Software Decelerators

Eric Keller, Gordon Brebner and Phil James-Roxby

Xilinx Research Labs, Xilinx Inc., U.S.A.
{Eric.Keller,Gordon.Brebner,Phil.James-Roxby}@xilinx.com

Abstract. This paper introduces the notion of a *software decelerator*, to be used in logic-centric system architectures. Functions are offloaded from logic to a processor, accepting a speed penalty in order to derive overall system benefits in terms of improved resource use (e.g. reduced area or lower power consumption) and/or a more efficient design process. The background rationale for such a strategy is the increasing availability of embedded processors 'for free' in Platform FPGAs. A detailed case study of the concept is presented, involving the provision of a high-level technology-independent design methodology based upon a finite state machine model. This illustrates easier design and saving of logic resource, with timing performance still meeting necessary requirements.

1 Introduction

The research literature in field-programmable logic contains many examples of 'hardware accelerators'. In short, these concern a processor-centric system model: algorithms are executed on a processor, with certain key functions being performed by an associated programmable logic array, the intention being to achieve greater overall performance. The exact arrangements may vary from the tightly-integrated case of a processor with an augmented instruction set (e.g. [3]) to the more loosely-coupled case of a processor interacting with a co-processing logic device (e.g. [6]).

This paper is concerned with a *logic-centric* system model, of the sort described for example in earlier papers by Brebner [2]. Here, the main computational focus is on logic circuitry, with other components — in particular processors — viewed as additional system components rather than central system components. Aside from computational differences, there are implied architectural differences, notably concerning serial data access and shared buses, features both tailored to processor behavior. The logic-centric model is well-suited to systems that react to, and are driven by, inputs received over time. Thus, in contrast to the usual processor-centric model, the environment, not the computer, is the controlling force. We feel that this view of will be of increasing relevance to real-life systems in the future.

In the logic-centric model, it is fairly natural to invert accepted wisdom about system organization. In this paper, we consider the benefits of 'software decelerators', inverting the notion of hardware accelerators. The basic idea is that algorithms are executed in programmable logic, with certain functions being performed by an associated processor. In general, this direction of migration is not likely to lead to speed increases — indeed quite the opposite — which is why we use the word 'decelerator'; however, overall system speed requirements will still be met. It is intended that there are other motivations that lead to an overall increase in the quality of the design process and/or the resulting systems. We discuss various possible motivations, and then describe one detailed case study experiment where the main motivation was to provide a high-level, technology-independent design methodology based upon finite state machines.

In this case study, the software decelerator technique concerns finite state machines being implemented on the embedded processor of a Platform FPGA. With this approach, the cost of implementing a machine in terms of logic resource usage is greatly reduced, since the processor is always present on the Platform FPGA, whether it is used or not. The emphasis differs from that of some conventional state machine design methodologies, such as Esterel Studio, the principal aims being to consume as few logic resources as possible, optimize the interfacing between logic and processor, and run code directly from the processor's built-in cache.

The paper is organized as follows. Section 2 considers the technological background that motivates software decelerators as a viable concept in system design. Then, Section 3 describes the case study, and Section 4 discusses initial experimental results. Finally, Section 5 contains some conclusions and directions for future work.

2 Technological background

2.1 Emergence of Platform FPGAs

Early Field Programmable Gate Array (FPGA) architectures consisted of an array of similar programmable logic elements interfaced to interconnection elements. With the emergence of the Platform FPGA, architectures have evolved to a pre-defined mix of very different elements — which in time will largely supersede monolithic logic arrays. Today, for example, the Xilinx Virtex -II Pro Platform FPGA contains configurable logic blocks, I/O blocks supporting many different I/O standards, distributed BlockRAM memories, internal access to the configuration memory, digital clock managers, gigabit transceivers, dedicated multiplier blocks and embedded PowerPC 405 processors. Platform FPGAs present both unique design challenges and unique design opportunities. Importantly, the mix of resources is pre-determined by the FPGA vendor rather than by the designer, and so a particular set of resources will be present whether or not the designer makes use of them, a situation unlike that in ASIC design.

Therefore, in designs that aim to maximize the use of the resources of a certain device size, the designer may often make decisions which on the surface appear non-intuitive. For example, in designs which use little BlockRAM memory but use many configurable logic blocks (CLBs), a designer may choose to implement certain logic functions by lookup tables in BlockRAM rather than in CLBs. This is despite the fact that this approach wastes much of the BlockRAM data width, and does not take advantage of their dual-port nature. Such an approach would be unthinkable in ASIC design, but makes a lot of sense in Platform FPGA design. This use of pre-existing resources in unusual ways is likely to become more and more prevalent as the mixture of resources becomes richer and richer. The overall message is that one has to be very fluid in terms of how and where computation, storage and communication are carried out in systems.

The logic-centric system paradigm is one approach to assist in managing the potential design space. This focuses on inputs, outputs and programmable logic circuitry as the core system architecture, with all other components (memory, processors, etc.) being seen as *assists* to the circuitry. Of course, it is also possible to view the system in other ways. For example, in a more conventional processor-centric manner, the Virtex-II Pro can also be seen as a 'motherboard on a chip', and indeed it has been demonstrated as such, with the Linux operating system running on the PowerPC processor.

2.2 Motivation for software decelerators

The concept of using software decelerators derives directly from the discussion of using Platform FPGA resources in unusual ways, and focuses on processors in particular for non-standard treatment. In fact, what is sought here is not necessarily a completely different and unusual harnessing of processors, rather some balance between the long and rich legacies of processor development and software engineering, and the new context of the logic-centric system.

The more general backdrop to this reconsideration of the role of processors is an examination of the role of any kind of *universal machine* within a logic-centric system. Other examples, simpler than a traditional microprocessor, would be programmable state machines or microcontrollers. In all such cases, there is a basic trade-off between speed — one normally expects a universal machine implementation of a function to be slower than a bespoke implementation — and other issues as diverse as chip area requirements and ease and speed of implementing new functions.

So far, in the development of Platform FPGAs that include processors, there has largely been a drive to maximize processor clock rates, reflecting a view either that the processor is the central system component or perhaps that the processor plays a key time-critical supporting role in the system. This drive parallels the continuing race to increase clock rates of microprocessor chips (although, very recently, there has been acknowledgement that raw speed is not everything, power consumption being of importance in an increasing proportion of systems). It is our belief that, as far as Platform FPGAs are concerned, the clock rate of the processor may not be the dominant concern for many future systems. This follows from a prediction that the processor may either be called upon relatively infrequently to carry out work in the logic-centric system, or may be called upon to perform work of a non time-critical nature. One recent example of such a system is the high throughput, low latency mixed-version IP router developed for the Virtex-II Pro [2].

As soon as the clock rate requirement is relaxed, it becomes possible to focus the treatment of a processor on other goals, like saving logic resource by employing the processor plus its supporting cache memory. In this paper, the goal is to combine this particular trade-off with the provision of a particular high-level design flow that hides the nature of the implementation from the user. A more straightforward and obvious application of software deceleration is just to make the overall system implementation process easier by allowing a designer access to the wide range of software tools (and human expertise) to implement functions on the processor, rather than implementing logic circuitry.

To ensure that software decelerators provide the anticipated benefits, and do not impose resource demands on an overall logic-centric system design, nor impose unnatural design methodologies on the user, there are some particular attributes that are desirable:

– The overall area consumed by a software decelerator implementation should not be greater than its logic circuitry counterpart unless there are other strong benefits.
– The interfacing between logic circuitry and processor should consume minimal programmable logic resources, and should be designed to shield the processor from the logic and vice-versa.
– The method of capturing designer intent should be independent of the actual implementation mechanism chosen for the Platform FPGA unless there is a particular benefit in permitting the use of certain familiar tools.
– The designer should be able to get accurate timing information, and resource usage information in general, for the overall logic-centric system, taking into account the behavior of the various non-logic system components that are being harnessed.

The case study that is presented in the subsequent sections illustrates that it is feasible to meet these goals, in the framework of software decelerator use.

3 Case study: FSM-based design methodology

Finite state machines (FSMs) are an important component of many digital systems, and can be implemented well on FPGAs. Particular FSMs may contain a large number of states, and may involve much computation to determine the next state and the state outputs based on varying inputs. However, they may actually have relatively relaxed timing constraints compared to the rest of a system, an attribute that points to possible software decelerator implementation.

The case study problem tackled was to implement FSMs as an example of a software decelerator. Referring back to three desirable attributes defined in Section 2.2, the interfacing hardware should consume minimal resources and act as a shield between the processor and the rest of the system — the rest of the system should not know it is working with a processor. Capture should be implementation independent, that is, the designer should not be aware that an embedded processor is being targeted. Finally, accurate timing and resource usage information should be obtainable so that the designer is able to get hardware-like metrics. The net effect of using a software decelerator in a system where the processor would otherwise be idle is that logic resources are freed, without penalizing the designer in terms of design style or quality of timing information.

There has been a fairly substantial body of prior work on implementing finite state machines in software. Perhaps the most notable effort is the Berekeley POLIS system [1]. POLIS is a complete co-design solution, which uses the codesign finite state machine (CFSM) as the central representation of the required system behavior. One significant difference between a CFSM and a classical FSM is that a CFSM allows that the reaction time to events will be non-zero, unlike the FSM which assumes a synchronous communication model.

POLIS allows the designer to implement CFSMs in either software or hardware, and since this is a co-design solution — a single CFSM can be partitioned into multiple CFSM sub-networks, and have different target implementations. The hardware CFSM sub-networks are constructed using standard logic synthesis techniques, and in this case a CFSM can execute a transition in a single clock cycle.

A CFSM sub-network chosen for software implementation is transformed to a program and a simple custom real time operating system (RTOS). The program is generated from a control/data flow graph, and is coded in C. The designer can use a timing estimator to find quickly the speed of the software, or instead produce an instrumented version of the code and run this on an actual processor. The instrumented version counts the actual cycles used, giving a more accurate way of extracting timing information. The custom RTOS consists of a scheduler for organizing the execution of the procedures, using policies such as rate-monotonic or deadline-monotonic scheduling. The RTOS also includes I/O drivers. The case study here is rather different in nature from POLIS, as it has very different aims, including minimizing logic-processor interfacing and code size.

3.1 System description

The case study involved producing a tool which takes a textual representation of a finite state machine and produces: a hardware platform that can be interfaced to existing logic circuitry; software to run on the embedded processor; and a timing report. The tool flow is shown in Figure 1.

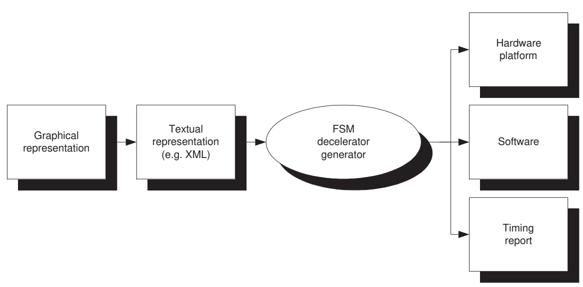

Fig. 1. Tool flow

3.2 Design entry

A number of methods are available for capturing FSMs. Tools such as Esterel Studio or Xilinx's StateCAD allow a designer to capture graphically the FSM as a state transition diagram. Designers annotate the state transition diagram with conditions for taking branches, and define the calculations for the state outputs. Alternatively, FSMs can be described in a conventional HDL. In order to support the maximum number of possible design methods, an XML grammar was defined to capture the functionality of an FSM. In a file containing a description following this grammar, the interface of the machine is specified first. For example:

```
⟨variables⟩
  ⟨variable name="rst" dir="in" width="1" registered="true"/⟩
  ⟨variable name="clk" dir="in" width="1" registered="true" /⟩
  ⟨variable name="phy_ad" dir="in" width="5" registered="true" /⟩
  ⟨variable name="mdio_tristate" dir="out" width="1" registered="true" /⟩
⟨/variables⟩
```

After this, the description specifies the states. An initial tag specifies the global conditions for the state machine, such as reset input, clock input, reset state, and synchronous or asynchronous reset. A description of the individual states follows this. A state has equations and transitions associated with it. Each equation assigns some value to an output. Input, constants and basic operators (add, sub, and, or, etc.) are used to form the right-hand side of the equation. Transitions include the next state and the condition when the transition occurs. Equations can also be associated with transitions. The time when the equation is executed depends on where the equation is located — in the state or in the transition.

```
⟨state name="stateADD"⟩
  ⟨equations⟩ ⟨equation lhs="out0" rhs="in1 + in2" /⟩ ⟨/equations⟩
  ⟨transitions⟩
    ⟨transition condition="else" next="state1"⟩
      ⟨equations⟩ ⟨equation lhs="ready" rhs="1" /⟩ ⟨/equations⟩
    ⟨/transition⟩
  ⟨/transitions⟩
⟨/state⟩
```

3.3 Logic-processor interfacing

In a conventional SoC design containing processors, the processor is normally connected to the rest of the logic via a system bus or other on-chip network [5]. The processor is a master on the network, initiating and responding to transfers. Since multiple masters may be present in a bus-based system, an arbiter is needed, and masters must first request the bus from it. In the case of an SoC implementation using Virtex-II Pro Platform FPGAs, the logic required to generate the arbiter, the bus itself and the bus interfaces on each of the slaves is implemented in the fabric of the FPGA.

Examining the interface of the PowerPC to the logic fabric in the Virtex-II Pro, there is some flexibility in choosing a method of transferring information between the processor and the logic, and vice-versa. In essence though, for all methods, communication is done over a bus at the processor/logic boundary. Three data buses are natively interfaced to the PowerPC: the On-Chip Memory (OCM) bus, and the Device Control Register (DCR) and Processor Local Bus (PLB) buses from the CoreConnect family of SoC interconnect.

The logic design task in the tool development was to take the XML description of the FSM, and produce the interfacing harness as illustrated in Figure 2. The role of this harness is to shield the software decelerator, so in effect the FSM code running on the processor looks like an FSM implemented in logic — that is, the rest of the system

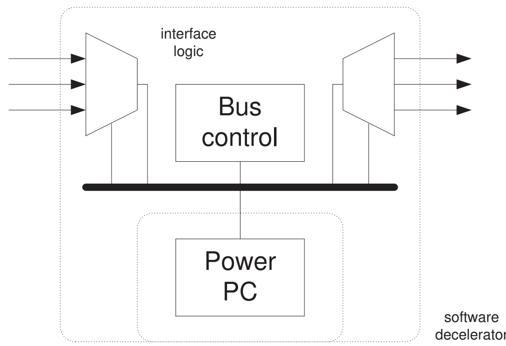

Fig. 2. Software decelerator architecture

should not see any processor specific signals. Moreover, the philosophy is to allow logic to communicate with software using the minimum amount of interfacing.

To interface with the embedded PowerPC processor, it is necessary to use one of the three native buses mentioned above. Simplifications to the interface logic can be made by relying on the fact that only one master (the processor) is present and that it only talks to one slave (the logic circuitry). In each case, the slave can assume it is being addressed all the time, and can simply write outputs and read inputs directly to and from the data bus. For the DCR and the PLB buses, the master interface in the processor assumes the existence of an acknowledge signal, but the logic to do this is very simple.

State machines are normally driven by a clock that dictates when the machine should move between states. For software decelerators in general, there will not be a relationship between the clocks of the rest of the system and the processor. For the state machine and general decelerators, there are two methods for dealing with clocks.

The first method is to use the clock for the state machine directly, as if it were a normal input. Since the processor operates at a much higher frequency than the rest of the system, it is possible for the processor to poll the clock input, and begin processing when it detects a rising edge. The limitation with this method for finite state machines is that the worse case state execution dictates whether the system will meet the timing requirements or not. Figure 3 shows a simplified timing diagram using this method.

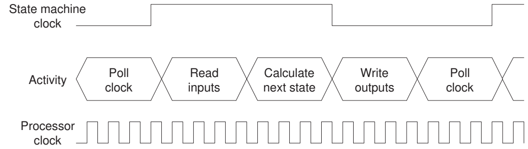

Fig. 3. Simplified timing diagram

A second method is to generate a clock pulse from the processor itself using a memory-mapped one-shot circuit. In this case, states would be allowed to take a different number of processor cycles to complete — the clock pulses would simply appear after a different number of processor cycles, but the external circuits would know when their inputs and outputs have been clocked.

3.4 Timing generation

The ability to get accurate timing information is crucial to the success of the software decelerator technique. The designer needs to be confident that the overall system, including the decelerator, meets overall timing constraints. To do this, a measure of the delay through the software decelerator is required, in exactly the same way as in hardware design, where the delay through logical elements is required.

Li and Malik present a good discussion of the state of the art of determining the worst-case execution time for software [4]. Similar techniques would be needed for general software decelerator use, where arbitrary code structures are permissible. However, in the case study, the structure of the generated software is strictly under the control of the tool. Therefore, given that the execution times for each instruction type are documented, the tool can count the actual number of processor cycles. Since the software runs out of cache and the time to perform bus transfers to the rest of the system is known, this cycle count is very accurate. This is different to the approach used by POLIS, which generates execution characteristics by instrumenting and running code.

3.5 Software design

Section 2.2 stated that interfacing should consume minimal resources, to make the software decelerator a value proposition to the designer. Similarly, other support for the processor (e.g. memory and clock control) should consume minimal resources. In the case of the PowerPC, it is possible to reduce external memory requirement to zero by using the instruction and data caches as main memory. This means that the whole executable needs to fit inside the 16Kb instruction cache. In the FSM case under consideration, where no lavish software support is required, extremely complex state machines would still fit inside the 16Kb limit.

It was decided to use assembly code directly for the software implementation. This had two specific advantages. The first advantage is that using assembler simplifies the problem of extracting timing information as described in Section 3.4. The other is that the PowerPC instructions *mfdcr* and *mtdcr*, which move data from and to the DCR bus respectively, can be used if the DCR bus is chosen as the interface. These instructions move data between a specified general-purpose register and the DCR, and thus are difficult to deal with from compiled high-level languages.

Every state uses the same template to create output assembly code. The most complex task is the translation to assembly code of the equations to determine the next state and the state outputs. Inputs are only read if they are used in these equations. The conditions for translations use the same method to calculate the value of the condition.

Special care was taken to maximize the usage of registers and only use the cache memory if needed. This can lead to more efficient code — a useful feature since the state machine is already operating at a much lower frequency than the FPGA fabric.

4 Experimental results

The tool was used on three state machines from the networking domain, with very different performance and I/O requirements. In each case, the clock is supplied by the system's environment. The first state machine (rs232echo) was an RS232 protocol handling machine, which echoed received inputs onto outputs, and the second one (miim) handled the Media Independent Interface (MII) of an Ethernet MAC which runs at 2.5MHz. The third state machine (tx_host_io) handles the host interface to a 10G Ethernet MAC. This machine clearly has a need for high performance, and is included here as an illustration of a case where a software decelerator is not likely to be chosen as the implementation technique. The first table shows the results for a direct logic circuit implementation from synthesized VHDL.

Machine	Input width	Output width	Number of states	Registers	LUTs	Required frequency
rs232echo	3	1	12	92	111	115 kHz
miim	33	20	33	26	61	2.5 MHz
tx_host_io	90	94	5	142	320	156 MHz

To determine the possible real estate savings through using a software decelerator, the new tool was run targeting each of the three available buses. The resource usage and the relative savings in terms of LUTs used compared with the direct logic implementation for each of the buses is shown in the next table.

	OCM			DCR			PLB		
Machine	Registers	LUTs	Ratio	Registers	LUTs	Ratio	Registers	LUTs	Ratio
rs232echo	1	4	3.6%	2	6	5.4%	4	8	7.2%
miim	20	38	62.3%	21	40	65.6%	23	42	68.9%
tx_host_io	94	75	23.4%	95	77	24.1%	97	79	24.7%

The OCM-based implementations are the smallest of the three for each example. Also, the OCM is as fast as the PLB bus for processor/logic interaction — in both cases they take four processor cycles, as opposed to the DCR which takes nine processor cycles. These figures assume the system bus operates at half the frequency of the PowerPC core, that is the PowerPC operates at 350MHz, and the bus clock runs at 175 MHz. In order to determine timing figures, each of these machines was implemented using the new tool, targeting the OCM bus. The final table shows the results for each of the example machines.

Machine	Worst-case performance (cycles)	Worst-case performance (MHz)	% of time in I/O	Code size (kbytes)	Code size as % of cache
rs232echo	40	8.75	30.95%	1416	8.6%
miim	74	4.730	25.22%	2968	18.1%
tx_host_io	135	2.593	33.99%	1952	11.9%

The rs232echo would work at all required baud rates, and the performance of the miim state machine is well inside the required 2.5MHz limit. These examples illustrate a software decelerator delivering the required performance, rather than an unnecessarily high performance. The tx_host_io state machine however, and as expected, does not operate at anything like the required frequency, and is an example of a case where high performance is very much the key focus. In each case, the code only occupies a fraction of the cache. Thus, it would be possible to run multiple state machines, that is multiple software decelerators, on the same processor, as long as any cumulative timing requirements are still be met.

It can be seen that I/O occupies a large proportion of time in each machine, and is clearly a limiting factor in the machine speeds that can be achieved. In the case of state machines, it may be possible to exploit the rich routing resources of the FPGA, and introduce parallel transfers by packing the inputs required for each state into a single word. This would require adjustments to the input multiplexor shown in Figure 2. The parallelized input signals could then be unpacked inside the processor by shift and masks or alternatively, instructions could be applied directly to the packed signals.

5 Conclusions and future work

This research has introduced software decelerators as a mechanism for harnessing the resources of an embedded processor in a Platform FPGA, making the point that maximizing raw processor speed is not likely to be an issue in many logic-centric systems. An application of this philosophy has been demonstrated through the FSM-based design methodology. This experiment shows encouraging initial results in terms of overall resource usage and ease of design. The authors are now evaluating the methodology on larger Xilinx customer designs containing multiple state machines. Future work foci will include: further study of the implications of adopting a logic-centric system model; automatic selection and synthesis of apt logic-processor interfaces; characteristics of soft and hard embedded processors; FSM-based architectural components; and the provision of domain-specific high-level design entry and tools.

References

1. F. Baloron, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, M. Chiodo, H. Hsieh, L. Lavagno, A. L. Sangiovanni-Vincentelli, and K. Suzuki. *Hardware-Software co-design of embedded systems: the POLIS approach*. Kluwer, 1997.
2. Gordon Brebner. Single-chip Gigabit mixed-version IP router on Virtex-II Pro. *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM02)*, pp.35–44, April 2002.
3. M. Dales. The Proteus processor - a conventional CPU with reconfigurable functionality. *9th Int. Workshop on Field Programmable Logic (FPL99)*, pp.431–437, September 1999.
4. Y. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *ACM/IEEE Design Automation Conference (DAC95)*, pp.456–461, June 1995.
5. M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. *ACM/IEEE Design Automation Conference (DAC01)*, pp.667–672, June 2001.
6. S. Singh and R. Slous. Accelerating Adobe Photoshop with reconfigurable logic. *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM98)*, pp.15–17, April 1998.