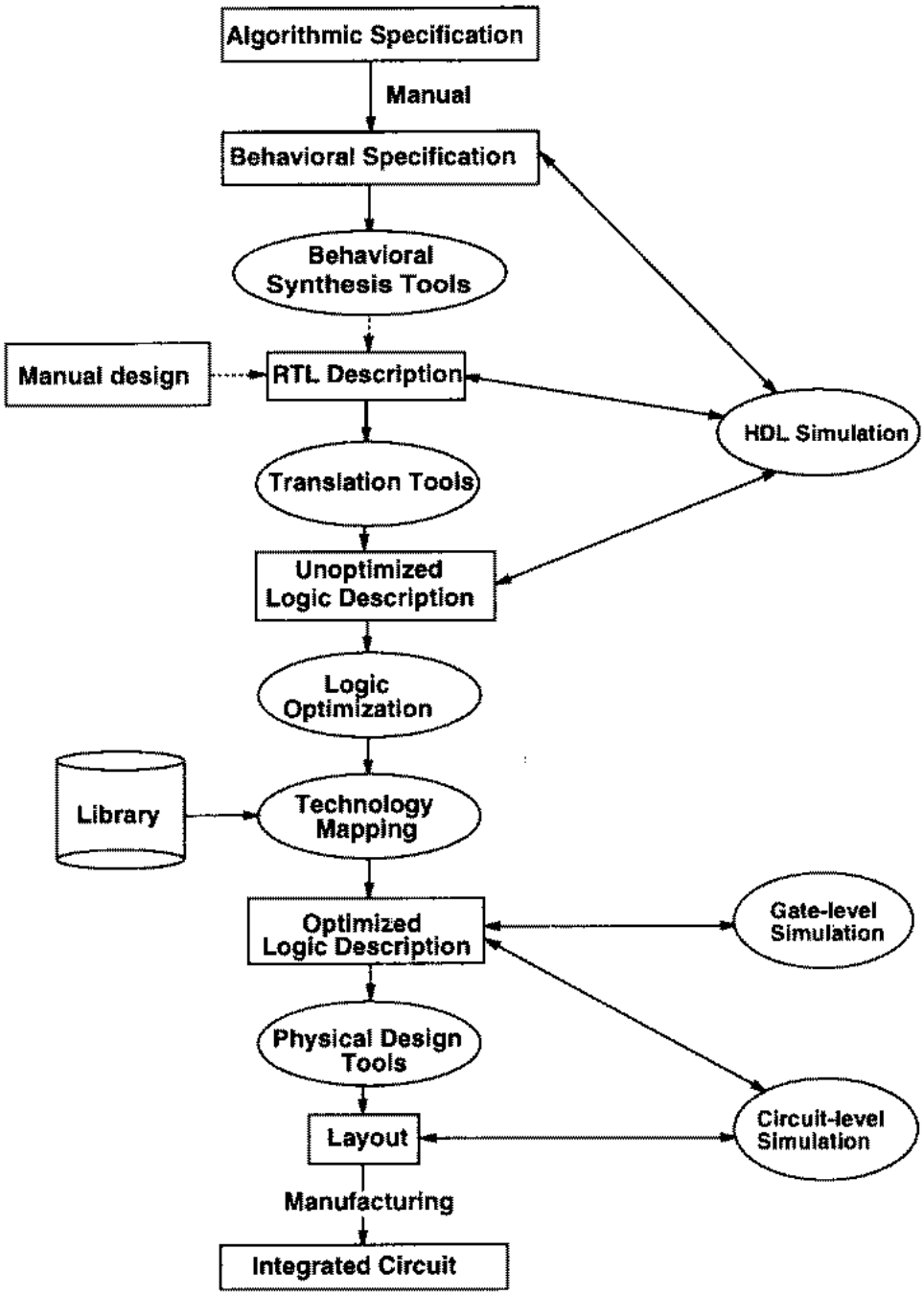
Program Verification and Hardware Synthesis

A common approach to hardware design is to write a program in a hardware description language and then compile it to a state machine using a synthesis system. Some correctness properties are naturally expressed at the programming level and established by program verification methods, but others are best specified in terms of the state transitions of the synthesised machine. I will give examples of both kinds of properties, and then discuss how they can be can be verified using a theorem-prover.

This talk is intended for a general audience.

Synthesis Design Flow

(From Kurt Keutzer's paper in FMCAD '96)



Register Transfer Level

- Currently RTL is the 'workhorse' level
  - lower levels for EE experts
  - higher levels still experimental
- Two views of RTL:
  - programming (HDL)
  - state machine (structure)
- Specification and verification needed with respect to both views
  - program verification for some properties
  - state space analysis for others

Verification

- Traditional verification uses simulation
  - event simulation
  - cycle simulation
- Formal verification uses automated proof
  - boolean equivalence checking
    - uses OBDDs etc
    - now standard
  - model checking
    - checks properties of state machines
    - currently used by Intel, TI, HP etc
  - theorem proving
    - uses powerful undecidable logics
    - long term promise
    - still a research area
- Reasons for formal verification
  - commercial: *better debugging*
  - safety critical: *save lives*
  - security critical: *ensure privacy/secrecy*

Rest of Talk

- Mainly RTL Verification
  - ideas relevant to behavioural level too
- Combine:
  - program verification
  - state machine analysis
- HDL semantics
  - based on simple hardware synthesis
- Discussion of needed theorem prover support
  - methodology, not details
- Some related current research at Cambridge

Program Specification

- Hoare triples:

  $\{precondition\}\ program\ \{postcondition\}$

- Semantics (total correctness):
  - if the *precondition* holds
  - then the *program* terminates
  - in a state in which the *postcondition* holds

- Example: a simple division program
  (`X` divided by `Y` gives quotient `Q` & remainder `R`)

```
{Y > 0}                                  (precondition)
 begin R = X; Q = 0;
       while (Y ≤ R)
        begin
         R = R - Y;
         Q = Q + 1;
        end
 end
{X = R+Y×Q ∧ R < Y}                      (postcondition)
```

Program Logic

- Much knowledge about verifying Hoare triples
  - establishing invariants
  - termination via 'variants'
  - weakest preconditions
- Standard
  - taught to undergraduates
  - textbooks
- Nice to mechanise
  - verification conditions

Hardware versus Program

- Continuously running
  - `always` <*statement*>
- Calculations may spread over several cycles
  - `@CLOCK` <*statement*>
- Need input/output protocol,
  various possibilities:
  - tri-state bus
  - handshake, e.g:
    * device available when `BUSY=0`
    * to start assert `Start=1`
    * results on `Q` and `R` when next `BUSY=0`

Division Program + I/O

```
always @CLOCK
  if (Start)
     begin
      X = In1; Y = In2;
      BUSY = 1;
      R = X; Q = 0;
      while (Y≤R)
       begin
        @CLOCK
        R = R-Y;
        Q = Q+1;
       end
      BUSY = 0;
     end
```

- `Start`, `Inp1`, `Inp2` controlled by environment
- `X`, `Y`, `Q`, `R`, `BUSY` controlled by program (initially `0`)
- device available when `BUSY=0`
- to start computation assert `Start=1`
- results on `Q` and `R` when next `BUSY=0`

Control + Data

HDL program specifies a machine:

$\mathcal{M}_{\texttt{DIV}} \stackrel{\text{def}}{=}$

```
always @CLOCK
 if (Start)
    begin
     X = In1; Y = In2;
     BUSY = 1;
     S_DIV
     BUSY = 0;
    end
```

that contains an embedded program:

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

```
R = X; Q = 0;
while (Y≤R)
 begin
  @CLOCK
  R = R-Y;
  Q = Q+1;
 end
```

Specifying Properties of Machines

- Machines determine sequences of states
  - one for each cycle
- Environment provides values for `In1`, `In2`
- Read inputs & update state every clock tick (RTL behaviour)
- Variables range over sequences of values
- Temporal operators specify properties of sequences

  $\Box$`(Y>0)`
  value of `Y` always greater than `0`

  $\Diamond$`(X = R+(Y×Q))`
  sometime `X` will equal `R+(Y×Q)`

  $\Box$`(BUSY=1` $\Rightarrow$ $\Diamond$`(BUSY=0))`
  if `BUSY=1` then sometime later `BUSY=0`

Correctness of HDL Divider

Need to show:

- If `Start` is asserted when `BUSY=0` then:
  - `Inp1` and `Inp2` read during that cycle
  - eventually `BUSY` becomes `0` again
  - `X = R+Y×Q ∧ R<Y` when next `BUSY=0`
- Can split these into:
  - program correctness (Hoare logic)

    $\{\texttt{Y > 0}\}$

    $\mathcal{S}_{\texttt{DIV}}$

    $\{\texttt{X = R+Y×Q} \wedge \texttt{R < Y}\}$
  - control correctness (temporal logic):

    `BUSY=1` $\Leftrightarrow$ `control inside` $\mathcal{S}_{\texttt{DIV}}$

    and

    $\Box$`(BUSY=1` $\Rightarrow$ $\Diamond$`(BUSY=0))`

Making Cycles Explicit

With respect to an HDL program:

```
always @CLOCK
 if (Start)
    begin
     X = In1; Y = In2;
     BUSY = 1;
     S_DIV
     BUSY = 0;
    end
```

How do we interpret Hoare triples:

$\{\texttt{Y > 0}\}\ \mathcal{S}_{\texttt{DIV}}\ \{\texttt{X = R+Y}\times\texttt{Q} \wedge \texttt{R<Y}\}$

and temporal formulas

$\texttt{BUSY=1} \Leftrightarrow \texttt{control inside}\ \mathcal{S}_{\texttt{DIV}}$

$\Box(\texttt{BUSY=1} \Rightarrow \Diamond(\texttt{BUSY=0}))$

Answer:

- convert HDL to a state machine
- then interpret w.r.t. input/state sequences

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' (Synopsys are helping)
- Engineer friendly

Compiling to State Machines

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

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

compiles to (using Verilog-like notation)

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

Another Sequencing Example

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

compiles to

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

Conditional Example

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

compiles to

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

While Example: divider

```
always @CLOCK                (State 0)
 if (Start)
    begin X = In1; Y = In2; BUSY = 1;
          R = X; Q = 0;
          while (Y ≤ R)
           @CLOCK             (State 1)
           begin
            R = R - Y; Q = Q + 1;
           end
           BUSY = 0;
    end
```

compiles to

```
 case (pc)
  0 :  pc = Start ? In2≤In1 ? 1 : 0 : 0
    ||  X = Start ? In1 : X
    ||  Y = Start ? In2 : Y
    ||  R = Start ? In1 : R
    ||  Q = Start ? 0 : Q
    ||  BUSY = Start ? In2≤In1 ? 1 : 0 : BUSY
  1 :  pc = Y≤(R-Y) ? 1 : 0
    ||  X = X
    ||  Y = Y
    ||  R = R-Y
    ||  Q = Q+1
    ||  BUSY = Y≤(R-Y) ? BUSY : 0
 endcase
```

$\mathtt{BUSY} = 1 \Leftrightarrow \mathtt{control\ inside}\ \mathcal{S}_{\mathtt{DIV}}$

can now be interpreted as

$\Box(\mathtt{BUSY} = 1 \ \Leftrightarrow \ \mathtt{pc} = 1)$

Divider + more states

```
always @CLOCK                (State 0)
 if (Start)
    begin X = In1; Y = In2;
          BUSY = 1;
          @CLOCK             (State 1)
          R = X; Q = 0;
          @CLOCK             (State 2)
          while (Y ≤ R)
           begin
            @CLOCK           (State 3)
            R = R - Y;
            @CLOCK           (State 4)
            Q = Q + 1;
           end
           BUSY = 0;
    end
```

- Allocating operations to states is behavioural synthesis
- Functional specification unchanged by additional states

Corresponding Machine

```
case (pc)
 0 :  pc = Start ? 1 : 0
   ||  X = Start ? In1 : X
   ||  Y = Start ? In2 : Y
   ||  R = R || Q = Q
   ||  BUSY = Start ? 1 : BUSY
 1 :  pc = 2
   ||  X = X || Y = Y
   ||  R = X
   ||  Q = 0
   ||  BUSY = BUSY
 2 :  pc = Y≤R ? 3 : 0
   ||  X = X || Y = Y || R = R || Q = Q
   ||  BUSY = Y≤R ? BUSY : 0
 3 :  pc = 4
   ||  X = X || Y = Y
   ||  R = R-Y
   ||  Q = Q
   ||  BUSY = BUSY
 4 :  pc = Y≤R ? 3 : 0
   ||  X = X || Y = Y || R = R
   ||  Q = Q+1
   ||  BUSY = Y≤R ? BUSY : 0
endcase
```

$\texttt{BUSY} = 1 \Leftrightarrow \texttt{control inside } \mathcal{S}_{\texttt{DIV}}$
can now be interpreted as
$\Box(\texttt{BUSY} = 1 \;\Leftrightarrow\; \texttt{pc} \in \{1, 2, 3, 4\})$

Program Proof + State Exploration

- Data processing verification via ordinary program logic
  - may require human guided reasoning (e.g. guessing invariants)
- Control correctness via state space of synthesised machine
  - often automatic (c.f. model checking)

---

design verification	=	program verification	+	machine analysis

Hoare Logic as Temporal Logic

- Hoare triples can be interpreted on machine behaviours
  - roughly

    $\{\mathcal{P}\}\ \mathcal{S}\ \{\mathcal{Q}\}$

    is:

    $\Box(\mathcal{P} \wedge \mathtt{pc}{\in}\mathcal{S} \Rightarrow \mathtt{pc}{\in}\mathcal{S}\ \mathsf{Until}\ \mathcal{Q} \wedge \neg\mathtt{pc}{\in}\mathcal{S})$

- Hoare-style reasoning principles derivable
  - for ideas (applied to real-time) see:

    M.J.C. Gordon, *A mechanized Hoare logic of state transitions*, in *A Classical Mind*, Festschrift for Professor C.A.R. Hoare edited by Roscoe, W., Prentice-Hall, 1994, pp. 143-159.
  - for RTL hardware see lecture notes on web

    `http://www.cl.cam.ac.uk/users/mjcg/`

Tower of Semantic Abstraction

Everything can be reduced to pure logic

Hoare Triple

$$\{\mathcal{P}\}\ \mathcal{S}\ \{\mathcal{Q}\}$$

Temporal Logic

$$\Box(\mathcal{P} \wedge \mathtt{pc}{\in}\mathcal{S} \Rightarrow \mathtt{pc}{\in}\mathcal{S}\ \mathsf{Until}\ \mathcal{Q} \wedge \neg\mathtt{pc}{\in}\mathcal{S})$$

Raw Logic

$$\forall t.\ \mathcal{P}(t)\ \wedge\ \mathtt{pc}(t){\in}\mathcal{S} \Rightarrow$$
$$\exists t'.\ t' \geq t\ \wedge\ \mathcal{Q}(t')\ \wedge\ \neg\mathtt{pc}(t'){\in}\mathcal{S}\ \wedge$$
$$(\forall t''.\ t \leq t''\ \wedge\ t'' < t' \Rightarrow \mathtt{pc}(t''){\in}\mathcal{S})$$

- $\mathcal{P}$ Until $\mathcal{Q}$
  means $\mathcal{Q}$ will eventually hold true
  and until it does $\mathcal{P}$ holds

Handling Multiple Descriptions

- Need to represent
  - Hoare triples
  - temporal formulas & state machines
  - verification conditions
    (could involve complex arithmetic
    or even real analysis – e.g. FP, DSP)
- Need a general purpose formalism
  - suitable for 'arbitrary mathematics'
  - mechanizable
- Several general systems exist
  - set theory
    (Isabelle/ZF, HOL-ST)
  - classical higher order logic
    (PVS, HOL, Isabelle/HOL, IMPS)
  - constructive type theory
    (Nuprl, Lego, Coq, Alf)

Pragmatic Requirements

- Combine general framework with 'best practice' specialised tools
- Many decision procedures known (hot research area: CAV etc.)
  - tautologies
  - linear arithmetic
  - temporal properties
- Partial decision procedures can often handle simple verification conditions (CADE)
  - inductive proofs (Boyer-Moore, Clam)
  - tableau methods

Existing Approaches to Integration

- Add 'trusted' external oracles to general proof system
  - OBDD and model checkers for PVS
  - arithmetic decision procedures to Isabelle
  - LTL in HOL (Karlsruhe)

  Features:
  - get state-of-the-art efficiency
  - only as sound as the oracle
  - low integration with other tools
- Efficient derived rules
  - tableau provers in Isabelle & HOL
  - linear arithmetic in HOL

  Features:
  - guaranteed sound
  - inefficient (not as bad as some say)
  - high integration with other tools

New Project (PROSPER)

- Attempt to have our cake and eat it
  - general purpose system
  - clean integration with external oracles
  - support for specific applications
  - theorem provenance tracking
- Approach
  - start by 'deconstructing' HOL98
    - theory database
    - rewriting engine
    - decision procedures and provers
    - interactive shell
  - devise protocol for external tools (maybe use XML to specify data formats)
- Experiments:
  - link to SMV model checker
  - link to NP Prover tautology checker
  - support Verilog and VHDL
- Vapourware!
  - but strong team . . . (Cambridge,Glasgow,Karlsruhe,IFAD,NP)

Grand Long-Term Goals

- Use industry-standard syntax
  - Verilog, VHDL, ...
- Develop different semantic views
  - familiar to engineers
  - compatible with standard design & verification flows
  - mutually consistent
- Provide semantically compatible tools
  - compilers, simulators, verifiers etc.

Ongoing Research at Cambridge

- Semantics of synthesisable Verilog (with Abhijit Ghosh of Synopsys)
  - industrial strength subset
  - simulation (event) semantics
  - state-machine (cycle) semantics
  - analysis of syntactic conditions for event and cycle semantics to agree
  - semantics based tools
- 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
  - implement design manipulation tools
  - compare Handel, Verilog design flows
  - postdoc: Myra VanInwegen

Conclusion

- Need diverse kinds of specifications
  - for different abstraction levels
- Need diverse verification tools
  - specialised algorithms
  - general theorem-proving
- Can embed specifications in powerful logics
  - gives unified framework
  - but hard to preserve efficiency
- Software verification methods useful for hardware
  - hardware and software theories merging

THE END