INSTRUCTION-SET ARCHITECTURES

William Bricken
October 2002

The application of rules to achieve transformation of forms can be achieved using may different implementation techniques. For example, the evaluation of a form can be accomplished by using procedural methods such as those embodied in assembly languages for vonNeumann processors, by using declarative methods such as pattern-matching, by using functional methods such as implementations of lambda calculus, by using object-oriented methods of message passing, by using graph restructuring methods, by using parallel processing methods such as vector processing, and by using a variety of methods specific to container-based forms such as map transformation, 3D block manipulation, path rearrangement, and several other approaches, both specific to container-based forms and general as computational techniques. It is important to recognize that each of these methods and techniques can be expressed as an abstract method independent of computational implementation, as a software method independent of hardware substrate, and as a hardware method that defines a particular hardware architecture and machine configuration.

With regard to hardware organization, boundary-based computation is equally well realized in stack-based procedural machines, functional lambda calculus machines, container-based LISP machines, and other architectures. The predominant type of machine available today is based on a vonNeumann architecture. What follows are outlines of two boundary-based techniques for instruction-set vonNeumann CPU architectures.

Bit-stream Method

The three primary atomic rules for container-based evaluation can be expressed as bit-stream transformations of *non-evaluated* forms. This adds larger computational steps to the bit-stream process at the cost of recursive subroutines. Figure I shows a simple encoding scheme for parens forms without atoms. The figure also extends this encoding scheme to bit-streams with atoms. The essential difference is that the addition of atoms requires more unique bit-codes to encode the unique atoms. Figure I shows that example of 12 unique atoms, requiring twelve unique codes. Adding two codes for the open and close parentheses, the number of bits required to encode these 14 items is 4, providing 2^4 different codes. The bit-stream then consists of 4-bit packets, called nibbles by the technical community. Figure II presents the three primary deletion-reduction rules as bit-stream transformations.

Algebraic Pervasion requires a pattern builder to get a copy of X as a template, and then using the constructed pattern as a pattern mask to identify X for Pervasion deletion. Even pattern-rearrangement can be handled using a bit-stream, as is indicated in Figure II by the Distribution rule. The result is an efficient bit-stream universal Turing machine. The information entropy is included in for each bit-stream rule in Figure II.

Assembly Language Method

As well as being implemented in hardware as a bit-stream and as a containment graph, a container-based circuit model can be implemented in *assembly language*, as a set of *machine instructions* for a general purpose or a specialized microprocessor. A customized microprocessor for container-based circuits would operate as an extreme RISC (reduced instruction set computer) processor, requiring only two instructions for computation, two for reading and writing to memory, and one to terminate. Such an implementation would have all the benefits of the bit-stream hardware implementation, except that rather than existing as hardware that processes a bit-stream, it would exist as hardware that processes an instruction set. The advantages include extremely rapid function evaluation, general programmability, extremely small hardware implementation, and fine-grain parallel processing capability. In contrast to the bit-stream method, the parens-microprocessor method would be able to take full advantage of partial evaluation, skipping instructions given some input bindings. The bit-stream approach must read the entire bit-stream. The same contrast exists compared to conventional circuitry, the parens-microprocessor would have all the benefits of the asynchronous containment graph approach, except that asynchronous termination would not require any sort of coordination, the processor would simply finish its current task faster. Thus, for example, should an 8-bit comparator circuit implemented in parens assembly language discover that the first bits do not match, the processor would immediately return a result, jumping over all other instructions to the END instruction. In conventional hardware, all gates would continue parallel processing. Thus the parens-microprocessor method can be both very fast and very low power consumption.

Container-based languages are context-free. Thus the implementation of a parens-encoded circuit in assembly language is a *straight-line program*, that is, one without loops and branches. Thus combinational circuits can be implemented and evaluated in a parens-microprocessor without complex control structures embedded in either the representation language of the circuit or the machine instructions of the microprocessor, even though the circuit itself can be arbitrarily complex. As a matter of pragmatism, the instruction set would include JUMP statements in order to skip unnecessary processing. The straight-line property here translates into the behavior that all jumps skip some *later* instructions and thus move more rapidly toward termination. Looping would never occur.

Figure III shows a pseudo-instruction set of a parens-microprocessor. This instruction set is indicative, without the detail of a functional instruction set. The *parens assembly language* in the figure consists of triples,

<label, instruction, arguments>

Labels give names to memory locations, instructions direct the microprocessor controller, and arguments provide specifics about what the control unit is acting upon. Instruction sequences can be constructed automatically by a compiler for parens forms; parens forms can be constructed automatically from either netlists as hardware descriptions or high-level programs as software descriptions. The parens-microprocessor instruction set controls memory access only, *no processing instructions are used.*

The number of machine level instructions for a process is a direct measure of the efficiency of a microprocessor. It is not uncommon for simple computational instructions expressed in a software language to expand into hundreds of machine instructions. These machine instructions are processed very rapidly, perhaps one instruction for every Hz of processor speed, that is, around a billion instructions each second. However, microprocessors are still hundreds of times slower than an ASIC implementation. Comparatively, the parens-microprocessor requires on average two or three instructions for each gate in a circuit model. Again on average, less than half of these instructions would be executed for a given input binding. Thus a quite realistic estimate of performance for a parens-microprocessor is one instruction per netlist gate.

The parens-microprocessor is unusual since the instruction set does not include arithmetic, logical, shift, compare, or move instructions as do conventional instruction sets. Thus, no ALU is required or needed. Rather the entire process is carried out solely by the microprocessor control unit and the load/store memory instructions. This processor should require no more than 8-bit wide data and control paths, and perhaps less with proper memory organization.

Although each parens form generates only one bit of information, that bit represents the entire process of an ASIC circuit. Importantly, the parens-microprocessor, due to its extremely small size and to the extreme ease of parallelizing parens forms, is inherently a massively parallel processor system. Rather than implementing vonNeumann-style computation, with wide instructions being completed one-at-a-time in sequence, the parens-microprocessor architecture would be a collection of perhaps thousands of replicated small units, each reducing a small fragment of the entire parens form of the specified functionality in parallel. This, again, is similar to the bit-stream processor described earlier herein.

Another unique aspect of the parens-microprocessor is that it is essentially a hardware emulator, the machine instructions encoding a physical circuit netlist. However, in any custom implementation, the instruction set could be extended to include conventional processing, such as a multiply-accumulator.

This hybrid approach unites high-level software programming with low-level hardware logic networks by by-passing the idea and the use of Hardware Description Languages (HDLs). Thus, some benefits of this aspect of the inventions include making hardware design accessible to microprocessor software programmers, making software description and abstraction available to netlist specifications, and eliminating several layers of the Tower of Babel that currently unites functional specification with semiconductor behavior. An advantage of this simplification is to potentially speed-up microprocessor performance by an order of magnitude, while both creating compatibility between hardware and software specification and design techniques, and maintaining current design practices.

The hierarchical abstraction and vectorization capabilities of container-based techniques play an important role in the design and performance of the parens parallel microprocessor. Specifically a vectorized parens form is strictly analogous to the wide bit-width of a conventional microprocessor. Rather than processing, say, 64 bits as a long word, 64 parallel processors compute the results of a wide word as 64 single instruction streams. Vectorization of the circuit netlist provides specifications for the dynamic configuration of the parallel processors. An advantage of this approach is that bit-width is entirely flexible and programmable, thus avoiding the limitations of fixed bit-width architectures that for any particular process may be either squanderous of hardware resources, or in the other extreme, drastically inadequate for the functionality being computed.

Figure IV shows an example assembly language program for the parens form consisting of four two-input logic gates. The subroutine is generic, implementing the functionality with any inputs. The five memory locations store the specific bindings for the four inputs and the one output. A compiler converts, assembles, optimizes and links the netlist specification of a circuit to construct the assembly level program stored in memory. Parens forms are used as an intermediate language for logic and control stream optimization. Each instruction is stored in the control portion of memory. The result is a software defined and reconfigurable circuit emulator that can be used for both input evaluation and abstract synthesis and optimization.

FIGURES

```
================================================================================
```

Encoding containers into bit-streams

```
                  (  =  1
                  )  =  0
                  x  =  x           -- multiple bits required
```

Example of twelve unique atoms

```
                  )  =  0000
                  a  =  0001
                  b  =  0010
                  c  =  0011
                  d  =  0100
                  e  =  0101
                  f  =  0110
                  g  =  0111
                  h  =  1000
                  i  =  1001
                  j  =  1010
                  k  =  1011
                  m  =  1100
                  (  =  1111

            not used:   1101
                        1110
```

Parsing example

```
                  (a (b ((c)(j k))))

 (    a    (    b    (    (    c    )    (    j    k    )    )    )    )

1111 0001 1111 0010 1111 1111 0011 0000 1111 1010 1011 0000 0000 0000 0000

================================================================================
```

Figure I: Encoding Parens Format into Bit-streams

```
==============================================================================

OCCLUSION                      ( ) A = ( )

            Bit-stream  transformation               Entropy

                10X  => 10_                    S => S+(1/2 length-of-X)

                X10  => _10


INVOLUTION                     ((A)) = A

            Bit-stream  transformation               Entropy

                11x00 => __x__                 S => S+2


PERVASION                      (A (A)) = (A ( ))

            Bit-stream  transformation               Entropy

                1X1X00 => 1X1_00               S => S+(1/2 length-of-X)


DISTRIBUTION                   ((A B)(A C)) = A ((B)(C))

            Bit-stream  transformation               Entropy

                11XY01XZ00 => X11Y01Z00        S => S

==============================================================================
```
Figure II: The Three Primary Reduction Rules as Bit-stream Transformations.

===

Parens assembly-level encoding and instructions

instruction = (<memory-pointer> <instruction-code> <arguments>)

instructions	*function*
(id LOAD <memory-location>)	get bindings
(id GOLO <register-value> <id>)	jump to instruction on 0 value
(id GOHI <register-value> <id>)	jump to instruction on 1 value
(id STOR <memory-location> <value>)	store results
(id END)	terminate subroutine

===

Figure III: Machine Instructions for a Container-based Microprocessor

```
==========================================================================

Input  circuit         (((m1)(m2 m3))(m1 (m4)))

Implementation

SUBROUTINE EXAMPLE

                  ( 1  LOAD  m1)
                  ( 2  GOLO  m1  9)                  -- jump m1=0
                  ( 3  LOAD  m2)
                  ( 4  GOHI  m2 11)                  -- jump m2=1
                  ( 5  LOAD  m3)
                  ( 6  GOHI  m3 11)                  -- jump m3=1
                  ( 7  STOR  m5  1)                  -- output=1
                  ( 8  GOLO   0 12)                  -- jump to END
                  ( 9  LOAD  m4)
                  (10  GOLO  m4  7)                  -- jump m4=0
                  (11  STOR  m5  0)                  -- output=0
                  (12  END)

Evaluation

            memory-location m1:     1                -- input value
            memory-location m2:     0                -- input value
            memory-location m3:     1                -- input value
            memory-location m4:     1                -- input value
            memory-location m5:  <result>            -- output value

                  ( 0  CALL EG)
                  ( 1  LOAD  m1)
                  ( 2  GOLO   1  9)                  -- <no-op>
                  ( 3  LOAD  m2)
                  ( 4  GOHI   1 11)                  -- <no-op>
                  ( 5  LOAD  m3)
                  ( 6  GOHI   1 11)                  -- jump m3=1
                  (11  STOR  m5  0)                  -- output=0
                  (12  END)

==========================================================================
```
Figure IV: An Example of
Container-based Microprocessor Machine Instructions and Evaluation