A Processor for Artificial Life Simulation

Matthias Böge and Andreas Koch

Technical University of Braunschweig, Dept. E.I.S., Gaußstr. 11, D-38106 Braunschweig, Germany {boege,koch}@eis.cs.tu-bs.de

Abstract. We present a processor architecture and initial implementation specialized for the simulation of biological evolutionary processes. The CPU simulates a MIMD shared memory computer and executes a set of instructions whose operation is subject to random mutations. Furthermore, it transparently provides memory management and thread control at the assembly level. The current implementation relies on a complex controller using two-level micro-code to generate the required control sequences.

1 Introduction

Computer simulation of digital "organisms" can be employed to experimentally examine natural evolutionary processes that might take millions of years in biological systems. Among the aspects observed are competitive exclusion and coexistence, host/parasite density dependent population regulation, the effect of parasites to enhance the diversity in a community of organisms, and the "evolutionary arms race" (hosts, parasites, hyper-parasites, . . .) [1].

TIERRA [2], which forms the basis for this work, is one of several "artificial life" simulation systems [3] [4] that have been developed in the past. It provides a software model of a virtual MIMD shared memory computer and operating system, with the organisms being programs that are executed in parallel. In contrast to conventional processors, the TIERRA CPU is designed to allow the evolution of the organisms by mutation (changing the program code on the fly) and recombination (exchanging code segments between programs). Organisms that still remain functional (contain valid instructions) after such modifications continue to live (run), while those that are damaged (contain invalid instructions) die (are marked for removal from execution and memory).

We present our first attempt to actually implement such a processor for evolvable computation in hardware.

2 TIERRA Virtual CPU

While TIERRA can be configured for multiple instruction sets and in terms of the number of registers and the stack depth, we will concentrate on just a single architecture with the following characteristics:

It has four general purpose registers named AX, BX, CX and DX, all of them 16 bits wide. The architecture also contains a 16-entry stack with an associated stack pointer SP as well as an instruction pointer IP. The instruction set executed encompasses the

32 operations shown in Table 1. A dedicated fault flag register FL is set when an invalid instruction would be executed next (which will be ignored instead). The program memory holding the organisms, called the *soup*, has a size of 64KB.

While this instruction set seems to be rather conventional, it is made unique by the following aspects: In order to simulate random mutations ("cosmic rays"), the execution of each instruction is randomly modified with a user-defined probability. One such modification is the alteration of a register specification. E.g., a destination of DX might become AX or CX for a specific instruction execution. Another mutation introduces +/- 1 inaccuracies into arithmetic and logic (e.g., shift distance) operations. Furthermore, organisms procreate by copying themselves, an operation that is also subject to random changes and thus creates mutated offspring. This approach might result in invalid instructions, which will be detected and skipped during the execution.

Furthermore, control flow instructions do not operate on absolute or relative addresses, but instead search for sequences of nop0 and nop1 instructions that label a specific location within the code. In this manner, organisms may recognize and jump to certain signatures within themselves or other organisms. Note that this approach requires the relevant instructions to actually search forward and/or backward through the soup for an arbitrarily long sequence of label nops, making them very slow to implement.

Similarly, the mal and div instructions are quite complex: mal implements a first-fit memory allocation scheme which also performs garbage-collection (see below) if required, while div spawns another thread of execution and thus simulates the separation of a child from a parent organism.

Code	Description
nop0	no-op, code 0x00
nop1	no-op, code 0x01
pushA	push AX on stack
pushB	push BX on stack
pushC	push CX on stack
pushD	push DX on stack
popA	pop AX from stack
popB	pop BX from stack
popC	pop CX from stack
popD	pop DX from stack
movDC	DX $\leftarrow$ CX
movBA	BX $\leftarrow$ AX
movii	[AX] $\leftarrow$ [BX]
subCAB	CX $\leftarrow$ AX $-$ BX
subAAC	AX $\leftarrow$ AX $-$ CX
incA	AX $\leftarrow$ AX $+1$
incB	BX $\leftarrow$ BX $+1$
incC	BX $\leftarrow$ BX $+1$
decC	CX $\leftarrow$ CX -1
not0	CX $\leftarrow$ CX *xor* 1
zero	CX $\leftarrow$ 0
shl	CX $\leftarrow$ $2 \cdot$CX
ifz	skip next instr if CX $\neq$ 0
jmpo	jump to nearest label
jmpb	jump backwards to label
call	call to nearest label
ret	return from call
adro	AX $\leftarrow$ find nearest label
adrf	AX $\leftarrow$ forward find label
adrb	AX $\leftarrow$ backward find label
mal	allocate memory
div	create new thread (cell division)

Table 1. TIERRA Instruction Set 0

In addition to instruction execution, the TIERRA virtual CPU also updates various data structures for memory management and scheduling. The most crucial of these are the reaper and slicer queues. The first is used to determine the "fitness" of an organism: Each time a program attempts to execute an invalid instruction, it is moved upwards in the reaper queue. Each time it successfully procreates using a mal/div combination, it is moved downwards. When the time comes, and a mal request requires more space than currently available, organisms die (are garbage collected) in the order of ascending fitness until the request can be satisfied. Conversely, when a new organism appears, it is allowed to execute immediately after separation from its parent by inserting it appropriately into the slicer queue, which allocates time slices to the organisms.

The software realization also updates various statistical functions that track the diversity and life cycles of organisms in the soup. In this first attempt, these operations were not considered for hardware implementation.

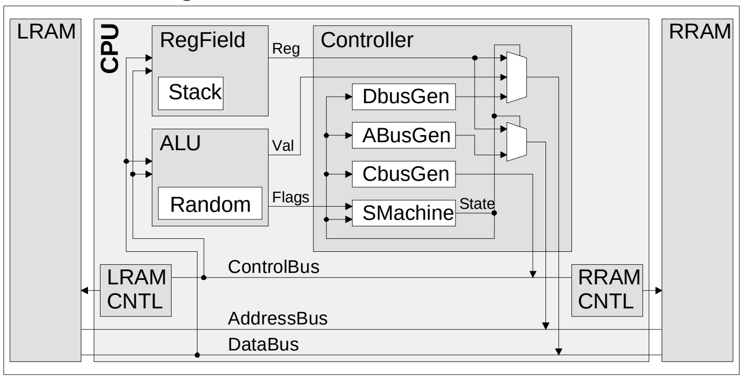
3 Hardware Realization

Even in this simplified version, our attempts at realizing the TIERRA processor push the limits of established FPGA technology. Only the very latest devices [5] are able to accommodate the synthesized circuits efficiently.

The initial target for TIERRA was the SPARXIL architecture [6] (three XC4010 FPGAs, two memory banks). Space limitations forced us to to abandon the planned RISC-like design with fast execution units dedicated to specific TIERRA functions (e.g., a label search engine), instead settling on a realization named TIERRA/1 with a higher degree of operation sharing, but heavily dependent on multiple levels of hard-wired micro-code.

Compared to most conventional CPUs, TIERRA/1 manages a diverse set of data structures. This includes the current data context for each cell (registers, stack, and offspring reference), a map of free memory locations, and the soup holding the program code for the cells itself. Additional locations are used to implement cell scheduling and garbage collection.

Figure 1. TIERRA/1 Hardware Architecture

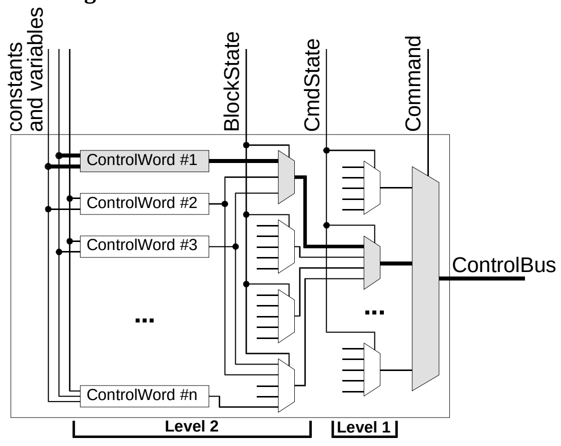


The hardware architecture is sketched in Figure 1. In addition to the registers and internal stack required by the base architecture, the ALU has been extended with a pseudo-random number generator based on a linear-feedback shift-register. All of these units together require only a small fraction of the total device area, the far larger remainder is occupied by the central controller.

4 Hierarchical Micro-Code

Small implementation size was the main design goal for the controller. To this end, it employs a hierarchical micro-code scheme with a considerable sharing of resources between steps. In addition to the instruction semantics defined by the TIERRA specification, the controller also performs the context switches required to emulate a MIMD architecture on a single processor.

Figure 2. Hierarchical micro-coded controller



The controller is partitioned into sub-units handling accesses to the data and address busses as well as the operation of the internal execution units. After fetching the instruction code, the controller selects one of 25 micro-code routines at the first hierarchy level (see Figure 2) selected by the Command input. Each of these routines consists of multiple micro-instructions (sequenced by CmdState). For example, a nop0 is processed in two micro-code steps, while the complex mal instruction requires 64 steps (also containing conditionals and loops). At the second hierarchy level, the micro-instructions themselves are composed from 129 different nano-code routines (sequenced by Block-State). These routines have a length of one to six nano-steps that apply control words to the various control signals. The control words contain both hardwired as well as data-dependent control bits. Each of the nano-routines is re-used an average of 2.6 times in the implementation of the micro-instructions, with the highest degree of sharing being 21 times. One nano-step currently requires three clock cycles (two memory accesses plus bus turn-around time).

Figure 3. Control flow for the SHL instruction

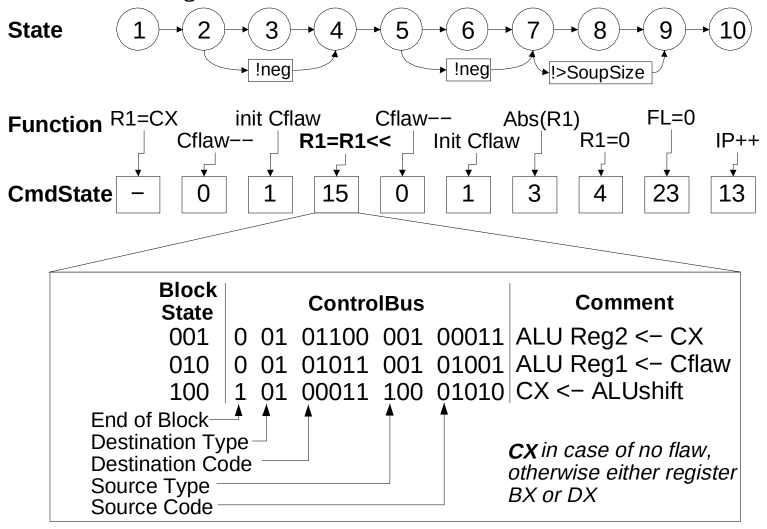


Figure 3 is an example the for the operation of the central controller. It sketches the execution of the "shift left" instruction, which ideally (assuming no mutation) shifts the CX register one bit to the left. The micro-code statements are sequenced by CmdState. In the State 1, we set the target register as CX, mutating it into BX or DX if Cflaw is 0. Afterwards, we decrement Cflaw and re-initialize it to a positive random value. In State 4, we perform the actual shift. If Cflaw is zero at this point, we randomly offset the shift distance by plus or minus 1 (leading to an effective shift by 2 or 0 bit positions). After decrementing and optionally re-initializing Cflaw, we constrain the shifted value to lie within SoupSize (since it might be used for addressing within the soup) and mark the instruction just executed as valid before advancing to the next one.

To illustrate the functionality of the nano-routines, we look at the internal implementation of the R1=R1<< micro-instruction, which consists of three nano-instructions generating 16-bit control words. These encode data sources and destination types (b01 and b001 indicate a register, b100 is an ALU output) as well as specifics (b01100 and b01101 are the two ALU input registers, b00011 is CX, b01001 the mutation flaw, and b01010 is the ALU shifter). The BlockState is one-hot encoded, and the nano-routine signals its completion by setting an end-of-block indicator.

5 Performance

Due to the strong control emphasis, the resulting circuit is extremely irregular, and has a large number of high-fanout (> 100) nets. It quickly became apparent that SPARXIL would not be large enough to accommodate this area-optimized design. But even larger chips are hard pressed (Table 2).

Table 2. Area and performance characteristics

Target Chip	%logic used	%delay in routing	MHz clock
XC4062XL-08	100	63	9.3
XC4085XL-3	68	74	5.4
XCV300-6	83	87	22.2

Interestingly, the synthesized design fits and routes in a XC4062XL, but is barely routable in the larger XC4085XL (taking numerous attempts at the maximum tool effort level). Furthermore, the performance of the XC4000-series based implementations (for which we already have prototyping hardware) is disappointing at best.

The very latest FPGAs such as the 300k gate XCV300 Virtex part seem better suited for our purposes. The design easily places and routes, and achieves a more respectable clock speed.

6 Lessons Learned

While our first attempt at implementing a processor for artificial life simulations must be considered less than successful from a performance view point, we learned several important lessons. The first is, that all but the latest devices are simply too small to implement even an area-optimized micro-coded design efficiently. Furthermore, even when state-of-the-art chips are used, the performance is far below those of pure software implementations running on today's CPUs. However, with FPGA capacities growing into the millions of gates, the initially envisioned version using dedicated (hardwired) execution units instead of multi-cycle micro-code might become feasible at last. We plan to follow these developments closely when making another attempt at tackling this fascinating problem.

References

1. Ray, T., "What Tierra Is", http://www.hip.atr.co.jp/˜ray/tierra/whatis.html, 03/1999
2. Ray, T., "Tierra Home Page", http://www.hip.atr.co.jp/˜ray/tierra/tierra.html, 03/1999
3. Adami, C., Brown, T., "Evolutionary Learning in the 2D Artificial Life System Avida", *Proc. of Artificial Life IV*, MIT Press, 1994
4. Menczer, F., Belew, R., "Latent Energy Environments", http://dollar.biz.uiowa.edu/˜fil/LEE/, 03/1999
5. Xilinx Inc., "Virtex 2.5V Field Programmable Gate Arrays", *Advance Product Specification*, 01/1999
6. Koch, A., Golze, U., "Practical Experiences with the SPARXIL Co-Processor", *Proc. Asilomar Conference on Signals, Systems, and Computers*, 11/1997