Modern Applied Science; Vol. 7, No. 3; 2013
ISSN 1913-1844 E-ISSN 1913-1852
Published by Canadian Center of Science and Education

FPGA Implementations of Ladder Diagrams

Neil W. Bergmann[1], Peter Waldeck[1] & Sunil K. Shukla[1]

[1] School of Information Technology and Electrical Engineering, University of Queensland, Brisbane, Australia

Correspondence: Neil W. Bergmann, School of Information Technology and Electrical Engineering, University of Queensland, Brisbane Q 4072, Australia. Tel: 61-7-3365-1182. Email: n.bergmann@itee.uq.edu.au

Received: January 23, 2013 Accepted: February 20, 2013 Online Published: February 26, 2013

doi:10.5539/mas.v7n3p64 URL: http://dx.doi.org/10.5539/mas.v7n3p64

Abstract

The performance of programmable logic controllers is often constrained by the microprocessor and the real-time firmware of the controller. Field programmable gate arrays (FPGAs) are an attractive potential implementation medium for high-speed control because of their fast and parallel execution and programmable nature. Ladder Diagrams are a standard graphical programming method for industrial controllers, but compilers from Ladder Diagrams to FPGA hardware do not yet exist. This paper explores the comparative speed of four different classes of FGPA implementation of Ladder Diagrams - Interpreted Software, Compiled Software, Interpreted Hardware and Compiled Hardware. It also explores parallel versus serial execution of Ladder Diagrams in hardware, and identifies timers as a major resource user in parallel implementations. Overall, a Shared Timer Serial Compiled Hardware system for FPGA implementation of Ladder Diagrams is recommended. Using comparable FPGA resources to other alternatives it provides a 20-600 times speed improvement over other solutions whilst maintaining correct Ladder Diagram semantics.

Keywords: FPGA, Programmable Logic Controller, ladder diagram, control systems

1. Introduction

Before the introduction of programmable controllers, relays were commonly used to implement control systems. With the advent of low-cost microprocessors, Programmable Logic Controllers (PLCs) replaced relays as the predominant control element (Bolton, 2009). As part of their heritage as relay replacements, PLCs are often programmed using a technique originally designed for relay networks - Ladder Diagrams (LD).

FPGAs (Field Programmable Gate Arrays) allow the design of programmable hardware, which theoretically should have much higher execution rates for logic-based control algorithms compared to software implementations. Furthermore, the parallel execution implied by Ladder Diagrams appears to be well-matched to the parallelism inherent in FPGA-based logic circuits.

There has been little published on the direct implementation of Ladder Diagrams control algorithms in FPGAs, and so this paper undertakes an initial investigation into the advantages and disadvantages of FPGA-based implementations. It does this by comparing a number of different LD implementation styles and analysing their relative performance when implemented on FPGAs.

1.1 Ladder Diagrams

Ladder Diagrams are a commonly used method for describing industrial control applications. An international standard, IEC 61131.3 (IEC, 2003) provides a standardised method for drawing and interpreting Ladder Diagrams. A brief introduction to LD semantics is described here, along with a short review of previous implementations of LD using reconfigurable devices.

A Ladder Diagram consists of "rungs", each of which evaluates a logic equation. The inputs to the equation are either inputs to the system or internal logic states. Similarly, the output of each logic equation may be either an output of the system or an internal state. The equation inputs are indicated by two vertical parallel lines and may be inverted by adding a diagonal line between the lines. The equation output is usually indicated by a circle or a pair of braces. And operations are formed by connecting two inputs adjacent to each other, in a similar fashion to drawing circuit elements in series. Or operations are formed by connecting two inputs one above the other, in a similar fashion to drawing circuit elements in parallel.

An example of three rungs is shown in Figure 1. In this example, internal state I2 is assigned the value of input

I1, output O1 is assigned the value of I2 AND (NOT I3) and output O2 is assigned the value of ((NOT I4) OR I6) AND I5.

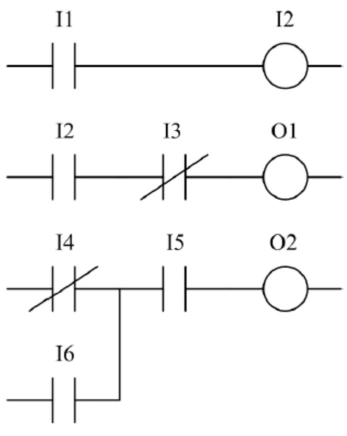

Figure 1. Ladder diagram example

Using this notation, complex sets of logic equations can be designed. Ladder Diagrams have now moved from being a language for describing relay-based logic, to an input design method for sequential programmable controllers. As such, the semantics of modern Ladder Diagrams reflects implementation on a sequential machine. Rungs are implicitly or explicitly numbered from top to bottom, and executed in order, and then the sequence is repeated continuously. This sequence of steps in evaluating logic is termed a system cycle. Sequential evaluation implies that the semantics of a Ladder Diagram are dependent on the order of the rungs. This is exactly opposite to conventional logic circuits, where combinational logic circuits are all executed in parallel, and outputs only updated at the end of a clock cycle. This "unnatural" semantics for execution of rungs introduces some difficulties when implementing rungs directly in digital logic, as explained later.

Apart from simple logic, additional functionality is required for practical applications. Among the most common additions are timers, which allow timed sequencing of events. A timer is defined by three components: a start condition, an output state and the duration of the timer. A timer is started when its start condition is set to true. The timer continues to run while the start condition is satisfied, until its (preset) duration has elapsed. Once this occurs, the timer is said to be expired, the output state of the timer is set to true and remains true until the start condition becomes false again. If at any point while the timer is running the start condition becomes false, the timer stops running and is reset.

In this project, we investigate an example LD application, which uses logic and timers, and investigate different methods for implementing the LD on an FPGA.

1.2 Why Ladder Logic on FPGAs?

The processors used in modern industrial controllers are usually general-purpose microprocessors. Modern processor design techniques such as pipelines and caches tend to improve average performance rather than worst-case performance which is often the parameter required for real-time systems. Processors are thus often over-specified to ensure that they are capable of meeting timing requirements, and so increasing the cost of the system. Reconfigurable logic can be tailored to a particular application, potentially resulting in a higher-performance system. The logic in an FPGA appears well suited to the binary operations of control logic.

Safety-critical systems often require multiple processing channels in order to provide assurance of correct operation. These processing channels may be either diverse (in order to protect against design faults) or redundant (in order to protect against hardware failures). Traditionally, this has required either a single processor performing the same computations multiple times (thus reducing the performance of the processor), or multiple processors operating in parallel (thus increasing the cost of the system). FPGAs allow multiple different computational cores to operate in parallel on the same chip, thus allowing diverse and/or redundant implementations to run on a single device.

1.3 Previous Works

Ladder Diagrams have been implemented in reconfigurable technology before, although relatively few implementations have been reported. Welch and Carletta (2000) have described a specialised FPGA fabric custom-designed for implementing ladder logic. This fabric simplifies the mapping and routing of a design and eliminates the need for translation of logic into alternative forms, thus simplifying and speeding up development using these devices. The fabric specifies a method for implementing the LD, but does not include a description of the Input/Output (I/O) connections of the device. At present, no devices utilizing this fabric have been manufactured.

Miyazawa et al. (1999) show how Ladder Diagrams could be implemented using translation to VHDL (VHSIC Hardware Description Langugae, where VHSIC is Very High Speed Integrated Circuit). The description shows how the semantics of sequential execution could be modeled by using either clock events or instantiation of flip-flops in VHDL. The code produced is simply a translation of the logic equations into VHDL. There is no analysis of the efficiency of such VHDL-based implementations on FPGAs.

Silva et al. (2007) describe an implementation of industrial control logic targeted at Altera FPGAs (hardware implementation) or Simatic PLC (software implementation). Petri nets are used to describe the design rather than Ladder Diagrams, so the implementations are not directly comparable. These are then translated into either VHDL or software, as appropriate for the target.

Du et al. (2009) have implemented a system which converts Ladder Diagrams into VHDL, and their system is particularly noteworthy because it allows analysis of the ladder Diagram to identify rungs which can be executed in parallel. Their input language is not any standard LD format, but they use their own generic Boolean language format, enhanced with counters and timers. The output of their system is a VHDL program - they have not investigated the efficiency and performance of the LD controllers when implemented on a real FPGA.

Ichikawa et al. (2011) have similarly produced a system which converts Ladder Diagrams into VHDL. They examine sequential design (one cycle per rung), levelised design (non-dependent rungs executed in parallel) and flat designs (where rungs depend only on input variables, and all can be executed in parallel). They have implemented a system for converting PLC programs for a particular Mitsubishi PLC into an FPGA implementation using their own FPGA board as a target. They showed similar hardware speedups to the work described in our project (one clock cycle per rung for sequential designs). However, it is not clear that they have implemented timers, which are a particularly resource-hungry construct, and their work does not target standard LD languages.

All of the current implementations of LD to FPGA implementations described above convert to a compiled hardware implementation, based on a VHDL circuit. If timers are implemented at all, then they are implemented with one physical timer for each logical timer. This work investigates some new approaches which extend this previous work. Firstly, it investigates both a compiled hardware version, and also an interpreted hardware implementation which provides a different speed-area tradeoff. Secondly, timers are identified as the major consumers of hardware resources for compiled implementations, and a shared timer implementation is investigated.

1.4 Project Objectives

To date, there has been limited investigation of the suitability of FPGAs as an implementation target for Ladder Diagrams, and this project extends this previous work by investigating a wider selection of FPGA-based implementation choices. The goal of this work is to investigate how the software based implementation of Ladder Diagrams, currently used in a range of industrial controllers, compares to implementations in reconfigurable logic. In order to do this, the semantics of the existing LD applications should be maintained so that the consistency of the system can be assured. This initial work explores several design options, investigating multiple design methods. This is both to explore these different options individually, and also to investigate the practicality of different diverse implementations on a single FPGA, as might be required in a safety-critical application.

2. Materials and Methods

Modern FPGAs allow designs to be implemented in a range of different ways, including software running on softcore processors. Implementations of the Boolean equations typical of industrial control applications designed using Ladder Diagrams can fit into two broad categories: interpreted and compiled. Interpreted implementations maintain the same base software or hardware across all applications, with logic equations specific to the application being interpreted by that base platform as a sequence of instructions or byte codes. This approach allows the interpreter to be extensively tested and verified to ensure correctness and safety. In contrast, compiled implementations evaluate the logic directly, with different hardware or software being generated for each application by a compiler. This approach is likely to offer improved performance, but does require the compilation process itself to be extensively tested and verified - a process which presents its own challenges.

Four different Ladder Diagram implementation styles have been developed and analysed, based on a typical set of simple LD equations. The equations were developed using an industry standard textual format which could be generated from a Ladder Diagram editing system. This format will be referred to as ORIG format in the remainder of this paper. All the systems use a simple communications application running on a host PC to display the value of all states, as well as accept inputs from the user. This application replicates the functionality of the I/O modules of a real industrial controller. In order to simplify communication with this application, all the systems use a Microblaze soft processor for I/O control in these experiments (Xilinx, 2013). Hardware implementations run as peripherals of this processor, while software implementations evaluate the logic directly on the Microblaze processor. These four different implementation styles are now described.

2.1 Interpreted Hardware

The most complex of the implementations, the Interpreted HW system is essentially a custom processor dedicated to evaluating Ladder Diagrams. The hardware has been designed in a modular fashion using a network architecture, with four modules (as well as a Microblaze processor) communicating over a common network structure, as shown in Figure 2.

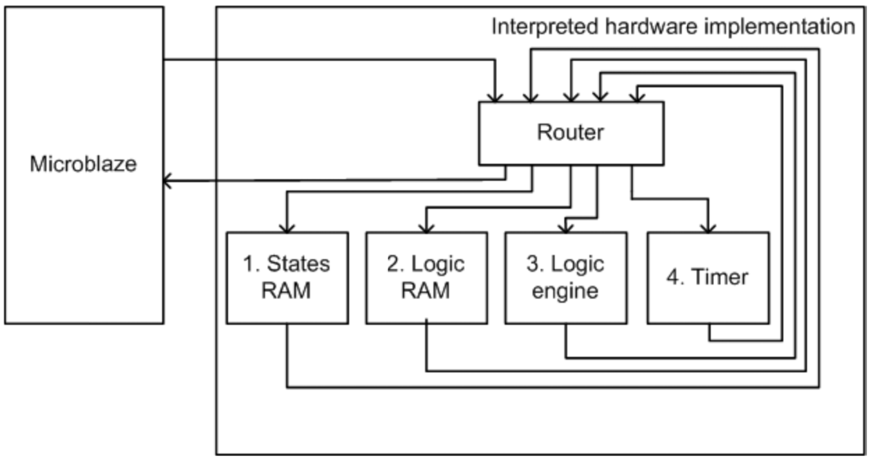

Figure 2. Hardware interpretation engine

This modular design assists in enabling extensive testing of individual components of the system, as well as allowing the addition of further modules as required in the future. Each of the modules is capable of issuing and receiving instructions from each other. Instructions are a uniform 32-bit format. The modules are as follows:

2.1.1 States RAM

This module stores the values of each of the states used by the system. Each state corresponds to either an input, output or an internal logic state. The module operates in a simple manner, handling read and write operations (*ReadReq* and *WriteState* respectively). It responds to *ReadReq* instructions by issuing *Notify* instructions. *WriteState* instructions update the value of the state stored.

2.2.2 Logic RAM

This module stores the instructions in the Ladder Diagram to be evaluated. Once initialized and started, the module continuously loops through the logic, requesting the value of states from the States RAM (using *ReadReq* instructions) and issuing *EvalLogic* instructions to the Logic engine. The logic is therefore evaluated sequentially rung by rung.

2.2.3 Logic Engine

This module performs the actual evaluation of each rung. The values of states are stored in latches, each of which is fed to appropriate logic to evaluate the rung. *EvalLogic* instructions specify a position within the engine structure and a logic value for that position. The output state is itself a position within the engine. Once an output state has been specified, the module issues a *WriteState* instruction containing the logic output of the rung.

2.2.4 Timer

This module implements timers. Each timer is defined by 4 characteristics: start condition, output condition, duration and current status (stopped, running or expired). Once started, the module continuously loops through the timers, requesting the value of each start condition. If the start condition changes to be true, the status of the timer is changed to running and an end time is calculated. Once the timer duration has elapsed, a *WriteState* instruction is issued, updating the output condition to true, provided that the start condition has remained true. In a similar manner, if the start condition becomes false, the status is updated to being stopped and a *WriteState* instruction is issued, setting the output condition to false.

The modules are connected via an on-chip network. All instructions between modules (and the Microblaze processor) are routed through this network. The network consists of a simple router which contains a single register containing the instruction to be routed, along with suitable multiplexers and logic for each module. Based on the address specified within the instruction, the data is directed to the buffer of the appropriate module.

2.3 Interpreted Software

The implementation of an Interpreted SW program for Ladder Diagrams on the Microblaze is relatively straightforward. A standard Microblaze system architecture is used, consisting of processor, timer, UART and memory controller cores. The software application makes use of the ORIG format logic as generated by the original tools and no translation is required. The application is written in C, which is well supported by the Microblaze tools. Execution is optimized by only evaluating a rung where necessary. The implementation of timers is implemented using the standard Microblaze timer peripheral core. It is set to expire at a regular period, equal to the interval of the timers in the application logic. On expiry, an interrupt handler updates the status of the timers appropriately.

2.4 Compiled Hardware

The system architecture for the Compiled HW implementation of Ladder Diagrams is shown in Figure 3. In this diagram, the Microblaze and logic wrapper are static while all other components are generated based on the specific Ladder Diagram.

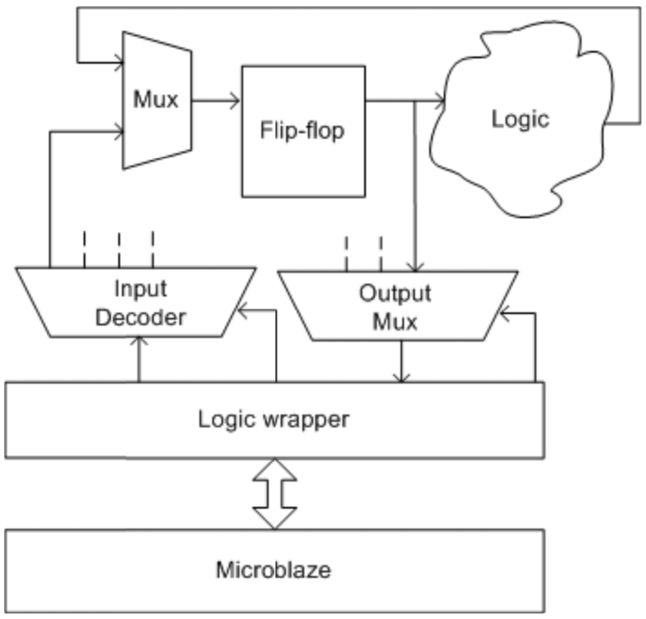

Figure 3. Compiled hardware architecture

The sizes of the input decoder and output multiplexer are dependent on the number of states present in the system. For each state, there is a set of 3 components as shown at the top of the diagram. The flip-flop stores the current value of the state, while the logic evaluates the next state. The logic may take input from any number of other states in order to generate the next state logic. This is fed through a multiplexer to the flip-flop inputs. The multiplexer allows the state to be written by the Microblaze. A software application on the host PC takes the ORIG format logic from the industry tools and generates VHDL containing suitable logic equations for that particular application. The multiplexer and flip-flops are generated using a for-generate statement in VHDL, dependent on the number of states required. The logic is generated for each state in the system, resulting in a sequence of statements of the form:

```
LogicOut(x) <= FlipFlopOut(y) AND FlipFlopOut(z);
```

Each of these statements corresponds to one rung in the LD program. Therefore each of the rungs are evaluated in parallel and latched into the flip-flops simultaneously. This no longer maintains sequential execution of rungs as found in the interpreted implementations described previously. Logic calculations which span several rungs will take several cycles to be resolved. Without care, there is a chance that parallel execution of rungs may lead to inconsistent or incorrect behaviour, however in many cases it will lead to enhanced performance, since all rungs are evaluated in a single clock cycle. We have called this non-compliant implementation the Parallel Compiled Hardware System.

It is also possible to build a compiled hardware implementation which executes rungs sequentially, and preserves the correct semantics. This is done by gating each of the output flip-flops with an enable bit which cycles around a ring counter (one bit per rung). This then executes rungs sequentially, and a system cycle for N rungs is N clock cycles. This is called the Serial Compiled Hardware System.

With additional effort, it would be possible to develop a CAD tool to group consecutive rungs into sets of independent rungs which could be executed in parallel, gaining the benefits of some parallel execution while preserving correct semantics, however developing such a tool is outside of the scope of this initial work.

In addition to the direct logic equations of most rungs, timers are implemented by replacing the logic with a timer component. The component specifies the information required for the timer to operate correctly. An example of this is:

```
timer16 : timer
generic map (
DURATION => 40000)
port map (
clk => clk,
rst => rst,
start => FlipFlopOut(75),
expiry => LogicOut(76));
```

During the development and testing of larger Compiled HW circuits, it was observed that a large proportion of the FPGA hardware (more than 80%) was consumed by timers. Each timer requires several 32-bit registers plus associated logic. By comparison, each logic rung may require just a few single-bit logic gates. Therefore, an additional version of the compiled hardware system was designed for circuits with a large number of timers, which uses a single timer module to implement all of the system timers. The timer module has individual expiry registers for each timer stored in a memory block, but shares the same time-count register and expiry-check logic. One logical timer can be checked each clock cycle. For parallel rung execution, this considerably increases the system cycle time, however with serial rung execution, the timer execution can be overlapped with rung execution and does not increase the execution time. Use of a shared timer gives two extra hardware variants: Shared Timer Parallel Compiled Hardware and Shared Timer Serial Compiled Hardware.

2.5 Compiled Software

The ladder logic equations may be solved by a sequence of logic statements in a high-level programming language rather than the interpreted form presented above. Rather than a software application interpreting the logic, the ORIG format logic can be used to generate code for compilation which directly evaluates the logic equations.

A large part of the application code from the Interpreted SW implementation has been reused, with communications code remaining the same. The logic itself is generated by a separate application on the host PC and generates a C source file consisting of a large set of equations of the form:

```
logic[x] = (logic[y] & logic[z]) || !logic[q];
```

This source file then forms part of the application running on the Microblaze processor. This implementation allows the logic to be executed more quickly than the interpreted form and maintains identical semantics as the rungs are evaluated in order in the same way as the interpreted implementation. Timers are generated in a separate C source file. The information required is the initialisation details, that is, the start condition, output condition and duration of each timer. Once initialised, the timers run in the same manner as the Interpreted SW, with an interrupt routine triggered by a timer interrupt updating the timer status.

3. Results

The performance of the various FPGA implementations of the same Ladder Diagram were explored by implementing a simple example program consisting 18 ladder rungs of various types and 3 timers. This size application was chosen in order to simplify the process of verifying that each implementation was correctly evaluating the logic. Each logic circuit was compiled onto a Xilinx Spartan 3E-1600 FPGA. Each implementation uses a Microblaze processor to provide data input and output, and any hardware-based logic used the same clock as the Microblaze, which typically runs at 50 MHz. In all cases, correct operation of the systems was confirmed for a variety of different input combinations.

3.1 Speed

The numbers of clock cycles taken by each implementation in evaluating the 18-rung application logic are shown in Table 1. These times exclude external communications overheads which are relatively constant across implementations. In most cases, the execution time varies with the input values. For some combinations of inputs, the software is able to optimize execution and thus evaluate the logic more quickly. The Interpreted HW implementation takes varying amounts of time due to communications interference. The execution time variation was not large – the minimum execution time was around 85-90% of the worst-case time. The table shows the worst-case execution time that was recorded for the example data sets.

Table 1. Execution times and resource use – simple example (18 rungs)

Implementation	Max. Execution Time (Clock Cycles)	FPGA Resource Use (Slices)
Interpreted SW	12819	2629
Interpreted HW	611	3024
Compiled SW	891	2629
Parallel Compiled HW	2	2731
Serial Compiled HW	19	2740

As expected, an interpreted approach is slower than a compiled approach. The significance of this overhead relative to the require system cycle time determines whether the interpreted approach is still an attractive option. As can be seen, the Interpreted SW is an order of magnitude slower than the Compiled SW system, while the Interpreted HW implementation is similar to the Compiled SW.

For the Parallel Compiled HW system, equations are evaluated in parallel in a single clock cycle, plus one additional cycle for logic value I/O. Communications overheads will most likely be far larger than the logic execution times in this case. For the Serial Compiled HW, each rung executes in one clock cycle, plus one additional cycle for I/O.

3.2 FPGA Resource Use

For each implementation, the hardware resources used on the FPGA for the simple 18-rung system were measured and the results are also shown in Table 1. FPGA resource use is measured in slices, where a slice is a basic FPGA logic unit consisting of two 4-input LUTs (Look Up Tables) and two flip-flops. The Xilinx Spartan 3E is a low-cost, low-end FPGA, and provides 14,752 slices in total. The software implementations represent the resource use of the Microblaze processor. The hardware implementations represent the resource use of the hardware co-processor (excluding the Microblaze). For the hardware implementations, the Microblaze only acts as an I/O processor; in a practical circuit there would be specialised additional I/O hardware to connect to control inputs and outputs.

Each of the interpreted implementations requires some form of storage for the logic equations. In the systems designed, this has been stored on the on-chip FPGA memory, but for larger examples this would require storage in external memory.

3.3 Larger Designs

For the Interpreted SW, Interpreted HW and Compiled SW implementations, the hardware cost is largely independent of the complexity of the Ladder Diagram size, and the execution time grows approximately linearly with the number of rungs. All of these implementations store the Ladder Diagram in memory, either as compiled processor code, or as Ladder Diagram byte codes for interpretation. The required memory quickly outgrows the available on-chip FPGA memory and an external memory system is required, with both cost and performance implications.

For the Compiled HW implementation, the size of the execution hardware depends strongly on the size of the Ladder Diagram. To measure the growth in size of the compiled hardware, a much larger system was compiled for this target - a railway interlocking application consisting of 1451 rungs and 246 timers. A more modern FPGA (Virtex4-LX25) was used as the potential target for these later experiments. It was found that this LD would use 113% of the available resources on the FPGA (even without the Microblaze I/O controller) for the Parallel Compiled HW version. The Serial Compiled HW version would require 118% of the available resources. For this reason, designs with a shared timer architecture, as explained earlier, were also investigated. Performance figures are given in Table 2, and these are are estimates based on the results of circuit compilation, and simulation-based testing. Several of the circuits were too large to fit on the target FPGA, so detailed testing of downloaded designs was not possible.

As well as the area and speed, an area-time product measurement is given in Table 2 which gives some measure of the relative computational efficiency of the implementations. The Parallel Compiled HW is by far the most efficient, because all rungs are executed every second cycle, however correct Ladder Diagram semantics is not maintained.

Table 2. Execution times and resource use - complex example (1451 rungs, 246 timers)

Implementation	Max. Execution Time (Clock Cycles)	FPGA Resource Use (Slices)	Area-Time (Slices x Cycles)/1000
Parallel Compiled HW	2	12,177	24
Serial Compiled HW	1452	12,903	18735
Parallel Compiled HW (Shared Timer)	246	3,419	844
Serial Compiled HW (Shared Timer)	1452	4,145	6018

4. Discussion

Existing Programmable Logic Controller systems use a combination of Interpreted SW and Compiled SW implementations. Indeed some safety critical systems use diverse implementations of both techniques for internal consistency checking. The aim of this research has been to determine the potential advantages of FPGAs as an alternate implementation strategy.

FPGA-based softcore processors already allow conventional software-based implementations of Ladder Diagrams to be used. There is no performance benefit here compared to conventional microprocessors - in fact softcore processors usually have a clock rate 2-5 times slower than conventional microprocessors. However, if an FPGA is already being used for hardware tasks such as I/O control, then the PLC processor can also be included on the FPGA to reduce chip count.

The more interesting FPGA-based architectures are the hardware-based implementations. The Interpreted HW architecture consumes similar FPGA resources to a softcore processor, and provides a 20 times speed-up compared to an interpreted SW architecture. The Compiled SW implementation provides similar performance to the Interpreted HW, and is probably a simpler and more portable solution. On a conventional microprocessor (rather than an FPGA softcore), the compiled software is likely to be faster. The design of the HW interpreter is also the most complex of all the options. The Interpreted HW implementation is probably only useful when a faster interpreted solution is required, either because of available Ladder Diagram design tools, or because of the need for multiple diverse solutions.

The compiled HW solution is the most interesting. For extremely high speed controllers, the Parallel Compiled HW can execute a complete system cycle in two clock cycles (even one cycle with pipelined I/O design). However, the fact that the system execution involves a different semantic model to standard sequential execution means that its use would be restricted to very specialised applications which require very high speed. It cannot be really considered as a legitimate Ladder Diagram implementation method. Additionally, timers require very large circuit area, and moving to a Shared Timer Parallel HW model greatly increases system cycle time.

For standard Ladder Diagram execution, a Serial Compiled HW implementation style is needed. Here, our investigations show that for practical Ladder Diagram implementations, area is dominated by timers. For this reason, the Shared Timer Serial Compiled HW provides a significant reduction in area (70% reduction compared to parallel timers in this example), with no decrease in performance since timer evaluation is overlapped with sequential rung execution.

Based on the simple 18-rung example, the Shared Timer Serial Compiled Hardware option is estimated to provide a performance speedup of 30 times compared to Interpreted HW, 40 times compared to Compiled SW and a 600 times improvement compared to Interpreted HW. The FPGA resource use of the Shared Timer Serial Compiled Hardware for the large 1451 rung example is similar to the hardware cost of the Microblaze software processor or the Interpreted HW engine, and should be a practical solution for many applications.

5. Conclusions

FPGAs have been shown to be an interesting implementation target for Ladder Diagrams, however they are not without their problems.

An Interpreted HW engine provides better performance than an Interpreted SW solution, but provides little benefit over a Compiled SW solution, and is probably only useful when a faster interpreted solution is required.

A Compiled HW solution is an interesting option but suffers from a mismatch between Ladder Diagram sequential semantics and FPGA logic parallel execution. A Serial Compiled HW system is needed to preserve Ladder Diagram semantics, but this then means that only one rung is active at a time. Both circuit size and system cycle time increase with the number of rungs. The use of multiple timers greatly increases circuit size, and so a shared timer architecture is prefered. For serial execution, the time for sequential evaluation of timers can be hidden within the sequential rung execution time. A potentially interesting area of future work is automatic identification of rungs that can be executed in parallel, which could significantly reduce execution time while preserving Ladder Diagram semantics.

Overall a Shared Timer Serial Compiled HW implementation model appears to offer the most interesting way forward, and will allow the design of complex Ladder Diagrams with considerably lower system cycle times, in the sub 1ms range. This would enable Ladder Diagrams to be used to program control systems in a wider range of high-speed control applications.

Acknowledgements

This work was supported by the Australian Research Council's Linkage Project LP0668067.

References

Bolton, W. (2009). *Programmable Logic Controllers* (5th ed.). Burlington MA: Newnes.

Du, D., Liu, Y., Guo, X., Yamazaki, K., & Fujishima, M. (2009). Study on LD-VHDL conversion for FPGA-based PLC implementation. *International Journal of Advanced Manufacturing Technology, 40*(11-12), 1181-1190. http://dx.doi.org/10.1007/s00170-008-1426-4

Ichikawa, S., Akinaka, M., Hata, H., Ikeda, R., & Yamamoto, H. (2011). An FPGA implementation of hard-wired sequence control system based on PLC software. *IEEJ Transactions on Electrical and Electronic Engineering, 6*(4), 367-375. http://dx.doi.org/10.1002/tee.20670

IEC: International Electrotechnical Commission. (2003). *International Standard 61131.3: Programmable Controllers – Part 3: Programming languages for programmable controllers*. Geneva: IEC.

Miyazawa, I., Nagao, T., Fukagawa, M., Itoh, Y., Mizuya, T., & Sekiguchi, T. (1999). Implementation of ladder diagram for programmable controller using FPGA. *Proceedings of 7th IEEE International Conference on Emerging Technologies and Factory Automation*, pp. 1381-1385. http://dx.doi.org/10.1109/ETFA.1999.813150

Silva, C. F., Quintans, C., Mandado, E., & Castro, M. A (2007). Methodology to implement logic controllers with both reconfigurable and programmable hardware. *Proceedings of the IEEE International Symposium* *on Industrial Electronics*, pp. 324-328. http://dx.doi.org/10.1109/ISIE.2007.4374620

Welch, J. T., & Carletta, J. (2000). A direct mapping FPGA architecture for industrial process control applications. *Proceedings of the International Conference on Computer Design*, pp. 595-598, http://dx.doi.org/10.1109/ICCD.2000.878352

Xilinx Inc. (2013). *MicroBlaze Soft Processor Core*. Retrieved January 23, 2013, from http://www.xilinx.com/tools/microblaze.htm