Specifying and Verification II

- **Topic of course** is the Specification and Verification of Hardware
- Assumes familiarity with Specification and Verification I (which concerns software, particularly using Hoare logic)
- The two courses are really a single course

---

Starting today

- Hardware oriented Hoare logic examples
  - apply Specification and Verification I ideas to hardware
- Modelling data
  - words as numbers or as bit arrays
- Programs as hardware
  - synthesis to state machines
- Compare program behaviour with hardware behaviour
  - intermediate states visible
- Motivate temporal logic
  - need to specify more than relationship between input and final result

---

Hoare Logic, Higher Order Logic and Temporal Logic

- Hoare logic can be used to verify programs in HDLs
- Hoare logic can be embedded in higher order logic
  - see last part of Specification and Verification I
- Higher order logic will be used to represent hardware structures
- Temporal logic (see later):
  - is used to specify properties
  - can be embedded in higher order logic
- Hoare Logic is for data reasoning, temporal logic for time (control)
- Need to choose appropriate logic – all live inside higher order logic
- Goal: software and hardware modelled in same language
  - programming languages get hardware features: ……… SystemC
  - hardware description languages get programming features: … SystemVerilog

---

Hardware Oriented Programs

- Hoare logic can be used to verify hardware algorithms
  - can reason about programs to develop hardware
  - not yet 'Industry Standard' practice
  - interesting research direction: applications to hardware/software co-design?
- Hoare logic ideas appear in some industrial methods
  - Intel's Symbolic Trajectory Evaluation (STE)
  
  \{stimulus\} <hardware model> \{response\}
  
  \[\text{Assertion Based Verification (ABV) for hardware}\]
  \[\text{annotates HDL source with assertions}\]
### Hoare Logic applied to hardware algorithms

- **Examples:** addition and multiplication
- Initially natural numbers will represent words
  - leads to messy details
  - later a type of words is introduced
- We will multiply natural numbers $a$ and $b$
  - assume they can be represented with $n$ bits

Write $ab$ to abbreviate $a \times b$

$a_i$ is $i$-th bit of the binary representation of $a$

(a, being the least significant bit)

$$a = 2^{i-1}a_{i-1} + 2^{i-2}a_{i-2} + \cdots + 2^0a_0$$

hence

$$ab = (2^{i-1}a_{i-1} + 2^{i-2}a_{i-2} + \cdots + 2^0a_0)b$$

$$= a_{i-1}2^{i-1}b + a_{i-2}2^{i-2}b + \cdots + a_02^0b$$

### Binary multiplication algorithm

- **Multiplying by $2$** corresponds to:
  - shifting one place to the left
  - adding a $0$ as the least significant bit
- **Denote** this operation by $b \rightarrow b0$

  - $2^k = b$
  - $2^k = b0$
  - $2^k + b$ (i.e. $b\downarrow$)

  - **Recall:** $ab = a_{i-1}2^{i-1}b + a_{i-2}2^{i-2}b + \cdots + a_02^0b$

  - Thus product of $a$ and $b$ is given by the sum:

    - $a_{i-1}b0$
    - $a_{i-2}b0$
    - $a_{i-3}b0$
    - $a_{i-4}b0$
    - ... $a_0b\downarrow$

    - the $i$th row is either all zeros (if $a_i$ is $0$)
    - or $b$ shifted $i$ places to the left (if $a_i$ is $1$)

  - $a_i b$ need $n$ bits $\Rightarrow$ product needs $2n$ bits

### Extracting bits and subwords

- Let $A[i]$ denote the $i$-th bit of the binary representation of $A$

  - $A[i] \in \{0, 1\}$

  - $A[0] = a$ is the least significant bit

- **Thus:**

  $$A[i] = (A \div 2^i) \mod 2$$

Define $A[n : m]$ to be the numerical value of the word comprising bits $n$ up to $m$ of $A$:

- $2^{m-n}A[m] + 2^{m-n-1}A[m-1] + \cdots + 2^0A[n] \text{ if } m > n$
- $A[n] \text{ if } m = n$
- $0 \text{ if } m < n$

Later we’ll represent words as bit-strings instead of as numbers

### Extracting bits and subwords

- Let $A[i]$ denote the $i$-th bit of the binary representation of $A$

  - $A[i] \in \{0, 1\}$

  - $A[0] = a$ is the least significant bit

- **Thus:**

  $$A[i] = (A \div 2^i) \mod 2$$

Define $A[n : m]$ to be the numerical value of the word comprising bits $n$ up to $m$ of $A$:

- $2^{m-n}A[m] + 2^{m-n-1}A[m-1] + \cdots + 2^0A[n] \text{ if } m > n$
- $A[n] \text{ if } m = n$
- $0 \text{ if } m < n$

Later we’ll represent words as bit-strings instead of as numbers

### Hoare logic verification of a multiplier

- **Add-shift multiplication program:**

  $$I := 0; \text{ PROD := 0;}
  \text{ WHILE } I < N \text{ DO}
  \text{ BEGIN PROD := PROD + A[I] \times (2^I \times B);}
  \text{ I := I + 1;}
  \text{ END}
  $$

- **Annotated Hoare specification:**

  $$(A = a \land B = b \land a < 2^N \land b < 2^M \land N > 0)
  \text{ I := 0; PROD := 0;}
  \text{ WHILE } I < N \text{ DO }
  \text{ BEGIN PROD := PROD + A[I] \times (2^I \times B);}
  \text{ I := I + 1;}
  \text{ END}
  \text{ (PROD = a \times b)}$$

- **Routine (not trivial) to verify using Hoare Logic**

  - reasoning about div and mod is horrible
Using `FOR`-commands instead of `WHILE`:

\[
\{ A = a \land B = b \land a < 2^N \land b < 2^N \land N > 0 \}
\]

\[
\text{TBEGIN PROD := 0; FOR I := 0 UNTIL N-1 DO BEGIN PROD := PROD + A[I] \times B; B := 2 \times B; END TEND}
\]

\[
\{ \text{PROD} = a \times b \}
\]

- Program corresponds directly to hardware (i.e. more like HDL)
  - three registers \( A, B \) and PROD
  - initially PROD is set to 0
  - \( A \) and \( B \) contain numbers to be multiplied

- \( I \)-th step of the multiplication:
  - adding \( A[I] \times B \) to \( \text{PROD} \)
  - then shifting \( B \) one bit to the left (i.e. multiplying it by 2)

---

Textbook multiplier:

- Simple textbook add-shift multiplier:

![Textbook multiplier diagram]

- Optimised version of naive algorithm
  - Can apply Hoare logic methods to verify correctness
    - see notes for (horrible) details of Hoare-style proof

---

Words as bit-strings (see notes for full details):

- Distinguish words from numbers – different type
  - Disadvantages: corresponds more to intuition – words have a size
- Size of a word is denoted by \( |w| \)
- \( n \)-th bit of \( w \) denoted by \( w(n) \)
- \( w[m : n] \) denotes bits \( m \) to \( n \) of \( w \)
- The word corresponding to a bit \( b \) is \( Bw(b) \)
- \( Bw(b) \) is the number represented by bit \( b \)
- \( V(w) \) is the natural number represented by word \( w \)
- \( W \) maps number \( m \) to the \( n \)-bit word representing it
- Concatenation of \( w_1 \) with \( w_2 \) denoted by \( w_1 \uplus w_2 \)
- \( w[n : n] \) denotes a word such that \( w[n] = b \) and is identical to \( w \) at all other bit positions (pad \( w \) with 0s at left if \( n \geq |w| \))
- The addition \( w_1 \uplus w_2 \) of \( w_1 \) and \( w_2 \) is defined by:
  \[
  w_1 \uplus w_2 = \left( W^{-1}(W(w_1) + W(w_2)) \right)
  \]
- \( b \cdot w \) equals \( w \) if \( b = \top \) and equals \( W|w| \) if \( b = \bot \)

---

Words vs bits:

- \( w[n : n] \) is the \( n \)-bit word consisting of \( w[n] \)
- \( w[n] \) : bool
- \( w[n : n] \) : word
- Bits and \( 1 \)-bit words are different types
- The word corresponding to a bit \( b \) is \( Bw(b) \)
- Thus: \( Bw(b)[0] = b \)
Representing Numbers

- **Natural number**: \( b_0 \cdots b_n \) represents \( 2^{n-1}b_{n-1} + 2^{n-2}b_{n-2} + \cdots + 2^0b_0 \)
- **Integer**: \( b_0 \cdots b_n \) represents
  \[-2^{n-1}b_{n-1} + 2^{n-2}b_{n-2} + \cdots + 2^0b_0 \]
  - this is the two's complement representation
- \( \mathcal{V}(w) \) is the natural number represented by \( w \)
  \[ \mathcal{V}(b_{n-1} \cdots b_0) = 2^{n-1}b_{n-1} + 2^{n-2}b_{n-2} + \cdots + 2^0b_0 \]
- Words can represent other values
  - e.g. floating point numbers, opcodes
- \( \mathcal{Bv}(b) \) is the number represented by \( b \)
  \[ \mathcal{Bv}(T) = 1 \quad \text{and} \quad \mathcal{Bv}(F) = 0 \]

Arithmetic on bits and words

- The sum of bits \( a \) and \( b \) and a carry-in bit \( c \)
  - is computed by \( a \oplus b \oplus c \) (where \( \oplus \) is 'exclusive or')
  - and the carry-out by \( (a \land b) \lor (a \land b) \)
- This is verified by:
  \[ \mathcal{Bv}(a \oplus b \oplus c) = (\mathcal{Bv}(a) + \mathcal{Bv}(b) + \mathcal{Bv}(c)) \mod 2 \]
  \[ \mathcal{Bv}((a \land b) \lor (a \land b)) = (\mathcal{Bv}(a) + \mathcal{Bv}(b) + \mathcal{Bv}(c)) \div 2 \]

Verification by enumeration

- **Sum**:

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( \mathcal{Bv}(a) + \mathcal{Bv}(b) + \mathcal{Bv}(c) )</th>
<th>( \mathcal{Bv}(a) + \mathcal{Bv}(b) + \mathcal{Bv}(c) \mod 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Carry**:

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( \mathcal{Bv}(a \land b \lor a \land b) )</th>
<th>( \mathcal{Bv}(a) + \mathcal{Bv}(b) + \mathcal{Bv}(c) )</th>
<th>( \mathcal{Bv}(a) + \mathcal{Bv}(b) + \mathcal{Bv}(c) \div 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Verification of a ripple-carry adder of any size

- Let \( R \) be:
  \[ 2^{i=1} \mathcal{Bv}(\text{CARRY}) + \mathcal{Bv}(\text{SUM}[I-1:0]) = \mathcal{V}(A[I-1:0] \lor \mathcal{B}[I-1:0]) \land \\
  \mathcal{A} = w_3 \land \mathcal{B} = w_2 \land \text{SUM} = W N 0 \land \text{CARRY} = F \land \\
  |w_3| \leq N \land |w_2| \leq B \land W > 4 \land \\
  \text{FOR } I := 0 \text{ UNTIL } N-1 \text{ DO } \{ \}
  \text{BEGIN} \\
  \text{SUM}[I] := A[I] \lor B[I] \lor \text{CARRY} ; \\
  \text{CARRY} := (A[I] \land B[I]) \lor (\text{CARRY} \land (A[I] \lor B[I])); \\
  \text{END} \\
  \{ 2^{i=1} \mathcal{Bv}(\text{CARRY}) + \mathcal{Bv}(\text{SUM}[I-1:0]) = \mathcal{V}(A[I-1:0] \lor \mathcal{B}[I-1:0]) \land \\
  \mathcal{A} = w_3 \land \mathcal{B} = w_2 \land \text{SUM} = N \land \text{CARRY} = F \land \\
  |w_3| \leq N \land |w_2| \leq B \land W > 4 \land \\
  \text{FOR } I := 0 \text{ UNTIL } N-1 \text{ DO } \{ \}
  \text{BEGIN} \\
  \text{SUM}[I] := A[I] \lor B[I] \lor \text{CARRY} ; \\
  \text{CARRY} := (A[I] \land B[I]) \lor (\text{CARRY} \land (A[I] \lor B[I])); \\
  \text{END} \\
  \}
- \( A, B \) are 8-bit words, \( \text{SUM}, \text{CARRY} \) are truthvalues, 1 is an integer
- Proof horrible (omitted)
Word multiplication program

- Simple add-shift multiplication
- Annotated correctness specification:
  \[
  \begin{align*}
  V(A) &= a \land V(B) = b \land PROD = W(2^N)0 \land \\
  |A| &\leq N \land |B| \leq N \land N > 0
  \end{align*}
  \]
  FOR I := 0 UNTIL N-1 DO
  \[
  2^IV(A[N-1 : I])b + V(PROD) = ab \land V(B) = 2^IB
  \]
  BEGIN
  PROD := PROD ⊎ A[I] • B;
  B := B ⌢ 0
  END
  \[
  V(PROD) = ab
  \]
- Can generate VCs and prove them (horrible – omitted)

Topic shift: From programs to hardware (i.e. synthesis)

- Consider a ripple-carry adder

\[
\begin{align*}
\text{FOR I := 0 UNTIL N-1 DO} & \\
\text{BEGIN} & \\
\text{SUM}[I] := A[I] \oplus B[I] \oplus CARRY; & \\
\text{CARRY} := (A[I] \land B[I]) \lor (CARRY \land (A[I] \oplus B[I])); & \\
\text{END}
\end{align*}
\]
- If a particular value of \(N\) is fixed, then the program can be unrolled into the normal circuit for an adder.
- For example take \(N = 3\) to get:

\[
\begin{align*}
\text{FOR I := 0 UNTIL 2 DO} & \\
\text{BEGIN} & \\
\text{SUM}[I] := A[I] \oplus B[I] \oplus CARRY; & \\
\text{CARRY} := (A[I] \land B[I]) \lor (CARRY \land (A[I] \oplus B[I])); & \\
\text{END}
\end{align*}
\]

N=3 adder

- 3-bit adder:

\[
\begin{align*}
\text{FOR I := 0 UNTIL 2 DO} & \\
\text{BEGIN} & \\
\text{SUM}[0] := A[0] \oplus B[0]; & \\
\text{END}
\end{align*}
\]
- Symboically executing yields logic equations:

\[
\begin{align*}
\text{SUM}[0] &= A[0] \oplus B[0]; & \\
\end{align*}
\]

Combinational logic

- Derived program is combinational logic:

\[
\begin{align*}
\text{SUM}[0] &= A[0] \oplus B[0]; & \\
\end{align*}
\]
- These are independent assignments
- Hoare Logic verifies any adder generated this way
What about non-combinational logic?

• Unrolling commands to combinational logic is sensible for the adder.

• Less so for multipliers:
  • straightforward to unroll a multiplier into combinational logic,
  • but resulting Boolean expressions will be huge,
  • evaluating in one clock cycle likely to make the cycle time too slow.

• Usually multipliers are sequential machines:
  • compute the product over a number of cycles,
  • might do the add and shift in a single cycle which would take 8 cycles,
  • might do add and shift on separate cycles, taking 2N shorter cycles.

• Decision of whether to implement a particular function as combinational or sequential logic, and if sequential, how much to do each cycle, is a decision which depends on engineering issues.

Specifying cycles

• Abstract view of multiplier:
  • computes a single multiplier change
  • from initial values of the registers
  • to final values

• Adequate for functional correctness:
  • i.e. it does multiplication

• Less abstract views needed for timing analysis

HDLs and events

• HDLs allow operations to be scheduled to clock cycles

• Statements can be prefixed by @
  • the symbol @ introduces an event control

• Multiplier that takes 8 cycles:
  \[
  \text{FOR } I := 0 \text{ UNTIL } N-1 \text{ DO}
  \begin{align*}
  \text{R} & := (R[0] . B \lor R[2N-1:1]) \times R[1:N-1] \\
  \end{align*}
  \]

• Multiplier that takes 2N cycles:
  \[
  \text{FOR } I := 0 \text{ UNTIL } N-1 \text{ DO}
  \begin{align*}
  \text{BEGIN}
  \text{SUM} & := R[0] . B \lor R[2N-1:1] \\
  \text{R} & := \text{SUM} \times R[1:N-1] \\
  \text{END}
  \end{align*}
  \]

• In Verilog, event controls can be more detailed
  • @posedge clk or @negedge clk

Need more than Hoare Logic

• Programs with added event controls can still be reasoned about using Floyd Hoare logic:
  • relation between initial and final state unchanged
  • @’s just determine intermediate states at clock ticks

• Consider this silly program:
  \[
  \text{FOR } I := 0 \text{ UNTIL } N-1 \text{ DO}
  \begin{align*}
  \text{BEGIN}
  \text{SUM} & := R[0] . B \lor R[2N-1:1] \\
  \text{R} & := \text{SUM} \times R[1:N-1] \\
  \text{END}
  \end{align*}
  \]

• Same initial-final relation, but B oscillates

• Hoare specifications only deal with initial-final relation, not intermediate states

• Temporal logic enables properties of intermediate states to be specified:
  • e.g., B stable (false for silly program above)
Division program from Specification and Verification I

- Division program:
  \[
  \begin{align*}
  &R := X; \\
  &Q := 0; \\
  &\text{WHILE } Y \leq R \text{ DO} \\
  &\quad \text{BEGIN} R := R - Y; Q := Q + 1 \text{ END}
  \end{align*}
  \]

- Implemented as a machine
  - registers \(X, Y, Q\) and \(R\)
  - on each cycle: subtract \(Y\) from \(R\); add 1 to \(Q\)

- Specification and Verification I:
  - program executes once and stops (maybe)

- Specification and Verification II:
  - program executes continuously
  - body of loop executed as combinational logic

Our toy language becomes an HDL

- To emphasize the continuously-running nature of hardware, recast division program as (where FOREVER is WHILE T DO):
  \[
  \begin{align*}
  &\text{FOREVER} \\
  &\quad \text{IF Load = 1} \\
  &\quad \quad \text{THEN } X := \text{In1}; Y := \text{In2}; \text{DONE} := 0; R := X; Q := 0 \\
  &\quad \quad \text{ELSE IF } Y \leq R \text{ THEN } R := R - Y; Q := Q + 1 \\
  &\quad \quad \text{ELSE DONE} := 1
  \end{align*}
  \]

- \(\text{In1}, \text{In2}\) and Load are inputs
  - whose value is determined by the environment (e.g., the user)

- \(X, Y, Q, R\) and \(\text{DONE}\) are registers
  - whose value is set by the program

- Environment sets the input Load to 1 to initialise registers

- To perform a division:
  - Load is set to 0
  - and held at this value until DONE = 1
  - so the environment must ensure that \(\text{DONE} = 0 \Rightarrow \text{Load} = 0\)

Programs as temporal statements

- Would like a generalised Hoare Logic specification:
  \[\{\text{If environment ensures always that: DONE} = 0 \Rightarrow \text{Load} = 0 \text{ and if Load is set to 1 when: } \text{In1} = x \land \text{In2} = y\}\]
  \[
  \begin{align*}
  &\text{FOREVER} \\
  &\quad \text{IF Load} = 1 \\
  &\quad \quad \text{THEN } X := \text{In1}; Y := \text{In2}; \text{DONE} := 0; R := X; Q := 0 \\
  &\quad \quad \text{ELSE IF } Y \leq R \text{ THEN } R := R - Y; Q := Q + 1 \\
  &\quad \quad \text{ELSE DONE} := 1
  \end{align*}
  \]

- Then \(x\) and \(y\) will be stored into \(X\) and \(Y\)
  - and on the next cycle \(\text{DONE}\) will be set to 0
  - and sometime later \(\text{DONE}\) will be be set to 1
  - and \(X\) and \(Y\) won’t change until \(\text{DONE}\) is set to 1
  - and when \(\text{DONE} \text{ goes to 1} \text{ we have: } x = R + y \times Q\)

- Stuff in red needs Temporal Logic
Brief history of temporal logic

- 1950s: philosophers invent temporal logic (A.N. Prior of Oxford)
- 1970s: Burstall, Pnueli, Lamport use temporal logic for programs
- 1980s: Emerson, Clarke and others introduce model checking
- 1980s: hardware verification examples studied
- 1990s: model checking catches on: Intel hires many logicians for P7 verification. Uses STE. Currently developing higher order logic tools (reFLect).
- 1997: Amir Pnueli gets the Turing Award in recognition of his contribution to the applications of temporal logic
- 2004: temporal notation for properties debated and standardised
  - semantics: CTL versus LTL
  - syntax: PSL and SVA 'aligned'
- 2005 onwards: Assertion Based Verification (ABV) grows
  - dynamic checking of properties by simulation (e.g., used at ARM)
  - static checking by model checking
- 2008: Clarke, Emerson & Sifakis get Turing prize for model checking
- 2008: Clarke gets 2008 CADE Herbrand Award

Note: work on formal methods leads to high prestige awards!

Rest of the course

- First look at 'raw' higher order logic for specification and verification
  - temporal logic is a notation for specifying properties of traces
  - first look at reasoning directly about traces in higher order logic
- Towards the end of the course we return to temporal logic
  - look at its constructs
  - semantics via a shallow embedding in higher order logic
  - look at the 'Industry Standard' logic PSL
  - overview some key ideas for model checking temporal logic properties
Limitations of the Method

- Formal proof can’t guarantee actual chips will work:
  - design models are not always accurate
  - there may be fabrication defects

- Specifications may not capture requirements:
  - large specifications may be unreadable
  - some input conditions may be ignored

Why Formal Specification?

Consider this device (J. Herbert’s example):

```
<table>
<thead>
<tr>
<th>datain</th>
<th>sample</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>out</td>
</tr>
</tbody>
</table>
```

This can be specified informally by

The input line `datain` accepts a stream of bits, and the output line `dataout` emits the same stream delayed by four cycles. The bus `out` is four bits wide. If the input `sample` is false then the 4-bit word at `out` is the last four bits input at `datain`. Otherwise, the output word is all zeros.

Hardware Verification Method

- Classical method of hardware verification:
  1. write a specification of intended behaviour
     `Spec`
  2. write specifications of the design components
     `Part-1, … Part-n`
  3. define a formal model of the design
     \[\vdash Design = Part-1 + \cdots + Part-n\]
  4. formulate and prove correctness
     \[\vdash Design satisfies Spec\]

- This general verification approach
  - underlies various specific formal methods
  - requires mechanized support for large designs
  - is usually applied hierarchically
Why Formal Specification?

The informal specification is

- vague: does ‘the last four bits input’ include the current bit?
- incomplete: what is the value at dataout during the first three cycles?
- unusable: a natural language specification can’t be simulated or compiled!

Specification Examples

- Simple combinational behaviour:

\[
\begin{array}{c}
i_1 \to o \\
i_2 \to \\
\end{array}
\]

\[
\Downarrow \text{Xor}(i_1, i_2, o) = (o = \neg(i_1 = i_2))
\]

- Bidirectional wires:

\[
\begin{array}{c}
g \\
s \to d \\
\end{array}
\]

\[
\Downarrow \text{Ntran}(g, s, d) = (g \Rightarrow (d = s))
\]

Specification Examples

Sequential (time-dependent) behaviour:

\[
\begin{array}{c}
d \to q \\
ck \to \\
\end{array}
\]

\[
\Downarrow \text{Dtype}(ck; d, q) = \forall t. q(t+1) = (\text{if Rise } ck t \text{ then } d t \text{ else } q t)
\]

\[
\Downarrow \text{Rise } ck t = \neg ck(t) \land ck(t+1)
\]

Formal Specification in HOL

- Consider the following device:

\[
\begin{array}{c}
a \to \text{Dev} \\
b \to d \\
\end{array}
\]

This is specified by a boolean term \( S[a, b, c, d] \) with free variables \( a, b, c, \) and \( d. \)

- The idea is that

- \( a, b, c, d \) model externally-observable values

- \( S[a, b, c, d] = \begin{cases} 
T & \text{if } a, b, c, \text{ and } d \text{ could occur simultaneously on the corresponding external wires of the device } \text{Dev} \\
F & \text{otherwise}
\end{cases} \)
Composing Behaviours

- Consider the following two devices:

  \[ \begin{align*}
  &a \quad \text{D}_1 \quad x \quad \text{D}_2 \quad b \\
  &S_1[a, x] \quad S_2[x, b]
  \end{align*} \]

- Logical conjunction (\( \land \)) models the effect of connecting components together:

  \[ \begin{align*}
  &a \quad \text{D}_1 \quad x \quad \text{D}_2 \quad b \\
  &S_1[a, x] \land S_2[x, b]
  \end{align*} \]

Hiding Internal Structure

- Consider the composite device

  \[ \begin{align*}
  &a \quad \text{D}_1 \quad x \quad \text{D}_2 \quad b \\
  &S_1[a, x] \land S_2[x, b]
  \end{align*} \]

- Existential quantification (\( \exists \)) models the effect of making wires internal to the design:

  \[ \begin{align*}
  &a \quad \text{D}_1 \quad x \quad \text{D}_2 \quad b \\
  &\exists x. S_1[a, x] \land S_2[x, b]
  \end{align*} \]

- Existential quantification is called a hiding operator—it ‘hides’ internal wires.

Specification of the Sampler

- We can specify the sampler formally by

  \[ \begin{align*}
  &\forall t: \text{time}.
  \\
  &\quad (\text{dataout}(t) = \text{datain}(t-4)) \\
  &\quad \land \\
  &\quad (\text{out}(t) = \begin{cases} \\
  \text{if } \text{sample}(t) \text{ then } F; F; F; F, \\
  \text{else } \left[ \text{datain}(t-4); \text{datain}(t-3); \text{datain}(t-2); \text{datain}(t-1) \right] \\
  \end{cases})
  \end{align*} \]

  The formal specification is
  - precise: ‘last four bits input’ doesn’t include current bit
  - complete: can infer that dataout equals datain(0) during the first three cycles.
  - usable: logic notation can be processed by machine
Hierarchical Verification

The hierarchical verification method:

Level 0

Model:
\[ \vdash M = \exists z. S_1 \land S_2 \]
Correctness:
\[ \vdash M \text{ sat } S \]

Level 1

Models:
\[ \vdash M_1 = \exists x. P_1 \land P_2 \]
\[ \vdash M_2 = \exists y. P_3 \land P_4 \]
Correctness:
\[ \vdash M_1 \text{ sat } S_1 \]
\[ \vdash M_2 \text{ sat } S_2 \]

Shallow embedding of Verilog

• Some typical structural Verilog

module COMP (p1, ... , pm);
wire w1, ..., wn;
COMP1 M1 (...);
COMP2 M2 (...);
endmodule

• Assume formulas for COMP1, COMP2 already defined

• Logical representation:

\[\text{COMP}(p_1, ..., p_m) = \exists w_1 \ldots w_n. \text{COMP1}(...) \land \text{COMP2}(...)\]

Formulating Correctness

• A key part of formal hardware verification is formalizing what ‘correctness’ means.

• The strongest formulation is equivalence:

\[ \vdash \forall v_1 \ldots v_n. M[v_1, \ldots, v_n] = S[v_1, \ldots, v_n] \]

• For partial specifications, use implication:

\[ \vdash \forall v_1 \ldots v_n. M[v_1, \ldots, v_n] \Rightarrow S[v_1, \ldots, v_n] \]

• In general, the satisfaction relationship

\[ \vdash M[v_1, \ldots, v_n] \text{ sat } S_{abs}(v_1), \ldots, abs(v_n) \]

must be one of abstraction. The specification will be an abstraction of the design model. Various kinds of abstractions on signals (abs) will be discussed later.

Hierarchical Design—Advantages

• Each type of module verified only once
  • the statement of its correctness will be reused many times

• Controls complexity through abstraction
  • each verification is done at the appropriate level of complexity
A Simple Correctness Proof

• Here is the design of a CMOS inverter:

Suppose we wish to verify that \( o = \neg i \).

There are three steps:
• define a model of the circuit in logic
• formulate the correctness of the circuit
• prove the correctness of the circuit

Design Model and Correctness

• We define the design model using composition and hiding, as follows:

\[ \vdash \text{Inv}(i, o) = \exists g \, p. \, \text{Pwr} \, p \land \text{Gnd} \, g \land \text{Ntran}(i, g, o) \land \text{Ptran}(i, p, o) \]

• Correctness is formulated by the equivalence:

\[ \vdash \forall i \, o. \, \text{Inv}(i, o) = (o = \neg i) \]

This follows by purely logical inference...

The Correctness Proof

• Definition of \( \text{Inv} \):

\[ \vdash \text{Inv}(i, o) = \exists g \, p. \, \text{Pwr} \, p \land \text{Gnd} \, g \land \text{Ntran}(i, g, o) \land \text{Ptran}(i, p, o) \]

• Expanding with definitions:

\[ \vdash \text{Inv}(i, o) = \exists g \, p. \, (p = T) \land (g = F) \land (i \Rightarrow (o = g)) \land (\neg i \Rightarrow (o = p)) \]

• By simple logical reasoning:

\[ \vdash \text{Inv}(i, o) = (i \Rightarrow (o = F)) \land (\neg i \Rightarrow (o = T)) \]

CMOS Primitives

• Formal specifications of primitives:

\[ \vdash \text{Ptran}(g, s, d) = (\neg g \Rightarrow (d = s)) \]
\[ \vdash \text{Ntran}(g, s, d) = (g \Rightarrow (d = s)) \]
\[ \vdash \text{Gnd} \, g = (g = F) \]
\[ \vdash \text{Pwr} \, p = (p = T) \]

This is the so-called switch model of CMOS.
The Correctness Proof continued

• Simplifying gives:
  \[ \vdash \text{Inv}(i,o) = (i \Rightarrow \neg o) \land (\neg i \Rightarrow o) \]
• By the law of the contrapositive:
  \[ \vdash \text{Inv}(i,o) = (o \Rightarrow \neg i) \land (\neg i \Rightarrow o) \]
• By the definition of boolean equality:
  \[ \vdash \text{Inv}(i,o) = (o = \neg i) \]
• Generalizing the free variables gives:
  \[ \forall i o. \vdash \text{Inv}(i,o) = (o = \neg i) \]

Scope of the Method

• The inverter example is, of course, trivial!
• But the same method has been applied to
  • a commercial CMOS cell library
  • several complete microprocessors (e.g. ARM)
  • floating point algorithms and hardware
• Features of the approach:
  • the specification language is just logic
    * logic can mimic HDL constructs
  • the rules of reasoning are also pure logic
    * special-purpose derived rules are possible
  • big formal proofs require machine assistance

Another Example

• An \((n+1)\)-bit ripple-carry adder:

  \[
  \begin{array}{c}
  \text{cout} \\
  \text{Add1} \\
  \vdots \\
  \text{Add1} \\
  \text{Add1} \\
  \text{Add1} \\
  \hline
  \end{array}
  \quad a_n \quad b_n \\
  \quad a_2 \quad b_2 \\
  \quad a_1 \quad b_1 \\
  \quad a_0 \quad b_0 \\
  \hline
  s_n \\
  s_2 \\
  s_1 \\
  s_0 \\
  \hline
  c_{\text{in}}
  \]

  We wish to prove that:
  \[ (2^{n+1} \times \text{cout}) + s = a + b + \text{cin} \]

  There are, as usual, three steps:
  • define a model of the circuit in logic
  • formulate the correctness of the circuit
  • prove the correctness of the circuit

Defining the Model: types

• Specification uses numbers, i.e. values of type \textit{num}
• Implementation uses words – values of type \textit{word}
  • \(n\)th bit of \(w\) denoted by \(w[n]\)
  • \(w[m:n]\) denotes bits \(m\) to \(n\) of \(w\)
  • \(\text{Bv}(b)\) is the number represented by bit \(b\)
  • \(\text{V}(w)\) is the natural number represented by word \(w\)
• Abstraction from words to numbers (data abstraction):
  \[ \vdash \text{Bv} b = \text{if} \ b \ \text{then} \ 1 \ \text{else} \ 0 \]
  \[ \vdash \text{V} w[0:0] = \text{Bv} w[0] \]
  \[ \vdash \text{V} w[n+1:0] = 2^{n+1}(\text{Bv} \ w[n+1]) + \text{V} \ w[n:0] \]
Defining the Model

• Recursive view of an \( n+1 \)-bit adder:

\[
\begin{array}{c}
\text{Add}1(\overline{a[n]}\overline{b[n]}c, \overline{a[n-1:0]}\overline{b[n-1:0]}c, a[n], b[n], \overline{cin}, \overline{s[n]}, \overline{s[n-1:0]}, \overline{cout})
\end{array}
\]

• Primitive recursive definition in logic:

\[
\begin{array}{c}
\text{AdderImp}(0)(a, b, cin, s, cout) = \text{Add1}(a[0], b[0], cin, s[0], cout) \\
\text{AdderImp}(n)(a, b, cin, s, cout) = \exists c. \text{Add1}(a[n], b[n], c, s[n], cout) \land \\
\text{AdderImp}(n-1)(a[n-1:0], b[n-1:0], cin, s[n-1:0], c)
\end{array}
\]

Formulation of Correctness

• Logical formulation of correctness:

\[
\begin{array}{c}
\text{Spec}
\end{array}
\]

\[
\forall n a b cin s cout. \\
\text{AdderImp}(n)(a, b, cin, s, cout) \Rightarrow \\
\text{Spec}(V a[n:0], V b[n:0], Bv cin, V s[n:0], Bv cout)
\]

• Note the data abstraction (\textit{abs} in an earlier slide)

• This is easy to prove (done later in the course)
Formulating Correctness

• Then correctness is stated by:

\[ \vdash \forall ck. \text{Inf}(\text{Rise } ck) \Rightarrow \forall d. \text{Dtype}(ck, d, q) \Rightarrow \text{Del}(d \text{ when } (\text{Rise } ck), q \text{ when } (\text{Rise } ck)) \]

• Note the formal validity condition:

\[ \vdash \text{Inf } P = \forall t. \exists t'. t' > t \land P(t') \]

Temporal Abstraction

• Example—abstracting to unit delay:

\[
\begin{align*}
\text{Del} & \quad i \\
\text{Del} & \quad o \\
\vdash \text{Del}(i, o) = \forall t. o(t+1) = i(t)
\end{align*}
\]

\[
\begin{align*}
\text{Dtype} & \quad d \\
\text{Dtype} & \quad q \\
\vdash \text{Dtype}(ck, d, q) = \forall t. q(t+1) = \text{if Rise } ck(t) \text{ then } d(t) \text{ else } q(t)
\end{align*}
\]

• Notions of time involved:

  • coarse grain of time—unit time = 1 clock cycle
  • fine grain of time—unit time ≈ 1 gate delay

Industry use of theorem proving

• Intel
  • floating point algorithms (uses HOL Light system)
  • hardware (uses internal tools Forte/reFL\textsuperscript{ect})

• AMD
  • floating point (uses ACL2 prover)

• Sun
  • high level architecture verification (PVS)

• Rockwell Collins
  • low level code verification (ACL2)

• Use of model checking widespread
  • discussed in latter part of the course

Formulating Correctness

• A mapping between time-scales:

\[
\begin{align*}
\text{abstract } t_a: & \quad f \\
\text{concrete } t_c: & \quad f \downarrow f \downarrow f \\
\text{clock } ck: & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ quad
Summary

- Specifying behaviour:
  - predicates—$S[a, b, c, d]$

- Specifying structure:
  - composition—$S_1[a, x] \land S_2[x, b]$
  - hiding—$\exists x. S_1[a, x] \land S_2[x, b]$

- Formulating correctness:
  - $\vdash \forall v_1 \ldots v_n. M[v_1, \ldots, v_n] = S[v_1, \ldots, v_n]$
  - $\vdash \forall v_1 \ldots v_n. M[v_1, \ldots, v_n] \Rightarrow S[v_1, \ldots, v_n]$
  - $\vdash \forall v_1 \ldots v_n. M[v_1, \ldots, v_n] \Rightarrow S[\text{abs } v_1, \ldots, \text{abs } v_n]$

- Abstraction
  - data: $w \mapsto V(w)$
  - temporal: $\text{sig} \mapsto \text{sig} \text{ when } (\text{Rise } clk)$
A 1-bit CMOS full adder

- Here is a diagram of a 1-bit full adder:

```
  +---+---+---+
  |   |   |   |
  +---+---+---+
  | Add | Ntran |
  +-----+-----+
  |     |     |
  +-----+-----+
  | Gnd | Pwr |
  +-----+-----+
```

- Lines a, b, cin, sum and cout carry the boolean values T or F.

- Specification of the adder:
  \[
  \text{Add}_1(a, b, \text{cin}, \text{sum}, \text{cout}) = (2 \times \text{Bv}(	ext{cout}) + \text{Bv}(	ext{sum}) + \text{Bv}(a) + \text{Bv}(b) + \text{Bv}(	ext{cin}))
  \]

- A correct implementation has:
  - lines a, b, cin, sum and cout
  - constrains a, b, cin, sum, and, cout as \(\text{Add}_1(a, b, \text{cin}, \text{sum}, \text{cout})\)

### Implementation

- A CMOS implementation of the adder:
  - lines with the same name are connected
  - lines p0, ..., p11 are internal
  - horizontal transistors are bidirectional

### Specification in logic

```
Add1,log(a, b, cin, sum, cout) =
  \[\text{Add}_1(a, b, \text{cin}, \text{sum}, \text{cout}) \land \text{Imp}^{11} \land \text{Gnd}^{11} \land \text{Pwr}^{11} \land \text{Ntran}^{11} \land \text{Ptran}^{11}\]
```

- Verify by Boolean algebra (tedious) or exhaustive enumeration

### An n-bit adder

- n-bit adder computes an n-bit sum and 1-bit carry-out from two n-bit inputs and a 1-bit carry-in

- Diagram:

```
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  | Add | Ntran |
  +-----+-----+-----+-----+-----+-----+-----+
  |     |     |     |     |     |     |     |
  +-----+-----+-----+-----+-----+-----+-----+
  | Gnd | Pwr |
  +-----+-----+-----+-----+-----+-----+-----+
```

- cin and cout carry single bits, i.e. Booleans

- a, b and sum carry n-bit words

- \(\text{Adder}^n\) specifies an \(n+1\)-bit adder !!!

- Example: \(\text{Adder}(3)\) specifies a 4-bit adder
Specification

- The definition of Adder is:
  \[ \text{Adder}(n)(a, b, cin, sum, cout) = (2^{n+1} \times \text{Br}(cout) + \text{V}(\text{sum} = 0 : 0)) = \text{V}(a[n : 0]) + \text{V}(b[n : 0]) + \text{V}(cin) \]

- Diagram of implementation:

- By primitive recursion:
  \[ \text{Adder}_\text{Imp}(0)(a, b, cin, sum, cout) = \text{Add}(0)(a, b, cin, sum, cout) \]
  \[ \text{Adder}_\text{Imp}(n+1)(a, b, cin, sum, cout) = \exists c. \text{Adder}_\text{Imp}(n)(a, b, cin, sum, c) \land \text{Add}(a[n+1], b[n+1], c), \text{sum}(n+1), cout) \]

Verification:

- Prove by induction on \( n \) that for all \( n \):
  \[ \text{Adder}_\text{Imp}(n)(a, b, cin, sum, cout) \]
  \[ \text{Adder}(n)(a, b, cin, sum, cout) \]

- Basic:
  \[ \text{Adder}_\text{Imp}(0)(a, b, cin, sum, cout) = \text{Add}(0)(a, b, cin, sum, cout) \]

- Expanding definitions of Adder Imp and Adder:
  \[ \text{Add}(a[0], b[0], cin, sum[0], cout) \]
  \[ (2^{n+1} \times \text{Br}(cout) + \text{V}(\text{sum} = 0 : 0)) = \text{V}(a[0 : 0]) + \text{V}(b[0 : 0]) + \text{V}(cin) \]

- Expanding definition of \( \text{Add} \) and simplifying:
  \[ (2 \times \text{Br}(cout) + \text{V}(\text{sum} = 0 : 0)) = \text{V}(a[0 : 0]) + \text{V}(b[0 : 0]) + \text{V}(cin) \]

- Follows by \( \text{V}(w[0 : 0]) = \text{Br}(w[0]) \)

Induction step

- Step:
  \[ (\text{Adder}_\text{Imp}(n)(a, b, cin, sum, cout) \Rightarrow \text{Adder}(n)(a, b, cin, sum, cout)) \Rightarrow (\text{Adder}_\text{Imp}(n+1)(a, b, cin, sum, cout) \Rightarrow \text{Adder}(n+1)(a, b, cin, sum, cout)) \]

- Assume:
  \[ (\text{Adder}_\text{Imp}(n)(a, b, cin, sum, cout) \Rightarrow \text{Adder}(n)(a, b, cin, sum, cout)) \]

- Then show:
  \[ \text{Adder}_\text{Imp}(n+1)(a, b, cin, sum, cout) \]
  \[ = \exists c. \text{Adder}_\text{Imp}(n)(a, b, cin, sum, c) \land \text{Add}(a[n+1], b[n+1], c), \text{sum}[n+1], cout) \]
  \[ = \exists c. (2^{n+1} \text{Br}(c) + \text{V}(\text{sum} = 0 : 0) = \text{V}(a[n : 0]) + \text{V}(b[n : 0]) + \text{V}(cin)) \land \text{Br}(cout) + \text{V}(\text{sum}[n+1]) = \text{Br}(a[n+1]) + \text{Br}(b[n+1]) + \text{Br}(c)) \]

Step continued

If:
\[ (A = B) \land (C = D) \]
then it follows that \( (A = 2^{n+1}C) = (B + 2^{n+1}D) \)
hence:
\[ \exists c. (2^{n+1} \text{Br}(c) + \text{V}(\text{sum}[n : 0]) = \text{V}(a[n : 0]) + \text{V}(b[n : 0]) + \text{Br}(cin)) \land \text{Br}(cout) + \text{V}(\text{sum}[n+1]) = \text{Br}(a[n+1]) + \text{Br}(b[n+1]) + \text{Br}(C) \]
\[ = \exists c. 2^{n+1} \text{Br}(c) + \text{V}(\text{sum}[n : 0]) + 2^{n+1} \text{Br}(cout) + 2^{n+1} \text{Br}(\text{sum}[n+1]) = \text{V}(a[n : 0]) + \text{V}(b[n : 0]) + \text{V}(cin) + 2^{n+1} \text{Br}(a[n+1]) + 2^{n+1} \text{Br}(b[n+1]) + 2^{n+1} \text{Br}(c) \]
\[ = \exists c. (\text{V}(\text{sum}[n+1 : 0]) + 2^{n+1} \text{Br}(cout) = \text{V}(a[n+1 : 0]) + \text{V}(b[n+1 : 0]) + \text{V}(cin)) \]
\[ = \text{Adder}(n+1)(a, b, cin, sum, cout) \]
Sequential Devices

- Pure combinational adder:
  \[ \text{Adder}(n)(a, b, cin, sum, cout) \equiv \]
  \[ (2^n\times Bv(cout) + V(sum[n:0]) = \]
  \[ V(a[n:0]) + V(b[n:0]) + Bv(cin)) \]
- \( a, b \) and \( sum \) range over words
- \( cin \) and \( cout \) range over bits (Booleans)

- Zero-delay adder:
  \[ \text{CombinationalAdder}(n)(a, b, cin, sum, cout) \equiv \]
  \[ \forall t. \text{Adder}(n)(a(t), b(t), cin(t), sum(t), cout(t)) \]
- \( a, b \) and \( sum \) range over functions from time to words
- \( cin \) and \( cout \) range over functions from time to bits

- Unit-delay adder:
  \[ \text{UnitDelayAdder}(n)(a, b, cin, sum, cout) \equiv \]
  \[ \forall t. \text{Adder}(n)(a(t), b(t), cin(t), sum(t+1), cout(t+1)) \]

Textbook add-shift multiplier

- A standard add-shift multiplier:
  
  ![Diagram of add-shift multiplier](image)

- This can be verified directly
- Verification can be done directly in HOL or using Hoare Logic
- HOL proof by induction on word size
  - essence the of proofs (the invariant) are the same
  - compare sections 1.8 and 2.7 of notes (only if you enjoy messy details)
An edge-triggered Dtype

- Register-transfer (RT) level:
  - abstract level in which devices are viewed as sequential machines
  - registers are modelled as unit-delay elements without explicit clock lines
  - used for previous multipliers

- Trace level (N.B. not standard terminology):
  - closer to HDL simulation timescale
  - clocks explicit, edges modelled
  - used for various degrees of ‘temporal granularity’

- Dtype – a fine grain trace level example


Specification of Dtype

If
• the clock \( \text{ck} \) has a rising edge at time \( t_1 \), and
• the next rising edge of \( \text{ck} \) is at \( t_2 \), and
• the value at \( d \) is stable for \( c_1 \) units of time before \( t_1 \) (\( c_1 \) is the setup time), and
• there are at least \( c_2 \) units of time between \( t_1 \) and \( t_2 \) (\( c_2 \) constrains the minimum clock period)
then
• the value at \( q \) will be stable from \( c_3 \) units of time after \( t_1 \) (\( c_3 \) is the start time) until \( c_4 \) units of time after \( t_2 \) (\( c_4 \) is the finish time), and
• the value at \( q \) between the start and finish times will equal the value held stable at \( d \) during the setup time.

Rising edges

- Page 43:
  \[ \text{Rise}_1(f)(t) = (f(t-1) = F) \land (f(t) = T) \]
- Page 65:
  \[ \text{Rise}_2(f)(t) = \neg f(t) \land f(t+1) \]
- However:
  \[ \forall f \; t > 0 \Rightarrow (\text{Rise}_1(f)(t) = \text{Rise}_2(f)(t-1)) \]
  \[ \forall f \; t \geq 0 \Rightarrow (\text{Rise}_2(f)(t) = \text{Rise}_1(f)(t+1)) \]
- In Accellera standard language PSL function \( \text{Rise}_1 \) is called \( \text{Rose} \)

Some temporal operators in Higher Order Logic

- Define:
  \[ \text{Next}(t_1,t_2)(f) = t_1 < t_2 \land f(t_2) \land \forall t_1 < t < t_2 \Rightarrow \neg f(t) \]
- Define:
  \[ \text{Stable}(t_1,t_2)(f) = \forall t_1 \leq t \land t < t_2 \Rightarrow (f(t) = f(t_1)) \]
- These are raw higher order logic not temporal logic
  - various temporal logics are described later
Dtype specification

- Logic specification:
  \[
  \text{Dtype}(c_1, c_2, c_3, c_4)(d, \overline{ck}, q) \equiv \\
  \forall t_1, t_2 \cdot \text{Rise}_e(\overline{ck})(t_1) \land \\
  \text{Next}(t_1, t_2)(\text{Rise}_e(\overline{ck})) \land \\
  (t_2 - t_1 > c_2) \land \\
  \text{Stable}(t_1 - c_1, t_1 + 1)(d) \\
  \Rightarrow \quad \text{(Stable}(t_1 + c_3, t_1 + c_4)(q) \land \ (q(t_2) = d(t_1)))
  \]

- $c_1$, $c_2$, $c_3$, and $c_4$ are timing constants
- value depends on how the device is fabricated

- Note that $\text{Next}(t_1, t_2)(\text{Rise}_e(\overline{ck}))$
  formed by applying $\text{Next}(t_1, t_2)$ to the predicate $\text{Rise}_e(\overline{ck})$

Implementation

- Can implement Dtype using NAND-gates:

egin{align*}
\text{NAND} & : p_1 \overline{p_2} = \overline{p_1} \\
\text{NAND3} & : p_1 p_2 \overline{p_3} = \overline{p_1} \overline{p_2} \\
\text{NAND} & : p_1 \overline{p_2} = \overline{p_1} \\
\text{NAND3} & : p_1 p_2 \overline{p_3} = \overline{p_1} \overline{p_2} \overline{p_3}
\end{align*}

- Unit delay model
  \[
  \text{NAND}(i_1, i_2, o) \equiv \forall t. \ o(t+1) = \neg (i_1(t) \land i_2(t))
  \]
  \[
  \text{NAND3}(i_1, i_2, i_3, o) \equiv \forall t. \ o(t+1) = \neg (i_1(t) \land i_2(t) \land i_3(t))
  \]
- Note: modelling at the fine-grain time level

Verification

- Dtype implementation in logic:
  \[
  \text{Dtype}_{\text{Imp}}(d, \overline{ck}, q) \equiv \\
  \exists p_1 \overline{p_2} p_3 \overline{p_4} p_5. \\
  \text{NAND}(p_2, d, p_1) \land \\
  \text{NAND3}(p_3, \overline{ck}, p_1, p_2) \land \\
  \text{NAND}(p_4, \overline{ck}, p_3) \land \\
  \text{NAND}(p_1, p_3, p_4) \land \\
  \text{NAND}(p_3, p_5, q) \land \\
  \text{NAND}(q, p_2, p_5)
  \]

- Correctness: find $\delta_1$, $\delta_2$, $\delta_3$, and $\delta_4$ and prove:
  \[
  \text{Dtype}_{\text{Imp}}(d, \overline{ck}, q) \Rightarrow \text{Dtype}(\delta_1, \delta_2, \delta_3, \delta_4)(d, \overline{ck}, q)
  \]

- Hard!

- Dtype is modelled at the trace level
  - fine grain time
  - explicit clock

A sequential RT level example: simple parity checker

- Input inp, an output out
- The $n$th output is $T$ \iff an even number of $T$'s input
- PARITY $f \ n$ \iff an even number of $T$'s in $f(1), \ldots, f(n)$
  \[
  \text{PARITY } f \ n \iff \text{Even number of } T \text{s in } f(1), \ldots, f(n)
  \]
  \[
  \forall n. \text{PARITY } f \ n \iff \ (f(1) = \overline{T}) \land \\
  \land \cdots \land \ (f(n) = \overline{T})
  \]
- Specification of the parity checking device:
  \[
  \forall t. \text{out}(t) = \text{PARITY } \text{inp}(t)
  \]
- Signals modelled as functions from numbers (times) to booleans
- Specification can be written as an equation between functions:
  \[
  \text{out} = \text{PARITY } \text{inp}
  \]
- Intuitively clear that specification will be satisfied if:
  \[
  \text{out}(0) = T \land \\
  \forall t. \text{out}(t+1) = \text{if inp}(t+1) \text{ then } \overline{T} \text{ else } \text{out}(t)
  \]
- Intuition can be verified by proving:
  \[
  \forall \text{inp}. \\
  \text{(out}(0) = T) \land \ (\forall t. \text{out}(t+1) = \text{if inp}(t+1) \text{ then } \overline{T} \text{ else } \text{out}(t)) \Rightarrow \\
  \forall t. \text{out}(t) = \text{PARITY } \text{inp}(t)
  \]
Notation for writing proofs & how proof assistants work

- Write formula to be proved (the goal) above a dotted line
- Write assumptions (numbered) below the line
- For example, initially we start with no assumptions

\[ \forall \text{inp out.} \]
\[ (\text{out } 0 = T) \land \\
(\forall t. \text{out}(t+1) = \text{if inp}(t+1) \text{ then } \neg \text{out}(t) \text{ else out}(t)) \Rightarrow \\
(\forall t. \text{out}(t) = \text{PARITY} \text{ inp}(t)) \]

---

First step is to consider arbitrary inp and out and then to assume the antecedents of the implication and try to prove the conclusion

\[ \forall t. \text{out}(t) = \text{PARITY} \text{ inp}(t) \]

---

Proof assistants let users perform proof steps on proof states

- The proofs here are derived from the HOL4 system, but other tools like ProofPower, Isabelle and PVS are based on related ideas
  - Details of proof state and proof steps differ
    - In HOL and ProofPower proof steps are performed via ML functions
    - Isabelle has a declarative interface, Isar, inspired by Mizar
    - In ACL2 and PVS proof steps are performed via Lisp functions

Next step: unfold definition of PARITY

- Recall definition of PARITY
  \[ \text{PARITY}(f, n) = \text{if } (f(n+1)) \text{ then } \neg \text{PARITY}(f, n) \text{ else PARITY}(f, n) \]
- Unfolding (rewriting with) the definition of PARITY in

\[ \text{out } 0 = \text{PARITY} \text{ inp } 0 \]
\[ \text{out}(t+1) = \text{PARITY} \text{ inp } (t+1) \]

---

Goal now easily proved

- Proof state from last slide

\[ \text{out } 0 = T \]
\[ \text{out}(t+1) = \text{if inp}(t+1) \text{ then } \neg \text{out}(t) \text{ else out}(t) \]

---

- Basic: goal follows from assumption 0
- Step: substitute assumption 2 into assumption 1
- Call theorem just proved UNIQUENESS LEMMA

UNIQUENESS LEMMA =
\[ \vdash \text{Vinp out.} \]
\[ (\text{out } 0 = T) \land \\
(\forall t. \text{out}(t+1) = \text{if inp}(t+1) \text{ then } \neg \text{out}(t) \text{ else out}(t)) \Rightarrow \\
\text{out}(t) = \text{PARITY} \text{ inp } t \]
Implementation

- Assume registers ‘power up’ storing F
- Thus the output at time 0 cannot be taken directly from a register
  - because the output of the parity checker at time 0 is specified to be T

Components

<table>
<thead>
<tr>
<th>Component</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONE</td>
<td>∀ t. out t = T</td>
</tr>
<tr>
<td>NOT(inp, out)</td>
<td>∀ t. out t = ¬(inp t)</td>
</tr>
<tr>
<td>MUX(sw, in1, in2, out)</td>
<td>∀ t. out t = if sw t then in1 t else in2 t</td>
</tr>
<tr>
<td>REG(inp, out)</td>
<td>∀ t. out t = if (t=0) then F else inp(t-1)</td>
</tr>
</tbody>
</table>

Implementation in HOL

<table>
<thead>
<tr>
<th>Component</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONE</td>
<td>∀ t. out t = T</td>
</tr>
<tr>
<td>NOT(inp, out)</td>
<td>∀ t. out t = ¬(inp t)</td>
</tr>
<tr>
<td>MUX(sw, in1, in2, out)</td>
<td>∀ t. out t = if sw t then in1 t else in2 t</td>
</tr>
<tr>
<td>REG(inp, out)</td>
<td>∀ t. out t = if (t=0) then F else inp(t-1)</td>
</tr>
</tbody>
</table>

Verification

- The following theorem will eventually be proved:
  |- V inp. PARITY_IMP(inp, out) ⇒ ∀ t. out t = PARITY inp t
- First prove a lemma (then theorem follows from UNIQUENESS_LEMMA)
  - The lemma (PARITY_LEMMA):
    |- V inp out. PARITY_IMP(inp, out) ⇒ (out 0 = T) ∧ ∀ t. out(t+1) = if inp(t+1) then ¬(out t) else out t
- First step: rewrite with component definitions, split conjunction
**Proof continued**

- Consider the \( t=0 \) case first

\[ \begin{align*}
\text{out} & \equiv T \\
\forall t \cdot \text{l1} t &= \neg \text{l2} t \\
\forall t \cdot \text{l3} t &= \text{if inp} t \text{ then } \text{l1} t \text{ else } \text{l2} t \\
\forall t \cdot \text{l2} t &= \text{if } t = 0 \text{ then } F \text{ else } \text{out}(t - 1) \\
\forall t \cdot \text{l4} t &= T \\
\forall t \cdot \text{l5} t &= \text{if } t = 0 \text{ then } F \text{ else } \text{l4}(t - 1) \\
\forall t \cdot \text{out} t &= \text{if } \text{l5} t \text{ then } \text{l3} t \text{ else } \text{l4} t
\end{align*} \]

- Easily follows (see stuff in blue)

- Now consider \( t+1 \) case

\[ \begin{align*}
\text{out}(t+1) &= \text{if inp}(t+1) \text{ then } \neg \text{out} t \text{ else } \text{out} t \\
\forall t \cdot \text{l1} t &= \neg \text{l2} t \\
\forall t \cdot \text{l3} t &= \text{if inp} t \text{ then } \text{l1} t \text{ else } \text{l2} t \\
\forall t \cdot \text{l2} t &= \text{if } t = 0 \text{ then } F \text{ else } \text{out}(t - 1) \\
\forall t \cdot \text{l4} t &= T \\
\forall t \cdot \text{l5} t &= \text{if } t = 0 \text{ then } F \text{ else } \text{l4}(t - 1) \\
\forall t \cdot \text{out} t &= \text{if } \text{l5} t \text{ then } \text{l3} t \text{ else } \text{l4} t
\end{align*} \]

- Goal is solved if left hand side, \( \text{out}(t+1) \), is expanded using 5

\[ \forall t \cdot \text{out} t = \text{if } \text{l5} (t+1) \text{ then } \text{l3} (t+1) \text{ else } \text{l4} (t+1) \]

- Goal follows from assumptions with a bit of calculation

**Combining lemmas**

- Call lemma just proved \text{PARITY\_LEMA}, so

\[ \text{PARITY\_LEMA} = \text{P{ARITY\_IMP}(inp,out)} \Rightarrow \]

\[ \text{PARITY\_IMP} (\text{inp,out}) \Rightarrow \]

\[ \begin{align*}
\text{PARITY\_IMP} (\text{inp,out}) &= (\forall t \cdot \text{out} t = \text{PARITY} \text{ inp} t) \\
\forall t \cdot \text{out} t &= \text{PARITY} \text{ inp} t
\end{align*} \]

- Recall

\[ \text{UNIQUENESS\_LEMA} = \text{P{ARITY\_IMP}(inp,out)} \Rightarrow \]

\[ \begin{align*}
\text{PARITY\_IMP} (\text{inp,out}) &= \forall t \cdot \text{out} t = \text{PARITY} \text{ inp} t \\
\text{UNIQUENESS\_LEMA} &= \Rightarrow \text{PARITY\_IMP} (\text{inp,out}) \Rightarrow \forall t \cdot \text{out} t = \text{PARITY} \text{ inp} t
\end{align*} \]

- Hence by transitivity of \( \Rightarrow \)

\[ \text{PARITY\_IMP} \text{ used abstract registers REG} \]

- Next: make model more concrete by using clocked Dtype

**Review**

- Specification: \( \forall t \cdot \text{out} t = \text{PARITY} \text{ inp} t \)

- Equivalent equation between functions: \( \text{out} = \text{PARITY} \text{ inp} \)

\[ \begin{align*}
\text{inp} &\rightarrow \text{out} \\
\text{PARITY} &\rightarrow \text{MUX} \\
\text{ONE} &\rightarrow \text{REG}
\end{align*} \]

\[ \begin{align*}
\text{PARITY\_IMP} (\text{inp,out}) &= \text{P{ARITY\_IMP}(inp,out)} = \\
\text{PARITY\_IMP} (\text{inp,out}) &= \exists l1 l2 l3 l4 l5. \neg l2 l1 \land \text{MUX}(\text{inp},l1,l2,l3) \land \text{REG}(\text{out},l2) \land \\
\text{ONE} &\land \text{REG}(14,15) \land \text{MUX}(15,13,14,15)
\end{align*} \]

- Verification: \( \text{PARITY\_IMP}(\text{inp,out}) \Rightarrow (\text{out} = \text{PARITY} \text{ inp}) \)
An incorrect implementation of the parity checker

\[ \neg (\text{CEQ} \text{ PARITY } f \ 0 = T) \]
\[ \land \ f \text{. PARITY } f \ (n+1) = \text{ if } f(n+1) \text{ then } \neg \text{PARITY } f \ n \text{ else PARITY } f \ n \]

The following implementation doesn’t work

\[
\begin{array}{c|c|c}
\text{inp} & \text{l1} & \text{l2} \\
\hline
\text{F} & \text{F} & \text{T} \\
\text{T} & \text{T} & \text{F} \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{inp} & \text{l1} & \text{l2} \\
\hline
\text{F} & \text{F} & \text{T} \\
\text{T} & \text{T} & \text{F} \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{inp} & \text{out} & \text{PARITY \ inp} \\
\hline
\text{F} & \text{F} & \text{T} \\
\text{T} & \text{T} & \text{F} \\
\end{array}
\]

Temporal refinement

- \text{PARITY, IMP} used abstract registers \text{REG}
- Next: make model more concrete by using clocked \text{Dtype}
- Recall the (course grained) trace level model of a \text{Dtype}:

\[
\begin{array}{c|c|c}
\text{inp} & \text{out} & \text{PARITY \ inp} \\
\hline
\text{F} & \text{F} & \text{T} \\
\text{T} & \text{T} & \text{F} \\
\end{array}
\]

Need a version of \text{Dtype} that powers up storing \text{F}

\[
\text{DtypeF}(ck,d,q) = (q \ 0 = F) \land \text{Dtype}(ck,d,q)
\]

Formulating Correctness

- A mapping between time-scales:

\[
\begin{array}{c|c|c}
\text{abstract} & \text{clock \ ck} & \text{concrete} \\
\hline
\text{F} & \text{F} & \text{T} \\
\text{T} & \text{T} & \text{F} \\
\end{array}
\]

- Define the temporal abstraction functions:

\[\forall P : P(n) = \text{ value of } s \text{ at the concrete time } t \text{ when } P \text{ true for } n \text{th time} \]

\[\text{Timed } P = \text{ the concrete time } t \text{ when } P \text{ true for } n \text{th time} \]

\[\text{Int } P = \exists t \exists t' > t \land P(t') \]

- From Melham’s Theorem:

\[\forall s. \text{Inf}(\text{Rise } ck) \vdash \text{Inf}(\text{DtypeF}(ck,d,q) \Rightarrow \text{REG}(d \text{ when } \text{Rise } ck), q \text{ when } \text{Rise } ck)) \]

\[\text{Int } P = \forall t. \exists t' > t \land P(t') \]
Digression on defining $\text{Timeof}$

- How do we define the temporal abstraction function:
  \[ \vdash \text{Timeof } P n = \text{the concrete time } t_c \text{ such that } P \text{ true for } n \text{th time} \]

- What if there is no time such that $P$ true for $n$th time
  - for example, if $P$ is never true

- Need to actually define:
  \[ \vdash \text{Timeof } P n = \text{the time } t_c \text{ such that } P \text{ true for } n \text{th time}, \text{ if such a time exists} \]

- But then what is $\text{Timeof } P n$ if no such time exists?

Hilbert's epsilon-operator to the rescue

- $\exists x.t[x]$ is an epsilon-term

- The meaning of $\exists x.t[x]$ is specified by an axiom:
  \[ \forall P. (\exists x. P x) \Rightarrow P(\exists x. P x) \]

- $\exists x.t[x]$ denotes some value, $v$ say, such that $t[v]$, if $\exists t. t[x]
- $\exists x.t[x]$ denotes some arbitrary value if $\forall t. \neg t[x]$
  - of the type of $t[x]$
  - all types are assumed non-empty

- The $\epsilon$-operator builds the Axiom of Choice into the logic

Definition of $\text{Timeof}$

- Recall the Next operator
  \[ \text{Next } t_1 t_2 \text{ sig } = t_1 \land \text{ sig } t_2 \land \forall t. \text{ t}<t \land t<t_2 \Rightarrow \neg(\text{ sig } t) \]

- Define $\text{IsTimeof } n \text{ sig } t$
  to mean “$t$ is when sig is true for the $n$-th time”
  \[(\text{IsTimeof } 0 \text{ sig } t = (\text{ sig } t \land \forall t'. t'<t \Rightarrow \neg(\text{ sig } t'))) \land \]

- Define $\text{Timeof using } \epsilon$-operator and $\text{IsTimeof}$
  $\text{Timeof } n = \epsilon t. \text{IsTimeof } n \text{ sig } t$

- $\text{IsTimeof}$ and $\text{Timeof}$ are higher-order total functions

Temporal abstraction

- Define $f@ck$ to be signal $f$ abstracted on rising edges of $ck$
  \[ \vdash f@ck = f \text{ when } (\text{Rise } ck) \]

- Recall definition of $\text{REG}$
  \[ \vdash \text{REG}(\text{inp},\text{out}) = \forall t. \text{ out } t = \text{ if } (t=0) \text{ then } F \text{ else } \text{ inp}(t-1) \]

- It follows easily that
  \[ \vdash \text{REG}(\text{inp},\text{out}) = (\text{out } 0 = F) \land \text{Del}(\text{inp},\text{out}) \]

- The properties below also follow (why?)
  \[ \vdash \text{Inf}(\text{Rise } ck) \Rightarrow \text{ DtypeF}(ck,d,q) \Rightarrow \text{REG}(d@ck, q@ck) \]
  \[ \vdash \text{MUX}(\text{switch}, \text{i1}, \text{i2}, \text{out}) \Rightarrow \}
  \[ \vdash \text{MUX}(\text{switch}@ck, \text{i1}@ck, \text{i2}@ck, \text{out}@ck) \]
  \[ \vdash \text{MUX}(\text{switch}@ck, \text{i1}@ck, \text{i2}@ck, \text{out}@ck) \]

- $\text{NOT}(\text{inp}, \text{out}) \Rightarrow \text{NOT}(\text{inp}@ck, \text{out}@ck)$

- $\text{ONE out} \Rightarrow \text{ONE}(\text{out}@ck)$

- Hint: $\vdash \forall f. (\forall x. P(x)) \Rightarrow (\forall x. P(f(x))) \quad \text{take } f = x \mapsto \text{c}\text{k} $
Cycle and trace versions

- Compare

\[
\begin{align*}
| - \text{PARITY_IMP}(\text{inp}, \text{out}) &= \exists l_1 l_2 l_3 l_4 l_5. \\
& (\neg (l_2, l_1) \land \text{MUX}(\text{inp}, l_1, l_2, l_3) \land \text{REG}(\text{out}, l_2) \\
& \land \text{ONE}(l_4) \land \text{REG}(l_4, l_5) \land \\
& \text{MUX}(l_5, l_3, l_4, \text{out}) ) \\
| - \text{DtypePARITY_IMP}(\text{ck}, \text{inp}, \text{out}) &= \exists l_1 l_2 l_3 l_4 l_5. \\
& (\neg (l_2, l_1) \land \text{MUX}(\text{inp}, l_1, l_2, l_3) \land \\
& \text{DtypeF}(\text{ck}, \text{out}, l_2) \land \text{ONE}(l_4) \land \\
& \text{DtypeF}(\text{ck}, l_4, l_5) \land \text{MUX}(l_5, l_3, l_4, \text{out}) )
\end{align*}
\]

- Hence by implications on previous slide

\[
| - \text{Inf}(\text{Rise ck}) \Rightarrow \text{DtypePARITY_IMP}(\text{ck}, \text{inp}, \text{out}) \Rightarrow \text{PARITY_IMP}(\text{inp}_{\text{ck}}, \text{out}_{\text{ck}})
\]

- This is a typical correctness result using temporal abstraction

Trace level verification

- Proved earlier

\[
| - \text{Inf}(\text{Rise \ text{ck}}) \Rightarrow \text{DtypePARITY_IMP}(\text{ck}, \text{inp}, \text{out}) \Rightarrow \text{PARITY_IMP}(\text{inp}_{\text{ck}}, \text{out}_{\text{ck}})
\]

- Specialising \text{inp} to \text{inp}_{\text{ck}} and \text{out} to \text{out}_{\text{ck}}

\[
| - \text{Inf}(\text{Rise \ text{ck}}) \Rightarrow \text{DtypePARITY_IMP}(\text{ck}, \text{inp}_{\text{ck}}, \text{out}_{\text{ck}}) \Rightarrow \text{PARITY_IMP}(\text{inp}_{\text{ck}}, \text{out}_{\text{ck}})
\]

- From previous slide

\[
| - \text{Inf}(\text{Rise \ text{ck}}) \Rightarrow \text{DtypePARITY_IMP}(\text{ck}, \text{inp}, \text{out}) \Rightarrow \text{PARITY_IMP}(\text{inp}_{\text{ck}}, \text{out}_{\text{ck}})
\]

- Hence, by transitivity of \Rightarrow

\[
| - \text{Inf}(\text{Rise \ text{ck}}) \Rightarrow \text{DtypePARITY_IMP}(\text{ck}, \text{inp}, \text{out}) \Rightarrow \text{PARITY_IMP}(\text{inp}_{\text{ck}}, \text{out}_{\text{ck}}) \Rightarrow \forall t. (\text{out}_{\text{ck}}) t = \text{PARITY} (\text{inp}_{\text{ck}}) t
\]

NEW TOPIC: modelling transistors

- Recall simple switch model of CMOS

\[
\begin{align*}
\text{Ptran}(g, s, d) &= (\neg g \Rightarrow (d = s)) \\
\text{Ntran}(g, s, d) &= (g \Rightarrow (d = s)) \\
\text{Gnd}(g) &= (g = \text{F}) \\
\text{Pwr}(p) &= (p = \text{T})
\end{align*}
\]

- This is the so-called switch model of CMOS.

The simple adder example

- This example shows non-obvious examples can be analysed

\[
\begin{align*}
\text{Add1}(a, b, \text{cin}, \text{sum}, \text{cout}) &= \exists p_0 p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 p_{10} p_{11}. \\
& (\text{Ptran}(p_1, p_0, p_2) \land \\
& \text{Ptran}(\text{cin}, p_0, p_3) \land \text{Ptran}(b, p_2, p_3) \land \\
& \text{Ptran}(a, p_2, p_4) \land \text{Ptran}(p_1, p_3, p_4) \land \text{Ntran}(a, p_4, p_5) \land \\
& \text{Ntran}(\text{cin}, p_4, p_6) \land \text{Ntran}(b, p_6, p_7) \land \text{Ntran}(a, p_7, p_8) \land \\
& \text{Ptran}(a, p_8, p_9) \land \text{Ptran}(\text{cin}, p_9, p_{10}) \land \text{Ptran}(b, p_8, p_{11}) \land \\
& \text{Ntran}(\text{cin}, p_9, p_{10}) \land \text{Ntran}(b, p_{10}, p_{11}) \land \text{Ntran}(a, p_9, p_{11}) \land \text{Ntran}(\text{cin}, p_10, p_{11}) \land \text{Ntran}(b, p_{10}, p_{11}) \land \\
& \text{Pwr}(p_0) \land \\
& \text{Ptran}(p_4, p_0, \text{sum}) \land \text{Ptran}(p_6, p_0, \text{sum}) \land \text{Gnd}(p_{11}) \land \\
& \text{Ptran}(p_{11}, p_0, \text{cout}) \land \text{Ptran}(p_{11}, p_{10}, \text{cout})
\end{align*}
\]

\[
| - \text{Add1}(a, b, \text{cin}, \text{sum}, \text{cout}) = (2 * Bv \text{cout} + \text{Bv sum} = \text{Bv a} + \text{Bv b} + \text{Bv cin})
\]
Problems with simple switch model

- Compare

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pwr</td>
<td>p1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>p2</td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gnd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Equivalent in simple switch model!

How transistors work

- Transistors conduct if there is a big enough voltage difference, $V_{TH}$
- say, between gate and source/drain
  - only conducts well if $V_g - V_a \geq V_{TH}$ or $V_a - V_g \geq V_{TH}$
  - only conducts well if $V_a - V_g \geq V_{TH}$ or $V_b - V_g \geq V_{TH}$
- If $V_a = V_g$, there is a voltage drop of about $V_{TH}$
- Example: ‘hi’ is 5v, ‘low’ is 0v

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>4v</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3v</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5v</td>
</tr>
</tbody>
</table>
```

- Weak output may not be able to switch transistors

What happens in the Simple Switch Model

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5v</td>
<td></td>
<td>4v</td>
</tr>
<tr>
<td>4v</td>
<td></td>
<td>3v</td>
</tr>
</tbody>
</table>
```

- From the definitions
  - $\forall p. \ Pwr \ p = (p = T)$
  - $\forall a \ b. \ \text{Stran} \ (g, a, b) = g \Rightarrow (a = b)$
  - $\forall \text{out}. \ \text{Bad out} = \exists l1 l2. \ \text{Pwr} \ l1 \wedge \text{Stran} \ (l1, l1, l2) \wedge \text{Stran} \ (l2, l2, \text{out})$

- It follows that
  - $\forall \text{out}. \ \text{Bad out} = \text{out}$

Consider two Xors when both inputs are $F$

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pwr</td>
<td>p1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>p2</td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gnd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Bad design has weak output
- Good design has strong output
- Need a better model to distinguish the designs
Difference switching model (Mike Fourman)

- Don’t identify boolean values and signal values
- Consider a type of values containing \( \text{Hi} \), \( \text{Lo} \) and other values

\[
\begin{align*}
Ntran(g, a, b) &= ((g = \text{Hi}) \land (a = \text{Lo}) \Rightarrow (b = \text{Lo})) \\
&\quad \land ((g = \text{Hi}) \land (b = \text{Lo}) \Rightarrow (a = \text{Lo}))
\end{align*}
\]

\[
\begin{align*}
Ptran(g, a, b) &= ((g = \text{Lo}) \land (a = \text{Hi}) \Rightarrow (b = \text{Hi})) \\
&\quad \land ((g = \text{Lo}) \land (b = \text{Hi}) \Rightarrow (a = \text{Hi}))
\end{align*}
\]

More compact definitions

\[
\begin{align*}
Ntran(g, a, b) &= (g = \text{Hi}) \Rightarrow ((a = \text{Lo}) = (b = \text{Lo})) \\
Ptran(g, a, b) &= (g = \text{Lo}) \Rightarrow ((a = \text{Hi}) = (b = \text{Hi}))
\end{align*}
\]

this is now equivalent to \( \neg (\text{out} = \text{Lo}) \)

Good and bad Xors now distinguished

Earlier examples still work

\[
\begin{align*}
|- \text{Strong } v &\Rightarrow ((v = \text{Hi}) \lor (v = \text{Lo})) \\
|- (TBv \text{ Hi} = 1) &\land (TBv \text{ Lo} = 0) \\
|- TAdd1Spec(a, b, cin, sum, cout) &\Rightarrow (2 * (TBv \text{ cout}) + TBv \text{ sum} + TBv a + TBv b + TBv cin) \\
\end{align*}
\]

Then it follows that

\[
\begin{align*}
|- \text{Strong } a \land \text{Strong } b \land \text{Strong } cin \\
&\Rightarrow \text{AddSpec}(a, b, cin, sum, cout) \Rightarrow 7 \text{AddSpec}(a, b, cin, sum, cout) \\
&\Rightarrow \text{Strong sum} \land \text{Strong cout}
\end{align*}
\]

Define

\[
\begin{align*}
|- \text{Strong } v &\Rightarrow ((v = \text{Hi}) \lor (v = \text{Lo})) \\
|- (TBv \text{ Hi} = 1) &\land (TBv \text{ Lo} = 0) \\
|- TAdd1Spec(a, b, cin, sum, cout) &\Rightarrow (2 * (TBv \text{ cout}) + TBv \text{ sum} + TBv a + TBv b + TBv cin) \\
\end{align*}
\]
Switch models only allow us to deduce

\[(\text{ph1} = \text{Hi}) \land (\text{ph2} = \text{Hi}) \Rightarrow ((\text{in} = \text{Hi}) \Rightarrow (\text{out} = \text{Hi})) \land ((\text{in} = \text{Lo}) \Rightarrow (\text{out} = \text{Lo}))\]

Actual behaviour is a shift register

- For simplicity, threshold effects ignored in what follows.
Phase 4: \( \text{ph1=} \text{Lo} \) and \( \text{ph2=} \text{Hi} \)

\[
\begin{align*}
\text{Pwr} & \quad \text{Gnd} \\
\text{Lo} & \quad \text{Hi} \\
\sim w & \quad \sim w
\end{align*}
\]

\[
(\text{ph1}(t+2) = \text{Hi}) \land (\text{ph2}(t+2) = \text{Lo}) \land \\
(\text{ph1}(t+3) = \text{Lo}) \land (\text{ph2}(t+3) = \text{Hi}) \\
\Rightarrow (\text{out}(t+3) = \text{in}(t+2))
\]

Characterisation of behaviour

\[
\begin{align*}
\text{Pwr} & \quad \text{Gnd} \\
\text{Pwr} & \quad \text{Gnd} \\
\text{ph1} & \quad \text{ph2} \\
\text{in} & \quad \text{out}
\end{align*}
\]

\[
(\text{ph1}(t) = \text{Hi}) \land (\text{ph2}(t) = \text{Lo}) \land \\
(\text{ph1}(t+1) = \text{Lo}) \land (\text{ph2}(t+1) = \text{Hi}) \\
\Rightarrow (\text{out}(t+1) = \text{in}(t))
\]

\[
\begin{align*}
(\text{ph1}(t) = \text{Hi}) & \land (\text{ph2}(t) = \text{Lo}) \land \\
(\text{ph1}(t+1) = \text{Lo}) & \land (\text{ph2}(t+1) = \text{Hi}) \\
(\text{ph1}(t+2) = \text{Hi}) & \land (\text{ph2}(t+2) = \text{Lo}) \\
\Rightarrow (\text{out}(t+2) = \text{in}(t))
\end{align*}
\]

\( \text{out}(t+3) \) value follows by \( t \rightarrow t+2 \) in first property

Unidirectional sequential model

- Four values: \( \text{Hi}, \text{Lo}, \text{Fl} \) (‘floating’), \( X \) (unknown/error)
- \( \neg(\text{Hi} = \text{Lo}) \land \neg(\text{Lo} = \text{Hi}) \land \\
\neg(\text{Hi} = \text{Fl}) \land \neg(\text{Fl} = \text{Hi}) \\
\neg(\text{Lo} = \text{Fl}) \land \neg(\text{Fl} = \text{Lo}) \\
\]

- Strong \( v = ((v = \text{Hi}) \lor (v = \text{Lo})) \)
- \( \neg \) Float \( v = (v = \text{Fl}) \)
- Join operator: \( U \)
- \( \neg v1 \land \neg v2 = \text{if Strong v1} \land \text{Float v2} \\
\text{then v1 else if Float v1} \land \text{Strong v2} \\
\text{then v2 else if Float v1} \land \text{Float v2} \\
\text{then Fl else X} \\
\neg \text{Join}(i1,i2,out) = \forall t. \text{out t} = (i1 t) U (i2 t) \]

Signals are functions of time

- Pwr out
- Gnd out
- Cap(i,out)

\[ \neg \text{Join}(i1,i2,out) = \forall t. \text{out t} = (i1 t) U (i2 t) \]

\[ \neg \text{Pwr out} = \forall t. \text{out t} = \text{Hi} \]

\[ \neg \text{Gnd out} = \forall t. \text{out t} = \text{Lo} \]

\[ \neg \text{Cap}(i,out) = \forall t. \text{out t} = \text{if Strong(i t) then i t else if Float(i t) \land Strong(i(t-1)) then i(t-1) else Fl} \]
Unidirectional sequential transistor models

\[ N_{\text{switch}}(g,i,out) = \forall t. \text{out}\ t = \begin{cases} i\ t & \text{if} \ g\ t = \text{Hi} \\ \text{Fl} & \text{if} \ (g\ t = \text{Lo}) \lor (i\ t = \text{Fl}) \\ \text{X} & \text{else} \end{cases} \]

\[ P_{\text{switch}}(g,i,out) = \forall t. \text{out}\ t = \begin{cases} i\ t & \text{if} \ g\ t = \text{Lo} \\ \text{Fl} & \text{if} \ (g\ t = \text{Hi}) \lor (i\ t = \text{Fl}) \\ \text{X} & \text{else} \end{cases} \]

Sequential shift register model

\[ \text{ShiftReg}(i,out,ph1,ph2) = \exists l1 \ l2 \ l3 \ l4 \ l5 \ l6 \ l7 \ l8 \ l9 \ l10 \ l11 \ l12 \ l13. \]
\[ N_{\text{switch}}(ph1,i,l1) \land \text{Cap}(l1,l2) \land P_{\text{w}}\ l3 \land P_{\text{switch}}(l2,l3,l4) \land N_{\text{switch}}(l2,l6,l5) \land G_{\text{nd}}\ l6 \land \text{Join}(l4,l5,l7) \land N_{\text{switch}}(ph2,l7,l8) \land \text{Cap}(l8,l9) \land P_{\text{w}}\ l10 \land P_{\text{switch}}(l9,l10,l11) \land N_{\text{switch}}(l9,l13,l12) \land G_{\text{nd}}\ l13 \land \text{Join}(l11,l12,out) \]

- Lots more state variables than in combinational switch model!

Correctness of sequential shift register model

\[ \text{ShiftReg}(in,out,ph1,ph2) \land \text{Strong}(in\ t) \land (ph1\ t = \text{Hi}) \land (ph2\ t = \text{Lo}) \land (ph1(t+1) = \text{Lo}) \land (ph2(t+1) = \text{Hi}) \land (ph1(t+2) = \text{Hi}) \land (ph2(t+2) = \text{Lo}) \Rightarrow \text{out}(t+2) = in\ t \]

A model of NMOS

- Need a new component: pullup

\[ \text{Pu}(i,out) = \forall t. \text{out}\ t = \begin{cases} \text{Hi} & \text{if} \ \text{Float}(i\ t) \text{ then } \text{Hi} \text{ else } i\ t \\ \text{Hi} & \text{if} \ i \text{ is strong then } \text{out} = i \\ \text{Hi} & \text{if} \ i \text{ is floating then } \text{out} = \text{Hi} \end{cases} \]
NMOS inverter

\[ \text{Inv}(i, out) = \exists l_1 l_2 l_3. \text{Cap}(i, l_1) \land \text{Gnd}(l_2) \land \text{Nswitch}(l_1, l_2, l_3) \land \text{Pu}(l_3, out) \]

\[ \text{Inv}(i, out) \Rightarrow ((i(t) = \text{Hi}) \Rightarrow (out(t+1) = \text{Lo})) \land ((i(t) = \text{Lo}) \Rightarrow (out(t+1) = \text{Hi})) \land ((i(t+1) = \text{Fl}) \Rightarrow (((i(t) = \text{Hi}) \Rightarrow (out(t+2) = \text{Lo})) \land ((i(t) = \text{Lo}) \Rightarrow (out(t+2) = \text{Hi})))) \]

Four phase NMOS shift register

\[ \text{FourPhaseShiftReg}(i, out, \text{ph1}, \text{ph2}, \text{ph3}, \text{ph4}) \]

\[ \text{inv}(i(t+1)) \land (\text{ph1}(t+1) = \text{Hi}) \land (\text{ph2}(t+1) = \text{Lo}) \land (\text{ph3}(t+1) = \text{Lo}) \land (\text{ph4}(t+1) = \text{Lo}) \land (\text{ph1}(t+2) = \text{Lo}) \land (\text{ph2}(t+2) = \text{Lo}) \land (\text{ph3}(t+2) = \text{Hi}) \land (\text{ph4}(t+2) = \text{Lo}) \land (\text{ph1}(t+3) = \text{Lo}) \land (\text{ph2}(t+3) = \text{Lo}) \land (\text{ph3}(t+3) = \text{Lo}) \land (\text{ph4}(t+3) = \text{Hi}) \Rightarrow (out(t+3) = i(t+1)) \]

Phase 1 (precharge internal node)

Colour scheme: Hi, Lo, Fl; threshold effects ignored

Phase 2 (input Lo, retain precharge)

Colour scheme: Hi, Lo, Fl and dotted means precharge
Phase 1 (precharge internal node)

\[(\phi_1(t) = \text{Hi}) \land (\phi_2(t) = \text{Lo}) \land (\phi_3(t) = \text{Lo}) \land (\phi_4(t) = \text{Lo}) \]

- out retains previous value

Phase 2 (input \text{Hi}, kill precharge)

\[(\phi_1(t+1) = \text{Lo}) \land (\phi_2(t+1) = \text{Hi}) \land (\phi_3(t+1) = \text{Lo}) \land (\phi_4(t+1) = \text{Lo}) \]

\[(i(t+1) = \text{Hi})

Phase 3 (precharge out, internal node retains value)

\[(\phi_1(t+2) = \text{Lo}) \land (\phi_2(t+2) = \text{Lo}) \land (\phi_3(t+2) = \text{Hi}) \land (\phi_4(t+2) = \text{Lo}) \]

\[(\text{out}(t+2) = \text{Hi})

Phase 4 (kill precharge)

\[(\phi_1(t+3) = \text{Lo}) \land (\phi_2(t+3) = \text{Lo}) \land (\phi_3(t+3) = \text{Lo}) \land (\phi_4(t+3) = \text{Hi}) \]

\[(\text{out}(t+3) = \text{Lo})

out retains previous value
Phase 3 (precharge out, internal node retains value)

\[
(\phi(t+2) = \text{Lo}) \land (\phi(t+2) = \text{Lo}) \land (\phi(t+2) = \text{Hi}) \land (\phi(t+2) = \text{Lo})
\]

Phase 4 (out retains precharge)

\[
(\phi(t+3) = \text{Lo}) \land (\phi(t+3) = \text{Lo}) \land (\phi(t+3) = \text{Lo}) \land (\phi(t+3) = \text{Hi})
\]
\[\text{out}(t+3) = \text{Hi}\]

Conclusions

- Simple switch model good for sanity checking
  - won’t catch threshold errors
  - purely combinational
- Threshold switch model catches threshold errors
  - proofs a bit harder (not much)
- Sequential models of dubious electrical validity
  - but they can sanity check functional correctness of designs
  - can handle subtle circuits

\[
\neg \text{FourPhaseShiftReg}(\text{in}, \text{out}, \phi1, \phi2, \phi3, \phi4)
\]
\[\land \neg \text{Strong}(\text{in}(t+1))
\]
\[\land (\phi1(t) = \text{Lo}) \land (\phi2(t) = \text{Hi}) \land (\phi3(t) = \text{Lo}) \land (\phi4(t) = \text{Lo})
\]
\[\land (\phi1(t+1) = \text{Lo}) \land (\phi2(t+1) = \text{Hi}) \land (\phi3(t+1) = \text{Lo}) \land (\phi4(t+1) = \text{Lo})
\]
\[\land (\phi1(t+2) = \text{Lo}) \land (\phi2(t+2) = \text{Lo}) \land (\phi3(t+2) = \text{Hi}) \land (\phi4(t+2) = \text{Lo})
\]
\[\land (\phi1(t+3) = \text{Lo}) \land (\phi2(t+3) = \text{Lo}) \land (\phi3(t+3) = \text{Lo}) \land (\phi4(t+3) = \text{Hi})
\]
\[\Rightarrow \neg \text{out}(t+3) = \text{in}(t+1)
\]
Would like a generalised Hoare Logic specification:

\[ \Rightarrow \{ \text{If environment ensures always that: } \text{DONE}=0 \Rightarrow \text{Load}=0 \text{ and if Load is set to 1 when: } \text{In1}=x \land \text{In2}=y \} \]

**FOREVER**

IF Load=1

THEN X := In1; Y := In2; DONE := 0; R := x; Q := 0

ELSE IF Y \leq R THEN R := R - Y; Q := Q + 1

ELSE DONE := 1

{ Then x and y will be stored into X and Y and on the next cycle DONE will be set to 0 and sometime later DONE will be set to 1 and X and Y won’t change until DONE is set to 1 and when DONE goes to 1 we have: } x = R + y \times Q

Stuff in red needs **Temporal Logic**.

### DONE SO FAR:

- Higher-order logic used directly for specification and verification
  - various abstraction levels from transistors to high-level behaviour

### COMING NEXT:

- Temporal logic
  - various constructs and time models: CTL, LTL
  - the ‘Industry Standard’ logic PSL
  - semantics via a shallow embedding in higher order logic
  - overview key ideas for model checking temporal logic properties
- Simulation (Verilog, VHDL) compared with formal verification

### Aside: finding bugs versus providing assurance

<table>
<thead>
<tr>
<th>Formal verification based debugging</th>
<th>Proof of correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>proof failure (\Rightarrow) bugs</td>
<td>proof success (\Rightarrow) assurance</td>
</tr>
<tr>
<td>practical for real code</td>
<td>expensive and often impractical</td>
</tr>
<tr>
<td>unsound models OK</td>
<td>make high fidelity models</td>
</tr>
<tr>
<td>unsafe implementation methods OK</td>
<td>important to use trustworthy tools</td>
</tr>
</tbody>
</table>

- A bug is a bug no matter how found!
- Assurance mainly supported by certification agencies
  - safety and security critical systems
- Companies (Intel, AMD, MS) mostly use FV for debugging

### A current research goal:

adapt bug-finding verification methods for correctness assurance

- validate models used for debugging
- deductive (hence sound) implementations of known verification methods

### NEW TOPIC: Model Checking

- Models as state transition systems
- Reachability properties
- Counterexamples (used for debugging)
- Binary Decision Diagrams – BDDs
- Symbolic reachability checking
- A general property language: CTL
- Semantics in HOL (shallow embedding)
- Examples of CTL properties
- Overview of model checking (explicit state and symbolic)
- Linear Temporal Logic (LTL)
- Expressibility, CTL*
- Interval Temporal Logic (ITL)
- Accellera Property Specification Language (Sugar/PSL)
Models are expressed as State Transition Systems

- Set of states: type states
- Set of initial states: predicate $B$
  - $B : \text{states} \to \text{bool}$
  - $B \ s$ means $s$ is an initial state
- State transition relation: $R$
  - $R : \text{states} \times \text{states} \to \text{bool}$
  - $R(s, s')$ means $s'$ a successor to $s$

- $\mathcal{R}$ defines a branching time model

Example: single state machine

- State transition function: $\delta$
  - $\delta : \text{states} \times \text{inputs} \to \text{states}$
- Define state transition relation:
  - $R(s, s') = \exists \text{inp. } s' = \delta(s, \text{inp})$
- Deterministic machine:
  - non-deterministic transition relation
  - existential quantification over inputs
  - so called “input non-determinism”

Example: n machines in parallel

- Assume $n$ state variables
  - $\text{states} = \text{states}_1 \times \cdots \times \text{states}_n$
  - $\vec{v} = (v_1, \ldots, v_n)$
- Assume $n$ transition functions
  - $\delta_i : \text{states} \times \text{inputs} \to \text{states}$, $(1 \leq i \leq n)$
- Note: each machine $\delta_i$ reads all inputs and states
- An $R$-step is a non-deterministically chosen step of one machine
  - $R(\vec{v}, \vec{v}') = \exists \text{inp.}$
    - $v'_1 = \delta_1(\vec{v}, \text{inp}) \land v'_2 = v_2 \land \cdots \land v'_n = v_n$
    - $v'_1 = v_1 \land v'_2 = v'_2 \land \cdots \land v'_n = \delta_n(\vec{v}, \text{inp})$
- Asynchronous parallel composition
Explicit state property checking

• Goal: check some property \( P \) holds of all reachable states
  • e.g. \( P(s) \) means \( s \) has no errors

• Represent sets of states somehow

• Start with \( S_0 = \{ s \mid B \leq s \} \)

• Iteratively compute with \( S_{i+1} = S_i \cup \{ s \mid \exists u. u \in S_i \land \mathcal{R}(u,s) \} \)

• Note \( S_0 \subseteq S_1 \subseteq S_2 \subseteq \cdots \)
  • if finite number of states then eventually reach an \( s \) such that \( S_i = S_{i+1} \)
  • so \( S_i \) is set of reachable states

• Now check \( P(s) \) for every reachable \( s \) (i.e. for every \( s \in S_i \))

Symbolic approach: representing sets as formulas

• Set \( \{ b_1, b_2, \ldots, b_n \} \) represented by formula \( v = b_1 \lor v = b_2 \lor \cdots \lor v = b_n \)
  • \( b_1, b_2, \ldots, b_n \) are truth-values (i.e. \( T \) or \( F \))
  • \( v \) is a boolean variable

• A set of states
  \( \{ \langle b_1, \ldots, b_m \rangle, \ldots, \langle b_n, \ldots, b_m \rangle \} \)
  is represented by a formula with \( m \) boolean variables:
  \( (v_1 = b_1 \land \cdots \land v_m = b_m) \lor \cdots \lor (v_1 = b_1 \land \cdots \land v_m = b_m) \)

• To test if \( \{ b_1, \ldots, b_m \} \) is in the set,
  just evaluate the formula with \( v_1 = b_1, \ldots, v_m = b_m \), i.e. evaluate:
  \( (v_1 = b_1 \land \cdots \land v_m = b_m) \lor \cdots \lor (v_1 = b_1 \land \cdots \land v_m = b_m) \)

Transition relations as Boolean Formulas

• Part of a handshake circuit
  (model at cycle level – registers are unit delays)

• Primed variables \( \langle \text{dreq}', \text{q0}', \text{dack}' \rangle \) represent ‘next state’

• Transition relation is:
  \( (\text{q0}' = \text{dreq}) \land (\text{dack}' = \text{dreq} \lor (\neg \text{dreq} \land \text{dack})) \)

• Transition relation equivalent to:
  \( (\text{q0}' = \text{dreq}) \land (\text{dack}' = \text{dreq} \lor (\neg \text{dreq} \land \text{dack})) \)

• Define \( \mathcal{N}_{\text{RECEIVED}} \) by:
  \( \mathcal{N}_{\text{RECEIVED}}(\langle \text{dreq}, \text{q0}, \text{dack} \rangle, \langle \text{dreq}', \text{q0}', \text{dack}' \rangle) = \)
  \( (\text{q0}' = \text{dreq}) \land (\text{dack}' = \text{dreq} \lor (\neg \text{dreq} \land \text{dack})) \)

• \text{dreq}' unconstrained, hence non-determinism

Symbolic reachability: sets of states are formulas

• Condition for a state \( s \) to be reachable
  in one \( \mathcal{R} \)-step from a state in \( B \)
  \( \exists u. B \land \mathcal{R}(u,s) \)

• Define \( \text{ReachBy} n \mathcal{R} B \) to be sets of states reachable in at most \( n \) steps:
  \( \exists u. B \land \mathcal{R}(u,s) \)

• \( \text{ReachBy} (n+1) \mathcal{R} B \) is
  \( \text{ReachBy} n \mathcal{R} B \land \mathcal{R}(u,s) \)

• Reachable states are states reachable in a finite number of steps:
  \( \exists n. \text{ReachBy} n \mathcal{R} B \land \mathcal{R}(u,s) \)

• Key property (equality between predicates represents set equality):
  \( \exists n. \text{ReachBy} n \mathcal{R} B = \text{ReachBy} (n+1) \mathcal{R} B \)
  \( \exists n. \text{ReachBy} n \mathcal{R} B = \text{ReachBy} n \mathcal{R} B \)
Represent formulas as Binary Decision Diagrams

- Reduced Ordered Binary Decision Diagrams (ROBDDs or BDDs for short) are a data-structure for representing Boolean formulas.
- Key features:
  - Canonical (given a variable ordering)
  - Efficient to manipulate
- Variables: \( v \) if \( v \) then \( 1 \) else \( 0 \) and \( \neg v \) if \( v \) then \( 0 \) else \( 1 \)
- Example: BDDs of variable \( v \) and \( \neg v \)

More BDD examples

- BDD of \( v_1 = v_2 \)

- Example: BDDs of \( v_1 \wedge v_2 \) and \( v_1 \vee v_2 \)

BDD of a transition relation

- BDDs of \( (v_1' = (v_1 = v_2)) \wedge (v_2' = (v_1 \oplus v_2)) \) with two different variable orderings

- Exercise: draw BDD of \( R_{\text{receiver}} \)

Standard BDD operations

- If formula \( f_x, f_z \) represents sets \( s_1, s_2 \), respectively then \( f_x \wedge f_z, f_x \vee f_z \) represent \( s_1 \cap s_2, s_1 \cup s_2 \), respectively
- Standard algorithms can compute boolean operation on BDDs.
- If \( f(x) \) represents \( \{ x \mid R(x) \} \) and \( g(x, x') \) represents \( \{ (x, x') \mid R(x, x') \} \) then \( \exists u. f(u) \wedge g(u, s) \) represents \( \{ s \mid \exists u. R(u, s) \} \)
- Exist algorithm to compute BDD of \( \exists u. h(u, e) \) from BDD of \( h(u, e) \)
  - BDD of \( \exists u. h(u, e) \) is BDD of \( h(T, e) \vee h(F, e) \)
- Given a BDD representing formula \( f \) with free variables \( v_1, \ldots, v_n \), there exists an algorithm to find truth-values \( b_1, \ldots, b_n \) such that if \( v_1 = b_1, \ldots, v_n = b_n \) then \( f \) evaluates to \( T \)
  - \( b_1, \ldots, b_n \) is a satisfying assignment (solution to SAT problem)
  - \( f(b_1, \ldots, b_n)/(v_1, \ldots, v_n) \) evaluates to \( T \)
  - used for counterexample generation (see later)
Reachable States via BDDs

- Represent $R(s, s')$ and $B$ as BDDs
- Iteratively compute BDDs of $S_0$, $S_1$, $S_2$ etc:
  - $S_0 = B$
  - $S_0 = S_0 \lor \exists u. S_0 \land R(u, s)$
  - $S_{n+1} = S_n \lor \exists u. S_0 \land R(u, s)$
- BDD of $\exists u. S_0 \land R(u, s)$ computed by:
  - $\exists u. (S_0[s/u]) \land R(s, s')(u, s')$

  Efficient using standard BDD algorithms (renaming, then conjunction, then existential quantification)

- At each iteration check $S_{n+1} = S_n$ efficient using BDDs, when $S_{n+1} = S_n$ can conclude
  - $R \ R' \ B \ s$
  - hence have computed BDD of $\text{Reach } R \ B \ s$

Example BDD optimisation: disjunctive partitioning

- Transition relation (asynchronous interleaving semantics):
  - $R(x, y, z), (x', y', z') =$
    - $(x' = \delta_1(x, y, z) \land y' = y \land z' = z) \lor$
    - $(x' = x \land y' = \delta_2(x, y, z) \land z' = z) \lor$
    - $(x' = x \land y' = y \land z' = \delta_3(x, y, z))$
- $\exists u. \ R(u, s)$ computed by $\exists u. (S_0[s/u]) \land R(s, s')(u, s')$
- More Details (Exercise: check the logic below)

Let $R(y, \pi, \tau)$ abbreviate $\text{ReachBy } R(B, \pi, \tau)$ then:
- $R(y, \pi, \tau) \land R((\pi, \tau, \gamma), (x, y, z))$
- $R(y, \pi, \tau) \land (x = \delta_1(\pi, \gamma) \land y = \gamma \land z = \tau) \lor$
- $(x = x \land y = \delta_2(\pi, \gamma) \land z = \tau) \lor$
- $(x = x \land y = \gamma \land z = \delta_3(\pi, \gamma))$
- $R(y, \pi, \tau) \land x = \delta_1(\pi, \gamma) \land y = \gamma \land z = \tau \lor$
- $(x = x \land y = \delta_2(\pi, \gamma) \land z = \tau) \lor$
- $(x = x \land y = \gamma \land z = \delta_3(\pi, \gamma))$
- $R(y, \pi, \tau) \land x = \delta_1(\pi, \gamma) \land y = \gamma \land z = \tau \lor$
- $(x = x \land y = \delta_2(\pi, \gamma) \land z = \tau) \lor$
- $(x = x \land y = \gamma \land z = \delta_3(\pi, \gamma))$
- $R(y, \pi, \tau) \land x = \delta_1(\pi, \gamma) \land y = \gamma \land z = \tau \lor$
- $(x = x \land y = \delta_2(\pi, \gamma) \land z = \tau) \lor$
- $(x = x \land y = \gamma \land z = \delta_3(\pi, \gamma))$
- $R(y, \pi, \tau) \land x = \delta_1(\pi, \gamma) \land y = \gamma \land z = \tau \lor$
- $(x = x \land y = \delta_2(\pi, \gamma) \land z = \tau) \lor$
- $(x = x \land y = \gamma \land z = \delta_3(\pi, \gamma))$
- $R(y, \pi, \tau) \land x = \delta_1(\pi, \gamma) \land y = \gamma \land z = \tau \lor$
- $(x = x \land y = \delta_2(\pi, \gamma) \land z = \tau) \lor$
- $(x = x \land y = \gamma \land z = \delta_3(\pi, \gamma))$

Avoiding building big BDDs

- Transition relation for three machines in parallel
  - $R(x, y, z), (x', y', z') =$
    - $(x' = \delta_1(x, y, z) \land y' = y \land z' = z) \lor$
    - $(x' = x \land y' = \delta_2(x, y, z) \land z' = z) \lor$
    - $(x' = x \land y' = y \land z' = \delta_3(x, y, z))$

- Recall:
  - $\text{ReachBy } (n+1) \ R \ B \ s$
  - $\equiv \text{ReachBy } n \ R \ B \ s \lor$
    - $\exists u. \text{ReachBy } n \ R \ B \ u \land R(u, s)$

- With $s = (x, y, z)$ it can be shown (see next slide):
  - $\text{ReachBy } (n+1) \ R \ B \ (x, y, z) =$
    - $\text{ReachBy } n \ R \ B \ (x, y, z) \lor$
      - $\exists u. \text{ReachBy } n \ R \ B \ (u, s) \land x = \delta_1(\pi, \gamma, z)) \lor$
      - $\exists u. \text{ReachBy } n \ R \ B \ (x, \gamma, z) \land y = \delta_1(\pi, \gamma, z)) \lor$
      - $\exists u. \text{ReachBy } n \ R \ B \ (x, y, \gamma) \land z = \delta_1(\pi, \gamma, z))$

- $R(u, s)$ not a subterm: ‘early quantification’, ‘disjunctive partitioning’
Verifications and Counterexamples

- Typical safety question:
  - Is $Q$ true in all reachable states?
  - I.e. is $Reach \mathcal{R} s \Rightarrow Q$ true?
- Compute BDD of $Reach \mathcal{R} s \Rightarrow Q$.
- Formula is true if BDD is the single node because $\top$ represented by a unique BDD (canonical property).
- If BDD is not $\top$ can get counterexample.

Example (from an exam)

Consider a 3x3 array of 9 switches.

Suppose each switch 1, 2, ..., 9 can either be on or off, and that toggling any switch will automatically toggle all its immediate neighbors. For example, toggling switch 5 will also toggle switches 2, 4, 6, and 8, and toggling switch 6 will also toggle switches 3, 5, and 9.

(a) Devise a state space [4 marks] and transition relation [6 marks] to represent the behavior of the array switches.

(b) You are given the problem of getting from an initial state in which even numbered switches are on and odd numbered switches are off, to a final state in which all the switches are off. Write down predicates on your state space that characterise the initial [2 marks] and final [2 marks] states.

(c) Explain how you might use a model checker to find a sequence of switches to toggle to get from the initial to final state. [6 marks]

You are not expected to actually solve the problem, but only to explain how to represent it in terms of model checking.

Generating Counterexample Traces

BDD algorithms can find satisfying assignments (SAT)

- Suppose $Reach \mathcal{R} s \Rightarrow Q$ is not true.
- Must exist an $s$ satisfying $Reach \mathcal{R} s \land \neg Q$.
- Find a counterexample algorithm:
  - Iteratively generate BDDs of $ReachBy i \mathcal{R} s \Rightarrow \neg Q$.
  - At each stage check if $ReachBy i \mathcal{R} s \land \neg Q$ is satisfiable.
  - Hence find first $i$ and, using SAT, a state $s_0$ such that $ReachBy i \mathcal{R} s_0 \land \neg Q$.
  - I.e. $ReachBy i \mathcal{R} s_0 \land \neg Q$.

- Then use BDD SAT to get $s_1, \ldots$, where $ReachBy (n-1) \mathcal{R} s_1 \land R(s_1,s_i) [s_i,1]$.
  - I.e. $ReachBy (n-1) \mathcal{R} s_n \land \neg Q$.

- Iteratively trace backwards to get $s_n, \ldots, s_0$ where for $0 < i \leq n$ $ReachBy (i-1) \mathcal{R} s_n \land R(s_n,s_i)$.

- Can sometimes apply partitioning, so BDD of $\neg Q$ not needed.

Solution

The state space can consist of the set of vectors $(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)$ where the boolean variable $v_i$ represents switch number $i+1$ and is true if and only if switch $i+1$ is off.

A transition relation $Trans$ is then defined by:

$$Trans((v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)) = ((v_0' = \neg v_0) \land (v_1' = \neg v_1) \land (v_2' = \neg v_2) \land (v_3' = \neg v_3) \land (v_4' = \neg v_4) \land (v_5' = \neg v_5) \land (v_6' = \neg v_6) \land (v_7' = \neg v_7) \land (v_8' = \neg v_8)) \quad \text{(toggle switch 1)}$$

$$\lor ((v_0' = \neg v_0) \land (v_1' = \neg v_1) \land (v_2' = \neg v_2) \land (v_3' = \neg v_3) \land (v_4' = \neg v_4) \land (v_5' = \neg v_5) \land (v_6' = v_6) \land (v_7' = \neg v_7) \land (v_8' = v_8)) \quad \text{(toggle switch 2)}$$

$$\lor ((v_0' = v_0) \land (v_1' = \neg v_1) \land (v_2' = \neg v_2) \land (v_3' = \neg v_3) \land (v_4' = \neg v_4) \land (v_5' = v_5) \land (v_6' = \neg v_6) \land (v_7' = \neg v_7) \land (v_8' = v_8)) \quad \text{(toggle switch 3)}$$

$$\lor ((v_0' = \neg v_0) \land (v_1' = v_1) \land (v_2' = v_2) \land (v_3' = \neg v_3) \land (v_4' = v_4) \land (v_5' = v_5) \land (v_6' = v_6) \land (v_7' = \neg v_7) \land (v_8' = v_8)) \quad \text{(toggle switch 4)}$$

$$\lor ((v_0' = v_0) \land (v_1' = v_1) \land (v_2' = v_2) \land (v_3' = \neg v_3) \land (v_4' = v_4) \land (v_5' = v_5) \land (v_6' = v_6) \land (v_7' = v_7) \land (v_8' = v_8)) \quad \text{(toggle switch 5)}$$

$$\lor ((v_0' = \neg v_0) \land (v_1' = v_1) \land (v_2' = \neg v_2) \land (v_3' = v_3) \land (v_4' = v_4) \land (v_5' = v_5) \land (v_6' = v_6) \land (v_7' = v_7) \land (v_8' = v_8)) \quad \text{(toggle switch 6)}$$

$$\lor ((v_0' = v_0) \land (v_1' = v_1) \land (v_2' = \neg v_2) \land (v_3' = v_3) \land (v_4' = \neg v_4) \land (v_5' = v_5) \land (v_6' = \neg v_6) \land (v_7' = v_7) \land (v_8' = v_8)) \quad \text{(toggle switch 7)}$$

$$\lor ((v_0' = \neg v_0) \land (v_1' = v_1) \land (v_2' = \neg v_2) \land (v_3' = v_3) \land (v_4' = v_4) \land (v_5' = v_5) \land (v_6' = v_6) \land (v_7' = \neg v_7) \land (v_8' = v_8)) \quad \text{(toggle switch 8)}$$

$$\lor ((v_0' = v_0) \land (v_1' = v_1) \land (v_2' = \neg v_2) \land (v_3' = v_3) \land (v_4' = v_4) \land (v_5' = v_5) \land (v_6' = v_6) \land (v_7' = v_7) \land (v_8' = v_8)) \quad \text{(toggle switch 9)}$$
Predicates $\text{Init}$, $\text{Final}$ characterising the initial and final states, respectively, are defined by:

$\text{Init}(v_0,v_1,v_2,v_3,v_4,v_5,v_6,v_7,v_8) = \\
\neg v_0 \land v_1 \land \neg v_2 \land v_3 \land \neg v_4 \land v_5 \land \neg v_6 \land v_7 \land \neg v_8$

$\text{Final}(v_0,v_1,v_2,v_3,v_4,v_5,v_6,v_7,v_8) = \\
\neg v_0 \land \neg v_1 \land \neg v_2 \land \neg v_3 \land \neg v_4 \land \neg v_5 \land \neg v_6 \land \neg v_7 \land \neg v_8$

Model checkers can find counter-examples to properties, and sequences of transitions from an initial state to a counter-example state. Thus we could use a model checker to find a trace to a counter-example to the property that $\neg \text{Final}(v_0,v_1,v_2,v_3,v_4,v_5,v_6,v_7,v_8)$.

Properties

- $\text{Reach } R \text{ s } \Rightarrow Q \text{ s }$ means $Q$ true in all reachable states
- Might want to verify other properties, e.g:
  1. $\text{DeviceEnabled}$ is always true somewhere along every path starting anywhere (i.e. it holds infinitely often along every path)
  2. From any state it is possible to get to a state for which $\text{Restart}$ holds
  3. Ack is true on all paths sometime between $i$ units of time later and $j$ units of time later.
- CTL is a logic for expressing such properties
- Exist efficient algorithms for checking them
- Model checking:
  - check property in a model
  - Emerson, Clarke & Sifakis, early 1980s – Turing award 2008
  - used in industry (e.g. IBM's RuleBase tool)
- Language wars: CTL vs LTL, PSL vs SVA

Concrete example

- Consider circuit below:
- Input: dreq, registers: q0, dack
- Timing Diagram:

If dreq rises, then it continues high, until it is acknowledged by a rise on dack.
- If dreq falls, then it will continue low until dack false.

Paths and computations

- Properties can asserted about complete computation trees (CTL)
- Properties can be asserted just about paths (LTL)
Paths, branching time and linear time

- Let \( \mathcal{R} \) have type \( \alpha \times \alpha \rightarrow \text{bool} \)
  - \( \mathcal{R} \) is a transition relation
  - \( \alpha \) ranges (intuitively) over states
- An \( \mathcal{R} \)-path is a function \( \sigma : \text{num} \rightarrow \alpha \) such that: \( \forall t. \mathcal{R}(\sigma(t), \sigma(t+1)) \)
- Path(\( \mathcal{R}, s \))\( \sigma \) means \( \sigma \) is an \( \mathcal{R} \)-path from \( s \)
  \[ \text{Path}(\mathcal{R}, s)\sigma = (\sigma(0)=s) \land \forall t. \mathcal{R}(\sigma(t), \sigma(t+1)) \]

- CTL is a branching time logic
  - properties may hold along all paths – A
  - properties may hold along some paths – E
- LTL is a linear time logic
  - only properties along all paths – no path quantifiers

Semantics of CTL (shallow embedding)

- A model is a pair \( (\mathcal{R}, s) \) — a transition relation and an initial state
- Define:
  \[ \text{Atom}(p) = \lambda (\mathcal{R}, s). p(s) \]
  \[ \neg P = \lambda (\mathcal{R}, s). \neg (P(\mathcal{R}, s)) \]
  \[ P \land Q = \lambda (\mathcal{R}, s). P(\mathcal{R}, s) \land Q(\mathcal{R}, s) \]
  \[ P \lor Q = \lambda (\mathcal{R}, s). P(\mathcal{R}, s) \lor Q(\mathcal{R}, s) \]
  \[ P \Rightarrow Q = \lambda (\mathcal{R}, s). P(\mathcal{R}, s) \Rightarrow Q(\mathcal{R}, s) \]
  \[ \text{AX} P = \lambda (\mathcal{R}, s). \forall \sigma. \text{Path}(\mathcal{R}, s)\sigma \]
  \[ \text{EX} P = \lambda (\mathcal{R}, s). \exists \sigma. \text{Path}(\mathcal{R}, s)\sigma \]
  \[ \text{A}[P \cup Q] = \lambda (\mathcal{R}, s). \forall \sigma. \text{Path}(\mathcal{R}, s)\sigma \]
  \[ \Rightarrow \exists i. Q(\mathcal{R}, \sigma(i)) \land \forall j. j < i \Rightarrow P(\mathcal{R}, \sigma(j)) \]
  \[ \text{E}[P \cup Q] = \lambda (\mathcal{R}, s). \exists \sigma. \text{Path}(\mathcal{R}, s)\sigma \]
  \[ \Rightarrow \exists i. Q(\mathcal{R}, \sigma(i)) \land \forall j. j < i \Rightarrow P(\mathcal{R}, \sigma(j)) \]

Computation Tree Logic (CTL)

- Syntax of CTL well-formed formulas:
  \[ \text{wff} ::= \text{Atom}(p) \]
  \[ \neg \text{wff} \]
  \[ \text{wff}_1 \land \text{wff}_2 \]
  \[ \text{wff}_1 \lor \text{wff}_2 \]
  \[ \text{wff}_1 \Rightarrow \text{wff}_2 \]
  \[ \text{AX} \text{wff} \]
  \[ \text{EX} \text{wff} \]
  \[ \text{A}[\text{wff}_1 \cup \text{wff}_2] \]
  \[ \text{E}[\text{wff}_1 \cup \text{wff}_2] \]

- Atomic formulas \( p \) are properties of states
  - sometimes just write "\( p \)" rather than "\( \text{Atom}(p) \)"
- General CTL formulas \( P \) are properties of models

The defined operator AF

- Define \( \text{AF} P = \text{A} [\top \cup P] \)
- \( \text{AF} P \) is true if \( P \) holds somewhere along every \( \mathcal{R} \)-path — \( P \) is \textit{w-invariant}
  \[ \text{AF} P = \lambda (\mathcal{R}, s). \exists \sigma. \text{Path}(\mathcal{R}, s)\sigma \]
  \[ \Rightarrow \exists i. P(\mathcal{R}, \sigma(i)) \land \forall j. j < i \Rightarrow P(\mathcal{R}, \sigma(j)) \]
  \[ = \lambda (\mathcal{R}, s). \exists \sigma. \text{Path}(\mathcal{R}, s)\sigma \]
  \[ \Rightarrow \exists i. P(\mathcal{R}, \sigma(i)) \land \forall j. j < i \Rightarrow P(\mathcal{R}, \sigma(j)) \]
The defined operator EF

- Define $\text{EF}P = E[T \cup P]$
- $\text{EF}P$ is true if $P$ holds somewhere along some $R$-path
  - i.e. $P$ potentially holds

$\text{EF}P$

$= E[T \cup P]$

$= \lambda (R, s).$

$\exists \tau.$

$\forall j. P(R, \tau(j)) \land j < i \Rightarrow T(R, \tau(j))$

The defined operator AG

- Define $\text{AG}P = \neg \text{EF}(\neg P)$
- $\text{AG}P$ is true if $P$ holds everywhere along every $R$-path

$\text{AG}P = \neg \text{EF}(\neg P)$

$= \lambda (R, s).$ $\neg (\exists \tau. \text{Path}(R, s) \sigma \land \exists i. \neg P(R, \sigma(i)))$

The defined operator EG

- $\text{EG}P$ is true if $P$ holds everywhere along some $R$-path

$\text{EG}P = \neg \text{AF}(\neg P)$

$= \lambda (R, s).$ $\neg (\forall \tau. \text{Path}(R, s) \sigma \Rightarrow \exists i. \neg P(R, \sigma(i)))$

The defined operator $\text{A}[P\text{WQ}]$

- $\text{A}[P\text{WQ}]$ is a ‘partial correctness’ version of $\text{A}[P\text{UQ}]$
- It is true if along a path if
  - $P$ always holds along the path
  - $Q$ holds sometime on the path, and until it does $P$ holds
- Define $\text{A}[P\text{WQ}]$

$= \lambda (R, s).$ $\neg (\exists \tau. \text{Path}(R, s) \sigma$

$\land \exists i. \neg P(R, \sigma(i)))$

Exercise: understand the next three slides
**A[PWQ] continued (1)**

- Continuing:
  \[ \lambda(R, s). \]
  \[ \neg(\exists s. \text{Path}(R, s) \sigma) \land \neg(\forall j. j < i \Rightarrow (P \land \neg Q)(R, \sigma(j))) \]

**A[PWF] = AG P**

- From last slide:
  \[ A[PWF] \]
  \[ = \lambda(R, s). \]
  \[ \forall s. \text{Path}(R, s) \sigma \]
  \[ \Rightarrow (\forall j. j < i \Rightarrow P(R, \sigma(j)) \land \neg Q(R, \sigma(j))) \]
  \[ \Rightarrow P(R, \sigma(i)) \lor Q(R, \sigma(i)) \]

**A[PWF] = \lambda(R, s). \forall s. \text{Path}(R, s) \sigma \Rightarrow (\forall j. j < i \Rightarrow P(R, \sigma(j)) \land \neg Q(R, \sigma(j))) \Rightarrow P(R, \sigma(i))**

- Set \( Q \) to be \( F \):
  \[ A[PWF] \]
  \[ = \lambda(R, s). \forall s. \text{Path}(R, s) \sigma \]
  \[ \Rightarrow (\forall j. j < i \Rightarrow P(R, \sigma(j)) \land \neg F(R, \sigma(j))) \]
  \[ \Rightarrow P(R, \sigma(i)) \lor F(R, \sigma(i)) \]

**A[PWF] = \lambda(R, s). \forall s. \text{Path}(R, s) \sigma \Rightarrow (\forall j. j < i \Rightarrow P(R, \sigma(j)) \land \neg F(R, \sigma(j))) \Rightarrow P(R, \sigma(i))**

- Simplify:
  \[ A[PWF] \]
  \[ = \lambda(R, s). \forall s. \text{Path}(R, s) \sigma \Rightarrow (\forall j. j < i \Rightarrow P(R, \sigma(j)) \Rightarrow P(R, \sigma(i))) \]

**Exercise: describe the property specified by A[PWF]**

- Exercise: this exercise illustrates the subtlety of writing CTL!
Example of current research

TCAD Newsletter – March 2010 Issue
Placing you one click away from the best new CAD research!

Regular Papers

Zheng, H.; “Compositional Reachability Analysis for Efficient Modular Verification of Asynchronous Designs”

Abstract: Compositional verification is essential to address state explosion in model checking. Traditionally, an over-approximate context is needed for each individual component in a system for sound verification. This may cause state explosion for the intermediate results as well as inefficiency for abstraction refinement. This paper presents an opposite approach, a compositional reachability method, which constructs the state space of each component from an under-approximate context gradually until a counter-example is found or a fixpoint in state space is reached. This method has an additional advantage in that counter-examples, if there are any, can be found much earlier, thus leading to faster verification. Furthermore, this modular verification framework does not require complex compositional reasoning rules. The experimental results indicate that this method is promising.

URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5419238&isnumber=5419222

Example CTL formulas

- EF(Started ∧ ¬Ready)
  It is possible to get to a state where Started holds but Ready does not hold
- AG(Req ⇒ AF Ack)
  If a request Req occurs, then it will eventually be acknowledged by Ack
- AG(AF DeviceEnabled)
  DeviceEnabled is always true somewhere along every path starting anywhere: i.e. DeviceEnabled holds infinitely often along every path
- AG(EF Restart)
  From any state it is possible to get to a state for which Restart holds

Summary of CTL operators (primitive + defined)

- CTL formulas:
  Atom(p) (Atomic formula - p: states→bool)
  ¬P (Negation)
  P ∧ Q (Conjunction)
  P ∨ Q (Disjunction)
  P ⇒ Q (Implication)
  AX P (All successors)
  EX P (Some successors)
  AF P (Somewhere – along all paths)
  EF P (Somewhere – along some path)
  AG P (Everywhere – along all paths)
  EG P (Everywhere – along some path)
  A[P U Q] (Until – along all paths)
  E[P U Q] (Until – along some path)
  A[P W Q] (Unless – along all paths)
  E[P W Q] (Unless – along some path)

- Say ‘P holds’ if P(R, s) for all initial states s

More CTL examples (1)

- AG(Req ⇒ A[Req U Ack])
  If a request Req occurs, then it continues to hold, until it is eventually acknowledged
- AG(Req ⇒ AX(A[¬Req U Ack]))
  Whenever Req is true either it must become false on the next cycle and remains false until Ack, or Ack must become true on the next cycle
  Exercise: is the AX necessary?
- AG(Req ⇒ (¬Ack ⇒ AX(A[Req U Ack])))
  Whenever Req is true and Ack is false then Ack will eventually become true and until it does Req will remain true
  Exercise: is the AX necessary?
More CTL examples (2)

- AG[Enabled ⇒ AG[Start ⇒ A[¬Waiting U Ack]]]
  
  If Enabled is ever true then if Start is true in any subsequent state then Ack will eventually become true, and until it does Waiting will be false

- AG[¬Req, ∧ ¬Req]
  
  Whenever Req and Req are false, then remain false until Start becomes true with Req still false

- AG[Req ⇒ AX(Ack ⇒ AF ¬Req)]
  
  If Req is true and Ack becomes true one cycle later, then eventually Req will become false

Some abbreviations

- $$\text{AX}_i P = \text{AX}(\text{AX}(\cdot \cdot \cdot \text{AX}(P) \cdot \cdot \cdot))$$
  
  i instances of AX
  
  $$P$$ is true on all paths $$i$$ units of time later

- $$\text{ABF}_{i,j} P = \text{AX}(P \lor \text{AX}(P \lor \cdot \cdot \cdot \text{AX}(P \lor \text{AX}(P) \cdot \cdot \cdot))$$
  
  $$j$$ - $$i$$ instances of AX
  
  $$P$$ is true on all paths sometime between $$i$$ units of time later and $$j$$ units of time later

- $$\text{AG}(\text{Req} ⇒ \text{AX}[\text{Ack} \land \text{ABF}_{1,6} (\text{Ack} \land A \text{[Wait U Reply]])])$$
  
  One cycle after Req, Ack should become true, and then Ack becomes true $$i$$ to $$6$$ cycles later and then eventually Reply becomes true, but until it does Wait holds from the time of Ack

- More abbreviations in the ‘Industry Standard’ language PSL

---

CTL model checking algorithm

- A model is a relation $$R$$

- A property is a CTL formula $$P$$

- Model checking: given CTL formula $$P$$ compute $$\{s \mid P(R, s)\}$$

- $$P(R, s_0)$$ true if and only if $$s_0 \in \{s \mid P(R, s)\}$$

- Assume set of states to be finite

  (infinite state model checking possible for some models)

- Already seen how to model check reachability

  $$\Delta G(\text{Atom} p)(R, s) = \forall s'. \text{Reach } R (\text{Eq } s) s' ⇒ p(s')$$

  so can model check AG of atomic properties – compute:

  $$\{s' \mid \text{Reach } R (\text{Eq } s) s' ⇒ p(s')\}$$,

  e.g. via BDD of

  Reach $$R (\text{Eq } s) s' ⇒ p(s')$$

Checking $$\text{EF} \text{Atom}(p)$$

- EF$$(\text{Atom} p)(R, s)$$ if $$p$$ holds along some path starting at $$s$$

  Mark all the states satisfying $$p$$

  Repeatedly mark all the states which have at least one marked successor until no change

  $$\{s \mid \text{EF}(\text{Atom} p)(R, s)\}$$ computed by generating:

  $$\mathcal{S}_0 = \{s \mid \text{Atom} p(R, s)\}$$

  $$\{s \mid p(s)\}$$

  $$\mathcal{S}_{i+1} = \mathcal{S}_i \cup \{s \mid \exists s'. R(s, s') \land s' \in \mathcal{S}_i\}$$

- $$\text{EF}(\text{Atom } p)$$ is true in marked states and false in unmarked states

- Algorithm similar for $$\text{AF}(\text{Atom} p)$$:

  Repeatedly mark all the states which have all successors marked

- To check AF $$\text{EF}(\text{Atom} p)$$:

  apply EF algorithm

  starting with resulting marking apply AF algorithm
Recall handshake example

- Part of a handshake circuit

- Transition relation:
  \[(q^0 = \text{dreq}) \land (\text{dack} = \text{dreq} \land (q^0 \lor \text{dack}))\]

- Define \( \text{RECEIVER} \):
  \[
  \text{RECEIVER}((\text{dreq},q_0,\text{dack}),(\text{dreq}',q_0',\text{dack}')) = (q^0 = \text{dreq} \land (\text{dack} = \text{dreq} \land (q^0 \lor \text{dack})))
  \]

- Primed variables \((\text{dreq}',q_0',\text{dack}')\) represent ‘next state’
- \(\text{dreq}'\) unconstrained, hence non-determinism

![Handshake Circuit Diagram]

Model checking \( \text{RECEIVER} \)

- Possible states for \( \text{RECEIVER} \):
  \[\{000, 001, 010, 011, 100, 101, 110, 111\}\]
  where \( \text{b}_0\text{b}_1\text{b}_2 \) denotes state
  \(\text{dreq} = \text{b}_2 \land q_0 = \text{b}_1 \land \text{dack} = \text{b}_0\)

- Graph of the transition relation:

  ![Transition Relation Graph]

- Above a state indicates membership of \( \mathcal{S} \)
  (defined below)

Example: \( EF(dreq \land q_0 \land \text{dack}) \)

- Define:
  \[P = \text{atom}(\lambda \text{b}_2\text{b}_1\text{b}_0, \text{b}_2 \land \text{b}_1 \land \text{b}_0)\]
  \[P(\text{RECEIVER}, \text{b}_2\text{b}_1\text{b}_0) = \text{b}_2 \land \text{b}_1 \land \text{b}_0\]

- Define:
  \[\mathcal{S}_0 = \{\text{b}_2\text{b}_1\text{b}_0 \mid P(\text{RECEIVER}, \text{b}_2\text{b}_1\text{b}_0)\}\]
  \[\mathcal{S}_{i+1} = \mathcal{S}_i \cup \{s \mid \exists s'. \mathcal{R}(s,s') \land s' \in \mathcal{S}_i\}\]

- \(\mathcal{S}_1 = \mathcal{S}_0 \cup \{\text{b}_2\text{b}_1\text{b}_0 \mid \exists \text{b}_2\text{b}_1\text{b}_0. (\text{b}_2' = \text{b}_2) \land (\text{b}_1' = \text{b}_2 \land \text{b}_1 \lor \text{b}_0) \land \text{b}_0'\text{b}_1'\text{b}_2' \in \mathcal{S}_1\}\)

Checking \( EF(dreq \land q_0 \land \text{dack}) \)

- Recall:
  \[\mathcal{S}_0 = \{\text{b}_2\text{b}_1\text{b}_0 \mid P(\text{RECEIVER}, \text{b}_2\text{b}_1\text{b}_0)\}\]
  \[\mathcal{S}_{i+1} = \mathcal{S}_i \cup \{\text{b}_2\text{b}_1\text{b}_0 \mid \exists \text{b}_2\text{b}_1\text{b}_0. (\text{b}_2' = \text{b}_2) \land (\text{b}_1' = \text{b}_2 \land \text{b}_1 \lor \text{b}_0) \land \text{b}_0'\text{b}_1'\text{b}_2' \in \mathcal{S}_i\}\]

- Define:
  \[\mathcal{S}_1 = \{111\}\]
  \[\mathcal{S}_2 = \{111, 101, 110\}\]
  \[\mathcal{S}_3 = \{111, 101, 110, 100\}\]
  \[\mathcal{S}_4 = \{111, 101, 110, 100, 000, 001, 010, 011\}\]
  \[\mathcal{S}_5 = \{111, 101, 110, 100, 000, 001, 010, 011\}\]

- Compute:
  \[\mathcal{S}_1 = \{111\}\]
  \[\mathcal{S}_2 = \{111, 101, 110\}\]
  \[\mathcal{S}_3 = \{111, 101, 110, 100\}\]
  \[\mathcal{S}_4 = \{111, 101, 110, 100, 000, 001, 010, 011\}\]
  \[\mathcal{S}_5 = \{111, 101, 110, 100, 000, 001, 010, 011\}\]

- Hence \(\forall s. EF(\text{atom}(\lambda \text{dreq}, \text{q}_0, \text{dack}), \text{dreq} \land \text{q}_0 \land \text{dack})/\text{RECEIVER}, s)\)
Symbolic model checking

- Represent sets of states with BDDs
- Represent transition relation with a BDD
- If BDDs of $P(R, s)$, $Q(R, s)$ are known, then BDDs of
  $\neg P(R, s)$
  $P(R, s) \land Q(R, s)$
  $P(R, s) \lor Q(R, s)$
  $P(R, s) \Rightarrow Q(R, s)$
can be computed using standard BDD algorithms
- If BDDs of $P(R, s)$, $Q(R, s)$ are known, then BDDs of
  $AX P(R, s)$, $EX P(R, s)$, $A[P \lor Q](R, s)$, $E[P \lor Q](R, s)$
computed using fairly straightforward algorithms (see textbooks)
- Model checking CTL generalises iteration for reachable states ($\Delta$)

History of Model checking

- CTL model checking invented by Emerson, Clarke and Sifakis
- Use of BDDs to represent and compute sets of states is called
  symbolic model checking
- Independently discovered by several people:
  Clarke & McMillan
  Coudert, Berthet & Madre
  Pixley
- SMV (McMillan) is a popular symbolic model checker
  http://www.cs.cmu.edu/~modelcheck/smv.html (original)
  http://www.kenmcmil.com/smv.html (Cadence extension by McMillan)
  http://nusmv.irst.itc.it/ (new implementation)
- Other temporal logics
  - Linear temporal logic (LTL): easier to use, more complicated to check
  - CTL*: combines CTL and LTL (also harder to check)
  - Industrial languages PSL and SVA designed to be 'engineer friendly'

Expressibility of CTL

- Consider the property
  "on every path there is a point after which \textit{p} is always true on that path."

- Consider

  \[ \begin{array}{c}
    a_0 \\
    a_1 \\
    a_2 \\
    \vdots
  \end{array} \]

- Property true, but cannot be expressed in CTL
  - would need something like $AF P$
  - where $P$ is something like "property \textit{p} true from now on"
  - but in CTL $P$ must start with a path quantifier $A$ or $E$
  - so cannot talk about current path, only about all or some paths
  - $AF AG (\text{always})$ is false (consider path $a_0 \ast a_0 \ast a_0 \ast a_0 \ast a_0 \ast$)

Linear Temporal Logic (LTL)

- CTL property is a predicate on a state in a tree: $P(R, s)$
- LTL property is a predicate on a path: $P(\sigma)$
- Syntax of LTL well-formed formulae:
  \[
  \text{wff} ::= \text{Atom} \mid \neg \text{wff} \mid \text{wff}_1 \lor \text{wff}_2 \mid X \text{wff} \mid F \text{wff} \mid G \text{wff} \mid [\text{wff}_1 \U \text{wff}_2]
  \]
- Notice no path quantifiers $A$ or $E$
**Semantics of LTL (shallow embedding)**

- **Define** \( \text{Tail}_m \sigma = \lambda n. \sigma(n+m) \)
- **Define:**
  - \( \text{Atom}(p) = \lambda \sigma. p(\sigma(0)) \)
  - \( \neg \alpha = \lambda \sigma. \neg \alpha(\sigma) \)
  - \( \alpha \lor \beta = \lambda \sigma. \alpha(\sigma) \lor \beta(\sigma) \)
  - \( \alpha*X = \lambda \sigma. X(\text{Tail}_1(\sigma)) \)
  - \( \alpha+F = \lambda \sigma. \exists m. \alpha(\text{Tail}_m(\sigma)) \)
  - \( \alpha+G = \lambda \sigma. \forall m. \alpha(\text{Tail}_m(\sigma)) \)
  - \( \alpha[U \beta] = \lambda \sigma. \exists i. \beta(\text{Tail}_i(\sigma)) \land \forall j. j < i \Rightarrow \alpha(\text{Tail}_j(\sigma)) \)

**Example:**
\[
X(\text{Atom}(p))(\sigma) = \text{Atom}(p)(\text{Tail}_1(\sigma)) = p(\sigma(1))
\]

**FG**

- \( \alpha^F \) is true if there is a point after which \( \alpha \) is always true
  \[
  \alpha^F = \lambda \sigma. (G(\alpha))(\text{Tail}_m(\sigma))
  \]

- **Recall:**

**CTL can express things that LTL can’t express**

- AG(EF P) says:
  “from every state it is possible to get to a state for which \( P \) holds”
- Can’t say this in LTL (proof omitted)
- Consider disjunction:
  “along every path there is a state from which \( P \) will hold forever or from every state it is possible to get to a state for which \( P \) holds”
- Can’t say this in either CTL or LTL! (proof omitted)
- CTL* combines CTL and LTL and can express this property

**CTL* defined mutually recursively**

- State formulas \( \text{swff} \):
  - Atomic formula
  - Negation
  - Disjunction
  - All paths
  - Some paths
- Path formulas \( \text{puff} \):
  - Every state formula is a path formula
  - Negation
  - Disjunction
  - Successor
  - Sometimes
  - Always
  - Until

**CTL** restricted with X, F, G, \([-U]\) preceded by A or E

**LTL** consists of CTL* formulas of form A+puff, where the only state formulas in puff are atomic

**Selection of primitives above arbitrary:** \( \lor, \neg, X, U, E \) enough
**CTL* semantics**

- Combining state semantics of CTL with path semantics of LTL:
  
  - $\text{Atom}(p) = \lambda(R, s). p(s)$
  - $\neg S = \lambda(R, s). \neg(S(R, s))$
  - $S_1 \lor S_2 = \lambda(R, s). S_1(R, s) \lor S_2(R, s)$
  - $AP = \lambda(R, s). \forall \sigma. \text{Path}(R, s)\sigma \Rightarrow P(R, \sigma)$
  - $EP = \lambda(R, s). \exists \sigma. \text{Path}(R, s)\sigma \land P(R, \sigma)$

- **PathForm**
  - $\neg P = \lambda(R, s). \neg(P(R, s))$
  - $P_1 \lor P_2 = \lambda(R, s). P_1(R, s) \lor P_2(R, s)$
  - $XP = \lambda(R, s). P(R, \text{Tail} i \sigma)$
  - $GP = \lambda(R, s). \forall m. P(R, \text{Tail} m \sigma)$
  - $[P_1 \cup P_2] = \lambda(R, s). \exists i. P_1(R, \text{Tail} i \sigma) \land \forall j. j < i \Rightarrow P_1(R, \text{Tail} j \sigma)$

- **Note semantics of state and path formulas have different types**
  - $\lambda(R, s)$ versus $\lambda(R, \sigma)$

- **Semantics looks simpler if we assume $R$ fixed**

---

**Fairness**

- May want to assume a component or the environment is ‘fair’
- **Example 1**: fair arbiter
  - the arbiter doesn’t ignore one of its requests forever
    - not every request need be granted
    - want to exclude infinite number of requests and no grant
- **Example 2**: reliable channel
  - no message continuously transmitted but never received
    - not every message need be received
    - want to exclude an infinite number of sends and no receive
- **Want if $P$ holds infinitely often along a path then so does $Q$**
- In LTL is expressible as $G(F P) \Rightarrow G(F Q)$

- **Can’t say this in CTL**
  - why not – what’s wrong with $AG(AF P) \Rightarrow AG(AF Q)$?
  - in CTL* expressible as $\lambda(G(F P) \Rightarrow G(F Q))$
  - fair CTL model checking is implemented in the model checking algorithm
  - fair LTL just needs a fairness assumption like $G(F P) \Rightarrow \ldots$

- **Fairness is a tricky and subtle subject**
  - several notions or fairness: ‘weak fairness’, ‘strong fairness’ etc
  - exist whole books on fairness

---

**Simplified CTL* semantics (textbook semantics)**

- Let Path $s \sigma$ abbreviate Path($R, s, \sigma$), then:
  
  - $\text{Atom}(p) = \lambda s. p(s)$
  - $\neg S = \lambda s. \neg(S(s))$
  - $S_1 \lor S_2 = \lambda s. S_1(s) \lor S_2(s)$
  - $AP = \lambda s. \forall \sigma. \text{Path}(s, \sigma) \Rightarrow P(s, \sigma)$
  - $EP = \lambda s. \exists \sigma. \text{Path}(s, \sigma) \land P(s, \sigma)$

- **PathForm**
  - $\neg P = \lambda s. \neg(P(s))$
  - $P_1 \lor P_2 = \lambda s. P_1(s) \lor P_2(s)$
  - $XP = \lambda s. P(\text{Tail} i s)$
  - $GP = \lambda s. \forall m. P(\text{Tail} m s)$
  - $[P_1 \cup P_2] = \lambda s. \exists i. P_1(\text{Tail} i s) \land \forall j. j < i \Rightarrow P_1(\text{Tail} j s)$

---

**Propositional modal $\mu$-calculus**

- **Modal $\mu$-calculus is an even more powerful property language**
- **Has fixed-point operators**
  - both maximal and minimal fixed points
  - model checking consists of calculating fixed points
  - many logics (e.g. CTL*) can be translated into $\mu$-calculus
- **Strictly stronger than CTL***
  - expressibility in $\mu$-calculus strictly increases as allowed nesting increases
  - need fixed point operators nested 2 deep for CTL*
- The $\mu$-calculus is **very non-intuitive to use!**
  - intermediate code rather than a practical property language
  - nice meta-theory and algorithms, but terrible usability!
**Interval Temporal Logic (ITL)**

- ITL specifies properties of intervals
- An interval is a sequence of states with a beginning and an end
- Useful for talking about ‘transactions’
- ITL specifies properties of finite intervals not infinite traces
- Has an executable subset called Tempura suitable for simulation
- Developed by Ben Moszkowski at Stanford then here at Cambridge
- Moszkowski is now at De Montford University

**Examples of ITL**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1; P_2$</td>
<td>$P_1$ holds then $P_2$ holds (overlapping state)</td>
</tr>
<tr>
<td>$P_1; \text{skip}; P_2$</td>
<td>$P_1$ holds then $P_2$ holds (no overlapping state)</td>
</tr>
<tr>
<td>$\text{skip}; P$</td>
<td>$P$ true on the next state</td>
</tr>
<tr>
<td>$\text{true}; P$</td>
<td>$P$ sometimes true</td>
</tr>
<tr>
<td>$\neg \text{true}; \neg P$</td>
<td>$P$ always true</td>
</tr>
</tbody>
</table>

**ITL (simplified and with expressions omitted)**

- **Syntax of ITL well-formed formulae:**
  
  \[
  \text{wff} ::= \text{Atom}(p) \quad \text{(Atomic formula)} \\
  | \text{true} \quad \text{(Truth)} \\
  | \neg \text{wff} \quad \text{(Negation)} \\
  | \text{wff}_1 \lor \text{wff}_2 \quad \text{(Disjunction)} \\
  | \text{skip} \quad \text{(interval with exactly two states)} \\
  | \text{wff}_1 \cdot \text{wff}_2 \quad \text{(Chop)} \\
  | \text{wff}^* \quad \text{(Repeat)}
  \]

- **Semantics (properties are predicates on intervals):**
  
  \[
  \text{Atom}(p) = \lambda \langle s_0 \cdots s_n \rangle. \ p(s_0) \\
  \text{true} = \lambda \langle s_0 \cdots s_n \rangle. \ T \\
  \neg P = \lambda \langle s_0 \cdots s_n \rangle. \ \neg (P(s_0 \cdots s_n)) \\
  P \lor Q = \lambda \langle s_0 \cdots s_n \rangle. \ P(s_0 \cdots s_n) \lor Q(s_0 \cdots s_n) \\
  \text{skip} = \lambda \langle s_0 \cdots s_n \rangle. \ n = 1 \\
  P \cdot Q = \lambda \langle s_0 \cdots s_n \rangle. \ \exists k. \ k \leq n \land P(s_0 \cdots s_k) \land Q(s_k \cdots s_n) \\
  P^* = \lambda \langle s_0 \cdots s_n \rangle. \\exists w_1 \cdots w_l. \ \langle s_0 \cdots s_n \rangle = w_1 \cdots w_l \land P w_1 \land \cdots \land P w_l
  \]

**Too many logics: CTL, ITL, CTL*, ITL, ...**

- Large variety of separate logics
- Can be viewed as idioms in higher order logic
- Can model complete hardware systems in higher order logic
- Can model programming languages and logics in higher order logic
- Why not dump ad hoc languages and just work in logic?
  - specialized logics support specialized specification and verification methods
  - compact assertions developed for specific applications
Assertion-based verification (ABV)

- Claimed that assertion based verification:
  
  "is likely to be the next revolution in hardware design verification"

- Basic idea:
  
  - document designs with formal properties
  - check properties using both simulation (dynamic) and model checking (static)

- Accellera organisation and IEEE are specifying languages

- Frequently used acronyms

  PSL: Property Specification Language
  OVL: Open Verification Library (Verilog modules)
  OVA: Open Vera Language
  SVA: System Verilog Assertions
  SVL: System Verilog assertion Library (SVA version of OVL)

- Problem: too many languages

  - PSL from Accellera Formal Verification Technical Committee
  - OVA/SVA from Accellera SystemVerilog Assertion Committee
  - OVL from Accellera Open Verification Library Technical Committee
  - all Accellera committees + some new IEEE committees?

- PSL and OVA/SVA have been ‘aligned’

- OVL is a checker library for dynamic property verification
  
  - currently VHDL, Verilog and PSL versions
  - eventually PSL version golden and others derived ................. maybe

IBM’s Sugar and Accellera’s PSL

- Sugar 1 is the property language of IBM’s RuleBase model checker
- Sugar 1 is CTL plus Sugar Extended Regular Expressions (SEREs)
- SEREs are ITL-like constructs
- Accellera ran a competition to select a ‘standard’ property language

- Finalists were IBM’s Sugar 2 and Motorola’s CBV

  - Intel/Synopsys ForSpec eliminated earlier
    (apparently industry politics involved)

  - Sugar 2 is based on LTL rather than CTL

    - has CTL constructs called “Optional Branching Extension” (OBE)
    - has clocking constructs for temporal abstraction

- Accellera purged “Sugar” from it property language

  - the word “Sugar” was too associated with IBM
  - language renamed to PSL
  - SEREs now Sequential Extended Regular Expressions

- People lobby to make PSL more like ForSpec (align with SVA)
PSL notation

<table>
<thead>
<tr>
<th>Previous notation</th>
<th>PSL ASCII notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \land Q$</td>
<td>$P &amp; Q$</td>
</tr>
<tr>
<td>$P \Rightarrow Q$</td>
<td>$P \rightarrow Q$</td>
</tr>
<tr>
<td>$\neg P$</td>
<td>$\neg P$</td>
</tr>
<tr>
<td>$XP$</td>
<td>next $P$</td>
</tr>
<tr>
<td>$FP$</td>
<td>eventually $P$</td>
</tr>
<tr>
<td>$GP$</td>
<td>always $P$</td>
</tr>
<tr>
<td>$[P \cup Q]!$</td>
<td>$P$ until $Q$</td>
</tr>
<tr>
<td>$[P \wedge Q]!$</td>
<td>$P$ until $Q$</td>
</tr>
</tbody>
</table>

$P$ ranges over states; $w$ ranges over finite lists of states; $r$ ranges over states.

Semantics:
- $\neg P$ is negation
- $XP$ is next $P$
- $FP$ is eventually $P$
- $GP$ is always $P$
- $[P \cup Q]!$ represents $P$ until $Q$
- $[P \wedge Q]!$ represents $P$ until $Q$

SEREs in HOL

Syntax:
- $r := \text{Atom}(p)$ (Atomic formula)
- $r \lor r'$ (Disjunction)
- $r \land r'$ (Conjunction)
- $r_1 \lor r_2$ (Fusion: ITL's chop)
- $r_1 \land r_2$ (Length matching conjunction)
- $r^*$ (Repeat)

Semantics:
- $\text{Atom}(p) = \lambda \sigma. p(\text{head } \sigma)$
- $r_1 \lor r_2 = \lambda \sigma. r_1 \sigma \lor r_2 \sigma$
- $r_1 \land r_2 = \lambda \sigma. \exists \sigma'. \exists \sigma''. \sigma \equiv \sigma' \land r_1 \sigma \land r_2 \sigma''$
- $r_1 \lor r_2 = \lambda \sigma. \exists \sigma'. \exists \sigma''. \sigma \equiv \sigma' \land r_1 \sigma \land r_2 \sigma''$
- $r_1 \land r_2 = \lambda \sigma. r_1 \sigma \land r_2 \sigma$
- $r^* = \lambda \sigma. w = \emptyset \lor \exists \sigma'. \exists \sigma''. \sigma \equiv \sigma'. \sigma'' \land r^* \sigma''$

PSL Foundation Language (FL)

Syntax:
- $f := \text{Atom}(p)$ (Atomic formula)
- $\neg f$ (Negation)
- $f_1 \lor f_2$ (Disjunction)
- $\text{next } f$ (successor)
- $\{r \} f$ (Suffix implication)
- $\{ r_1 \} f \Rightarrow \{ r_2 \}$ (Suffix next implication)
- $[f \text{ until } f']$ (Unit)

Semantics:
- $\text{Atom}(p) = \lambda \sigma. p(\sigma(0))$
- $\neg f = \lambda \sigma. \neg(f \sigma)$
- $f_1 \lor f_2 = \lambda \sigma. f_1 \sigma \lor f_2 \sigma$
- $\text{next } f = \lambda \sigma. f(\text{Tail } i(\sigma))$
- $\{r \} f = \lambda \sigma. \exists \sigma'. \exists \sigma''. \sigma = \sigma' \land r \sigma \land f \sigma''$
- $\{ r_1 \} f \Rightarrow \{ r_2 \} = \lambda \sigma. \exists \sigma'. \exists \sigma''. \sigma = \sigma' \land r_1 \sigma \Rightarrow \exists \sigma''. \sigma'' = \sigma'' \land r_2 \sigma''$
- $[f \text{ until } f'] = \lambda \sigma. \exists \sigma'. f_1(\text{Tail } i(\sigma)) \land \forall j. j < i \Rightarrow f_j(\text{Tail } j(\sigma))$

There is also an Optional Branching Extension (OBE)
- completely standard CTL: EX, EF[U], EG etc.
**Combining SEREs with LTL formulas**

- Formula $[r]f$ means LTL formula $f$ true after SERE $r$
- Example
  After a sequence in which $\text{req}$ is asserted, followed four cycles later by an assertion of $\text{grant}$, followed by a cycle in which $\text{abortin}$ is not asserted, we expect to see an assertion of $\text{ack}$ some time in the future.
- Can represent by
  
  $$\text{always } (\text{req;}[\times 3];\text{grant};!\text{abortin})(\text{eventually}! \text{ack})$$

**Examples of defined notations: consecutive repetition**

- Define
  
  \[
  \begin{align*}
  r[+] &= \{ r;r[+] \} \\
  r[1] &= \begin{cases} \text{false[1]} & \text{if } i=0 \\ r;r;\ldots;r & \text{otherwise (i repetitions of } r) \end{cases} \\
  r[1..j] &= \{ r[1] \} \cup \{ r[1+i] \} \cup \ldots \cup \{ r[j] \} \\
  [=] &= \text{true[=]} \\
  [>] &= \text{true[>]}
  \end{align*}
  \]

  Example
  Whenever we have a sequence of $\text{req}$ followed by $\text{ack}$, we should see a full transaction starting the following cycle. A full transaction starts with an assertion of the signal $\text{start}\_\text{trans}$, followed by one to eight consecutive data transfers, followed by the assertion of signal $\text{end}\_\text{trans}$. A data transfer is indicated by the assertion of signal $\text{data}$.

  \[
  \text{always } (\text{req};\text{ack}) \Rightarrow \{ \text{start}\_\text{trans};\{!\text{data}[1];\text{data}[2];\ldots;\text{data}[8]\};\text{end}\_\text{trans} \}
  \]

**Fixed number of non-consecutive repetitions**

- Example
  Whenever we have a sequence of $\text{req}$ followed by $\text{ack}$, we should see a full transaction starting the following cycle. A full transaction starts with an assertion of the signal $\text{start}\_\text{trans}$, followed by one to eight not necessarily consecutive data transfers, followed by the assertion of signal $\text{end}\_\text{trans}$. A data transfer is indicated by the assertion of signal $\text{data}$.

  \[
  \text{can represent by}
  \text{always } (\text{req};\text{ack}) \Rightarrow \{ \text{start}\_\text{trans};\{!\text{data}[1];\text{data}[2];\ldots;\text{data}[8]\};\text{end}\_\text{trans} \}
  \]

- Define
  \[
  b[=1] = \{ !b[1];b[1];!b[1] \}
  \]

  Then have a nicer representation

  \[
  \text{always } (\text{req};\text{ack}) \Rightarrow \{ \text{start}\_\text{trans};\text{data}[=8];\text{end}\_\text{trans} \}
  \]
Variable number of non-consecutive repetitions

- Example
  Whenever we have a sequence of `req` followed by `ack`, we should see a full transaction starting the following cycle. A full transaction starts with an assertion of the signal `start_trans`, followed by one to eight not necessarily consecutive data transfers, followed by the assertion of signal `end_trans`. A data transfer is indicated by the assertion of signal `data`.

- Define
  \[
  b[= i..j] = \{b[= i]\} \mid \{b[= (i+1)]\} \mid \ldots \mid \{b[= j]\}
  \]

- Then
  \[
  \text{always}(\text{req;}\text{ack}) \implies \{\text{start_trans;}\text{data}[= 1..8];\text{end_trans}\}
  \]

- These examples are meant to illustrate how PSL/Sugar is much more readable than raw CTL or LTL.

Clocking

- Basic idea: `b@clk` abstracts `b` on rising edges of `clk`
- Can clock SEREs (`r@clk`) and formulas (`/\@clk`)
- Can have several clocks
- Official semantics messy due to clocking
- Can ‘translate away’ clocks by pushing `@clk` inwards
  - rules given in PSL manual
  - roughly: `b@clk` --- `{<clk>(r),clk & b}
- Same idea as temporal abstraction: `b` at `clk`

Model checking PSL

- SEREs checked by generating a finite automaton
  - recall: regular expressions can be recognised by finite automata
  - these automata are called “satellites”
- FL checked using standard LTL methods
- OBE checked by standard CTL methods
- Can also check formula for runs of a simulator
  - this is dynamic verification
  - semantics handles possibility of finite paths - messy!

PSL layer structure

- **Boolean layer** has atomic predicates
- **Temporal layer** has LTL (FL) and CTL (OBE) properties
- **Verification layer** has commands for how to use properties
  - e.g. assert, assume
    \[
    \text{assert always} \quad (!en1 \& en2))
    \]
  - \-------------- temporal layer
  - \-------------- verification layer
- **Modelling layer** has HDL constructs for specifying inputs and auxiliary hardware
PSL/Sugar summary

- Combines together LTL, ITL and CTL
- Regular expressions – SEREs
- LTL – Foundation Language formulas
- CTL – Optional Branching Extension
- Relatively simple set of primitives + definitional extension
- Boolean, temporal, verification, modelling layers
- Semantics for static and dynamic verification (needs strong/weak distinction)

New Topic: Simulation or Event semantics

- HDLs use discrete event simulation
  - changes to variables ⇒ threads enabled
  - enabled threads executed non-deterministically
  - execution of threads ⇒ more events
- Combinational thread:
  \[ \text{always } @ (v_1 \text{ or } \cdots \text{ or } v_n) \ v := E \]
  - enabled by any change to \( v_1, \ldots, v_n \)
- Positive edge triggered sequential threads:
  \[ \text{always } @ (\text{posedge } clk) \ v := E \]
  - enabled by \( clk \) changing to \( T \)
- Negative edge triggered sequential threads:
  \[ \text{always } @ (\text{nedge } clk) \ v := E \]
  - enabled by \( clk \) changing to \( F \)

Simulation

- Given
  - a set of threads
  - initial values for variables read or written by threads
  - a sequence of input values
  - (inputs are variables not in LHS of assignments)
- simulation algorithm ⇒ a sequence of states

- Simulation is non-deterministic

Combinational threads in series

\[ \text{in} \xrightarrow{\delta} l_1 \xrightarrow{\delta} l_2 \xrightarrow{\delta} l_3 \xrightarrow{\delta} \text{out} \]

- HDL-like specification:
  \[
  \begin{align*}
  \text{always } & @ (\text{in}) \ l_1 := f(\text{in}) \ \ldots \ \ldots \ \text{thread T1} \\
  \text{always } & @ (l_1) \ l_2 := g(l_1) \ \ldots \ \ldots \ \text{thread T2} \\
  \text{always } & @ (l_2) \ \text{out} := h(l_2) \ \ldots \ \ldots \ \text{thread T3}
  \end{align*}
\]

- Suppose \textit{in} changes to \( v \) at simulation time \( t \)
  - T1 will become enabled and assign \( f(v) \) to \( l_1 \)
  - \( l_1 \)'s value changes then T2 will become enabled
    (still simulation time \( t \))
  - T2 will assign \( g(f(v)) \) to \( l_2 \)
  - \( l_2 \)'s value changes then T will become enabled
    (still simulation time \( t \))
  - T3 will assign \( h(g(f(v))) \) to \( \text{out} \)
  - simulation quiesces
    (still simulation time \( t \))

- Steps at same simulation time happen in \( \delta \)-time
  (VHDL jargon)
Semantic gap
- Designers use HDLs and verify via simulation
- Formal verifiers use logic and verify via proof
  - trace semantics
- Problem: show consistency between semantics
- Goal:
  \[
  \text{traces} = \text{sequences of quiescent simulation states}
  \]

Outline (see Section 4.4 of Notes for details):
- first analyse sets of combinational threads
- identify conditions for “non-looping”
- simulation terminates \(\Rightarrow\) trace semantics \((\text{partial correctness})\)
- simulation always terminates “quiesces” \((\text{total correctness})\)
- extend to sequential threads

Trace defined by a simulation run
- Simulation defines a tree of states

Sequential threads – event semantics
- Consider two Dtypes in series:
  - \(\text{always @} (\text{posedge } \text{clk}) \ l := \text{in}\)
  - \(\text{always @} (\text{posedge } \text{clk}) \ \text{out} := \ l\)
  - If \(\text{posedge } \text{clk}\):
    - both threads become enabled
    - race condition
  - Right thread executed first:
    - \(\text{out}\) gets previous value of \(\ l\)
    - then left thread executed
    - \(\text{so } \ l\) gets value input at \(\ \text{in}\)
  - Left thread executed first:
    - \(\ l\) gets input value at \(\ \text{in}\)
    - then right thread executed
    - \(\text{in } \text{out}\) gets input value at \(\ \text{in}\)

Sequential threads – trace semantics
- Trace semantics:
  \[
  \begin{align*}
  \langle \forall t. \ l(t+1) \rangle &= \langle \text{Rise } \text{clk} \ t \rightarrow \text{in } t \ | \ l(t) \rangle \wedge \\
  \langle \forall t. \ \text{out}(t+1) \rangle &= \langle \text{Rise } \text{clk} \ t \rightarrow \ l(t) \ | \ \text{out}(t) \rangle
  \end{align*}
  \]
  - Corresponds to right thread executed first
- How to ensure event and trace semantics agree?
  - Method 1: use non-blocking assignments:
    - \(\text{always @} (\text{posedge } \text{clk}) \ l <= \text{in};\)
    - \(\text{always @} (\text{posedge } \text{clk}) \ \text{out} <= \ l;\)
    - \(\text{non-blocking assignments (<=)}\) in Verilog
      - RHS of all non-blocking assignments first computed
      - assignments done at end of simulation cycle
  - Method 2: make simulation cycle VHDL-like
Verilog versus VHDL simulation cycles

- Verilog-like simulation cycle:
  - Choose an enabled thread
  - Execute the chosen thread
  - Fire event controls to enable new threads
  - Execute until quiescent
  - Then advance simulation time

- VHDL-like simulation cycle:
  - Execute all enabled threads in parallel
  - Fire event controls to enable new threads
  - Execute until quiescent
  - Then advance simulation time

VHDL event semantics

- Recall HDL:
  ```
  always @(posedge clk) l := in
  always @(posedge clk) out := l
  ```
- If posedge `clk`:
  - both threads become enabled
- VHDL semantics:
  - both threads executed in parallel
  - `out` gets previous value of `l`
  - `in` parallel `l` gets value input at `in`
- Now no race
- Event semantics matches trace semantics

Summary of dynamic versus static semantics

- Simulation (event) semantics different from trace semantics
- No standard event semantics (Verilog versus VHDL)
- Verilog: need non-blocking assignments
- VHDL semantics closer trace semantics

Summary of Specification I and II

- Software specification and verification
  - Hoare logic: partial and total correctness
  - proof by invariants and variants
  - mechanisation via VCs (WP or SP)
  - only nice for simple languages
  - can apply Hoare logic to behavioral view of hardware
- Higher order logic (HOL)
  - unifying general logic
  - supports Hoare logic via embedding
  - supports temporal logics via embedding
  - can directly represent hardware behavior and structure (3, 8)
  - hardware verification as pure logic proof
  - relating models: event vs trace vs RTL vs cycles
- Hardware specification and verification
  - automatic FV uses state machine models, fit nicely into HOL
  - reachable states calculated by iteration (fixed point)
  - symbolic representations: BDDs
  - model checking of properties (CTL, LTL, ITL, PSL)
  - event simulation used in industry

THE END - HAVE A GOOD VACATION!