On Data Forwarding in Deeply Pipelined Soft Processors

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

ABSTRACT

We can design high-frequency soft-processors on FPGAs that exploit deep pipelining of DSP primitives, supported by selective data forwarding, to deliver up to 25% performance improvements across a range of benchmarks. Pipelined, in-order, scalar processors can be small and lightweight but suffer from a large number of idle cycles due to dependency chains in the instruction sequence. Data forwarding allows us to more deeply pipeline the processor stages while avoiding an associated increase in the NOP cycles between dependent instructions. Full forwarding can be prohibitively complex for a lean soft processor, so we explore two approaches: an external forwarding path around the DSP block execution unit in FPGA logic and using the intrinsic loopback path within the DSP block primitive. We show that internal loopback improves performance by 5% compared to external forwarding, and up to 25% over no data forwarding. The result is a processor that runs at a frequency close to the fabric limit of 500 MHz, but without the significant dependency overheads typical of such processors.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable Architecture*

Keywords

Field programmable gate arrays; soft processors; digital signal processing

1. INTRODUCTION

Processors find extensive use within FPGA systems, from management of system execution and interfacing, to implementation of iterative algorithms outside of the performance-critical datapath [13]. In recent work, soft processors have been demonstrated as a viable abstraction of hardware resources, allowing multi- processor systems to be built and programmed easily. To maximise the performance of such soft processors, it is important to consider the architecture of the FPGA in the design, and to leverage unique architectural capabilities wherever possible. Architecture-agnostic designs, while widespread and somewhat portable, typically suffer from sub-par performance. Consider the LEON3 [1] soft processor: implemented on a Virtex 6 FPGA with a fabric that can support operation at over 400 MHz, it barely achieves a clock frequency of 100 MHz. Such designs, while useful for auxiliary tasks, cannot be used for core computation. Even when processors do not constitute the core computation, their sequential operation represents a hard limit on overall system performance, as per Amdahl's law [3].

Recent work on architecture-focused soft processor design has resulted in a number of more promising alternatives. These processors are designed considering the core capabilities of FPGA architecture, often benefiting from the performance and efficiency advantages of the hard macro blocks present in modern devices. Using such primitives also results in a power advantage over equivalent functions implemented in LUTs. Octavo [14] is one such architecture that builds around an Altera Block RAM to develop a soft processor that can run at close to the maximum Block RAM frequency. iDEA [8] is another example that makes use of the dynamic programmability of FPGA DSP blocks to build a lean soft

Table 1: Assembly code comparison for $a \cdot x^2 + b \cdot x + c$ for a hypothetical 3-cycle processor. LOOPx operands indicate a loopback that does not need to be written back to the register file.

(a) Original Assembly	(b) Assembly with Loopback
li $1, x	li $1, x
li $2, a	li $2, a
li $3, b	li $3, b
li $4, c	li $4, c
mul $5, $1, $2	mul loop0, $1, $2
nop	mul $6, loop0, $1
nop	mul loop1, $1, $3
mul $6, $5, $1	add loop2, loop1, $6
mul $5, $1, $3	add $8, loop2, $4
nop	nop
nop	nop
add $7, $5, $6	sw $8, 0($y)
nop	
nop	
add $8, $7, $4	
nop	
nop	
sw $8, 0($y)	

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FPGA'15, February 22–24, 2015, Monterey, California, USA.
Copyright © ACM 978-1-4503-3315-3/15/02 ...$15.00.
http://dx.doi.org/10.1145/2684746.2689067.

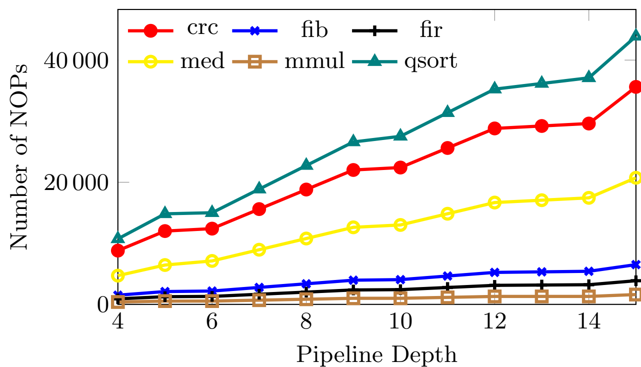

Figure 1: NOP counts as pipeline depth increases with no data forwarding.

processor which achieves a frequency close to the fabric limits of the DSP blocks found in Xilinx FPGAs. However, many hard blocks require deep pipelining to achieve maximum performance, and this results in long pipelines when they are used in a soft processor. Octavo and iDEA soft processor pipelines are 10 cycles deep to maximize operating frequency on the respective Altera and Xilinx devices. For a single-issue processor, deep execution pipelines result in significant penalties for dependent instructions, where NOPs have to be inserted to resolve dependencies, and can also increase jump penalties. In Figure 1, we show the rise in NOP counts for a deeply pipelined DSP block based soft processor, across a range of benchmark programs, as the pipeline depth is increased. These numbers are obtained as later described in Section 4.2, and demonstrate the significant penalty incurred at high pipeline depths.

In this paper, we explore how data forwarding can be added to the iDEA soft processor. Being based on the Xilinx DSP48E1, iDEA can be deployed across all Xilinx Artix-7, Kintex-7, Virtex-7, and Zynq device families. It is also easily portable to the DSP48E2 primitive found the next generation Xilinx UltraScale architecture. Since we do not have access to internal pipeline stages within the multi-stage DSP block, a standard approach to data forwarding, by adding a feedback path around the execution unit, would still require some NOP padding. We instead take advantage of a unique feature of the DSP48E1 primitive: the feedback path designed for multiply-accumulate operations when the DSP block is used to implement digital filters. This path allows the DSP-block output to be accessed at the input to the ALU stage one cycle later, potentially avoiding the need for this result to be written-back to the register file. We demonstrate that using this path for data forwarding results a significant impact on overall processor performance, while having minimal impact on area and frequency. We demonstrate a simple example of this phenomenon in Table 1 where the original code executes in 18 cycles while the optimized version with loopback requires only 12 cycles. The toolflow we develop automatically flags dependent operations for loopback in the instruction encoding.

The key contributions of this paper are:

- Parametric design of the iDEA soft processor to allow deep pipelining and loopback of operands.
- A detailed comparison between the proposed loopback forwarding and an external forwarding path.
- Development of a compiler flow and assembly backend that analyzes code for loopback potential and makes appropriate modifications to generated assembly.
- Preliminary results of IR-level analysis of the CHStone benchmark to identify opportunities for loopback in larger benchmarks.

2. RELATED WORK

Processors on FPGAs: Soft processors continue to be the method of choice for adding a level of software programmability to FPGA-based systems. They generally find use in the auxiliary functions of the system, such as managing non-critical data movement, providing an interface for configuration, or implementing the cognitive functions in an adaptive system. They provide a familiar interface for application programmers to work with the rest of the system, while supporting varied features based on need. Commercial soft processors include the Xilinx MicroBlaze [19], Altera Nios II [2], ARM Cortex-M1 [4], and LatticeMico32 [15], 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. This trade-off between portability and efficiency is clear when one considers that the performance of vendor-specific processors is much better than cross-platform designs.

Although a hard processors such as the PowerPC CPUs in Virtex-2 Pro and the ARM cores in the newer Zynq SoCs can offer better performance than an equivalent soft processor, they are inappropriate for situations where lightweight control and co-ordination [13] are required. Their fixed position in the FPGA fabric can also complicate design, and they demand supporting infrastructure for logic interfacing. When a hard processor is not used, or under-utilised, this represents a significant waste of silicon resources. In fact, the embedded PowerPCs from the Virtex-2 Pro series never gained significant traction and were dropped for subsequent high-density FPGA families.

Octavo: To maximise performance, it becomes necessary to reason about FPGA architecture capabilities, and there have been numerous efforts in this direction. Octavo [14] 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 device. A deep pipeline is necessary to support this high operating frequency. However, such a pipeline would suffer from the need to pad dependent instructions to overcome data hazards as a result of the long pipeline latency. The authors sidestep this issue by designing Octavo as a multi-issue processor, thus dependent instructions are always sufficiently far apart for such NOP padding not to be needed. However, this only works for highly-parallel code; when the soft processor is used in a sequential part of a computation, it will fail to deliver the high performance required to avoid the limits stated by Amdahl's law. Furthermore, no compiler tool flow has yet been developed for Octavo.

iDEA: We developed an alternative approach to such architecture-driven soft processor design in [8]. Here, we took advantage of the dynamic control signals of the Xil-

inx DSP block to build a soft processor that achieves a frequency of over 400 MHz on a Xilinx Virtex 6. In [6], we performed extensive benchmarking and highlighted the performance penalty of padding NOPs on total runtime, somewhat negating the benefits of high frequency.

Managing Dependencies in Processor Pipelines: A theoretical method for analysing the effect of data dependencies on the performance of in-order pipelines is presented in [9]. 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 [11]. Data dependency of sequential instructions can be resolved statically in software or dynamically in hardware. Tomasulo's algorithm, allows instructions to be executed out of order, where those not waiting for any dependencies are executed earlier. For dynamic resolution in hardware, extra functional units are needed to handle the queueing of instructions and operands in reservation stations. Additionally, handling out-of-order execution in hardware requires intricate hazard detection and execution control. Synthesising a basic Tomasulo scheduler [5] on a Xilinx Virtex-6 yields an area consumption of 20× the size of a MicroBlaze, and a frequency of only 84 MHz. This represents a significant overhead for a small FPGA-based soft processor, and the overhead increases for deeper pipelines.

Data forwarding is a well-established technique in processor design, where results from one stage of the pipeline can be accessed at a later stage sooner than would normally be possible. This can increase performance by reducing the number of NOP instructions required between dependent instructions. It has been explored in the context of general soft processor design, VLIW embedded processors [17], as well as instruction set extensions in soft processors [12]. In each case, the principle is to allow the result of an ALU computation to be accessed sooner than would be possible in the case where write back must occur prior to execution of a subsequent dependent instruction.

Our Approach: In [7], we quantified the pipeline depth/performance trade-off in the design of iDEA and explored the possible benefits of a restricted forwarding approach. In this paper, we show that the feedback path typically used for multiply-accumulate operations allows us to implement a more efficient forwarding scheme that can significantly improve execution time in programs with dependencies, going beyond just multiply-add combinations. We compare this to the previously proposed external forwarding approach and the original design with no forwarding. Adding data forwarding to iDEA decreases runtime by up to 25% across range of small benchmarks, and we expect similar gains in large benchmarks.

3. SOFT PROCESSOR ARCHITECTURE

3.1 Overview

iDEA is based on a classic 32-bit, 5-stage load-store RISC architecture with instruction fetch, decode, execute, and memory stages, followed by a 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 (see Figure 2). For this to be feasible, we use a dedicated adder to compute addresses rather than doing this through the ALU. We have used a MIPS-like ISA to enable existing open-source compilers to

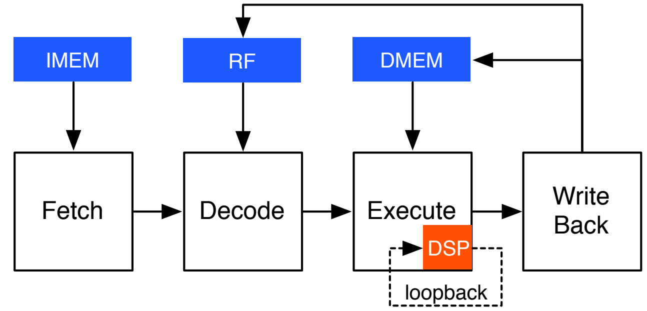

Figure 2: iDEA high-level pipeline overview.

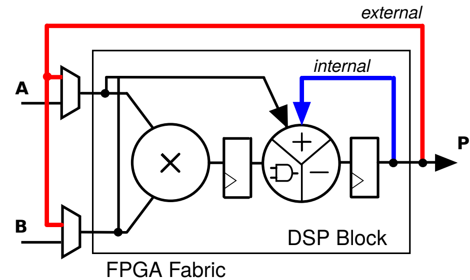

Figure 3: Execution unit datapath showing internal loopback and external forwarding paths.

target our processor. A more detailed description of the iDEA instruction set and architecture are presented in [6].

Each processor stage can support a configurable number of pipeline registers. The minimum pipeline depth for each stage is one. Fewer register stages results in a processor that achieves a lower clock frequency and fewer NOPs required between dependent instructions (See Figure 1). However, while deep pipelining of the core can result in a higher frequency, it also increases the dependency window for data hazards, hence requiring more NOPs for dependent instructions. We study this trade-off in Section 4.1.

The DSP block is utilized as the core execution unit in iDEA. All arithmetic and logical instructions use subsets of DSP block functionality. Program control instructions such as branch also use the DSP block to perform subtraction of two values in order to make a branch decision. The instruction and data memories are built using Block RAMs, while the register file is built using a quad-port RAM32M LUT-based memory primitive. The key novelty in this paper is our exploitation of the loopback path for data forwarding. The DSP block contains an internal loopback path that passes the DSP block output back into the final functional unit in its pipeline, without exiting the execute stage. This enables implementation of a fast multiply-accumulate operation in a digital filter. Through suitable selection of multiplexer controls we can use this loopback path to enable data forwarding, as described in the following section.

3.2 DSP Block Loopback Support

The DSP block is composed of a multiplier and ALU along with registers and multiplexers that control configuration options. More recent DSP blocks also contain a pre-adder allowing two inputs to be summed before entering the multi-

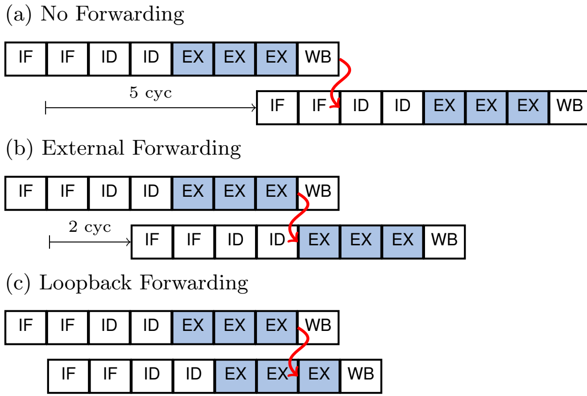

Figure 4: Instruction dependency with (a) no forwarding, (b) external forwarding, and (c) loopback forwarding.

plier. The ALU supports addition/subtraction and logic operations on wide data. The required datapath configuration is set by a number of control inputs, and these are dynamically programmable, which is the unique feature allowing use of a DSP block as the execution unit in a processor [8].

When implementing digital filters using a DSP block, a multiply-accumulate operation is required, so the result of the final adder is fed back as one of its inputs in the next stage using a loopback path, as shown in Figure 3. This path is internal to the DSP block and cannot be accessed from the fabric, however the decision on whether to use it as an ALU operand is determined by the OPMODE control signal. The OPMODE control signal chooses the input to the ALU from several sources: inputs to the DSP block, output of multiplier, or output of the DSP block. When a loopback instruction is executed, the appropriate OPMODE instructs the DSP block to take one of its operands from the loopback path. We take advantage of this path to implement data forwarding with minimal area overhead.

3.3 Data Forwarding

In Figure 4 (a), we show the typical operation of an instruction pipeline without data forwarding. In this case, a dependent instruction must wait for the previous instruction to complete execution and the result to be written back to the register file before commencing its decode stage. In this example, 5 clock cycles are wasted to ensure the dependent instruction does not execute before its operand is ready. This penalty increases with the higher pipeline depths necessary for maximum frequency operation on FPGAs.

The naive approach to implementing data forwarding for such a processor would be to pass the execution unit output back to its inputs. Since we cannot access the internal stages of the DSP block from the fabric, we must pass the execution unit output all the way back to the DSP block inputs. This *external* approach is completely implemented in general purpose logic resources. In Figure 4 (b), this is shown as the last execution stage forwarding its output to the first execution stage of the next instruction, assuming the execute stage is 3 cycles long. This still requires insertion of up to 2 NOPs between dependent instructions, depending on how many pipeline stages are enabled for the DSP block (execution unit). This feedback path also consumes fabric resources, and may impact achievable frequency.

Using the loopback path that is internal to the DSP block enables the result of a previous ALU operation to be ready as an operand in the next cycle, eliminating the need to pad subsequent dependent instruction with NOPs. The proposed loopback method is not a complete forwarding implementation as it does not support all instruction dependencies and only supports one-hop dependencies. Instead, it still allows us to forward data when the immediate dependent instruction is any ALU operation except a multiplication. Figure 4 (c) shows the output of the execute stage being passed to the final cycle of the subsequent instruction's execute stage. In such a case, since the loopback path is built into the DSP block, it does not affect achievable frequency, and eliminates the need for any NOPs between such dependent instructions.

We can identify loopback opportunities in software and a loopback indication can be added to the encoded assembly instruction. We call these one-hop dependent instructions that use a combination of multiply or ALU operation followed by an ALU operation a loopback pair. For every arithmetic and logical instruction, we add an equivalent loopback counterpart. The loopback instruction performs the same operation as the original, except that it receives its operand from the loopback path (i.e. previous output of the DSP block) instead of the register file. The loopback opcode is differentiated from the original opcode by one bit difference for register arithmetic and two bit for immediate instructions.

Moving loopback detection to the compilation flow keeps our hardware simple and fast. In hardware loopback detection, circuitry is added at the end of execute, memory access, and write back stages to compare the address of the destination register in these stages and the address of source registers at the execute stage. If the register addresses are the same, then the result is forwarded to the execute stage. The cost of adding loopback detection for every pipeline stage after execute can be severe for deeply pipelined processors, unnecessarily increasing area consumption and delay.

3.4 NOP-Insertion Software Pass

Dependency analysis to identify loopback opportunities is done in the compiler's assembly. For dependencies that cannot be resolved with this forwarding path, sufficient NOPs are inserted to overcome hazards. When a subsequent dependent arithmetic operation follows its predecessor, it can be tagged as a loopback instruction, and no NOPs are required for this dependency. For the external forwarding approach, the number of NOPs inserted between two dependent instructions depends on the DSP block's pipeline depth (the depth of the execute stage). We call this the number of ALU NOPs. A summary of this analysis scheme is shown in Algorithm 1. We analyze the generated assembly for loopback opportunities with a simple linear-time heuristic. We scan the assembly line-by-line and mark dependent instructions within the pipeline window. These instructions are then converted by the assembler to include a loopback indication flag in the instruction encoding. We also insert an appropriate number of NOPs to take care of other dependencies.

4. EXPERIMENTS

Hardware: We implement the modified iDEA processor on a Xilinx Virtex-6 XC6VLX240T-2 FPGA (ML605

Algorithm 1: Loopback analysis algorithm.

```
Data: Assembly
Result: LoopbackAssembly<vector>
w ← Number of pipeline stages − number of IF stages;
for i ← 0 to size(Assembly) do
    window ← 0;
    DestInstr ← Assembly[i];
    for j ← 1 to w-1 do
        SrcInstr ← Assembly[i − j];
        if depends(SrcInstr,DestInstr) then
            loopback ← true;
            depth ← j;
            break;
        end
    end
    for j ← 0 to w-1 do
        if loopback then
            LoopbackAssembly.push_back(Assembly[i] |
            LOOPBACK_MASK) ;
        end
        else
            LoopbackAssembly.push_back(Assembly[i]);
            for k ← 0 to j-1 do
                LoopbackAssembly.push_back(NOP);
            end
        end
    end
end
```

platform) using the Xilinx ISE 14.5 tools. We use area constraints to help ensure high clock frequency and area-efficient implementation. We generate various processor combinations to support pipeline depths from 4–15. We evaluate performance using instruction counts for executing embedded C benchmarks. Input test vectors are contained in the source files and the computed output is checked against a hard- coded golden reference, thereby simplifying verification. For experimental purposes, the pipeline depth is made variable through a parameterizable shift register at the output of each processor stage. During automated implementation runs in ISE, a shift register parameter is incremented, increasing the pipeline depth beyond the default of one. We enable retiming and register balancing, which allows registers to be moved forwards or backwards to improve frequency. We also enable the shift register extraction option. In a design where the ratio of registers is high, and shift registers are abundant, this option helps balance LUT and register usage. ISE synthesis and implementation options are consistent throughout all experimental runs.

Compiler: We generate assembly code for the processor using the LLVM- MIPS backend. We use a post-assembly pass to identify opportunities for data forwarding and modify the assembly accordingly, as discussed in Section 3.4. We verify functional correctness of our modified assembly code using a customized simulator for internal and external loopback, and run RTL ModelSim simulations of actual hardware to validate different benchmarks. We repeat our validation experiments for all pipeline depth combinations. We show a high-level view of our experimental flow in Figure 5.

In-System Verification: Finally, we test our processor on the ML605 board for sample benchmarks to demonstrate functional correctness in silicon. The communication between the host and FPGA is managed using the open source

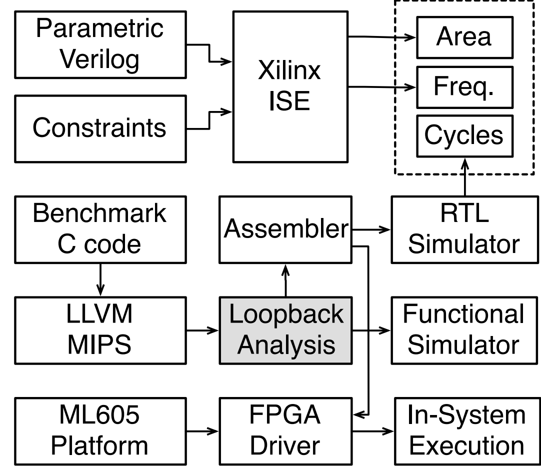

Figure 5: Experimental flow.

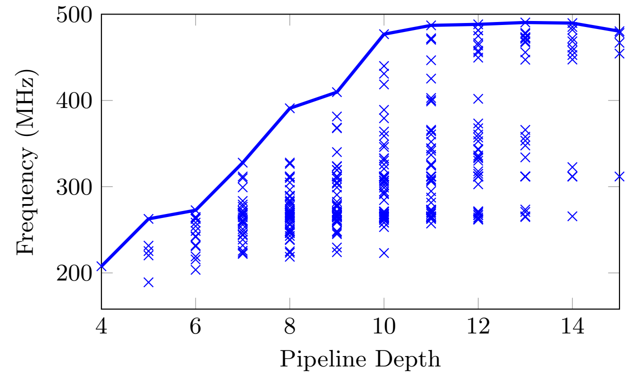

Figure 6: Frequency of different pipeline combinations with internal loopback.

FPGA interface framework in [18]. We verify correctness by comparing the data memory contents at the end of functional and RTL simulation, and in-FPGA execution.

4.1 Area and Frequency Analysis

Since the broad goal of iDEA 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 stages from 1–5 for each stage: fetch, decode, and execute, and the resulting overall pipeline depth is 4–15 (the writeback stage is fixed at 1 cycle).

Impact of Pipelining: Figure 6 shows the frequency achieved for varying pipeline depths between 4–15 for a design with internal loopback enabled. Each depth configuration represents several processor combinations as we can distribute these registers in different parts of the 4-stage pipeline. The line traces points that achieve the maximum frequency for each pipeline depth. The optimal combination of stages, that results in the highest frequency for each depth, is presented in Table 2.

While frequency increases considerably up to 10 stages, beyond that, the increases are modest. This is expected

Table 2: **Optimal combination of stages and associated NOPs at each pipeline depth (WB = 1 in all cases)**

Depth	IF	ID	EX	NOPs	ALUNOPs
4	1	1	1	2	0
5	1	2	1	3	0
6	2	2	1	3	0
7	2	1	3	4	2
8	2	2	3	5	2
9	2	2	4	6	2
10	3	2	4	6	2
11	3	2	5	7	2
12	3	3	5	8	2
13	4	3	5	8	2
14	5	3	5	8	2
15	4	5	5	10	2

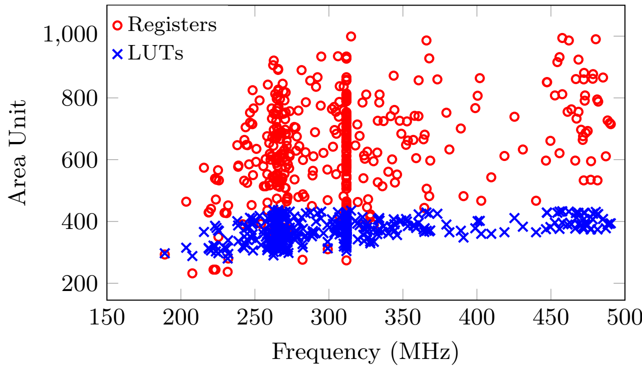

Figure 7: Resource utilization of all pipeline combinations with internal loopback.

as we approach the raw fabric limits around 500 MHz. For each overall pipeline depth, we have selected the combination of pipeline stages that yields the highest frequency for all experiments. With an increased pipeline depth, we must now pad dependent instructions with more NOPs, so these marginal frequency benefits can be meaningless in terms of wall clock time for an executed program. In Fig. 4, we illustrated how a dependent instruction must wait for the previous result to be written back before its instruction decode stage. This results in required insertion of 5 NOPs for that 8 stage pipeline configuration. For each configuration, we determine the required number of NOPs to pad dependent instructions, as detailed in Table 2.

Figure 7 shows the distribution of LUT and register consumption for all implemented combinations. Register consumption is generally higher than LUT consumption, and this becomes more pronounced in the higher frequency designs. Figure 8 shows a comparison of resource consumption between the designs with no forwarding, internal loopback, and external forwarding. External forwarding generally consumes the highest resources for both LUTs and registers. The shift register extraction option means some register chains are implemented instead using LUT-based SRL32 primitives, leading to an increase in LUTs as well as registers as the pipelines are made deeper.

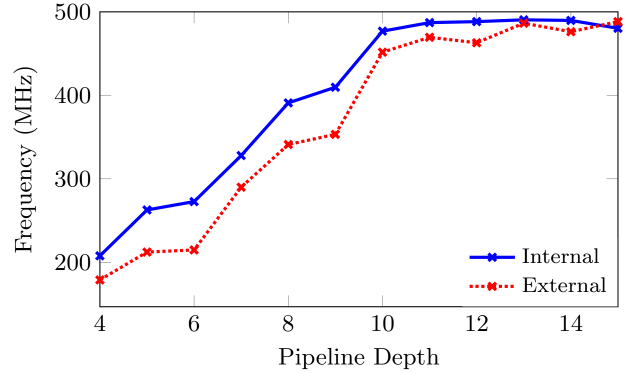

Figure 9: Frequency with internal loopback and external forwarding.

Table 3: Static cycle counts with and without loopback for a 10 cycle pipeline with % savings.

Bench mark	Total Inst.	Loopback	
		Inst.	%
crc	32	3	9
fib	40	4	10
fir	121	1	0.8
median	132	11	8
mmult	332	3	0.9
qsort	144	10	7

Impact of Loopback: Implementing internal loopback forwarding proves to have a minimal impact on area, of under 5%. External forwarding generally uses slightly more resources, though the difference is not constant. External forwarding does lag internal forwarding in terms of frequency for all pipeline combinations, as shown in Figure 9, however, the difference diminishes as frequency saturates at the higher pipeline depths. Though we must also consider the NOP penalty of external forwarding over internal loopback.

4.2 Execution Analysis

Static Analysis: In Table 3, we show the percentage of occurrences of consecutive loopback instructions in each benchmark program. Programs that show high potential are those that have multiple independent occurrences of loopback pairs, or long chains of consecutive loopback pairs. Independent pairs of loopbacks are common in most programs, however for `crc` and `fib`, we can find a chain of up to 3 and 4 consecutive loopback pairs respectively.

Dynamic Analysis: In Table 4, we show the actual execution cycle counts without forwarding, with external forwarding, and with internal loopback, as well as the percentage of executed instructions that use the loopback capability. Although `fib` offers the highest percentage of loopback occurrences in static analysis, in actual execution, `crc` achieves the highest savings due to the longer loopback chain, and the fact that the loopback-friendly code is run more frequently.

Internal Loopback: In Figure 10, we show the Instructions per Cycle (IPC) savings for a loopback-enabled processor over the non-forwarding processor, as we increase pipeline depth. Most benchmarks have IPC improvements

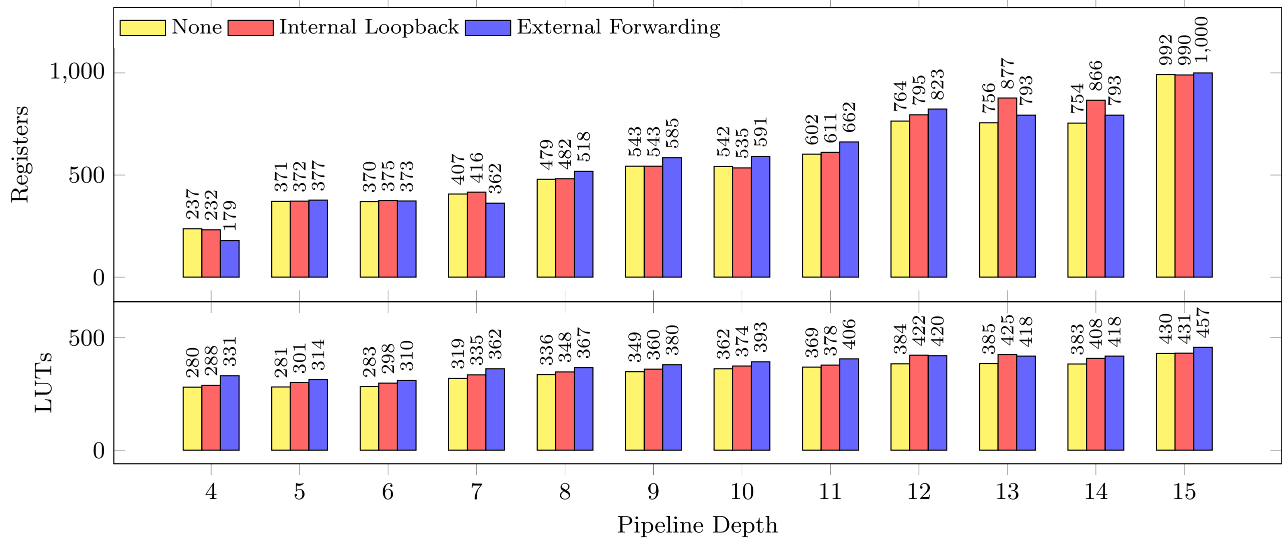

Figure 8: Resource utilization of highest frequency configuration for no forwarding, internal loopback, and external forwarding.

Table 4: Dynamic cycle counts with and without loopback for a 10 cycle pipeline with % savings.

Bench mark	Loopback				
	Without	External	%	Internal	%
crc	28,426	22,426	21	20,026	29
fib	4,891	4,211	14	3,939	19
fir	2,983	2,733	8	2,633	11
median	1,5504	14,870	4	14,739	5
mmult	1,335	1,322	0.9	1,320	1
qsort	32,522	30,918	5	30,386	7

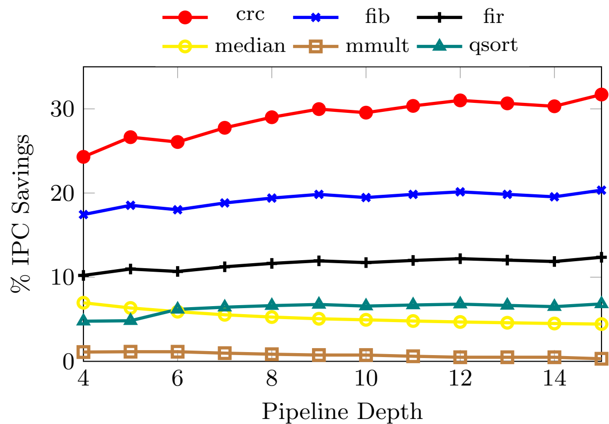

Figure 10: IPC improvement when using internal DSP loopback.

between between 5–30% except the `mmult` benchmark. For most benchmarks, we note resilient improvements across pipeline depths. From Table 4 we can clearly correlate the IPC improvements with the predicted savings.

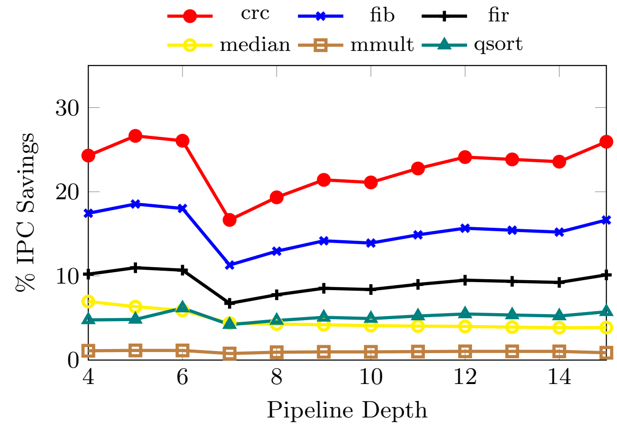

Figure 11: IPC improvement when using external loopback.

External Loopback: Figure 11 shows the same analysis for external forwarding. It is clear that external forwarding is not as improved as internal loopback, since we do not totally eliminate NOPs in chains of supported loopback instructions. For pipeline depths of 4–6, the IPC savings for internal and external loopback are equal, since the execute stage is 1 cycle (refer to Table 2), and hence neither forwarding method requires NOPs between dependent instructions. For external forwarding, when the execute stage is $K > 1$ cycles, we need $K - 1$ NOPs between dependent instructions, which we call ALU NOPs. Table 2 shows the number of NOPs for every pipeline combination and the corresponding ALU NOPs for external forwarding. As a result of the extra NOP instructions, the IPC savings decline marginally in Figure 11 and stay relatively low.

Impact of Internal Loopback on Wall-Clock Time Figure 12 shows normalised wall-clock times for the different benchmarks. We expect wall-clock time to decrease as

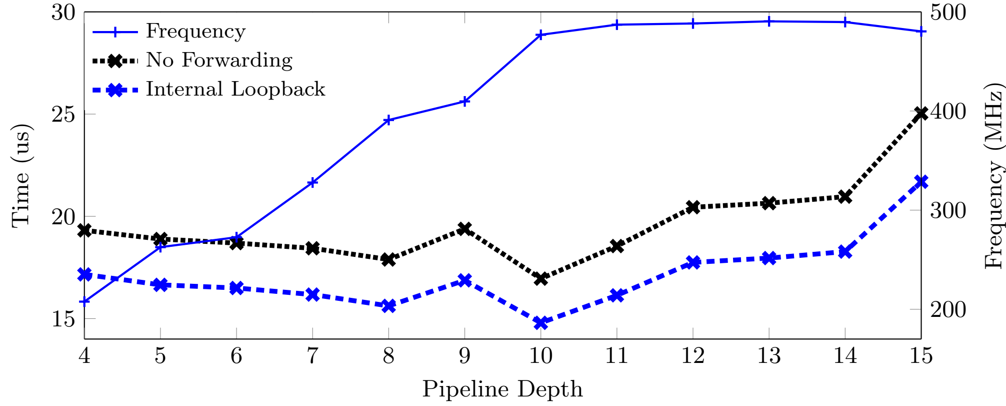

Figure 12: Frequency and geomean wall clock time with and without internal loopback enabled.

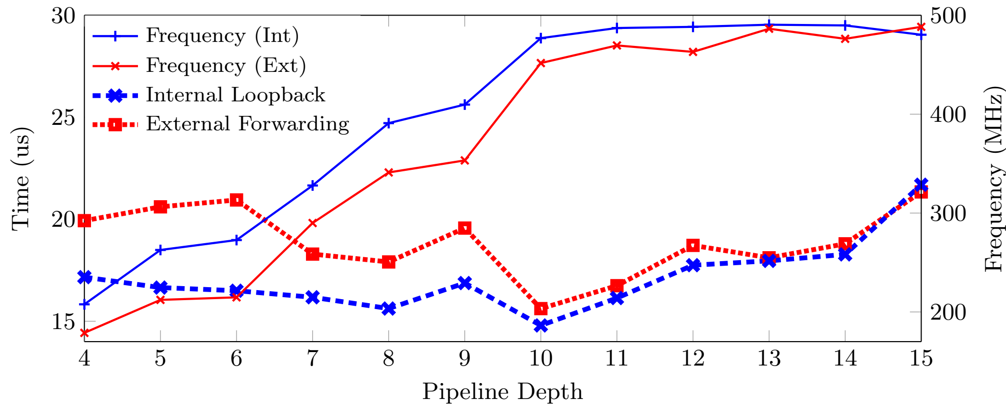

Figure 13: Frequency and geomean wall clock time on designs incorporating internal loopback and external forwarding.

we increase pipeline depth up to a certain limit. At sufficiently high pipeline depths, we expect the overhead of NOPs to cancel the diminishing improvements in operating frequency. There is an anomalous peak at 9 stages due to a more gradual frequency increase, visible in Figure 6, along with a configuration with a steeper ALU NOP count increase as shown in Table 2. The 10 cycle pipeline design gives the lowest execution time for both internal loopback and non-loopback. Such a long pipeline is only feasible when data forwarding is implemented, and our proposed loopback approach is ideal in such a case, as we can see from the average 25% improvement in runtime across these benchmarks.

Comparing External Forwarding and Internal Loopback Figure 13 shows the maximum frequency and normalised wall clock times for for internal loopback and external forwarding. As previously discussed, external forwarding results in higher resource utilisation and reduced frequency. At 4–6 cycle pipelines, the lower operating frequency of the design for external forwarding results in a much higher wall-clock time for the benchmarks. While the disparity between external and internal execution time is significant at shallower pipeline depths, the gap closes as depth increases. This is due to the saturation of frequency at pipeline depths greater than 10 cycles and an increase in the insertion of ALU NOPs. The 10 cycle pipeline configuration yields the lowest execution time for all three designs, with internal loopback achieving the lowest execution time.

5. FURTHER INVESTIGATION

We developed our soft processor for small compact loop bodies that execute control oriented code on the FPGA with a large number of complex instruction dependencies. Furthermore, in the long run, we expect to develop a parallel array of these lightweight soft processors to operate in tandem for compute-intensive parallel tasks. However, our processor is also capable of supporting larger programs from the CHstone benchmarks suite. Additionally, we have only considered loopback analysis at the post-assembly stage. This assumes that chains of dependent operations are always kept in close proximity by the compiler. In reality, this may not be the case, and a compiler pass (before backend code-generation) could increase the effectiveness of this approach by ensuring that compatible dependent instructions are kept in sequence.

CHStone compatibility: To explore the applicability of this approach in more complex applications supported with compiler-assisted analysis, we profiled LLVM [16] IR representations of 8 benchmarks from the CHStone benchmark suite [10]. Static analysis shows a significant number of compatible dependency chains. We also use the LLVM profiler

Table 5: LLVM IR Profiling Results for CHStone

Bench mark	Static			Dynamic		
	Instr.	Occur.	%	Instr.	Occur.	%
adpcm	1367	184	13	71,105	8,300	11
aes	2259	51	2	30,596	3,716	12
blowfish	1184	314	26	711,718	180,396	25
gsm	1205	82	6	27,141	1,660	6
jpeg	2388	95	4	1,903,085	131,092	6
mips	378	15	3	31,919	123	0.3
mpeg	782	80	10	17,032	60	0.3
sha	405	64	15	990,907	238,424	24

and just-in-time (JIT) compiler to obtain dynamic counts of possible loopback occurrences. The results in Table 5 show a mean occurrence of over 10% within these benchmarks. We cannot currently support CHStone completely due to missing support for 32b instructions and other development issues.

Program Size Sensitivity: We also synthesized RTL for iDEA with increased instruction memory sizes to hold larger programs. We observed, that iDEA maintains its optimal frequency for up to 8 BRAMs. Beyond this, frequency degrades by 10–30% to support routing delays and placement effects of these larger memories. However, we envision tiling multiple smaller soft processors with fewer than 8 BRAMs to retain frequency advantages for a larger multiprocessor system, and to reflect more closely the resource ratio on modern FPGAs. Each processor in this system would hold only a small portion of the complete system binary.

6. CONCLUSIONS AND FUTURE WORK

We have shown an efficient way of incorporating data forwarding in DSP block based soft processors like iDEA. By taking advantage of the internal loopback path typically used for multiply accumulate operations, it is possible to allow dependent ALU instructions to immediately follow each other, eliminating the need for padding NOPs. The result is an increase in effective IPC, and 5– 30% (mean 25%) improvement in wall clock time for a series of benchmarks when compared to no forwarding and a 5% improvement when compared to external forwarding. We have also undertaken an initial study to explore the potential for such forwarding in more complex benchmarks by analysing LLVM intermediate representation, and found that such forwarding is supported in a significant proportion of dependent instructions. We aim to finalise full support for the CHStone benchmark suite as well as open-sourcing the updated version of iDEA and the toolchain we have described.

7. REFERENCES

[1] Aeroflex Gaisler. *GRLIB IP Library User's Manual*, 2012.

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

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, pages 483–485, 1967.

[4] ARM Ltd. *Cortex-M1 Processor*, 2011.

[5] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul. Putting it all together - formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8:411–430, 2006.

[6] 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*, 7(3):19, 2014.

[7] H. Y. Cheah, S. A. Fahmy, and N. Kapre. Analysis and optimization of a deeply pipelined FPGA soft processor. In *Proceedings of the International Conference on Field Programmable Technology (FPT)*, pages 235–238, 2014.

[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)*, pages 151–158, Dec. 2012.

[9] P. G. Emma and E. S. Davidson. Characterization of branch and data dependencies in programs for evaluating pipeline performance. *IEEE Transactions on Computers*, 36:859–875, 1987.

[10] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. In *Journal of Information Processing*, 2009.

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

[12] R. Jayaseelan, H. Liu, and T. Mitra. Exploiting forwarding to improve data bandwidth of instruction-set extensions. In *Proceedings of the Design Automation Conference*, pages 43–48, 2006.

[13] 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.

[14] 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)*, pages 219–228, Feb. 2012.

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

[16] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[17] M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalom. Exploiting data forwarding to reduce the power budget of VLIW embedded processors. In *Proceedings of Design, Automation and Test in Europe, 2001*, pages 252–257, 2001.

[18] 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)*, pages 128–135, Dec. 2013.

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