VLSI DESIGN
1996, Vol. 4, No. 2, pp. 135–139
Reprints available directly from the publisher
Photocopying permitted by license only

The Use of Field-Programmable Gate Arrays for the Hardware Acceleration of Design Automation Tasks

NEIL J. HOWARD, ANDREW M. TYRRELL and NIGEL M. ALLINSON*

Department of Electronics, University of York, York YO1 5DD, UK.

This paper investigates the possibility of using Field-Programmable Gate Arrays (FPGAs) as reconfigurable co-processors for workstations to produce moderate speedups for most tasks in the design process, resulting in a worthwhile overall design process speedup at low cost and allowing algorithm upgrades with no hardware modification. The use of FPGAs as hardware accelerators is reviewed and then achievable speedups are predicted for logic simulation and VLSI design rule checking tasks for various FPGA co-processor arrangements.

1 INTRODUCTION

Many special hardware acceleration engines have been proposed or built [1], with significant speedups over software. However, the VLSI design process is characterized by *many* computer-intensive stages that are quite different in nature. The use of an accelerator for one stage does not dramatically improve the productivity of the overall design process and those accelerators in use serve to allow substantially more logic and fault simulation rather than by cutting design times. The continual development of algorithms also counts against hardwired accelerators. Virtually all design automation tasks are still performed solely on general purpose computers, particularly workstations.

The development of static RAM based Field-Programmable Gate Arrays means that *reconfigurable* co-processors can now be realized that could accommodate the diversity and dynamic nature of design automation tasks. The speedup achieved for any one task will be less than that for a complex dedicated hardware accelerator but, applied to speed up tasks throughout the design process, it may provide a substantial benefit for a *very low* hardware cost. Also, algorithms could be upgraded without hardware modification. This paper investigates this possibility.

The technique is not without its drawbacks. Section 2 reviews the use of FPGAs as hardware accelerators, arguing that these drawbacks do not apply to the acceleration of design automation tasks. Section 3 then demonstrates what speedups are achievable using two applications as examples.

2 FPGAS AS HARDWARE ACCELERATORS

FPGAs [2] are a recent development of programmable logic, with much higher logic and register capacities than programmable logic arrays (PLAs). FPGAs gen-

*Corresponding author. Current address: Department of Electrical Engineering and Electronics, UMIST, Manchester, M60 1QD, UK.

erally contain a two-dimensional array of cells which implement a small amount of logic with configurable routing between them, surrounded by input/output cells. Dramatically reduced lead times and set-up costs make FPGAs an attractive alternative to conventional low-density gate arrays. However, as a general rule of thumb, FPGAs can only achieve about 10% of the functionality and 30% of the speed of custom VLSI circuits of the same silicon area and fabrication technology, because of their configurability.

The design synthesis tools for FPGAs are considerably more complex than for PLAs and are similar to those for conventional gate arrays, involving mapping a netlist into cells, placement, routing and simulation.

An important development over previous generations of programmable logic is that, though most FPGAs use static RAM to define the device configuration rather than the familiar PROM-type fuses, and hence are *re*-configurable. Configurations are downloaded into the FPGAs on power-up from an external ROM or microprocessor in a few milliseconds. Here lies the potential of using FPGAs as configurable co-processors. A custom-designed configuration can be downloaded into the FPGA to assist in the processing of some specific application. The general implementation is as 'programmable active memories' [4]—the inner loops of the host's software are transferred to a hardware implementation and downloaded into the FPGA. The master processor writes operands to the memory-mapped FPGA and the results are read back. For each application, the FPGA provides the host with a limited number of customized 'meta-instructions'.

Applications to benefit most from this acceleration technique are repetitive, decomposable to small elements (in order to fit inside a single FPGA or a small number of FPGAs; 'tight' software loops are required), non-mathematical (complex arithmetic operations and functions are better done on dedicated, floating-point, co-processors) and computationally irregular. Regarding this last criterion, the range of applications to be run on the FPGA must be diverse, otherwise more dedicated hardware is preferable; the flexibility that FPGA computation offers must be needed.

Applications such as specialized numerical applications (e.g. encryption), DSP and image processing, cellular automata and other systolic arrays and neural networks have been demonstrated on a number of FPGA machines [3–7], each with many FPGAs connected to banks of memory, with speedups of 2 to 3 orders of magnitude over software quoted. However, the programming of even trivial tasks on these machines is complex and the development of support tools is an important ongoing research subject. Eventually, the aim is to have a high-level description which can be compiled to produce an object code for the host and configuration files for the FPGA. This concept has been demonstrated by Athanas's [8] 'C' compiler which automatically identifies code to put into an FPGA and separates this to produce compiled code for the host and a netlist. The subsequent stages of placement and routing are extensive and ensuring that the FPGA configurations comply with the target implementation's timing is difficult to automate without sacrificing flexibility.

Without fully automated compilers, this technique of using FPGAs for hardware acceleration cannot be applied generally and is confined to a niche role. It is unlikely that two or more different applications which are suitable for this technique are actually *required* by one user on the same platform. The applications on their own are too regular so that full-custom devices are more appropriate as, for example, with the many DSP chips.

However, design automation demands a set of *different* tasks to be performed by the same user, all amenable to FPGA acceleration. The FPGAs coprocessor could be used to implement many different customized co-processors on the single hardware device. The widespread use of third-party design tools means that the presence of FPGAs accelerators can be transparent to the user. The partitioning of accelerated tasks between hardware and software, the design of both and the layout of the hardware design on the FPGA are all the responsibility of the tool vendors and would have to be done (and optimized) manually.

3 FPGA ACCELERATION OF VLSI DESIGN TOOLS

It is assumed that the co-processor is in close proximity to the main processor, sharing the data/address busses and hence memory of the processor, rather than fitting the co-processor on a standard extension/interface port of a workstation. This might be achieved by retro-fitting (having the processor and co-processor on a small board or multi-chip module that plugs into the workstation's processor socket) or it might be worthwhile providing space on a workstation's processor board for such a co-processor. The hardware costs for this technique will be very low compared to other accelerators and the upgradability advantages of software will be retained.

The processor and co-processor can interact with one another in a variety of manners; in increasing order of complexity:-

Passive Indirect: The FPGAs are memory mapped into the master processor's address space. Data is transferred between the master's registers and those within the FPGAs. Data transfers between memory and the FPGAs must go via the master processor;

Passive Direct: As above except that memory to FPGAs data transfers can be done within a single memory cycle;

DMA: The FPGAs is used with a co-processor compatible master which can relinquish bus control for the FPGAs to make direct memory accesses;

Autonomous: The FPGAs implements a customizable processor which can take over from the master to execute whole sections of processing-intensive code. Instructions for this processor are stored in main memory but the customized instructions will be complex and typically run for many memory cycles.

Of the many design automation tasks, design rule checking and simulation are considered here as their algorithms are generally simpler and less specialized than those of the other tasks. Both applications are considered for the various co-processor arrangements with calculated execution times compared to that for software with no co-processor.

3.1 Design Rule Checking

The case considered here employs the 'scan-line' technique [9] in which the polygons are pre-sorted into 'bins' depending on their x-coordinate range and, within each bin, polygons are pre-sorted into ascending order of their lower y-coordinate. Polygons may be angled at 45°.

In order to reduce the number of comparisons of polygons to check minimum distances etc., the algorithm scans through bins vertically upwards with a bar (scan-line) of height y_{min}. At each position of the scan-line, the algorithm maintains a list of those polygons which overlap with the scan-line, deleting and appending polygons as the scan-line shifts. Only those polygons within the scan-line are compared with one another for checking a minimum distance of y_{min}.

Primitive operations required include testing whether two polygons overlap, abutt, are electrically connected, or are separated by a required minimum distance. The operation considered here for timing comparison is the stepping through a scan-line list to detect polygons that overlap with a reference polygon. For the hardware accelerated case, polygon records are packed into two 32-bit words. A reference polygon is loaded into the FPGA first and then, for each polygon in the scan-line list, the record is loaded and a flag indicating the 'overlap' test result can be read. For more complex co-processor arrangements, the stepping through the list and stopping at an overlapping polygon or the end of the list is handled by the FPGA rather than the master.

3.2 Logic Timing Simulation

The example considered here is an event-driven compiled-code logic simulator [10] which is extended to include timing [11]. To summarise, this technique:-

1. only calculates the outputs of blocks which have inputs changing (events) at the current simulation time, rather than updating every subcircuit in the design. If the outputs change, new events will be generated which will propagate the change to all fan-out blocks. These will be updated at a subsequent simulation time;
2. maintains many lists of output events—one for each time step. Output events are appended to event lists depending on the block and net propagation delays;
3. avoids the significant time overhead of retrieving data from complex data structures by compiling code before running the simulation. The data is contained within the compiled code that is executed during simulation.

The examples assume a rich set of primitive functions, with multiple logic levels and driving strengths and different rising and falling transition times. Execution times considered are for updating a 2-input gate and a full adder, generating a new event if required and inserting it into the correct event list. A block diagram for a DMA co-processor is shown in Figure 1. Other co-processor arrangements are modifications of this.

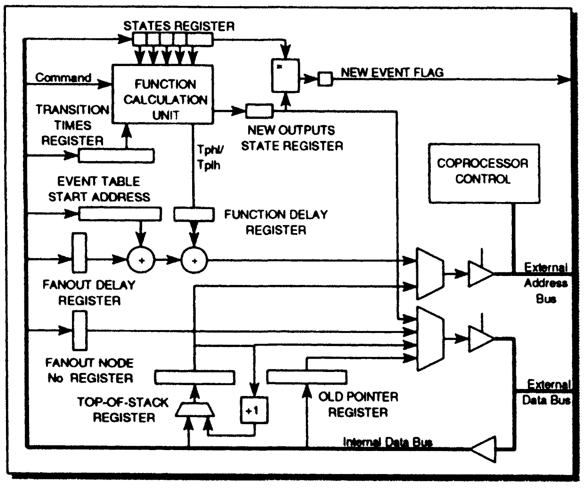

FIGURE 1 Simulation Co-processor Configuration Block Diagram

3.3 Speedup Results

Speedup results are obtained by comparing hand-crafted assembly code of hardware-assisted and normal software for the core operations described above. The figures will indicate an upper limit of the speedup achievable for the application as a whole. Subsidiary tasks (such as the compilation stage for simulation, or scan-line shifting for DRC) may also be accelerated to get the application speedup close to the speedup figures presented here.

Speedups have been calculated from the average number of memory accesses (instruction fetch words and data transfers) for a MC68020 processor and are given in Table I. The low speedups are surprisingly poor and can largely be attributed to the way that in software:-

1. many processing operations incur no overhead and wholely overlap with memory accesses;
2. look-up tables can be used very effectively (for simulation);
3. not all operands need to be accessed to determine that two polygons do not overlap (for DRC) whereas all operands are required for a hardware comparison.

Athanas [8] claims a raw speedup for a logic simulation engine instruction '*LogicEv*' on an FPGA computer of 18, but a very different result of only 1.3 is achieved here where real simulation problems have been considered *and compared with optimized software*. Such high speedups have not even been achieved with the most efficient processor-FPGA arrangements.

The number of FPGA cells required to implement a suite of design automation tasks has not been considered. However, as an example of the quite low functionality of FPGAs, the core of the 'overlap' test for

TABLE I Speedup Factors for Various Implementations

Operation	Passive Indirect	Passive Direct	DMA	Autonomous
DRC Overlap search	1.5	1.6	7.7	7.7
2-input gate simulation	1.3	1.4	3.4	4.2
Full adder simulation	1.7	1.7	4.9	6.0

DRC, consisting of 15 adders and 10 comparators consumes 51% of one of the largest current FPGAs-,the XC3090 [2]. A number of FPGAs would be required to form a reconfigurable array of sufficient size. Timing simulations on this 'overlap' function indicate worst-case propagation delays of between 25 and 100uns, depending on the speed grade of the FPGA, figures that are commensurate with the access times of memory. With careful design, wait states can be avoided.

The speedup results demonstrate that the simplest FPGA co-processor arrangements are ineffective and that only moderate speedups are achievable with more complex architectures. Design automation tasks may involve relatively simple operations amenable to FPGA implementation but the sheer volume of data movements prevents any substantial speedup on a single data stream system. Further improvements would require separate memories for the FPGA co-processor with the associated drawback of inflexibility.

4 CONCLUSIONS

This paper has discussed the suitability of using FPGAs for accelerating design automation tasks in order achieve moderate speedups throughout the design process in a cost-effective and flexible manner. This is achieved by reconfiguring FPGAs to be used as co-processors that have been optimized for the current task in hand.

Various architectures have been considered, all of which implement co-processors that share the master processor's memory in order to retain most of the flexibility of software and incur only a low cost. Speedup results show the importance of the relationship between the FPGA and the master processor. The simplest arrangements are ineffective and only modest speedups (of about 5) have been achieved with the more complex processor/FPGA interfaces.

Whilst it is expected that this technique *can* provide a level of performance for a much smaller amount of hardware than by using conventional parallel techniques, the desriability and feasibility of implementing this technique in modern workstations remains to be demonstrated.

Acknowledgements

This work has been funded by the UK Science and Engineering Research Council and Defence Research Agency, grant ref. GR/ F92152.

References

[1] T. Blank, 'A Survey of Hardware Accelerators Used in Computer-Aided Design', IEEE *Design & Test*, pp. 21–39, Aug. 1984.

[2] 'The Programmable Gate Array Data Book', Xilinx Inc., San Jose Ca., 1991.

[3] T. A. Kean, 'Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation', PhD Thesis, Computer Science Dept., Edinburgh Univ., Dec. 1989.

[4] P. Bertin *et al*, 'Introduction to Programmable Active Memories', from *Systolic Array Processors*, eds. J. McCanny *et al*, pp. 301–309, Prentice-Hall, 1989.

[5] M. Gokhale *et al*, 'Building and Using a Highly Parallel Programmable Logic Array', IEEE *Computer*, pp. 81–89, Jan. 1991.

[6] C. E. Cox and W. E. Blanz, 'GANGLION—A Fast FPGA Implementation of a Connectionist Classifier', IEEE *Journal of Solid-State Circuits*, Vol. 27, no. 3, pp. 288–299, March 1992.

[7] N. J. Howard *et al*, 'Zelig: A Novel Parallel Computing Machine using Reconfigurable Logic', *2nd Euromicro Workshop on Parallel and Distributed Processing*, Malaga, Jan. 1994.

[8] P. M. Athanas and H. F. Silverman, 'Processor Reconfiguration through Instruction-Set Metamorphosis', IEEE *Computer*, Vol. 26, no. 3, pp. 11–18, March 1993.

[9] P. T. Chapman and K. Clark, 'The Scan Line Approach to Design Rules Checking: Computational Experiences', ACM/IEEE *21st Design Automation Conference*, pp. 235–241, 1984.

[10] D. M. Lewis, 'A Hierarchical Compiled Code Event-Driven Logic Simulator', IEEE *Transactions on Computer-Aided Design*, Vol. 10, no. 6, pp. 726–737, June 1991.

[11] P. Agrawal and W. J. Dally, 'A Hardware Logic Simulation System', IEEE *Transactions on Computer-Aided Design*, Vol. 9, no. 1, pp. 19–29, Jan. 1990.