Analysis and Optimization of a Deeply Pipelined FPGA Soft Processor

Hui Yan Cheah, Suhaib A. Fahmy, Nachiket Kapre
School of Computer Engineering
Nanyang Technological University, Singapore
Email: hycheah1@e.ntu.edu.sg

Abstract—**FPGA soft processors have been shown to achieve high frequency when designed around the specific capabilities of heterogenous resources on modern FPGAs. However, such performance comes at a cost of deep pipelines, which can result in a larger number of idle cycles when executing programs with long dependency chains in the instruction sequence. We perform a full design-space exploration of a DSP block based soft processor to examine the effect of pipeline depth on frequency, area, and program runtime, noting the significant number of NOPs required to resolve dependencies. We then explore the potential of a restricted data forwarding approach in improving runtime by significantly reducing NOP padding. The result is a processor that runs close to the fabric limit of 500MHz with a case for simple data forwarding.**

I. Introduction and Related Work

Processors are widely used within FPGA systems, from management of execution and interfacing, to implementation of iterative algorithms outside of the performance-critical datapath. When a soft processor is used in a sequential part of a computation [1], a low frequency would fail to deliver the high performance required to avoid the limits stated by Amdahl's law. Maximizing the performance of such soft processors requires us to consider the architecture of the FPGA in the design, and to leverage unique architectural capabilities wherever possible. Soft processors can be designed to be independent of architecture, and hence, portable, or to leverage architectural features of the underlying FPGA.

Commercial soft processors include the Xilinx MicroBlaze [2], Altera Nios II [3], ARM Cortex-M1 [4], and LatticeMico32 [5], in addition to the open-source LEON3. All of these processors have been designed to be flexible, extensible, and general, but suffer from not being fundamentally built around the FPGA architecture. The more generalised a core is, the less closely it fits the low-level target architecture, and hence, the less efficient its implementation in terms of area and speed. Consider the LEON 3 soft processor: implemented on a Virtex 6 FPGA with a fabric that can support operation at over 400MHz, it barely achieves a clock frequency of 100MHz [6].

Recent work on architecture-focused soft processors has resulted in a number of more promising alternatives. These processors are designed considering the core capabilities of the FPGA architecture and benefit from the performance and efficiency advantages of the hard macro blocks present in modern devices. These processors typically have long pipelines to achieve high frequency. Such long pipelines suffer from the need to pad dependent instructions to overcome data hazards as a result of the long pipeline latency.

Octavo [7] builds around an Altera RAM Block to develop a soft processor that can run at close to the maximum RAM Block frequency. It is a multi-threaded 10-cycle processor that can run at 550 MHz on a Stratix IV, representing the maximum frequency supported by Block RAMs in that architecture. iDEA [8], [9] makes use of the dynamic programmability the Xilinx DSP48E1 primitive to build a lean soft processor which achieves a frequency close to the fabric limits. In the case of iDEA, the long pipeline was shown to result in a large number of NOPs being required between dependent instructions. Octavo was designed as a multi-issue processor to overcome this issue, but as a result, is only suitable for highly parallel code.

In this paper, we conduct a detailed design space exploration to determine the optimal pipeline depth for the iDEA DSP Block based soft-processor [8]. We also make a case for a restricted forwarding scheme to overcome the dependency overhead due to the long pipeline.

II. Deep Pipelining in Soft-processors

iDEA is based on a classic 32-bit 5-stage load-store RISC architecture with instruction fetch, decode, execute, and memory stages followed by write-back to the register file. We tweak the pipeline by placing the memory stage in parallel with the execute stage to lower latency, effectively making this a 4-stage processor. Each stage of our processor can support a configurable number of pipeline registers. The minimum pipeline depth for each stage is one. As we increase the number of pipeline stages, clock frequency increases, until it plateaus, as later shown in Fig. 3. We explore all possible combinations of depths for the different stages, and pick, for each overall processor depth, the configuration that gives the highest frequency. To achieve maximum frequency using a primitive like the DSP block, it must have its multiple pipeline stages enabled. iDEA uses the DSP block as its execution unit and a Block RAM as the instruction and data memory, and as a result, we expect a long pipeline to be required to reach fabric frequency limits. By taking a fine-grained approach to pipelining the remaining logic, we can ensure that we balance delays to achieve high frequency. Since the pipeline stages in the DSP block are fixed, arranging registers in different parts of the pipeline can have a more pronounced impact on frequency.

While deep pipelining of the processor results in higher frequency, it also increases the dependency window for data hazards, hence requiring more NOPs for dependent instructions, as shown in Fig. 1 for a set of benchmarks. Fig. 2 shows pipeline depths of 7, 8 and 9 cycles, respectively, with

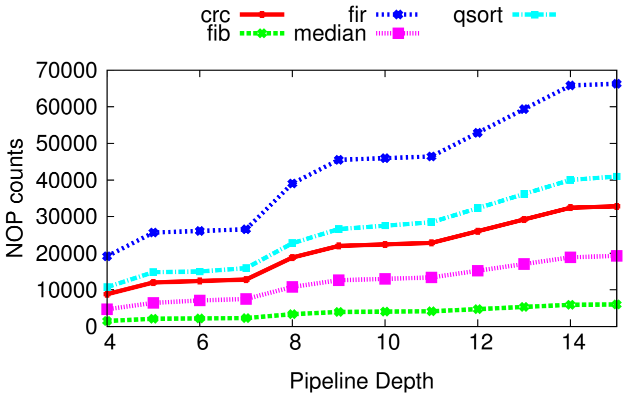

Fig. 1: NOP counts with increasing pipeline depth.

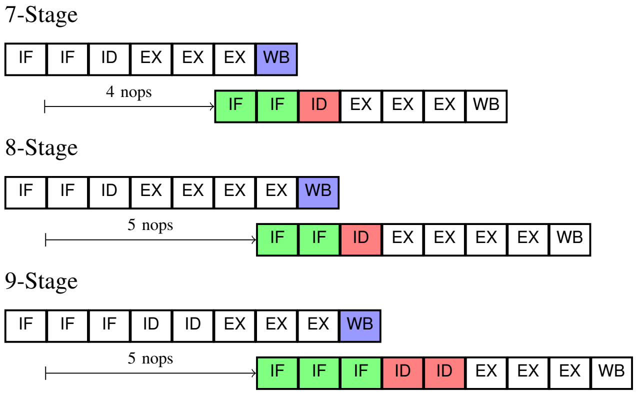

Fig. 2: Dependencies for pipeline depths of 7, 8 and 9 stages.

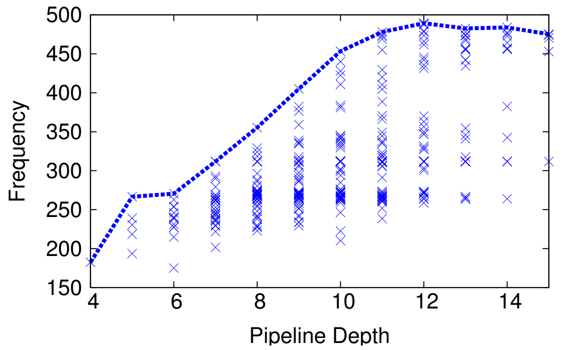

Fig. 3: Frequency of different pipeline combinations.

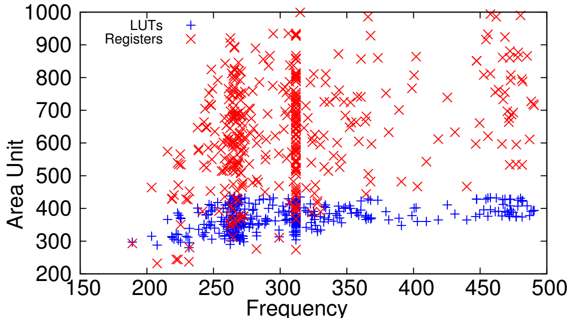

Fig. 4: Resource utilization of all pipeline combinations.

fetch, decode, execute and write back stages in each instruction pipeline.

To prevent a data hazard, an instruction dependent on the result of a previous instruction must wait until the computed data is written back to the register file before fetching operands. The second instruction can be fetched, but cannot move to the decode stage (in which operands are fetched), until the instruction on which it is dependent has written back its results. In the case of a 7-stage pipeline with the pipeline configuration shown, 4 NOPs are required between dependent instructions. Since there are many ways we can distribute processor pipeline cycles between the different stages, an increase in processor pipeline depth does not always mean more NOPs are needed. Consider the 8 and 9-stage configurations in Fig. 2. Since the extra stage in the 9 cycle configuration is an IF stage, that can be overlapped with a dependent instruction, no additional NOPs are required than for the given 8 cycle configuration. This explains why the lines in Fig. 1 do not increase uniformly. However, due to the longer dependency window, a longer pipeline depth with the same number of NOPs between consecutive dependent instructions may still have a slightly higher total instruction count.

III. Experiments

We implement the soft processor on a Xilinx Virtex-6 XC6VLX240T-2 FPGA (ML605 platform) using the Xilinx ISE 14.5 tools. We generate various processor combinations to support pipeline depths from 4–15. The pipeline depth is made variable through a parameterizable shift register at the output of each processor stage. During automated implementation runs in ISE, the shift register count is adjusted to increase the pipeline depth. We enable retiming and register balancing to exploit the extra registers in the datapath. Each pipeline configuration is synthesised, mapped, and placed and routed to obtain final area and speed results. Backend CAD implementation options are consistent throughout all experimental runs. We are able to test our processor on the FPGA using the PCI-based driver in [10].

We benchmark the instruction count performance of our processor using embedded C benchmarks. These benchmarks are compiled using the LLVM-MIPS compiler, and emitted assembly is then analysed for dependencies. A sliding window is used to ensure all dependencies are suitably addressed by inserting sufficiency NOPs as required for the specific pipeline configuration being tested.

A. *Area and Frequency Analysis*

Since the broad goal of our design is to maximize soft processor frequency while keeping the processor small, we perform a design space exploration to help pick the optimal combination of pipeline depths for the different stages. We vary the number of pipeline cycles from 1–5 for each stage: fetch, decode, and execute, and the resulting overall pipeline depth is 4–15 cycles, with writeback fixed at 1 cycle.

Fig. 3 shows the frequency achieved for varying pipeline depths between 4–15. Each overall depth can be built using varying combinations of stage depths, as we can distribute these registers in different parts of the 4-stage soft processor. The line shows the trend for the maximum frequency achieved

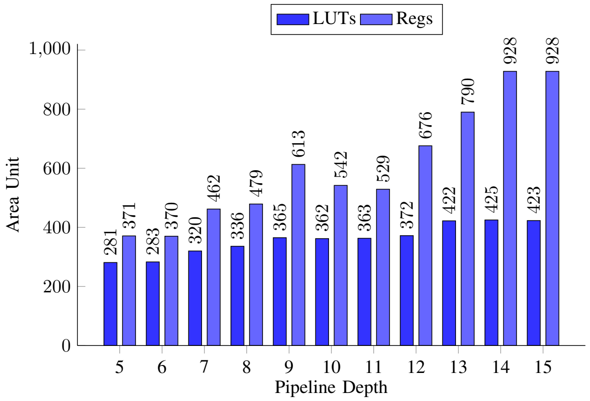

Fig. 5: Resource utilization of highest frequency configuration.

TABLE I: Optimal combination of stages and associated NOPs at each pipeline depth (WB = 1)

Pipeline Depth	IF	ID	EX	Req'd #NOPs
4	1	1	1	2
5	1	2	1	3
6	2	2	1	3
7	2	2	2	4
8	2	2	3	5
9	2	3	3	6
10	3	2	4	6
11	4	2	4	6
12	4	3	4	7
13	4	4	4	8
14	4	5	4	9
15	5	5	4	9

at each overall pipeline depth. The optimal combination of stages for each depth is presented in Table I.

Fig. 4 shows the distribution of LUT and register consumption for all implemented combinations by frequency achieved. Register consumption is generally higher than LUT consumption, and this becomes more pronounced in the higher frequency designs. A majority of the designs (58%) reach frequencies of between 250 MHz and 310 MHz, illustrated by denser regions in the plot. These represent configurations where the DSP block is at a depth threshold where it is in the critical path and requires further pipeline stages to be enabled to increase performance. From Fig. 3, we see that frequency increases considerably up to 11 stages, peaks at 12, and starts to decline slightly beyond that. This is expected as we approach the raw fabric limits around 500MHz.

B. Execution Analysis

Increasing processor frequency through pipelining is just one part of the performance equation. As we showed in Fig. 1, increases in pipeline depth also require increased padding between dependent instructions. Hence, we expect that the marginal frequency benefits of very long pipelines to be offset by increased NOPs in executed code, and hence a lower value of instructions per cycle (IPC). In Table I, we show the required number of NOPs between sequential dependent instructions to avoid data hazards.

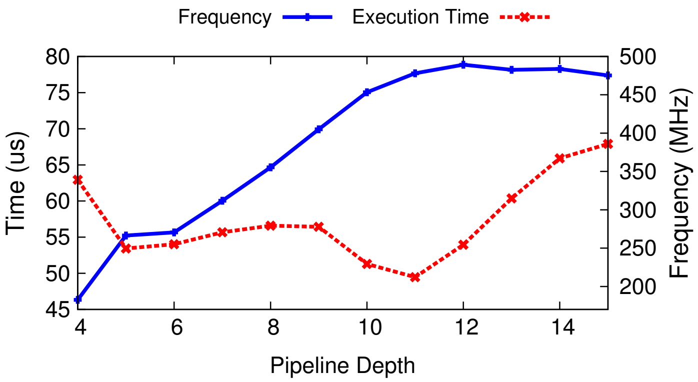

Fig. 6: Frequency and geomean wall-clock time at various pipeline depths.

Fig. 6 shows the normalised wall-clock times for the benchmarks we tested. As expected, wall-clock time decreases as we increase pipeline depth up to a certain point. For pipeline lengths of 5–9, there is a slight increase in wall-clock time as the benefits of frequency increases are offset by an increasing number of NOPs. From 9–11, the increase in NOPs is limited, allowing the frequency gains to result in a runtime benefit. From 11-stages onwards, NOPs begin to increase, while the frequency trend slows, resulting in execution time increasing again. From Fig. 6, the 11-cycle pipeline configuration gives the lowest execution time for this set of benchmarks.

IV. A Case for Simple Data Forwarding

Balancing the benefits of increased frequency with the NOP penalty of deeper pipelines is essential in determining the optimal pipeline depth for such processors. As we have seen, longer pipelines mean more idle cycles between instructions, and hence a low effective IPC. Analysis of the effect of data dependencies on the performance of in-order pipelines has been discussed in [11]. An optimal pipeline depth is derived based on balancing pipeline depth and achieved frequency, with the help of program trace statistics. A similar study for superscalar processors is presented in [12].

Data dependency of sequential instructions can be resolved statically in software or dynamically in hardware. Data forwarding paths can help reduce the padding requirements between dependent instructions, and these are present in modern processors. However, a full forwarding scheme typically allows forwarding between different stages of the pipeline, and so can be costly since additional multiplexed paths are required to facilitate this flexibility. With a longer pipeline, and more possible forwarding paths, such an approach becomes infeasible for a lean fast soft processor. Some schemes provide forwarding paths which must then be exploited in the assembly, while other dynamic approaches allow the processor to make these decisions on the fly. Tomasulo's algorithm, allows instructions to be executed out of order considering dependencies. However, implementing it in a soft processor would require significant additions in hardware, resulting in an area and frequency overhead that would be excessive for a small FPGA-based soft processor, and this overhead increases for deeper pipelines.

In our case, while dynamic forwarding, or even elaborate static forwarding would be too complex, a restricted

TABLE II: Dynamic cycle counts with 11-stage pipeline with % of NOPs savings.

Benchmark	Total NOPs	Consecutive Dependant NOPs	Reduced Consecutive Dependant NOPs	Reduced Total NOPs
crc	22,808	7,200 (32%)	2,400	18,008 (−21%)
fib	4,144	816 (20%)	272	3,600 (−13%)
fir	46,416	5,400 (12%)	1,800	42,816 (−8%)
median	13,390	1,212 (9%)	404	12,582 (−6%)
qsort	28,443	1,272 (4%)	424	27,595 (−3%)

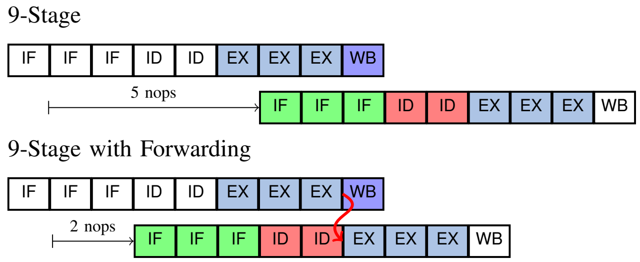

Fig. 7: Forwarding configurations, showing how subsequent instruction can commence earlier in the pipeline.

forwarding approach may be possible and could result in a significant overall performance improvement. Rather than add a forwarding path from every stage after the decode stage back to the execute stage inputs, we can consider just a single path. In Table II, we analyse the NOPs inserted in more detail. Out of all the NOPs, we can see that a significant proportion are between consecutive instructions with dependencies (4–30%). These could be overcome by adding a single path allowing the result of an instruction to be used as an operand in a subsequent instruction, avoiding the need for a writeback. We propose adding a single forwarding path between the output of the execute stage, and its input to allow this. Fig. 7 shows how the addition of this path in a 9-stage configuration would reduce the number of NOPs required before a subsequent dependent instruction to just 2, compared to 5 in the case of no forwarding.

In Table II, we show how the addition of this path reduces the number of NOPs required to resolve such consecutive dependencies, and hence the reduction in overall NOPs required. As this fixed forwarding path is only valid for subsequent dependencies, it does not eliminate NOPs entirely, and non-adjacent dependencies are still subject to the same window. However, we can see a significant reduction in the overall number of NOPs and hence, cycle count for execution of our benchmarks across a range of pipeline depths. These savings are shown in Fig. 8. We can see significant savings of between 4 and 30% for the different benchmarks. This depends on how often such chains of dependent instructions occur in the assembly and how often they are executed.

V. Conclusions and Future Work

We have demonstrated how to achieve an optimal pipeline depth for a DSP-based soft-processor. We have completed a full design space exploration in which we varied the overall pipeline depth, allowing for a combination of different depth configurations for each processor stage, determining the optimal configuration for each overall depth. We then investigated the resulting runtime for a set of benchmark programs, and determined that an 11-cycle configuration leads to the lowest execution time. We also conducted an initial study to explore the potential of a simple data forwarding approach for such lean soft processors, and determined that a fixed forwarding path can reduce instruction count by 4–30%. We aim to explore the effect of this path on the processor area and frequency and investigate other opportunities for forwarding in future work.

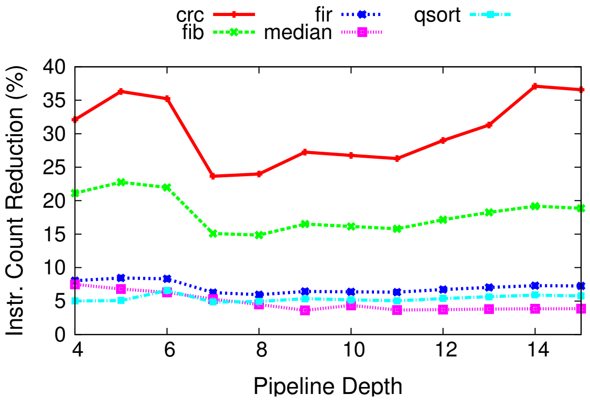

Fig. 8: Reduced instruction count with data forwarding.

References

[1] N. Kapre and A. DeHon, “VLIW-SCORE: Beyond C for sequential control of SPICE FPGA acceleration,” in *Proceedings of the International Conference on Field Programmable Technology (FPT)*, Dec. 2011, pp. 1–9.

[2] *UG081: MicroBlaze Processor Reference Guide*, Xilinx Inc., 2011.

[3] *Nios II Processor Design*, Altera Corpration, 2011.

[4] *Cortex-M1 Processor*, ARM Ltd., 2011. [Online]. Available: http://www.arm.com/products/processors/cortex-m/cortex-m1.php

[5] *LatticeMico32 Processor Reference Manual*, Lattice Semiconductor Corp., 2009.

[6] *GRLIB IP Library User’s Manual*, Aeroflex Gaisler, 2012.

[7] C. E. LaForest and J. G. Steffan, “Octavo: an FPGA-centric processor family,” in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, Feb. 2012, pp. 219–228.

[8] 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)*, Dec. 2012, pp. 151–158.

[9] H. Y. Cheah, F. Brosser, S. A. Fahmy, and D. L. Maskell, “The iDEA DSP Block Based Soft Processor for FPGAs,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 7, no. 3, pp. 19:1–19:23, 2014.

[10] K. Vipin, S. Shreejith, D. Gunasekara, S. A. Fahmy, and N. Kapre, “System-level FPGA device driver with high-level synthesis support,” in *Proceedings of the International Conference on Field Programmable Technology (FPT)*, Dec. 2013, pp. 128–135.

[11] P. G. Emma and E. S. Davidson, “Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance,” *IEEE Transactions on Computers*, vol. 36, pp. 859–875, 1987.

[12] A. Hartstein and T. R. Puzak, “The optimum pipeline depth for a microprocessor,” *ACM Sigarch Computer Architecture News*, vol. 30, pp. 7–13, 2002.