Micronet-based CISC Architectures

D. K. Arvind and C. Bainbridge
Institute for Computing Systems Architecture
Division of Informatics, University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland.
Email: dka@dcs.ed.ac.uk

Abstract

We have in the past investigated the design and implementation of micronet-based architectures of the scalar[AR94], superscalar[AM99], VLIW[AS97] and Multithreaded[AHKR01] kind, which were all based on a RISC-like instruction set. In this paper, we present a preliminary design, based around a micronet core, of an asynchronous Complex Instruction Set Computer (CISC) architecture. A TRANSLATOR *module converts the CISC instructions into ones which are native to the micronet datapath, called the* milliops *instructions. This translation can be accomplished either in hardware, as in Intel's x86 architectures, or in software, in the style of Transmeta's Crusoe architecture. The design environment enables C programs targeted at the CISC architecture, to be executed on a RTL model of the micronet core, and its execution and power consumption can be visualised over space and time. This allows a systematic study of the effect of compiler and architectural optimisations of micronet-based CISC processors, on the performance of application benchmarks.*

1 Introduction

We aim to develop high-performance, low-power programmable SOC cores. We seek high performance, by exploiting the available concurrency in the applications, and low power, through the asynchronous implementation of the SOC cores. The core architecture combines computational structures such as micronets or networks-on-a-chip, with software programmability. A system based on the micronet model can be viewed recursively as *networks* of entities which compute concurrently and communicate asynchronously. The idea of the 'processor-as-a-network' has been investigated in the cases of micronet-based scalar[AR94], superscalar[AM99], VLIW[AS97] and Multithreaded[AHKR01] architectures, which were all based on a RISC-like instruction set. In this work we investigate the design of CISC architectures based around a pre-designed and pre-verified micronet core.

2 The Architecture

Figure 1 illustrates the functional blocks of a generic CISC architecture based around a micronet core. At the start of a typical *fetch-execute* cycle, the CONTROL UNIT (CU) requests the value of the PROGRAM COUNTER (PC) from the REGISTER BANK (RB), and a *milliop* instruction from the TRANSLATOR. The latter reads the PC value on the bus, acknowledges the transfer with the RB, and reads the CISC instruction from the MEMORY. The variable-length CISC instruction is first translated into its equivalent opcode instructions and loaded into a 10-byte buffer.

This buffer is processed to produce a sequence of *milliop* instructions, which are presented individually to the CU. The *milliop* instruction set, which is native to the micronet datapath, is a stripped-down version of the MIPS instruction set. The CU executes one instruction at a time, and does not have prior knowledge of either the PC or the length of the milliop sequences. When the CU requests the next milliop instruction and should the TRANSLATOR have reached the end of the sequence, then it signals the CU, which in turn requests the PC content to be presented on the bus – and so the cycle continues.

On receiving a *milliop*, the CU reads the two operands onto the 32-bit operand buses of the respective functional units, and locks the write-back register. The register bank consists of 8-bit registers, each of which can be specified as a register name.

As is usual in CISC architectures, register hierarchies can be created in the following way: in the case of the IA32 architecture for instance, both AH and AL are 8-bit registers, with AX representing the concatenation of AH and AL. It is important that such relationships are recognised, since an instruction writing back to AL, should lock the register, AL, but not the register, AH. Similarly, reading on AX will stall until registers AH and AL are both unlocked.

We wish to study the architecture systematically by scaling the number of functional units. Towards this end, the milliops are passed by the CU to an arbiter, which in turn presents it to any functional unit which is available. If the unit can handle the instruction, then it raises a *can do* signal. The arbiter then selects the unit, and raises a *grant* signal to indicate that the unit should complete the milliop. If units are not currently available to handle the milliop, then the CU will stall until one does and signals a *can do* back. The arbiter, therefore, requires no prior knowledge of the functional units - either their number or the operations that they are capable of. This allows the system to be scaled easily in the simulations, with functional units processing different milliops simultaneously.

A subset of the Intel IA-32 architecture was modelled in C++ and simulated using a distributed event driven simulator library[AS01]. Each discrete entity wass modelled as a C++ class, which inherits from an 'Entity' base class. These classes are connected using typed channels through which values can be presented, along with trigger events to signal changes. Development was carried out on Sun Sparc and Linux x86 computers. As these computers have different byte orders and standard libraries, the communication between them were encoded as XDR. The simulation records power and signal events to XML formatted files, for later examination.

3 The Visualiser

The visualiser was written in Python and uses the Gnome/GTK display libraries. The XML logs from the simulator are parsed to extract information about signal and power events. These are then plotted onto a GnomeCanvas, which can be manipulated using the user interface, as illustrated in Figure 2. It shows the code on the left-hand side of the display which can be scrolled vertically, together with power and signal events in the top and bottom halves of the right-hand side.

The user can select the details for plotting. Individual signals can be selected, and timing and instruction information can be displayed alongside the coloured activity bars. Changes in the colour indicate a change of state, which can be investigated by switching to a *data* view and moving the pointer over the coloured rectangle. A textual representation of the data, which can vary from a single bit to a complex C++ class, will be displayed in the *data* view.

Acknowledgements

This project was partially supported in the form of an equipment grant by the Scottish Higher Educational Funding Council (*Design Clusters for System Level Integration*). Christopher Bainbridge is funded by a studentship from the Engineering and Physical Sciences Research Council (EPSRC).

References

[AR94] Arvind, D. K. and Rebello, V. E. F. 1994. "Instruction-level Parallelism in Asynchronous Processor Architectures", In Proc. of the Third International Workshop of Algorithms and Parallel VLSI Architectures, Elsevier, Leuven, Belgium, August 1994.

[AS97] Arvind, D. K. and Sotelo-Salazar, S. 1997. "Scheduling Instructions with Uncertain Latencies in Asynchronous Architectures", In Proc. *EUROPAR'97*, Passau, Germany, Aug. 1997, Springer-Verlag.

[AM99] Arvind, D. K. and Mullins, R. D. 1999. "A fully Asynchronous Superscalar Architecture", In Proc. *PACT99 - Int. Conf. on Parallel Architectures and Compilation Techniques*, Newport Beach, USA, 13-15 Oct. 1999, IEEE Press.

[AHKR01] Arvind, D. K., Hossell, J., Koppe, A., and Rangaswami, R. 2001. "Java Compilation for Multi-threaded Architectures", In Proc. 9th Workshop on Compilers for Parallel Computers, Edinburgh, Scotland, June 2001.

[AS01] Arvind, D. K. and Schneiders, J. 2001. "Area Virtual Time", In Proc. 5th IEEE Int. Workshop on Distributed Simulation and Real-Time Applications, IEEE Press, Cincinnati, Ohio, August 2001.

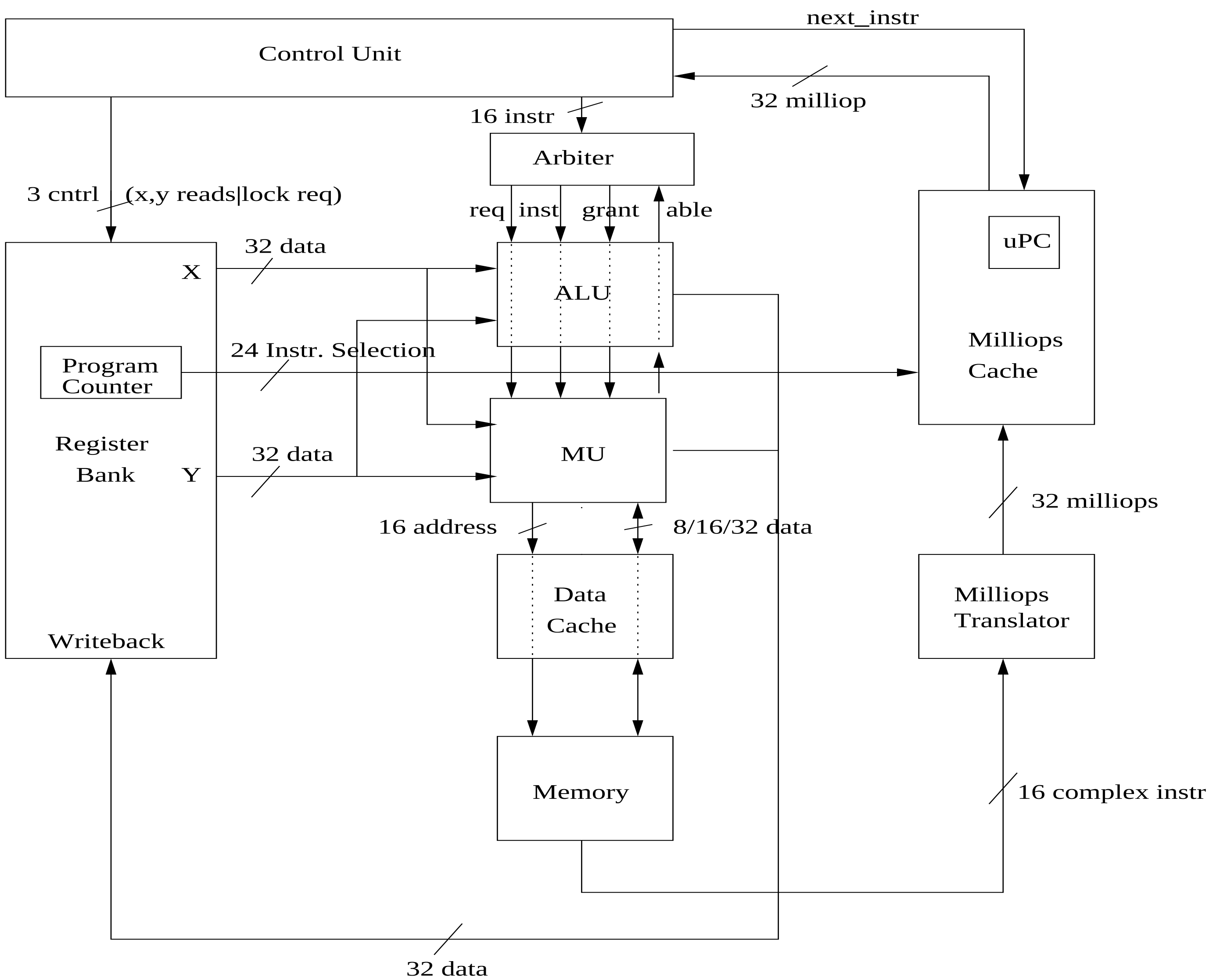

Figure 1. The micronet-based CISC architecture

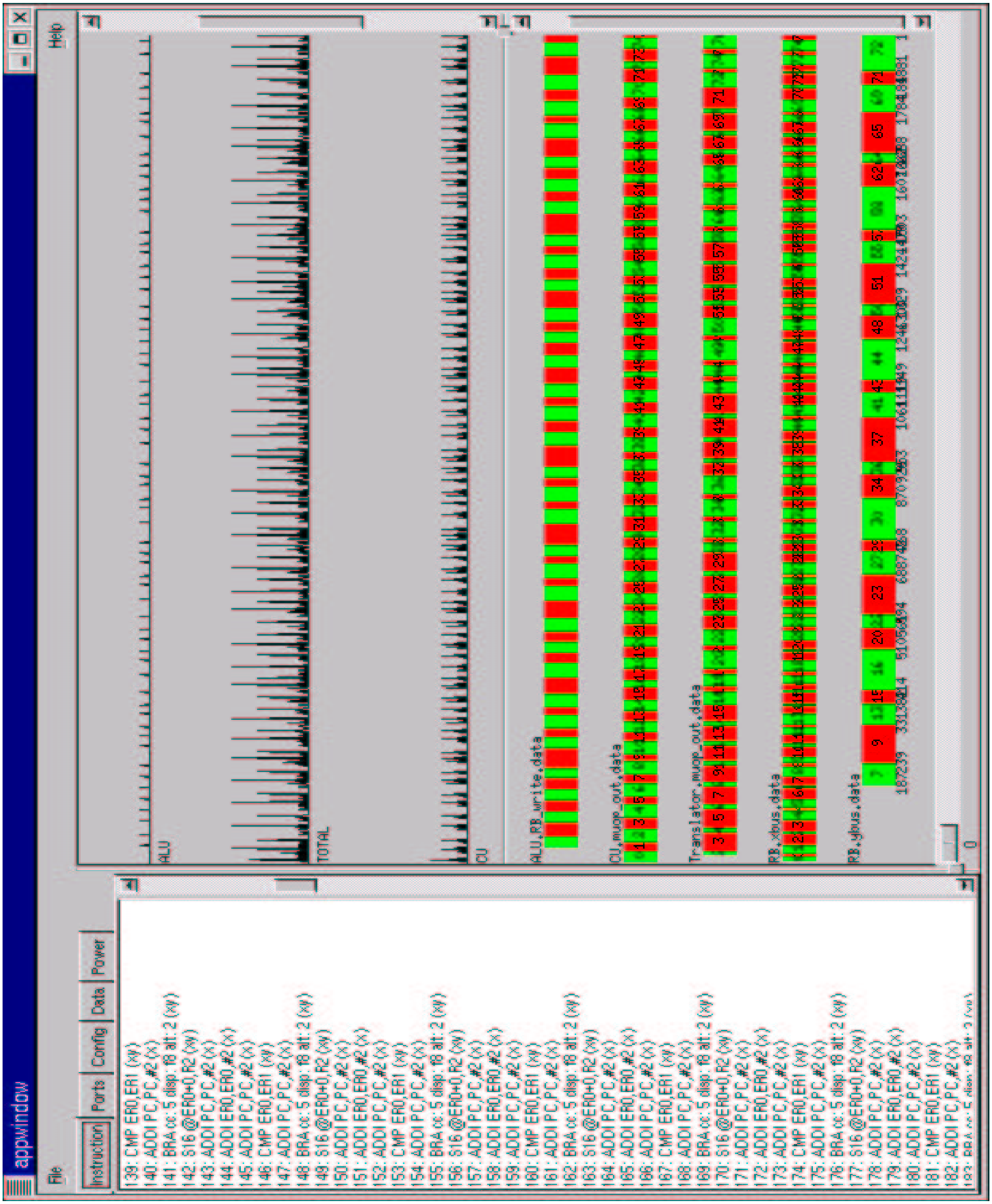

Figure 2. Graphical interface of the design environment