Verilator and SystemPerl

Wilson Snyder,
wsnyder@wsnyder.org

http://www.veripool.com

June, 2004

Sun microsystems

VeriPool

Agenda

- Introduction

- Design Goals

- Benefits

- Our Tool Flow
  - Verilator: Verilog to SystemC
  - SystemPerl: Making SystemC Less Verbose

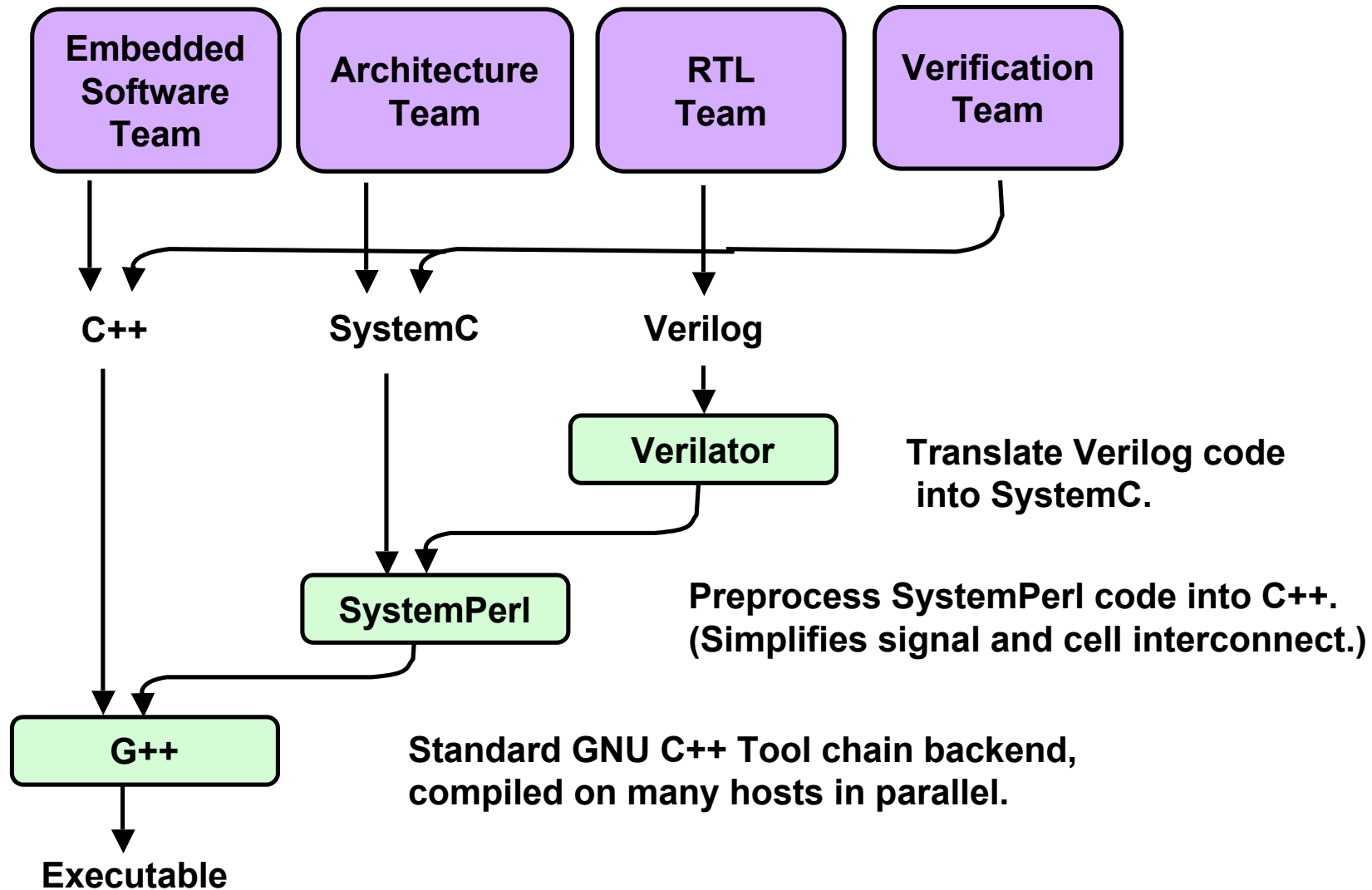- Improving SystemC Compile Times

- Obtaining the Tools

Introduction

- In 2000, we were starting a all new project and could choose all new tools
  - Wanted Verilog, for easy synthesis and related tools
  - Wanted C++, to share code with our embedded application
  - Wanted object oriented language, for test benches
  - Wanted behavioral modeling

- Needed to handle a large design
  - Four 3-6 million gate designs
  - Over 20 designers
  - Over 1.2M lines of code in 4,700 files

- What we came up with is my topic today:
  - SystemC and Verilog, together!

Benefits

- ## Faster Architectural Development
  – SystemC allows rapid behavioral model development
  – C++ allows tie-ins with embedded software

- ## Faster RTL Development
  – Verilog is standard language, and what the downstream tools want.
  – Behavioral model provides reference for RTL developers.
  – Waveforms "look" the same across RTL or SystemC, no new tools.

- ## Faster Verification Development
  – Verification tests can be developed against the fast behavioral model then run against slower RTL
  – Every chip and subchip can each be either behavioral or RTL
  – C++ hooks can be added to the Verilog
  – Automatic coverage analysis

CAD Flowchart

```
┌──────────────┐  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐
│   Embedded   │  │ Architecture │  │     RTL      │  │ Verification │
│   Software   │  │     Team     │  │     Team     │  │     Team     │
│     Team     │  │              │  │              │  │              │
└──────────────┘  └──────────────┘  └──────────────┘  └──────────────┘
```

C++   **SystemC**   **Verilog**

Verilator   Translate Verilog code
                       into SystemC.

SystemPerl   Preprocess SystemPerl code into C++.
                     (Simplifies signal and cell interconnect.)

G++   Standard GNU C++ Tool chain backend,
                     compiled on many hosts in parallel.

Executable

What Verilator Does

- Verilator converts Synthesizable Verilog into C++
  - Always statements, wires, etc
  - No time delays (a <= #{n} b;)
  - Only two state simulation (no tri-state busses)
  - Unknowns are randomized (even better then having Xs)

- Creates C++ classes for each level in the design

- Creates own interconnect and signal formats
  - Original version used sc_signals, but they are >10x slower!

- Creates a "pure" SystemC wrapper around the design
  - Hides the internal signals and sensitivity lists from the user

Example Translation

- Inputs and outputs map directly to bool, uint32_t or sc_bv's:

```verilog
module Convert;
    input clk
    input [31:0] data;
    output [31:0] out;

    always @ (posedge clk)
        out <= data;
endmodule
```

```cpp
#include "systemperl.h"
#include "verilated.h"

SC_MODULE(Convert) {
    sc_in_clk           clk;
    sc_in<uint32_t>  data;
    sc_out<uint32_t> out;

    void eval();
}

SP_CTOR_IMP(Convert) {
    SP_CELL(v,VConvert);
    SC_METHOD(eval);
    sensitive(clk);
}
...
```

Talking C++ inside Verilog

- Verilator allows C++ code to be embedded directly in Verilog

```
`systemc_include
  #include "MDebug.h"
```
Place at the top of the generated header file.

```
`systemc_header
  public:
      int debug();
```
Place inside the class definition of the generated header file.

```
`systemc_ctor
  __message = MDebug::debug();
```
Place in the constructor of the generated C++ file.

```
`systemc_implementation
  int debug() {
    return __message;
  }
```
Place in the generated C++ file.

```
`verilog
  always @ (posedge clk)
    if ($c1("debug()"))
      $write("Debug message...\n");
```
Use the C++ text "debug()" that returns a one bit value for this expression.

Verilator Optimizations

- Verilator performs many standard compiler optimizations
  - Netlist optimizations
    - wire b=~a;
    - wire c=~b;
    - wire d=c;   // Inside the simulator, it will become "d=a"
  - Constant folding
  - Module, function and task inlining
  - Levelization
  - Coverage analysis

- End result is Verilog simulation as fast as the leading Verilog-only simulators.
  - It would beat them, but the SystemC kernel is slow…

SystemPerl

- Verilator outputs a dialect of SystemC, SystemPerl.

  (Though Verilator also has option to output straight C++.)

- SystemPerl makes SystemC faster to write and execute
  - We needed only 43% as much SystemC code
  - Standardizes Pin and Cell interconnect
  - Lints interconnect
  - Automatically connects sub-modules in "shell" modules
    - So, adding a signal to low-level modules doesn't require editing the upper level modules.
  - Adds "use" statements for linking all necessary library files
  - Creates compiled tracing code (5x faster then SystemC's tracing.)

- Reducing code means faster development
  - And less debugging!

Faster SystemC Compiles

- Our model has 1,200 SystemC Modules
  - Compile time would be >> 4 hours on 2GHz system
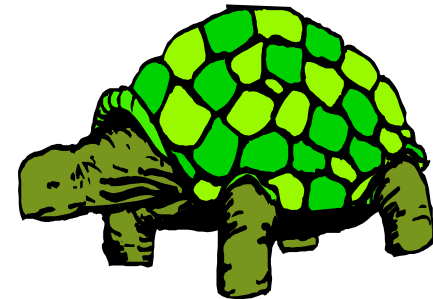
- How we fixed it [and tips you might benefit from…]
  - Cache objects so same source creates same object instantly
    - Make::Cache from my website
  - Use make -j parallel make on many machines (30x faster)
    - Schedule::Load package from my website
  - Compile multiple modules in one GCC run (10x faster)
    - a_CONCAT.cpp made by SystemPerl
      - #include "aSomething.cpp"
      - #include "aAnother.cpp"
    - Thus reduces total number of GCC runs
  - Now it's 7 minutes to compile…

Avoid Includes!

- SystemC documentation suggests the bad practice of putting SC_CTOR implementation in the header file.
  - If a low level module changes, you need to recompile EVERYTHING!

- Instead, remove all unnecessary #includes in header files!
  - Move any implementation code, such as constructors to the .cpp file
  - Declare SubModules as just "class SubModule"
  - Only #include submodules in the .cpp file

```
// FileName.h
class SubModule;
SC_MODULE(Foo) {
   …
   SubModule* subcell;
   …
   SC_CTOR(Foo);
};
```

```
// FileName.cpp
#include "SubModule.h"
SP_CTOR_IMP(Foo) {
    …
}
```

Conclusions

- ## With the SystemPerl and Verilator methodology we
  - Enable high level SystemC modeling
  - Write standard Verilog RTL
  - Can interchange Verilog <-> SystemC on major modules
  - Run as fast as major simulators.
  - Have a license-free environment.

- ## Multiple languages suit each team best
  - Faster Development, faster time to market

- ## Free runtime is good
  - $$ we would have spent on simulator runtime licenses went to computes.

Download Verilator from Veripool.com

- Downloading Verilator and SystemPerl:
  - GNU Licensed
  - C++ and Perl Based
  - http://www.veripool.com

- Also free on my site:
  - Dinotrace – Waveform Viewer w/Emacs annotation
  - Make::Cache - Object caching for faster compiles
  - Schedule::Load – Load Balancing (ala LSF)
  - Verilog-Mode - /*AUTO…*/ Expansion, Highlighting
  - Verilog-Perl – Verilog Perl preprocessor and signal renaming
  - Vpm – Assertion preprocessor
  - Vregs – Extract register and class declarations from documentation
  - Vrename – Rename signals across many files (incl SystemC files)