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

Rising edges

Notes are confused!

- 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:
  \[
  \forall t, t > 0 \Rightarrow (\text{Rise}_1(f)(t) = \text{Rise}_1(f)(t-1))
  \]
  \[
  \forall t, t \geq 0 \Rightarrow (\text{Rise}_2(f)(t) = \text{Rise}_2(f)(t+1))
  \]

- In Accellera standard language PSL function \( \text{Rise}_1 \) is called \( \text{Rise} \)

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 \( t_1 \) 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.

Some temporal operators in Higher Order Logic

- Define:
  \[
  \text{Next}(t_1, t_2)(f) = t_1 < t_2 \land f(t_2) \land \forall t. t_1 < t \land t < t_2 \Rightarrow \neg f(t)
  \]

- 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:**
  \[
  \text{Dtype}(c_1, c_2, c_3, c_4)(d, ck, q) \equiv \\
  \forall t_1, t_2. \text{Rise}_1(ck)(t_1) \land \\
  \text{Next}(t_1, t_2)(\text{Rise}_1(ck)) \land \\
  (t_2 - t_1 > c_2) \land \\
  \text{Stable}(t_1 - c_1, t_1 + 1)(d) \\
  \Rightarrow \\
  (\text{Stable}(t_1 + c_3, t_2 + c_4)(q) \land (q(t_2) = d(t_1)))
  \]

- **Note that**
  \[
  \text{Next}(t_1, t_2)(\text{Rise}_1(ck))
  \]
  formed by applying \[
  \text{Next}(t_1, t_2)
  \]
  to the predicate \[
  \text{Rise}_1(ck)
  \]

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

- **Note that**
  \[
  \text{Next}(t_1, t_2)(\text{Rise}_1(ck))
  \]

**Implementation**

- Can implement Dtype using NAND-gates:

**Verification**

- Dtype implementation in logic:

  \[
  \text{Dtype}_{\text{Imp}}(d, ck, q) \equiv \\
  \exists p_1, p_2, p_3, p_4, p_5. \\
  \text{NAND}(p_2, d, p_1) \land \\
  \text{NAND}_3(p_3, ck, p_1, p_2) \land \\
  \text{NAND}(p_4, ck, p_3) \land \\
  \text{NAND}(p_1, p_3, p_4) \land \\
  \text{NAND}(p_3, p_5, q) \land \\
  \text{NAND}_2(q, p_2, p_5)
  \]

- Correctness: find \(d_1, d_2, d_3, d_4\) and \(d_5\) and prove:

  \[
  \text{Dtype}_{\text{Imp}}(d, ck, q) \Rightarrow \text{Dtype}(d_1, d_2, d_3, d_4)(d, ck, q)
  \]

  **Hard!**

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

**A sequential RT level example: simple parity checker**

- **Input** \(\text{inp}\), an output \(\text{out}\)
- **The 0th output is T \iff an even number of T’s input**
- **PARITY f n iff an even number of T’s in f(1), …, f(n)**

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

- **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:**

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

  **Intuition can be verified by proving:**

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

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

------------------------------------

- First step is to consider arbitrary \text{inp\ and\ out}\ and\ then\ to\ assume
the antecedents of the implication and try to prove the conclusion

∀\ 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
  |- (\forall\ \text{f. PARITY}\ \text{f}\ 0 = T)
  \forall\ f. \text{PARITY}\ f\ (n+1) = \text{if}\ f(n+1)\ \text{then}\ \neg\ \text{PARITY}\ f\ n\ \text{else}\ \text{PARITY}\ f\ n

- Unfolding (rewriting with) the definition of PARITY in

∀\ t. \text{out}(t+1) = \text{if}\ \text{inp}(t+1)\ \text{then}\ \neg\ (\text{out}\ t)\ \text{else}\ \text{out}\ t

Unfolding (rewriting with) the definition of PARITY in

out 0 = PARITY \text{inp} 0

------------------------------------

[the basis of the induction]

0. out 0 = T
1. ∀t. out (t+1) = if inp (t+1) then ¬(out t) else out t

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

------------------------------------

[the step of the induction]

0. out 0 = T
1. ∀t. out (t+1) = if inp (t+1) then ¬(out t) else out t
2. out t = PARITY \text{inp} t

Yields

∀\ t. \text{out}\ t = \text{PARITY}\ \text{inp}\ t

Goal now easily proved

- Proof state from last slide

out 0 = T

------------------------------------

0. out 0 = T
1. ∀t. out (t+1) = if inp (t+1) then ¬(out t) else out t

out (t+1) = if inp (t+1) then ¬(PARITY \text{inp} t) else PARITY \text{inp} t

------------------------------------

0. out 0 = T
1. ∀t. out (t+1) = if inp (t+1) then ¬(out t) else out t
2. out t = PARITY \text{inp} t

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

UNIQUENESS_LEMMA =
|- \text{Vinp out.}
(out 0 = T) \land
(\forall\ t. \text{out}(t+1) = \text{if}\ \text{inp}(t+1)\ \text{then}\ \neg\ (\text{out}\ t)\ \text{else}\ \text{out}\ t) \Rightarrow
\forall\ t. \text{out}\ t = \text{PARITY}\ \text{inp}\ t

A Proof by induction

- Start with the following proof state

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

------------------------------------

- As on previous slide, consider arbitrary \text{inp\ and\ out}\ and\ then\ to
assume the antecedents of the implication

∀\ t. \text{out}\ t = \text{PARITY}\ \text{inp}\ t

------------------------------------

0. out 0 = T
1. ∀t. out (t+1) = if inp (t+1) then ¬(out t) else out t

Now do induction on t – this creates a proof state with two subgoals

out 0 = PARITY \text{inp} 0

------------------------------------

[the basis of the induction]

0. out 0 = T
1. ∀t. out (t+1) = if inp (t+1) then ¬(out t) else out t

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

------------------------------------

[the step of the induction]

0. out 0 = T
1. ∀t. out (t+1) = if inp (t+1) then ¬(out t) else out t
2. out t = PARITY \text{inp} t

[induction hypothesis added to assumptions]
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

\[
\begin{align*}
\text{ONE out} &= \forall t. \text{out } t = T \\
\text{NOT}(\text{inp}, \text{out}) &= \forall t. \text{out } t = \neg (\text{inp } t) \\
\text{MUX}(\text{sw}, \text{in}1, \text{in}2, \text{out}) &= \forall t. \text{out } t = \text{if } \text{sw } t \text{ then } \text{in}1 t \text{ else } \text{in}2 t \\
\text{REG}(\text{inp}, \text{out}) &= \forall t. \text{out } t = \text{if } (t=0) \text{ then } F \text{ else } \text{inp}(t-1)
\end{align*}
\]

Verification

- The following theorem will eventually be proved:
  \[
  \forall \text{inp out. PARITY_IMP(inp, out)} \Rightarrow \forall t. \text{out } t = \text{PARITY inp } t
  \]
- First prove a lemma (then theorem follows from \text{UNIQUENESS_LEMA})
- The lemma (\text{PARITY_LEMA}):
Proof continued

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

\[
\begin{align*}
0 & \quad \forall t. l_1 t = \neg l_2 t \\
1 & \quad \forall t. l_3 t = \text{if} \ inp t \ \text{then} \ l_1 t \ \text{else} \ l_2 t \\
2 & \quad \forall t. l_2 t = \text{if} \ t = 0 \ \text{then} \ F \ \text{else} \ out(t-1) \\
3 & \quad \forall t. l_4 t = T \\
4 & \quad \forall t. l_5 t = \text{if} \ t = 0 \ \text{then} \ F \ \text{else} \ l_4(t-1) \\
5 & \quad \forall t. out t = \text{if} \ l_5 t \ \text{then} \ l_3 t \ \text{else} \ l_4 t
\end{align*}
\]

Easily follows (see stuff in blue)

Now consider \( t+1 \) case

\[
\begin{align*}
\text{out}(t+1) & = \text{if} \ inp(t+1) \ \text{then} \ \neg (\text{out} t) \ \text{else} \ \text{out} t \\
0 & \quad \forall t. l_1 t = \neg l_2 t \\
1 & \quad \forall t. l_3 t = \text{if} \ inp t \ \text{then} \ l_1 t \ \text{else} \ l_2 t \\
2 & \quad \forall t. l_2 t = \text{if} \ t = 0 \ \text{then} \ F \ \text{else} \ out(t-1) \\
3 & \quad \forall t. l_4 t = T \\
4 & \quad \forall t. l_5 t = \text{if} \ t = 0 \ \text{then} \ F \ \text{else} \ l_4(t-1) \\
5 & \quad \forall t. out t = \text{if} \ l_5 t \ \text{then} \ l_3 t \ \text{else} \ l_4 t
\end{align*}
\]

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

\[
\forall t. out t = \text{if} \ l_5 t \ \text{then} \ l_3 t \ \text{else} \ l_4 t
\]

Goal follows from assumptions with a bit of calculation

Combining lemmas

- Call lemma just proved PARITY_LEMMA, so

\[
\text{PARITY_LEMMA} =
\]

\[
\vdash \text{Vinp. out} \\
\text{PARITY_IMP (in, out)} \\
\Rightarrow \\
\text{(out \( 0 \) \( = \) \( T \)) \ \wedge} \\
\forall t. \text{out}(t+1) = \text{if} \ \text{inp}(t+1) \ \text{then} \ \neg (\text{out} t) \ \text{else} \ \text{out} t
\]

- Recall

\[
\text{UNIQUENESS_LEMMA} =
\]

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

- Hence by transitivity of \( \Rightarrow \)

\[
\vdash \text{Vinp. out. PARITY_IMP (inp, out)} \Rightarrow \forall t. \text{out} t = \text{PARITY inp} t
\]

- PARITY_IMP used abstract registers REG

- Next: make model more concrete by using clocked Dtype

Review

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

- Equivalent equation between functions: \( \text{out} = \text{PARITY} \ \text{inp} \)
An incorrect implementation of the parity checker

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

- The following implementation doesn’t work

<table>
<thead>
<tr>
<th>inp</th>
<th>l1</th>
<th>l2</th>
<th>out</th>
<th>PARITY inp</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

Temporal refinement

- PARITY_IMP used abstract registers REG
- Next: make model more concrete by using clocked Dtype
- Recall the (course grained) trace level model of a Dtype:

\[ \text{Dtype}_{\text{ck}, \ d, \ q} = \forall t. \ q(t+1) = (\text{Rise}_{\text{ck}} t \rightarrow d t) \mid q t\]

- Need a version of Dtype that powers up storing F

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

Formulating Correctness

- A mapping between time-scales:

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

  \[ \text{Time}_{\text{of}} P n = \text{the concrete time } t \text{ when } P \text{ true for nth time}\]

  \[ s \text{ when } P = s \circ (\text{Time}_{\text{of}} P)\]

- From Melham’s Theorem:

  \[ \forall k. \ \text{Inf}(\text{Rise}_{\text{ck}} k) \rightarrow \forall d. \ \text{Dtype}_{\text{ck}, d, q} = \text{REG}(d \text{ when } (\text{Rise}_{\text{ck}} k), q \text{ when } (\text{Rise}_{\text{ck}} k))\]

- Int P means “P true infinitely often”

  \[ \text{Inf } P = \forall t. \exists t' > t \land P t'\]
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 \( P \text{ true for } n\text{th time} \)
  - 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

- \( \text{ex. } t[x] \) is an epsilon-term
- The meaning of \( \text{ex. } t[x] \) is specified by an axiom:
  \[ \forall P. (\exists t. P t) \Rightarrow P(\text{ex. } P t) \]
- \( \text{ex. } t[x] \) denotes some value, \( v \) say, such that \( P[v] \), if \( \exists t. t[x] \)
- \( \text{ex. } t[x] \) 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
  \[ \text{Next } t1 \ t2 \ \text{sig } = t1 < t2 \land \text{sig } t2 \land \forall t. t < t1 \land t < t2 \Rightarrow \neg \text{(sig } t) \]
- Define \( \text{IsTimeof n sig } t \) to mean “\( t \) is when \( \text{sig} \) is true for the \( n\)-th time”
  \[ (\text{IsTimeof } 0 \text{ sig } t = (\text{sig } t \land \forall t'. t' < t \Rightarrow \neg (\text{sig } t'))) \land (\text{IsTimeof } (n+1) \text{ sig } t = \exists t'. \text{IsTimeof } n \text{ sig } t' \land \text{Next } t' t \text{ sig}) \]
- Define \( \text{Timeof} \) using \( \epsilon \)-operator and \( \text{IsTimeof} \)
  \[ \text{Timeof } \text{sig } n = t. \text{IsTimeof } n \text{ sig } t \]
- \( \text{IsTimeof} \) and \( \text{Timeof} \) are higher-order total functions

Temporal abstraction

- Define \( f@ck \) to be signal \( f \) abstracted on rising edges of \( ck \)
  \[ |- f@ck = f \text{ when } (\text{Rise } ck) \]
- Recall definition of \( \text{REG} \)
  \[ |- \text{REG}(\text{inp, out}) = \forall t. \text{out } t = \text{if } (t=0) \text{ then } F \text{ else } \text{inp}(t-1) \]
- It follows easily that
  \[ |- \text{REG}(\text{inp, out}) = (\text{out } 0 = F) \land \text{Del}(\text{inp, out}) \]
- The properties below also follow (why?)
  \[ |- \text{Inf}(\text{Rise } ck) \Rightarrow \text{DtypeF(ck,d,q)} \Rightarrow \text{REG}(d@ck, q@ck) \]
  \[ |- \text{MUX}(\text{switch, i1, i2, out}) \Rightarrow \text{MUX}(\text{switch@ck, i1@ck, i2@ck, out@ck}) \]
  \[ |- \text{NOT}(\text{inp, out}) \Rightarrow \text{NOT}(\text{inp@ck, out@ck}) \]
  \[ |- \text{ONE out} \Rightarrow \text{ONE(out@ck}) \]
- Hint: \( \vdash \forall t. (\forall x. P(x)) \Rightarrow (\forall x. P(f(x))) \) take \( f = x \mapsto i@ck \)
Cycle and trace versions

- **Compare**

  |- PARITY_IMP(inp,out) =
    | | l1 l2 l3 l4 l5.
    | | NOT(l2,l1) ∧ MUX(inp,l1,l2,l3) ∧ REG(out,l2) ∧
    | | ONE l4 ∧ REG(l4,l5) ∧ MUX(l5,l3,l4,out)

  |- DtypePARITY_IMP(ck,inp,out) =
    | | l1 l2 l3 l4 l5.
    | | NOT(l2,l1) ∧ MUX(inp,l1,l2,l3) ∧ DtypeF(ck,out,l2) ∧
    | | ONE l4 ∧ DtypeF(ck,l4,l5) ∧ MUX(l5,l3,l4,out)

- **Hence by implications on previous slide**

  |- Inf(Rise ck) ⇒ DtypePARITY_IMP(ck,inp,out) ⇒ PARITY_IMP(inp@ck, out@ck)

  |- ∀ inp out. PARITY_IMP(inp,out) ⇒ ∀ t. out t = PARITY inp t

- Specialising inp to inp@ck and out to out@ck

  |- PARITY_IMP(inp@ck, out@ck) ⇒ ∀ t. (out@ck) t = PARITY (inp@ck) t

- **From previous slide**

  |- Inf(Rise ck) ⇒ DtypePARITY_IMP(ck,inp,out) ⇒ PARITY_IMP(inp@ck, out@ck)

  |- ∀ t. (out@ck) t = PARITY (inp@ck) t

- **Hence, by transitivity of ⇒**

  |- Inf(Rise ck) ⇒ DtypePARITY_IMP(ck,inp,out) ⇒ ∀ t. (out@ck) t = PARITY (inp@ck) t

- This is a typical correctness result using temporal abstraction

NEW TOPIC: modelling transistors

- **Recall simple switch model of CMOS**

  
  ```
  g s d
  ⊢ Ptran(g,s,d) = (¬g ⇒ (d = s))
  
  g s d
  ⊢ Ntran(g,s,d) = (g ⇒ (d = s))
  
  ⊢ Gnd g = (g = F)
  
  ⊢ Pwr p = (p = T)
  ``

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

Trace level verification

- **Proved earlier**

  |- ∀ inp out. PARITY_IMP(inp,out) ⇒ ∀ t. out t = PARITY inp t

- **Specialising inp to inp@ck and out to out@ck**

  |- PARITY_IMP(inp@ck, out@ck) ⇒ ∀ t. (out@ck) t = PARITY (inp@ck) t

- **From previous slide**

  |- Inf(Rise ck) ⇒ DtypePARITY_IMP(ck,inp,out) ⇒ PARITY_IMP(inp@ck, out@ck)

  |- ∀ t. (out@ck) t = PARITY (inp@ck) t

- **Hence, by transitivity of ⇒**

  |- Inf(Rise ck) ⇒ DtypePARITY_IMP(ck,inp,out) ⇒ ∀ t. (out@ck) t = PARITY (inp@ck) t

The simple adder example

- **This example shows non-obvious examples can be analysed**

  ```
  Add1(a,b,cin,sum,cout) =
  ∃ p0 p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11.
  Ptran(p1,p0,p2) ∧ Ptran(cin,p0,p3) ∧ Ptran(b,p2,p3) ∧
  Ptran(a,p2,p4) ∧ Ptran(p1,p3,p5) ∧ Ptran(a,p4,p5) ∧
  Ntran(p1,p4,p6) ∧ Ntran(h,p5,p6) ∧ Ntran(p1,p6,p11) ∧
  Ntran(cin,p6,p11) ∧ Ptran(a,p6,p7) ∧ Ptran(b,p6,p7) ∧
  Ptran(a,p8,p9) ∧ Ptran(p1,p7,p10) ∧ Ptran(b,p8,p10) ∧
  Ntran(cin,p1,p10) ∧ Ntran(b,p11,p11) ∧ Ntran(a,p9,p11) ∧
  Ntran(b,p9,p11) ∧ Ntran(a,p10,p11) ∧ Pwr(p0) ∧
  Ptran(p4,p2,sum) ∧ Ntran(p4,sum,p10) ∧ Gnd(p11) ∧
  Ptran(p1,p0,cout) ∧ Ntran(p1,cout,p11)
  ```

  ```
  Add1(a,b,cin,sum,cout) = (2 * Bv cout + Bv sum = Bv a + Bv b + Bv cin)
  ```
Problems with simple switch model

- Compare

![Diagram of simple switch model]

- 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 $V_g - V_b \geq V_{TH}$

  - Only conducts well if $V_a - V_g \geq V_{TH}$ or $V_b - V_g \geq V_{TH}$

- If $V_g = V_a$ there is a voltage drop of about $V_{TH}$

- Example: ‘hi’ is 5v, ‘low’ is 0v

  ![Diagram of voltage levels]

- Weak output may not be able to switch transistors

What happens in the Simple Switch Model

- From the definitions

  1- $\forall p. \ Pwr \ p = (p = T)$
  2- $\forall g \ a \ b. \ Ntran \ (g,a,b) = g \Rightarrow (a = b)$
  3- $\forall out. \ Bad \ out = \exists l1 \ l2. \ Pwr \ l1 \land Ntran \ (l1,l1,l2) \land Ntran \ (l2,l2,\ out)$

- It follows that

  1- $\forall out. \ Bad \ out = \ out$

Consider two Xors when both inputs are $F$

- Compare

![Diagram of Xors]

- 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 Hi, Lo and other values

\[
\text{Ntran}(g,a,b) = ((g=\text{Hi}) \land (a=\text{Lo}) \Rightarrow (b=\text{Lo})) \\
\land ((g=\text{Hi}) \land (b=\text{Lo}) \Rightarrow (a=\text{Lo}))
\]

\[
\text{Ptran}(g,a,b) = ((g=\text{Lo}) \land (a=\text{Hi}) \Rightarrow (b=\text{Hi})) \\
\land ((g=\text{Lo}) \land (b=\text{Hi}) \Rightarrow (a=\text{Hi}))
\]

More compact definitions

\[
\text{Ntran}(g,a,b) = (g=\text{Hi}) \Rightarrow ((a=\text{Lo}) = (b=\text{Lo}))
\]

\[
\text{Ptran}(g,a,b) = (g=\text{Lo}) \Rightarrow ((a=\text{Hi}) = (b=\text{Hi}))
\]

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

Good and bad Xors now distinguished

Earlier examples still work

- Define
  \[- \text{Strong } v = ((v = \text{Hi}) \lor (v = \text{Lo})) \]
  \[- \text{TBv } \text{Hi} = 1 \land \text{TBv } \text{Lo} = 0 \]
  \[- \text{TAdd1Spec}(a,b,cin,sum,cout) = (2(\text{TBv cout} \lor \text{TBv sum} \lor \text{TBv } a \lor \text{TBv } b \lor \text{TBv cin})) \]

- Then it follows that
  \[- \text{Strong } a \land \text{Strong } b \land \text{Strong } cin \]
  \[- \text{AddSpec}(a,b,cin,sum,cout) \Rightarrow \text{TAdd1Spec}(a,b,cin,sum,cout) \]
  \[- \text{Strong } sum \land \text{Strong } cout \]
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 1: \(\text{ph1}=\text{Hi}\) and \(\text{ph2}=\text{Lo}\)

Phase 2: \(\text{ph1}=\text{Lo}\) and \(\text{ph2}=\text{Hi}\)

Phase 3: \(\text{ph1}=\text{Hi}\) and \(\text{ph2}=\text{Lo}\)
Phase 4: \( ph1=\text{Lo} \) and \( ph2=\text{Hi} \)

Characterisation of behaviour

Unidirectional sequential model

Signals are functions of time
Unidirectional sequential transistor models

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

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

Sequential shift register model

\[ \text{ShiftReg}(i, \text{out}, ph1, ph2) = \exists l1, l2, l3, l4, l5, l6, l7, l8, l9, l10, l11, l12, l13. \]
\[ \text{Nswitch}(ph1, i, l1) \land \text{Cap}(l1, l2) \land \text{Pwr} l3 \land \text{Pswitch}(l2, l3, l4) \land \text{Nswitch}(l2, l6, l5) \land \text{Gnd} l6 \land \text{Join}(l4, l5, l7) \land \text{Nswitch}(ph2, l7, l8) \land \text{Cap}(l8, l9) \land \text{Pwr} l10 \land \text{Pswitch}(l9, l10, l11) \land \text{Nswitch}(l9, l13, l12) \land \text{Gnd} l13 \land \text{Join}(l11, l12, \text{out}) \]

- Lots more state variables than in combinational switch model!

Correctness of sequential shift register model

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

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

A model of NMOS

- Need a new component: pullup

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

\[
\text{Inv}(i, out) = \\
\text{Cap}(i, 11) \land \text{Gnd}(12) \land \neg \text{Switch}(11, 12, 13) \land \text{Pu}(13, out)
\]

\[
\begin{align*}
\text{Inv}(i, out) & \Rightarrow (i_t = \text{Hi}) \Rightarrow (out_{t+1} = \text{Lo}) \\
& \Rightarrow ((i_{t+1} = \text{Hi}) \Rightarrow (out_{t+1} = \text{Lo}) \\
& \quad \lor ((i_{t+1} = \text{Lo}) \Rightarrow (out_{t+1} = \text{Hi}))
\end{align*}
\]

Four phase NMOS shift register

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

\[
\begin{align*}
& \land \text{Strong}(i_{t+1}) \\
& \land (ph1_t = \text{Hi}) \land (ph2_t = \text{Lo}) \land (ph3_t = \text{Lo}) \land (ph4_t = \text{Lo}) \\
& \land (ph1(t+1) = \text{Lo}) \land (ph2(t+1) = \text{Hi}) \land (ph3(t+1) = \text{Lo}) \land (ph4(t+1) = \text{Lo})
\end{align*}
\]

Phase 1 (precharge internal node)

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

\[
\begin{align*}
& (\text{phi}_t = \text{Hi}) \land (\text{phi}_{t+1} = \text{Lo}) \land (\text{phi}_{t+2} = \text{Lo}) \land (\text{phi}_{t+3} = \text{Lo})
\end{align*}
\]

Phase 2 (input Lo, retain precharge)

Colour scheme: Hi, Lo, Fl and dotted means precharge

\[
\begin{align*}
& (\text{phi}_{t+1} = \text{Lo}) \land (\text{phi}_{t+2} = \text{Hi}) \land (\text{phi}_{t+3} = \text{Lo}) \land (\text{phi}_{t+4} = \text{Lo})
\end{align*}
\]

\[
(i + 1) = \text{Lo}
\]
Phase 1 (precharge internal node)

\[(\text{ph1}(t) = \text{Hi}) \land (\text{ph2}(t) = \text{Lo}) \land (\text{ph3}(t) = \text{Lo}) \land (\text{ph4}(t) = \text{Lo})\]

- out retains previous value

Phase 2 (input Hi, kill precharge)

\[(\text{ph1}(t+1) = \text{Lo}) \land (\text{ph2}(t+1) = \text{Hi}) \land (\text{ph3}(t+1) = \text{Lo}) \land (\text{ph4}(t+1) = \text{Lo})\]

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

Phase 3 (precharge out, internal node retains value)

\[(\text{ph1}(t+2) = \text{Lo}) \land (\text{ph2}(t+2) = \text{Lo}) \land (\text{ph3}(t+2) = \text{Hi}) \land (\text{ph4}(t+2) = \text{Lo})\]

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

Phase 4 (kill precharge)

\[(\text{ph1}(t+3) = \text{Lo}) \land (\text{ph2}(t+3) = \text{Lo}) \land (\text{ph3}(t+3) = \text{Lo}) \land (\text{ph4}(t+3) = \text{Hi})\]

\[(\text{out}(t+3) = \text{Lo})\]
Phase 3 (precharge out, internal node retains value)

\[(\text{ph1}(t+2)=\text{Lo}) \land (\text{ph2}(t+2)=\text{Lo}) \land (\text{ph3}(t+2)=\text{Hi}) \land (\text{ph4}(t+2)=\text{Lo})\]

Phase 4 (out retains precharge)

\[(\text{ph1}(t+3)=\text{Lo}) \land (\text{ph2}(t+3)=\text{Lo}) \land (\text{ph3}(t+3)=\text{Lo}) \land (\text{ph4}(t+3)=\text{Hi})\]

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

Four phase NMOS shift register model

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

\[\exists l1 \ l2 \ l3 \ l4 \ l5 \ l6 \ l7 \ l8 \ l9 \ l10 \ l11. \]

\[\text{Nswitch(ph1,ph1,l1)} \land \text{Nswitch(i,l3,l2)} \land \text{Nswitch(ph2,ph1,l3)} \land \text{Join(l1,l2,l4)} \land \text{Cap(l4,l5)} \land \text{Cap(l5,l6)} \land \text{Cap(l6,l7)} \land \text{Nswitch(ph3,ph3,l8)} \land \text{Nswitch(l7,l10,l9)} \land \text{Nswitch(ph4,ph3,l10)} \land \text{Join(l8,l9,l11)} \land \text{Cap(l11,out)}\]

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

\[\wedge \text{Strong(i}(t+1))\]

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

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

\[\wedge \text{Strong(i}(t+1))\]

\[\wedge (\text{ph1}(t \ =\text{Hi}) \land (\text{ph2}(t \ =\text{Lo}) \land (\text{ph3}(t \ =\text{Lo}) \land (\text{ph4}(t \ =\text{Lo}) \land (\text{ph1}(t+1)=\text{Lo}) \land (\text{ph2}(t+1)=\text{Hi}) \land (\text{ph3}(t+1)=\text{Lo}) \land (\text{ph4}(t+1)=\text{Lo}) \land (\text{ph1}(t+2)=\text{Lo}) \land (\text{ph2}(t+2)=\text{Lo}) \land (\text{ph3}(t+2)=\text{Hi}) \land (\text{ph4}(t+2)=\text{Lo}) \land (\text{ph1}(t+3)=\text{Lo}) \land (\text{ph2}(t+3)=\text{Lo}) \land (\text{ph3}(t+3)=\text{Lo}) \land (\text{ph4}(t+3)=\text{Hi}) \Rightarrow (\text{out}(t+3) = \text{in}(t+1))\]
An earlier slide on Hoare logic for hardware

- Would like a generalised Hoare Logic specification:

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

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 \implies bugs</td>
<td>proof success \implies assurance</td>
</tr>
<tr>
<td>practical for real code</td>
<td>expensive and often impractical</td>
</tr>
<tr>
<td>unsound models OK</td>
<td>create 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 \textit{states}
- Set of initial states: predicate \( B \)
  - \( B \) \( s \) means \( s \) is an initial state
- State transition relation: \( R \)
  - \( R \) : \( \text{states} \times \text{states} \rightarrow \text{bool} \)
  - \( R(s, s') \) means \( s' \) a successor to \( s \)

\[ R \] defines a \textbf{branching time} model

Example: single state machine

- State transition function: \( \delta \)
  \( \delta : \text{states} \times \text{inputs} \rightarrow \text{states} \)
- 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
  \( \delta_i : \text{states} \times \text{inputs} \rightarrow \text{states} \quad (1 \leq i \leq n) \)
  - Each machine \( \delta_i \) reads all inputs and states
- An \( R \)-step is a non-deterministically chosen step of one machine
  \[ R(\vec{v}, \vec{v'}) = \exists \text{inp. } \begin{align*}
      v'_1 &= \delta_1(\vec{v}, \text{inp}) \\
      v'_2 &= \delta_2(\vec{v}, \text{inp}) \\
      &\vDash \cdots \\
      v'_n &= \delta_n(\vec{v}, \text{inp})
  \end{align*} \]
  \( v'_1 = v_1 \land v'_2 = v_2 \land \cdots \land v'_n = v_n \)
- 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 \( S_0 = \{ s \mid B \} \)
  - Iteratively compute with \( S_{n+1} = S_n \cup \{ s \mid \exists u. u \in S_n \land R(u,s) \} \)
  - Note \( S_0 \subseteq S_1 \subseteq S_2 \subseteq \cdots \)
    - if finite number of states then eventually reach an \( n \) such that \( S_n = S_{n+1} \)
    - so \( S_n \) is set of reachable states
- Now check \( P(s) \) for every reachable \( s \) (i.e. for every \( s \in S_n \))

Symbolic approach: representing sets as formulas

- Set \( \{ b_1, b_2, \ldots, b_n \} \) represented by formula \( v = b_1 \lor b_2 \lor \cdots \lor b_n \)
  - \( b_1, b_2, \ldots, b_n \) are truth-values (i.e. \( T \) or \( F \))
  - \( v \) is a boolean variable
  - \( b \in \{ b_1, b_2, \ldots, b_n \} \) if and only if \( \vdash (v = b_1 \lor b_2 \lor \cdots \lor b_n)[b/v] \)
- A set of states
  \[ \{(b_1, \ldots, b_m), (b_1, \ldots, b_m)\} \]
  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 \( (dreq', q0', dack') \) represent 'next state'
- Transition relation is:
  \[ (q0' = dreq) \land (dack' = dreq \land (q0 \lor (\neg q0 \land dack))) \]
- Transition relation equivalent to:
  \[ (q0' = dreq) \land (dack' = dreq \land (q0 \lor dack)) \]
- Define \( R_{\text{receive}} \) by:
  \[ R_{\text{receive}}((dreq, q0, dack), (dreq', q0', dack')) = (q0' = dreq) \land (dack' = dreq \land (q0 \lor dack)) \]
- \( dreq' \) unconstrained, hence non-determinism

Symbolic reachability: sets of states are formulas

- Condition for a state \( s \) to be reachable
  in one \( R \)-step from a state in \( B \)
  \[ \exists u. B \cup R(u, s) \]
- Define \( \text{ReachBy} \) \( n. R \) \( B \) to be set of states reachable in at most \( n \) steps:
  \[ \text{ReachBy}\ 0. R \ B \ s = B \ s \]
  \[ \vdash \text{ReachBy} (n+1). R \ B \ s = \text{ReachBy} n. R \ B s \land \exists u. \text{ReachBy} n. R \ B u \land R(u, s) \]
- Reachable states are states reachable in a finite number of steps:
  \[ \vdash \text{Reach} R \ B \ s = \exists n. \text{ReachBy} n. R \ B s \]
- Key property (equality between predicates represents set equality):
  \[ \vdash (\text{ReachBy} n. R \ B = \text{ReachBy} (n+1). R \ B) \]
  \[ \wedge (\text{Reach} R = \text{ReachBy} n. R \ B) \]
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
- Example: BDDs of \( v \) and \( \neg v \)

\[
\begin{array}{c}
0 \quad 1 \\
0 \quad 1 \\
0 \quad 1 \\
\end{array}
\]

\[
\begin{array}{c}
0 \quad 1 \\
0 \quad 1 \\
0 \quad 1 \\
\end{array}
\]

- Example: BDDs of \( v_1 \land v_2 \) and \( v_1 \lor v_2 \)

\[
\begin{array}{c}
0 \quad 1 \\
0 \quad 1 \\
0 \quad 1 \\
\end{array}
\]

\[
\begin{array}{c}
0 \quad 1 \\
0 \quad 1 \\
0 \quad 1 \\
\end{array}
\]

More BDD examples

- BDD of \( v_1 = v_2 \)

\[
\begin{array}{c}
0 \quad 1 \\
0 \quad 1 \\
0 \quad 1 \\
\end{array}
\]

- BDD of \( v_1 \neq v_2 \)

\[
\begin{array}{c}
0 \quad 1 \\
0 \quad 1 \\
0 \quad 1 \\
\end{array}
\]

BDD of a transition relation

- BDDs of

\[
\begin{array}{c}
(v_1' = (v_1 = v_2)) \land (v_2' = (v_1 \oplus v_2))
\end{array}
\]

with two different variable orderings

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

Standard BDD operations

- If formulas \( f_1, f_2 \) represents sets \( s_1, s_2 \), respectively then \( f_1 \land f_2, f_1 \lor f_2 \) represent \( s_1 \cap s_2, s_1 \cup s_2 \), respectively
- Standard algorithms can compute boolean operation on BDDs.
- If \( f(x) \) represents \( \{ x | R(x) \} \) and \( g(s,s') \) represents \( \{ (s,s') | R(s,s') \} \)
  then \( \exists u. f(u) \land g(u,s) \) represents \( \{ s | \exists u. R(u,s) \} \)
- Exist algorithm to compute BDD of \( \exists u. h(u,v) \) from BDD of \( h(u,v) \)
  - BDD of \( \exists u. h(u,v) \) is BDD of \( h(T,v) \lor h(F,v) \)
- 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 $S_0, S_1, S_2, S_3$ etc:
  - $S_0 s = B s$
  - $S_1 s = S_0 s \lor S_0 u \land R(u, s)$
  - $S_2 s = S_1 s \lor S_0 u \land R(u, s)$
  - $S_3 s = S_2 s \lor S_0 u \land R(u, s)$
- BDD of $\exists u. S_0 u \land R(u, s)$ computed by:
  - $\exists u. (S_0 u)[s/s] \land R(s, x')(u, s')$
- efficient using standard BDD algorithms (renaming, then conjunction, then existential quantification)

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

Avoiding building big BDDs

- Transition relation for three machines in parallel
  
  $R(x, y, z) \land (x', y', z') =$
  
  $(x' = \delta_i(x, y, z) \land y' = y \land z' = z) \lor$
  
  $(x' = x \land y' = \delta_j(x, y, z) \land z' = z) \lor$
  
  $(x' = x \land y' = y \land z' = \delta_k(x, y, z))$

- Recall:
  - ReachBy $(n+1) R B s$
    - 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):
  - ReachBy $(n+1) R B (x, y, z)$
    - ReachBy $n R B (x, y, z) \lor$
      - $(\exists \pi. \text{ReachBy } n R B (\pi, y, z) \land x = \delta_i(\pi, y, z)) \lor$
      - $(\exists \pi. \text{ReachBy } n R B (x, \pi, z) \land y = \delta_j(x, \pi, z)) \lor$
      - $(\exists \pi. \text{ReachBy } n R B (x, y, \pi) \land z = \delta_k(x, y, \pi))$

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

Example BDD optimisation: disjunctive partitioning

More Details (Exercise: check the logic below)

Let $Ry(\pi, \eta, \tau)$ abbreviate ReachBy $R B (\pi, \eta, \tau)$ then:

$Ry(\pi, \eta, \tau) \land \exists \pi. \text{ReachBy } R B (\pi, \eta, \tau)$

$\land R((\pi, \eta, \tau), (x, y, z))$

= $(\exists \delta_1 \exists \delta_2 \exists \delta_3 \exists \delta_4 \exists \delta_5 \exists \delta_6 \exists \delta_7 \exists \delta_8 \exists \delta_9)\,$

$Ry(\pi, \eta, \tau) \land \exists \pi. \text{ReachBy } R B (\pi, \eta, \tau)$

$\land R((\pi, \eta, \tau), (x, y, z))$

= $(\exists \delta_1 \exists \delta_2 \exists \delta_3 \exists \delta_4 \exists \delta_5 \exists \delta_6 \exists \delta_7 \exists \delta_8 \exists \delta_9)\,$

$Ry(\pi, \eta, \tau) \land \exists \pi. \text{ReachBy } R B (\pi, \eta, \tau)$

$\land R((\pi, \eta, \tau), (x, y, z))$

= $(\exists \delta_1 \exists \delta_2 \exists \delta_3 \exists \delta_4 \exists \delta_5 \exists \delta_6 \exists \delta_7 \exists \delta_8 \exists \delta_9)\,$

$Ry(\pi, \eta, \tau) \land \exists \pi. \text{ReachBy } R B (\pi, \eta, \tau)$

$\land R((\pi, \eta, \tau), (x, y, z))$

= $(\exists \delta_1 \exists \delta_2 \exists \delta_3 \exists \delta_4 \exists \delta_5 \exists \delta_6 \exists \delta_7 \exists \delta_8 \exists \delta_9)\,$

$Ry(\pi, \eta, \tau) \land \exists \pi. \text{ReachBy } R B (\pi, \eta, \tau)$

$\land R((\pi, \eta, \tau), (x, y, z))$

= $(\exists \delta_1 \exists \delta_2 \exists \delta_3 \exists \delta_4 \exists \delta_5 \exists \delta_6 \exists \delta_7 \exists \delta_8 \exists \delta_9)\,$
Verification and Counterexamples

- Typical safety question:
  - is $Q$ true in all reachable states?
  - i.e. is $\text{Reach } R \cdot S \Rightarrow Q$ true?
- Compute BDD of $\text{Reach } R \cdot S \Rightarrow Q$
- Formula is true if BDD is the single node
  - because $T$ represented by a unique BDD (canonical property)
- If BDD is not $\Box$ 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 neighbours. 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 of 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 characterises the initial [2 marks] and final [2 marks] states.
(c) Explain how you might use a model checker to find a sequences 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 $\text{Reach } R \cdot S \Rightarrow Q$ is not true
- Must exist $s$ satisfying $\text{Reach } R \cdot S \wedge \neg Q$
- Find counterexample algorithm:
  - iteratively generate BDDs of $\text{ReachBy } i \cdot R \cdot S \cdot (i = 0, 1, \ldots)$
  - at each stage check if $\text{ReachBy } i \cdot R \cdot S \wedge \neg(Q \cdot s)$ is satisfiable
  - hence find first $s$ and, using SAT, a state $s_0$ such that $\text{ReachBy } n \cdot R \cdot S \wedge \neg(Q \cdot s_0)$
  - i.e. $\text{ReachBy } n \cdot R \cdot S \cdot s_0 \wedge \neg(Q \cdot s_0)$
- Then use BDD SAT to get $s_0$, where $\text{ReachBy } (n-1) \cdot R \cdot S \wedge R \cdot (s, s_0)$
  - i.e. $\text{ReachBy } (n-1) \cdot R \cdot S_{n-1} \wedge R \cdot (s_{n-1}, s_0)$
- Iteratively trace backwards to get $s_{n-1}, s_{n-2}$ where for $0 < i \leq n$ $\text{ReachBy } (i) \cdot R \cdot S_{i-1} \wedge R \cdot (s_{i-1}, s_i)$
- Can sometimes apply partitioning, so BDD of $R$ 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 $T$.

A transition relation $\text{Trans}$ is then defined by:

$\text{Trans}((v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)) =
\begin{cases} 
(v_0' = v_0) \land (v_1' = v_1) \land (v_2' = v_2) \land (v_3' = v_3) \land (v_4' = v_4) \\
(v_5' = v_5) \land (v_6' = v_6) \land (v_7' = v_7) \land (v_8' = v_8) 
\end{cases}$

(toggle switch 1)

$\begin{cases} 
(v_0' = v_0) \land (v_1' = v_1) \land (v_2' = v_2) \land (v_3' = v_3) \land (v_4' = v_4) \\
(v_5' = v_5) \land (v_6' = v_6) \land (v_7' = v_7) \land (v_8' = v_8) 
\end{cases}$

(toggle switch 2)

$\begin{cases} 
(v_0' = v_0) \land (v_1' = v_1) \land (v_2' = v_2) \land (v_3' = v_3) \land (v_4' = v_4) \\
(v_5' = v_5) \land (v_6' = v_6) \land (v_7' = v_7) \land (v_8' = v_8) 
\end{cases}$

(toggle switch 3)

$\begin{cases} 
(v_0' = v_0) \land (v_1' = v_1) \land (v_2' = v_2) \land (v_3' = v_3) \land (v_4' = v_4) \\
(v_5' = v_5) \land (v_6' = v_6) \land (v_7' = v_7) \land (v_8' = v_8) 
\end{cases}$

(toggle switch 4)

$\begin{cases} 
(v_0' = v_0) \land (v_1' = v_1) \land (v_2' = v_2) \land (v_3' = v_3) \land (v_4' = v_4) \\
(v_5' = v_5) \land (v_6' = v_6) \land (v_7' = v_7) \land (v_8' = v_8) 
\end{cases}$

(toggle switch 5)

$\begin{cases} 
(v_0' = v_0) \land (v_1' = v_1) \land (v_2' = v_2) \land (v_3' = v_3) \land (v_4' = v_4) \\
(v_5' = v_5) \land (v_6' = v_6) \land (v_7' = v_7) \land (v_8' = v_8) 
\end{cases}$

(toggle switch 6)

$\begin{cases} 
(v_0' = v_0) \land (v_1' = v_1) \land (v_2' = v_2) \land (v_3' = v_3) \land (v_4' = v_4) \\
(v_5' = v_5) \land (v_6' = v_6) \land (v_7' = v_7) \land (v_8' = v_8) 
\end{cases}$

(toggle switch 7)

$\begin{cases} 
(v_0' = v_0) \land (v_1' = v_1) \land (v_2' = v_2) \land (v_3' = v_3) \land (v_4' = v_4) \\
(v_5' = v_5) \land (v_6' = v_6) \land (v_7' = v_7) \land (v_8' = v_8) 
\end{cases}$

(toggle switch 8)

$\begin{cases} 
(v_0' = v_0) \land (v_1' = v_1) \land (v_2' = v_2) \land (v_3' = v_3) \land (v_4' = v_4) \\
(v_5' = v_5) \land (v_6' = v_6) \land (v_7' = v_7) \land (v_8' = v_8) 
\end{cases}$

(toggle switch 9)
Predicates $Init$, $Final$ characterising the initial and final states, respectively, are defined by:

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

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

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 $\neg Final(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)$.

Properties

- $\text{Reach } R s \Rightarrow Q s$ means $Q$ true in all reachable states
- Might want to verify other properties, e.g:
  1. $Device\text{Enabled}$ 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 $Restart$ 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:

  ![Circuit Diagram](image)

  - Input: $dreq$, registers: $q0$, $dack$
  - Timing Diagram:

    ![Timing Diagram](image)

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

- 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 $(R, s)$ — a transition relation and an initial state
- Define:
  - $\text{Atom}(p) = \lambda(R, s). \ p(s)$
  - $\neg p = \lambda(R, s). \ \neg (P(R, s))$
  - $P \land Q = \lambda(R, s). \ P(R, s) \land Q(R, s)$
  - $P \lor Q = \lambda(R, s). \ P(R, s) \lor Q(R, s)$
  - $P \Rightarrow Q = \lambda(R, s). \ P(R, s) \Rightarrow Q(R, s)$
  - $AXP = \lambda(R, s). \ \forall \sigma'. R(s, s') \Rightarrow P(R, s')$
  - $EXP = \lambda(R, s). \ \exists \sigma'. R(s, s') \land P(R, s')$
  - $A[P \ U Q] = \lambda(R, s). \ \forall \sigma . \text{Path}(R, s)\sigma$
    \[ \Rightarrow \exists i . Q(R, \sigma(i)) \land \sigma(j < i \Rightarrow P(R, \sigma(j))) \]
  - $E[P \ U Q] = \lambda(R, s). \ \exists \sigma . \text{Path}(R, s)\sigma$
    \[ \Rightarrow \exists i . Q(R, \sigma(i)) \land \lambda j . \sigma(j < i \Rightarrow P(R, \sigma(j))) \]

Computation Tree Logic (CTL)

- Syntax of CTL well-formed formulas:
  - $\text{wff} ::= \text{Atom}(p)$ (Atomic formula)
  - $\neg \text{wff}$ (Negation)
  - $\text{wff}_1 \land \text{wff}_2$ (Conjunction)
  - $\text{wff}_1 \lor \text{wff}_2$ (Disjunction)
  - $\text{wff}_1 \Rightarrow \text{wff}_2$ (Implication)
  - $AX\text{wff}$ (All successors)
  - $EX\text{wff}$ (Some successors)
  - $A[\text{wff}_1 \ U \ \text{wff}_2]$ (Until – along all paths)
  - $E[\text{wff}_1 \ U \ \text{wff}_2]$ (Until – along some path)
- 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 $AFP = A[\top \ U \ P]$
- $AFF$ is true if $P$ holds somewhere along every $R$-path – $P$ is invariant
  \[ A[\top \ U \ P] = A[\top \ U \ P] \land \forall \sigma . \text{Path}(R, s)\sigma \Rightarrow \exists i . P(R, \sigma(i)) \land \forall j . j < i \Rightarrow \top(R, \sigma(j)) \]
  \[ = A[\top \ U \ P] \land \forall \sigma . \text{Path}(R, s)\sigma \Rightarrow \exists i . P(R, \sigma(i)) \land \forall j . j < i \Rightarrow P(R, \sigma(j)) \]
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

  $$\text{EF} P = E[T \cup P']$$
  $$= \lambda(R, s). \exists \sigma. \text{Path}(R, s) \sigma \land \exists i. P(R, \sigma(i)) \land \forall j. j < i \Rightarrow T(R, \sigma(j))$$

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

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

The defined operator **EG**

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

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

The defined operator **A[PWQ]**

- $A[PWQ]$ is a ‘partial correctness’ version of $A[PUQ]$
- 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 $A[PWQ]$
  $$A[PWQ] = \neg \text{EF}(\neg P)$$
  $$= \lambda(R, s). \neg \exists \sigma. \text{Path}(R, s) \sigma \land \exists i. \neg P(R, \sigma(i))$$
  $$= \lambda(R, s). \forall \sigma. \text{Path}(R, s) \sigma \land \forall i. \neg P(R, \sigma(i))$$

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

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

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

Continuing (2)

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

Exercise: does this correspond to earlier description of A[PWQ]?

- this exercise illustrates the subtlety of writing CTL!

A[PWF] = AG P

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

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

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

Exercise: describe the property specified by A[PWF]