BEHAVIOURAL SYNTHESIS OF AN ADAPTIVE VITERBI DECODER

M. Zwolinski, J. S. Reeve

School of Electronics and Computer Science, University of Southampton, UK
{mz, jsr}@ecs.soton.ac.uk

Keywords: High-level synthesis; Viterbi decoder; BCH decoder; VHDL; FPGA.

Abstract

The synthesis of a hardware implementation of a Viterbi decoder from a behavioural specification is discussed. This is applied to a parallelized version of a BCH decoder. A parameterizable high-level VHDL model of the parallel decoder has been developed. Scalability of the parallel decoder in hardware is demonstrated. An extension of this technique to an adaptive decoder is discussed.

1 Introduction

Hardware implementations of DSP functions offer many advantages over software approaches. Hardware is implicitly parallel, therefore allowing much greater throughput for a given clock frequency and significant opportunities for reducing the overall power consumption. Hardware implementations are often thought of as being less flexible, but with the increasing capacity of FPGAs, it is now possible to build "soft hardware" – in other words, hardware that can be reconfigured to perform new functions as the need arises.

While DSP processors are usually programmed in a subset of C, dedicated DSP hardware is often designed at a much lower level of abstraction. Although it is common to derive algorithms using tools such as Matlab, the implementation itself is done using a hardware description language (VHDL or Verilog) at register transfer level (RTL) or below. RTL is broadly analogous to assembly language programming – there has been no equivalent to a C-level language for DSP design. From the designer's point of view, this is a significant disadvantage. It is extremely difficult to determine the optimal structure of a system in terms of area, speed and power. Indeed, it may be impossible to satisfy all the implementation constraints, thus an engineering compromise is required. If a single RTL design takes weeks or even months, it is clear that alternative structures cannot be readily explored.

Behavioural hardware synthesis seeks to bridge the gap between algorithms and RTL. The effects of compromises between conflicting objectives can be explored in a matter of minutes. Thus many alternative implementations can be examined rapidly. For this work, we have used a behavioural synthesis tool, MOODS [5], developed within the University over a period of some 15 years. MOODS takes as its input a high-level, algorithmic description of a hardware system, written in a subset of VHDL. Using a global optimisation algorithm (simulated annealing) and a set of weighted constraints, MOODS can explore the design space and generate a near-optimal implementation at RTL, suitable for further refinement.

MOODS was designed as a general purpose synthesis tool. One of the objectives of this study was to examine its suitability as a DSP design tool. At the same time, the Viterbi decoding algorithm [4] has been recast in a form suitable for implementation in a parallel processing environment [3]. In other words, the amount of inter-processor communications is minimised. A second objective was to determine if this recast algorithm could be written in such a way in VHDL as to produce an efficient, highly parallel hardware implementation. By mapping each *processor* onto a VHDL *process*, the coarse parallel structure would be maintained. In principle, the algorithm is parameterizable for different codes and for different numbers of processors/processes. The third objective was, therefore, to design a parameterizable version of the synthesisable VHDL, thus allowing configurations with different decoding schemes and different numbers of processes. Our final objective was to consider how different decoding schemes might be dynamically swapped onto an FPGA fabric and to determine how the multiple objective optimisation in terms of area, speed etc., might be extended to include the decoding scheme.

In this paper, we report how the Viterbi algorithm was mapped onto an FPGA structure, maintaining the parallel processing structure. Limitations in the implementation of the VHDL compiler in MOODS became apparent, but these did not significantly impede the proof of concept. Similarly, a generic VHDL version of the algorithm can be written. We will present our achievements to date in implementing a dynamically reconfigurable version of the algorithm on an FPGA.

2 Parallel Viterbi decoder

The parallel version of the Viterbi decoding algorithm has been described in detail elsewhere [3]. Here, we will simply summarise the approach. We have implemented the decoding of Bose-Chaudhuri-Hocquenghem (BCH) codes [1, 2], but this approach could equally be applied to convolution codes. Equally, we implement hard decision decoding, but again this approach can be adapted to soft decision decoding.

The Viterbi algorithm is simple, but it has high memory requirements. In order to efficiently parallelise the algorithm, the memory should be evenly distributed between the

processing elements. Similarly, for an efficient parallel implementation, the connectivity between processing elements should be restricted.

BCH codes are a generalisation of Hamming code that allows multiple error correction. They are class of cyclic codes that append n-k parity bits to a message of k bits so that each code word is n bits long. The code parameters (n, k, d_{min}) are of the form $n = 2^m - 1$, $n - k < mt$ and the minimum Hamming distance is $d_{min} < 2t + 1$. (m and t are integers.) The codes are specified by their generator polynomials which have the general form $g_0 + g_1D + \ldots. g_{n-k}D^{n-k}$. The encoding process is usually described in terms of a shift register.

In our implementation of the Viterbi decoder, the shift register encoder is regarded as a state machine in which the state, R, is a base 10 number that represents the bit pattern $\{r_0r_1\ldots r_{n-k-1}\}$ and the number of possible states is 2^{n-k} [3].

In the following description, G is the generator polynomial represented as a binary number, $Q = 2^{n-k-1}$, and $g = G \oplus Q$ is the generator with the top bits of G and Q set to 0. The machine changes to particular state depending on the top bit of the register, that is on whether R is greater or less than Q and on the value of the next input bit of the message.

In the encoding process, if the register is in state R_i after the ith message bit has been input, then we have $R^{i+1} = 2R^i$ if either $R^i<Q$ and the message bit is 0, or $R>Q$ and the message bit is 1. Conversely, $R^{i+1} = (2R^i) \oplus g$ if either $R^i<Q$ and the message bit 1, or $R>Q$ and the message bit is 0. Note, the current states are always modelled as $2R$ and $(2R^i) \oplus g$ in either decoding or encoding. The state transition template is shown in Figure 1.

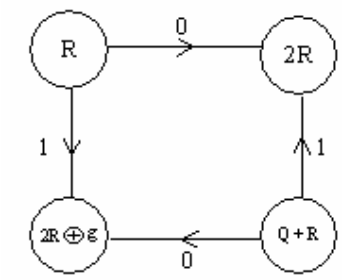

Figure 1. The building block for the state transition diagram for all cyclic codes

In the decoding process, if the current state is $R^i = 2R^i$ and the previous state was R^i-1 = R^i, the input ith bit must be 0. If the ith bit in the input sequence is 1, there is an error in the input sequence. If the current state is $R^i = (2R^i) \oplus g$ and the previous state was R^i-1 = $R^i + Q$, the ith input bit must be 0. If it is 1, there is an error in the input sequence.

The same rules apply for $R^i = 2R^i$, $R^{i-1} = R^i + Q$ and $R^i = (2R^i) \oplus g$, $R^{i-1} = R^i$. This is what we use in modelling our decoder. We compare the current state with the previous state and calculate which bit should be in the sequence. The state transition diagram for the BCH (7,4,3) which has a generator G=013 in octal is shown in Figure 2.

The parallelisation strategy is to distribute the states evenly among the available processing elements. The states are partitioned among 2^m processors, so that there are $P = 2^{n-k-m}$ states on each processor. $Q = 2^{n-k-1}$, so the number of states less than Q on each processor is $2^{n-m-k-1}$.

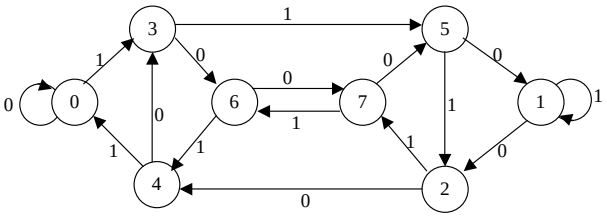

Figure 2 The state transition diagram for BCH (7,4,3) code with $G = 013_8$

States R and $R+Q$ are required to update states $2R$ and $(2R) \oplus g$, so by placing states R and $R+Q$ on the same processor we need to send only one message (containing the Hamming weights and paths from states R and $R+Q$)

3 Behavioural Synthesis

The objective of behavioural synthesis is to map algorithms onto hardware. As yet, behavioural synthesis tools are not widely used. Most automatic hardware synthesis is done at the register transfer level (RTL), at which there is a very clear relation between the structure of the description and the structure of the final hardware. Behavioural synthesis works from a higher level of abstraction, implying that there are many possible hardware structures. This gives the designer the freedom to think about the functionality of the design without worrying unduly about the structure. Many possible implementations can be quickly analysed. On the other hand, refinement from a more abstract level than RTL means that the final implementation may not be as good as that produced by an expert human designer. This situation is analogous to that which existed some years ago with software programming languages. Assembly language was perceived to be more efficient, but the immense productivity gains that result from using high-level languages means that assembly language programming is now limited to very specialized applications. The implicit parallelism of hardware means that behavioural synthesis is intrinsically more difficult than program compilation, but it can be expected that use of such synthesis tools will grow with time.

The MOODS synthesis tool [] has been a vehicle for research into behavioural synthesis for some 15 years at the University of Southampton. High-level specifications are written in a subset of VHDL. In this style of coding, the design may be divided into a number of concurrent processes, within which VHDL is written as a procedural programming language.

During the synthesis process, the control and data flows are extracted from each process. Thus each process is implemented as a state machine that controls various elements within the datapath. At first the implementation is very simplistic, with one data operation per control state and with each operation mapped to its own functional unit. As the synthesis progresses, the design is optimised by introducing concurrency and sharing operations between a smaller

number of functional units. There are about 20 simple transforms that can modify a part of the control and/or data paths. The optimisation is performed using the technique of simulated annealing, in which a transform and a part of the control/data structure are each randomly chosen. If the application of that transform to the given part of the structure would not change the functionality of the system, the transform is applied if it reduces the overall cost of the system. Typically, the cost is defined in terms of the speed and area of the implementation, but the cost can be extended to include factors such as power and testability. In this respect, MOODS is atypical of behavioural synthesis tools. The various cost constraints are decoupled, whereas in other approaches a complex, multi-dimensional optimisation has to be performed in one step.

Here, we only optimise in terms of speed and area. In this situation, we can use a heuristic, based upon simulated annealing, in which specific transforms are applied in a given order. Again, for simplicity, we also reduce the cost function weights to either high or low priority. (Thus only four combinations are possible: area is high priority, delay is low; area is low, delay is high; or both are high or both are low.)

No optimisation is attempted between processes. In the case study presented here, the system consists of a number of functionally identical, communicating processes. Therefore, we synthesise only one such process. The output from MOODS is RTL VHDL, which is passed on to a low-level synthesis tool. Thus we can intercept the output and create multiple instances of the single process by instantiating the RTL implementation, together with the interconnection framework within an RTL wrapper. (This same wrapper can also be used to instantiate multiple copies of the behavioural code, to allow the high-level description to be simulated.)

3.1 VHDL Coding Style

As noted, a behavioural subset of VHDL is used as the input language to MOODS. For this application, each processor is modelled as a single process within an entity/architecture. The code for each processor runs to about 200 lines and is therefore too long to reproduce here. The use of a VHDL process allows the use of procedural coding constructs such as for loops and if statements. The structure of the code is not, therefore, significantly different to what it would be if written in another high-level programming language. Indeed, the only HDL-specific construct used is a wait statement – needed only for behavioural simulation. We do use VHDL-specific data types, such as std_logic_vector, but only for convenience.

Although the processor code is parameterizable, restrictions in both the standard VHDL syntax and in our compiler mean that we use packages to define the BCH code and the number of processors. This is a minor inconvenience.

Finally, the RTL wrapper is also parameterised using values in the packages.

4 Results and Discussion

The function of the behavioural VHDL code was tested by simulation. Similarly, the functionality of the VHDL output generated by MOODS and of that generated following placement and routing was verified by simulation. It is therefore taken as read that all the simulation results cited below represent fully functional decoding operations. Because of the practical difficulties in interfacing an FPGA to other suitable hardware for input generation and output checking, we have restricted this study to simulations, but there is no reason to suppose that these results are anything but accurate. The results shown here are based on synthesis to Xilinx Virtex II FPGAs. In all cases, optimisation the area and delay optimisation priorities were both high and, where possible, and where possible, tri-state buffers were used to implement multiplexers.

Although the VHDL code for a single processor is the same in all cases, we would expect that more complex BCH codes would result in larger structures. (Note that a processor for a given BCH code remains the same, irrespective of the number of processors used.) Table 1 shows the MOODS *estimates* of the critical path delay in terms of cycles (not the decoding time!) and area for different BCH codes, confirming that the processor size and speed do grow as expected.

Code	Cycles	CLBs
(7,4,3)	55	653
(15,11,3)	158	903
(31,26,3)	430	1608
(63,57,3)	1146	3687

Table 1 Estimated speed/area of processor for different codewords

While simulation at the behavioural level can verify functional correctness, it can tell us nothing about execution speed. Post-synthesis, RTL, simulation will show how many clock cycles are needed to perform an operation. Table 2 shows the *decoding* time in clock cycles for different BCH codes with different numbers of processors.

	Number of processors					
Code	1	2	4	8	16	32
(7,4,3)	825	419	213			
(15,11,3)	3337	1739	921	513		
(31,26,3)	14349	7232	3683	1909	1023	
(63,57,3)	65355	32809	16537	8401	4333	23010

Table 2 RTL simulation of decoding time for different codeword (clock cycles)

These values are plotted on logarithmic axes in Figure 3. It can be seen that the execution time scales almost perfectly with the number of processors.
Finally, the estimates in Table 1, can be compared with the actual design size, following place and route. Table 3 shows the size for the (7,4,3) code. The number of CLBs is less than that predicted by MOODS – this is to be expected as MOODS does not perform low-level logic optimization. The hardware is capable of running with clock speeds of up to about 15MHz.

Processors	CLBs	TBUFs
1	459, 1%	1904, 5%
2	935, 2%	3812, 11%
4	1857, 5%	7632, 23%

Table 3 Place and Route statistics, (7,4,3) with tri-state buffers on Virtex3200

We have therefore fulfilled the first three of our objectives. We have used MOODS as a design tool for DSP hardware, although there are clearly deficiencies in the way that parameters are handled. We have implemented a parallel version of the Viterbi decoder that demonstrates scalability and we have written the VHDL in such a way that it can be used for different codes and different numbers of processors without modification. The fourth objective was to implement an adaptive decoder.
From Table 1, we can clearly see that with more complex codes, the processor size grows. Given that these are estimates, the hardware roughly doubles in size with each code. Thus we cannot simply move from one code to another by changing some parameters. Instead, we would have to dynamically reconfigure an FPGA. This is the basis of continuing work.

5 Conclusion

Overall, this work demonstrates that behavioural hardware synthesis is a suitable tool for DSP hardware design. It is apparent that the subset of VHDL chosen and the particular compiler implementation is not entirely appropriate for this particular domain. Nevertheless, this work demonstrates that the principle of multiple objective optimisation can significantly reduce the design cycle time and that this approach opens up a number of new research directions for further DSP-specific hardware optimisation.

Acknowledgements

The authors would like to thank the following students for their contributions to this work: Anne Godicheau, Andrew Basey, Myo Tun Aung, Kosala Amarasinghe, Simon Green.

References

[1] R.C. Bose and D.K. Ray-Chaudhuri, "On a class of error-correcting binary group codes", *Information and Control*, **3**, 68-79, (1960).
[2] A. Hocquenghem, "Codes correcteurs d'erreurs", *Chiffres (Paris)*, **2**, 147-156, (1959).
[3] J.S. Reeve, K. Amarasinghe, "A FPGA implementation of a parallel Viterbi decoder for block cyclic and convolution codes", *IEEE International Conference on Communications* **5**, 2596 – 2599, (2004).
[4] A.J. Viterbi, "Error bounds for convolution codes and an asymptotically optimum decoding algorithm", *IEEE Transactions on Information Theory*, **13**, 260-269, (1967).
[5] A.C. Williams, A.D. Brown and M. Zwolinski "Simultaneous optimisation of dynamic power, area and delay in behavioural synthesis", *IEE Proc. C&DT*, **147**, 383-90, (2000).

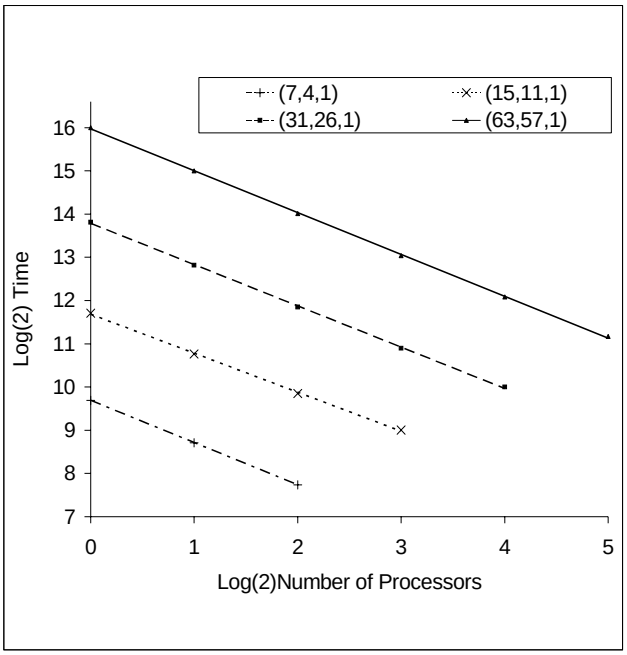

Figure 3 Number of processors versus execution time for 4 different codes.