Unified View of Hardware and Software for Systems-on-a-Chip

In the future, we will see multi-million gate devices with greater than 80% content provided by pre-packaged hardware and software IP, in the form of cores or system-level macros (SLMs).

The VSI Alliance FAQ

CHALLENGE:

Provide methods for specifying components:

- applicable to hardware or software;
- compatible with existing languages & tools;
- with a clear semantic foundation.

Grand Long-Term Goals

- Use industry-standard syntax
  - Verilog, VHDL, C, C++, Java ...
- Develop different semantic views
  - familiar to engineers
  - compatible with standard design & verification flows
  - mutually consistent
- Provide semantically compatible tools
  - compilers, simulators, verifiers etc.
- N.B. Multiple semantic views accepted
  - event simulation,
    cycle simulation,
    synthesis

The Rest of the Talk

- A miniature case study
  - derives from recent research
  - also to be used in Spring 98 course
- Some conclusions
- Brief overview of related research projects at Cambridge

Miniature Case Study

Program Proof and Hardware Synthesis

- Derived from
  - research on Verilog synthesis semantics
  - design of courseware for undergraduates
- Based around Verilog HDL
- Combine verification of
  - program text
  - synthesised machine
- Illustrates coming together of hardware and software methods
- Illustrates one tiny fragment of grand plan!

A Familiar Example

- A program to compute the factorial:

```
begin N = Inp; FACT = 1;
      while (N > 1)
        begin
        FACT = FACT*N;
        N = N-1;
       end
end
```

- Easily verified with Hoare Logic:

$\vdash \{\texttt{Inp} = n \wedge n \geq 0\}$ (precondition)

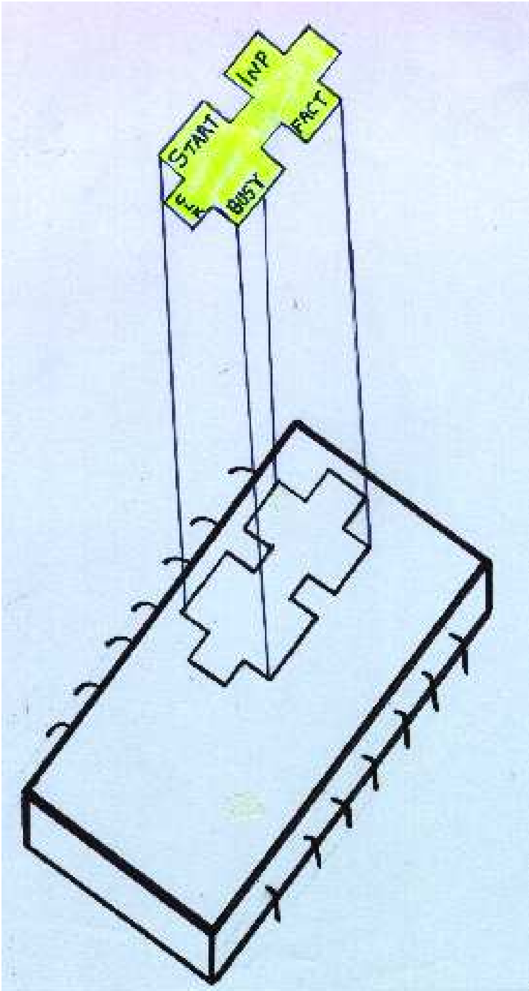
```
begin N = Inp; FACT = 1;
      while (N>1)
        begin
        FACT = FACT*N;
        N = N-1;
       end
end
```

$\{\texttt{FACT} = n!\}$ (postcondition)

System-on-a-chip Components

- Continuously running
  - `always` *<statement>*
- Calculations may spread over several cycles
  - `@(posedge clk)` *<statement>*
- Need input/output protocol, various possibilities:
  - tri-state bus
  - handshake, e.g:
    - device available when `BUSY=0`
    - to start assert `Start=1`
    - result on `FACT` when next `BUSY=0`
- Finite precision
  - $12! < 2^{32} < 13!$
  - ignored here

Factorial as a SOC Component



- `Start`, `Inp`, `clk` controlled by environment
- `FACT`, `BUSY` controlled by program

Hardware Factorial

```
always @(posedge clk)
 if (Start)
    begin N = Inp;
          FACT = 1;
          BUSY = 1;
          while (N>1)
           begin
            @(posedge clk)
            FACT = FACT*N;
            N = N-1;
           end
           BUSY = 0;
    end
```

- `Start`, `Inp` controlled by environment
- `N`, `FACT`, `BUSY` controlled by program
  - initialised to `0`
- device available when `BUSY=0`
- to start computation assert `Start=1`
- result available on `FACT` when next `BUSY=0`

Control + Data

Hardware factorial consists of a machine:

`always @(posedge clk)` $\mathcal{S}_{\texttt{FACT}}$

that repeatedly executes an embedded program:

$\mathcal{S}_{\texttt{FACT}} \stackrel{\text{def}}{=}$

```
if (Start)
   begin N = Inp;
         FACT = 1;
         BUSY = 1;
         while (N > 1)
          begin
           @(posedge clk)
           FACT = FACT * N;
           N = N-1;
          end
          BUSY = 0;
   end
```

Could place `@(posedge clk)` differently

Correctness of Hardware Factorial

Need to show:

- If `Start` is asserted when `BUSY=0` then:
  - value n of `Inp` is read during that cycle
  - eventually `BUSY` becomes `0` again
  - `FACT=`n`!` when next `BUSY=0`
- Can split these into:
  - program correctness

$$\vdash \ \{\texttt{Inp} = n \ \wedge \ n \geq 0 \ \wedge \ \texttt{Start} = 1\} \ \mathcal{S}_{\texttt{FACT}} \ \{\texttt{FACT} = n!\}$$

  - control correctness:

$$\vdash \ \texttt{BUSY} = 1 \ \text{iff control is inside} \ \mathcal{S}_{\texttt{FACT}}$$

Making Cycles Explicit

`always @(posedge clk)` $\mathcal{S}_{\texttt{FACT}}$

How do we formalise:

$\vdash$ `BUSY` $=$ `1` iff control is inside $\mathcal{S}_{\texttt{FACT}}$

- Decompose $\mathcal{S}_{\texttt{FACT}}$ into `clk`-cycles
  - i.e. convert to a state machine
- Generate state transition invariants

Alternatively: employ a richer Hoare logic:

$$\{\mathcal{P}\}\ \mathcal{S}\ \{\mathcal{Q}\}[\mathcal{I}]\langle t\rangle$$

- $\mathcal{P}$ is precondition
- $\mathcal{Q}$ is postcondition
- $\mathcal{I}$ is invariant during computation
- t is a time bound

This is STA logic (intricate – not used here)

Synthesis Semantics

- HDL semantics = translation to machine
- Definitional synthesis,
  not optimised implementation!
  - c.f. definitional interpreters for programming languages
- Doesn't reveal IP of proprietary tools
- Approach being used to define semantics of synthesisable Verilog
  - i.e. 'Synopsys subset'
  - Synopsis are helping ...
- Very 'engineer friendly'

Compiling to State Machines

- Introduce a 'program counter' `pc`
  - intialised to `0`
  - encodes control state
  - one state for each `@(posedge clk)`
- Symbolically execute
  - from each `@(posedge clk)`
  - to next `@(posedge clk)`
- Example

```
always @(posedge clk)     (State 0)
 begin
  X = Inp;
  @(posedge clk)          (State 1)
  X = X + 1;
 end
```

compiles to (in enhanced Verilog notation):

```
case (pc)
 0 : @(posedge clk)
       pc = 1 || X = Inp  (parallel assignment)
 1 : @(posedge clk)
       pc = 0 || X = X + 1
endcase
```

Another Sequencing Example

```
always @(posedge clk)      (State 0)
 begin
  X = Inp1; Y = X + Inp2;
  @(posedge clk)           (State 1)
  OUT = X + Y;
 end
```

compiles to

```
case (pc)
 0 : @(posedge clk)
         pc  = 1
     ||  X   = Inp1
     ||  Y   = Inp1 + Inp2
     ||  OUT = OUT
 1 : @(posedge clk)
         pc  = 0
     ||  X   = X
     ||  Y   = Y
     ||  OUT = X + Y
endcase
```

Conditional Example

```
always @(posedge clk)      (State 0)
 begin
  if (Choose) X = In1; else X = In2;
  @(posedge clk)           (State 1)
  OUT = X + 1;
 end
```

compiles to

```
case (pc)
 0 : @(posedge clk)
        pc  = 1
     || X   = Choose ? In1 : In2
     || OUT = OUT
 1 : @(posedge clk)
        pc  = 0
     || X   = X
     || OUT = X + 1
endcase
```

While Example: Multiplication

```
always @(posedge clk)    (State 0)
 begin
  X = Inp1; Y = Inp2; OUT=X;
  while (Y > 1)
   begin
    @(posedge clk)        (State 1)
    OUT = OUT+X; Y = Y-1;
   end
 end
```

compiles to

```
case (pc)
 0 : @(posedge clk)
       pc  = Inp2 > 1 ? 1 : 0
    || X   = Inp1
    || OUT = Inp1
    || Y   = Inp2
 1 : @(posedge clk)
       pc  = Y-1 > 1 ? 1 : 0
    || X   = X
    || OUT = OUT+X
    || Y   = Y-1
  endcase
```

Compiling the Hardware Factorial

```
always @(posedge clk)                    (State 0)
 if (Start)
    begin N = Inp; FACT = 1; BUSY = 1;
          while (N > 1)
           begin
            @(posedge clk)        (State 1)
            FACT = FACT*N;
            N = N-1;
           end
           BUSY = 0;
    end
```

compiles to

```
case (pc)
 0 : @(posedge clk)
      pc   = Start ? (Inp>1?1:0) : 0
   || FACT = Start ? 1 : FACT
   || N    = Start ? Inp : N
   || BUSY = Start ? (Inp>1?1:0) : BUSY
 1 : @(posedge clk)
      pc   = N-1 > 1 ? 1 : 0
   || FACT = FACT*N
   || N    = N-1
   || BUSY = N-1 > 1 ? BUSY : 0
endcase
```

State Transition Invariants

```
always @(posedge clk) 𝒮
```

compiles to

$$
\begin{array}{l}
\texttt{case (pc)} \\
\quad \texttt{0 : @(posedge clk)} \\
\qquad\qquad \texttt{pc = } pc_0 \\
\qquad \| \quad \overline{\mathcal{V}} = \overline{\mathcal{E}}_0 \\
\\
\quad \texttt{1 : @(posedge clk)} \\
\qquad\qquad \texttt{pc = } pc_1 \\
\qquad \| \quad \overline{\mathcal{V}} = \overline{\mathcal{E}}_1 \\
\qquad \vdots \\
\\
\texttt{endcase}
\end{array}
$$

$P(pc, \overline{\mathcal{V}})$ is invariant iff $P_{init} \wedge P_0 \wedge P_1 \wedge \cdots$

$$
\begin{array}{ll}
P_{init}: & P(0, \overline{0}) \\
P_0: & \forall \overline{\mathcal{V}}.\ P(0, \overline{\mathcal{V}}) \Rightarrow P(pc_0, \overline{\mathcal{E}}_0) \\
P_1: & \forall \overline{\mathcal{V}}.\ P(1, \overline{\mathcal{V}}) \Rightarrow P(pc_1, \overline{\mathcal{E}}_1) \\
\vdots &
\end{array}
$$

Factorial Invariants

- ```
  case (pc)
   0 : @(posedge clk)
         pc   = Start ? (Inp>1?1:0) : 0
       || FACT = Start ? 1 : FACT
       || N    = Start ? Inp : N
       || BUSY = Start ? (Inp>1?1:0) : BUSY
   1 : @(posedge clk)
         pc   = N-1 > 1 ? 1 : 0
       || FACT = FACT*N
       || N    = N-1
       || BUSY = N-1 > 1 ? BUSY : 0
  endcase
  ```

- $P(\texttt{pc}, \texttt{FACT}, \texttt{N}, \texttt{BUSY}) \stackrel{\text{def}}{=} (\texttt{pc} = 0) \Leftrightarrow (\texttt{BUSY} = 0)$

  P_{init} : $(0 = 0) \Leftrightarrow (0 = 0)$

  P_0 : $(0 = 0) \Leftrightarrow (\texttt{BUSY} = 0)$
  $\Rightarrow$
  $((\texttt{Start} ? (\texttt{Inp}>1 ? 1 : 0) : 0) = 0)$
  $\Leftrightarrow ((\texttt{Start} ? (\texttt{Inp}>1 ? 1 : 0) : \texttt{BUSY}) = 0)$

  P_1 : $(1 = 0) \Leftrightarrow (\texttt{BUSY} = 0)$
  $\Rightarrow$
  $((\texttt{N}-1>1 ? 1 : 0) = 0)$
  $\Leftrightarrow ((\texttt{N}-1>1 ? \texttt{BUSY} : 0) = 0)$
  ```

The Story So Far

Synthesisable HDL has form

`always @(posedge clk)` $\mathcal{S}$

- Algorithmic verification via ordinary program logic
  - may require human guided reasoning (e.g. guessing invariants)
- Control correctness via synthesised machine
  - easy to verify
  - often automatic

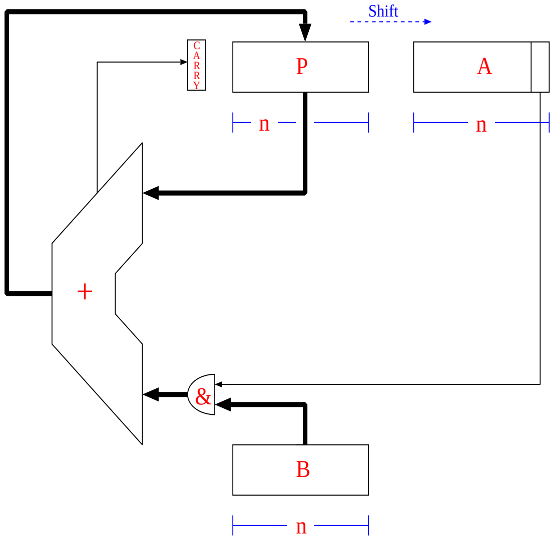
Component verification	=	program verification	+	machine analysis

Refinement

- `FACT = FACT*N` usually takes several cycles
- For example:

```
always @(posedge clk)
  begin
   X = FACT; Y = N;
   while (Y > 1)
    begin
     @(posedge clk)
     FACT = FACT+X; Y = Y-1;
    end
  end
```

- More realistically:



Refined Factorial Program

Refining `FACT = FACT*N` yields:

```
always @(posedge clk)                    (State 0)
 if (Start)
    begin N = Inp;
          FACT = 1; BUSY = 1;
          while (N>1)
           begin
            @(posedge clk)               (State 1)
            X = FACT; Y = N;
            while (Y > 1)
             begin
              @(posedge clk)             (State 2)
              FACT = FACT + X;
              Y = Y - 1;
             end
            N = N-1;
           end
           BUSY = 0;
    end
```

- Hoare Logic is compositional, so can refine proof to show factorial function is computed.
- Showing $(\mathtt{pc} = 0) \Leftrightarrow (\mathtt{BUSY} = 0)$ requires new machine to be synthesised.

Refined Factorial Machine

```
case (pc)
 0 : @(posedge clk)
       pc   = Start ? (Inp>1?1:0) : 0
    || X    = X
    || FACT = Start ? 1 : FACT
    || Y    = Y
    || N    = Start ? Inp : N
    || BUSY = Start ? (Inp>1?1:0) : BUSY
 1 : @(posedge clk)
       pc   = N>1 ? 2 : (N-1>1?1:0)
    || X    = FACT
    || FACT = FACT
    || Y    = N
    || N    = N>1 ? N : N-1
    || BUSY = N>1 ? BUSY : (N-1>1?BUSY:0)
 2 : @(posedge clk)
       pc   = Y-1>1 ? 2 : (N-1>1?1:0)
    || X    = X
    || FACT = FACT+X
    || Y    = Y-1
    || N    = Y-1>1 ? N : N-1
    || BUSY = Y-1>1 ? BUSY : (N-1>1?BUSY:0)
endcase
```

Compositional Machine Analysis

- Naive approach just generates verification conditions and solves them with brute force
  - general
  - inefficient
- More sophisticated approach uses special algorithms based on metatheorems
  - $(\texttt{pc} = \texttt{0}) \Leftrightarrow (\texttt{BUSY} = \texttt{0})$ by checking:
    - `BUSY` set to `1` on entering program
    - `BUSY` set to `0` on exiting program
- General theory of compositional properties might exist (or be discoverable) . . .

Conclusions (Case Study)

- Can apply program verification methods to real HDL (Verilog)
- Augment with state exploration methods
- Need to reconcile state-machine semantics with simulation semantics
  - good progress
    www.cl.cam.ac.uk/users/mjcg/Verilog/V/V.html
- Techniques are accessible
  - can be taught to undergraduates

Conclusions (SOC)

- Systems-on-a-chip virtual components pose challenges
  - uniform modelling for "mix and match" of hardware and software
  - detailed specifications musn't reveal IP (for both simulation or verification)
  - need new levels of abstraction for system level prototyping
  - need guaranteed consistency between different model-granularities
- Much research to be done on both concepts and engineering
- Contribution here:

  uniform synthesis-oriented treatment of software and hardware

Some Research at Cambridge (I)

- Semantics of synthesisable Verilog
  - industrial strength subset
    (help from Abhijit Ghosh of Synopsys)
  - simulation (event) semantics
  - state-machine (cycle) semantics
  - analysis of syntactic conditions for event and cycle semantics to agree
- Building various tools:
  - semantics extraction
    (guaranteed correct synthesis)
  - automatic post-design verification
    (e.g. dataflow analysis – Greaves/Larsen)
  - linking automatic and user guided formal verification

Some Research at Cambridge (II)

- Simulation core for VHDL and Verilog
  - common simulation cycle
  - rigorously specified and analysed
  - application to OMI
  - just started (Daryl Stewart's PhD)
- Hardware compilation workbench
  - based around Ian Page's Handel language (a high-level HDL for programmers)
  - implement design manipulation tools
  - compare Handel, Verilog design flows
  - postdoc: Myra VanInwegen
- Prosper
  - add formal verification to standard CAD
  - with Glasgow, Karlsruhe, IFAD, NP (EU funding almost in place)

THE END