THE SCHEME MACHINE:
A CASE STUDY IN PROGRESS IN DESIGN DERIVATION AT SYSTEM LEVELS

Steven D. Johnson

Indiana University

The Scheme Machine is one of several design projects of the Digital Design Derivation group at Indiana University. It differs from the other projects in its focus on issues of system design and its connection to surrounding research in programming language semantics, compiler construction, and programming methodology underway at Indiana and elsewhere. The genesis of the project dates to the early 1980s, when digital design derivation research branched from the surrounding research effort in programming languages. Both branches have continued to develop in parallel, with this particular project serving as a bridge. However, by 1990 there remained little real interaction between the branches and recently we have undertaken to reintegrate them.

On the software side, researchers have refined a mathematically rigorous (but not mechanized) treatment starting with the fully abstract semantic definition of Scheme and resulting in an efficient implementation consisting of a compiler and virtual machine model, the latter typically realized with a general purpose microprocessor. The derivation includes a number of sophisticated factorizations and representations and is also deep example of the underlying engineering methodology.

The hardware research has created a mechanized algebra supporting the tedious and massive transformations often seen at lower levels of design. This work has progressed to the point that large scale devices, such as processors, can be derived from first-order finite state machine specifications. This is roughly where the language oriented research stops; thus, together, the two efforts establish a thread from the highest levels of abstract specification to detailed digital implementation.

The Scheme Machine project challenges hardware derivation research in several ways, although the individual components of the system are of a similar scale to those we have worked with before. The machine has a custom dual-ported memory to support garbage collection. It consists of four tightly coupled processes---processor, collector, allocator, memory---with a very non-trivial synchronization relationship. Finally, there are deep issues of representation for the run-time objects of a symbolic processing language.

The research centers on verification through integrated formal reasoning systems, but is also involved with modeling and prototyping environments. Since the derivation algebra is based on an executable modeling language, there is opportunity to incorporate design animation in the design process. We are looking for ways to move smoothly and incrementally from executable specifications into hardware realization. For example, we can run the garbage collector specification, a Scheme program, directly against the physical memory prototype, and similarly, the instruction processor model against the heap implementation.

The Scheme Machine:

A Case Study in Progress
In Design Derivation at System Levels

Steven D. Johnson
DDD Project, Hardware Methods Laboratory
Computer Science Department
Indiana University

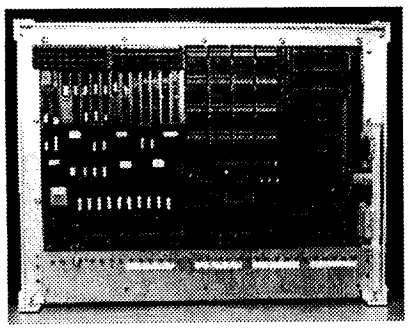
sjohnson@cs.indiana.edu
http://www.cs.indiana.edu/hmg/hmg.html

Thanks: NSF/MIP92 08745, NASA/NGT-50861

Outline

- The Scheme Machine Prototype
- Context, Motivation, and Goals of the Research
- Components of the Scheme Machine
- Issues in Verification
- Incremental Construction of Hardware
- Summary and Status

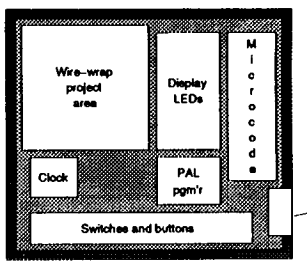
The Scheme Machine Prototype

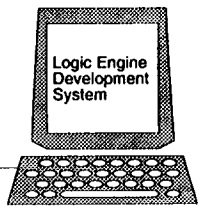


Emulator in Actel FPGAs, PLDs and standard DRAM simms.

The Logic Engine

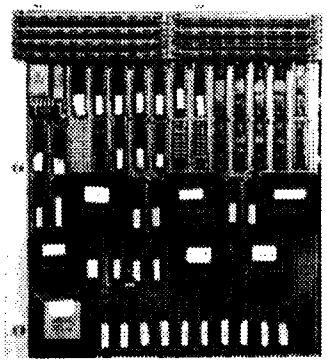
An environment for sytem prototyping, VLSI emulation, and functional testing





(Scheme and/or C)

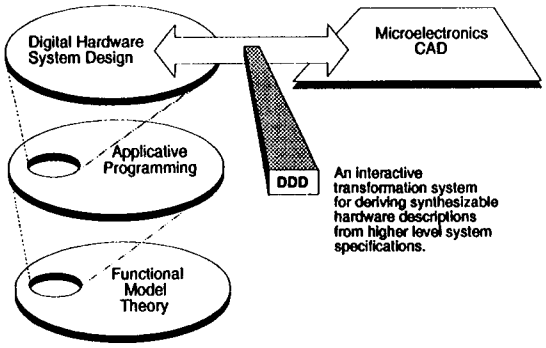
Scheme Machine
Memory:
$2 \times 4 \times 1,000,000$ bytes
Design Size:
20,000 gates (?)
Speed:
2 MHz+ (or $1.25 \times$ DRAM)



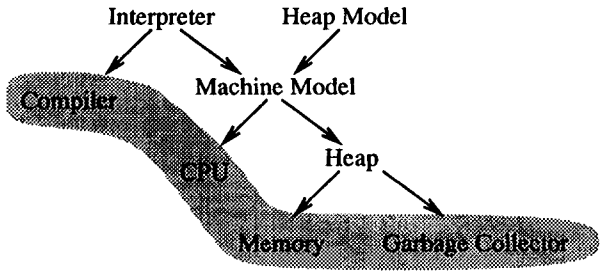
Context of the Research

1982 M. Wand, *Semantics-directed machine architecture* (POPL 92)
1983 S.D. Johnson, *Synthesis of Digital Designs from Recursion Equations*, Ch. 5 (PhD dissertation)
1984 W. Clinger, *The Scheme 311 Compiler* (L&FP 84)
1987 S.D. Johnson, et. al. *A Tactical Framework for Digital Design* (WG10.2 FM/VLSI).
1988 R. Wehrmeister, *Derivation of an SECD Machine* (IU/CS TR #290)
1990 D. Boyer and K. Rath derive boolean system for a Scheme machine.
1991 IEEE Std 1178-1990 *IEEE Standard for the Scheme Programming Language.*
1992 M. Wand, D. Friedman, C. Haynes, *Essentials of Programming Languages*, Ch. 12.
1993 B. Burger, *The Scheme Machine* (IU/CS TR #413).
1995 M. Wand, J. Guttman, J. Ramsdell, et.al. *VLISP: A Verified Implementation of Scheme* (*J. Lisp and Symbolic Computation*).

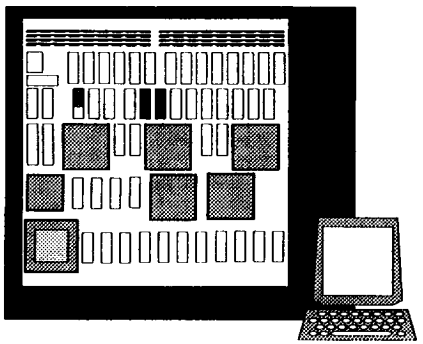
Digital Design Derivation Project



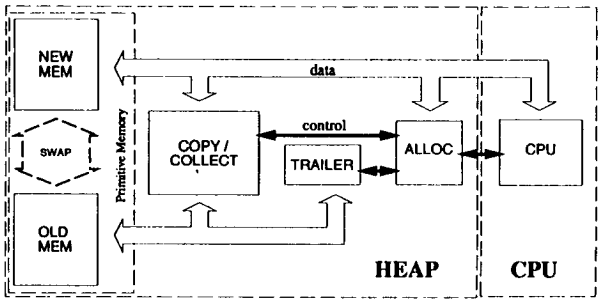
The Scheme Machine Project



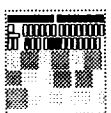
Scheme Machine Prototyping Environment



Scheme Machine Architecture

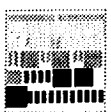


Components of the Scheme Machine



Memory
- o Dual-ported semispaces, multiplexed bus
- o typical case: 2 opns. per 1.25 cycles
- o refresh inhibits clock
- o manual design, unverified (one bug so far)

Heap Interface
- o collector, allocator, invalidator
- o derived from XFSM.
- o synchronization with CPU not verified
- o algorithm not verified (VLISP?)

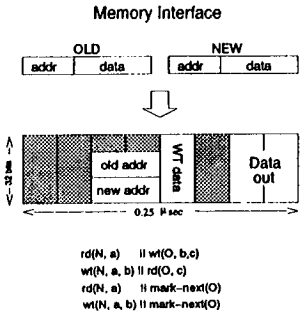


Processor
- o derived from X-FSM only.
- o SPEC close to VLISP machine?

Issues in Verification

- Higher level behavior specification
- Memory model
- Process synchronization
- Data abstraction hierarchy
- Multiple views

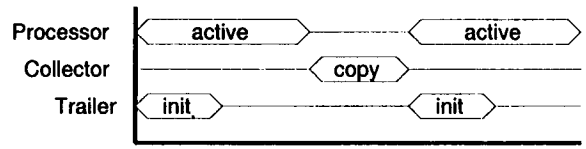
Memory Model



Data Abstraction

- semantic model: ρ: *Var* → *Val*
- compiler factorization: ρ_c: *Var* → *StaticOffset*
  ρ_r: *StaticOffset* → *Store* → *Val*
- Operational model: Rep[ρ_r] = *List(Vector)*
- Implementation model: *List*: *HeapReference*
  Vector: (*HeapReference*, *Offset*)
- Representation: *HeapReference*: (*Tag*, *Address*)
  Offset: *Address*
  Tag: {list, int, . . .}
  Address: $\mathcal{Z}_{2^{24}}$
- HW Realization: *Address*, *Offset*: *BitVector*(24)
  Tag: *BitVector*(8)

Process Synchronization

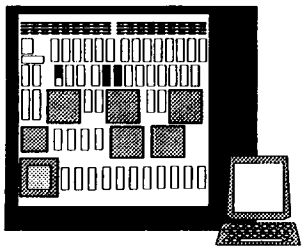


- In Scheme Machine, heap integrity is guaranteed by hardware, with overhead absorbed (to be measured) by the parallel sweep.
- In VLISP, heap integrity is guaranteed by the compiler, but is interruption considered?

Multiple Views

- The CPU sees the heap as an assorted collection of cells, including numbers, list cells of various flavors, vectors, strings, *etc.*
- The allocator and collector see the heap as an array of words subject to certain invariants.
- The invalidator sees the heap as a sequence of bits.

Incremental Construction of Hardware



Summary and Status

- Design derivation extends and adapts programming methodology to hardware targets
- Language implementation, Scheme in this case, is a standard example,
  - exposes hard (higher order) modeling issues
  - a form of closure/completeness (Scheme Machine derivation & emulation could be run on the Scheme Machine).
- the Scheme Machine system specification exposes new verification problems in process coordination, algorithmic correctness, ...
- the design environment explores the gap between modeling and implementation.
- components of the Scheme Machine, or their variations are viable products.

Incremental Construction of Hardware

- The software specification of the processor runs directly against both the software specification of the heap and (with functional test points exposed) the hardware prototype.
- The software specification of the processor, allocator, and collector, run directly against both the modeling environment's heap image and the hardware realization.
- We are seeking to develop an environment where modeling, simulation, emulation, and realization are closely integrated with verification.
- We are seeking to integrate multiple reasoning tools to apply to the verification problem.