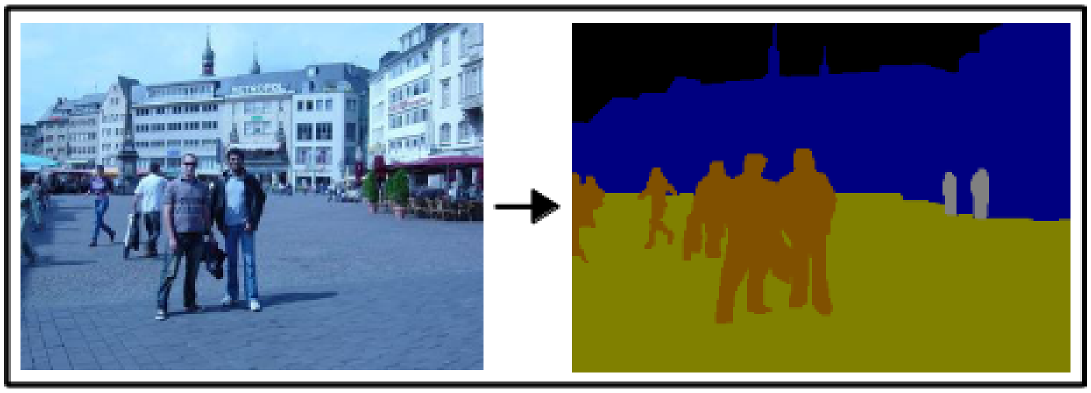


Fig 3. *Scene Decomposition App using OpenGM*
http://hci.iwr.uni-heidelberg.de/opengm2/, 2014
and used under fair use, 2014

Dimple is a software tool package for performing inference on PGMs [6]. It was developed at Lyric Labs and also serves as a software front-end for the GP5 development environment. It is implemented in Java, but also has a Matlab API. Dimple supports the standard FG algorithms, including Sum-Product, Min-Sum, and a few others. Notably, Dimple also supports continuous domain variables, whereas most FG/PGM software suites support only discrete variables.

SumProductLab is a Matlab package that enables easy construction and execution of FG solvers [7]. A user specifies constraint (factor) calculations, defines the structure of the graph, and provides evidence (a priori data), and the software executes belief propagation (the Sum-Product algorithm).

3.2 Factor Graph Hardware

Information decoding is a common application for FG's. An FG solver, implemented as an LDPC (low density parity check) decoder, is capable of repairing damaged codewords received through a noisy channel, such as wireless data transmission. Throughput is a major consideration for such an application, and so for this reason decoders as FG solvers are implemented in hardware. A wide variety of hardware LDPC decoder implementations exist. Some are fully parallel, such as [8] and [9], some are partially parallel, such as [10], and still others are bit-wise serial such as [11]. What they all have in common though, is that they may only repair codewords constructed according to the coding with which they were designed. To repair a codeword constructed with a different coding would require an entirely new design.

The GP5 sought to avoid such a constraint. It is a programmable instruction set co-processor designed to solve FGs. The user specifies a graph structure and a factor function, selects an algorithm, and provides a priori data. But it has several shortcomings that hinder performance and prevent it from being a desirable replacement for application specific hardware FG solvers. First is that it performs full evaluations of factor functions, which have computational complexity $O(n*d^n)$, where d is the domain, and n is the number of connected nodes [1]. This is often unnecessary, as approximations are usually sufficient for such calculations and can reduce the complexity by significant amounts [2]. Secondly, it is implemented as a co-processor, and has no memory subsystem. Consequently, it cannot retrieve its own instructions. It must interrupt its master processor when its instruction sequencers begin to run low, so that it may be fed more. Finally, the master processor to which it is connected is a MicroBlaze soft-core processor implemented on a Xilinx Virtex-6 FPGA. The interrupt latency associated with the MicroBlaze is likely a major contributor to its poor performance. A turbo code decoder implemented on the GP5, using a codeword length of 1024, has realizable throughput of 590 bits per second. So while GP5 is fully programmable and able to solve a plethora of FG structures and applications, its performance makes it ill-suited for those applications with real-time constraints.

Hardware designs also exist for solving PGM problems, but without using FG's. For example, this work includes a Sudoku solver as a FG, but Sudokus can be solved in other ways. As a PGM with a discrete domain, only so many board states can exist in a Sudoku puzzle. A brute force approach using high speed logic and parallel computation can be applied. An FPGA implementation of a such a Sudoku solver was developed at Delft University of Technology in the Netherlands [12]. Another Sudoku solver application developed at Technical University of Crete uses a method called *simulated annealing* [13]. It uses a function as a means of evaluating the board state and seeks to minimize (or maximize) the function while introducing random perturbations.

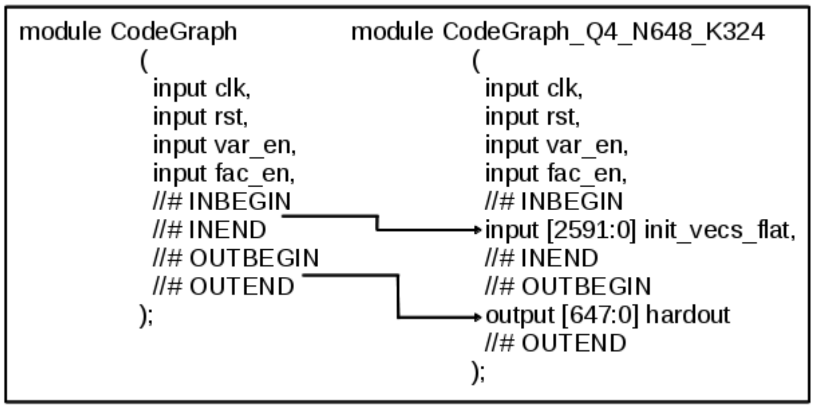
Chapter 4
The Framework

The system presented is a framework for generating hardware designs for factor graph solvers, and will henceforth be referred to as the *Generator Framework*. Its goal is to allow a developer to specify certain aspects of a factor graph and the desired computation, and return a hardware design, ready for implementation. Factor graph problems can vary in four criteria, and these four criteria constitute the full definition of a solver. They are graph structure, factor computation, iterative algorithm, and termination. This kind of generic model for a solver permits programmability based on the defining criteria. A programmable hardware FG-Solver generator would permit easy prototyping of FG designs, reduce the time to implement a viable solver, relax the often high skill requirements inherent in hardware description language (HDL) coding, and ease the costs associated with hardware development time. The only drawback from having an auto-generated design would be loss of customizability. But for the presented work, this point is moot since the system produces Verilog HDL source code, and not black-box IP. The user may modify the design as desired.

A system that seeks to automate the generation of FG-Solver hardware designs must allow a developer to specify the desired options for the four aforementioned criteria. Ideally, such a system would not require the developer to write any source code, and would completely automate the HDL generation. The Generator Framework has fully accomplished this goal in some of the necessary criteria, and partially in others. The degrees to which complete automation of each criteria has been achieved will be discussed in the subsequent sections.

4.1 System Structure

The Generator Framework consists of a system of generator scripts and hardware description language templates that work in tandem to produce the HDL source code for a generated hardware design. Each module within a generated design that is dependent on any FG criteria has an associated generator script and a template. The template is a Verilog source file that is essentially a "canvas" on which the generator script may "paint." The template contains Verilog boiler-plate code such as module name declaration, parameters, and scope delimiters such as `endmodule`. It also contains any code that is guaranteed not to vary with different generator parameters. This includes things such as certain port definitions (*clk, rst, en,* etc), wires and assignments, and all or parts of synchronous `always` blocks. Lastly, it contains tags that are used by the generator scripts to find the positions within the template where code should be inserted. Such code insertions include module input/output ports, local parameter values, wire and register declarations and assignments, and module instantiations. Fig. 4 shows an example of a template tag before and after insertion by a generator script.



```
module CodeGraph
            (
             input clk,
             input rst,
             input var_en,
             input fac_en,
             //# INBEGIN
             //# INEND
             //# OUTBEGIN
             //# OUTEND
            );
```

```
module CodeGraph_Q4_N648_K324
            (
             input clk,
             input rst,
             input var_en,
             input fac_en,
             //# INBEGIN
             input [2591:0] init_vecs_flat,
             //# INEND
             //# OUTBEGIN
             output [647:0] hardout
             //# OUTEND
            );
```

Fig 4. *Code Generation*

A *generator script* is a Python script that takes parameters as command line arguments. The command line arguments represent generation parameters that are defined by the design criteria. Common parameters include the number of connections for variable and factor node modules, bit vector length for messages, and a unique design identifier. The script opens the appropriate template for writing and inserts code according to the generation parameters and the positions of the tags. Each tag within the template has a corresponding block of code within the generator script that will locate the tag and insert HDL code. The generator script will also make modifications to the module name, appending identifiers of parameters. This allows multiple runs of a generator script on the same template to generate varying modules for use within the same design. For example, a design may require two variable nodes, one with three connections to factor nodes, and one with four. One run of the generator script will append the module name with '_C3', representing the connections parameter's value of three, and the next run will append the module name with '_C4'. Finally, the script will save the newly generated module, using the modified module name as the filename.

4.2 Solver Hierarchy and Generation

An FG Solver consists of four components. These are the Graph, Scheduler, Input Stage, and Termination module. The Graph is a structure of connected variable and factor nodes. The Scheduler controls algorithm execution. The Input Stage stores the initial a priori data. And the Termination module tests the beliefs for a solution. Fig 5 shows a block diagram of the architecture of an FG Solver module.

Even though the structure of an FG Solver module doesn't vary with application, it still requires code generation and insertion. This is due to the variability of the bus widths of the initial data and belief data. Similarly, the Input State requires little code generation. It is essentially just a simple storage register of arbitrary size, clocked and enabled. Because of its simplicity, it is not explained in a dedicated section. The remainder of this section briefly covers the generation of each of the other FG Solver components.

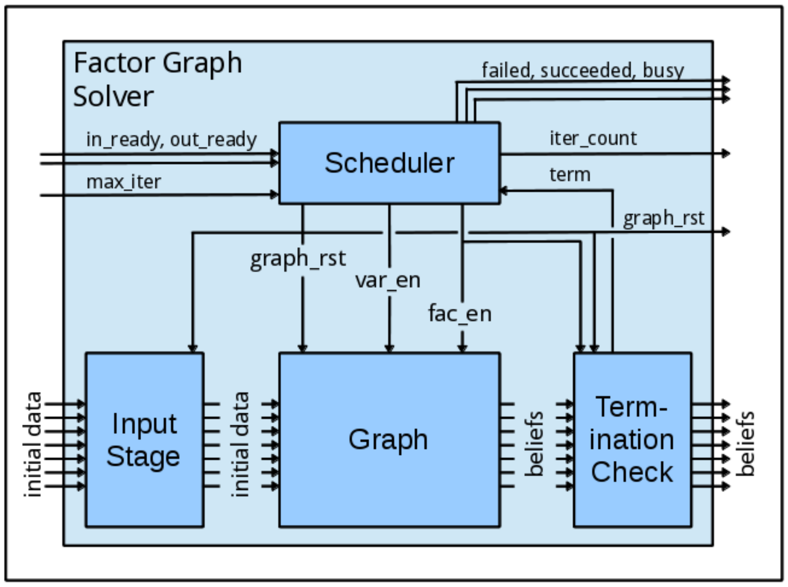


Fig 5. *Block Diagram of the FG Solver*

4.3 Variable Node Generation

Variable nodes vary according to several factors, such as message representation, number of connections, domain, and computational algorithm. All variable node modules share a similar interface. They take a clock, reset, and an enable signal as control inputs. Their data input consist of the output messages from the factor nodes to which they are connected. Their data output are the messages they compute to be passed along to those same factor nodes from which they received their input. Finally, they output a hard belief, which represents the most likely domain value for the variable node. Variable nodes are generated from templates that contain two insertion tags in the module interface: one for input signals and one for output signals. The number and width of these signals are governed by the number of connections the variable node shares with factor nodes, as well as the domain of the variable node. The generator scripts handle these determinations and appropriately populate the interface with the correct width and number of I/O signals for the new variable node. After the interface, the body of the variable node module must be generated. The body of a variable node for one algorithm can be substantially different from that of another. For this reason, a different variable node generator script exists for each supported algorithm. Currently, variable nodes for two algorithms are supported: the Sum-Product algorithm (S-P), and the Min-Sum (M-S) algorithm.

Variable node generation for the Sum-Product algorithm supports *N*-domain nodes with 16-bit floating-point probability representation, with any number of connections. The domain of a variable is the number of values that the node may represent. For example, a 4-domain variable node reflects the probability that the node could be any of four discrete values, perhaps 1, 2, 3, and 4. Each probability is represented by a 16-bit float. So the node stores a vector of probabilities, where the vector length is the domain. Using our 4-domain example, the node could possibly store the vector [0.5, 0.0, 0.25, 0.25], where 0.5 is the probability of value 1, 0.0 is the probability of value 2, etc. A variable node of *N*-domain is actually a hierarchical module, consisting of *N* instantiations of 1-domain variable nodes. Whereas an *N*-domain variable node represents each probability of all discrete domain values, a 1-domain variable node represents the probability of one single discrete domain value. See Fig. 6 for a visual representation of an example 4-domain variable node that is connected to three factor nodes.

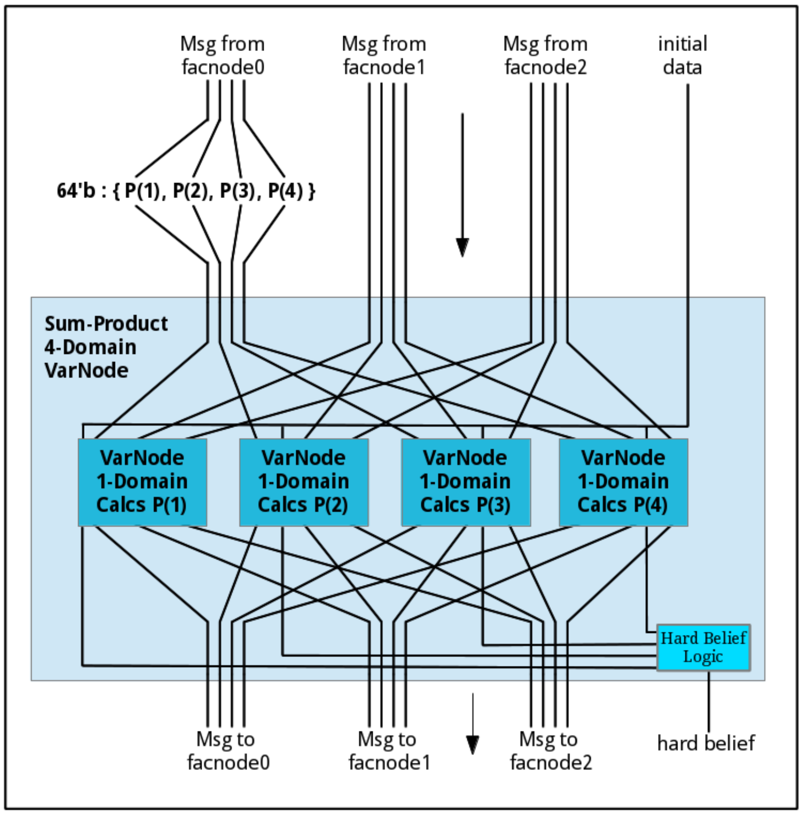

Msg from facnode0
Msg from facnode1
Msg from facnode2
initial data
64'b : { P(1), P(2), P(3), P(4) }
Sum-Product 4-Domain VarNode
VarNode 1-Domain Calcs P(1)
VarNode 1-Domain Calcs P(2)
VarNode 1-Domain Calcs P(3)
VarNode 1-Domain Calcs P(4)
Hard Belief Logic
Msg to facnode0
Msg to facnode1
Msg to facnode2
hard belief

Fig 6. *4-Domain Variable Node*

The incoming messages from the factor nodes are *16*N* bit wide vectors representing probabilities for each of the discrete domain elements. In an *N*-domain variable node, each message is split so that the respective probability values can be passed to the appropriate 1-domain variable node; therefore, each 1-domain variable node receives a part of the incoming message from each the factor nodes. An *N*-domain variable node also outputs a hard belief, representing the discrete domain value with the highest probability. The hard belief is computed with a max-decoder module that will be discussed later with algorithm termination. The 1-domain variable node instantiations, max-decoder instantiation, and temporary wire assignments constitute the generated code body of the *N*-domain variable node.

A 1-domain S-P variable node computes the probability of a single discrete value, since its domain is one. Its output messages are calculated according to (1) [2], and its output belief according to (2) [2]. The message and belief products are computed using zero latency 16-bit floating-point multipliers, and are assigned synchronously, so long as the enable signal is asserted. So with zero latency internal computation and a final synchronous assignment, the latency of the 1-domain S-P variable node is one clock cycle. Since an *N*-domain S-P variable node is merely hierarchical, it too is one cycle latency. Fig. 7 shows the architecture of a 1-domain S-P variable node. Cascading zero latency floating-point multipliers result in long combinational paths and restrict the clock frequency of the FG-Solver. This is done to keep control signalling simple during development. The possibilities and implications of using *N*-latency floating-point operators are discussed in the Conclusion section.

$$q_{n \to m}(x) = P(n = x) \cdot \prod_{m' \in M_{n,m}} r_{m' \to n}(x) \tag{1}$$

$$q_n(x) = P(n = x) \cdot \prod_{m \in M_n} r_{m \to n}(x) \tag{2}$$

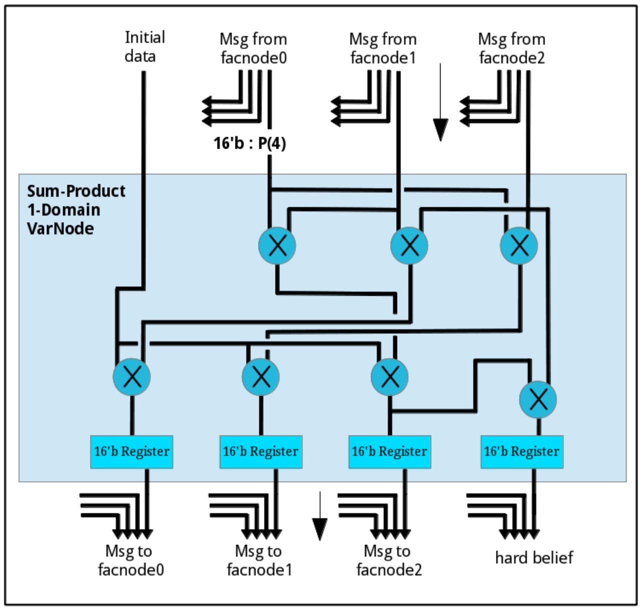


Fig 7. *1-Domain Variable Node for the Sum-Product Algorithm*

The notations of equations (1) and (2), as well as those of subsequent equations, is as follows: *q* represents a variable node computation, *r* represents a factor node computation, *n* represents a particular variable node, *m* represents a particular factor node, *N* represents the set of all variable nodes, and *M* represents the set of all factor nodes. A notation such as $q_{n\rightarrow m}(x)$ should be read as, "the variable node message from node *n* to node *m* regarding discrete *x*." The notation M_n should be read as, "the set of all factor nodes connected to variable node *n*." Lastly, the notation $M_{n,m}$ should be read as, "the set of all factor nodes connected to variable node *n*, except for *m*."

Variable node generation for the M-S algorithm supports 1-domain nodes with *N*-bit signed integer log-likelihood representation, with any number of connections. This variable node type is intended for binary domain applications, such as decoders. A single M-S variable node stores the probability of being a zero. A 2-domain hierarchical node is unnecessary, since the probability of a one is intrinsically stored within the probability of a zero, as P(1) = 1 - P(0). Furthermore, the probability is stored as a log-likelihood ratio (LLR), as calculated in (3) [14]. Output messages are calculated according to (4) [15], and the output beliefs according to (5) [15].

$$LLR = log\frac{P(0)}{P(1)} \tag{3}$$

$$q_{n\rightarrow m}(x) = LLR(n=x) + \sum_{m' \in M_{n,m}} r_{m'\rightarrow n}(x) \tag{4}$$

$$q_n(x) = LLR(n=x) + \sum_{m \in M_n} r_{m\rightarrow n}(x) \tag{5}$$

The message and belief sums are computed using zero latency saturation integer arithmetic. A hard belief is calculated by examining the sign of the belief sum. A positive belief LLR implies that P(0) > P(1) and so *0* is output as the hard belief. A negative belief LLR results in a hard belief of *1*. Fig. 8 shows the architecture of a 1-domain S-P variable node.

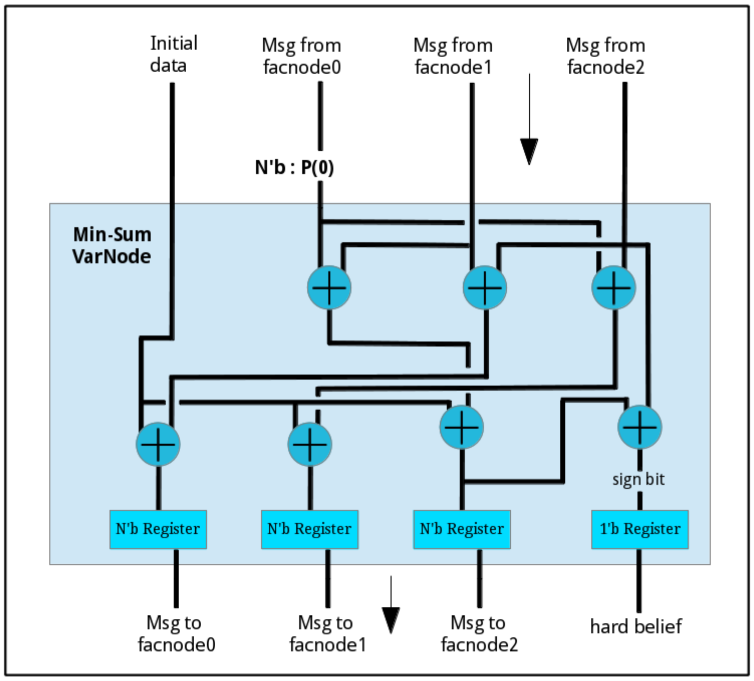


Fig 8. *1-Domain Variable Node for Min-Sum Algorithm*

Finally, the M-S variable node generator supports 'hybrid' message passing. Hybrid-*N* message passing is a technique for reducing wires and routing complexity by passing messages with *N* fewer quantization bits than are used within the nodes for computation. For example, if 4-bit quantization is used within the nodes, hybrid-1 message passing would only pass three (*4 - 1 = 3*) bit messages. Rather than implement this option with new interfaces for the nodes, it is accomplished by disregarding the least significant bit within the nodes during computation. It must be noted however, that this method relies on the synthesis tools to recognize that the least significant bit wires of the inter-node connections are not driving anything and can be optimized away. Currently only hybrid-1 message passing is supported. Hybrid-2 message passing attempted, though with mixed results. The effects of hybrid-1 message passing on performance and resource utilization are examined in Section 5 when the first application is discussed, as will a discussion of the attempt at hybrid-2.

4.4 Factor Node Generation

Like a variable node, a factor node varies depending on message representation, number of connections, domain, and computation algorithm. It receives clock, reset, and enable as control inputs, and its input and output data messages are generated based on the connections, representation, and domain. Though unlike a variable node, a factor node does not output a belief. There exists a more fundamental difference as well. A variable node is algorithm specific. Different applications may use common variable node modules, so long as they both employ the same algorithm. For instance, an application using the Sum-Product algorithm may re-use the variable modules generated by another S-P application. A factor node however, is not just algorithm specific. It is also application specific. In an abstract sense, a factor node represents a constraint of the PGM used to model a problem. As different problems have different constraints, so too must different applications use different factor nodes. Currently, the Generator Framework supports the generation of factor nodes for two applications: Min-Sum for LDPC decoders, and Sum-Product for a Sudoku solver.

Min-Sum LDPC factor node generation supports 1-domain nodes, with *N*-bit signed integer LLR representation, with any number of connections. The representation of probabilities as LLRs is identical to that of the Min-Sum variable node. Output messages are calculated according to (6) [15], using *XOR* logic on the signs of the input messages and modules that compute absolute-value minimums. *XOR* assignments and abs-min modules are zero latency, and the output messages are assigned synchronously. So the latency of the M-S LDPC factor node is one cycle, matching that of the M-S variable node. The *XOR* assignments, abs-min instantiations, and synchronous output message assignments constitute the generated code body of the M-S LDPC factor node module. Fig. 9 shows the architecture of a 1-domain M-S factor node.

$$r_{m\rightarrow n}(x) = \Big[\prod_{n' \in N_{m,n}} sign(q_{n'\rightarrow m}(x))\Big]\cdot min_{n' \in N_{m,n}} |q_{n'\rightarrow m}(x)| \tag{6}$$

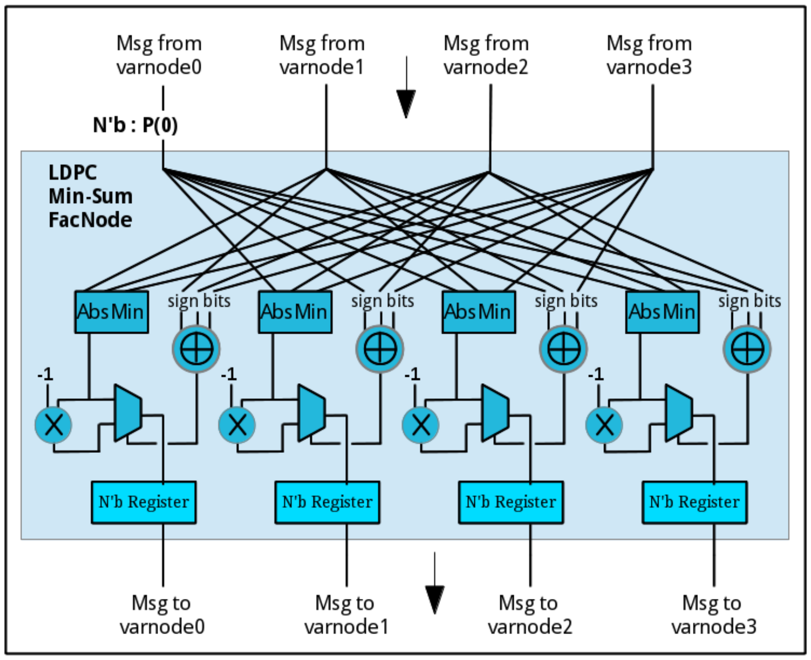


Fig 9. *1-Domain Factor Node for Min-Sum Algorithm*

Hybrid-1 messaging is supported for M-S LDPC factor nodes. A hybrid factor node module's interface remains the same, and the least significant bit wires of the input messages are disregarded, just as with a hybrid M-S variable node.

Sum-Product Sudoku factor node generation supports *N*-domain nodes, with 16-bit floating point probability representation, with any number of connections. Such *N*-domain factor nodes are hierarchical, just as *N*-domain S-P variable nodes, and are comprised of *N* instantiations of 1-domain factor nodes. Like their variable node counterparts, the input messages are divided and routed to the inputs of the appropriate 1-domain factor nodes, and the outputs of the 1-domain factor nodes are routed and concatenated to create the output messages of the *N*-domain factor node. These connections and the instantiations of the 1-domain factor nodes constitute the generated code body of the *N*-domain S-P Sudoku factor node module. Fig. 10 shows the architecture of a 4-domain S-P factor node.

A 1-domain S-P Sudoku factor node calculates the marginal summaries for one discrete value of the domain. It computes its output messages according to (7) [2], and is an approximation of the true summary computation.

$$r_{m\to n}(x) = \prod_{n' \in N_{m,n}} (1.0 - q_{n'\to m}(x)) \qquad (7)$$

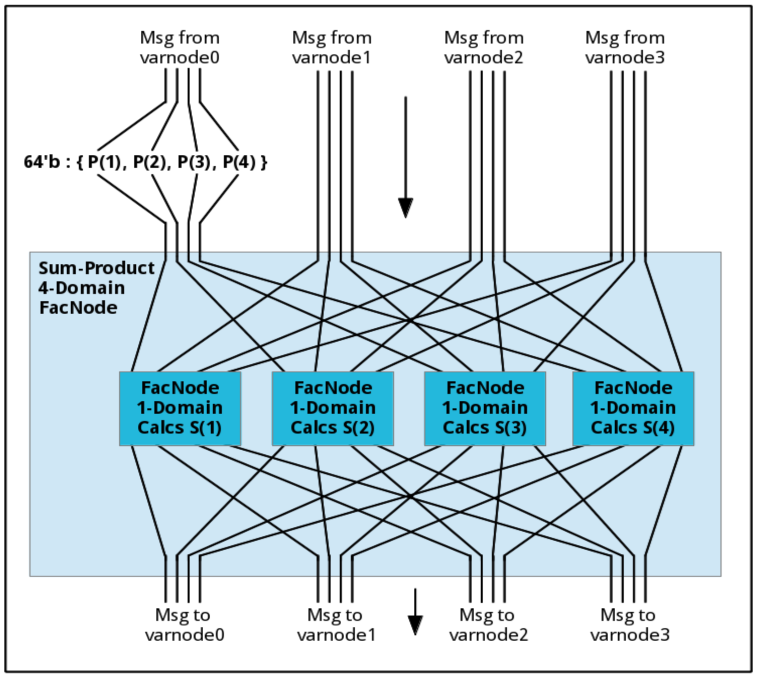

Msg from varnode0
Msg from varnode1
Msg from varnode2
Msg from varnode3
64'b : { P(1), P(2), P(3), P(4) }
Sum-Product 4-Domain FacNode
FacNode 1-Domain Calcs S(1)
FacNode 1-Domain Calcs S(2)
FacNode 1-Domain Calcs S(3)
FacNode 1-Domain Calcs S(4)
Msg to varnode0
Msg to varnode1
Msg to varnode2
Msg to varnode3

Fig 10. *4-Domain S-P Factor Node*

Products are computed using zero latency 16-bit floating-point subtractors and multipliers. Output messages are assigned synchronously, provided that the enable signal is asserted. So with zero latency internal computation and synchronous output assignment, the latency of the 1-domain S-P Sudoku factor node module is 1 cycle, as is any *N*-domain factor node module comprised of them. The multiplier and subtractor instantiations and the synchronous output assignments constitute the generated code body of the 1-domain S-P Sudoku factor node module. Fig. 11 shows the architecture of a 4-domain factor node.

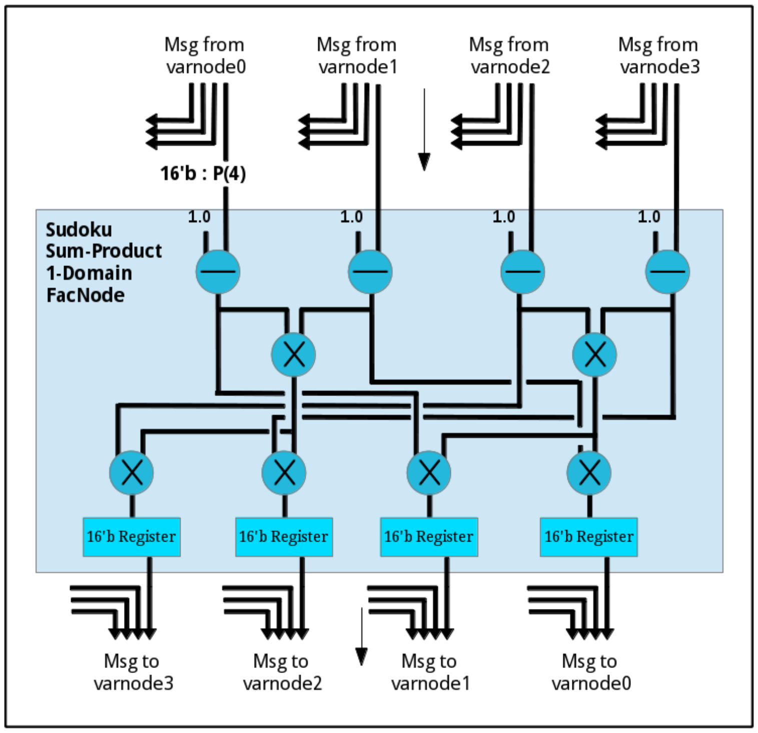
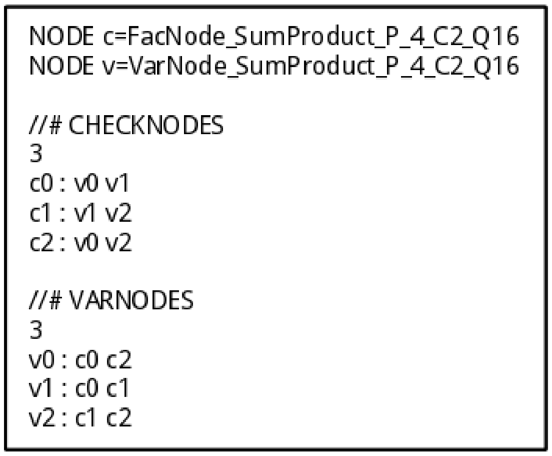


Fig 11. *4-Domain Factor Node*

4.5 Graph Construction

Graph module generation is dependent only upon the structure of the factor graph, and the quantization of the messages and beliefs. It is completely independent of application and algorithm. The graph template requires code insertion in five places. The first and second steps in graph generation are modifying the interface for a priori input and hard belief output. The initial data are input to the graph as a monolithic bus, all concatenated together. Likewise the hard belief data is output from the graph as a single concatenated bus. The next step is to insert code for declaration of input and output memories (i.e., multi-dimensional bit arrays), and messaging memories. The initial data is moved from the monolithic bus into a memory so that it may be easily referenced, avoiding complex sub-vector referencing. Its width is the width of a single initial datum intended for a single variable node. The memory's depth matches the number of variable nodes. The hard beliefs memory's width is the width of a single hard belief output from a variable node, and the depth matches the number of variable nodes. Two more memories are created: one for variable node to factor node messages, and one for factor node to variable node messages. Their widths are equal to the width of a single message, and their depths are equal to the number of edges in the graph. Next, a Verilog `generate` block must be inserted to make the assignments from the initial data bus to the initial data memory, and from the hard beliefs memory to the hard beliefs bus. Finally, node instantiations must be inserted.

To insert the node instantiations and complete the generation of the graph module, the structure of the factor graph must be input to the Generator Framework. This is accomplished via a *graph definition* file. A simple example of a graph definition file is shown in Fig 12. In the first section, node mappings are defined. Local node type identifiers are mapped to Verilog module names. The next two sections describe the connections for each factor and variable node. Both sections share the same format. The first line of a section contains the delimiter. Next is the section's node count. Then follow the connection definitions for each node. The first term in the connection line represents the local node type and number of the node currently being defined. The remaining terms that follow the colon identify the nodes to which the currently defined node is connected. In other words, the 2nd 'c' type node (c2) is connected to the 0th and



```
NODE c=FacNode_SumProduct_P_4_C2_Q16
NODE v=VarNode_SumProduct_P_4_C2_Q16

//# CHECKNODES
3
c0 : v0 v1
c1 : v1 v2
c2 : v0 v2

//# VARNODES
3
v0 : c0 c2
v1 : c0 c1
v2 : c1 c2
```

Fig 12. *Graph Definition File*

2nd ‘v’ type nodes (v0, v2). The graph module generator uses this connection data to instantiate an appropriate number of variable nodes and factor nodes, whose module types and input and output connections are specified by the graph definition file. After module instantiations, the graph module is a complete representation of the structure of the factor graph.

4.6 Algorithm Termination

Algorithms on factor graphs are iterative. They must either cease after a number of iterations, or reach a terminating condition. Such a terminating condition represents a solution, and is dependent upon the hard belief outputs of the variable nodes. A solution is specific to an application, so different applications require different termination modules. But while the calculation of the terminating condition varies, the general structure of termination modules remain the same. The interface does change, though slightly. All termination modules take a clock, reset, and enable as control signals, and output a termination signal. What varies are the bus widths of the input and output hard beliefs. For example, a solver that has 20 variable nodes and whose hard beliefs are 2 bits each will require 40-bit input and output buses. A `generate` block also varies according to the hard belief width. It's purpose is to create an easily indexed memory for the beliefs. These variances are handled by the generator script. What is not handled is the terminating condition calculation.

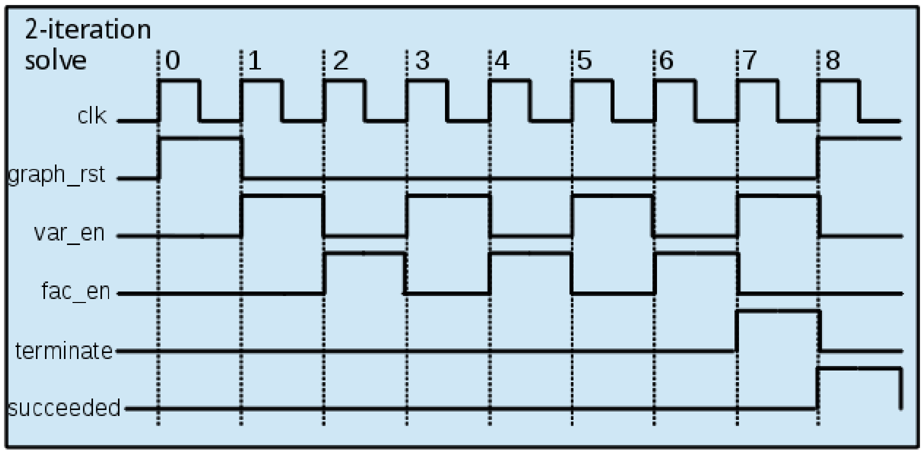
The terminating condition calculation must inserted manually by the developer. For compatibility with the currently supported scheme of control signal scheduling, it must be computed with zero cycle latency, using wires and asynchronous assignments. The template includes an `always` block for synchronous set and reset of the output termination signal, and does not require modification. The developer must ensure that the name of the wire storing the result of their calculation matches the wire name used in the final synchronous assignment.

The termination module is enabled by the same signal that enables factor nodes. So when it is determined that a terminating condition has been met, factor nodes are simultaneously computing their messages. Now that the termination signal is asserted, it cannot be acted upon until the next cycle, when variable nodes are computing their messages and hard beliefs. So by the time termination can be acted upon, the hard beliefs that triggered the terminating condition may have changed. For this reason, the terminating module stores the hard belief solution whenever it is enabled.

4.7 Control Signaling

Factor graph solvers require control signals. Resets and enables must be properly scheduled to ensure the correct execution of an iterative algorithm. The Scheduler module is responsible for this control, and works in a closed-loop fashion. It acts as a source for the graph's reset and enable signals, and receives feedback from the termination module. It is also responsible for tracking algorithm iterations and outputting solver status signals.

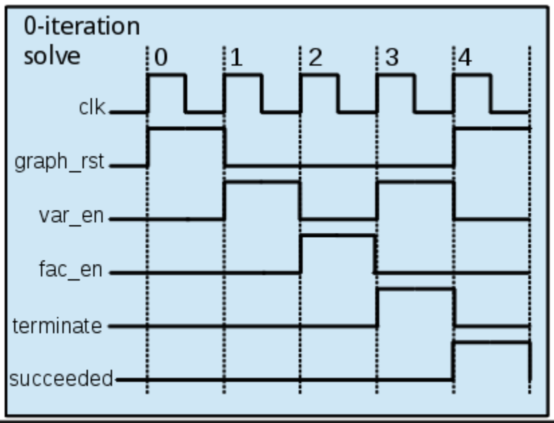
When the data producer (input) and consumer (output) assert that they are ready, the Scheduler begins an execution sequence. First, the Scheduler asserts the graph reset (*graph_rst*) signal. This stages the input data, and resets the graph nodes and termination module. The variable node enable (*var_en*) signal is asserted next, causing the variable nodes to compute their messages and beliefs. Next the factor enable (*fac_en*) signal is asserted, causing the factor nodes to compute their messages. The variable enable and the factor enable are asserted alternately until the termination signal is asserted, or until a maximum number of iterations has been reached. The first variable enable assertion isn't part of a true iteration, because the factor messages arriving at the variable nodes are reset. So initially, the variable nodes' messages are simply the a priori data. Afterwards, a full algorithm iteration consists of a factor enable assertion followed by a variable enable assertion. All control signal assertions are held for a single cycle of the FG-Solver clock.



clk 1 : Var node and fac node msgs reset, beliefs reset, input state set.
clk 2 : Initial var node msgs and beliefs are set.
clk 3 : Fac nodes compute msgs.
clk 4 : Var nodes compute msgs, beliefs. One full iteration complete.
clk 6 : Second full iteration complete. Beliefs now represent a solution.
clk 7 : Algorithm termination conditions satisfied. Terminate is asserted.
clk 8 : Succeeded asserted. Next sequence may begin.

Fig 13. *2-Iteration Solver Sequence Timing Diagram*

Fig. 13 shows a timing diagram for a solver execution sequence, which reaches a terminating condition after two full iterations. The figure provides annotations for the timing diagram sequence, explaining each step towards completion. All synchronous assignments are performed on the positive edge of the solver clock. Graph reset consumes one cycle, first variable enable consumes one cycle, a full iteration consumes two cycles, termination signal assertion consumes one cycle, and solver completion consumes one cycle. So, an execution sequence that requires *N* iterations consumes *2*N + 4* cycles. To easily visualize why the four extra cycles are necessary, Fig 14 shows a timing diagram for a zero-iteration execution sequence. This would occur if the a priori data itself satisfies the terminating condition, and so represents a solution.



clk 1 : Var node and fac node msgs reset, beliefs reset, input state set.
clk 2 : Initial var node msgs, beliefs are set. Beliefs represent solution.
clk 3 : Termination conditions satisfied. Terminate is asserted.
clk 4 : Succeeded asserted. Next sequence may begin.

Fig 14. *0-Iteration Solver Sequence Timing Diagram*

The Scheduler asserts one of two completion signals when the execution sequence has finished: *succeeded* or *failed*. Provided that the initial execution sequence conditions are met, the Scheduler may simultaneously assert the graph reset signal and begin a new sequence. The Scheduler also outputs a *busy* status signal, and an iteration counter.

The Scheduler is implemented as a Mealy state machine. While it is a part of the Generator Framework, it is currently a static module. User defined control scheduling is not yet supported.

4.8 Input/Output Subsystems

The Generator Framework currently supports only Ethernet input and output. Fig. 15 shows a system-level block diagram that includes the Top level hierarchy of a generated

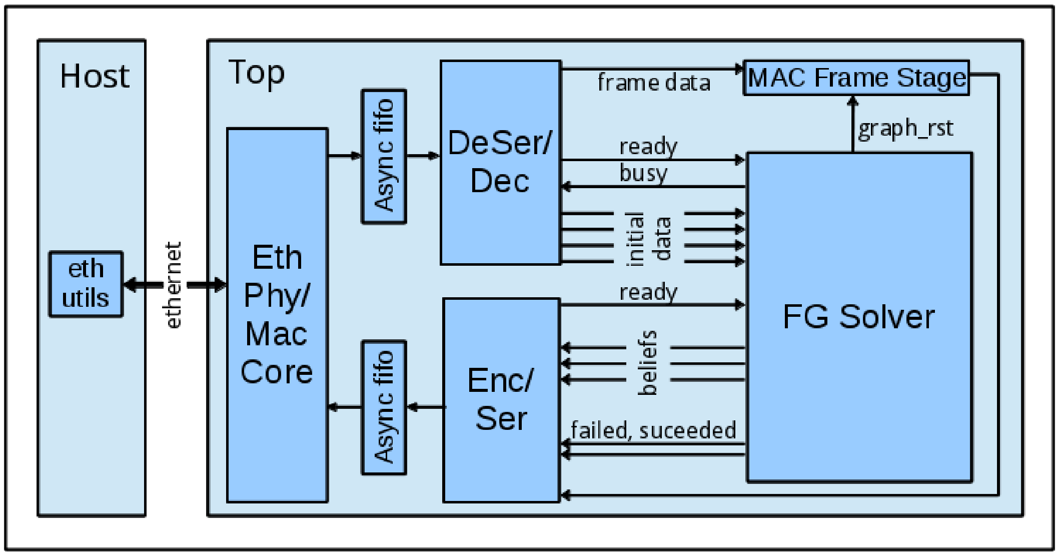


Fig 15. *Top Level Architecture*

design. Input and output data are sourced from and sinked to an Ethernet Phy/MAC core via an AXI-Streaming interface. Development implementations have targeted a Xilinx Virtex-6 XC6VLX240T-1 FGPA on a ML605 development board. The Ethernet Phy/MAC core and asynchronous fifo's that are connected to it are Xilinx CoreGen generated IP and are platform specific. The remaining I/O architecture is device independent. So an implementation targeting a different device or platform would require that those modules be replaced appropriately. The only requirement to maintain compatibility with the rest of the Ethernet I/O system is that the fifo's must have 8-bit data width AXI-S interfaces.

The Deserializer/Decoder takes serial bytes from the input fifo, and then decodes the MAC frame. It presents the frame header to the MAC Frame Stage module for temporary storage while the solver works, and presents the deserialized data to the FG-

Solver. The MAC Frame Stage also stores a solver metadata header, sent with the initial data. This metadata consists of a command byte, a parameter byte, and a two byte tag. The command byte is intended as a solver selector, to enable multiplexing amongst multiple solvers on a device, though this is currently not supported. The parameter byte is used to set the maximum number of iterations for an execution sequence. The tag bytes simply serve as an identifier for initial input data, so that software may match against received output belief data. After execution completion, the Encoder/Serializer constructs a MAC frame from the MAC frame header, solver metadata header, and beliefs vector, and then serializes and sends it to the output fifo. First though, it alters the command byte of the solver metadata header to indicate a success or failure, and alters the parameter byte to reflect the number of iterations taken during the execution sequence.

Host-side communication is achieved with a set of ethernet utilities implemented in Python. The system's eth_utils module provides methods for transmit, receive, and for opening a Datalink Layer raw socket object. To transmit data to an implemented design, a user calls the tx method and specifies the MAC addresses of the source host and device target, along with a payload. The payload consists of a command byte, parameter byte, two tag bytes, and the initial solver data, all concatenated. To receive results, a user calls the rx method and specifies the same MAC addresses, and the number of bytes to receive. For an *N*-byte solver result, *N+4* bytes must be specified, to account for the solver metadata header. The solver automatically sends back the results of the execution sequence, but the user must call the rx method to retrieve it from the socket.

Chapter 5
Application 1: LDPC Decoder Basket

The requirements for the first application selection are three-fold. It is intended to be developed in tandem with the Generator Framework itself. Therefore, it should be simple so that debugging the Generator Framework is less complex. Second, it should be dynamic, able to vary in response and performance given different parameters. Lastly, it is desirable to select an application whose results can be compared with other published work. For these reasons, Low Density Parity Check (LDPC) decoders were selected as the first application. The applicable message passing algorithm needs simple arithmetic and requires only small quantization, so debugging via an HDL simulator or logic analyzer is simplified. LDPC decoders can vary along many parameters, such as codeword length, rate, quantization, maximum permitted algorithm iterations, and the amount of noise applied to codewords. Their response and performance are accurately and precisely measured as bit error rate, average iterations, and throughput versus applied noise. So a basket of LDPC decoders whose characteristics can be varied and whose responses can be measured and compared makes an excellent choice for a first application. In addition, there are published results of hardware implementations of LDPC decoders with which to compare.

5.1 LDPC Codes Explained

An LDPC code is a linear forward error correcting code that is defined by a generator matrix and a check matrix. A message of length *K*, when multiplied by a *KxN* generator matrix G, results in a codeword of length *N*. A *(N-K)xN* check matrix H, when multiplied by a transposed codeword of length *N*, results in a parity vector. If the parity vector contains all zeros, then the codeword is good. Fig. 16 shows an example [16] of a codeword generated from a message, and parities checked from a codeword.

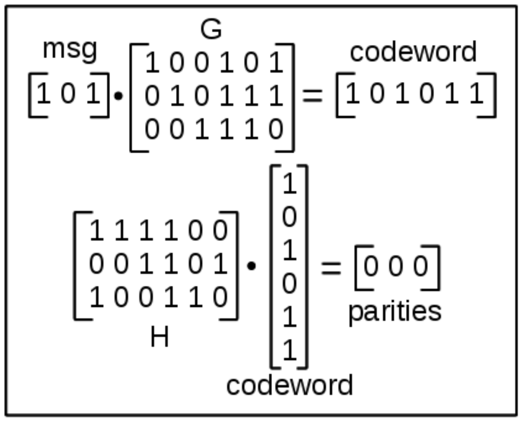


Fig 16. *Codeword generation and parity checking of LDPC code*

LDPC decoders map well to factor graphs. Each bit of a codeword is represented by a variable node and corresponds to a column in the check matrix H. The first variable node, representing the first bit of the codeword, is connected to factor nodes 1 and 3, according to examination of the first column of H. Each parity check is represented by a factor node and corresponds to a row in H. The first factor node is connected to variable nodes 1, 2, 3, and 4, as seen by examining the first row of H. Thus the parity check matrix H provides a direct mapping to a factor graph that represents the LDPC code. Fig. 17 shows a factor graph representation of the LDPC code presented above.

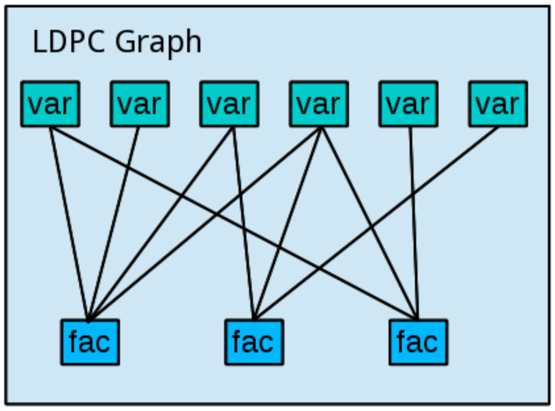


***Fig 17.** Factor graph of LDPC code*

5.2 Results

The test set for this application includes 37 designs made with the Generator Framework. A Python configuration script was used to store user-input parameter data for each design. Design generation, implementation, and testing was fully automated, again with Python scripts. Testing was conducted on a Xilinx Virtex 6 VC6VLX240T, using an ethernet interface.

5.2.1 Codeword Length Variation

First, codeword length variation is tested. The decoder response of six codeword lengths is presented, where length N ∈ {120, 200, 400, 800, 1008, 1156}. Two codes of the same length could have very different characteristics and response, resulting from differences in their graph structure. For this reason, for each length N except for N=1008 and N=1156, five codes and decoders were constructed, so that the variations in response could be averaged. Third party software was employed for code construction. The first four codes were generated with LDPC-codes [17], and the last two with MainPEG [18]. All codes are rate R=1/2, which refers to the ratio of message length to codeword length. Each code is also (3,6)-regular, meaning that variable nodes are connected to three factor nodes, and factor nodes are connected to six variable nodes. This is not a strict requirement however. The code generation software will make slight modifications to variable and factor node connectivity to prevent certain structural characteristics in the factor graph that are detrimental to performance, such as cycles of length four [19]. Figs. 18 and 19 show Bit Error Rate (*BER*) and Average Iterations (I_{avg}), respectively, vs. Signal-to-Noise Ratio (*SNR*).

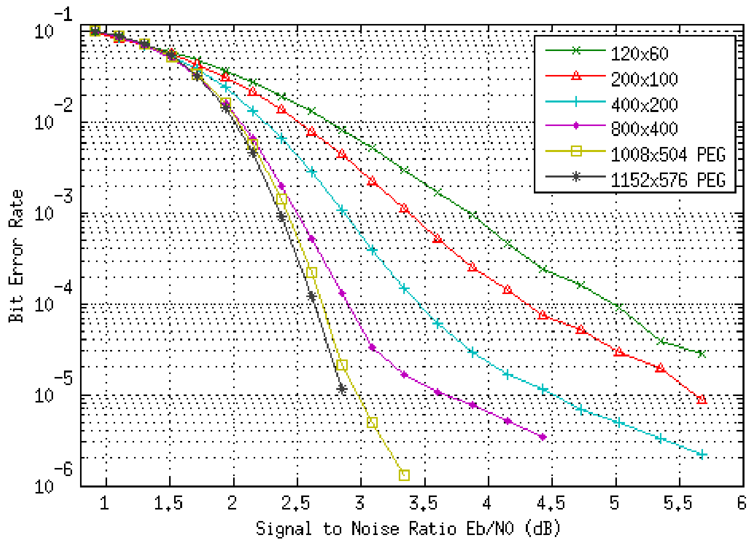


Fig 18. *BER vs. SNR for varying N*

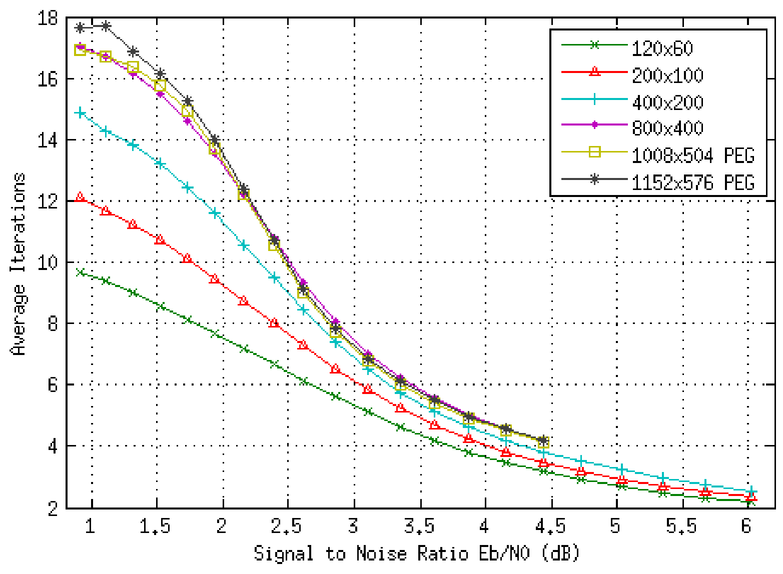


Fig 19. *I_{avg} vs SNR for varying N*

Better performance is characterized by lower *BER*, and in Fig. 5.3 it is shown that performance increases with each step up in codeword length. This is the expected behavior. Small cycles within an LDPC graph adversely affect decoder performance. Since column and row weights (the number of 1's in a row or column of the parity check matrix) do not change with increasing codeword length, the average length of the cycles in the graph increases instead. This results in better performance for larger codewords [19]. Decoder response is also characterized by how pronounced is the *waterfall* region. This is the sharply negative slope region of a data series in the plot. As codeword length increases, the waterfall region becomes more pronounced with ever increasingly negative slope, further verifying the correct response from the decoders. Fig. 5.4 shows the average number of iterations required to successfully decode a noisy codeword. The response of the decoder set is as expected. Larger codes require more iterations to propagate data through longer graph cycles. This difference becomes exaggerated as *SNR* falls and the decoders have to work longer to repair the effects of higher noise.

5.2.2 Codeword Rate Variation

Next the effect of varying the rate of the codewords is examined. Figs. 20 shows a plot of *BER* vs *SNR* for five codewords of N=360, whose rates are 1/2, 2/3, 3/4, 5/6, and 8/9. Lower rate codes are better suited for high noise environments since a larger proportion of the codeword is devoted to error correction. Higher rate codes are better suited for lower noise environments where there is less required error correction and the system can take advantage of the higher message data rate. From the plot, the better performers are the lower rate codes, which have higher error correcting capability. This is the expected response, and further verifies correct operation.

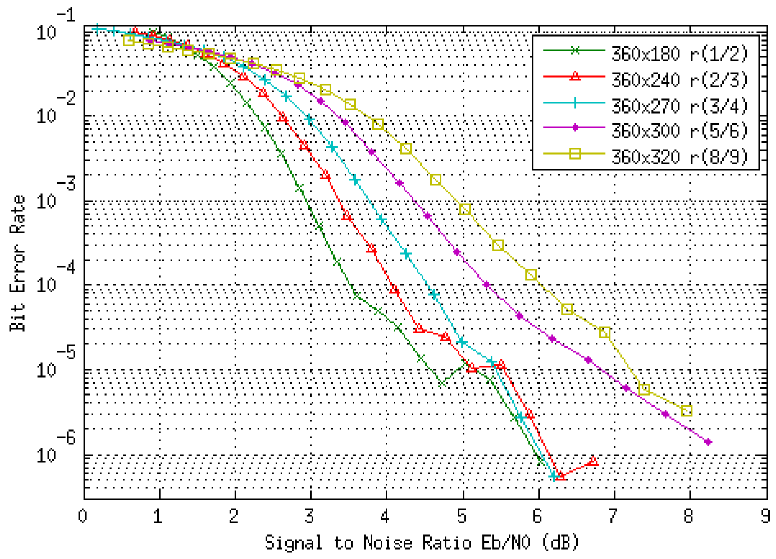


Fig 20 *BER vs SNR for varying rate codes*

5.2.3 Message Quantization Variation

Changing the quantization will elicit a response from the decoders as well. While the Generator Framework is capable of producing designs using an arbitrary quantization, only 4-bit quantization was employed. Implementing 3- or 5-bit quantization decoders is possible, but not desirable for the testing environment. This is because of the word size constraints of contemporary computer architecture. Four bits pack easily into bytes; three and five do not. Such quantizations would require complex bit manipulations in software, and were therefore avoided. However, we can measure the differences in response between a full quantization decoder and a decoder using the hybrid-1 message quantization discussed in Section 4. Figs. 21 and 22 display *BER* and I_{avg}, respectively, vs. *SNR* for full and hybrid-1 quantization decoders with N=400 and R=1/2. Each data series represents an average performance of five decoders.

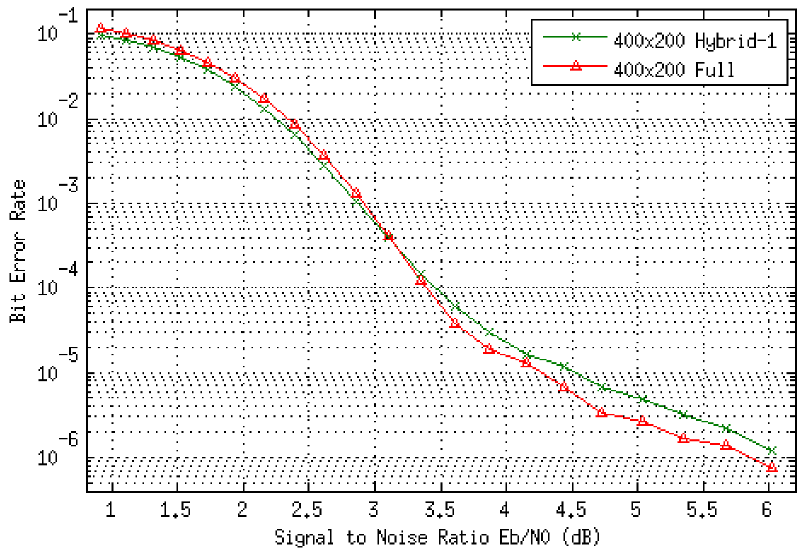


Fig 21. *BER vs. SNR for hybrid-1/full*

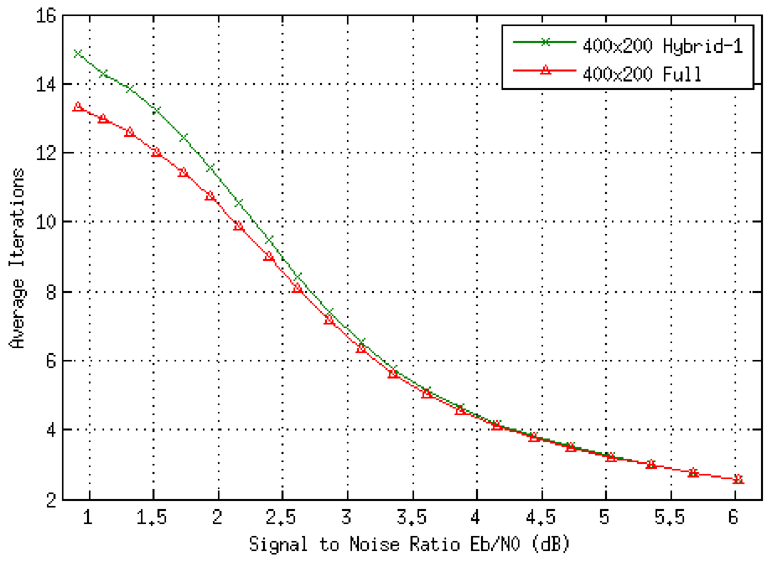


Fig 22. *I$_{avg}$ vs SNR for hybrid-1/full*

One would expect that the full quantization decoders would consistently outperform the hybrid-1 decoders across all *SNR*, since the hybrid-1 decoders are essentially throwing away information by disregarding the lowest message bits. But the full quantization decoders only performed better when *SNR* was greater than ~3.2 dB. However, examining the average iterations plot tells more of the story. It shows that the full quantization decoders consistently needed fewer iterations to complete, across all *SNRs*. This implies that the slightly worse *BER* performance of the full quantization decoders at low *SNR* is likely due to a few codewords with many bit errors, rather than a more even distribution of bit errors among codewords. By examining a plot of Word Error Rate (*WER*) vs *SNR*, this conjecture is verified. Fig. 23 shows that the *WER* performance for the full quantization decoders is indeed better than the hybrid decoders.

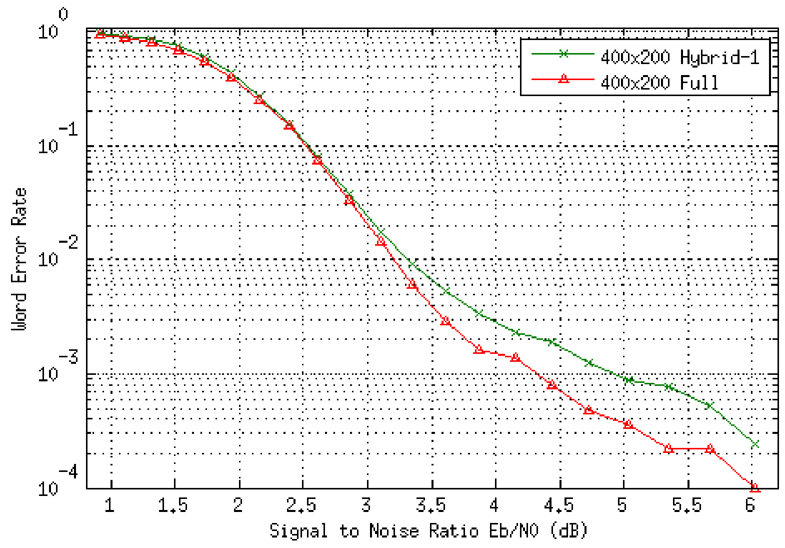


Fig 23. *WER vs SNR for hybrid-1/full*

A possible explanation for the slightly better low-noise *BER* performance of the hybrid decoder could be related to the especially noisy codewords. Since the noise is Additive White Gaussian, some codewords will be more noisy than others, even at the same *SNR*. Some of these codewords may also have a noise sampling applied in such a way that they are no longer possible to decode, given the structure of the coding graph. Changing the message quantization could perhaps introduce data perturbations that cause the beliefs to converge on the solution. This is only a conjecture, but is supported by the fact that the *WER* plot suggests that only a few codewords are to blame for the *BER* plot abnormality.

	FFs	LUTs	Wires	SNR at BER=10e-6
400x200 Full	13,882	44,889	242,402	5.2 dB
400x200 Hybrid-1	11,082 (-20.2%)	22,473 (-50.0%)	122,300 (-49.5%)	5.7 dB

Table 1. *Full vs Hybrid-1 Message Quantization*

The benefits and drawbacks of hybrid message passing are revealed in Table 1. By reducing the quantization of the messages by one bit, a 20% reduction in flip flops, a 50% reduction in lookup tables, and a 50% reduction in wires can be realized. There is a quantifiable cost however. The hybrid-1 quantization performs as well at 5.7 dB Eb/N0 as the full quantization does at 5.2 dB. In other words, the hybrid-1 quantization required a less noisy channel environment to perform as well as the full quantization.

5.2.4 Maximum Iterations Variation

The last parameter variation is the maximum number of iterations (I_{max}) that are permitted in an execution sequence. Six different values for max iterations were attempted on a N=800, R=1/2 decoder. They are 5, 10, 15, 20, 25, and 250. Fig. 24 shows *BER* vs *SNR* for each value of I_{max}.

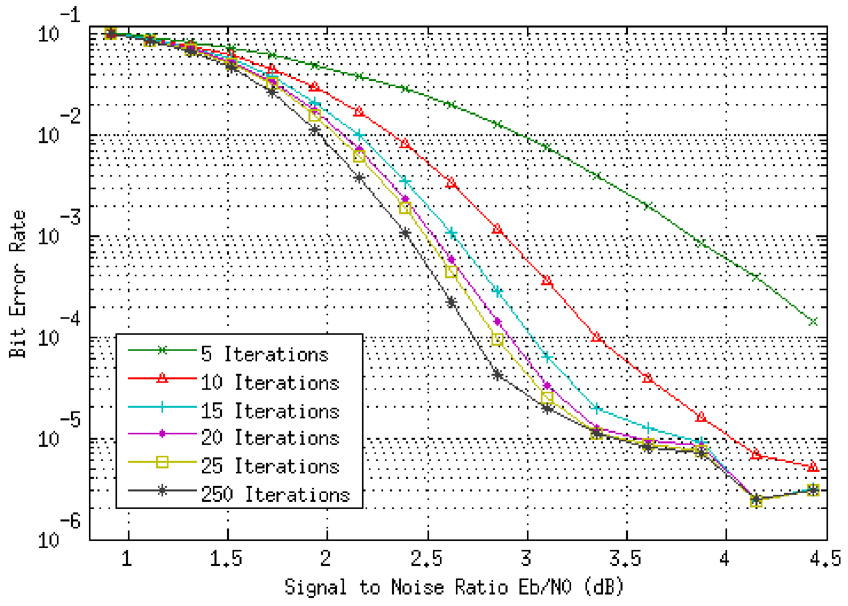


Fig 24. *BER vs. SNR for varying I_{max}*

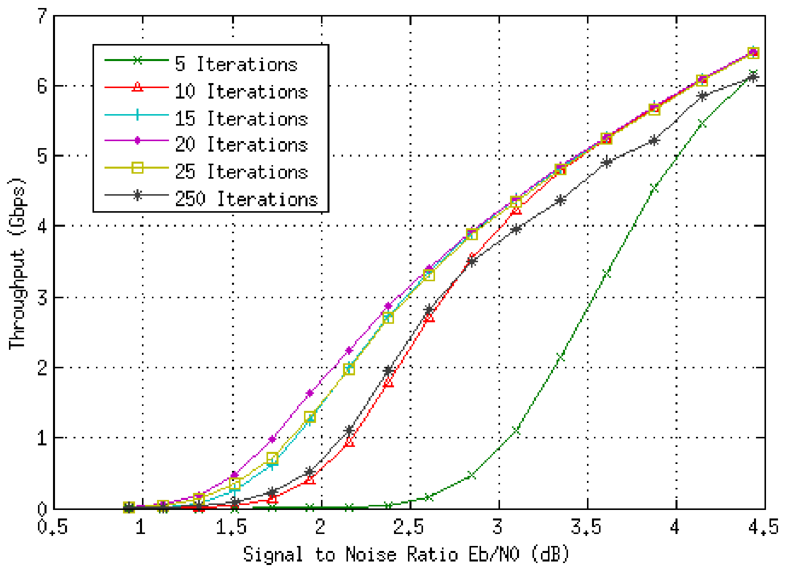


Fig 25. *Throughput vs SNR for varying I_{max}*

If a decoder is allowed to run longer to repair a noisy codeword, one would expect a better decoding performance. This is true, if the *BER* is the measure of performance. Fig. 24 shows that the more iterations are permitted, the lower the *BER*. However, the *BER* plot does not provide any insight into how efficient is the decoder. Fig. 25 shows a plot of throughput, in gigabits per second, vs *SNR*. Throughput is calculated according to (8) and (9).

$$Cycles\ per\ Word = [(1-WER)\cdot I_{avg} + WER \cdot I_{max}]\cdot 2 + 4 \qquad (8)$$

$$Throughput = \frac{(1-WER)\cdot F_{clk}\cdot N}{Cycles\ per\ Word} \qquad (9)$$

Here we see that each successive increase in I_{max} results in a higher throughput, until I_{max}=20. Then, I_{max}=25 is slightly worse, and I_{max}=250 is much worse. This is because certain noisy codewords that are simply unable to be repaired are allowed to prolong their execution sequence, resulting in wasted clock cycles and reduced throughput. Interestingly, this plot suggests that for all noise environments from 0.5 dB to 4.5 dB, I_{max}=20 is consistently as good or better than all other values.

5.3 Comparison with Published Work

Finally, this application set permits a side-by-side comparison with published results of similarly implemented decoders. Table 2 shows the implementation statistics of two decoders from this work compared against three similarly sized and quantized codewords from other published works [8] and [9].

	This Work	[8]	This Work	[8]	[9]
Length	1008	1000	1152	1152	1152
Quantization	4-3 (hybrid-1)	4-3 (hybrid-1)	4-3 (hybrid-1)	4-3 (hybrid-1)	4-2 (hybrid-2)
Device	Virtex 6 VC6VLX240T	Virtex 5 XC5VLX110	Virtex 6 VC6VLX240T	Virtex 5 XC5VLX155	Virtex 5 XC5VLX110T
FFs	25,702	24,012	29,154	27,372	15,107
LUTs	53,748	61,761	61,200	72,290	39,024
Clock	100 MHz	154.3 MHz	100 MHz	154.3 MHz	138 MHz
Eb/N0 at BER 10e-6	3.4 dB	3.5 dB	3 dB	3.5 dB	3.8 dB
Avg. Iterations at 3.5 dB SNR	5.6	5.8	5.7	5.8	8.3
Avg. Throughput at 3.5 db SNR	6.6 Gbps	14.5 Gbps	7.5 Gbps	16.9 Gbps	9.6 Gbps
Normalized Throughput at 3.5 dB SNR	6.6 Gbps	4.7 Gbps	7.5 Gbps	5.5 Gbps	7.0 Gbps

Table 2. *Comparison of Work (performance and resource utilization)*

First, a comparison between this work (henceforth called *TW*) and [8] will be drawn. TW and [8] are compared with respect to two different sized decoders. TW required slightly more flip flops than [8], with the two decoder designs requiring 6.6% and 6.1% more, respectively. It should be noted, however, that it is unclear whether [8] included the I/O system when accounting for its resource utilization. Also, TW used a codeword size that is eight bits larger than [8]. This is a negligible difference, but still accounts for some of the increased flip flop usage. TW required substantially fewer LUTs than [8], needing 14.9% and 18.1% less, respectively. Both TW and [8] show a similar noise level for a *BER* of 10e-6 for the smaller decoder. But TW's larger decoder outperforms that of [8] by 0.5 dB at a *BER* of 10e-6. This difference is likely due to the structure of the code itself, rather than hardware design. Average iterations are consistent between TW and

[8]. For both works, input and output latency are neglected when throughput is calculated. In TW, variable nodes and factor nodes alternate their computation. [8] uses a fully pipelined design, putting the variable nodes and factor nodes to work on every cycle and working on two codewords simultaneously. For this reason, it is expected that [8] should have twice the throughput as TW, at the same clock speed. And since [8]'s clock is roughly 1.5 times that of TW, one should expect that its throughput would be 3 *(2 x 1.5)* times greater. [8]'s throughput is actually 2.2 and 2.3 times that of TW for each decoder, respectively. This speaks well of the efficiency of the designs created by the Generator Framework. And the consistency in decoder performance between the two works serves as yet another verification of the system's correctness. Table 2 also shows a normalized comparison of throughput, using TW as a baseline, and adjusting the other works' throughput measurements with respect to clock frequency and pipelining.

Next, a comparison is drawn between TW and [9]. It regards only one size decoder, corresponding to the larger of the two decoders evaluated in [8]. The decoder implemented in [9] uses hybrid-2 message passing, which accounts for the 48.2% fewer flip flops and the 36.2% fewer lookup tables. The reduced message quantization degrades decoding performance by +0.8 dB at 10e-6 BER, and +2.6 average iterations, with respect to TW . This closely matches the results claimed in [8] with their experimentation with different quantization schemes. A comparison of throughput, normalized for clock frequency, places TW slightly ahead of [9], with a difference of 0.5 Gbps. This is likely due to [9] requiring more iterations to complete execution sequences. While a direct comparison cannot be made between TW and [9] due to quantization variations, the differences in measured decoder response are as expected.

5.3.1 Hybrid-2 Attempt

Hybrid-2 message quantization was attempted so that a direct comparison with [9] would be possible. Hybrid-*N* for *N*>1 is not implemented in the Generator Framework. A generated decoder was modified by hand for hybrid-2. It is expected that decoder performance would decrease, given the reduced quantization. But the observed decrease was substantial enough to imply that it was not operating as expected. It is likely this is due to the method of implementation of the hybrid feature. In the hybrid-1 implemented decoders, nodes compute four bit messages to pass on, but disregard the least significant bit of the messages they receive, allowing the synthesis tools to optimize away the associated wires. This way, computational code in the node source files require no modification. Verified empirically, this is a sufficient method for hybrid-1 quantization. But since the factor nodes of the M-S algorithm use a minimum function in their computation, too much relevant data may be thrown away with hybrid-2. The nodes implemented in [8] and [9] implemented this feature differently, by computing two-bit messages to send.

5.4 Optimizations

An interesting optimization was discovered during the development of this application. As decoders of increasing size were attempted, the implementation tools began to have difficulty routing the designs. This was in large part due to each node requiring two globally routed signals in addition to the clock: *graph_rst*, and either *var_en* or *fac_en*. `BUFG` primitives helped somewhat, but decoders of size N>500 still had much difficulty routing. An effort was made to reduce or eliminate as many of the control signals as possible. Since the variable nodes do not internally store their a priori data and it can be reset externally, the variable nodes themselves do not require a reset. After a graph reset, the variable nodes should only output their a priori data as output messages. This requires that the factor nodes be initialized during a graph reset so that they output an identity message, either *0* for M-S algorithm, or *1.0* for S-P algorithm. So the factor nodes do require a reset signal. Variable nodes and factor nodes must alternate their computation, in a ping-pong fashion; however, they can accomplish this without their enable signals. Instead of waiting for the opposite nodes to compute their messages, all nodes compute messages at every clock. This results in variable nodes performing legitimate message calculations while the factor nodes perform junk message calculations, followed oppositely by factor nodes performing legitimate message calculations while the variable nodes perform junk calculations. Now no nodes require an enable signal. Only the termination module requires an enable signal, the factor enable. This is so the algorithm termination condition is only checked when the variable nodes are outputting legitimate a posteriori belief data. If we consider a (3,6)-regular decoder application, there are twice as many variable nodes as factor nodes. For an *N* sized decoder, this would require *3*N,* (*(N + N/2)*2*) global control signal fanouts. Reducing the resets to only factor nodes results in reduction of *N* global fanouts, and completely eliminating enable signals results in a reduction of *3/2*N* global fanouts. Altogether this results in a reduction of *5/6*N*, or over 83% of all globally routed signals, and permitted decoder designs up to N=1200 to be successfully implemented with reasonable routing times.

There are two drawbacks associated with this approach. The first is increased power consumption. Nodes are drawing power to perform wasteful computation on every other clock cycle. Also, this optimization now precludes the possibility of having more complex computation scheduling, such as implementing higher latency and resource sharing at both the graph level and the node level. Since these are future implementation objectives, the control signalling was not removed from the Generator Framework, and optimized designs simply comment out the reset and enable conditionals in the node implementations.

5.5 Supporting Software

Supporting software for this application is a mixture of custom, and third-party. Code generation is handled by LDPC-codes and MainPEG [17], [18]. Generator matrix calculation is conducted with Sage [20]. Custom software includes methods for handling message generation, codeword generation, symbol mapping, channel noise simulation, quantization and scaling, data file maintenance, codeword checking, and more. These are packaged within the linearcode_utils Python module. Automated data collection is handled by custom software and is packaged within the reporting_utils Python module.

5.6 Evaluation

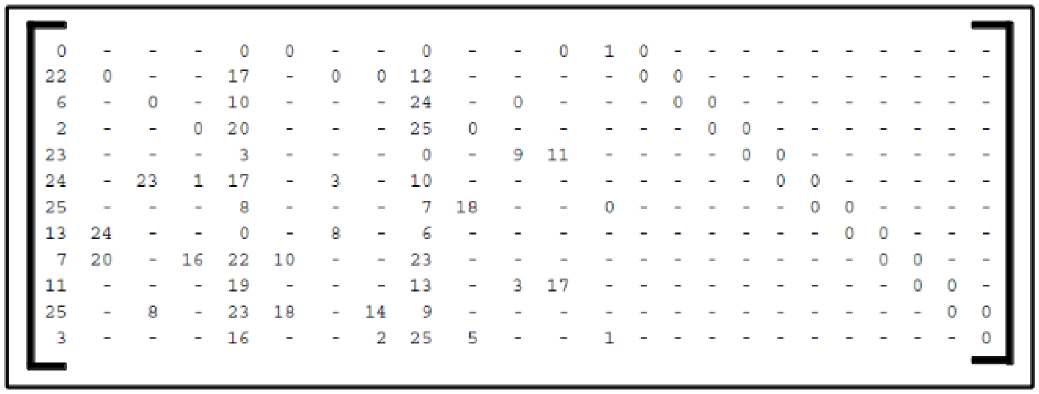
Application 1 has met each of its intended criteria. It served as a sort of scaffold on which to build and debug the Generator Framework. It permitted precise and accurate parameterized response testing of the designs. And it provided a basis against which to compare the similar works of others. This application has verified the correct operation and efficiency of the generated designs, and by implication, the correct operation and efficiency of the Generator Framework.

Chapter 6
Application 2: 802.11n Decoders

Testing lots of randomly generated LDPC codes has its merits. It enables a good verification of the system, and provides insight into its capabilities. But it doesn't allow for a strict check against the known behavior of a specific code. Nor does it tell if the system can handle real-world, commercially deployed codes. To answer these questions, the second application constructed with the Generator Framework is a subset of the 802.11n decoders.

6.1 802.11n LDPC Codes Explained

The IEEE 802.11n wireless networking protocol specifies twelve LDPC coding schemes [21]. All are irregular block circulant LDPC codes. Such codes are characterized by the structure of their check matrices. They contain rows and columns of concatenated, column-shifted identity matrices of a given size. A *matrix prototype* expresses this structure. Fig. 26 shows the matrix prototype of a 802.11n codeword, N=648, R=1/2 [21]. Each element in the matrix represents a square sub-matrix of dimension 27. A blank element represents a 27x27 zero matrix. A non-blank element *n* represents a 27x27 identity matrix that has been column-shifted right by *n* places. When elaborated, this matrix prototype represents a check matrix for the 648-bit code.



$$\begin{bmatrix}
0 & - & - & - & 0 & 0 & - & - & 0 & - & - & 0 & 1 & 0 & - & - & - & - & - & - & - & - & - & - \\
22 & 0 & - & - & 17 & - & 0 & 0 & 12 & - & - & - & - & 0 & 0 & - & - & - & - & - & - & - & - & - \\
6 & - & 0 & - & 10 & - & - & - & 24 & - & 0 & - & - & - & 0 & 0 & - & - & - & - & - & - & - & - \\
2 & - & - & 0 & 20 & - & - & - & 25 & 0 & - & - & - & - & - & 0 & 0 & - & - & - & - & - & - & - \\
23 & - & - & - & 3 & - & - & - & 0 & - & 9 & 11 & - & - & - & - & 0 & 0 & - & - & - & - & - & - \\
24 & - & 23 & 1 & 17 & - & 3 & - & 10 & - & - & - & - & - & - & - & - & 0 & 0 & - & - & - & - & - \\
25 & - & - & - & 8 & - & - & - & 7 & 18 & - & - & 0 & - & - & - & - & - & 0 & 0 & - & - & - & - \\
13 & 24 & - & - & 0 & - & 8 & - & 6 & - & - & - & - & - & - & - & - & - & - & 0 & 0 & - & - & - \\
7 & 20 & - & 16 & 22 & 10 & - & - & 23 & - & - & - & - & - & - & - & - & - & - & - & 0 & 0 & - & - \\
11 & - & - & - & 19 & - & - & - & 13 & - & 3 & 17 & - & - & - & - & - & - & - & - & - & 0 & 0 & - \\
25 & - & 8 & - & 23 & 18 & - & 14 & 9 & - & - & - & - & - & - & - & - & - & - & - & - & - & 0 & 0 \\
3 & - & - & - & 16 & - & - & 2 & 25 & 5 & - & - & 1 & - & - & - & - & - & - & - & - & - & - & 0
\end{bmatrix}$$

Fig 26. *Block Circulant Matrix Prototype (802.11n, 648x324)*

The twelve codes in the specification are combinations of three codeword lengths and four rates. The lengths are 648, 1296, and 1944. The rates are 1/2, 2/3, 3/4, and 5/6. For this application, all four rates of the 648-bit decoders were attempted.

6.2 Results

The Generator Framework readily produced designs for each decoder, though only three were able to be fully implemented by the Xilinx tools. The R=5/6 decoder failed to pass place-and-route, due to routing congestion issues. Fig. 27 shows a plot of *BER* vs. *SNR* for the three implemented decoders. The best performing code is indicated by the lowest *BER*. Each one of the three codes performs best within a certain region of noise. This is an expected characteristic of the codes. The codes in the 802.11 spec are engineered to have such properties so the best performing code in a given noise environment can be selected. From the plot, the R=1/2 code performs best when signal strength is below about 3.2 dB Eb/N0. The R=2/3 code performs best from 3.2 dB to 4.3 dB, and the R=3/4 code performs best when the signal strength is above 4.3 dB.

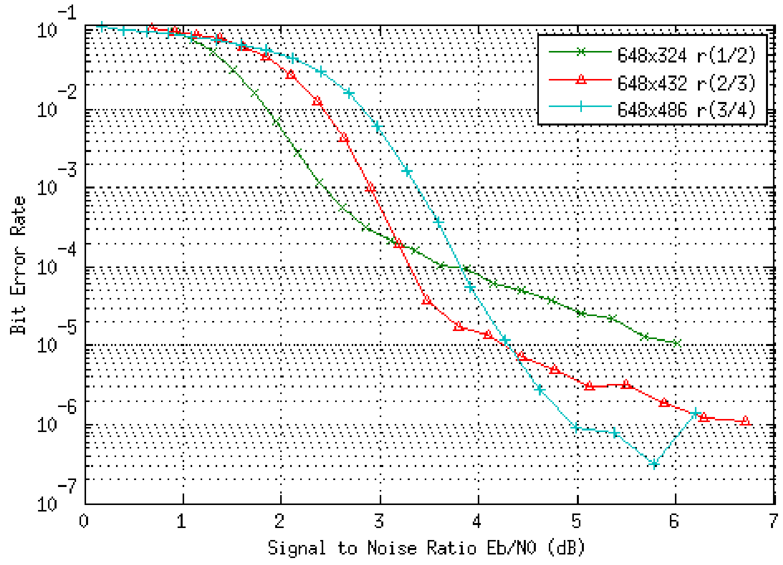


***Fig 27.** BER vs. SNR*

A decoder can easily keep track of the average number of iterations it takes to repair noisy codewords. Fig. 28 shows I_{avg} vs. *SNR*. Fig 29 shows a zoomed perspective. Here it is shown that in three different regions of *SNR*, each of the codes is a best performer in one of them. Rate R=1/2 performs best is below 3.18 dB, R=2/3 between 3.18 dB and 4.4 dB, and R=3/4 above 4.4 dB. This closely matches the *BER* results obtained. So, an 802.11n receiver may use I_{avg} as an indicator of what is the best code given a channel noise environment.

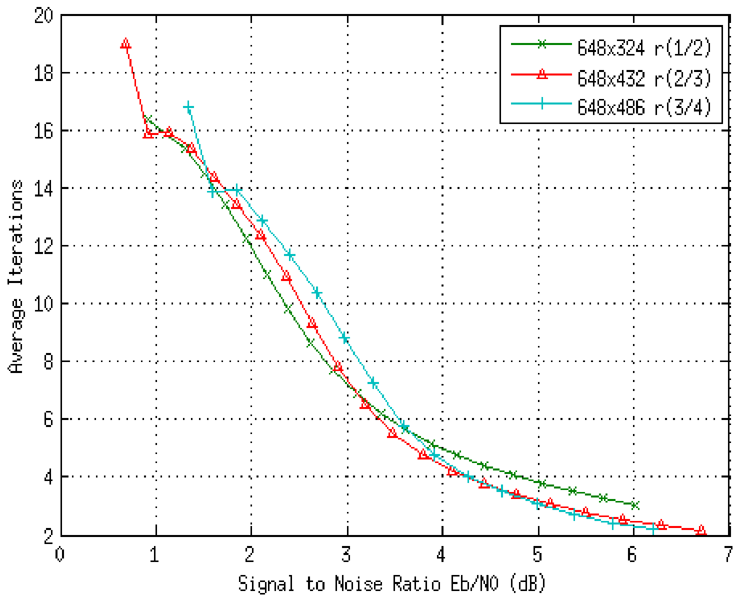


Fig 28. *I$_{avg}$ vs SNR*

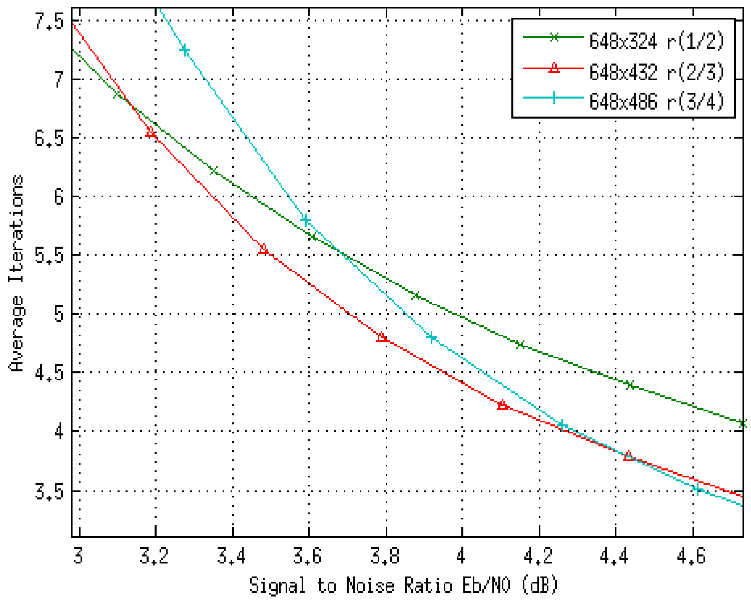


Fig 29. *Zoomed I$_{avg}$ vs SNR*

But for a communications protocol, what really is the best measure of performance is throughput. Fig. 30 shows Throughput vs. *SNR*. Fig. 31 shows a zoomed perspective. Again, there are three regions of *SNR* where one each of the three codes is uniquely best. Though here one of the regions is not quite the same. The R=1/2 code has highest throughput below 2.8 dB *SNR*, R=2/3 between 2.8 dB and 4.3 dB, and R=3/4 above 4.3 dB. The threshold defining the best code between R=1/2 and R=2/3 is somewhat different than noted before in the *BER* and I_{avg} plots. But still they are close. So while not perfect, I_{avg} is a good indicator of what code will return the best throughput for a given channel noise environment.

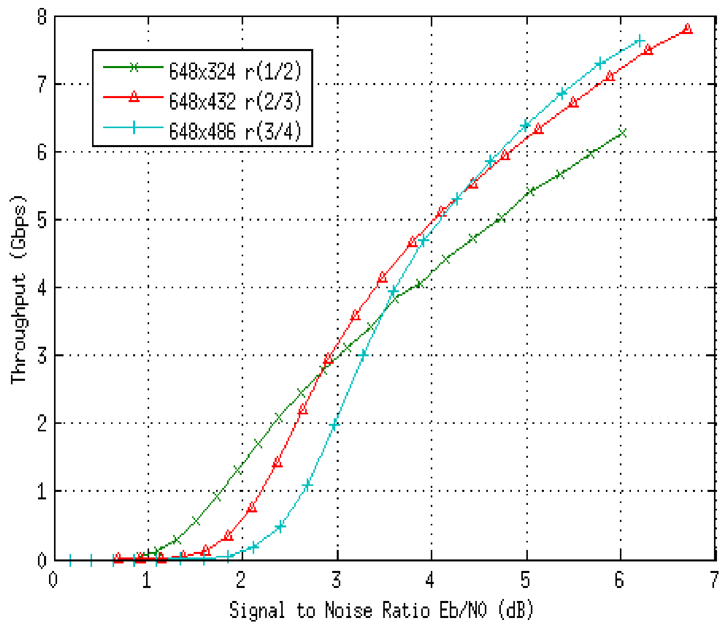


Fig 30. *Throughput vs SNR*

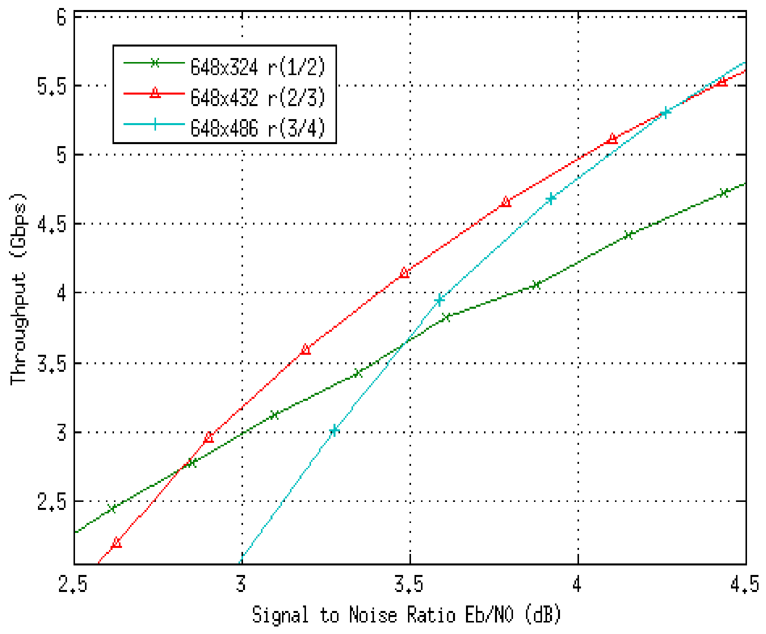


Fig 31. *Zoomed Throughput vs SNR*

6.3 Supporting Software

Graph definition files for this application were created using custom software. Methods to elaborate the matrix prototypes in the 802.11n spec are implemented in the linearcode_utils Python module.

6.4 Evaluation

This FG application has served to verify its two intended points. First, it shows that the designs returned by the Generator Framework have indeed performed just as 802.11n decoders are engineered to work. And secondly, it shows that the Generator Framework is capable of constructing complex designs for real-world applications.

Chapter 7
Application 3: Sudoku

The third and final application for this work was selected so as to be as dissimilar from the first two as possible. It purpose is to demonstrate the robustness of the system, and to show that it is not limited to generating forward error correcting decoders. A Sudoku solver was selected for this last application.

7.1 Sudoku as a PGM

Sudoku is a puzzle that can be represented as a PGM. Commonly, a Sudoku puzzle takes the form of a 9x9 board of squares. Fig. 32 shows an example of an initial board state.

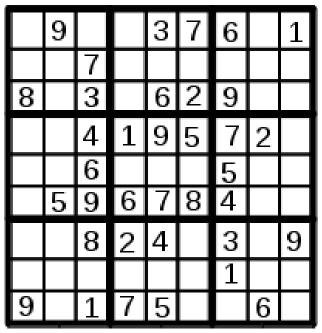


Fig 32. *Example 9x9 Sudoku Board*

Each row, column, and 3x3 subsection are constrained by a common rule: a number may not repeat. So two threes could not exist in the same row, and two fives could not exist in the same subsection, etc. This problem maps very well to a FG. Each square is represented by a variable node, and each constraint is represented by a factor node. Each variable node has three connections, since each square is constrained by row, column, and subsection. And each factor node has nine connections, since each constraint (row, column, or subsection) covers nine squares. Fig. 33 portrays a factor graph for a 9x9 Sudoku board.

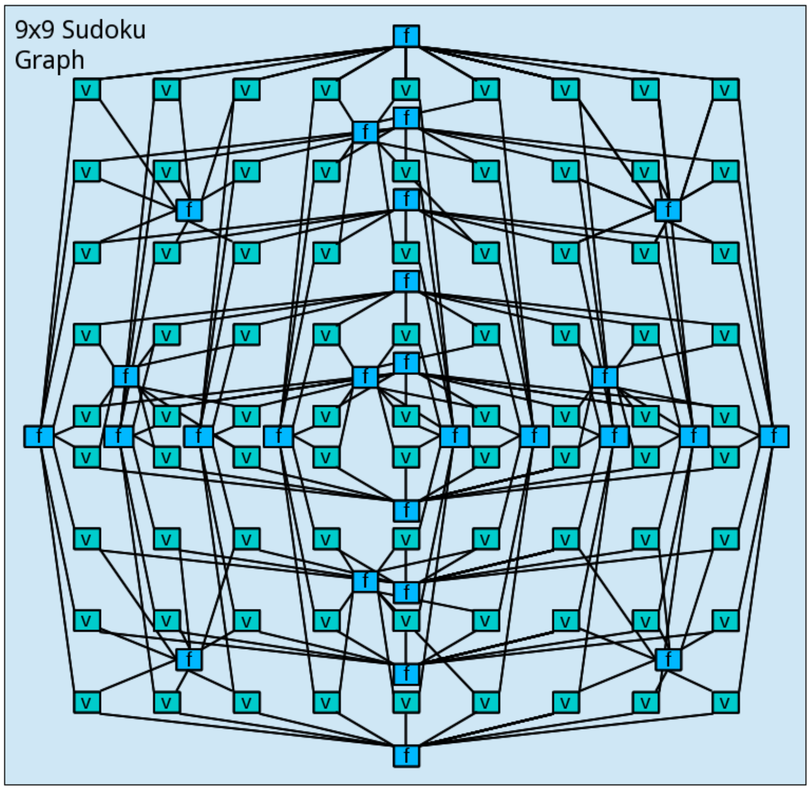


Fig 33. *Factor Graph for 9x9 Sudoku*

Each variable node stores a vector of probabilities, representing the chance of being each value in the domain. The first square in the upper left corner of the board in Fig. 7.1 has {9, 3, 7, 6, 1} in its row, {8, 9} in its column, and {9, 8, 7, 3} in its subsection. So the probability that square one is any of those values is zero. The probability that it is one of {2, 4, 5} is one. Given no other information, it is safe to assume an equal initial probability among the values in the possible set. Then the initial probability vector for square one is [0, .33, 0, .33, .33, 0, 0, 0, 0]. These vectors are calculated for each square and they represent the a priori data for the FG-Solver.

The FG in Fig. 33 contains 81 variable nodes and 27 factor nodes. Each variable node and factor node are 9-domain, since there are nine discrete values that a square may assume in the puzzle. That makes a total of 729 1-domain variable nodes and 243 1-domain factor nodes. Given that each of those nodes contains many floating point operations, and that each line in the graph in Fig. 7.2 represents 144 wires *(9-domain x 16 bits)*, the complexity and scale of this graph is well beyond what an FPGA can handle. For this reason, Application 3 is a generated solver for a 4x4 Sudoku board. Fig. 34 shows a factor graph for a 4x4 Sudoku.

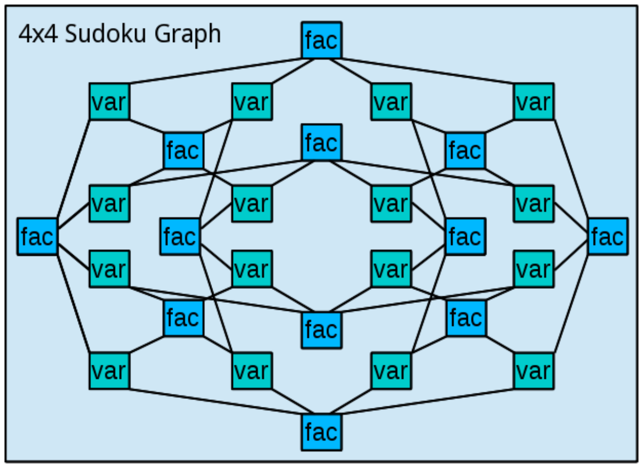


Fig 34. *Factor Graph for 4x4 Sudoku*

Here there are only 16 variable nodes and 12 factor nodes, and they are 4-domain. Connection lines in the graph represent only 48 wires *(4 x 16)*, and there are far fewer floating point operations needed in each node since there are far fewer connections per node. A 4x4 Sudoku board is a well scaled problem for FPGA implementation.

7.2 Results

The system does not have the capability of outputting intermediate message data, so we cannot see the algorithm work at the lowest level. But the system is capable of specifying the maximum number of iterations allowed. So by performing successive runs while step-wise increasing the maximum allowed iterations, we can see how the hard beliefs converge on a solution. Fig. 35 shows the hard beliefs of an execution sequence that solves in four iterations. The first board shows the initial state of the board, and each board following an arrow represents the beliefs after successive iterations.

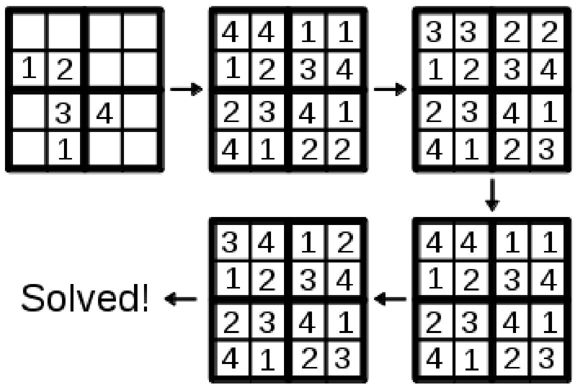


Fig 35. *Hard belief outputs for successful execution sequence*

7.3 Comparison With Other Work

The SumProductLab Matlab library includes a Sudoku solver among its example programs. The program was executed upon a starting board state identical to that of Fig. 35. Both SumProductLab and Application 3 of this work required four iterations to solve the Sudoku. The results are shown in Table 3. It is self evident that Application 3 is capable of solving Sudokus, but this comparison of required iterations serves to validate that it is properly executing the S-P algorithm. Also shown is the execution time required. Pitting hardware computation against software computation is never a fair comparison, but the magnitude of the speedup belies just how well suited FG solvers are for hardware. For this particular example, hardware is faster by a factor of over 450,000.

4x4 Sudoku (board shown above)	This Work	SumProductLab [7]
Number of Iterations	4	4
Execution Time	480 ns	0.218 s

Table 3. *Sudoku Solver Comparison*

7.4 Timeline of Tool Flow

Fig. 36 shows a timeline of the tool chain steps required to generate, implement, and execute the Application 3 design. It is drawn to scale, with the exception of the design generation and execution steps, which require too little time to be accurately drawn against the implementation times. The design setup time is approximate, since it requires by-hand coding, such as writing the graph definition file and the termination condition calculation. Total time is approximately 2387 seconds. Design generation consumes three one thousandths of one percent (0.0032%) of the total time.

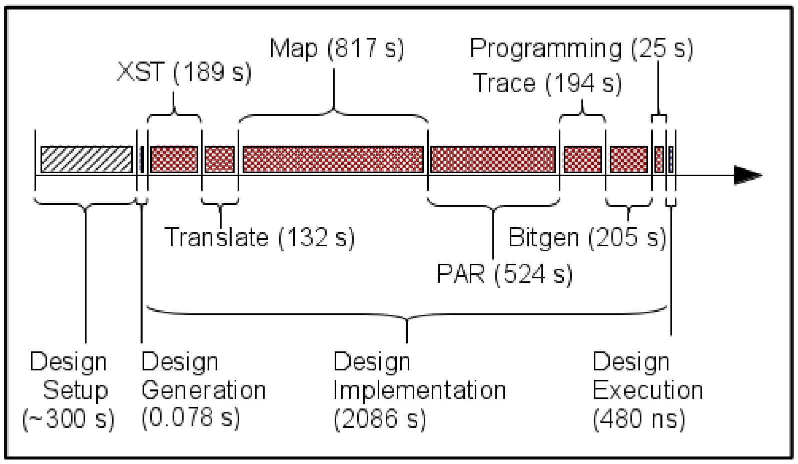


Fig 36. *Tool chain timeline*

7.5 Supporting Software

Supporting software is custom written as a Python module, sudoku_utils. It contains methods for calculating initial a priori data from a board state, and methods for constructing boards from returned beliefs. Methods from eth_utils are used for communication between host and device.

7.6 Evaluation

The purpose of this application was to demonstrate the breadth of problems that can be tackled by the designs constructed with the Generator Framework. This application has done this by solving a puzzle that is popular and understood by many, while being completely different and disjoint from the applications previously presented.

Chapter 8
Conclusion

The Generator Framework is capable of constructing a FG-Solver for any problem that can be modeled as a PGM, provided three criteria are met. The first relates to algorithm selection; the problem must be solvable by the S-P algorithm if a priori data is in probability domain, or the M-S algorithm if the a priori data is in log-likelihood domain. To satisfy the second criteria, the user must construct a custom factor node (if the current two generator scripts are insufficient), tailored to the desired algorithm and application. The third criteria is satisfied when the user modifies the termination module to suit their application requirements. Unimplemented, but currently possible applications could include audio or image noise reduction FG-Solvers.

8.1 Framework Limitations

The current state of the system has many limitations. Some are implementation specific, and some are inherent in the Generator Framework itself. Examples of implementation specific limitations would be an 802.11n LDPC N=1944 R=5/6 decoder, or a 16x16 Sudoku solver. The Generator Framework is capable of constructing designs for these problems, but their resource requirements would exceed that of any commercially available FPGA device. Limitations of the Generator Framework refer to what kind of problems it is incapable of solving. In the context of the factor graph structure, the Generator framework is currently incapable of creating designs that require additional node types beyond factor and variable nodes. An example of this would be a FG-Solver for a turbo product decoder. Its factor graph contains coefficient nodes that require preprocessing before algorithm execution. This introduces another limitation of the Generator Framework. It cannot support designs that require complex control signalling. Such designs would include those needing more node types than variable and factor, as well as designs whose nodes are partitioned into sets that each have unique control signal scheduling requirements. An example of the latter would be an FG-Solver for a Repeat-Accumulate decoder. The final major limitation of the Generator Framework is the lack of support for additional iterative algorithms such as Max-Product or Max-Sum. But algorithm specific computation takes place in the factor nodes, which require application specific tailoring. So the necessity for custom factor nodes for an application covers this limitation.

8.2 Improvements and Implications

Certain improvements to the Generator Framework could overcome many of these limitations. A modification to support pipelined, partially parallel message computation could reduce resource requirements, path lengths, and routing complexity at the expense of higher cycle latencies and more complex control signalling. Higher cycle latencies could be offset to a degree by an increase in clock speed due to reduced path lengths.

Support for user selectable latencies and hardware reuse, in a manner similar to Xilinx CoreGen IP configuration options, could allow for feasible implementation of much larger designs. Modifications to graph construction and control scheduling could support graph partitioning, opening the door for the generation of a wider variety of factor graphs, and also permitting partially parallel hardware reuse on a larger scale. Also, it is perhaps possible to support a single, generic factor node, deprecating the need for application specific custom factor nodes. Such an improvement could require that the user only supply a factor function, perhaps as a lookup table.

8.3 The Value of a FG Hardware Generator

The practical application of factor graphs is seemingly dominated by image processing and iterative decoding of forward error correcting codes. But they are also applied to a vast variety of other problems including software verification and bug-finding [22], machine learning and parameter estimation [23], and computer vision [24], to name a few. The broad scope of the problems being tackled by factor graphs and the fact that software suites exist for solver programming imply that there is a demand, even if only academic, for factor graph computation tools. An internet search suggests that no generic factor graph hardware computation tools are available. Still, hardware tools could play an important role. Iterative algorithms on factor graphs can be very computationally complex, and complexity can grow quickly with even a small increase problem size. This computational cost can be prohibitive for software, especially for problems that require real-time results. While lots of computation is required, much of it can be done in parallel, making factor graph applications well suited for hardware. A system for simply and quickly generating hardware designs for factor graph solvers could be useful and desirable.

Bibliography

[1] S. Hershey, J. Bernstein, B. Bradley, A. Schweitzer, N. Stein, T. Weber, and B. Vigoda, "Accelerating Inference: towards a full Language, Compiler and Hardware stack," *arXiv preprint arXiv:1212.2991*, 2012.

[2] S. Khan, S. Jabbari, S. Jabbari, and M. Ghanbarinejad, "Solving Sudoku Using Probabilistic Graphical Models."

[3] B. Andres, T. Beier, and J. H. Kappes, "OpenGM2," Apr-2014. [Online]. Available: http://hci.iwr.uni-heidelberg.de/opengm2/. [Accessed: 10-Apr-2014].

[4] B. Andres, J. H. Kappes, T. Beier, U. Kothe, and F. A. Hamprecht, "Probabilistic image segmentation with closedness constraints," in *Computer Vision (ICCV), 2011 IEEE International Conference on*, 2011, pp. 2611–2618.

[5] S. Gould, R. Fulton, and D. Koller, "Decomposing a scene into geometric and semantically consistent regions," in *Computer Vision, 2009 IEEE 12th International Conference on*, 2009, pp. 1–8.

[6] "dimple." [Online]. Available: http://dimple.probprog.org/home. [Accessed: 17-Apr-2014].

[7] "SumProductLab for Factor Graphs - File Exchange - MATLAB Central." [Online]. Available:http://www.mathworks.com/matlabcentral/fileexchange/26607-sumproductlab-for-factor-graphs. [Accessed: 17-Apr-2014].

[8] A. Balatsoukas-Stimming and A. Dollas, "FPGA-based design and implementation of a multi-GBPS LDPC decoder," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, 2012, pp. 262–269.

[9] V. A. Chandrasetty and S. M. Aziz, "An area efficient LDPC decoder using a reduced complexity min-sum algorithm," *Integration, the VLSI Journal*, vol. 45, no. 2, pp. 141–148, Mar. 2012.

[10] J. Cho, N. R. Shanbhag, and W. Sung, “Low-power implementation of a high-throughput LDPC decoder for IEEE 802.11 N standard,” in *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, 2009, pp. 040–045.

[11] A. Darabiha, A. C. Carusone, and F. R. Kschischang, “A bit-serial approximate min-sum LDPC decoder and FPGA implementation,” in *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, 2006, p. 4–pp.

[12] K. Van Der Bok, M. Taouil, P. Afratis, and I. Sourdis, “The TU Delft sudoku solver on FPGA,” in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, 2009, pp. 526–529.

[13] P. Malakonakis, M. Smerdis, E. Sotiriades, and A. Dollas, “An FPGA-Based Sudoku Solver based on Simulated Annealing Methods,” in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, 2009, pp. 522–525.

[14] H.-A. Loeliger, “An introduction to factor graphs,” *Signal Processing Magazine, IEEE*, vol. 21, no. 1, pp. 28–41, 2004.

[15] M. Peyiç, H. A. Baba, İ. Hamzaoğlu, and M. Keskinöz, “Low Power IEEE 802.11 n LDPC Decoder Hardware,” 2008.

[16] “Factor Graph,” *Wikipedia*, Jan-2014. [Online]. Available: http://en.wikipedia.org/wiki/Factor_graph.

[17] R. M. Neal, “Software for Low Density Parity Check Codes,” 2012. [Online]. Available: http://www.cs.utoronto.ca/~radford/ftp/LDPC-2012-02-11/index.html.

[18] X. Hu, E. Eletheriou, and D. Arnold, “Progressive edge-growth (PEG) algorithm MainPEG.” [Online]. Available: http://www.pudn.com/downloads71/sourcecode/book/...detail257395.html.

[19] S. J. Johnson, *Iterative error correction: turbo, low-density parity-check and repeat-accumulate codes*. Cambridge, UK ; New York: Cambridge University Press, 2010.

[20] “Sage Mathematical Software System.” [Online]. Available: www.sagemath.org.

[21] International Organization for Standardization, International Electrotechnical Commission, and Institute of Electrical and Electronics Engineers, *Information technology telecommunications and information exchange between systems-- local and metropolitan area networks-- specific requirements. Part 11, Part 11,*. Geneva; New York: ISO : IEC ; Institute of Electrical and Electronics Engineers, 2012.

[22] T. Kremenek, A. Y. Ng, and D. R. Engler, “A Factor Graph Model for Software Bug Finding.,” in *IJCAI*, 2007, pp. 2510–2516.

[23] K. Rohanimanesh, M. Wick, S. Singh, and A. McCallum, “MAP inference in Large Factor Graphs with Reinforcement Learning,” 2008.

[24] S. Z. Li, *Markov random field modeling in computer vision*. New York: Springer-Verlag, 1995.