The Proteus Processor — A Conventional CPU with Reconfigurable Functionality

Michael Dales

Department of Computing Science, University of Glasgow,
17 Lilybank Gardens, Glasgow, G12 8RZ, Scotland.

Abstract. This paper describes the starting position for research beginning at the Department of Computing Science, University of Glasgow. The research will investigate a novel microprocessor design incorporating reconfigurable logic in its ALU, allowing the processor's function units to be customised to suit the currently running application. We argue that this architecture will provide a performance gain for a wide range of applications without additional programmer effort. The paper gives some background information into similar research, argues the benefits of this idea compared to current research, and lists some of the questions that need to be answered to make this idea a reality. We end by specifying the initial work plan for the project.

1 Motivation

There has been a lot of research into combining microprocessor technology with Field Programmable Logic (FPL) (typically using a Field Programmable Gate Array (FPGA)). Whilst these solutions claim some performance over a conventional hard–code CPU, they typically make significant alteration to the CPU, or make the reconfigurable logic detached, available over a bus. In both cases, significant work is required by the programmer to make general purpose applications take advantage of the hardware.

Another recent trend is the appearance of specialised function units in some microprocessors. As silicon technology is progressing to smaller die sizes, Integrated Circuit (IC) designers have more and more gates at their disposal. In CPU designs, they have more gates on their die than they need to build just a CPU, so the designers are looking for extra ways to utilise these gates. Typically designers use two methods to utilise this extra resource. The first is to add more memory onto the CPU, e.g. bigger caches. The second is to add more Functional Units (FUs) to the processor, such as the new MMX instructions added by Intel to its Pentium processors [8] and the 3D acceleration built into the new Cyrix MediaGX processors [3]. These designs add specialised instructions to the CPU in order to accelerate certain types of application (typically graphics based).

With more specialised function units being added to CPUs we begin to see a lower utilisation of the silicon. The new 3D graphics FUs added are fine for CAD programs or games, but are not required for databases or word processing. Another problem is that the functionality in these FUs must either be generalised versions of a problem, in which case it may not solve a programmer's specific problem [1], or tackle a specific problem and become out of date (e.g. MMX adds graphics functions working at a colour depth

of 8 bits per pixel (bpp), but most programs now are looking at greater colour depths, such as 16 and 24 bpp).

In this paper we present a novel architecture that we argue will provide a performance increase for a wide range of applications without significant overhead for the programmer. Instead of using the extra available gates to add new problem specific functions to the ALU of a conventional microprocessor, our architecture places reconfigurable logic inside the ALU, from where its functionality can be accessed in the same way as any other ALU instruction. Thus, we have a single processing device that utilises the flexibility of reconfigurable logic with a modern high speed hard–coded CPU design.

2 Current Work

Several attempts are being made to utilise the advantages of FPL in the field of microprocessor design. Here we look at each of the different categories in turn, followed by a discussion of these efforts as a whole.

2.1 Coupling a CPU and an FPGA over a bus

In this category, a conventional microprocessor is connected to an FPGA over a bus. Efforts in this area can be split into those which couple the two parts loosely and those that use tight coupling.

In loosely coupled designs, an FPGA is placed on a separate piece of silicon and attached across an external bus (e.g. PCI bus), just as a 3D acceleration card might be today [10]. Typically such a device will have a certain amount of dedicated memory attached to it (like with a video card) to reduce the number of individual bus transfers. Circuits are loaded onto the FPGA, data fed across the bus, processed and passed back. As the FPGA is on its own piece of silicon it can have a high cell count, allowing for complex circuits to be instantiated on it. Typically, no low–level support is provided for the programmer — they have to explicitly load a circuit onto the FPGA and transfer any data between it and the CPU. Some run–time system support methodologies for such devices have been investigated [2], but have failed to gain widespread acceptance.

The second method moves the FPGA onto the same die as the processor [7, 12]. This reduces the space available for the FPGA, as it has to share the available gate space with the CPU. However, this means that data transfers between the CPU and the FPGA are quicker as the bus between the two parts is much shorter and can be clocked at a higher rate. Note, however, that the FPGA is usually connected via a separate bus from the main internal data–paths. With the closer coupling comes lower–level support for accessing the FPGA. The CPU can have instructions built in for interacting with the FPGA, such as “load design” and “run circuit for *x* clock ticks”.

2.2 Building a CPU on an FPGA

The second method involves replacing the conventional processor with an FPGA and applying some form of CPU framework on top [14, 13, 4]. Different groups have tackled

this idea in different ways, but they share a common principle. In such a scenario, only the actual parts of the CPU required for a given application are instantiated; this allows middling complex circuits to be instantiated on the CPU. Because the routing of the CPU can be reconfigured, these designs typically make use of an optimised data–path for their circuit layout. Solutions in this category implicitly have low–level support for the programmer. Typically they use new programming models — they work in a fundamentally different way from current CPU designs.

2.3 Static Framework with a Reconfigurable Core

The third category describes systems that have taken a microprocessor shell and placed FPL inside this to allow its functionality to be tailored during execution. Two examples of this approach can be seen in [6] and [9]. The former is based on a data–procedural architecture, called an Xputer. This design utilises an area of FPL in which it can place multiple FUs. The second is closer to a conventional microprocessor, placing a block of reconfigurable logic inside its equivalent of the ALU, allowing the contents of the FPL to be accessed as if it were a conventional instruction. Both of these processor designs provide low–level support for the reconfigurable functionality.

2.4 Discussion

The FPGA based CPU solution has a problem as a general processor. By moving everything onto the FPGA, the entire CPU can be reconfigured, however this is not typically required. Consider operations such as 2s complement arithmetic, boolean logic, and effective address calculation, along with techniques such as pipelining and branch prediction. These standard components are always used and do not need to be reconfigurable. Thus by placing these components on an FPGA they will suffer a performance loss and fail to utilise the flexible nature of the platform. By adding a level of indirection (the reconfigurable logic) between the *entire* CPU and the silicon, it is likely that the general performance of such a CPU will not match a generic hard–coded CPU.

Now consider the problems of latency and synchronisation. The bus technique suffers from latency problems as the data being processed has to move outside the normal CPU data–paths. This technique also suffers from synchronisation problems: how is the processing in the FPGA interleaved with the normal processing of the CPU? This becomes more apparent if being used in a multiprogramming environment.

Finally, if we contrast the the approaches described above from the point of view of the programmer we see another problem. Although the tighter coupling gives programmers greater low–level support, most of the designs use a non–standard programming model. This is a drawback as it inhibits the wide–spread acceptance of such a technology — very little existing software can easily be ported to the new platform. The importance of supporting legacy systems can be seen clearly in the Intel x86 family, which still supports programs written back in the early 80's.

Thus, from a general perfromance point of view, we see a preference for the static framework and reconfigurable core technique, and from a programming view a preference for the use of a traditional Von Neumann programming model.

3 The Proteus Architecture

The idea behind our Proteus[1] processor is to place a smaller amount of reconfigurable logic inside the ALU of a conventional microprocessor where it would be tightly integrated into the existing architecture. In this model, along with the normal ALU functions, such as add and subtract, we propose to deploy a bank of Reconfigurable Function Units (RFUs) whose functionality can be determined at run–time, with applications specifying the functions that they require. Functions can be loaded into the RFUs as and when they are needed, suggesting a much better utilisation of the silicon. When functions are no longer needed they can be removed to make way for new circuits. This turns the ALU into a Reconfigurable ALU (RALU)[2]. An application could use multiple RFUs at once, loading in the set of circuits it needs. The circuits loaded into the RALU could change over time. If an application needs to load another circuit but there are no free RFUs then the contents of one could be swapped out (cf paging in a virtual memory system). This results in a CPU that can be customised to suit the application currently in use, be it database, spreadsheet, word processor or game, allowing each one to take advantage of custom FUs. These FUs can be tailored to the exact needs of a given problem, matching the programmer's requirements.

By placing this logic inside the ALU it will have access to the processor's internal data–paths, removing the latency problem described above. At a higher–level, it will appear as if the CPU has a dynamic instruction set, one that meets the needs of each program. The RFUs have the same interface as any other FU in the CPU. This approach allows the CPU to behave normally in terms of pipelining and caching — the new instructions are just like normal ALU instructions in that respect. Programs can be written using a normal programming model. The programs simply need to load their FU designs into the RALU before they are required and from then on invoking a custom function can be made syntactically the same as issuing any other instruction to a traditional ALU: all it requires is an opcode and a list of standard operands.

This technique is different from altering the microcode in the processor's control unit. Some modern processors, such as Intel's Pentium Pro and Pentium II processors, allow the microcode inside the processor to be updated [5]. Changing microcode allows machine instructions to carry out a different sequence of operations, but only operations which already exist on the processor. It does not allow any new low-level functionality to be added. Our proposal does allow for new low-level constructs to be added to the processor.

3.1 The Reconfigurable ALU

The external interface of the RALU is similar to a normal ALU. An ALU contains a set of operations of which any one can be chosen per instruction. The RALU is similar, containing a set of RFUs, any of which can be used in an individual instruction. The difference is that in the RALU the actual operations contained within each RFU can change (independently) over time to suit the currently executing application(s).

[1] Proteus - A sea-god, the son of Oceanus and Tethys, fabled to assume various shapes. O.E.D.

[2] Note that this is different from the *rALU* described in [6] — see Section 3.1 for a more detailed description of our RALU.

A conventional ALU takes in two operands *a* and *b*, and returns a result *x*. The function the ALU carries out on the operands is determined by control lines. The RALU presents the same interface to the CPU for conventional operation. It also has two additional inputs: lines which take in a circuit bit–stream to configure the individual RFUs and additional control lines needed during reconfiguration.Thus, apart from when being configured, the RALU can be used in the same way as a traditional ALU.

Inside each RFU is an area of FPL. This logic can take inputs from the operands *a* and *b* and, when selected by the *op* lines, place a result on *x*. The FPL inside each RFU is independent of that in other RFUs, thus reprogramming one will not affect the others. The amount and type of the FPL inside an RFU is an area of research (see Section 4.1).

4 Areas of Possible Research

4.1 Hardware Design

Obviously, there are the low–level aspects to consider. At the design stage we need to look at the relationship between the RALU and the rest of the CPU. It may be that the traditional ALU and RALU can be put together into a single unit, or should be kept separate. The CPU layout will be easier if the two parts are combined, but there could be good reasons for not doing so. Many modern CPU designs only place single cycle instructions in the ALU, moving multi–cycle instructions into external FUs which can run in parallel. It is likely that we would like to allow RFUs to run for multiple clock cycles, suggesting we adapt such an architecture instead.

An important question is how much FPL should each RFU contain, and how many RFUs to place in the RALU. These factors determine how complex a circuit (and thus how much functionality) can be encoded in each RFU, and how many RFUs an application can use at a given time. If the RFUs are too few or too small, applications may be constantly swapping circuits in and out, wasting time (cf thrashing in virtual memory). Too many or too big RFUs could lead to space and routing problems.

We also need to consider the necessary changes required to the control algorithm for a CPU to support the new functionality. New instructions will be required so that the run time environment for a program can set up the circuits it needs, and new instructions for invoking the functions will be needed. A related question is whether there should be caches for RFU circuits, similar to the caches for instructions and data which are already found inside CPUs?

If we are to allow the Operating System (OS) to virtualise the RALU resource (see Section 4.2), then the application can not make assumptions as to where a specific circuit will reside in the RALU (cf to virtual/physical address translation in a virtual memory system). A layer of indirection between an instruction the application issues and the RFU called will be required. The translation operation will need to be very fast — even a tiny overhead can add up to a large one if an application relies heavily on the RALU.

4.2 Operating System Issues

On the software level, we have the question of resource management of the RFUs. Just as an OS in a multiprogramming environment must manage resources such as memory

and CPU time in a secure and efficient manner, there needs to be both a mechanism and policy for management of the RFUs. Potentially the RFUs can be split amongst multiple programs, virtualising the resource. If this is done, then we need to consider how many RFUs should be dedicated to a single program, and what sort of replacement policy should be used. This could possibly also have an adverse effect on context switch times it we need to swap FUs in and out along with everything else.

Another issue concerns maintaining the pretence of a Virtual Machine (VM) to application programs. The program's functionality, when encoded as a circuit in a RFU has access to the underlying hardware, by-passing all the OS security checks. This means that malicious or badly written programs can pose security threats to other programs, or possibly hog resources (e.g. a circuit in an infinite loop). This poses the same questions that have been addressed in research into extendible OSs [11].

4.3 Programmer Support

For the new CPU design to be accessible, there are high-level programming issues to be considered too. If these issues are not tackled, then it is unlikely that the technology will be widely accepted, despite the anticipated performance increase.

One obvious question is whether software designed to utilise this new architecture will show an improvement compared to that written for a hard–coded CPU (although at this stage it is assumed that there will be some improvement). Applications will incur new overheads in terms of managing their circuits, which might detract from the performance of the system. Questions such as how many instructions need to be optimised into a single new FU to be efficient need to be examined and answered.

It is unreasonable to assume that most software engineers want to design circuits to go with their program, and software companies will not like the added expense of hiring hardware engineers. Thus, the hurdle of generating circuits needs to be removed, or at least made considerably smaller. One way to do this is to have pre–built libraries of circuits, which the programmer can invoke through stubs compiled into their program. To the programmer it will look like a normal function call that is eventually mapped onto hardware.

This technique could be improved, however, by having a post–compilation stage which analyses the output of the compiler for sections of the program that are often called and can be synthesised automatically into RFU designs. This approach is more flexible, releasing software engineers from a fixed set of circuits. This raises the important question of what functionality can we extract from programs to place into circuits? A simple way is to look for anything which takes two inputs and one output as a traditional FU does, but more complex algorithms, especially those based on iteration, may not be so easy to find. Another issue is proving that the synthesised circuit is functionally equivalent to the software instructions it replaces.

5 Current Position

Research on this project will begin in October 1999, funded by EPSRC and Xilinx Edinburgh. In the initial phase we will address the low–level design issues of a microprocessor with a RALU. The viability of such a platform will be examined, investigating

such issues as size and number of RFUs, and the type of logic that should go inside them. If this stage is successful then we plan to take an existing conventional CPU design and modify it to include a RALU. This will be simulated, and possibly prototyped, to prove the concept works.

This fundamental low–level work can then be used as a starting point for further work. This includes full analysis of the performance of the proposed architecture compared with both conventional microprocessors and the approaches taken by others (as described in Section 2), along with research into the topics discussed in Section 4.

6 Conclusion

In this paper we have presented a novel architecture for combining traditional CPU design with reconfigurable logic based on the idea of a Reconfigurable ALU, as well as highlighting many research areas that need to be investigated before such a design becomes practical. We argue that this new architecture, along with proper support in the OS and programming tools, will provide a noticeable performance gain for a wide range of applications without a significant overhead for the programmer.

References

[1] M. Abrash. Ramblings In Real Time. *Dr. Dobbs Source Book*, November/December 1996.

[2] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. de Wit. A Dynamic Reconfiguration Run-Time System. In *IEEE Workshop on FPGAs for Custom Computing Machines*, April 1997.

[3] Cyrix Corporation. Cyrix MediaGX Processor Frequently Asked Questions, January 1999.

[4] A. Donlin. Self Modifying Circuitry — A Platform for Tractable Virtual Circuitry. In *8th International Workshop on Field Programmable Logic and Applications*, September 1998.

[5] E.E. Times. Intel preps plan to bust bugs in Pentium MPUs, June 1997. 30th June.

[6] R. W. Hartenstein, K. Schmidt, H. Reinig, and M. Weber. A Novel Compilation Technique for a Machine Paradigm Based on Field-Programmable Logic. In *International Workshop on Field Programmable Logic and Applications*, September 1991.

[7] J. R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *IEEE Workshop on FPGAs for Custom Computing Machines*, September 1997.

[8] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. Intel Corporation, January 1998.

[9] S. Sawitzki, A. Gratz, and R. G. Spallek. CoMPARE: A Simple Reconfigurable Processor Architecture Exploiting Instruction Level Parallelism. In *Proceedings of the 5th Australasian Conference on Parallel and Real-Time Systems*, 1998.

[10] S. Singh, J. Paterson, J. Burns, and M. Dales. PostScript rendering in Virtual Hardware. In *7th International Workshop on Field Programmable Logic and Applications*, September 1997.

[11] C. Small and M. Seltzer. A Comparison of OS Extension Technologies. In *Proceedings of the USENIX 1996 annual technical conference*, January 1996.

[12] Triscend Corporation. Triscend corporation web page, January 1999.

[13] M. J. Wirthlin and B. L. Hutchings. A Dynamic Instruction Set Computer. In *IEEE Workshop on FPGAs for Custom Computing Machines*, April 1995.

[14] M. J. Wirthlin, B. L. Hutchins, and K. L. Gilson. The Nano Processor: a Low Resource Reconfigurable Processor. In *IEEE Workshop on FPGAs for Custom Computing Machines*, April 1994.