A Comparative Evaluation of High-Level Hardware Synthesis Using Reed–Solomon Decoder

Abhinav Agarwal, Man Cheuk Ng, and Arvind

***Abstract*—Using the example of a Reed–Solomon decoder, we provide insights into what type of hardware structures are needed to be generated to achieve specific performance targets. Due to the presence of run-time dependencies, sometimes it is not clear how the C code can be restructured so that a synthesis tool can infer the desired hardware structure. Such hardware structures are easy to express in an HDL. We present an implementation in Bluespec, a high-level HDL, and show a 7.8× improvement in performance while using only 0.45× area of a C-based implementation.**

***Index Terms*—Bluespec, C-based design, case study, high-level synthesis.**

I. Introduction

DSP community perceives several advantages in using a C-based design methodology [14], [6]—having a concise source code allows faster design and simulation, technology-dependent physical design is isolated from the source and using an untimed design description allows high-level exploration by raising the level of abstraction. Several EDA vendors provide tools for this purpose [11], [15], [3], [5], [9]. In this letter, we use a specific example to explore to what degree a particular performance target can be achieved using such tools.

C-based tools fall into two distinct categories—those that adhere to pure C/C++ semantics like Catapult-C [11], PICO [15] and C-to-Silicon Compiler [3], and those that deviate from the pure sequential semantics by allowing new constructs, like SpecC [5], SystemC [13] and BachC [9], (see [4] for a detailed discussion of this topic). In this study, we used a popular C-based tool that synthesizes hardware directly from standard C/C++ and allows annotations and user specified settings for greater customization. Such annotations are most effective in those parts of the source code that have static loop bounds and statically determinable data dependencies. In this letter, we give examples where it is essential to exploit parallelism, the extent of which depends on run-time parameters. It is difficult for the user to restructure some of these source codes to allow the C-based tool to infer the desired hardware structure. These hardware structures can be designed using any HDL; we used a high-level HDL, Bluespec SystemVerilog [2], which makes it easy to express the necessary architectural elements to achieve the desired performance.

Manuscript received May 25, 2010; revised June 23, 2010; accepted June 23, 2010. Date of publication June 28, 2010; date of current version September 17, 2010. This work was supported by Nokia sponsored research at CSAIL, MIT. This manuscript was recommended for publication by R. Gupta.

The authors are with the Computer Science and Artificial Intelligence Laboratory (CSAIL), Massachusetts Institute of Technology, Cambridge, MA 02139 USA (e-mail: abhiag@mit.edu; mcn02@mit.edu; arvind@mit.edu).

Digital Object Identifier 10.1109/LES.2010.2055231

II. The Application: Reed–Solomon Decoder

Reed–Solomon codes [12] are a class of error correction codes frequently used in wireless protocols. In this letter, we present the design of a Reed–Solomon decoder for an 802.16 protocol receiver [8]. The target operating frequency for the FPGA implementation of our designs was set to 100 MHz. To achieve the 802.16 target throughput of 134.4 Mbps at this frequency, the design needs to accept a new 255 byte input block every 1520 cycles. During the design process, our goal was also to see if the number of cycles can be reduced even further because the "extra performance" can be used to decrease voltage or frequency for low power implementations.

A. Decoding Process

Reed–Solomon decoding algorithm [16] consists of five steps:

1) syndrome computation by evaluating the received polynomial at various roots of the underlying Galois Field (GF) primitive polynomial;
2) error locator polynomial and error evaluator polynomial computation through the Berlekamp–Massey algorithm using the syndrome;
3) error location computation using Chien search which gives the roots of the error locator polynomial;
4) error magnitude computation using Forney's algorithm;
5) error correction by subtracting the computed errors from the received polynomial.

Each input block is decoded independently of other blocks. A Reed–Solomon encoded data block consists of k information symbols and $2t$ parity symbols for a total of $n\ (= k + 2t)$ symbols. The decoding process is able to correct a maximum of t errors.

III. Generating Hardware From C/C++

A. The Initial Design

The decoding algorithm was written in a subset of C++ used by the tool for compiling into hardware. Each stage of the Reed–Solomon decoder was represented by a separate function and a top-level function invokes these functions sequentially. The different functions share data using array pointers passed as arguments. High-level synthesis tools can automatically generate a finite state machine (FSM) associated with each C/C++ function once the target platform (Xilinx Virtex II FPGA) and the target frequency (100 MHz) has been specified. For our Reed–Solomon code, with n as 255 and t as a parameter with a maximum value of 16, the tool generated a hardware design that required 7.565 million cycles per input block, for the worst case error scenario. The high cycle count was due to the fact that the tool produced an FSM for each computation loop that exactly mimicked its sequential execution. We next discuss how we reduced this cycle count by three orders of magnitude.

B. Loop Unrolling to Increase Parallelism

C-based design tools exploit computational loops to extract fine-grain parallelism [7]. Loop unrolling can increase the amount of parallelism in a computation and data-dependency analysis within and across loops can show the opportunities for pipelined execution. For example, the algorithm for syndrome calculations consists of two nested for-loops. For a typical value of $t = 16$, the innerloop computes 32 syndromes sequentially. All of these can be computed in parallel if the innerloop is unfolded. Most C-based design tools can automatically identify the loops that can be unrolled. By adding annotations to the source code, the user can specify which of these identified loops need to be unrolled and how many times they should be unrolled. For unrolling, we first selected the for-loops corresponding to the Galois Field (GF) Multiplication, which is used extensively throughout the design. Next, the inner for-loop of Syndrome computation was unrolled. The inner for-loop of the Chien search was also unrolled. To perform unrolling we had to replace the dynamic parameters being used as loop bounds by their static upper bounds. These unrolling steps cumulatively lead to an improvement of two orders of magnitude in the throughput, achieving 19 020 cycles per input block. Still, this was only 7% of the target data throughput.

C. Expressing Producer–Consumer Relationships

To further improve the throughput, two consecutive stages in the decoder need to be able to exploit fine-grain producer-consumer parallelism. For example, once the Chien search module determines a particular error location, that location can be forwarded immediately to the Forney's algorithm module for computation of the error magnitude, without waiting for the rest of error locations. Such functions are naturally suited for pipelined implementations. But this idea is hard to express in sequential C source descriptions, and automatic detection of such opportunities is practically impossible.

For simple loop structures the compiler can infer that both the producer and consumer operate on data symbols in-order. It can use this information to process the data in a fine-grained manner, without waiting for the entire block to be available. Consider the code segment shown in Fig. 1. The C-based tool appropriately generates streaming hardware for this code in the form shown in Fig. 2. This hardware passes one byte at a time between the blocks to allow maximum overlapped execution of the producer and consumer processes for a single data block.

However, the presence of dynamic parameters in for-loop bounds can obfuscate the sharing of streamed data and makes it difficult to apply static dataflow optimizations [10]. For example, consider the code segment shown in Fig. 3, where the length of the intermediate array produced and the producer loop iterations which produce its values are dynamically determined based on the input.

The hardware generated by the C-based tool for this code is shown in Fig. 4. The compiler generates a large RAM for

```
void producer(char input[255],
              char intermediate[255])
{
  for (int i=0; i<255; i++)
    intermediate[i]=input[i]+i;
}
void consumer(char intermediate[255],
              char output[255])
{
  for (int i=0; i<255; i++)
    output[i]=intermediate[i]-i;
}
```

Fig. 1. Simple streaming example—source code.

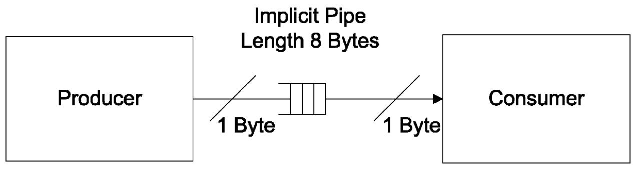

Fig. 2. Simple streaming example—hardware output.

```
void producer(char input[255], char length,
             char intermediate[255], char *count)
{
  *count = 0;
  for (int i=0; i<length; i++)
    if (input[i]==0)
      intermediate[(*count)++]=input[i]+i;
}
void consumer(char intermediate[255], char *count,
             char output[255])
{
  for (int i=0; i<*count; i++)
    output[i]=intermediate[i]-i;
}
```

Fig. 3. Complex streaming example—Source code.

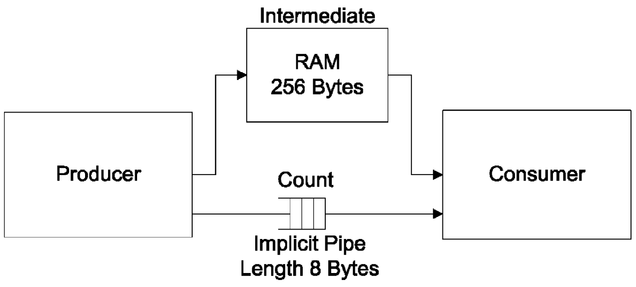

Fig. 4. Complex streaming example—Hardware output.

sharing one instance of the *intermediate* array between the modules. Furthermore, to ensure the program semantics, the compiler does not permit the two modules to access the array simultaneously, preventing overlapped execution of the two modules. It is conceivable that a clever compiler could detect that the production and consumption of data-elements is in order and then set up a pipelined producer-consumer structure properly. However, we expect such analysis for real codes to be quite difficult and brittle in practice.

Some C-based tools support an alternative buffering mechanism called ping-pong memory which uses a double buffering technique, to allow some overlapping execution, but at the cost of extra hardware resources. Using this type of double buffer, our design's throughput improved to 16 638 cycles per data block.

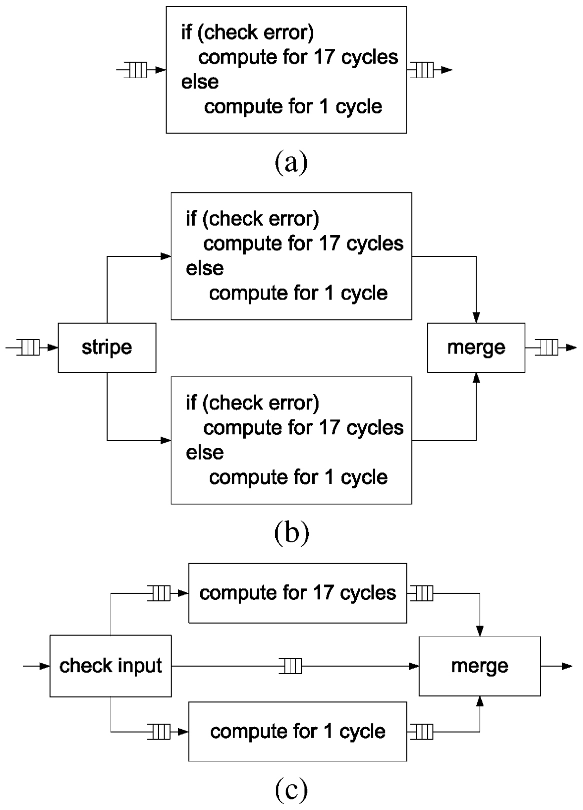

Fig. 5. Forney's algorithm implementation. (a) Original structure. (b) Unrolled structure. (c) Modified structure.

D. Issues in Streaming Conditionals

The compilers are generally unable to infer streaming architectures if the data blocks are accessed conditionally by the producer or consumer. For example, in Forney's algorithm the operation of the main for-loop is determined by a conditional check whether the location is in error or not. The input data structure contains k $(= 223)$ symbols out of which at most t $(= 16)$ symbols can be in error. Let us further assume it takes 17 cycles to process a symbol in error and only one to process a symbol not in error. The processing of symbols is independent of each other but the output stream must maintain the order of the input. If the compiler is unable to detect this independence, it will process these symbols in-order sequentially, taking as much as $17t + (k - t) = k + 16t = 479$ cycles [see Fig. 5(a)]. On the other hand, if the compiler can detect the independence of conditional loop iterations and we ask the tool to unroll it two times, we get the structure shown in Fig. 5(b). If the errors are distributed evenly, such a structure may double the throughput at the cost of doubling the hardware. The preferred structure for this computation is the pipeline shown in Fig. 5(c), which should take $\max(17t, k-t) = 272$ cycles to process all k symbols. Notice Fig. 5(c) takes considerably less area than Fig. 5(b) because it does not duplicate the error handling hardware.

The C-based tool was not able to generate the structure shown in Fig. 5(c). We think it will be difficult for any C-based synthesis tool to infer such a conditional structure. First, it is always difficult to detect if the iterations of an outer loop can be done in parallel. Second, static scheduling techniques rely on the fact that different branches take equal amount of time, while we are trying to exploit the imbalance in branches.

To further improve the performance and synthesis results, we made use of common guidelines [14] for code refinement. Adding hierarchy to Berlekamp computations and making its complex loop bounds static by removing the dynamic variables from the loop bounds, required algorithmic modifications to ensure data consistency. By doing so, we could unroll the Berlekamp module to obtain a throughput of 2073 cycles per block. However, as seen in Section V, even this design could only achieve 66.7% of the target throughput and the synthesized hardware required considerably more FPGA resources than the other designs.

E. Fine-Grained Processing

Further optimizations require expressing module functions in a fine-grained manner, i.e., operating on a symbol-by-symbol basis. This leads to considerable complexity as modules higher in hierarchy have to keep track of individual symbol accesses within a block. The modular design would need to be flattened completely, so that a global FSM can be made aware of fine-grained parallelism across the design. The abstractions provided by high-level sequential languages are at odds with these types of concurrent hardware structures and make it difficult for algorithm designers to express the intended structures in C/C++. Others have identified the same tension [4]. This is the reason for the inefficiency in generated hardware which we encountered during our study. The transaction granularity on which the functions operate is a tradeoff between performance and implementation effort. Coarse-grain interfaces where each function call processes a complete array is easier for software programmers but fine-grain interface gives the C compiler a better chance to exploit fine-grained parallelism.

IV. Implementation in Bluespec

Bluespec encourages the designer to consider the decoding algorithm in terms of concurrently operating modules, each corresponding to one major functional block. Modules communicate with each other through bounded first-in–first-outs (FIFOs) as shown in Fig. 6. Each module's interface simply consists of methods to enqueue and dequeue data with underlying Bluespec semantics taking care of control logic for handling full and empty FIFOs. It is straightforward to encode desired architectural mechanisms and perform design exploration to search for an optimal hardware configuration. Bluespec supports polymorphism, which allows expression of parameterized module interfaces to vary granularity of data communication between modules. The pipeline in Fig. 6 is latency insensitive in the sense that its functional correctness does not depend upon the size of FIFOs or the number of cycles each module takes to produce an output or consume an input. This provides great flexibility in tuning any module for better performance without affecting the correctness of the whole pipeline.

A. Initial Design

In Bluespec design, one instantiates the state elements, e.g., registers, memories, and FIFOs, and describes the behavior using atomic rules which specify how the values of the state

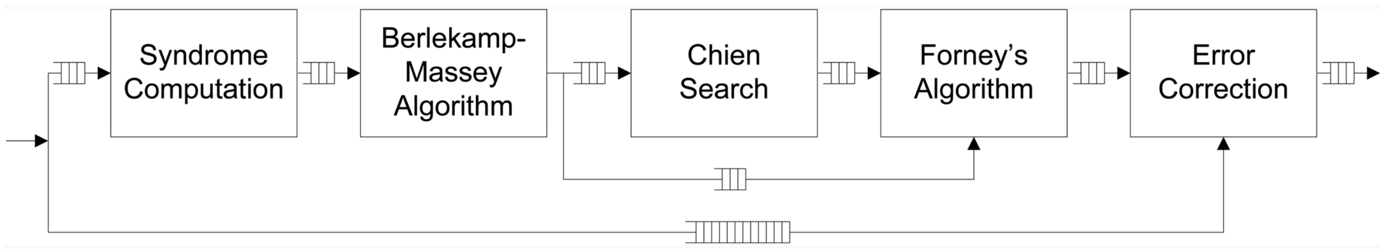

Fig. 6. Bluespec interface for the decoder.

```
rule compute_syndrome (True);
  let new_syn = syn;
  let product = gf_mult(new_syn[j],alpha(j+1));
  new_syn[j]  = gf_add(r_in_q.first(), product);
  if (j + 1 >= 2*t)
    j <= 0; r_in_q.deq();
    if (i + 1 == n)
      s_out_q.enq(new_syn);
      syn <= replicate(0);
      i   <= 0;
    else
      i <= i + 1;
  else
    syn <= new_syn;
    j   <= j + 1;
endrule
```

Fig. 7. Initial version of compute-syndrome rule.

```
rule compute_syndrome (True);
  let new_syn = syn;
  for (Byte p = 0; p < par; p = p + 1)
    let product  = gf_mult(in_q.first,alpha(i+p+1));
    new_syn[i+p] = gf_add(new_syn[i+p], product);
  if (j + par >= 2*t)
    j <= 0; in_q.deq();
    if (i + 1 == n)
      out_q.enq(new_syn);
      syn <= replicate(0);
      i   <= 0;
    else
      i <= i + 1;
  else
    syn <= new_syn;
    j   <= j + par;
endrule
```

Fig. 8. Parameterized parallel version of the compute-syndrome rule.

elements can be changed every cycle. The FSM, with its Muxes and control signals, is generated automatically by the compiler. For example, for Syndrome computation, the input and output of the module are buffered by two FIFOs, r_in_q and s_out_q and it has three registers: syn for storing the temporary value of the syndrome, and i and j for loop bookkeeping. The entire FSM is represented by a single rule called *compute_syndrome* in the module as shown in Fig. 7. This rule models the semantics of the two nested for-loops in the algorithm. The GF arithmetic operations, *gf_mult* and *gf_add*, and *alpha* are purely combinational library functions.

We implemented each of the five modules using this approach. This initial design had a throughput of 8161 cycles per data block. This was 17% of the target data throughput. It should be noted that even in this early implementation, computations in different modules can occur concurrently on different bytes of a single data block boosting the performance.

B. Design Refinements

Bluespec requires users to express explicitly the level of parallelism they want to achieve, which can be parameterized similar to the degree of loop unrolling in C-based tools. We illustrate this using the Syndrome Computation module. This module requires $2t$ GF Mults and $2t$ GF Adds per input symbol, which can be performed in parallel. Our initial implementation only performs one multiplication and one addition per cycle. By modifying the rule as shown in Fig. 8, the module can complete **par** iterations per cycle. The code is nearly identical to the original with the modifications highlighted in bold. The only change is the addition of a user specified static variable **par** which controls the number of multiplications and additions the design executes per cycle.

We unrolled the computations of the other modules using this technique, which allowed the design to process a block every 483 cycles. At this point, the design throughput was already 315% of the target performance. It was possible to boost the performance even further by using some of the insight into algorithmic structures discussed in Section III. For example, at this point in the design cycle we found that the Forney's algorithm module was the bottleneck, which could be resolved by using a split conditional streaming structure shown in Fig. 5(c). This structure can be described in BSV using individual rules triggering independently for each of the steps shown as a box in Fig. 5(c). This design allowed the Forney's Algorithm module to process an input block every 272 cycles. The sizes of FIFO buffers in the system also have a large impact on the overall system throughput and area. It is trivial to adjust the sizes of the FIFOs with the BSV library. Exploration of various sizes through testbench simulations allowed fine-tuning of the overall system to get a system throughput of 276 cycles per input block, which was 5.5× of the target throughput, as seen in Section V.

V. Results

At the end of the design process, the RTL outputs of C and Bluespec design flows were used to obtain performance and hardware synthesis metrics for comparison. Both the RTL designs were synthesized for Xilinx Virtex-II Pro FPGA using Xilinx ISE v8.2.03i. The Xilinx IP core for Reed Solomon decoder, v5.1 [17], was used for comparison. The designs were simulated to obtain performance metrics. Fig. 9 summarizes the results. The C-based design achieved only 23% of the Xilinx IP's data rate while using 201% of the latter's equivalent gate count, while the Bluespec design achieved 178% of the IP's data rate with 90% of its equivalent gate count.

Design	**C-based tool**	**Bluespec**	**Xilinx**
Lines of Source Code	1046	1759	—*
LUTs	29549	5863	2067
FFs	8324	3162	1386
Block RAMs	5	3	4
Equivalent Gate Count	596, 730	267, 741	297, 409
Frequency (MHz)	91.2	108.5	145.3
Throughput (Cyc/Blk)	2073	276	660
Data rate (Mbps)	89.7	701.3	392.8

Fig. 9. Comparison of source code size, FPGA resources and performance (*source code was not available for the Xilinx IP).

VI. Conclusion

Using the Reed–Solomon decoder as an example, we have shown that even for DSP algorithms with relatively simple modular structure, architectural issues dominate in determining the quality of hardware generated. Identifying the right microarchitecture requires exploring the design space, i.e., a design needs to be tuned after we have the first working design. Examples of design explorations include pipelining at the right level of granularity, splitting streaming conditionals to exploit computationally unbalanced branches, sizing of buffers and caches, and associated caching policies. The desired hardware structures can always be expressed in an HDL like Verilog, but it takes considerable effort to do design exploration. HDLs like Bluespec bring many advantages of software languages in the hardware domain by providing high-level language abstractions for handling intricate controls and allowing design exploration through parameterization.

C-based design flow offers many advantages for algorithmic designs—the designer works in a familiar language and often starts with an executable specification. The C-based synthesis tools can synthesize good hardware when the source code is analyzable for parallelism and resource demands. The compiler's ability to infer appropriate dataflow and parallelism, and the granularity of communication, decreases as the data-dependent control behavior in the program increases. It is difficult for the user to remove all such dynamic control parameters from the algorithm, as seen in the case of Forney's algorithm, and this leads to inefficient hardware. For our case study, we were not able to go beyond 66.7% of the target performance with the C-based tool. It is not clear to us if even a complete reworking of the algorithm would have yielded the target performance. The source codes and the transformations are available [1] for the interested readers.

Acknowledgment

The authors are thankful to J. Hicks and G. Raghavan at Nokia for their assistance.

References

[1] A. Agarwal and M. C. Ng, Reed-Solomon Decoder [Online]. Available: www.opencores.org/project,reedsolomon

[2] Bluespec, Inc., Bluespec SystemVerilog Language [Online]. Available: www.bluespec.com

[3] Cadence, C-to-Silicon Compiler [Online]. Available: www.cadence.com

[4] S. A. Edwards, "Challenges of synthesizing hardware from C-like languages," *IEEE Design Test Comput.*, vol. 23, no. 5, May 2006.

[5] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*. Norwood, MA: Kluwer, 2000.

[6] Y. Guo, D. McCain, J. R. Cavallaro, and A. Takach, "Rapid prototyping and SoC design of 3G/4G wireless systems using an HLS methodology," *EURASIP J. Embed. Syst.*, vol. 2006, no. 1, pp. 18–18, 2006.

[7] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Loop shifting and compaction for the high-level synthesis of designs with complex control flow," in *Proc. Design, Autom. Test. Eur.*, Paris, France, 2004.

[8] IEEE Standard 802.16. Air Interface for Fixed Broadband Wireless Access Systems IEEE, 2004.

[9] T. Kambe, A. Yamada, K. Nishida, K. Okada, M. Ohnishi, A. Kay, P. Boca, V. Zammit, and T. Nomura, "A C-based synthesis system, Bach, and its application," in *Proc. Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Yokohama, Japan, 2001.

[10] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, pp. 24–35, 1987.

[11] Mentor Graphics, Catapult-C [Online]. Available: www.mentor.com

[12] T. K. Moon, *Error Correction Coding-Mathematical Methods and Algorithms*. New York: Wiley-Interscience, 2005.

[13] SystemC Language Open SystemC Initiative [Online]. Available: www.systemc.org

[14] G. Stitt, F. Vahid, and W. Najjar, "A code refinement methodology for performance-improved synthesis from C," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD'06)*, San Jose, CA, 2006.

[15] Synfora, PICO Platform [Online]. Available: www.synfora.com

[16] S. B. Wicker and V. Bhargava, *Reed-Solomon Codes and Their Applications*. New York: IEEE, 1994.

[17] Xilinx, CORE Reed Solomon Decoder IP v5.1 [Online]. Available: www.xilinx.com/ipcenter/coregen/coregen_iplist_71i_ip2.htm