Efficient Mapping of Mathematical Expressions into DSP Blocks

Bajaj Ronak, Suhaib A. Fahmy
School of Computer Engineering
Nanyang Technological University, Singapore
{ronak1,sfahmy}@ntu.edu.sg

***Abstract*—Mapping complex mathematical expressions to DSP blocks through standard inference from pipelined code is inefficient and results in significantly reduced throughput. In this paper, we demonstrate the benefit of considering the structure and pipeline arrangement of DSP blocks during mapping. We have developed a tool that can map mathematical expressions using RTL inference, through high level synthesis with Vivado HLS, and through a custom approach that incorporates DSP block structure. We can show that the proposed method results in circuits that run at around double the frequency of other methods, demonstrating that the structure of the DSP block must be considered when scheduling complex expressions.**

I. Introduction

FPGAs have always provided programmable logic and routing interconnect to support implementation of arbitrary circuits, however the cost of this flexibility has been that they are slow and consume significantly more power when compared to ASIC implementations. As FPGAs have gained wider adoption across application domains, vendors have sought to improve the efficiency of mapping often used functions. Hence, hard macro blocks have been introduced that implement these functions directly in silicon, thus consuming less area and power, and running at a higher clock speed that the equivalent function implemented in logic.

DSP blocks are an example of this trend. Since FPGAs are primarily used to accelerate complex computation through the use of custom datapaths, a key factor in overall performance is the efficiency of fundamental arithmetic operations. These capable blocks offer high throughput for arithmetic functions and flexibility to enable simple processors to be built [1].

While functions that fit in a single DSP block can be synthesised efficiently from pipelined RTL code, we have found that more complex functions requiring multiple DSP blocks suffer from lower performance [2]. A standard RTL description of a mathematical function can be heavily pipelined, for example after each operation, however, since this pipelining may not take into account the structure and internal stages of the DSP block, the resulting synthesised design may exhibit sub-standard performance since the DSP blocks are combined in a way that does not allow them to run at full speed.

In this paper, we compare the performance, in terms of area and speed, when mapping mathematical expressions through a standard RTL approach, high-level synthesis, and a method that considers the internal structure and pipelining of the DSP block. We focus on arithmetic dataflow graphs containing multiple addition/subtraction and multiplication operations. We have developed a tool that takes an input expression and generates synthesisable RTL code using standard pipelining techniques, along with a version based on DSP block structure, and finally a Vivado HLS implementation. Xilinx tools are then used to implement the variations and report the resulting resource requirements and frequency. We show that considering the internal structure and pipeline stages of the DSP block offers a throughput advantage of over 100% with a marginal change in resources, compared to a pipelined RTL or Vivado HLS implementation.

II. Related Work

As DSP blocks can offer increased performance when mapping applications, many signal and image processing and other algorithms have been optimised with implementations tailored to make use of the features of the DSP block.

Meanwhile, system design is increasingly being done at higher levels of abstraction. The main challenge here is that some optimisations made in the conversion to RTL may prevent efficient mapping to the hard macros available in the device. HLS tools may fully schedule the graph in a way that does not suit DSP blocks running at full speed.

FloPoCo [3] is an open-source tool which generates custom floating-point cores, outputting synthesisable VHDL, leveraging the DSP blocks and memories available on modern FPGAs. However, it only considers DSP blocks as fast multipliers, and does not consider the other sub-blocks, except insofar as the synthesis tools are able to infer them.

General mapping to hard blocks has been considered in various implementation flows. Verilog-to-Routing (VTR) [4] is an end-to-end tool which maps a Verilog description of a circuit to an FPGA architecture. Its front-end synthesis uses ODIN-II [5], which is optimised for some embedded blocks, but, complex embedded blocks are considered “black boxes”.

As of yet, we have not found tools that can automatically map to flexible, multi-function hard macro blocks in an efficient way. Standard pipelined RTL works for a simple datapath with just a single DSP block, but when more complex functions requiring multiple DSP blocks to be composed together are synthesised, the performance is generally poor with regard to the capabilities of the hard marco.

III. Design Techniques

There are three main stages in the DSP48E1 pipeline: a preadder, a multiplier, and an ALU stage. Specific configuration

inputs allow the dataflow through the DSP block to make use of whichever of these features are required. Expanding the set of possible configuration options results in a set of 29 different DSP Block datapath configurations.

Given an add-multiply datapath, a designer can implement it in a number of ways. A very simple approach is to elaborate the function in a combinational manner, add a number of pipeline stages at the end, and let the synthesis tool retime the circuit. While not an ideal approach, one might expect the synthesis tools to be able to do this correctly. An experienced designer, on the other hand, would likely write a pipelined RTL representation of the function. A common practice is to add a pipeline stage after each computational node, and balance the other branches of the pipeline to maintain alignment. In this scenario, the final schedule is explicitly determined by the designer, and the synthesis tools use this to infer DSP blocks. Another approach that has increased in popularity is to define the function in a high level language and rely on high-level synthesis tools to map this to a hardware implementation. Some tools offer the user a range of options for how deep to pipeline and how many resources to use.

None of these techniques, however, take into account the internal structure of the DSP block. In a combinational implementation with retiming, we might expect the synthesis tools to be able to do this, however, the results obtained show that retiming large functions fails to add enough pipeline stages. In a scheduled pipelined RTL implementation, the designer has already fixed the operation schedule, and so, only where parts of the graph map suitably to DSP blocks, will inference be efficient. Finally, high-level synthesis tools map to generic intermediate RTL that is similar in nature to that generated by an experienced designer.

Considering the internal structure of DSP blocks, we propose a technique for which the expression is first decomposed into portions that can be mapped to DSP blocks, and then the required pipelining applied, with consideration for the structure and pipeline of these individual blocks.

It is worth noting that the approach of direct instantiation of DSP blocks may not be a preferable method where the flow graph is only a part of a larger system. To overcome this, we also propose a method which is DSP architecture aware, but, implements the RTL equivalent of DSP blocks, instead of instantiating them directly.

The proposed tool we have developed takes an expression as input and generates the implementation methods discussed above automatically, allowing us to compare their performance in a systematic and fair way. We now discuss how these different techniques are implemented in our proposed tool.

A. Combinational Logic with Retiming: Comb

For *Comb*, all the nodes of the flow graph are implemented as combinational logic. We then add extra pipeline registers at the output node. We enable the register balancing feature in Xilinx ISE, allowing the tool to retime the design by moving the registers to intermediate stages.

B. Scheduled Pipelined RTL: Pipe

For *Pipe*, we create both As Soon As Possible (ASAP) and As Late As Possible (ALAP) scheduled variations.

C. High-level Synthesis: HLS

We use Vivado HLS from Xilinx for this approach, mainly because it is likely to be the most architecture aware of any of the high-level synthesis tools available. Each node is implemented as an expression, and *directives* are used to guide the RTL implementation to fully pipeline the design.

D. Direct DSP Block Instantiation: Inst

The tool decomposes an expression into DSP blocks. Subgraphs of the expression that match one of the 29 templates we have identified are extracted and for each, a DSP48E1 primitive is directly instantiated, with all the control signals set as required.

E. DSP Architecture Aware RTL: DSPRTL

Rather than instantiate the primitive, we replace each instance of a template with RTL code that directly reflects the template's structure. This variation will make it clear whether it is the instantiation of the primitives, or the structure of the graph that has a fundamental effect on the resulting circuit.

F. Ensuring Fair Comparison

There are a number of factors that could impact the fairness of comparison. First, the overall latency of an implementation may impact the resource requirements as extra registers are needed to balance the different paths. At the same time, we would not like to adversely affect the efficiency of standard approaches. For the Comb implementation, we add as many retiming pipeline stages after the combinational logic as are determined by the Inst method. For the Pipe methods however, we let the schedule determine the number of stages. In the HLS implementations, we fix the latency of the generated RTL implementation to the same as that of the Inst implementation, to give the tool sufficient slack to optimise the implementation.

Another factor that must be considered is wordlength. The output of a DSP block is wider than the inputs, and hence, paths from one DSP block to another should either be truncated, or the latter operations should be wider (likely using more than one DSP block). We choose to truncate, ensuring that the integer part is preserved and the least significant fractional bits are trimmed. This is equivalent to multi-stage fixed-point implementation, though we can optimise for known and fixed inputs. In the interests of fairness, we manage the wordlengths manually, and the tool ensures that the same wordlenghts are maintained across all implementations. This is necessary because various automated synthesis approaches may infer wider operators in intermediate stages, thus skewing the results.

In all the techniques, we enable the register balancing feature of Xilinx ISE for fair comparison.

IV. Experimentation Tool

We have developed a tool for generating synthesisable RTL implementations for all the methods described above. It functions as shown in Fig. 1.

The first two stages are common to all techniques, and the third stage is where the distinct aspects of each implementation

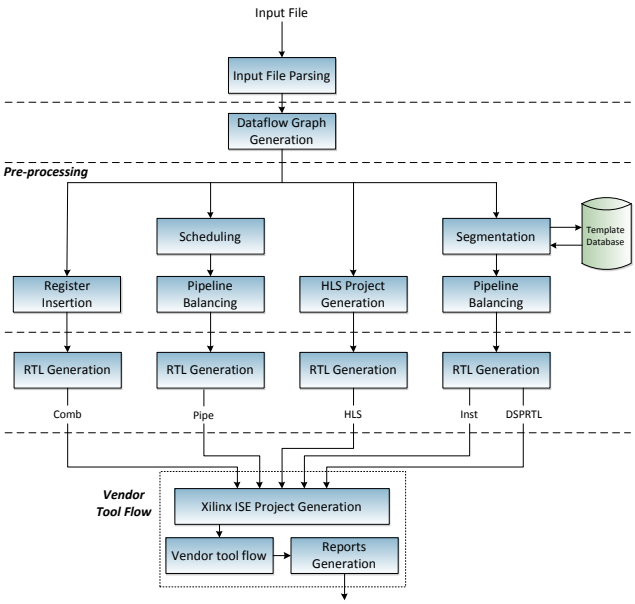

Fig. 1: Tool flow for exploring DSP block mapping.

are completed. RTL in Verilog HDL is generated for all the designs which then go through the same vendor implementation flow for final result generation.

Input File Parsing: The front-end of the tool accepts a text file listing the inputs of the expression and their wordlengths. For the purposes of functional verification, the input file can also contain a list of test cases. The input expression is written as a series of two-input operations. Power of 2 multiplications can be combined with other operations and implemented using shift.

Example: $16x^5 - 20x^3 + 5x \Rightarrow (x(4x^2(4x^2 - 5) + 5))$

Dataflow Graph Generation: A dataflow graph structure is generated from the parsed instructions, mapping each instruction as a node.

Pre-processing:

For *Comb*, we add extra registers at the output node that will be absorbed by the synthesis tool during retiming. For *Pipe*, As Soon As Possible (ASAP) and As Late As Possible (ALAP) schedules are determined for the graph, and the pipelines are correctly balanced to ensure alignment.

For *HLS*, the flow graph is implemented in C++ and all required files for Vivado HLS execution and testing are automatically generated.

For *DSPRTL* and *Inst*, the flow graph is partitioned into sub-graphs with each sub-graph mapping to one of the templates in the template database. Before segmentation, a *level* is assigned to each node according to the topological order of nodes. Segmentation is done in a greedy manner. As a DSP48E1 primitive can accommodate at most 3 nodes (corresponding to the three stages of the block), the algorithm selects a maximum of three connected nodes, does the template matching, and selects the template covering maximum number of nodes. The DSP48E1 hardware primitive is designed in such a way that intermediate outputs are not available, without bypassing subsequent sub-blocks. Hence, nodes with more than one output are always terminal when searching the database.

TABLE I: Graph nodes and pipeline length

Expr	Inputs	Adders/Subs	Muls	Stages Pipe/Others
Chebyshev	1	2	3	6/16
Mibench2	3	8	6	7/19
Quad Spline	7	4	13	7/25
SG Filter	2	6	6	8/19

When the pre-adder is used, all four pipeline stages should be enabled to achieve maximum performance. For other combinations, the same performance can be achieved with 3 pipeline stages. By default, the tool automatically selects the template with a suitable number of pipeline stages. The tool allows pre-selection of specific templates and pipeline depths to enable systematic analysis of the impact of these choices. After covering the complete dataflow graph with DSP48E1 primitives, input and output edges of the sub-graphs are mapped to appropriate ports of the DSP48E1 primitives and a tree of templates is generated. Paths through the graph are then balanced with sufficient registers.

RTL Generation: After *Pre-processing*, Verilog files implementing the datapaths for all the methods are generated. For *Comb*, pipeline registers equal to the number of stages in the Inst method are added at the end. For *HLS*, we execute the *script.tcl* file of the generated Vivado HLS project. For the *Inst* method, the tool generates instantiations of all the templates determined in the previous stage and also generates their respective Verilog files. *DSPRTL* is a variation of *Inst*, where instead of instantiating the DSP48E1 primitives directly, a pipelined representation of the individual template structures is used. Hence, this is general RTL but reflecting DSP block structure.

The wordlengths of the inputs of each node are explicitly set equal to those in *Inst*. We truncate the outputs of the DSP blocks at intermediate stages, according to the input ports of next DSP block they connect to.

Vendor Tool Flow: After generating RTL files for all the above methods, they are synthesised through the vendor tools. Since this can take time, we have automated the process through a series of scripts, which compile the results for further analysis.

V. Results

We implemented the Chebyshev polynomial, Mibench2 filter, Quadratic Spline, and Savitzky-Golay filter from [6]. We prepared the input files for all these expressions, and passed them through the tool flow.

Table I shows the number of operations (adder/subtractor and multipliers) for all expressions. It also shows the number of pipeline stages in each resulting implementation. Comb, HLS, Inst, and DSPRTL methods have same number of pipeline stages, derived from Inst method. The number of pipeline stages for Pipe depends on the schedule and is significantly less. This represents the approach taken by a skilled designer who does not consider DSP block structure.

All implementations target the Virtex 6 XC6VLX240T-1 FPGA as found on the ML605 development board, and use Xilinx ISE 14.6 and Xilinx Vivado HLS 2013.4.

To compare the resource usage between different methods, we also present the area usage in equivalent LUTs, where, $LUT_{eqv} = nLUT + nDSP \times (196)$, where 196 is the ratio of LUTs to DSP blocks on the target device, a proxy for area consumption.

Overall, we have observed that Comb exhibits the lowest throughput of all methods. For Pipe, we implement both ASAP and ALAP scheduling, and choose the one which gives higher frequency. Pipe shows much better performance compared to Comb, though the HLS approach is often better. Inst has the highest frequency in all the cases. Direct instantiation of DSP48E1 primitives is not ideal, as it leads to complex code, and prevents the circuit from being mapped to any other architecture.

Although, we expected DSPRTL to offer performance comparable to Inst, it initially fell short, performing close to HLS. One variation we explored is to add an extra pipeline stage at the output of each template in the RTL. This breaks the possibly long routing path between subsequent DSP blocks in the graph. We found that this significantly improved the throughput of DSPRTL, matching Inst (to within a few percent), but with the added flexibility of having general RTL code. Of course, this comes at a cost of increased register usage.

We explored mapping to templates with 4 pipeline stages, and templates with a mix of 3 and 4 stages, and the resulting performance difference was minimal ($\pm 5\%$). Hence we use a mix of 3 and 4 stage templates, as it may result in reduced latency without affecting throughput. One possible DSP block template is a 3-input adder, using the pre-adder and ALU blocks and bypassing the multiplier. We found frequency to be unaffected by whether 3-input adders are mapped to DSP blocks or in logic, so we use logic to preserve DSP blocks.

A comparison of resource usage and maximum frequency for all four expressions is shown in Table II. Combining frequency results, the geometric mean performance of Inst and DSPRTL implementations is 6.6× over Comb, 2.2× over Pipe, and 2× over the HLS implementation. This shows that considering the structure of the DSP block when pipelining a complex expression has a considerable impact on overall performance.

From the results shown, perhaps the key useful finding is that the DSP-based RTL implementation is in fact very close in performance to the implementation that instantiates DSP blocks explicitly. This suggests that the synthesis tools are good at inferring individual blocks, but can only do this when the structure has been considered in the way an expression has been scheduled. This is also beneficial, since it means techniques for scheduling can be applied within an HLS context, without the need for explicit instantiation.

TABLE II: Resource usage and max frequency for all designs

Expr	Method	DSPs	LUTs	Equv LUTs (DSPs+LUTs)	Regs	Max Freq (MHz)
Chebyshev	Comb	3	68	656	96	82
	Pipe	3	39	627	72	211
	HLS	3	218	806	326	223
	DSPRTL	3	80	668	131	473
	Inst	3	41	629	154	464
Mibench2	Comb	6	96	1272	104	75
	Pipe	6	177	1353	185	199
	HLS	4	771	1555	975	234
	DSPRTL	6	241	1417	571	473
	Inst	6	251	1427	596	469
Quad Spline	Comb	13	100	2648	115	58
	Pipe	13	171	2719	132	167
	HLS	12	1246	3598	1708	223
	DSPRTL	13	502	3050	818	470
	Inst	13	389	2937	845	455
SG Filter	Comb	6	72	1248	109	70
	Pipe	6	79	1255	79	274
	HLS	5	639	1619	599	226
	DSPRTL	6	158	1334	361	471
	Inst	6	136	1312	327	473

VI. Conclusion

We have developed a tool that implements mathematical expressions using DSP blocks. It allows us to compare the performance of multiple approaches to mapping graphs to DSP blocks. By considering the DSP48E1 primitive's structure and pipelining, we are able to map expressions in a manner that maximises frequency at a minimal cost of additional LUTs. The result is a doubling of performance over a well-pipelined RTL implementation that does not consider DSP block structure, or over HLS mapping.

The greedy approach to segmenting the graph, while sufficient to demonstrate this finding, can be improved. We aim to implement an improved segmenting algorithm that also considers accuracy in deciding which inputs to use for which paths, and can optionally leave some nodes in LUTs when this makes sense. We are also keen on extending this work to floating-point operations with the iterative approach demonstrated in [7]. We will then investigate mapping to a smaller number of DSP blocks with resource sharing using the dynamic control signals.

References

[1] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell, "iDEA: A DSP block based FPGA soft processor," in *Proceedings of the International Conference on Field Programmable Technology (FPT)*, 2012, pp. 151–158.

[2] B. Ronak and S. Fahmy, "Evaluating the efficiency of DSP Block synthesis inference from flow graphs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 727–730.

[3] F. De Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.

[4] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, "The VTR project: architecture and CAD for FPGAs from verilog to routing," in *Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2012, pp. 77–86.

[5] P. Jamieson, K. Kent, F. Gharibian, and L. Shannon, "Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research," in *Proceedings IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2010, pp. 149–156.

[6] S. Gopalakrishnan, P. Kalla, M. Meredith, and F. Enescu, "Finding linear building-blocks for RTL synthesis of polynomial datapaths with fixed-size bit-vectors," in *Proceedings of International Conference on Computer-Aided Design*, 2007, pp. 143–148.

[7] F. Brosser, H. Y. Cheah, and S. Fahmy, "Iterative floating point computation using FPGA DSP blocks," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2013.