Hierarchical Asynchronous Circuit Kompiler Toolkit (HACKT)

David Fang
fangism@google.com

"I don't think it is a technical issue, but an infrastructure support problem. It's the chicken-or-egg question all over again: we cannot easily design asynchronous systems because appropriate tools aren't available. And there are no tools, the EDA houses say, because there is no demand for them."

Bernard Cole, August, 2002 [1]

***Abstract*—The self-timed circuits (also known as asynchronous circuits) community has been divided over styles of circuit design, and consequently has not been able to cross-leverage tools, commercial or open-source. Unification of tools is a challenge because few research groups use the same techniques, description languages, or conventions. Without a singular demand, EDA tool companies have no incentive to develop tools for self-timed circuits. Self-timed circuits tools often follow one of the following outcomes: abandonment (after developers graduate), or kept proprietary as intellectual property. We present HACKT, a self-timed circuits toolset that was forged in academic fires, tempered in industry, proven with production silicon, supports a multitude of design styles, and is already open-source[1].**

I. Self-timed Circuits Overview

Self-timed circuits communicate using handshakes between locally synchronized processes, instead of using a global control signal like a clock. A clock edge indicates when it is safe to sample a signal, and is timed with margins to ensure safety. With self-timed circuits, handshaking protocols use an explicit acknowledgment to indicate when it is safe for a responder to update a signal. Self-timed circuits' behavior can be modeled at a high level of abstraction as concurrent dataflow pipelines. *Data-driven decomposition* and *projection* are transformations that refine concurrent programs into primitive handshake elements using explicit communication of values over channels [2], [3]. Decomposition introduces fine-grain pipelining, which helps achieve high performance. Since timing behavior is independent from correctness, a large class of *slack-elastic* designs have the liberty of adjusting the amount of pipelining for performance and energy optimization [4]. Self-timed circuits are data-driven and self-idling, making them well-suited for energy-constrained applications with bursty activity. Conditional communication elements (e.g. splits, merges) can idle entire paths for further energy reduction. Clockless, handshaked interfaces make self-timed circuits truly modular in design, and simple to integrate and reuse, without the need for frequency matching or clock domain crossing. Self-timed circuits are robust to timing variations (due to process variation, thermal and power supply dynamics), which makes them correct by construction, without the need for global timing closure.

There is an entire conference devoted to the study and application of self-timed circuits, ASYNC [5]. The general self-timed circuits community, however, remains fractured over the multitude of self-timed circuit design families. Tradeoffs exist over the spectrum of design styles, ranging from conservative (fewest timing assumptions, easiest to formally verify) to the aggressive (more effort to verify local timing assumptions). The tools we present are amenable to a wide range of self-timed circuit families.

II. Self-timed Circuits Toolchain

Hierarchical Asynchronous Circuit Kompiler Toolkit (HACKT) is a collection of EDA tools whose original purpose was to aid in designing and verifying quasi delay-insensitive (QDI) self-timed circuits. QDI circuits are the most timing-conservative design discipline that is Turing-complete: they assume no relationship between gate delays, and only isochronic forks on wires [6], [7]. The language used by the tools is so low-level (transistors and wires) that it can be used to describe most digital logic and circuit topologies, thereby making it usable for specifying most self-timed circuit design families, and just as suitable for designing synchronous circuits.

III. Language

A. HAC

HAC (Hierarchical Asynchronous Circuits) is the structural language for expressing a hierarchy of instances and connectivity, without any behavior. Behavior and logic are expressed in two sub-languages: CHP and PRS. HAC syntax is summarized in Appendix A. Despite the name, it is agnostic to circuit design methodology, and suitable for synchronous circuits.

B. CHP

The Communicating Hardware Processes (CHP) language is a dialect of CSP that expresses high-level computation and communication actions [8], [9]. It is primarily used for prototyping self-timed systems without concern for handshaking protocols and other circuit implementation details. In practice, CHP often serves as a concurrent dataflow reference

[1]The HACKT project was developed while the author was affiliated with Cornell University (2004-2008) and Achronix Semiconductor (2008-2014), prior to joining Google.

model for verifying self-timed implementations. CHP syntax is summarized in Appendix B.

C. PRS

The Production Rule Set (PRS) language expresses gate-level logic and circuit topologies. Every rule of PRS is of the form $G \rightarrow S$, where G is a boolean expression, and S is a pull-up or pull-down action. Boolean operators $\&$ and $|$ translate directly to series and parallel connections, and are a convenient shorthand that synthesize directly to SPICE netlists. Synchronous circuits can be expressed using rules that explicitly involve a clock signal, which are typically in flip-flops. PRS syntax is summarized in Appendix C.

D. Example

CHP and PRS do not intermingle, so they are compiled and simulated separately. We illustrate how these languages work using a single-bit pipeline buffer as an example.

Listing 1. CHP specification for a token buffer

```
// buf is a single-place pipeline element
// that passes a value from channel L to R
defproc buf(chan?(bool) L; chan!(bool) R) {
  bool data;
  chp {
    *[ L?(data); R!(data) ]
  }
}
```

The CHP program in Listing 1 specifies the pipeline behavior for a buf, (infinitely read from channel L, and send that data out on channel R) but does not specify its implementation. L and R are native channel types that are only specified to transport a boolean value. Data types such as `bool` and `int` in CHP are abstract, and could be implemented with different encodings.

The PRS program in Listing 2 shows one possible implementation, using a 4-phase handshake protocol with an active-low acknowledgement on channels L and R. A single boolean variable is encoded dual-rail. This variant is called a *weak-condition half-buffer* and is quasi delay-insensitive [10].

For brevity, the Listing 2 omits staticizers (also known as keepers), reset logic, and transistor sizes. Explicit reset logic is needed for functional simulation and verification. Staticizers and transistor sizes are needed for netlist generation, analog simulation and verification, and physical design. Physical design methodology (such as library development and characterization targeting each process node) can be built on top of these tools.

E. Circuit Library

HACKT is distributed with a modest library of circuits in PRS:

- Standard combinational logic elements, SRAM cell
- 4-phase handshake QDI cells: buffers, copiers, splits, merges, alternators

The provided library elements are intended for functional simulation, and defined without transistor sizes.

Listing 2. One possible PRS implementation of Listing 1

```
// dual-rail channel with
// active-low acknowledgement 'e'
// protocol: 4-phase
defchan e1of2 <: int<1> (bool d[2]; bool e) { }

// weak-condition half-buffer implementation,
// without reset
defproc buf(e1of2? L; e1of2! R) {
  _c1of2 _r;
  bool rv;
  prs {
    (:i:2:
      // Mueller C-element: pull-down, pull-up
       R.e &  L.d[i]   -> _r.d[i]-
      ~R.e & ~L.d[i]   -> _r.d[i]+

      // output driver
      _r.d[i]          => R.d[i]-
    )

    // R completion detection (NAND)
    _r.d[0] & _r.d[1]  => rv-

    // L acknowledgement driver
    rv                 => L.e-
  }
}
```

IV. Programs

HACKT's design entry is text only; there is no graphical schematic entry. All tools are invoked from the command-line and text-based. The front-end tools are:

- `haco` compiles source to an un-elaborated object file.
- `haccreate` elaborates the design into another object file. Elaboration also checks connectivity: type, direction, drivers, point-to-point use of channels.

The back-end tools operate on an elaborated object file:

- `hacprsim` simulates PRS (interactive or scripted).
- `hachpsim` simulates CHP (interactive or scripted).
- `hacknet` generates SPICE netlists from PRS.

A. PRS Simulator (hacprsim)

`hacprsim` is the most mature tool in the HACKT suite. `hacprsim` is a discrete event simulator that can be driven interactively in the simulator's shell or scripted. An event is a change in the value of a `bool` in the system, which can cause subsequent events to be scheduled in the global event queue. Gate delays are modeled as the time between when an event is scheduled (guard becomes true) to when the event fires, changing a `bool`'s value.

The following key features of `hacprsim` are useful for verifying self-timed circuits:

- Randomized timing modes, that vary gate delays, are very effective at catching timing assumptions and race conditions missed by designers.
- Detecting event instability (such as a glitch, when a guard of a production rule becomes false before the

output transition occurs). Instability indicates a failure to acknowledge a transition before allowing a guard's evaluation to change, or an implicit timing assumption.
- Detecting interference, contention between opposing pulls. Such errors can occur in non-combinational logic.
- Conservative and contagious X-propagation. X's can arise from instabilities or interference (with zero delay).
- Checking invariant expressions of signals.

The `channel-*` family of commands is dedicated to setting up channel-based environments that interface with the DUT by driving and responding to signal changes. Natively supported protocols include: 4-phase, 2-phase, bundled-data, level-encoded dual-rail, and clocked (posedge, negedge, any-edge) [11], [12]. Simulator-defined channels can be used to source values on an input channel, consume-and-check values on an output channel, or observe transactions on any channel in the DUT's hierarchy. Channel query commands can report the current state or value of a channel, as well as which phase of its protocol it is currently in (and whose turn is next).

For debugging, there are commands to show the status, fanin, and fanout of any signal in the hierarchy. The global event queue is viewable to the user at all times, and also features some commands to re-order or manipulate pending events for directed scenario testing. There are also commands to query for signals that are in various states: 0, 1, X, floating, interfering. The `backtrace` command gives a recent causality chain of events that led to the state of a particular node, and can also be interpreted as a critical path when simulating with realistic delays. The `why-*` family of commands recursively traces drivers at one instant in time to report why a signal is in a particular state, displaying the result in an ASCII tree. The simulator's interpreter offers filesystem-like view of the design hierarchy for convenience (e.g. `cd, pushd, ls`).

`hacprsim` offers control over output verbosity (to stdout), and also supports exporting trace files in its own internal format as well as VCD. Checkpoint saving and restoration are useful features for resuming long-running simulations and performing post-mortem analysis.

`hacprsim` is also available as a VPI co-simulation plug-in that has been demonstrated with Synopsys VCS and Cadence Incisive [13]. Co-simulation combines the benefit of the verification-rich features of languages like SystemVerilog with the diagnostic features and detail-level of hacprsim, and has been crucial for verifying mixed-discipline circuits. The co-simulation module offers all of the features of the standalone simulator except for direct control over the PRS event queue, which is relegated to the host Verilog simulator. The tool suite includes some scripts that help automate connecting signals between `hacprsim` and the corresponding Verilog stub modules.

B. CHP Simulator (hacchpsim)

`hachpsim` is another event-driven simulator for simulating concurrent dataflow execution, whose events consist of value changes and changes in channel states (full, empty). This is most useful for developing the concurrent dataflow models for self-timed circuits. One can test equivalence between two CHP programs, where one may be a decomposed or transformed version of the other. It can also serve as the functional model for verifying implementations in PRS. `hachpsim` has the same style of interactive interpreter as `hacprsim`. `hachpsim` also offers its own trace file format and check-pointing, but does not have its own co-simulation module.

C. Netlist Generator (hacknet)

`hacknet` translates PRS into hierarchical SPICE netlists, which in turn are inputs to LVS and analog simulators (like SPICE), and are critical for physical design. The syntax-directed translation is straightforward, as rules in the PRS language dictate circuit topology: &s are series connections, |s are parallel connections, pull-ups are sourced by power supplies, pull-downs are sourced by ground, and outputs are connected to drains. The generated netlists are technology-configurable: a configuration file controls scaling and transistor dimensions (such as lambda, min-width, min-length), parasitic load, and name-mangling. `hacknet` processes every unique module in the design hierarchy once, making it highly scalable. It can also emit output in Cadence Spectre format as well as Verilog modules with fake device primitives.

V. HACKT Facts

- Homepage: http://vlsi.cornell.edu/~fang/hackt/ (last updated 2015)
- Source: http://github.com/fangism/hackt
- License: GPL2 (since 2007)
- Team: David Fang (1 primary developer, since 2004)
- Written in C++ (mostly pre-C++11 style), 200k LoC
- Build: GNU tools: autoconf, automake, libtool (`configure && make`)
- Status: Stable, last major development work in 2014
- Qualities:
  - Extensive test suite, 13k+ integration tests
  - Portable across generations of compilers and platforms (ca. 2002 through 2014)
  - Scalable to multi-billion gate-equivalent designs
- Known customers: Achronix Semiconductor (2008 to present)

VI. Summary

HACKT is ...

- A suite of EDA tools: simulators, netlist generator
- Suitable for self-timed circuits and synchronous circuits
- Suitable for academic use and chip production in industry
- Open-source and free to use

Appendix A
HAC Syntax

Primitives: `defproc` is like `module` in Verilog. `bool` is like `wire` in Verilog. There is no equivalent to `reg`, which conveys state-holding sequential logic.
Port direction: `?` is an `input`, `!` is an `output`, e.g.

```
defproc mux2(bool? in[2], select; bool! out) { }
```

Connections:

- positionally by instance's ports:

  ```
  instance_name(a, b, c);
  ```

- or with alias statements:

  ```
  a = bar.x;
  ```

Parameterization:

- Definition:

  ```
  template <pbool X; pint Y>
  defproc foo(...) {...}
  ```

- Instantiation: `foo<p, q> bar(...);`

Multi-dimensional arrays:

```
bool values[4][5][6];
```

Loop generate:

```
(;i:N: buffer[i](data[i], data[i+1]); )
```

Conditional generate:

```
[  A&&B -> corner_tile ct(...);
[] E    -> edge_tile et(...);   // else-if
[] else -> normal_tile nt(...);
]
```

User-defined data structures (also parameterizable): `defchan e1of2 <: int<1> (bool d[2]; bool e) { }`

APPENDIX B
CHP SYNTAX

chp `{ }`	enter CHP sub-language in `defproc`
`*[...]`	repeat forever (most processes)
`X?(x)`	receive on channel X into variable x (blocking)
`Y!(y)`	send value of y onto channel Y (blocking)
`a := ...;`	assign RHS expression to variable a
`[G->S` `[]else->T]`	if G, do S, else do T (deterministic selection)
`A,B`	do A and B in parallel
`A;B`	do A before B

APPENDIX C
PRS SYNTAX

prs `{ }`	enter PRS sub-language in `defproc`
expr `-> x+`	when *expr* is true, pull x up
expr `-> x-`	when *expr* is true, pull x down
expr `=> x-`	infer combinational inverse for opposite pull
expr `#> x-`	invert literals for opposite pull (for C-elements)
`p ->`	when p is true, pull
`~p ->`	when p is false, pull
`p & q`	logical AND
`p | q`	logical OR

CMOS constraint: Only guards with negated literals can pull-up (PMOS), and only non-negated literals may pull-down (NMOS).

Examples:

NAND gate:

```
~p | ~q   => z+
```

Mueller C-element (3-input, non-inverting):

```
a & b & c #> _o-
_o        => o-
```

Ratioed weak-feedback keeper (using a rule attribute):

```
[weak=1] ~o -> _o+
```

Combinational feedback keeper:

```
~o & (~a | ~b | ~c) => _o+
```

Advanced netlisting features:

Precharging internal node with en PMOS pull-up:

```
en &{~en+} x & y   -> _o-
```

Defining and using shared internal node $@zin$:

```
en & z     -> @zin-
~@zin & y1 -> _o1-
~@zin & y2 -> _o2-
```

Transistor-sizing: gate a is W units wide, L units long, high-voltage threshold variant

```
a<W,L;hvt> -> b-
```

Power supply overriding, for multi-power domain design:

```
prs<Vdd2> { rules... }
```

REFERENCES

[1] B. Cole, "Will self-timed asynchronous logic rescue CPU design?," Aug. 2002.

[2] C. G. Wong and A. J. Martin, "Data-driven process decomposition for circuit synthesis," in *8th IEEE International Conference on Electronics, Circuits and Systems*, vol. 1, pp. 539–546, Sept 2001.

[3] R. Manohar, T.-K. Lee, and A. J. Martin, "Projection: A synthesis technique for concurrent systems," in *Proceedings of the 5th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, ASYNC '99, (Washington, DC, USA), pp. 125–, IEEE Computer Society, 1999.

[4] R. Manohar and A. J. Martin, "Slack elasticity in concurrent computing," in *Proceedings of the Mathematics of Program Construction*, MPC '98, (London, UK, UK), pp. 272–285, Springer-Verlag, 1998.

[5] "IEEE Symposium on Asynchronous Circuits and Systems (ASYNC)." http://asyncsymposium.org/.

[6] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Proceedings of the Sixth MIT Conference on Advanced Research in VLSI*, AUSCRYPT '90, (Cambridge, MA, USA), pp. 263–278, MIT Press, 1990.

[7] R. Manohar and A. J. Martin, "Quasi-delay-insensitive circuits are turing-complete," in *2nd International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1995.

[8] A. J. Martin, "Compiling communicating processes into delay-insensitive VLSI circuits," tech. rep., Pasadena, CA, USA, 1986.

[9] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, Aug. 1978.

[10] A. M. Lines, "Pipelined asynchronous circuits," Master's thesis, California Institute of Technology, Pasadena, CA, USA, 1995.

[11] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design: A Systems Perspective*. Springer Publishing Company, Incorporated, 1st ed., 2010.

[12] M. E. Dean, T. E. Williams, and D. L. Dill, "Efficient self-timing with level-encoded 2-phase dual-rail (LEDR)," in *Proceedings of the University of California Santa Cruz Conference on Advanced Research in VLSI*, (Cambridge, MA, USA), pp. 55–70, MIT Press, 1991.

[13] S. Sutherland, *The Verilog PLI Handbook: A Tutorial and Reference Manual on the Verilog Programming Language Interface*. Norwell, MA: Springer, 2002.