Hardware Scripting in Gel

Jonathan Bachrach
MIT CSAIL
32 Vassar Street, Room 227
Cambridge, MA 02139
Email: bachrach@mit.edu

Dany Qumsiyeh
MIT CSAIL
32 Vassar Street, Room 386C
Cambridge, MA 02139
Email: danyq@mit.edu

Mark Tobenkin
MIT CSAIL
32 Vassar Street, Room 386C
Cambridge, MA 02139
Email: mmt@mit.edu

***Abstract*—Gel is a hardware description language that enables quick scripting of high level designs and can be easily extended to new design patterns. It is expression oriented and extremely succinct. Modules are described as functions and composed through function calls. Types and bit widths are inferred automatically to guarantee correctness. Together these features reduce hardware development time, allowing complex designs to be scripted quickly. A simulator and logic analyzer are available to help in the development process. A compiler has been developed that translates Gel to Verilog, and a number of applications have been demonstrated. This paper introduces the core language, demonstrates its extensibility, and shows how design patterns can easily be created. Finally, we compare a few applications written in Gel against equivalents written in Verilog.**

***Index Terms*—hardware description, programmable gate arrays, programming languages, compilers**

I. Introduction

FPGAs fill an important gap between software and custom hardware: they offer the quick turnaround time of software with the runtime efficiency of custom hardware. Custom hardware is important for certain applications that demand low energy consumption, low latency, parallelism, and/or small packaging. Typical applications include software radios, medical imaging, computer vision, cryptography, computer hardware emulation, and digital signal processing.

Design time has become the bottleneck of hardware development, and hardware design still takes much longer than software design. Hardware description languages such as Verilog (and VHDL) offer software specifications of hardware, but unfortunately have many shortcomings. Modules are difficult to reuse, as they often depend on particular timing and wiring constraints. Module parameterization is also limited, often requiring separate generator programs and scripts. Current popular languages tend to be tedious, long winded and brittle.

The Gel approach is to script hardware, writing descriptions functionally and inferring types automatically. In so doing, Gel descriptions become concise and highly composable, and avoid the need for explicit wires. Gel has automatic type inference allowing the user to eliminate most bit width declarations. The compiler supports extensive partial evaluation and common subexpression elimination. Finally, an extensible core library is provided.

Gel has been used to build a number of hardware components including audio, video, and processor cores. These were developed with the help of a simulator that includes logic analyzer functionality. Designs can be viewed graphically and traced, triggered, and single stepped.

This paper provides a first overview of Gel.

II. Assumptions and Design Choices

In its current form, Gel makes a number of assumptions that simplify its implementation without greatly restricting its usefulness. In particular, it assumes directed input and output signals and a global synchronous clock. Furthermore, it is not concerned with spatial layout, which is left to vendor-specific design tools. In the future, we look towards supporting bidirectional signals and asynchronous circuits.

Gel is built out of a simple core language with a minimum number of syntactic constraints. This philosophy provides great flexibility for implementing new design patterns. We would like the language to be agnostic to particular hardware design paradigms. Each paradigm and design pattern can be built as an independent layer on top of core Gel and used when appropriate. Simple and low level designs as well as large systems and high level designs are possible within the same language.

Finally, Gel utilizes Scheme as a host language, but could be implemented in other languages as well. Scheme was chosen for its functional programming support, simple syntax, and macros. In the future, Gel may be implemented as an independent compiler, rather than a construct within Scheme.

III. Scheme Basics

Gel incorporates the syntax of Scheme, so the following is a brief introduction to Scheme. This functional language uses a simple and consistent prefix syntax, where syntactic forms called *s-expressions* are made up of numbers, names and lists:

```
sexpr == number | name | list
list  == '(' sexpr ... ')'
```

Evaluation proceeds recursively according to the contents of the s-expression. Numbers are self evaluating, variables look up values in their local environments, and lists are evaluated according to their first element. If the first element of the list

is one of a number of reserved names, then its evaluation proceeds according to a special rule. Otherwise the list is considered a function call, and the function and arguments are evaluated and then applied. The special forms used are `quote` which returns its argument unevaluated, `lambda` which introduces an anonymous function, `let` which introduces initialized local variables, `define` which introduces initialized variables and functions, and `if` which evaluates one of two expressions depending on a predicate:

```
number
variable
('quote' name)
('lambda' parameters value)
('let' ((name init) ...) value)
('define' name value)
('define' (name parameter ...) body ...)
('if' predicate consequent alternative)
(function argument ...)
```

Beyond the basic arithmetic, Scheme has a number of powerful list functions:

```
('apply' f arg ...)
  (apply + (list 1 2 3)) = (+ 1 2 3)
('map' f list ...)
  (map + (list 1 2) (list 3 4)) = (list (+ 1 3) (+ 2 4))
('range' min max step)
  (range 0 10 2) = (list 0 2 4 6 8)
('fold-right' f init list)
  (fold-right + 0 (list 1 2 3)) = (+ 1 (+ 2 (+ 3 0)))
('fold-left' f init list)
  (fold-left + 0 (list 1 2 3)) = (+ (+ (+ 0 1) 2) 3)
```

where `apply` calls the given function with arguments from the given args and list elements, `map` produces a list of function applications with arguments formed from successive elements of given lists and `range` produces a list of numbers starting at `min`, incrementing by `step` and strictly less than `max`. `fold-left` and `fold-right` accumulate successive elements of a list using a binary function `f`. Consult [1] for more information on the Scheme language.

IV. Gel Core

This section explains the primitive constructs of the language. The following components are all that need be implemented by a Gel compiler, as the more advanced patterns can be defined in terms of these. In its current form, Gel uses Scheme as a host language, where Gel primitives are defined in Scheme so as to construct appropriate hardware graphs. The primitive data elements of Gel are signals representing buses of wires. These primitives and the compositional elements of the language are described below.

A. Literals

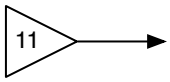
Literals are written in Scheme syntax. Example integers are specified in the usual way as follows:

```
0
11
-14
```

with

```
11
```

producing the following hardware:

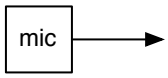


B. Input and Outputs

Inputs and outputs are represented as Gel functions. For example, the `mic` input would be used as follows:

```
(mic)
```

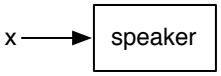
producing the following hardware:



The `speaker` output function would be used as follows:

```
(speaker x)
```

and produce the following hardware:



outputting the `x` signal to the speaker.

C. Tuples and Structures

Signals can be combined into bundles using tuples:

```
(tup (mic) 1)
```

and tuple elements can be extracted using `elt` and a constant index. For example, the following

```
(elt (tup (mic) 1) 0)
```

would produce the same output as `(mic)`.

Named tuples can be defined using `h:define-struct` [1] as follows:

```
('h:define-struct' struct-name field-name ...)
```

and would produce a structure constructor and field accessors. For example, a signal with an end of signal flag could be defined as follows:

```
(h:define-struct segment val eos?)
```

and would be equivalent to:

```
(define (segment val eos?) (tup val eos?))
(define (segment-val  s) (elt s 0))
(define (segment-eos? s) (elt s 1))
```

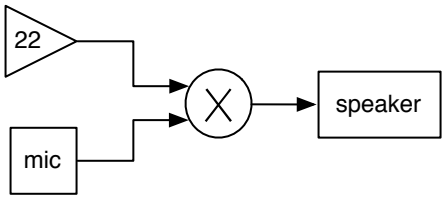
[1]The `h:` prefix signifies the Gel package, and distinguishes it from Scheme's structure definition form.

D. Combinational Logic

More complicated Gel computations can be specified using functional composition. For example,

```
(speaker (h:* 22 (mic)))
```

produces the following hardware



which outputs an amplified microphone value to the speaker and

```
(speaker (h:if (h:> (mic) 0) (h:- (mic)) (mic)))
```

computes the absolute value of the microphone input. Expressions can be built using the full set of Verilog operators such as:

```
h:+   h:*   h:-
h:<<  h:>>
h:not h:and h:or
h:>   h:>=  h:==  h:<   h:<=
h:if
```

The `:` operator can be used to extract bit fields and `cat` can be used to concatenate bit fields:

```
(: val {len | start end})
(cat val ...)
```

Expressions written in Gel are implicitly parallel. For example, two parallel square waves can be summed together on the same line:

```
(speaker (h:+ (square-wave 2) (square-wave 1)))
```

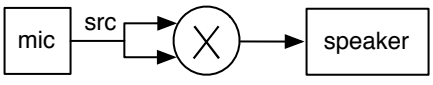
where the value of the entire `par` expression is the value of `e_n`.

E. Fan Out

Hardware outputs can be fanned out with the Scheme `let` form. For example, the following:

```
(let ((src (mic))) (h:* src src))
```

produces the following hardware:



and parameters can be bound to variables and used within expressions:

```
(let ((fac 22)) (h:* fac (mic)))
```

The `h:let` special form allows tuple extractions:

```
(h:let (((tup val ready?) (serial-read rx)))
  (h:if ready? val -1))
```

using a tuple pattern on the left side of the let binding, and bit field extractions:

```
(h:let (((cat (: op 8) (: src 8)) (read-inst insts)))
  (h:if (h:== op op-led) (led src) ...))
```

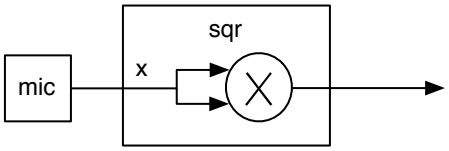
using a `cat` pattern, where `op` and `src` are 8 bit fields in instructions. The `h:let*` form is the sequential binding version of `h:let`, meaning that successive bindings have previous bindings in scope.

F. Abstraction

Scheme functions can be used to define hardware abstractions. For example, signals can be squared as follows:

```
(define (h:sqr x) (h:* x x))
(h:sqr (mic))
```

producing



and inputs can be made positive as follows:

```
(define (h:abs x) (h:if (h:> x 0) x (h:- x)))
(h:abs (mic))
```

G. Registers

The function `reg` constructs registers triggered on the rising clock edge. Registers can be wrapped around signals producing delayed copies:

```
(reg (mic))
```

and then used to access and compare them. For example, the positive edge function compares the current and previous value of a signal to detect the positive edge:

```
(define (posedge x)
  (h:and x (h:not (reg x))))
```

An `n`-clock cycle delay can be constructed recursively:

```
(define ((tap-n x) n)
  (if (= n 0) x
      (reg ((tap-n x) (- n 1)))))
```

When `reg` is applied to a procedure, it automatically composes with it. For example:

```
(map (reg h:+) xs ys)
```

is equivalent to:

```
(map (lambda (x y) (reg (h:+ x y))) xs ys)
```

H. Feedback

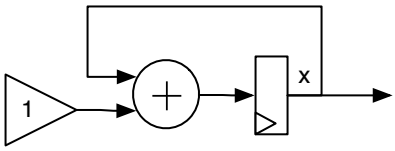
Feedback loops can be created using the `h:letrec` special form, which is similar to Scheme's `letrec` special form allowing variables to be bound and referenced recursively:

```
(h:letrec ((var init next) ...) expr)
```

For example, an infinite counter might be defined as follows:

```
(h:letrec ((x 0 (reg (h:+ x 1)))) x)
```

producing



However, we cannot build counters with infinite size. Section VI discusses how to specify the precision of feedback variables so that they can actually be compiled.

Single variable loops can be abbreviated using the `rep` form:

```
(rep var init next)
```

which is equivalent to:

```
(h:letrec ((var init next)) var)
```

Using `rep`, the counter can be defined more simply as:

```
(rep x 0 (reg (h:+ x 1)))
```

A finite counter of a given range can be constructed as follows:

```
(define (counter max)
  (rep count max (overflow (h:+ (reg count) 1) max)))
```

with `overflow` defined as:

```
(define (overflow n max)
  (h:if (h:> n max) 0 n))
```

Counters can be used to define pulse trains by outputting `1` when the counter reaches `0`:

```
(define (pulse n)
  (h:== (counter (- n 1)) 0))
```

From there, a square wave can be specified in terms of toggling at each pulse:

```
(define (square-wave period)
  (toggle (pulse period)))
```

where toggling is defined as alternating between `0` and `1` based on a pulse input:

```
(define (toggle p)
  (rep x 0 (reg (h:if p (h:not x) x))))
```

`default` specifies the reset value for a feedback variable:

```
(define (default x init)
  (if *reset* init x))
```

and `*reset*` is the reset signal defined as a fluid variable in Scheme.

I. Memory

Gel provides a convenient interface to block RAMs. ROMs are defined as a function which takes an address and returns the associated data. `rom` constructs such a function from provided data:

```
((rom data) addr)
```

The following example steps through the contents of a ROM with three entries:

```
(define myrom (rom (list 11 22 33)))
(myrom (counter 2))
```

producing `11, 22, 33, 11, 22, 33, ...` We can create an `n`-value sine lookup table using a ROM initialized as follows:

```
(define (sin-table A n)
  (rom (map (lambda (t) (round (* A (sin t))))
            (range (- pi) pi (/ (* 2 pi) n)))))

(define (sin-wave A n)
  ((sin-table A n) (counter n)))
```

RAMs are like ROMs except that they allow writing. Their interface is curried to match the ROM function signature once write data is provided as follows:

```
(((ram size) waddr wdata we) raddr)
```

where `waddr` is the write address, `wdata` is the write data, and `we` is the write enable signal. RAMs could be used to build an audio recorder where a button switches between recording or playback:

```
(define (audio-recorder n)
  (let ((addr (counter n)))
    (((ram n) addr (mic) (button)) (h:if (button) 0 addr))))
```

V. Advanced Gel

The previous section demonstrated how the special forms, expressions, and functional abstraction of the Gel core can be used to quickly build up a number of highly reusable modules. We now introduce more powerful functional and syntactic abstractions that can be used to create new design patterns and more sophisticated applications.

A. Meta Programming

Expressions can be constructed using the full power of Scheme. For example, a tapped delay line can be constructed by mapping `tap-n` across a list of numbers:

```
(define (taps x n)
  (map (tap-n x) (range 0 n 1)))
```

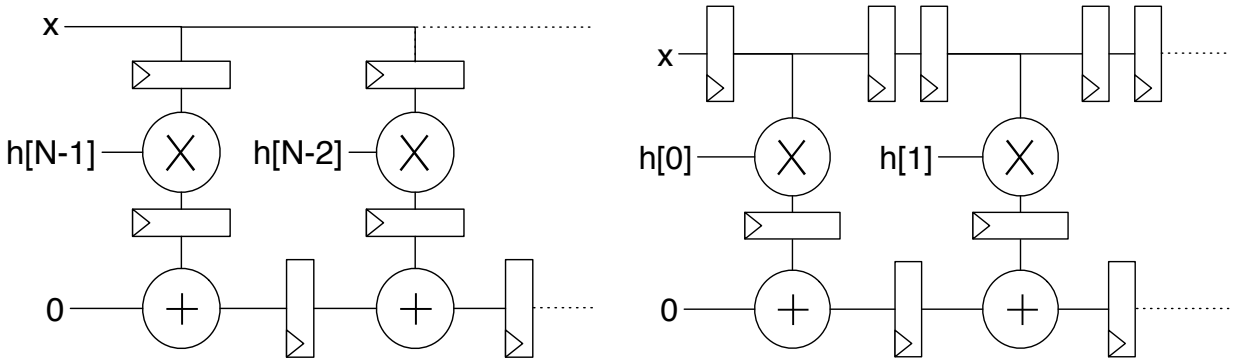
With a list of coefficients, one can now quickly construct the traditional "inner-product" FIR filter. Given a signal x and an impulse response h of duration N, the general FIR filter equation can be written:

$$y[n] = \sum_{k=0}^{N-1} x[n-k]h[k]$$

and implemented in Gel as:

```
(define (inner-product-fir hs x)
  (let ((xs (taps x (length hs))))
    (apply h:+ (map h:* hs xs))))
```

More sophisticated FIR topologies also benefit greatly from Gel's automatic bit-width inference. The following code presents both the transposed and systolic FIR topologies (see [2]), shown in Fig. 1:



(a) Schematic view of a Transposed FIR Realization.

(b) Schematic view of a Systolic FIR Realization.

Fig. 1. Example FIR Filter Topologies

```
(define (transposed-fir hs x)
  (fold-right (reg h:+) 0
    (map (lambda (h) (reg (h:* h (reg x)))) hs)))

(define (systolic-fir hs x)
  (let ((taps (map (tap-n x) (range 1 (* 2 (length hs)) 2))))
    (fold-left (reg h:+) 0
      (map (reg h:*) hs taps))))
```

Other design patterns such as balanced trees are also easy to express. The following implements an n-way mux as a balanced decision tree:

```
(define (h:ref sel . args)
  (let* ((len (length args))
         (mid (quotient len 2)))
    (if (= len 1)
        (car args)
        (h:if (h:< sel mid)
          (apply h:ref
            (cons sel (sublist args 0 mid)))
          (apply h:ref
            (cons (+ sel mid) (sublist args mid len)))))))
```

Thus, these would be equivalent:

```
(h:ref a x y z)
(h:if (h:< x 1) x (h:if (h:== x 2) z y))
```

Gel's meta-programming facility can be used to build arbitrarily complex parameterized hardware structures such as tessellations, shuffle networks, etc. Unlike solutions involving explicit type parameterization [3], Gel's automatic type inference allows combinators to be built easily and independently of the input/output types.

B. Parallel Feedback

Section IV-H presents simple feedback forms (e.g., `h:letrec` and `rep`) that allow variables to be updated independently, albeit in the context of each other. Sometimes variable update expressions are related or share common subexpressions, and it is more convenient to update the variables in parallel. However, not all variables need to be updated each clock cycle. Our solution is to use keyword values with defaults. The `cyc` form introduces a parallel state variable update mechanism as follows:

```
('cyc' (go (var init [default]) ...) update output)
```

where `go` is the parallel state constructor taking keyword arguments, `var` is a state variable name, `init` is its initial value, and `default` is its optional default update value, `update` is an update expression that uses `go` to construct a next value, and `output` is the output value produced. The result can be considered equivalent to:

```
(h:letrec (((tup var ...) (tup init ...) update))) output)
```

where `update` evaluates to a `tup` expression with updates for each variable.

An example usage of `cyc` is a simple two-stage RISC processor composed of a set of registers and a simple three operand instruction format:

```
(define-enum
  op-noop op-lit op-add op-lt? op-eq?
  op-bri op-bra op-ld op-st)

(define w 8)

(define (cpu code n-regs mem-size)
  (cyc (go (pc (: 0 w) (h:+ pc 1)) (dst (: 0 w) w)
           (val (: 0 w)) (rwe 0 0)
           (mwa (: 0 w)) (mwe 0)  (mre 0))
    (h:let* ((inst ((rom code) pc))
             ((cat (: op w) (: ra w) (: rb w) (: rc w)) inst)
             (regs ((vec n-regs) dst val rwe))
             (b    (h:if (h:== dst rb) val (regs rb)))
             (c    (h:if (h:== dst rc) val (regs rc)))
             (m    (((ram mem-size) mwa val mwe) (h:+ b c))))
      (h:if mre
            (go 'dst ra 'val m 'rwe 1 'mre 0)
        (h:case op
          ((op-lit)  (go 'dst ra 'val (cat ra rb) 'rwe 1))
          ((op-add)  (go 'dst ra 'val (h:+ b c) 'rwe 1))
          ((op-lt?)  (go 'dst ra 'val (h:< b c) 'rwe 1))
          ((op-eq?)  (go 'dst ra 'val (h:== b c) 'rwe 1))
          ((op-bra)  (go 'pc (h:+ pc (cat rb rc))))
          ((op-bri)  (go 'pc (h:+ pc (h:== c 0) 1 (cat ra rb))))
          ((op-ld)   (go 'dst ra 'mre 1 'pc pc))
          ((op-st)   (go 'val b  'mwa (h:+ c ra) 'mwe 1))
          ((op-noop) (go))
          (#t        (go 'pc pc)))))
    val))
```

where `define-enum` defines constants of increasing non-negative values and `w` defines the width of a instruction field. The processor maintains a program counter `pc`, and destination register index `dst`, a value `val`, a register write enable flag `rwe`, a memory write address `mwa`, a memory

write enable flag `mwe`, and a a memory read enable flag `mre`. The main body of the parallel feedback performs instruction decoding and a potential memory read, with each branch of the code updating the state variables needed to read/write the registers/memory.

C. Finite State Machines

Finite state machines (FSM) can be build out of the `cyc` parallel feedback form by introducing an implicit current state variable, replacing the update expression with a current state dispatch, and by automatically defining the state constants. The `fsm` form becomes:

```
('fsm' (go state (var init [ default ]) ...)
    ((state expr) ...)
  output)
```

Now that we have FSM's, a serial port reader can be defined quite simply as follows:

```
(define (serial-read rx period)
  (fsm (go (word (: 0 8)) (bit (: 0 3))
           (is-ready 0) (n period))
     ((stopped
        (h:if (h:not rx)
              (go wait 'n (h:/ period 2) 'is-ready 0 'bit 0)
              (go stopped)))
      (wait
        (h:if (h:== n 0)
              (go read 'word (h:bitior (h:<< word 1) rx))
              (go wait 'n (h:- n 1))))
      (read
        (h:if (h:== bit 8)
              (go stopped 'word word 'is-ready 1)
              (go wait 'bit (h:+ bit 1) 'n period))))
    (tup word is-ready)))
```

where the state machine paces the reading of the signal. The serial reader starts in the stopped state waiting for `rx` to go low, at which point it goes to the wait state with the wait counter `n` set to half a period. The wait state decrements `n` until it hits zero at which time it goes to the read state sampling the `rx` signal and incorporating it into the word. When nine bits have been accumulated, the ready signal is turned on and the state machine is reset to the stopped state.

D. Segments

FSM's are a general purpose way of sequencing logic, but sometimes simpler mechanisms can be used to compose sequencers. In this section, we introduce finite signals, called *segments* which are signals with an associated end of segment flag. In the audio world, they could be used to sequence between audio sources or effects. For example, the following:

```
(loop (wait (* 2 sec) (square-wave p))
      (wait (* 3 sec) (sin-wave p n)))
```

would produce a square wave for 2 seconds followed by a sin wave for 3 seconds. This would be clumsy to represent as a FSM but is succinctly expressed using segments.

We can define a segment as a structure of value and end of the segment signals as follows:

```
(h:define-struct segment val eos?)
```

We could then truncate a signal by attaching a counter as the end of segment signal as follows:

```
(define (wait delay x)     (segment x (counter delay)))
(define (until trigger x) (segment x (h:not trigger)))
(define (while trigger x) (segment x trigger))
```

From here we can sequence between a list of segments by switching from segment to segment upon the end of segment signals as follows:

```
(define (reset-thunk reset thunk)
  (fluid-let ((*reset* reset)) (force thunk)))

(define (seq fseg1 . fsegs)
  (let ((n (+ (length fsegs) 1)))
    (cyc (go (index (: 0 n))
             ((tup val eos?) (reset-thunk #t fseg1)))
      (let ((segs (map (lambda (s)
                         (reset-thunk (posedge index) s))
                       (cons fseg1 fsegs))))
        (h:let (((tup nval neos?) (apply h:ref index segs)))
          (go (default (overflow (h:+ index neos?) n) 0)
              (tup nval (negedge (h:== index (- n 1))))))))
      (segment val eos?))))
```

where index is used to select segments and is updated on the posedge of the end of segment signal, segments are represented as thunks (i.e., zero argument anonymous functions) which are evaluated in a proper reset environment such that at the beginning of each segment, the `*reset*` signal is brought high, signaling `default` expressions or `h:letrec` variables to reinitialize. The sequence is itself a segment which produces an end of signal when it has sequenced through each constituent segment once.

Finally, `loop` can be built out of `seq` by never ending:

```
(define (loop . segments)
  (while #t (segment-val (apply seq segments))))
```

E. Samples

Often signals must be sampled unevenly and these discrete signals must be composed through operations. We called these signals *samples* and they can be represented as a tuple with value and ready signals:

```
(h:define-struct sample val ready?)
```

Continuous signals can be converted to samples through sampling:

```
(define (every x delay)
  (sample x (pulse delay)))
```

and samples can be accumulated with the following:

```
(define (accum f init sample)
  (rep (tup val n) (tup init 0)
    (h:let (((tup sval ready?) sample))
      (h:if ready?
          (tup (f (reg val) sval) (h:+ (reg n) 1))
          (reg (tup val n))))))

(define (bit-accum init sample)
  (accum (lambda (word bit) (h:bitior (h:<< word 1) bit))
         init sample))
```

which produces a tuple of sample values and number of samples. Now we are in a good position to rewrite the original serial reader code as follows:

```
(define (serial-read rx period)
  (loop
    (delay (while rx (sample 0 #f)))
    (delay (wait  (h:/ period 2) (sample 0 #f)))
    (delay (h:let (((tup word n)
                    (bit-accum (: 0 8) (every rx period))))
             (until (h:== n 8) (sample word (h:== n 8)))))))
```

where `delay` is the Scheme thunk creating operator.

VI. Compilation

Gel compiles a design to a single Verilog module, which can then be mapped to hardware using existing design tools. The first stage of Gel compilation generates an Abstract Syntax Tree (AST) with stubs in place of all feedback variables. The second step resolves all stubs and turns the AST into a graph. The third phase infers the types and bit widths of all the nodes. Fourth, the graph is optimized, replacing operations over constant inputs with their application and eliminating common subexpressions. The last phase generates Verilog from the graph. In the next subsections, we present the type inference, optimization, and code generation phases.

A. Type Inference

In hardware, signals must be represented by a fixed number of wires, using a representation such as signed or unsigned binary values. Most hardware description languages require this type and bit-width to be specified throughout the design. Unless specified, these parameters are automatically inferred in Gel to guarantee the correctness of a calculation. In feedback cases where bit-widths are difficult to infer, the compiler may require that a type declaration be asserted somewhere in the feedback path.

In Gel, the bit width of a literal is the minimum number of bits used to represent it in two's complement form, and the sign of literal wire is determined from the sign of the literal.

Signals can be either signed or unsigned fixed width numbers or tuples (`tup`). Tuples are heterogeneous structures of unnamed fields. The number of bits in a given signal can be obtained using `sizeof`:

```
(sizeof (mic))
```

The width and sign of integers can be specified using the `:` operator:

```
(: 0 8)
(: 0 -8)
```

for eight bit unsigned and signed zero respectively. In other words, negative bit widths represent signed numbers.

In general, the bit widths of feedback variables need to be specified, either through an initial value or type declaration:

```
(rep var init|type update)
```

where `type` is composed of one of the following expressions:

```
(type-of val)
(numt width)
(tupt type ...)
```

where `width` can be negative to specify a signed number.

From there, Gel automatically infers types and bit widths assuring sufficient precision for results of operators. Inputs have defined bit widths and sign, and literals have size sufficient to fit their contents and are signed if negative. Numeric operator nodes have bit width sufficient so as not to loose information in the worst case. Sign is propagated through contagiously and conservatively. Tuple types are inferred from the `tup` function and then are propagated through. Feedback variables introduced with `h:letrec` require an initial value or a type declaration to limit the bit width. Finally, outputs have defined bit widths and sign and do the proper conversion to conform to their declaration.

B. Input and Outputs

Hardware inputs and outputs are explicitly notated with `h:input` and `h:output` functions:

```
(h:input name type)
(h:output x name type)
```

and used as follows:

```
(h:input mic -16)
(h:output 0 speaker -16)
```

Input and output functions can be generated as:

```
(define-input  mic -16)
(define-output speaker -16)
```

and used as follows:

```
(speaker 0)
```

C. Optimizations

Gel performs a number of optimization to simplify the data flow graph. The type inferencer propagates constancy through the graph and the optimizer then replaces operator nodes tagged as constant by the type inferencer with the value computed by applying the operation on the constant inputs. Tuple `elt` operations are replaced with the corresponding argument of the generating `tup` call. This replaces all tuples with their constituent signals and makes tuples a very lightweight mechanism. Finally, redundant subexpressions are identified and replaced with a single copy.

D. Running the Compiler

Compiling to Verilog is performed by calling the compiler on a list of input / output arguments:

```
(compile { input | output } ...)
```

For example, we could compile a simple audio pass through example as follows:

```
(compile (speaker (mic)))
```

E. Code Generation

Code generation proceeds as multiple passes over the data flow graph. First, inputs, outputs, wires and registers are collected and named. Then, wire and register declarations are emitted. Next, combination logic is generated as a series of assignments to wires. Finally, registers are updated in a posedge clocked always block.

As an example, the following Gel program:

```
(speaker (counter 8))
```

turns into the following Verilog code:

```
module top (speaker, clk, reset);
  reg [3:0]T0;
  output [15:0]speaker;
  input [0:0]reset;
  input [0:0]clk;
  wire [4:0]T3;
  wire [0:0]T4;
  wire [4:0]T5;
  wire [3:0]T6;
  wire [3:0]T7;

  assign out = T0;
  assign T3 = T0 + 1'd1;
  assign T4 = T3 > 4'd8;
  assign T5 = T4 ? 1'd0 : T3;
  assign T6 = T5[6'd3:1'd0];
  assign T7 = reset ? 4'd8 : T6;
  always @ (posedge clk) begin
    T0 <= T7;
  end
end
```

VII. Simulator

Gel provides a simulator that makes it easier to develop and debug designs. Signals can be traced and viewed in a variety of formats as shown in Fig 2(a). Gel allows many signals which would require names in Verilog to be anonymous. To help with debugging these signals, named probes can be wrapped around expressions causing the corresponding signal to be traced in the simulator. For example,

```
(toggle (probe pulse (pulse period)))
```

would add a pulse trace to the simulator. Triggers can be set when traces equal satisfy user defined conditions. The simulator can single step or continue until triggers are hit.

A graphical view of the circuit, shown in Fig. 2(b) can be displayed and traces for graphical nodes can be interactively added by clicking on the nodes. The user can control the view using zooming and panning. Finally, the circuit can be single stepped and breakpoints can be inserted.

VIII. Related Work

Many other strategies for describing hardware at a high level have been proposed. The most common approaches are outlined below, and compared to Gel.

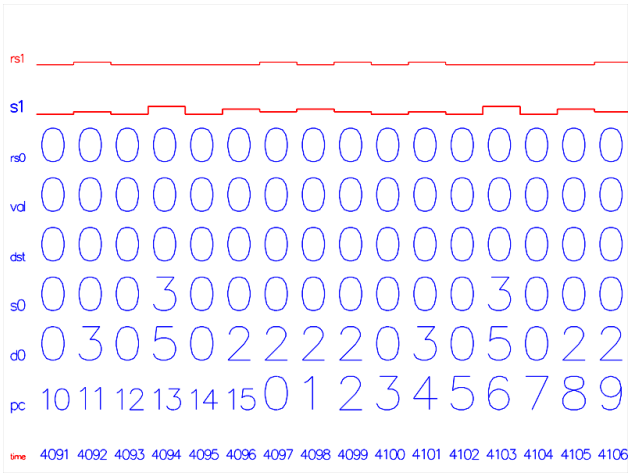
Many languages fall under the category of high level language to HDL translators. These attempt to compile programs in common high level languages (possibly with some language extensions) to hardware, taking advantage of any implicit parallelism. Many HDLs such as SystemC [4] and Altera's C2H [5] compile C-like language to hardware. Shard [6] compiles recursive Scheme programs to a data-flow hardware design. These languages are forgiving of programmers who only have experience with software, but they limit expressibility for those familiar with hardware and the associated design and layout tradeoffs. While a given calculation or behavior may be easy to express, a specific hardware topology or timing arrangement may be impossible to describe. Gel is flexible enough to express most hardware designs, however, and allows high level abstractions which make design patterns easy to express.

Some languages are based on particular design patterns for hardware. BlueSpec System Verilog (BSV) [3] uses guarded atomic actions to make designs easier to analyze. Esterel [7] uses event-based statements to program hardware for reactive systems. DIL [8] is an intermediate language targeted at stream processing and hardware virtualization. Design patterns like these can make certain applications or functions extremely natural to express, but may not be the best for all situations. Gel is intended to be a simple platform that can be easily extended to capture these useful design patterns. By using such a flexible platform, projects can incorporate whichever design pattern or language most easily expresses the design. Large hardware projects could have sub-circuits each programmed in different ways, appropriate to the subproblem.

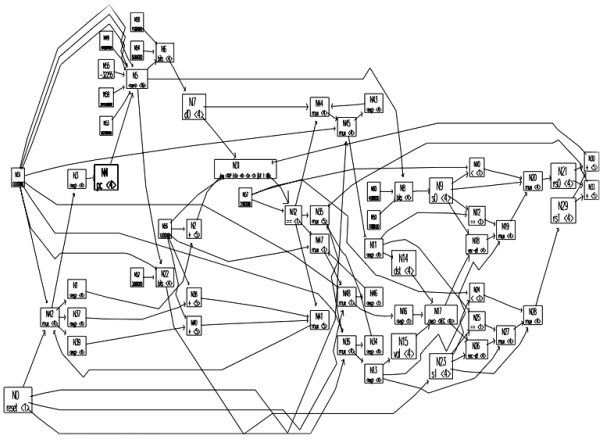
Other languages focus on specifying the spatial layout of designs. Lava [9] is a Haskell-based HDL where hardware layout is explicit and complex interconnect and layout patterns can be abstracted. PamBlox treats C++ classes as hardware modules, with placement methods that can be inherited or overridden. Gel is intended not for optimizing the layout of a preexisting design, but for quickly scripting hardware and capturing high level design patterns. While Gel could be extended to allow placement instructions, we hope to be able to build abstractions which allow you to work with signals and objects at a high level, without thinking of wires or the FPGA placement details.

Finally, some languages embed a familiar hardware programming model into another language, so as to overcome the macro limitations of languages such as Verilog. Verischemelog [10] provides a Scheme syntax for specifying modules in a similar format to Verilog. JHDL [11] equates Java classes with modules. HML [12] uses standard ML functions to wire together a circuit. These approaches allow familiar and powerful languages to be macro languages for the common net-list hardware description. While these languages are theoretically as capable as Gel, they effectively require designs to be described in Verilog's module and bus format.

Gel combines a number of features touched upon in other languages. DIL, Esterel, and HML have some form of automatic typing or bit-width inference. BlueSpec, Lava, Shard, Verischemelog, and HML are based on functional languages.



(a) Logic analyzer traces for the RISC processor.



(b) Circuit view of the RISC processor.

Fig. 2. The Gel simulator.

TABLE I
SYNTHESIS COMPARISON OF GEL VS. VERILOG

	LUTs		MHz	
	Gel	Verilog	**Gel**	Verilog
Serial RX	**49**	28	**195**	230
Transposed FIR	**254**	251	**194**	198
Systolic FIR	**384**	368	**194**	202

High level language translators such as HandelC, SystemC, Altera's C2H, and Shard represent signals as simple symbols which can be operated on, rather than wired up as a net-list. Each of these languages has limitations or design principles which constrain the ways a programmer can express hardware designs. While Gel may not be right for all designs, we believe that the combination of these features in a simple and small language allows Gel a unique power of expression and abstraction.

IX. STATUS AND RESULTS

Currently, Gel is entirely written in MIT Scheme. It runs on any platform MIT Scheme runs on, which is Windows, Linux, and MacOSX. Finally, it is unreleased.

Table 1 presents a comparison of several example applications from this paper and hand-written Verilog equivalents. The transposed and systolic FIR examples were generated for a 16-bit input, with 9 taps ranging from 7 to 14 bits in magnitude. The serial receiver was synthesized for a 38.4 Kbps baud-rate. All of these designs were built targeting a Xilinx Spartan 3E xc3s500e-fg320, clocked at 50 MHz. For the FIR applications, Gel performs favorably; the automatic type inference manages the bit growth as necessary for FIR designs. However, the serial receiver performs poorly compared to the hand-written Verilog equivalent. The Verilog synthesis tools inferred a subtractor, instead of a down counter, from the `serial-read` function. Future compiler revisions will target overcoming this limitation.

The cost of automatic type inference is that certain arithmetic instructions will grow the data path, though in a conservative fashion. Feedback data widths are trimmed back based on either the width of an initial value or a declaration. The user can further control the growth through judicious placement of declarations.

Because of the automatic type inference, Gel code is much more succinct, reusable and composable. For example, the serial port reader code is 78 lines of Verilog and 16 lines of Gel in FSM form and 7 lines in segment form. Furthermore, the Gel code is built out of components that can be reused in many other circuits.

X. NEXT STEPS

Gel appears to be descriptive and succinct for the examples we have presented, but we need to tackle more complex applications. Bit-width inference enhances the composability of modules. However, we have yet to devise ways to combine modules without worrying about timing and data formats. Gel allows for powerful abstractions, but more abstractions are needed for things such as transactions and pipelining. We have been experimenting with a mechanism for automatic pipelining, but more work is needed.

We have implemented a number of design patterns. Early results are encouraging, but more work is needed. We will endeavor to apply Gel to more paradigms and raise the level of abstractions so that, for example, video processing can be expressed in a concise algebraic form.

So far Gel is very efficient for smaller designs, but needs a story for virtualization for larger ones. We think that we can parameterize the virtualization in a natural and platform specific manner. Previous work on Gooze [13] gives us some measure of confidence.

While the distinction between the host and embedded language is at times cumbersome, having a host language is a net win. We would like to at least explore moving away from Scheme and unifying the language to better understand the tradeoffs.

XI. Conclusion

Gel provides a natural and concise scripting solution for hardware. It supports polymorphic signals that are well suited for a variety of hardware types. Automatic type inference allows the programmer to specify modules in a very reusable manner. The meta-programming facility allows the user to define novel and powerful libraries for specific hardware paradigms and design patterns, keeping the core language small and paradigm agnostic. Applied to FPGAs, these pieces together allow hardware development time to better match the quick turnaround expected only for software. A simulator is provided permitting convenient and interactive hardware development. We have built a number of hardware examples, and shown that their succinct description still produces performance comparable to current standards. While more advanced examples will be needed to prove Gel's utility in real applications, current results indicate that Gel is a promising new approach to hardware design.

Acknowledgments

This work has been supported by a NSF bio inspired computing grant CCF-0621897. We would like to thank Dominic Rizzo for his helpful comments and for designing and implementing our demo FPGA board.

References

[1] H. Abelson and G. Sussman, *The Structure and Interpretation of Computer Programs*. MIT Press and McGraw-Hill, 1985.

[2] Xilinx Inc., *DSP: Designing for Optimal Results*. Edition 1.0, 2005. [Online]. Available: http://www.xilinx.com/publications/books/dsp/dsp-book.pdf

[3] R. Nikhil, "Future programming of FPGAs," 2004. [Online]. Available: http://www.bluespec.com/pdfs_and_docs/NikhilPanelFPGA2004.pdf

[4] P. R. Panda, "SystemC: a modeling platform supporting multiple design abstractions," in *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*. New York, NY, USA: ACM, 2001, pp. 75–80.

[5] D. Lau, O. Pritchard, and P. Molson, "Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions," in *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 45–56.

[6] X. Saint-Mleux, M. Feeley, and J.-P. David, "SHard: a scheme to hardware compiler," in *Proceedings of the 2006 Scheme and Functional Programming Workshop*, 2006.

[7] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992. [Online]. Available: citeseer.ist.psu.edu/berry92esterel.html

[8] S. Goldstein and M. Budiu, "Fast compilation for pipelined reconfigurable fabrics," in *ACM/FPGA Symposium on Field Programmable Gate Arrays*, 1999.

[9] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in haskell," in *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 1998, pp. 174–184.

[10] J. Jenning and E. Beuscher, "Verischemelog: Verilog embedded in scheme," in *Proceedings of DSL'99: The 2nd conference on Domain Specific Languages*, October 1999.

[11] P. Bellows and B. Hutchings, "JHDL - an HDL for reconfigurable systems," in *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1998, pp. 175–184. [Online]. Available: citeseer.ist.psu.edu/bellows98jhdl.html

[12] Y. Li and M. Leeser, "HML, a novel hardware description language and its translation to VHDL," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 8, no. 1, pp. 1–8, Feb 2000.

[13] J. Bachrach, "Gooze: a stream processing language," in *Lightweight Languages 2004*, November 2004. [Online]. Available: http://ll4.csail.mit.edu/

[14] M. Budiu, M. Sakr, K. Walker, and S. C. Goldstein, "BitValue inference: Detecting and exploiting narrow bitwidth computations," in *European Conference on Parallel Processing (EUROPAR)*, 1999, pp. 969–979.