Specification 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

The notes contain general and background material for the course.
Some of the material in them may not be covered in the lectures.
Some details and examples are only presented in the lectures.
The examinable material is what is actually covered in the lectures

---

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 chose 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}
  - Assertion Based Verification (ABV) for hardware
    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_0 being the least significant bit)

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

hence

$$\begin{array}{rcl} ab & = & (2^{n-1}a_{n-1} + 2^{n-2}a_{n-2} + \cdots + 2^0 a_0)b \\ & = & 2^{n-1}a_{n-1}b + 2^{n-2}a_{n-2}b + \cdots + 2^0 a_0 b \\ & = & a_{n-1}2^{n-1}b + a_{n-2}2^{n-2}b + \cdots + a_0 2^0 b \end{array}$$

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 $b \mapsto b{\cdot}0$, then:

$$\begin{array}{rcl} 2^0 b & = & b \\ 2^1 b & = & b{\cdot}0 \\ 2^2 b & = & b{\cdot}00 \\ & \vdots & \\ 2^n b & = & b{\cdot}\underbrace{0\cdots0}_{n\ 0s} \end{array}$$

- Recall: $ab = a_{n-1}2^{n-1}b + a_{n-2}2^{n-2}b + \cdots + a_0 2^0 b$
- Thus product of a and b is given by the sum:

$$\begin{array}{lr} & a_0 b \\ + & a_1 b{\cdot}0 \\ + & a_2 b{\cdot}00 \\ + & a_3 b{\cdot}000 \\ & \vdots \\ + & a_{n-1}b{\cdot}0\cdots0 \end{array}$$

  - the ith row is either all zeros (if a_i is 0)
  - or b shifted i places to the left (if a_i is 1)
- a, b need n-bits $\Rightarrow$ product needs $2n$ bits

Extracting bits and subwords

- Let $\mathtt{A}[n]$ denote the n-th bit of the binary representation of $\mathtt{A}$
- $\mathtt{A}[n]$ is a number 1 or 0
- $\mathtt{A}[0]$ is the least significant bit
- Thus:

$$\mathtt{A}[n] = (\mathtt{A}\ \mathtt{div}\ 2^n)\ \mathtt{mod}\ 2$$

- Define $\mathtt{A}[m:n]$ to be the numerical value of the word comprising bits n upto to m of $\mathtt{A}$:

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

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

Hoare logic verification of a multiplier

- Add-shift multiplication program:

```
I := 0; PROD := 0;
WHILE I < N DO
 BEGIN PROD := PROD + A[I] × (2^I × B);
       I := I + 1;
 END
```

- Annotated Hoare specification:

$\{\mathtt{A} = a \ \wedge\ \mathtt{B} = b \ \wedge\ a < 2^{\mathtt{N}} \ \wedge\ b < 2^{\mathtt{N}} \ \wedge\ \mathtt{N} > 0\}$

```
I := 0; PROD := 0;
WHILE I < N DO {I ≤ N ∧ 2^I A[N-1 : I]B + PROD = ab}
 BEGIN PROD := PROD + A[I] × (2^I × B);
       I := I + 1;
 END
```

$\{\mathtt{PROD} = a \times b\}$

- Routine (not trivial) to verify using Hoare Logic
  - reasoning about `div` and `mod` is *horrible*

Using FOR-commands instead of WHILE

$\vdash$ $\{A = a \ \wedge \ B = b \ \wedge \ a < 2^N \ \wedge \ b < 2^N \ \wedge \ N > 0\}$

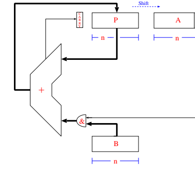
```
PROD := 0;
FOR I := 0 UNTIL N-1 DO
 BEGIN PROD := PROD + A[I]×B;
       B := 2×B;
 END
```

$\{PROD = a \times b\}$

- 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 PROD
  - then shifting B one bit to the left (i.e. multiplying it by 2)

Textbook multiplier

- Simple textbook add-shift multiplier:



- 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
  - Advantages: corresponds more to intuition – words have a size
  - Disadvantage: can't use off-the-shelf theory of arithmetic
- 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 $\mathtt{Bw}(b)$
- $\mathtt{Bv}(b)$ is the number represented by bit b
- $\mathtt{V}(w)$ is the natural number represented by word w
- W n maps number m to the n-bit word representing it
- Concatenation of w_1 with w_2 denoted by $w_1 \cdot w_2$
- $w\{n \leftarrow b\}$ 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: $w_1 \uplus w_2 \ = \ \mathtt{W}\ (\mathtt{max}(|w_1|, |w_2|) + 1)\ (\mathtt{V}(w_1) + \mathtt{V}(w_2))$
- $b.w$ equals w if $b = \mathtt{T}$ and equals $\mathtt{W}\ |w|\ 0$ if $b = \mathtt{F}$

Words vs bits

- $w[n:n]$ is the 1-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 $\mathtt{Bw}(b)$
- Thus: $\mathtt{Bw}(b)[0] \ = \ b$

Representing Numbers

- **Natural number:** $b_{n-1} \cdots b_0$ represents
$$2^{n-1}{\times}b_{n-1} + 2^{n-2}{\times}b_{n-2} + \cdots + 2^0{\times}b_0$$

- **Integer:** $b_{n-1} \cdots b_0$ represents
$$-2^{n-1}{\times}b_{n-1} + 2^{n-2}{\times}b_{n-2} + \cdots + 2^0{\times}b_0$$
  - this is the two's complement representation

- $\mathtt{V}(w)$ is the natural number represented by a w
$$\mathtt{V}(b_{n-1} \cdots b_0) \;=\; 2^{n-1}{\times}b_{n-1} + 2^{n-2}{\times}b_{n-2} + \cdots + 2^0{\times}b_0$$

- Words can represent other values
  - e.g. floating point numbers; opcodes

- $\mathtt{Bv}(b)$ is the number represented by b
$$\mathtt{Bv}(\mathtt{T}) \;=\; 1 \quad \text{and} \quad \mathtt{Bv}(\mathtt{F}) \;=\; 0$$

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 $(a \wedge b) \vee (c \wedge (a \oplus b))$

- This is verified by:
$$\mathtt{Bv}(a \oplus b \oplus c) \;=\; (\mathtt{Bv}(a) + \mathtt{Bv}(b) + \mathtt{Bv}(c)) \;\mathtt{mod}\; 2$$
$$\mathtt{Bv}((a \wedge b) \vee (c \wedge (a \oplus b))) \;=\; (\mathtt{Bv}(a) + \mathtt{Bv}(b) + \mathtt{Bv}(c)) \;\mathtt{div}\; 2$$

Verification by enumeration

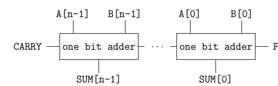
- Sum:

a	b	c	$\mathtt{Bv}(a \oplus b \oplus c)$	$(\mathtt{Bv}(a) + \mathtt{Bv}(b) + \mathtt{Bv}(c))$ mod 2
1	1	1	1	1
1	1	0	0	0
1	0	1	0	0
1	0	0	1	1
0	1	1	0	0
0	1	0	1	1
0	0	1	1	1
0	0	0	0	0

- Carry:

a	b	c	$\mathtt{Bv}((a \wedge b) \vee (c \wedge (a \oplus b)))$	$(\mathtt{Bv}(a) + \mathtt{Bv}(b) + \mathtt{Bv}(c))$ div 2
1	1	1	1	1
1	1	0	1	1
1	0	1	1	1
1	0	0	0	0
0	1	1	1	1
0	1	0	0	0
0	0	1	0	0
0	0	0	0	0

Verification of a ripple-carry adder of any size



- Let R be:
$2^{\mathtt{I}}\mathtt{Bv}(\mathtt{CARRY}) + \mathtt{V}(\mathtt{SUM[I-1:0]}) = \mathtt{V}(\mathtt{A[I-1:0]}) + \mathtt{V}(\mathtt{B[I-1:0]}) \;\wedge$
$\mathtt{A} = w_1 \;\wedge\; \mathtt{B} = w_2$

- Consider the following annotated specification:

$\{\mathtt{A} = w_1 \;\wedge\; \mathtt{B} = w_2 \;\wedge\; \mathtt{SUM} = \mathtt{W\ N\ 0} \;\wedge\; \mathtt{CARRY} = \mathtt{F} \;\wedge$
$|w_1| \leq \mathtt{N} \;\wedge\; |w_2| \leq \mathtt{N} \;\wedge\; \mathtt{N} > 0\}$

```
FOR I := 0 UNTIL N-1 DO {R}
  BEGIN
    SUM[I]:=A[I]⊕B[I]⊕CARRY;
    CARRY:=(A[I]∧B[I])∨(CARRY∧(A[I]⊕B[I]));
  END
```

$\{2^{\mathtt{N}}\mathtt{Bv}(\mathtt{CARRY}) + \mathtt{V}(\mathtt{SUM[N-1:0]}) \;=\; \mathtt{V}(\mathtt{A[N-1:0]}) + \mathtt{V}(\mathtt{B[N-1:0]})$
$\mathtt{A} = w_1 \;\wedge\; \mathtt{B} = w_2\}$

- `A`, `B` are `N`-bit words, `SUM`, `CARRY` are truthvalues, `I` is an integer

- Proof horrible (omitted)

Word multiplication program

- **Simple add-shift multiplication**
- **Annotated correctness specification:**

```
{V(A) = a  ∧  V(B) = b  ∧  PROD = W(2N)0  ∧
 |A| ≤ N  ∧  |B| ≤ N  ∧  N > 0}
 FOR I:=0 UNTIL N-1 DO
   {2^I V(A[N-1 : I])b + V(PROD) = ab  ∧  V(B) = 2^I b}
  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**

```
FOR I := 0 UNTIL N-1 DO
 BEGIN
  SUM[I]:=A[I]⊕B[I]⊕CARRY;
  CARRY:=(A[I]∧B[I])∨(CARRY∧(A[I]⊕B[I]));
 END
```

- **If a particular value of `N` is fixed, then the program can be unrolled into the normal circuit for an adder.**
- **For example take $\mathtt{N} = 3$ to get:**

```
FOR I := 0 UNTIL 2 DO
 BEGIN
  SUM[I]:=A[I]⊕B[I]⊕CARRY;
  CARRY:=(A[I]∧B[I])∨(CARRY∧(A[I]⊕B[I]));
 END
```

N=3 adder

- **3-bit adder:**

```
FOR I := 0 UNTIL 2 DO
 BEGIN
  SUM[I]:=A[I]⊕B[I]⊕CARRY;
  CARRY:=(A[I]∧B[I])∨(CARRY∧(A[I]⊕B[I]));
 END
```

- **Assuming initially `CARRY` = `F`; `FOR`-command unrolls to:**

```
SUM[0]:=A[0]⊕B[0]⊕F;
CARRY:=(A[0]∧B[0])∨(F∧(A[0]⊕B[0]));
SUM[1]:=A[1]⊕B[1]⊕CARRY;
CARRY:=(A[1]∧B[1])∨(CARRY∧(A[1]⊕B[1]));
SUM[2]:=A[2]⊕B[2]⊕CARRY;
CARRY:=(A[2]∧B[2])∨(CARRY∧(A[2]⊕B[2]));
```

- **Symbolically executing yields logic equations:**

```
SUM[0]:=A[0]⊕B[0];
SUM[1]:=A[1]⊕B[1]⊕(A[0]∧B[0]);
SUM[2]:=A[2]⊕B[2]⊕
          ((A[1]∧B[1])∨((A[0]∧B[0])∧(A[1]⊕B[1])));
CARRY :=(A[2]∧B[2])∨
        (((A[1]∧B[1])∨((A[0]∧B[0])∧(A[1]⊕B[1])))∧
         (A[2]⊕B[2]));
```

Combinational logic

- **Derived program is combinational logic:**

```
SUM[0]:=A[0]⊕B[0];
SUM[1]:=A[1]⊕B[1]⊕(A[0]∧B[0]);
SUM[2]:=A[2]⊕B[2]⊕
          ((A[1]∧B[1])∨
           ((A[0]∧B[0])∧(A[1]⊕B[1])));
CARRY :=(A[2]∧B[2])
        ∨
        (((A[1]∧B[1])∨
          ((A[0]∧B[0])∧(A[1]⊕B[1])))∧
         (A[2]⊕B[2]));
```

- **These are independent assignments**
  - boolean expressions for computing the values of `SUM` and `CARRY` directly in terms of the `A[0]`, `A[1]`, `A[2]`, `B[0]`, `B[1]` and `B[2]`
- **This process yields logic for adders of arbitrary (fixed) bit-widths**
- **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 `N` 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 state 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

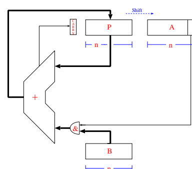
- HDLs allow operations to be scheduled to clock cycles
- Statements can be prefixed by `@`
  - the symbol `@` introduces an *event control*
- Multiplier that takes `N` cycles:

```
FOR I := 0 UNTIL N-1 DO
 @R := (R[0].B ⊎ R[2N-1:N])⁀R[N-1:1]
```

`R = CARRY⁀P⁀A`

- Multiplier that takes `2N` cycles:

```
FOR I := 0 UNTIL N-1 DO
 BEGIN
  @SUM := R[0].B ⊎ R[2N-1:N];
  @R := SUM⁀R[N-1:1]
 END
```



- 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:

```
FOR I := 0 UNTIL N-1 DO
 BEGIN
  @SUM := R[0].B ⊎ R[2N-1:N];
  B := ¬B;
  @R := SUM⁀R[N-1:1];
  B := ¬B;
 END
```

- 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:**

```
R:=X;
Q:=0;
WHILE Y≤R DO
  BEGIN R:=R-Y; Q:=Q+1 END
```

- **Implemented as a machine**
  - registers X, Y, Q and R
  - a subtracter and incrementer
  - 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

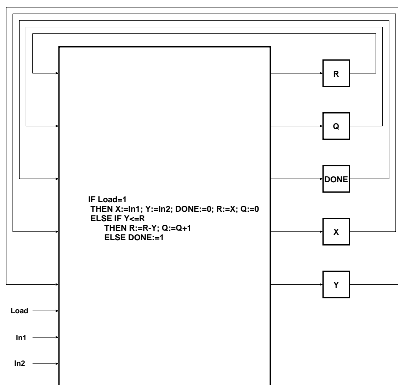
- To emphasize the continuously-running nature of hardware, recast division program as (where FOREVER is WHILE T DO):

```
FOREVER
  IF Load=1
   THEN X:=In1; Y:=In2; DONE:=0; R:=X; Q:=0
   ELSE IF Y≤R THEN R:=R-Y; Q:=Q+1
                ELSE DONE:=1
```

  - In1, In2 and Load are **inputs** whose value is determined by the environment (e.g. the user)
  - X, Y, Q, R and 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 DONE=0 ⇒ Load=0

FOREVER C

- Each iteration step consists of
  - Circuit C computes new values of registers from current values and inputs
  - then updating the registers



Programs as temporal statements

- Would like a generalised Hoare Logic specification:

⊢ {*If environment ensures always that:* DONE=0 ⇒ Load=0
and if Load *is set to* 1 *when:* In1 $= x \wedge$ In2 $= y$}

```
FOREVER
 IF Load=1
  THEN X:=In1; Y:=In2; DONE:=0; R:=X; Q:=0
  ELSE IF Y≤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 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**

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 other 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 ($reFL^{ect}$).
- 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

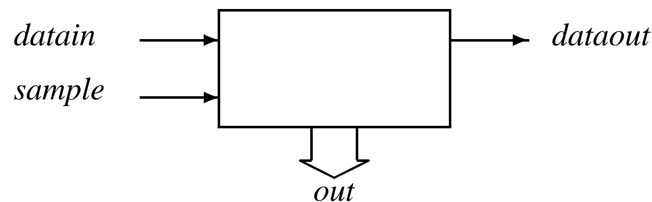
- Formal proof can't guarantee actual chips will work:
  - design models are not always accurate
  - there may be fabrication defects
- Specifications may not capture requirements:
  - large specifications may be unreadable
  - some input conditions may be ignored

Modelling Hardware in Higher Order Logic

Original slides by Tom Melham and Michael Norrish
(edited by Mike Gordon)

Why Formal Specification?

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



This can be specified *informally* by

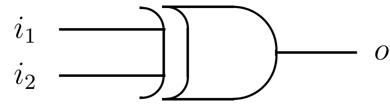
> The input line *datain* accepts a stream of bits, and the output line *dataout* emits the same stream delayed by four cycles. The bus *out* is four bits wide. If the input *sample* is false then the 4-bit word at *out* is the last four bits input at *datain*. Otherwise, the output word is all zeros.

Hardware Verification Method

- Classical method of hardware verification:
  1. write a specification of intended behaviour
     Spec
  2. write specifications of the design components
     Part-$1, \ldots$ 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

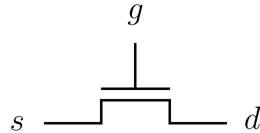
Specification Examples

- Simple combinational behaviour:



$\vdash \mathsf{Xor}(i_1, i_2, o) = (o = \neg(i_1 = i_2))$

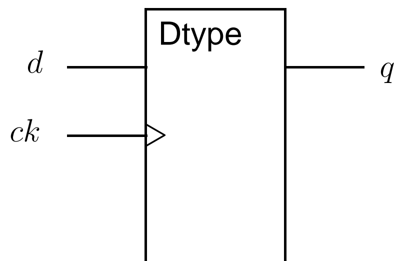
- Bidirectional wires:



$\vdash \mathsf{Ntran}(g, s, d) = (g \Rightarrow (d = s))$

Specification Examples

Sequential (time-dependent) behaviour:



$\vdash \mathsf{Dtype}(ck, d, q) = \forall t.\, q(t{+}1) = (\mathsf{if}\ \mathsf{Rise}\ ck\ t\ \mathsf{then}\ d\ t\ \mathsf{else}\ q\ t)$

$\vdash \mathsf{Rise}\ ck\ t = \neg ck(t) \wedge ck(t{+}1)$

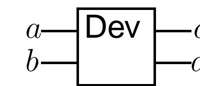
Why Formal Specification?

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!

Formal Specification in HOL

- Consider the following device:

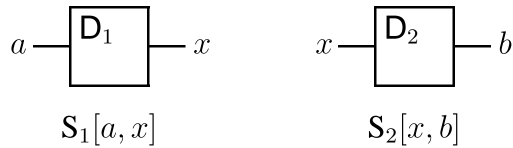


  This is specified by a boolean term $S[a, b, c, d]$ with free variables a, b, c, and d.

- The idea is that
  - a, b, c, d model externally-observable values
  - $\mathrm{S}[a, b, c, d] = \begin{cases} \mathsf{T} & \text{if } a, b, c, \text{ and } d \text{ could occur simultaneously on the corresponding external wires of the device } \mathsf{Dev} \\ \mathsf{F} & \text{otherwise} \end{cases}$

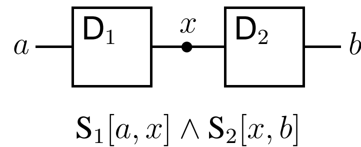
Composing Behaviours

- Consider the following two devices:



$\mathrm{S}_1[a, x]$ $\mathrm{S}_2[x, b]$

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



$\mathrm{S}_1[a, x] \wedge \mathrm{S}_2[x, b]$

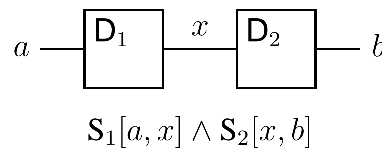
Specification of the Sampler

- We can specify the sampler formally by

$$\begin{aligned}
&\forall t{:}\mathit{time}.\\
&\quad (\mathit{dataout}(t) = \mathit{datain}(t{-}4))\\
&\quad \wedge\\
&\quad (\mathit{out}(t) = \ \mathsf{if}\ \mathit{sample}(t)\\
&\qquad\qquad \mathsf{then}\ [\mathsf{F}; \mathsf{F}; \mathsf{F}; \mathsf{F}]\\
&\qquad\qquad \mathsf{else}\ \ [\mathit{datain}(t{-}4);\ \mathit{datain}(t{-}3);\\
&\qquad\qquad\qquad \mathit{datain}(t{-}2);\ \mathit{datain}(t{-}1)\ ])
\end{aligned}$$

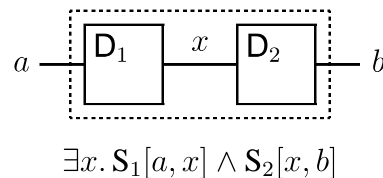
Hiding Internal Structure

- Consider the composite device



$\mathrm{S}_1[a, x] \wedge \mathrm{S}_2[x, b]$

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



$\exists x.\, \mathrm{S}_1[a, x] \wedge \mathrm{S}_2[x, b]$

- Existential quantification is called a *hiding* operator—it 'hides' internal wires.

Specification of the Sampler

- We can specify the sampler formally by

$$\begin{aligned}
&\forall t{:}\mathit{time}.\\
&\quad (\mathit{dataout}(t) = \mathit{datain}(t{-}4))\\
&\quad \wedge\\
&\quad (\mathit{out}(t) = \ \mathsf{if}\ \mathit{sample}(t)\\
&\qquad\qquad \mathsf{then}\ [\mathsf{F}; \mathsf{F}; \mathsf{F}; \mathsf{F}]\\
&\qquad\qquad \mathsf{else}\ \ [\mathit{datain}(t{-}4);\ \mathit{datain}(t{-}3);\\
&\qquad\qquad\qquad \mathit{datain}(t{-}2);\ \mathit{datain}(t{-}1)\ ])
\end{aligned}$$

- 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

Hiererchical Verification

The hierarchical verification method:

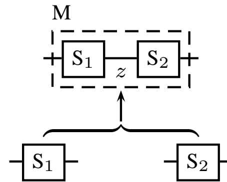
Level 0

Model:

$\vdash \mathrm{M} = \exists z.\, \mathrm{S}_1 \wedge \mathrm{S}_2$

Correctness:

$\vdash \mathrm{M} \underset{F}{\mathsf{sat}} \mathrm{S}$



Level 1

Models:

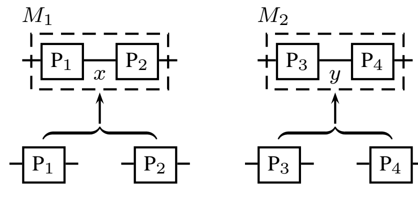
$\vdash \mathrm{M}_1 = \exists x.\, \mathrm{P}_1 \wedge \mathrm{P}_2$

$\vdash \mathrm{M}_2 = \exists y.\, \mathrm{P}_3 \wedge \mathrm{P}_4$

Correctness:

$\vdash \mathrm{M}_1 \underset{G}{\mathsf{sat}} \mathrm{S}_1$

$\vdash \mathrm{M}_2 \underset{G}{\mathsf{sat}} \mathrm{S}_2$



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:

$$\texttt{COMP}(\texttt{p1}, \ldots, \texttt{pm}) = \exists \texttt{w1} \ldots \texttt{wn}.\ \texttt{COMP1}(\ldots) \wedge \texttt{COMP2}(\ldots)$$

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

Formulating Correctness

- A key part of formal hardware verification is formalizing what ‘correctness’ *means*.
- The strongest formulation is *equivalence*:

$$\vdash \forall v_1 \ldots v_n.\ \mathrm{M}[v_1, \ldots, v_n] = \mathrm{S}[v_1, \ldots, v_n]$$

- For *partial* specifications, use *implication*:

$$\vdash \forall v_1 \ldots v_n.\ \mathrm{M}[v_1, \ldots, v_n] \Rightarrow \mathrm{S}[v_1, \ldots, v_n]$$

- In general, the satisfaction relationship

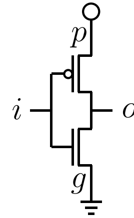
$$\vdash \mathrm{M}[v_1, \ldots, v_n] \underset{abs}{\mathsf{sat}} \mathrm{S}[abs(v_1), \ldots, abs(v_n)]$$

  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.

Design Model and Correctness

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

$$\vdash \mathsf{Inv}(i,o) =$$
$$\exists g\, p.\ \mathsf{Pwr}\ p \wedge \mathsf{Gnd}\ g\ \wedge$$
$$\mathsf{Ntran}(i,g,o) \wedge \mathsf{Ptran}(i,p,o)$$



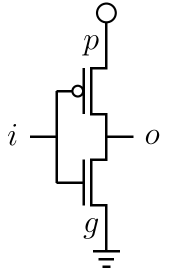
- Correctness is formulated by the equivalence:

$$\vdash \forall i\, o.\ \mathsf{Inv}(i,o) = (o = \neg i)$$

This follows by purely logical inference…

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



The Correctness Proof

- Definition of Inv:

$$\vdash \mathsf{Inv}(i,o) =$$
$$\exists g\, p.\ \mathsf{Pwr}\ p \wedge \mathsf{Gnd}\ g\ \wedge$$
$$\mathsf{Ntran}(i,g,o) \wedge \mathsf{Ptran}(i,p,o)$$

- Expanding with definitions:

$$\vdash \mathsf{Inv}(i,o) =$$
$$\exists g\, p.\ (p = \mathsf{T}) \wedge (g = \mathsf{F})\ \wedge$$
$$(i \Rightarrow (o = g)) \wedge (\neg i \Rightarrow (o = p))$$

- By simple logical reasoning:

$$\vdash \mathsf{Inv}(i,o) = (i \Rightarrow (o = \mathsf{F})) \wedge (\neg i \Rightarrow (o = \mathsf{T}))$$

CMOS Primitives

- Formal specifications of primitives:

g, s, d $\quad \vdash \mathsf{Ptran}(g,s,d) = (\neg g \Rightarrow (d = s))$

g, s, d $\quad \vdash \mathsf{Ntran}(g,s,d) = (g \Rightarrow (d = s))$

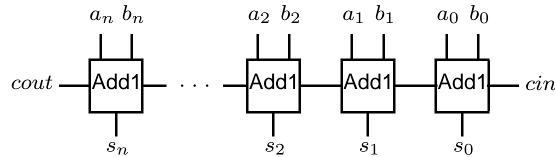
g $\quad \vdash \mathsf{Gnd}\ g = (g = \mathsf{F})$

p $\quad \vdash \mathsf{Pwr}\ p = (p = \mathsf{T})$

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

Another Example

- An $(n{+}1)$-bit ripple-carry adder:



- We wish to prove that:

$$(2^{n+1} \times cout) + s = a + b + cin$$

- 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

The Correctness Proof continued

- Simplifying gives:

$$\vdash \mathsf{Inv}(i, o) = (i \Rightarrow \neg o) \wedge (\neg i \Rightarrow o)$$

- By the law of the contrapositive:

$$\vdash \mathsf{Inv}(i, o) = (o \Rightarrow \neg i) \wedge (\neg i \Rightarrow o)$$

- By the definition of boolean equality:

$$\vdash \mathsf{Inv}(i, o) = (o = \neg i)$$

- Generalizing the free variables gives:

$$\vdash \forall i\, o.\ \mathsf{Inv}(i, o) = (o = \neg i)$$

Defining the Model: types

- Specification uses numbers, i.e. values of type num
- Implementation uses words – values of type $word$
  - n^{th} bit of w denoted by $w\,[\,n\,]$
  - $w\,[\,m : n\,]$ denotes bits m to n of w
  - $\mathsf{Bv}(b)$ is the number represented by bit b
  - $\mathsf{V}(w)$ is the natural number represented by word w
- Abstraction from words to numbers (data abstraction):

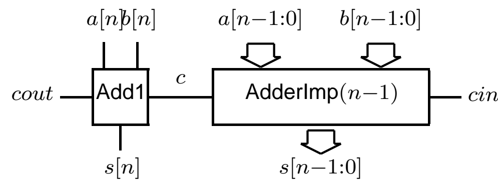
$$\begin{array}{lcl} \vdash \mathsf{Bv}\ b & = & \mathsf{if}\ b\ \mathsf{then}\ 1\ \mathsf{else}\ 0 \\ \vdash \mathsf{V}\ w[0:0] & = & \mathsf{Bv}\ w[0] \\ \vdash \mathsf{V}\ w[n{+}1:0] & = & 2^{n+1}(\mathsf{Bv}\ w[n{+}1])\ +\ \mathsf{V}\ w[n:0] \end{array}$$

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

Defining the Model

- Recursive view of an $n{+}1$-bit adder:



- Primitive recursive definition in logic:

$$\mathsf{AdderImp}(0)(a, b, cin, s, cout) = \mathsf{Add1}\,(a[0], b[0], cin, s[0], cout)$$

$$\mathsf{AdderImp}\ n\ (a, b, cin, s, cout) = \exists c.\ \mathsf{Add1}\,(a[n], b[n], c, s[n], cout) \wedge \mathsf{AdderImp}(n{-}1)(a[n{-}1{:}0], b[n{-}1{:}0], cin, s[n{-}1{:}0], c)$$

Formulation of Correctness

- Logical formulation of correctness:

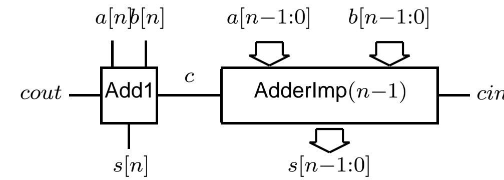
$$\mathsf{Spec}\ n\ (a, b, cin, s, cout) = ((2^{n+1}\ cout){+}s = a{+}b{+}cin)$$

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

- Note the data abstraction (*abs* in an earlier slide)
- This is easy to prove (done later in the course)

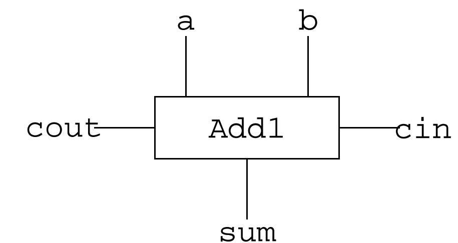
Defining the Model: recursive definition

- If $n > 0$ an $(n{+}1)$-bit adder is built from an n-bit adder



Defining the Model: Add1

- Diagram of a 1-bit full adder:



- Lines `a`, `b`, `cin`, `sum` and `cout` carry boolean values
- Specification (note data abstraction from *bool* to *num*):

$$\mathsf{Add1}(\mathtt{a}, \mathtt{b}, \mathtt{cin}, \mathtt{sum}, \mathtt{cout}) = (2{\times}\mathsf{Bv}(\mathtt{cout}) + \mathsf{Bv}(\mathtt{sum}) = \mathsf{Bv}(\mathtt{a}) + \mathsf{Bv}(\mathtt{b}) + \mathsf{Bv}(\mathtt{cin}))$$

Formulating Correctness

- Then correctness is stated by:

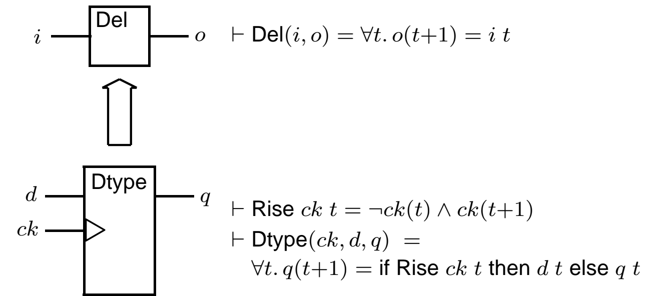
$$\vdash \forall ck.\ \mathsf{Inf}(\mathsf{Rise}\ ck) \Rightarrow$$
$$\forall d\ q.\ \mathsf{Dtype}(ck, d, q) \Rightarrow$$
$$\mathsf{Del}(d\ \mathsf{when}\ (\mathsf{Rise}\ ck), q\ \mathsf{when}\ (\mathsf{Rise}\ ck))$$

- Note the formal *validity condition*:

$$\vdash \mathsf{Inf}\ P = \forall t.\ \exists t'.\ t' > t\ \wedge\ P\ t'$$

Temporal Abstraction

- Example—abstracting to unit delay:



- 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

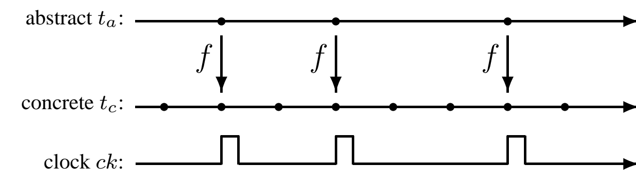
- Intel
  - floating point algorithms (uses HOL Light system)
  - hardware (uses internal tools Forte/reFL$^{\text{ect}}$)
- AMD
  - floating point (uses ACL2 prover)
- Sun
  - high level architecture verification (PVS)
- Rockwell Collins
  - low level code verification (ACL2)

. . . .

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

Formulating Correctness

- A mapping between time-scales:



- Define the temporal abstraction functions:

$\vdash$ Timeof P n = *the time on* t_c *such that P true for nth time*

$\vdash$ $signal$ when $P = signal \circ (\mathsf{Timeof}\ P)$

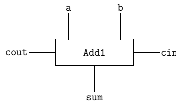
where $(f \circ g)x\ =\ f(g\ x)$ [∘ is function composition]

Summary

- Specifying behaviour:
  - predicates—$\mathrm{S}[a, b, c, d\,]$
- Specifying structure:
  - composition—$\mathrm{S}_1[a, x] \wedge \mathrm{S}_2[x, b]$
  - hiding—$\exists x.\ \mathrm{S}_1[a, x] \wedge \mathrm{S}_2[x, b]$
- Formulating correctness:
  - $\vdash \forall v_1\ \ldots\ v_n.\ \mathrm{M}[v_1, \ldots, v_n] = \mathsf{S}[v_1, \ldots, v_n]$
  - $\vdash \forall v_1\ \ldots\ v_n.\ \mathrm{M}[v_1, \ldots, v_n] \Rightarrow \mathrm{S}[v_1, \ldots, v_n]$
  - $\vdash \forall v_1\ \ldots\ v_n.\ \mathrm{M}[v_1, \ldots, v_n] \Rightarrow \mathrm{S}[abs\ v_1, \ldots, abs\ v_n]$
- Abstraction
  - data: $w \mapsto \mathsf{V}(w)$
  - temporal: $sig \mapsto sig\ \mathsf{when}\ (\mathsf{Rise}\ clk)$

A 1-bit CMOS full adder

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



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

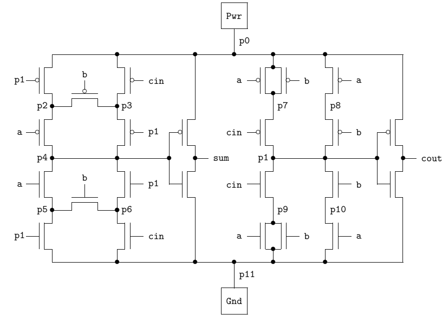
- **Specification of the adder:**

  $\texttt{Add1}(a, b, cin, sum, cout) \equiv$
  $(2{\times}\texttt{Bv}(cout) + \texttt{Bv}(sum) = \texttt{Bv}(a) + \texttt{Bv}(b) + \texttt{Bv}(cin))$

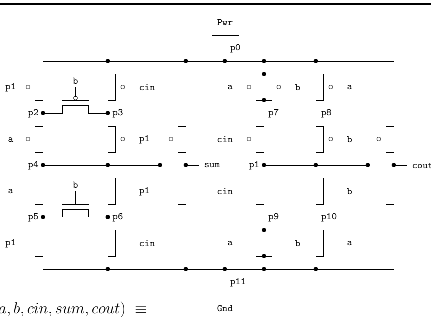
- **A correct implementation has:**
  - **lines** `a`, `b`, `cin`, `sum` **and** `cout`
  - **constrains** a, b, cin, sum **and,** $cout$ **so** $\texttt{Add1}(a, b, cin, sum, cout)$

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



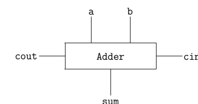
$\texttt{Add1_Imp}(a, b, cin, sum, cout) \equiv$
$\exists p_0\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6\ p_7\ p_8\ p_9\ p_{10}\ p_{11}.$
$\quad \texttt{Ptran}(p_1, p_0, p_2) \wedge \texttt{Ptran}(cin, p_0, p_3) \wedge \texttt{Ptran}(b, p_2, p_3) \wedge \texttt{Ptran}(a, p_2, p_4) \wedge$
$\quad \texttt{Ptran}(p_1, p_3, p_4) \wedge \texttt{Ntran}(a, p_4, p_5) \wedge \texttt{Ntran}(p_1, p_4, p_6) \wedge \texttt{Ntran}(b, p_5, p_6) \wedge$
$\quad \texttt{Ntran}(p_1, p_5, p_{11}) \wedge \texttt{Ntran}(cin, p_6, p_{11}) \wedge \texttt{Ptran}(a, p_0, p_7) \wedge \texttt{Ptran}(b, p_0, p_7) \wedge$
$\quad \texttt{Ptran}(a, p_0, p_8) \wedge \texttt{Ptran}(cin, p_7, p_1) \wedge \texttt{Ptran}(b, p_8, p_1) \wedge \texttt{Ntran}(cin, p_1, p_9) \wedge$
$\quad \texttt{Ntran}(b, p_1, p_{10}) \wedge \texttt{Ntran}(a, p_9, p_{11}) \wedge \texttt{Ntran}(b, p_9, p_{11}) \wedge \texttt{Ntran}(a, p_{10}, p_{11}) \wedge$
$\quad \texttt{Pwr}(p_0) \wedge \texttt{Ptran}(p_4, p_0, sum) \wedge \texttt{Ntran}(p_4, sum, p_{11}) \wedge$
$\quad \texttt{Gnd}(p_{11}) \wedge \texttt{Ptran}(p_1, p_0, cout) \wedge \texttt{Ntran}(p_1, cout, p_{11})$

- 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:**



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

- `a`, `b` **and** `sum` **carry** n**-bit words**

- `Adder` n **specifies an** n+1**-bit adder !!!**

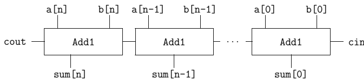
- **Example:** `Adder`(3) **specifies a** 4**-bit adder**

Specification

- **The definition of Adder is:**

$$\mathtt{Adder}(n)(a,b,cin,sum,cout) \equiv \\ (2^{n+1}\times\mathtt{Bv}(cout) + \mathtt{V}(sum\,\mathtt{[}n:0\mathtt{]}) = \\ \mathtt{V}(a\,\mathtt{[}n:0\mathtt{]}) + \mathtt{V}(b\,\mathtt{[}n:0\mathtt{]}) + \mathtt{Bv}(cin))$$

- **Diagram of implementation:**



- **By primitive recursion:**

$$\mathtt{Adder_Imp}(0)(a,b,cin,sum,cout) \equiv \\ \mathtt{Add1}(a\,\mathtt{[0]}, b\,\mathtt{[0]}, cin, sum\,\mathtt{[0]}, cout)$$

$$\mathtt{Adder_Imp}(n{+}1)(a,b,cin,sum,cout) \equiv \\ \exists c.\ \mathtt{Adder_Imp}(n)(a,b,cin,sum,c) \wedge \\ \mathtt{Add1}(a\,\mathtt{[}n{+}1\mathtt{]}, b\,\mathtt{[}n{+}1\mathtt{]}, c, sum\,\mathtt{[}n{+}1\mathtt{]}, cout)$$

Verification:

- **Prove by induction on n that for all n:**

$$\mathtt{Adder_Imp}(n)(a,b,cin,sum,cout) \\ \Rightarrow \\ \mathtt{Adder}(n)(a,b,cin,sum,cout)$$

- **Basis:**

$$\mathtt{Adder_Imp}(0)(a,b,cin,sum,cout) \\ \Rightarrow \\ \mathtt{Adder}(0)(a,b,cin,sum,cout)$$

- **Expanding definitions of Adder_Imp and Adder:**

$$\mathtt{Add1}(a\,\mathtt{[0]}, b\,\mathtt{[0]}, cin, sum\,\mathtt{[0]}, cout) \\ \Rightarrow \\ (2^{0+1}\times\mathtt{Bv}(cout) + \mathtt{V}(sum\,\mathtt{[}0:0\mathtt{]}) = \mathtt{V}(a\,\mathtt{[}0:0\mathtt{]}) + \mathtt{V}(b\,\mathtt{[}0:0\mathtt{]}) + \mathtt{Bv}(cin))$$

- **Expanding definition of Add1 and simplifying:**

$$(2\times\mathtt{Bv}(cout) + \mathtt{Bv}(sum\,\mathtt{[0]}) = \mathtt{Bv}(a\,\mathtt{[0]}) + \mathtt{Bv}(b\,\mathtt{[0]}) + \mathtt{Bv}(cin)) \\ \Rightarrow \\ (2\times\mathtt{Bv}(cout) + \mathtt{V}(sum\,\mathtt{[}0:0\mathtt{]}) = \mathtt{V}(a\,\mathtt{[}0:0\mathtt{]}) + \mathtt{V}(b\,\mathtt{[}0:0\mathtt{]}) + \mathtt{Bv}(cin))$$

- **Follows by $\mathtt{V}(w\,\mathtt{[}0:0\mathtt{]}) = \mathtt{Bv}(w\,\mathtt{[0]})$**

Induction step

- **Step:**

$$(\mathtt{Adder_Imp}(n)(a,b,cin,sum,cout) \Rightarrow \\ \mathtt{Adder}(n)(a,b,cin,sum,cout)) \\ \Rightarrow \\ (\mathtt{Adder_Imp}(n{+}1)(a,b,cin,sum,cout) \Rightarrow \\ \mathtt{Adder}(n{+}1)(a,b,cin,sum,cout))$$

- **Assume:**

$$(\mathtt{Adder_Imp}(n)(a,b,cin,sum,cout) \Rightarrow \\ \mathtt{Adder}(n)(a,b,cin,sum,cout))$$

- **Then show:**

$$\mathtt{Adder_Imp}(n{+}1)(a,b,cin,sum,cout)$$
$$= \exists c.\ \mathtt{Adder_Imp}(n)(a,b,cin,sum,c) \wedge \\ \mathtt{Add1}(a\,\mathtt{[}n{+}1\mathtt{]}, b\,\mathtt{[}n{+}1\mathtt{]}, c, sum\,\mathtt{[}n{+}1\mathtt{]}, cout)$$
$$\Rightarrow \exists c.\ \mathtt{Adder}(n)(a,b,cin,sum,c) \wedge \\ \mathtt{Add1}(a\,\mathtt{[}n{+}1\mathtt{]}, b\,\mathtt{[}n{+}1\mathtt{]}, c, sum\,\mathtt{[}n{+}1\mathtt{]}, cout)$$
$$= \exists c.\ (2^{n+1}\mathtt{Bv}(c) + \mathtt{V}(sum\,\mathtt{[}n:0\mathtt{]}) = \mathtt{V}(a\,\mathtt{[}n:0\mathtt{]}) + \mathtt{V}(b\,\mathtt{[}n:0\mathtt{]}) + \mathtt{Bv}(cin)) \\ \wedge \\ (2\mathtt{Bv}(cout) + \mathtt{Bv}(sum\,\mathtt{[}n{+}1\mathtt{]}) = \mathtt{Bv}(a\,\mathtt{[}n{+}1\mathtt{]}) + \mathtt{Bv}(b\,\mathtt{[}n{+}1\mathtt{]}) + \mathtt{Bv}(c))$$

Step continued

If:

$(A = B) \wedge (C = D)$

then it follows that ($\Rightarrow$)

$(A + 2^{n+1}C) = (B + 2^{n+1}D)$

hence:

$$\exists c.\ \overbrace{(2^{n+1}\mathtt{Bv}(c) + \mathtt{V}(sum\,\mathtt{[}n:0\mathtt{]})}^{A} = \overbrace{\mathtt{V}(a\,\mathtt{[}n:0\mathtt{]}) + \mathtt{V}(b\,\mathtt{[}n:0\mathtt{]}) + \mathtt{Bv}(cin)}^{B} \\ \wedge \\ \underbrace{(2\mathtt{Bv}(cout) + \mathtt{Bv}(sum\,\mathtt{[}n{+}1\mathtt{]})}_{C} = \underbrace{\mathtt{Bv}(a\,\mathtt{[}n{+}1\mathtt{]}) + \mathtt{Bv}(b\,\mathtt{[}n{+}1\mathtt{]}) + \mathtt{Bv}(c)}_{D}$$

$$\Rightarrow \exists c.\ \overbrace{2^{n+1}\mathtt{Bv}(c) + \mathtt{V}(sum\,\mathtt{[}n:0\mathtt{]})}^{A} + \overbrace{2^{n+1}2\mathtt{Bv}(cout) + 2^{n+1}\mathtt{Bv}(sum\,\mathtt{[}n{+}1\mathtt{]})}^{2^{n+1}C} \\ = \overbrace{\mathtt{V}(a\,\mathtt{[}n:0\mathtt{]}) + \mathtt{V}(b\,\mathtt{[}n:0\mathtt{]}) + \mathtt{Bv}(cin)}^{B} \\ + \underbrace{2^{n+1}\mathtt{Bv}(a\,\mathtt{[}n{+}1\mathtt{]}) + 2^{n+1}\mathtt{Bv}(b\,\mathtt{[}n{+}1\mathtt{]}) + 2^{n+1}\mathtt{Bv}(c)}_{2^{n+1}D}$$

$$= \exists c.\ (\mathtt{V}(sum\,\mathtt{[}n{+}1:0\mathtt{]}) + 2^{n+2}\mathtt{Bv}(cout) = \mathtt{V}(a\,\mathtt{[}n{+}1:0\mathtt{]}) + \mathtt{V}(b\,\mathtt{[}n{+}1:0\mathtt{]}) + \mathtt{Bv}(cin))$$
$$= (\mathtt{V}(sum\,\mathtt{[}n{+}1:0\mathtt{]}) + 2^{n+2}\mathtt{Bv}(cout) = \mathtt{V}(a\,\mathtt{[}n{+}1:0\mathtt{]}) + \mathtt{V}(b\,\mathtt{[}n{+}1:0\mathtt{]}) + \mathtt{Bv}(cin))$$
$$= \mathtt{Adder}(n{+}1)(a,b,cin,sum,cout))$$

Sequential Devices

- **Pure combinational adder:**

  $\texttt{Adder}(n)(a, b, cin, sum, cout) \equiv$
  $(2^{n+1} \times \texttt{Bv}(cout) + \texttt{V}(sum\texttt{[}n:0\texttt{]}) =$
  $\texttt{V}(a\texttt{[}n:0\texttt{]}) + \texttt{V}(b\texttt{[}n:0\texttt{]}) + \texttt{Bv}(cin))$

- a, b **and** sum **range over words**

- cin **and** $cout$ **range over bits (Booleans)**

- **Zero-delay adder:**

  $\texttt{Combinational_Adder}(n)(a, b, cin, sum, cout) \equiv$
  $\forall t.\ \texttt{Adder}(n)(a(t), b(t), cin(t), sum(t), cout(t))$

- a, b **and** sum **range over functions from time to words**

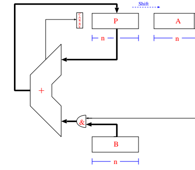
- cin **and** $cout$ **range over functions from time to bits**

- **Unit-delay adder:**

  $\texttt{Unit_Delay_Adder}(n)(a, b, cin, sum, cout) \equiv$
  $\forall t.\ \texttt{Adder}(n)(a(t), b(t), cin(t), sum(t{+}1), cout(t{+}1))$

Textbook add-shift multiplier

- **A standard add-shift multiplier:**



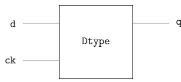
- **This can be verified directly**

- **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 degress of 'temporal granularity'**
- `Dtype` **– a fine grain trace level example**



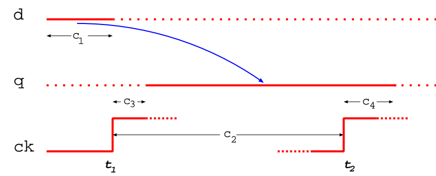
Specification of `Dtype`

If

- the clock `ck` has a rising edge at time t_1, and
- the next rising edge of `ck` is at t_2, and
- the value at `d` is stable for $\mathtt{c}_1$ units of time before t_1 ($\mathtt{c}_1$ is the *setup time*), and
- there are at least $\mathtt{c}_2$ units of time between t_1 and t_2 ($\mathtt{c}_2$ constrains the *minimum clock period*)

then

- the value at `q` will be stable from $\mathtt{c}_3$ units of time after t_1 ($\mathtt{c}_3$ is the *start time*) until $\mathtt{c}_4$ units of time after t_2 ($\mathtt{c}_4$ 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

Notes are confused!

- **Page 43:**

  $\mathtt{Rise}_1(f)(t) \equiv (f(t-1) = \mathtt{F}) \wedge (f(t) = \mathtt{T})$
- **Page 65:**

  $\mathtt{Rise}_2(f)(t) = \neg f(t) \wedge f(t+1)$
- **However:**

  $\forall f\ t.\ t > 0 \Rightarrow (\mathtt{Rise}_1(f)(t) = \mathtt{Rise}_2(f)(t-1))$

  $\forall f\ t.\ t \geq 0 \Rightarrow (\mathtt{Rise}_2(f)(t) = \mathtt{Rise}_1(f)(t+1))$
- **In Accellera standard language PSL function** $\mathtt{Rise}_1$ **is called** `Rose`

Some temporal operators in Higher Order Logic

- **Define:**

  $\mathtt{Next}(t_1,t_2)(f) \equiv t_1 < t_2 \ \wedge\ f(t_2) \ \wedge\ \forall t.\ t_1 < t \wedge t < t_2 \Rightarrow \neg f(t)$
- **Define:**

  $\mathtt{Stable}(t_1,t_2)(f) \equiv \forall t.\ t_1 \leq t \wedge t < t_2 \Rightarrow (f(t){=}f(t_1))$
- **These are raw higher order logic not temporal logic**
  - **various temporal logics are described later**

`Dtype` specification

- **Logic specification:**

$\texttt{Dtype}(\texttt{c}_1,\texttt{c}_2,\texttt{c}_3,\texttt{c}_4)(d,ck,q) \ \equiv$
$\forall t_1\ t_2.\ \texttt{Rise}_1(ck)(t_1)\ \wedge$
$\quad \texttt{Next}(t_1,t_2)(\texttt{Rise}_1(ck))\ \wedge$
$\quad (t_2{-}t_1 > \texttt{c}_2)\ \wedge$
$\quad \texttt{Stable}(t_1{-}\texttt{c}_1, t_1{+}1)(d)$
$\quad \Rightarrow$
$\quad (\texttt{Stable}(t_1{+}\texttt{c}_3, t_2{+}\texttt{c}_4)(q)\ \wedge\ (q(t_2) = d(t_1)))$

- **$\texttt{c}_1$, $\texttt{c}_2$, $\texttt{c}_3$ and $\texttt{c}_4$ are timing constants**
  - value depends on how the device is fabricated
- **Note that**

  $\texttt{Next}(t_1,t_2)(\texttt{Rise}_1(ck))$

  formed by applying

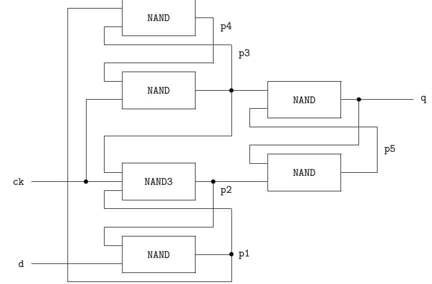
  $\texttt{Next}(t_1,t_2)$

  to the predicate

  $\texttt{Rise}_1(ck)$

Implementation

- **Can implement Dtype using NAND-gates:**



- Unit delay model

  $\texttt{NAND}(i_1,i_2,o)\ \equiv\ \forall t.\ o(t{+}1) = \neg(i_1(t) \wedge i_2(t))$

  $\texttt{NAND3}(i_1,i_2,i_3,o)\ \equiv\ \forall t.\ o(t{+}1) = \neg(i_1(t) \wedge i_2(t) \wedge i_3(t))$

- Note: modelling at the fine-grain time level

Verification

- **Dtype implementation in logic:**

$\texttt{Dtype_Imp}(d,ck,q)\ \equiv$
$\exists p_1\ p_2\ p_3\ p_4\ p_5.$
$\quad \texttt{NAND}(p_2,d,p_1)\ \wedge\ \texttt{NAND3}(p_3,ck,p_1,p_2)\ \wedge$
$\quad \texttt{NAND}(p_4,ck,p_3)\ \wedge\ \texttt{NAND}(p_1,p_3,p_4)\ \wedge$
$\quad \texttt{NAND}(p_3,p_5,q)\ \wedge\ \texttt{NAND}(q,p_2,p_5)$

- **Correctness: find δ_1, δ_2, δ_3 and δ_4 and prove:**

  $\texttt{Dtype_Imp}(d,ck,q) \Rightarrow \texttt{Dtype}(\delta_1,\delta_2,\delta_3,\delta_4)(d,ck,q)$

- **Hard!**

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

A sequential RT level example: simple parity checker

- **Input `inp`, an output `out`**
- **The nth output is `T` $\Leftrightarrow$ an even number of `T`'s input**
- **`PARITY` f n iff an even number of `T`'s in $f(1)$, ... , $f(n)$**

```
|- (∀f. PARITY f 0 = T)  ∧
   (∀n f. PARITY f (n+1) = if f(n+1) then ¬PARITY f n else PARITY f n)
```

- **Specification of the parity checking device:**

```
∀t. out t = PARITY inp t
```

- **Signals modelled as functions from numbers (times) to booleans**
- **Specification can be written as an equation between functions:**

```
out = PARITY inp
```

- **Intuitively clear that specification will be satisfied if:**

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

- **Intuition can be verified by proving:**

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

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

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

A Proof by induction

- Start with the following proof state

```
  ∀inp out.
    (out 0 = T) ∧
    (∀t. out(t+1) = if inp(t+1) then ¬(out t) else out t) ⇒
    (∀t. out t = PARITY inp t)
------------------------------------
```

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

```
∀t. out t = PARITY 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 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 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 inp t     [induction hypothesis added to assumptions]
```


Next step: unfold definition of PARITY

- Recall definition of PARITY

```
|- (∀f. PARITY f 0 = T)
   ∧
   ∀n f. PARITY f (n+1) = if f(n+1) then ¬PARITY f n else PARITY f n
```

- Unfolding (rewriting with) the definition of PARITY in

```
out 0 = PARITY 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 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 inp t
```

- Yields

```
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 inp t else PARITY 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 inp t
```


Goal now easily proved

- Proof state from last slide

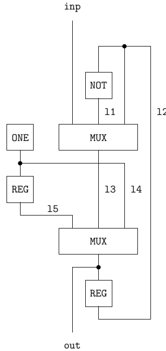
```
out 0 = T
------------------------------------
  0.  out 0 = T
  1.  ∀t. out(t+1) = if inp(t+1) then ¬(out t) else out t

out(t+1) = if inp(t+1) then ¬PARITY inp t else PARITY 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 inp t
```

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

```
UNIQUENESS_LEMMA =
 |- ∀inp out.
      (out 0 = T) ∧
      (∀t. out(t+1) = if inp(t+1) then ¬(out t) else out t) ⇒
      ∀t. out t = PARITY inp t
```

Implementation

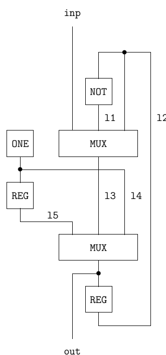
- Assume registers 'power up' storing F
- Thus the output at time 0 cannot be taken directly from a register
  - because the output of the parity checker at time 0 is specified to be T



Components

```
|- ONE out = ∀t. out t = T

|- NOT(inp, out) = ∀t. out t = ¬(inp t)

|- MUX(sw,in1,in2,out) = ∀t. out t = if sw t then in1 t else in2 t

|- REG(inp,out) = ∀t. out t = if (t=0) then F else inp(t-1)
```


Implementation in HOL




```
|-  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)
```


Verification

- The following theorem will eventually be proved:

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

- First prove a lemma (then theorem follows from UNIQUENESS_LEMMA)
- The lemma (PARITY_LEMMA):

```
∀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

```
 out 0 = T
 ------------------------------------
   0.  ∀t. l1 t = ¬l2 t
   1.  ∀t. l3 t = if inp t then l1 t else l2 t
   2.  ∀t. l2 t = if t = 0 then F else out(t - 1)
   3.  ∀t. l4 t = T
   4.  ∀t. l5 t = if t = 0 then F else l4 (t - 1)
   5.  ∀t. out t = if l5 t then l3 t else l4 t

 out(t+1) = if inp(t+1) then ¬(out t) else out t
 ------------------------------------
   0.  ∀t. l1 t = ¬l2 t
   1.  ∀t. l3 t = if inp t then l1 t else l2 t
   2.  ∀t. l2 t = if t = 0 then F else out(t - 1)
   3.  ∀t. l4 t = T
   4.  ∀t. l5 t = if t = 0 then F else l4 (t - 1)
   5.  ∀t. out t = if l5 t then l3 t else l4 t
```

Proof continued

- Consider the t=0 case first

```
out 0  =  T
------------------------------------
  0.  ∀t. l1 t = ¬l2 t
  1.  ∀t. l3 t = if inp t then l1 t else l2 t
  2.  ∀t. l2 t = if t = 0 then F else out(t - 1)
  3.  ∀t. l4 t  =  T
  4.  ∀t. l5 t = if t = 0 then F else l4 (t - 1)
  5.  ∀t. out t = if l5 t then l3 t else l4 t
```

- Easily follows (see stuff in blue)
- Now consider t+1 case

```
out(t+1) = if inp(t+1) then ¬(out t) else out t
------------------------------------
  0.  ∀t. l1 t = ¬l2 t
  1.  ∀t. l3 t = if inp t then l1 t else l2 t
  2.  ∀t. l2 t = if t = 0 then F else out(t - 1)
  3.  ∀t. l4 t = T
  4.  ∀t. l5 t = if t = 0 then F else l4 (t - 1)
  5.  ∀t. out t = if l5 t then l3 t else l4 t
```

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

  ∀t. out t = if l5 t then l3 t else l4 t

- See next slide ...

Proof continued

- Use assumption 5 to expand blue term, but not red terms

```
out(t + 1) = if inp(t+1) then ¬out t else out t
------------------------------------
  0.  ∀t. l1 t = ¬l2 t
  1.  ∀t. l3 t = if inp t then l1 t else l2 t
  2.  ∀t. l2 t = if t = 0 then F else out(t - 1)
  3.  ∀t. l4 t = T
  4.  ∀t. l5 t = if t = 0 then F else l4 (t - 1)
  5.  ∀t. out t = if l5 t then l3 t else l4 t
```

- Result is

```
(if l5 (t+1) then l3 (t+1) else l4 (t+1)) =
(if inp(t+1) then ¬(out t) else out t)
------------------------------------
  0.  ∀t. l1 t = ¬l2 t
  1.  ∀t. l3 t = if inp t then l1 t else l2 t
  2.  ∀t. l2 t = if t = 0 then F else out(t - 1)
  3.  ∀t. l4 t = T
  4.  ∀t. l5 t = if t = 0 then F else l4 (t - 1)
  5.  ∀t. out t = if l5 t then l3 t else l4 t
```

- Goal follows from assumptions with a bit of calculation

Combining lemmas

- Call lemma just proved PARITY_LEMMA, so

```
PARITY_LEMMA =
 |- ∀inp out.
      PARITY_IMP (inp,out) ⇒
      (out 0 = T) ∧
      ∀t. out(t+1) = if inp(t+1) then ¬(out t) else out t
```

- Recall

```
UNIQUENESS_LEMMA =
 |- ∀inp out.
      (out 0 = T) ∧
      (∀t. out(t+1) = if inp(t+1) then ¬(out t) else out t)
      ⇒
      ∀t. out t = PARITY inp t
```

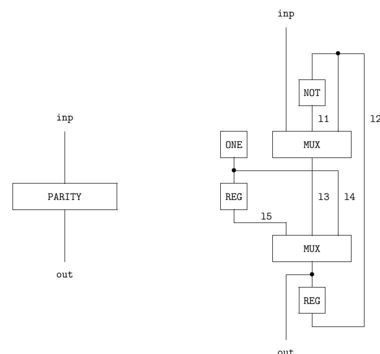
- Hence by transitivity of ⇒

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

- PARITY_IMP used abstract registers REG
- Next: make model more concrete by using clocked Dtype

Review

- Specification: ∀t. out t = PARITY inp t
- Equivalent equation between functions: out = PARITY inp



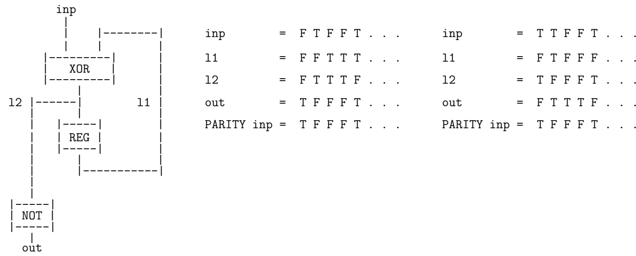

```
|- PARITY_IMP(inp,out) =
     ∃l1 l2 l3 l4 l5.
       NOT(l2,l1) ∧ MUX(inp,l1,l2,l3) ∧ REG(out,l2) ∧
       ONE l4     ∧ REG(l4,l5)        ∧ MUX(l5,l3,l4,out)
```

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

An incorrect implementation of the parity checker

```
|- (∀f. PARITY f 0 = T)
   ∧
   ∀n f. PARITY f (n+1) = if f(n+1) then ¬PARITY f n else PARITY f n
```

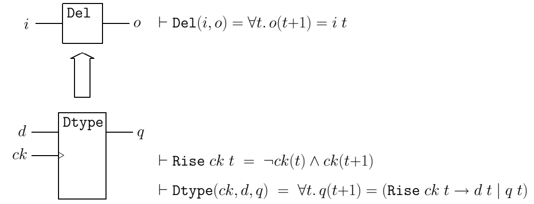
- **The following implementation doesn't work**



inp	= F T F F T . . .	inp	= T T F F T . . .
l1	= F F T T T . . .	l1	= F T F F F . . .
l2	= F T T T F . . .	l2	= T F F F T . . .
out	= T F F F T . . .	out	= F T T T F . . .
PARITY inp	= T F F F T . . .	PARITY inp	= T F F F T . . .

Temporal refinement

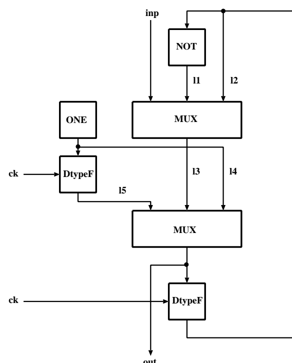
- PARITY_IMP used abstract registers REG
- **Next: make model more concrete by using clocked** Dtype
- **Recall the (course grained) trace level model of a** Dtype**:**



- **Need a version of** Dtype **that powers up storing** F

```
DtypeF(ck,d,q) = (q 0 = F) ∧ Dtype(ck,d,q)
```

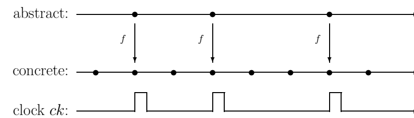

Trace level version of the Parity device




```
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)
```


Formulating Correctness

- **A mapping between time-scales:**



- **Define the temporal abstraction functions:**

  $(s$ when $P)(n) =$ *value of s at the concrete time t when* P *true for nth time*

  $\vdash$ Timeof P $n =$ *the concrete time t when* P *true for nth time*

  $\vdash s$ when $P = s \circ ($Timeof $P)$

- **From Melham's Theorem:**

  $\vdash \forall ck.\ \texttt{Inf}(\texttt{Rise}\ ck) \Rightarrow$
  $\quad \forall d\ q.\ \texttt{DtypeF}(ck, d, q) \Rightarrow \texttt{REG}(d\ \texttt{when}\ (\texttt{Rise}\ ck), q\ \texttt{when}\ (\texttt{Rise}\ ck))$

- Inf P **means "P true infinitely often"**

  Inf $P = \forall t.\ \exists t'.\ t' > t\ \wedge\ P\ t'$

Digression on defining `Timeof`

- How do we define the temporal abstraction function:

  ⊢ `Timeof P` n = *the concrete time* t_c *such that* `P` *true for nth time*

- What if there is no time such that `P` true for nth time
  - for example, if P is never true

- Need to actually define:

  ⊢ `Timeof P` n = *the time* t_c *such that* `P` *true for nth time, if such a time exists*

- But then what is `Timeof P` n if no such time exists?

Hilbert's epsilon-operator to the rescue

- $\epsilon x.\ t[x]$ is an epsilon-term
- The meaning of $\epsilon x.\ t[x]$ is specified by an axiom:

  $\forall P.\ (\exists x.\ P\ x) \Rightarrow P(\epsilon x.\ P\ x)$

- $\epsilon x.\ t[x]$ denotes some value, v say, such that $t[v]$, if $\exists t.\ t[x]$
- $\epsilon x.\ t[x]$ denotes some arbitrary value if $\forall t.\ \neg t[x]$
  - of the type of $t[x]$
  - all types are assumed non-empty
- The ϵ-operator builds the Axiom of Choice into the logic

Definition of `Timeof`

- Recall the `Next` operator

  ```
  Next t1 t2 sig = t1<t2 ∧ sig t2 ∧ ∀t. t1<t ∧ t<t2 ⇒ ¬(sig t)
  ```

- Define `IsTimeof n sig t`
  to mean "`t` is when `sig` is true for the `n`-th time"

  ```
  (IsTimeof 0 sig t = (sig t ∧ ∀t'. t'<t ⇒ ¬(sig t')))
  ∧
  (IsTimeof (n+1) sig t = ∃t'. IsTimeof n sig t' ∧ Next t' t sig)
  ```

- Define `Timeof` using ϵ-operator and `IsTimeof`

  ```
  Timeof sig n = εt. IsTimeof n sig t
  ```

- `IsTimeof` and `Timeof` are higher-order total functions

Temporal abstraction

- Define f`@ck` to be signal f abstracted on rising edges of `ck`

  ```
  |-  f@ck = f when (Rise ck)
  ```

- Recall definition of `REG`

  ```
  |- REG(inp,out) = ∀t. out t = if (t=0) then F else inp(t-1)
  ```

- It follows easily that

  ```
  |- REG(inp,out) = (out 0 = F) ∧ Del(inp,out)
  ```

- The properties below also follow (why?)

  ```
  |- Inf(Rise ck) ⇒ DtypeF(ck,d,q) ⇒ REG(d@ck, q@ck)

  |- MUX(switch, i1, i2, out)
     ⇒
     MUX(switch@ck, i1@ck, i2@ck, out@ck)

  |- NOT(inp, out) ⇒ NOT(inp@ck, out@ck)

  |- ONE out ⇒ ONE(out@ck)
  ```

- **Hint:** $\vdash\ \forall f.\ (\forall x.\ P(x)) \Rightarrow (\forall x.\ P(f(x)))$ take $f = x \mapsto x$`@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)
```

- use $(A \Rightarrow B) \wedge (\cdots\ A\ \cdots)\ \Rightarrow\ (\cdots\ B\ \cdots)$
- then use $(A \Rightarrow B) \wedge (\exists l.\ A)\ \Rightarrow\ (\exists l.\ B)$
- then use $(\exists l.\ \cdots\ l$ `on ck` $\cdots)\ \Rightarrow\ (\exists l.\ \cdots\ l\ \cdots)$

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@cl`

```
|- 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)
```

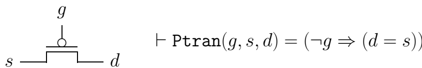
- **Hence, by transitivity of** $\Rightarrow$

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

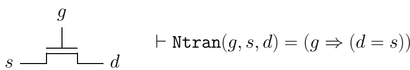
- **This is a typical correctness result using temporal abstraction**

NEW TOPIC: modelling transistors

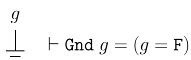
- **Recall simple switch model of CMOS**



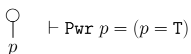
$\vdash$ `Ptran`$(g, s, d) = (\neg g \Rightarrow (d = s))$



$\vdash$ `Ntran`$(g, s, d) = (g \Rightarrow (d = s))$



$\vdash$ `Gnd` $g = (g =$ `F`$)$

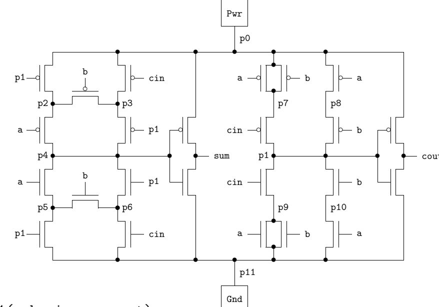


$\vdash$ `Pwr` $p = (p =$ `T`$)$

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

The simple adder example

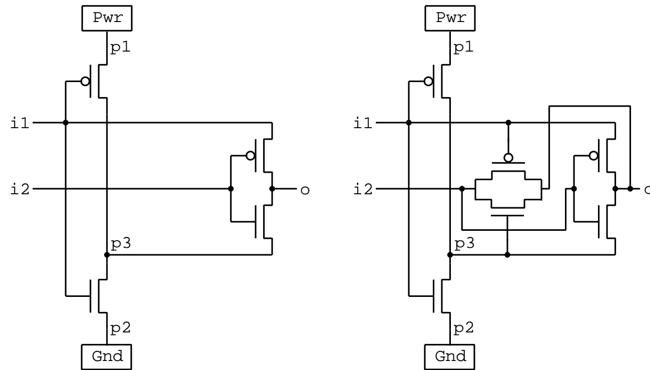
- **This example shows non-obvious examples can be analysed**




```
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,p4)   ∧ Ntran(a,p4,p5)    ∧
         Ntran(p1,p4,p6)    ∧ Ntran(b,p5,p6)    ∧ Ntran(p1,p5,p11)  ∧
         Ntran(cin,p6,p11)  ∧ Ptran(a,p0,p7)    ∧ Ptran(b,p0,p7)    ∧
         Ptran(a,p0,p8)     ∧ Ptran(cin,p7,p1)  ∧ Ptran(b,p8,p1)    ∧
         Ntran(cin,p1,p9)   ∧ Ntran(b,p1,p10)   ∧ Ntran(a,p9,p11)   ∧
         Ntran(b,p9,p11)    ∧ Ntran(a,p10,p11)  ∧ Pwr(p0)           ∧
         Ptran(p4,p0,sum)   ∧ Ntran(p4,sum,p11) ∧ 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

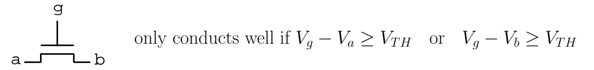
- Compare

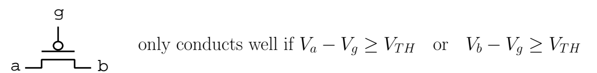


- Equivalent in simple switch model!

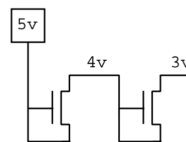
How transistors work

- Transistors conduct if there is a big enough voltage difference, V_{TH} say, between gate and source/drain



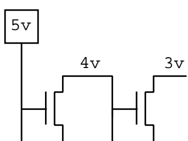


- If $V_g = V_a$ there is a voltage drop of about V_{TH}
- Example: 'hi' is $5v$, 'low' is $0v$



- Weak output may not be able to switch transistors

What happens in the Simple Switch Model



- From the definitions

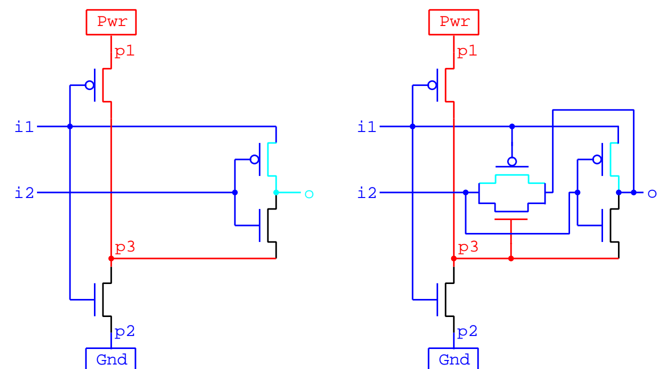
```
|- ∀p. Pwr p = (p = T)

|- ∀g a b. Ntran (g,a,b) = g ⇒ (a = b)

|- ∀out. Bad out =
          ∃l1 l2. Pwr l1 ∧ Ntran (l1,l1,l2) ∧ Ntran (l2,l2,out)
```

- It follows that

```
|- ∀out. Bad out = out
```


Consider two Xors when both inputs are F

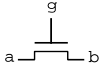
- Compare



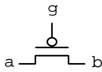
- Bad design has weak output
- Good design has strong output
- Need a better model to distinguish the designs

Difference switching model (Mike Fourman)

- Don't identify boolean values and signal values
- Consider a type of values containing `Hi`, `Lo` and other values

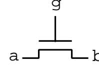



```
Ntran(g,a,b) = ((g=Hi) ∧ (a=Lo) ⇒ (b=Lo))
               ∧
               ((g=Hi) ∧ (b=Lo) ⇒ (a=Lo))
```

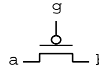



```
Ptran(g,a,b) = ((g=Lo) ∧ (a=Hi) ⇒ (b=Hi))
               ∧
               ((g=Lo) ∧ (b=Hi) ⇒ (a=Hi))
```

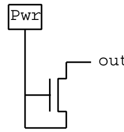

More compact definitions




```
Ntran(g,a,b) = (g=Hi) ⇒ ((a=Lo) = (b=Lo))
```

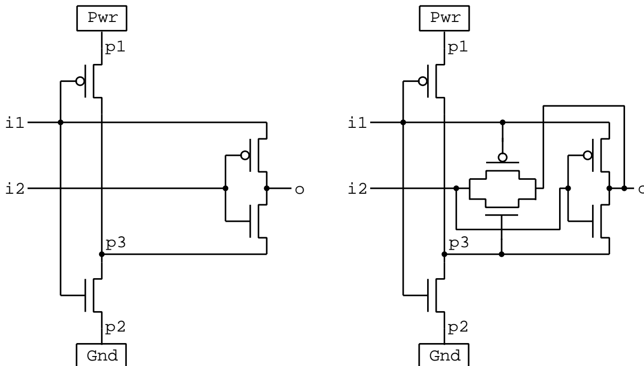



```
Ptran(g,a,b) = (g=Lo) ⇒ ((a=Hi) = (b=Hi))
```



this is now equivalent to ¬(`out` = `Lo`)

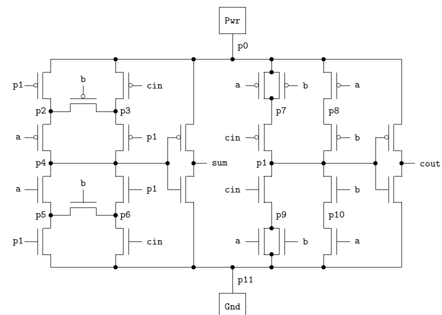
Good and bad Xors now distinguished




```
((i1=Hi) ∧ (i2=Hi) ⇒ (out = Lo))  ∧
((i1=Hi) ∧ (i2=Lo) ⇒ (out = Hi))  ∧
((i1=Lo) ∧ (i2=Hi) ⇒ ¬(out = Lo)) ∧
((i1=Lo) ∧ (i2=Lo) ⇒ ¬(out = Hi))
```

```
((i1=Hi) ∧ (i2=Hi) ⇒ (out = Lo)) ∧
((i1=Hi) ∧ (i2=Lo) ⇒ (out = Hi)) ∧
((i1=Lo) ∧ (i2=Hi) ⇒ (out = Hi)) ∧
((i1=Lo) ∧ (i2=Lo) ⇒ (out = Lo))
```


Earlier examples still work



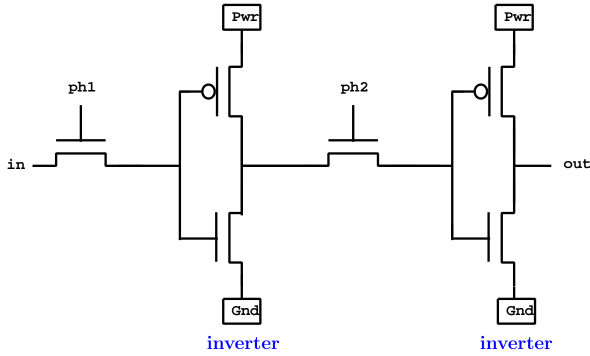
- Define

```
|- Strong v = ((v = Hi) ∨ (v = Lo))
|- (TBv Hi = 1) ∧ (TBv Lo = 0)
|- TAdd1Spec(a,b,cin,sum,cout) =
     (2*(TBv cout) + TBv sum = TBv a + TBv b + TBv cin)
```

- Then it follows that

```
|- Strong a ∧ Strong b ∧ Strong cin
   ⇒
   TAdd1Imp(a,b,cin,sum,cout) ⇒ TAdd1Spec(a,b,cin,sum,cout)
                                 ∧ Strong sum ∧ Strong cout
```

Sequential shift register

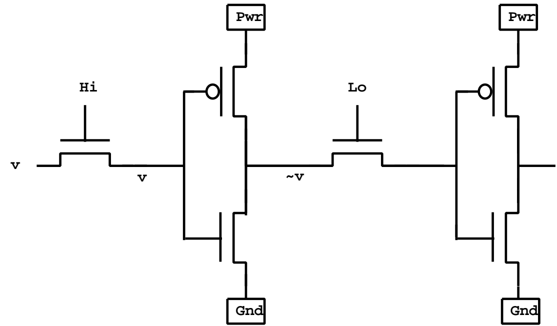


- Switch models only allow us to deduce

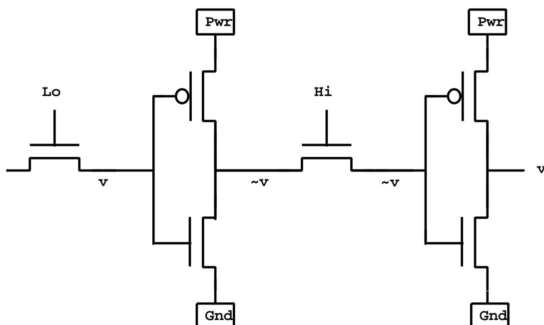
  (ph1=Hi) ∧ (ph2=Hi) ⇒ ((in=Hi) ⇒ (out=Hi)) ∧ ((in=Lo) ⇒ (out=Lo))

- Actual behaviour is a shift register
  - for simplicity threshold effects ignored in what follows

Phase 1: ph1=Hi and ph2=Lo

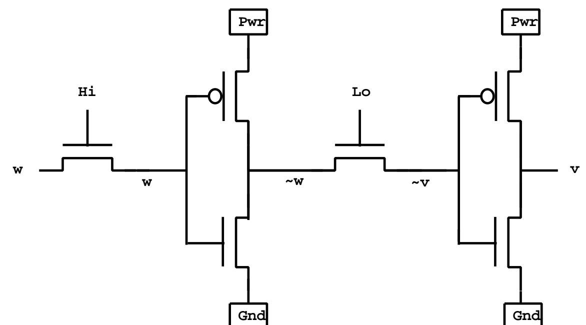


Phase 2: ph1=Lo and ph2=Hi



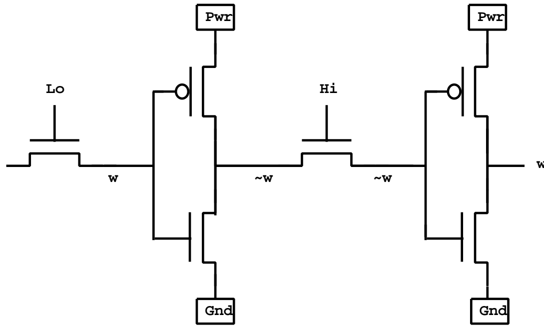

```
(ph1 t    = Hi) ∧ (ph2 t    = Lo) ∧
(ph1(t+1) = Lo) ∧ (ph2(t+1) = Hi)
⇒
(out(t+1) = in t)
```


Phase 3: ph1=Hi and ph2=Lo



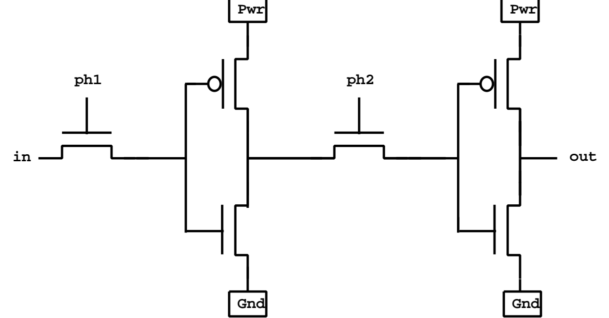

```
(ph1 t    = Hi) ∧ (ph2 t    = Lo) ∧
(ph1(t+1) = Lo) ∧ (ph2(t+1) = Hi) ∧
(ph1(t+2) = Hi) ∧ (ph2(t+2) = Lo)
⇒
(out(t+2) = in t)
```

Phase 4: ph1=Lo and ph2=Hi




```
(ph1(t+2) = Hi) ∧ (ph2(t+2) = Lo) ∧
(ph1(t+3) = Lo) ∧ (ph2(t+3) = Hi)
⇒
(out(t+3) = in(t+2))
```


Characterisation of behaviour




```
(ph1 t    = Hi) ∧ (ph2 t    = Lo) ∧
(ph1(t+1) = Lo) ∧ (ph2(t+1) = Hi)
⇒
(out(t+1) = in t)

(ph1 t    = Hi) ∧ (ph2 t    = Lo) ∧
(ph1(t+1) = Lo) ∧ (ph2(t+1) = Hi) ∧
(ph1(t+2) = Hi) ∧ (ph2(t+2) = Lo)
⇒
(out(t+2) = in t)
```

- `out(t+3)` **value follows by** `t` ↦ `t+2` **in first property**

Unidirectional sequential model

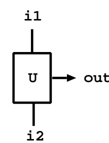
- **Four values:** `Hi`, `Lo`, `Fl` **(‘floating’),** `X` **(unknown/error)**

```
|- ¬(Hi = Lo) ∧ ¬(Lo = Hi) ∧
   ¬(Hi = Fl) ∧ ¬(Fl = Hi) ∧
   ¬(Lo = Fl) ∧ ¬(Fl = Lo)

|- Strong v = ((v = Hi) ∨ (v = Lo))

|- Float v = (v = Fl)
```

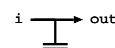
- **Join operator:** `U`

```
|- v1 U v2 = if Strong v1 ∧ Float v2
              then v1 else
             if Float v1 ∧ Strong v2
              then v2 else
             if Float v1 ∧ Float v2
              then Fl else X

|- Join(i1,i2,out) = ∀t. out t = (i1 t) U (i2 t)
```


Signals are functions of time

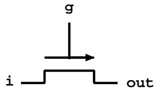



```
|- Join(i1,i2,out) = ∀t. out t = (i1 t) U (i2 t)

|- Pwr out = ∀t. out t = Hi

|- Gnd out = ∀t. out t = Lo
```

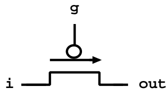



```
|- Cap(i,out) = ∀t. out t = if Strong(i t) then i t else
                            if t=0 then X else
                            if Float(i t) ∧ Strong(i(t-1)) then i(t-1)
                            else Fl
```

Unidirectional sequential transistor models

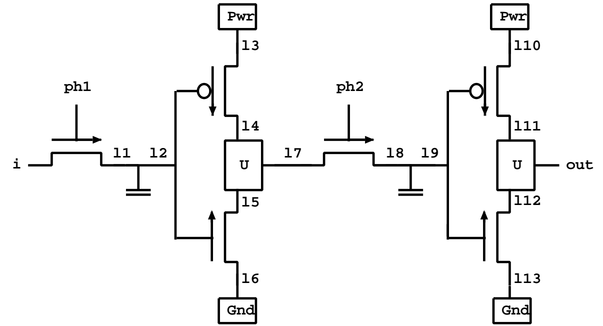



```
|- Nswitch(g,i,out) = ∀t. out t = if g t = Hi then i t else
                                  if (g t = Lo ) ∨ (i t = Fl) then Fl
                                  else X
```




```
|- Pswitch(g,i,out) = ∀t. out t = if g t = Lo then i t else
                                  if (g t = Hi) ∨ (i t = Fl) then Fl
                                  else X
```

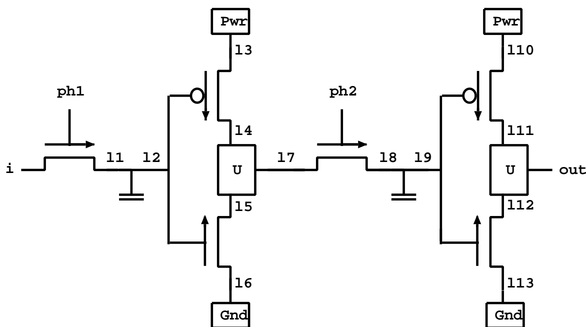

Sequential shift register model




```
|- ShiftReg(i,out,ph1,ph2) =
    ∃l1 l2 l3 l4 l5 l6 l7 l8 l9 l10 l11 l12 l13.
      Nswitch(ph1,i,l1) ∧ Cap(l1,l2) ∧
      Pwr l3 ∧ Pswitch(l2,l3,l4) ∧ Nswitch(l2,l6,l5) ∧ Gnd l6 ∧
      Join(l4,l5,l7) ∧ Nswitch(ph2,l7,l8) ∧ Cap(l8,l9) ∧
      Pwr l10 ∧ Pswitch(l9,l10,l11) ∧ Nswitch(l9,l13,l12) ∧ Gnd l13 ∧
      Join(l11,l12,out)
```

- **Lots more state variables than in combinational switch model!**

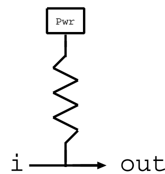
Correctness of sequential shift register model




```
|- ShiftReg(in,out,ph1,ph2) ∧ Strong(in t) ∧
   (ph1 t     = Hi) ∧ (ph2 t     = Lo) ∧
   (ph1(t+1) = Lo) ∧ (ph2(t+1) = Hi)
   ⇒
   (out(t+1) = in t)

|- ShiftReg(in,out,ph1,ph2) ∧ Strong(in t) ∧
   (ph1 t     = Hi) ∧ (ph2 t     = Lo) ∧
   (ph1(t+1) = Lo) ∧ (ph2(t+1) = Hi) ∧
   (ph1(t+2) = Hi) ∧ (ph2(t+2) = Lo)
   ⇒
   (out(t+2) = in t)
```


A model of NMOS

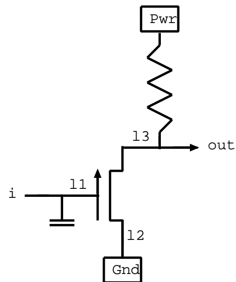
- Need a new component: pullup




```
|- Pu(i,out) = ∀t. out t = if Float(i t) then Hi else i t
```

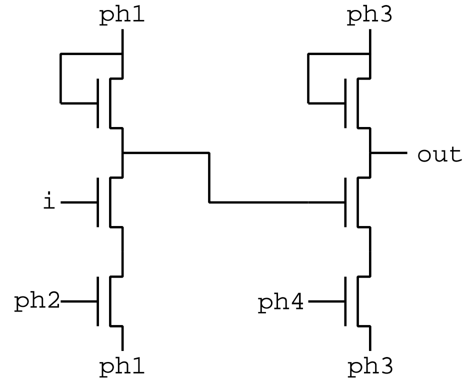
- If `i` is strong then `out = i`
- If `i` is floating then `out = Hi`

NMOS inverter




```
|- Inv(i,out) =
    ∃l1 l2 l3.
     Cap(i, l1) ∧ Gnd l2 ∧ Nswitch(l1,l2,l3) ∧ Pu(l3,out)

|- Inv(i,out)
    ⇒
    ((i t = Hi) ⇒ (out t = Lo)) ∧
    ((i t = Lo) ⇒ (out t = Hi)) ∧
    ((i(t+1) = Fl) ⇒ (((i t = Hi) ⇒ (out(t+1) = Lo))
                      ∧
                      ((i t = Lo) ⇒ (out(t+1) = Hi))))
```

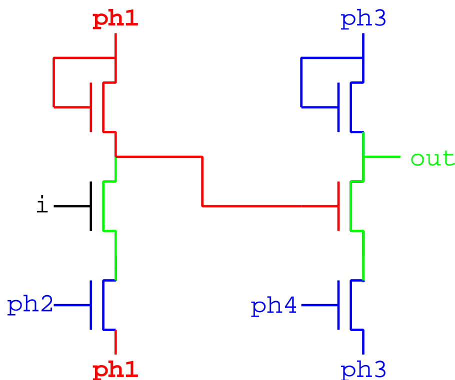

Four phase NMOS shift register




```
|- FourPhaseShiftReg(i,out,ph1,ph2,ph3,ph4)
   ∧ Strong(i(t+1))
   ∧ (ph1 t    =Hi) ∧ (ph2 t    =Lo) ∧ (ph3 t    =Lo) ∧ (ph4 t    =Lo)
   ∧ (ph1(t+1)=Lo) ∧ (ph2(t+1)=Hi) ∧ (ph3(t+1)=Lo) ∧ (ph4(t+1)=Lo)
   ∧ (ph1(t+2)=Lo) ∧ (ph2(t+2)=Lo) ∧ (ph3(t+2)=Hi) ∧ (ph4(t+2)=Lo)
   ∧ (ph1(t+3)=Lo) ∧ (ph2(t+3)=Lo) ∧ (ph3(t+3)=Lo) ∧ (ph4(t+3)=Hi)
   ⇒
   (out(t+3) = i(t+1))
```


Phase 1 (precharge internal node)

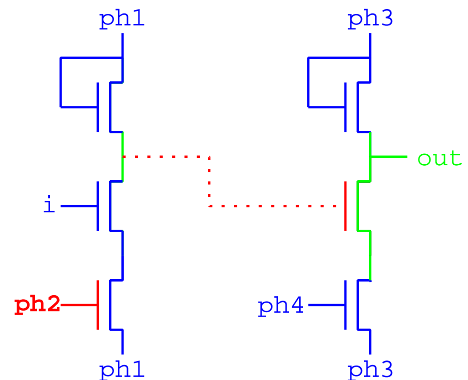
Colour scheme: Hi, Lo, Fl; threshold effects ignored




```
(ph1 t = Hi) ∧ (ph2 t = Lo) ∧ (ph3 t = Lo) ∧ (ph4 t = Lo)
```

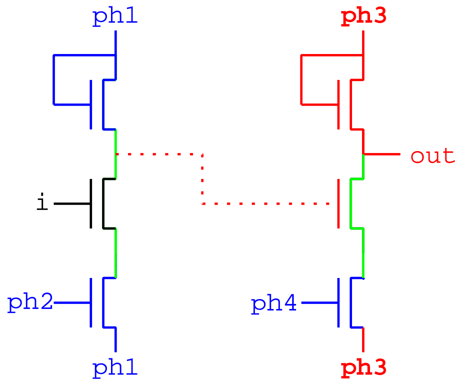

Phase 2 (input Lo, retain precharge)

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




```
(ph1(t+1)=Lo) ∧ (ph2(t+1)=Hi) ∧ (ph3(t+1)=Lo) ∧ (ph4(t+1)=Lo)

(i(t+1) = Lo)
```

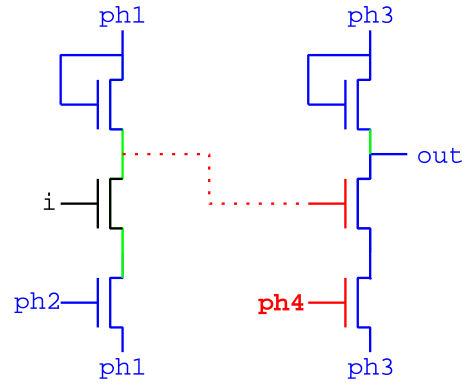
Phase 3 (precharge out, internal node retains value)



(ph1(t+2)=Lo) ∧ (ph2(t+2)=Lo) ∧ (ph3(t+2)=Hi) ∧ (ph4(t+2)=Lo)

(out(t+2) = Hi)

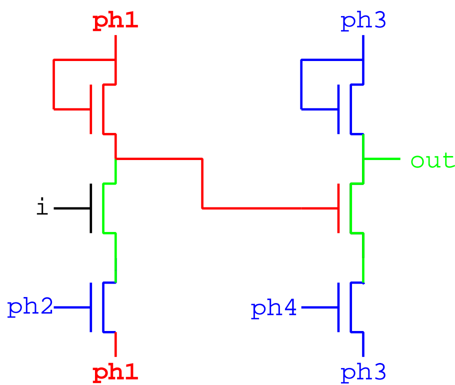
Phase 4 (kill precharge)



(ph1(t+3)=Lo) ∧ (ph2(t+3)=Lo) ∧ (ph3(t+3)=Lo) ∧ (ph4(t+3)=Hi)

(out(t+3) = Lo)

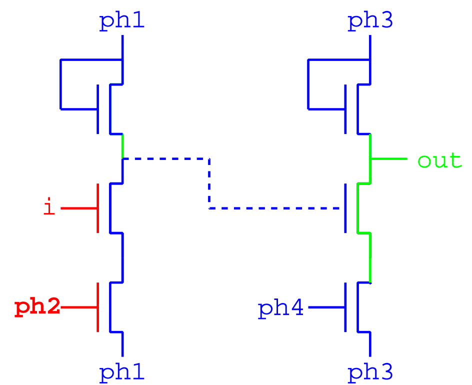
Phase 1 (precharge internal node)



(ph1 t = Hi) ∧ (ph2 t = Lo) ∧ (ph3 t = Lo) ∧ (ph4 t = Lo)

- out retains previous value

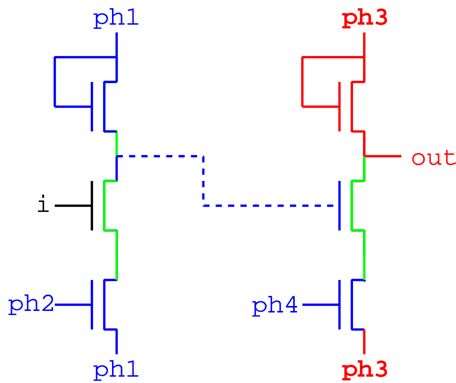
Phase 2 (input Hi, kill precharge)



(ph1(t+1)=Lo) ∧ (ph2(t+1)=Hi) ∧ (ph3(t+1)=Lo) ∧ (ph4(t+1)=Lo)

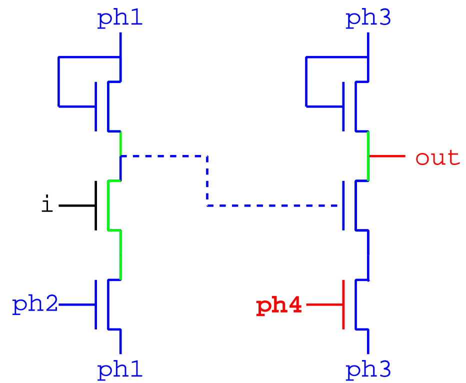
(i(t+1) = Hi)

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



`(ph1(t+2)=Lo) ∧ (ph2(t+2)=Lo) ∧ (ph3(t+2)=Hi) ∧ (ph4(t+2)=Lo)`

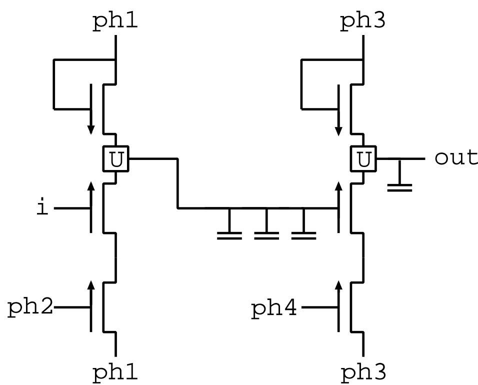
Phase 4 (`out` retains precharge)



`(ph1(t+3)=Lo) ∧ (ph2(t+3)=Lo) ∧ (ph3(t+3)=Lo) ∧ (ph4(t+3)=Hi)`

`out(t+3) = Hi`

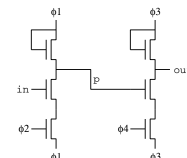
Four phase NMOS shift register model




```
|- FourPhaseShiftReg(i,out,ph1,ph2,ph3,ph4) =
   ∃l1 l2 l3 l4 l5 l6 l7 l8 l9 l10 l11.
    Nswitch(ph1,ph1,l1) ∧ Nswitch(i,l3,l2) ∧ Nswitch(ph2,ph1,l3) ∧
    Join(l1,l2,l4) ∧ Cap(l4,l5) ∧ Cap(l5,l6) ∧ Cap(l6,l7) ∧
    Nswitch(ph3,ph3,l8) ∧ Nswitch(l7,l10,l9) ∧ Nswitch(ph4,ph3,l10) ∧
    Join(l8,l9,l11) ∧ Cap(l11,out)
```


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




```
|- FourPhaseShiftReg(in,out,ph1,ph2,ph3,ph4)
   ∧ Strong(in(t+1))
   ∧ (ph1 t   =Hi) ∧ (ph2 t   =Lo) ∧ (ph3 t   =Lo) ∧ (ph4 t   =Lo)
   ∧ (ph1(t+1)=Lo) ∧ (ph2(t+1)=Hi) ∧ (ph3(t+1)=Lo) ∧ (ph4(t+1)=Lo)
   ∧ (ph1(t+2)=Lo) ∧ (ph2(t+2)=Lo) ∧ (ph3(t+2)=Hi) ∧ (ph4(t+2)=Lo)
   ∧ (ph1(t+3)=Lo) ∧ (ph2(t+3)=Lo) ∧ (ph3(t+3)=Lo) ∧ (ph4(t+3)=Hi)
   ⇒ (out(t+3) = in(t+1))
```

An earlier slide on Hoare logic for hardware

- Would like a generalised Hoare Logic specification:

⊢ {*If environment ensures always that:* DONE=0 ⇒ Load=0
and if Load *is set to* 1 *when:* $\texttt{In1} = x \wedge \texttt{In2} = y$}

```
FOREVER
 IF Load=1
  THEN X:=In1; Y:=In2; DONE:=0; R:=X; Q:=0
  ELSE IF Y≤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 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:* $x = \texttt{R} + y \times \texttt{Q}$}

- Stuff in red needs **Temporal Logic**

Specification and Verification II

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

Formal verification based debugging	Proof of correctness
proof failure ⇒ bugs	proof success ⇒ assurance
practical for real code	expensive and often impractical
unsound models OK	needs high fidelity models
unsafe implementation methods OK	important to use trustworthy tools

- 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

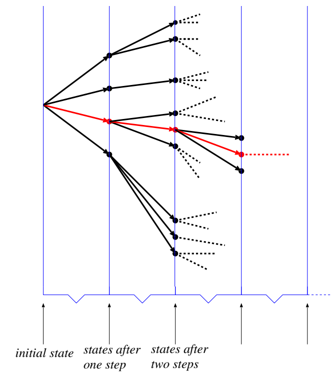
NEW TOPIC: Model Checking

- 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** $\mathcal{B}$
  - $\mathcal{B} : states \rightarrow bool$
  - $\mathcal{B}\ s$ **means** s **is an initial state**
- **State transition relation:** $\mathcal{R}$
  - $\mathcal{R} : states \times states \rightarrow bool$
  - $\mathcal{R}(s, s')$ **means** s' **a successor to** s

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



Example: single state machine

- **State transition function:** δ

  $\delta : states \times inputs \rightarrow states$
- **Define state transition relation:**

  $\mathcal{R}(s, s') = \exists inp.\ s' = \delta(s, inp)$
- **Deterministic machine:**
  - non-deterministic transition relation
  - existential quantification over inputs
  - so called "input non-determinism"

Example: n machines in parallel

- **Assume** n **state variables**
  - $states = states_1 \times \cdots \times states_n$
  - $\vec{v} = (v_1, \ldots, v_n)$
- **Assume** n **transition functions**

  $\delta_i : states \times inputs \rightarrow states_i \qquad (1 \leq i \leq n)$
- **Note: each machine** δ_i **reads all inputs and states**
- **An** $\mathcal{R}$**-step is a non-deterministically chosen step of one machine**

$$
\begin{array}{l}
\mathcal{R}(\vec{v}, \vec{v}') = \\
\ \exists inp. \\
\quad v_1' = \delta_1(\vec{v}, inp) \ \wedge\ v_2' = v_2 \ \wedge \cdots \wedge\ v_n' = v_n \\
\quad \vee \\
\quad v_1' = v_1 \ \wedge\ v_2' = \delta_2(\vec{v}, inp) \ \wedge \cdots \wedge\ v_n' = v_n \\
\quad \vee \\
\quad \vdots \\
\quad \vee \\
\quad v_1' = v_1 \ \wedge\ v_2' = v_2 \ \wedge \cdots \wedge\ v_n' = \delta_n(\vec{v}, inp)
\end{array}
$$

- **Asynchronous parallel composition**

Explicit state property checking

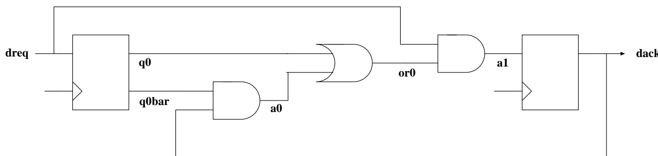
- Goal: check some property P holds of all reachable states
  - e.g. $P(s)$ means s has no errors
- Represent sets of states somehow
- Start with $S_0 = \{s \mid \mathcal{B}\ s\}$
- Iteratively compute with $S_{n+1} = S_n \cup \{s \mid \exists u.\ u \in S_n \wedge \mathcal{R}(u,s)\}$
- Note $S_0 \subseteq S_1 \subseteq S_2 \subseteq \cdots$
  - 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 $v = b_1 \vee v = b_2 \vee \ldots \vee v = b_n$
  - $b_1, b_2, \ldots, b_n$ 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 $\vdash\ (v = b_1 \vee v = b_2 \vee \ldots \vee v = b_n)[b/v]$
- A set of states
  $\{(b_{11}, \ldots, b_{1m}), \ldots, (b_{n1}, \ldots, b_{nm})\}$
  is represented by a formula with m boolean variables:
  $(v_1 = b_{11} \wedge \ldots \wedge v_m = b_{1m}) \vee \ldots \vee (v_1 = b_{n1} \wedge \ldots \wedge v_m = b_{nm})$
- To test if $(b_1, \ldots, b_m)$ is in the set,
  just evaluate the formula with $v_1 = b_1$, ..., $v_m = b_m$, i.e. evaluate:
  $((v_1 = b_{11} \wedge \ldots \wedge v_m = b_{1m}) \vee \ldots \vee (v_1 = b_{n1} \wedge \ldots \wedge v_m = b_{nm}))[(b_1, \ldots, b_m)/(v_1, \ldots, v_m)]$

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) ∧ (dack' = dreq ∧ (q0 ∨ (¬q0 ∧ dack)))
```

- Transition relation equivalent to:

```
(q0' = dreq) ∧ (dack' = dreq ∧ (q0 ∨ dack))
```

- Define $\mathcal{R}_{\texttt{RECEIVER}}$ by:

```
R_RECEIVER((dreq,q0,dack),(dreq',q0',dack')) =
  (q0' ⇔ dreq) ∧ (dack' ⇔ dreq ∧ (q0 ∨ dack))
```

- `dreq'` unconstrained, hence non-determinism

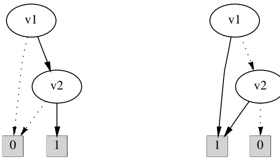
Symbolic reachability: sets of states are formulas

- Condition for a state s to be reachable
  in one $\mathcal{R}$-step from a state in $\mathcal{B}$
  $\exists u.\ \mathcal{B}\ u \wedge \mathcal{R}(u,s)$
- Define `ReachBy` $n\ \mathcal{R}\ \mathcal{B}$ to be set of states reachable in at most n steps:
  $\vdash$ `ReachBy` $0\ \mathcal{R}\ \mathcal{B}\ s\ =\ \mathcal{B}\ s$
  $\vdash$ `ReachBy` $(n{+}1)\ \mathcal{R}\ \mathcal{B}\ s\ =$
  `ReachBy` $n\ \mathcal{R}\ \mathcal{B}\ s$
  $\vee$
  $\exists u.$ `ReachBy` $n\ \mathcal{R}\ \mathcal{B}\ u \wedge \mathcal{R}(u,s)$
- Reachable states are states reachable in a finite number of steps:
  $\vdash$ `Reach` $\mathcal{R}\ \mathcal{B}\ s\ =\ \exists n.$ `ReachBy` $n\ \mathcal{R}\ \mathcal{B}\ s$
- Key property (equality between predicates represents set equality):
  $\vdash$ (`ReachBy` $n\ \mathcal{R}\ \mathcal{B}$ = `ReachBy` $(n{+}1)\ \mathcal{R}\ \mathcal{B}$)
  $\Rightarrow$
  (`Reach` $\mathcal{R}\ \mathcal{B}$ = `ReachBy` $n\ \mathcal{R}\ \mathcal{B}$)

Represent formulas as Binary Decision Diagrams

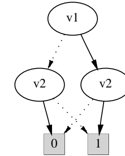
- Reduced Ordered Binary Decision Diagrams (ROBDDs or BDDs for short) are a data-structure for representing Boolean formulas
- Key features:
  - canonical (given a variable ordering)
  - efficient to manipulate
- Variables: `v = if v then 1 else 0` and `¬v = if v then 0 else 1`
- Example: BDDs of variable v and ¬v

- Example: BDDs of v1 ∧ v2 and v1 ∨ v2

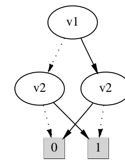


More BDD examples

- BDD of v1 = v2



- BDD of v1 ≠ v2

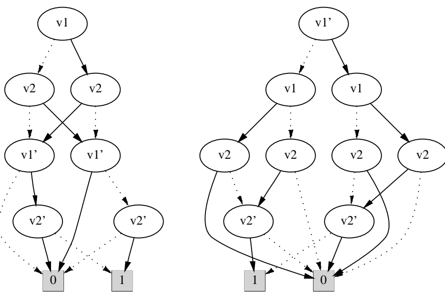


BDD of a transition relation

- BDDs of

  $(\mathtt{v1}' = (\mathtt{v1} = \mathtt{v2})) \wedge (\mathtt{v2}' = (\mathtt{v1} \oplus \mathtt{v2}))$

  with two different variable orderings



- Exercise: draw BDD of $\mathcal{R}_{\mathtt{RECEIVER}}$

Standard BDD operations

- If formulas f_1, f_2 represents sets s_1, s_2, respectively then $f_1 \wedge f_2$, $f_1 \vee f_2$ represent $s_1 \cap s_2$, $s_1 \cup s_2$ respectively
- Standard algorithms can compute boolean operation on BDDs.
- If $f(x)$ represents $\{x \mid \mathcal{B}(x)\}$ and $g(s, s')$ represents $\{(s, s') \mid \mathcal{R}(s, s')\}$ then $\exists u.\ f(u) \wedge g(u, s)$ represents $\{s \mid \exists u.\ \mathcal{R}(u, s)\}$
- 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 $h(\mathtt{T}, v) \vee h(\mathtt{F}, v)$
- Given a BDD representing formula f with free variables v_1, …, v_n there exists an algorithm to find truth-values b_1, …, b_n such that if $v_1 = b_1$, …, $v_n = b_n$ then f evaluates to T
  - b_1, …, 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 $\mathcal{R}(s,s')$ and $\mathcal{B}\ s$ as BDDs
- Iteratively compute BDDs of $\mathcal{S}_0\ s$, $\mathcal{S}_1\ s$, $\mathcal{S}_2\ s$ etc:

$$\begin{array}{lcl} \mathcal{S}_0\ s & = & \mathcal{B}\ s \\ \mathcal{S}_1\ s & = & \mathcal{S}_0\ s \vee \exists u.\ \mathcal{S}_0\ u \wedge \mathcal{R}(u,s) \\ \mathcal{S}_2\ s & = & \mathcal{S}_1\ s \vee \exists u.\ \mathcal{S}_1\ u \wedge \mathcal{R}(u,s) \\ \vdots & & \\ \mathcal{S}_{n+1}\ s & = & \mathcal{S}_n\ s \vee \exists u.\ \mathcal{S}_n\ u \wedge \mathcal{R}(u,s) \end{array}$$

- BDD of $\exists u.\ \mathcal{S}_i\ u \wedge \mathcal{R}(u,s)$ computed by:

  $\exists u.\ (\mathcal{S}_i\ s)[u/s] \wedge \mathcal{R}(s,s')[(u,s)/(s,s')]$

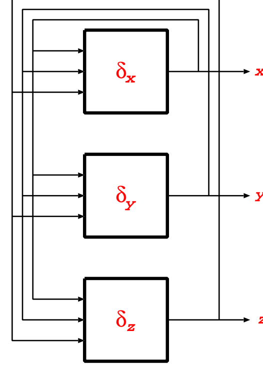
  efficient using standard BDD algorithms
  (renaming, then conjuction, then existential quantification)

- At each iteration check $\mathcal{S}_{n+1}\ s = \mathcal{S}_n\ s$ efficient using BDDs,
  when $\mathcal{S}_{n+1}\ s = \mathcal{S}_n\ s$ can conclude

  $\texttt{Reach}\ \mathcal{R}\ \mathcal{B}\ \ s\ =\ \mathcal{S}_n\ s$

  hence have computed BDD of $\texttt{Reach}\ \mathcal{R}\ \mathcal{B}\ \ s$

Example BDD optimisation: disjunctive partitioning



- Transition relation (asynchronous interleaving semantics):

$$\begin{array}{l} \mathcal{R}((x,y,z),(x',y',z')) = \\ \quad (x'=\delta_x(x,y,z)\ \wedge\ y'=y\ \wedge\ z'=z) \vee \\ \quad (x'=x\ \wedge\ y'=\delta_y(x,y,z)\ \wedge\ z'=z) \vee \\ \quad (x'=x\ \wedge\ y'=y\ \wedge\ z'=\delta_z(x,y,z)) \end{array}$$

Avoiding building big BDDs

- Transition relation for three machines in parallel

$$\begin{array}{l} \mathcal{R}((x,y,z),(x',y',z')) = \\ \quad (x'=\delta_x(x,y,z)\ \wedge\ y'=y\ \wedge\ z'=z) \vee \\ \quad (x'=x\ \wedge\ y'=\delta_y(x,y,z)\ \wedge\ z'=z) \vee \\ \quad (x'=x\ \wedge\ y'=y\ \wedge\ z'=\delta_z(x,y,z)) \end{array}$$

- Recall:

$$\begin{array}{l} \texttt{ReachBy}\ (n{+}1)\ \mathcal{R}\ \mathcal{B}\ s \\ \quad = \texttt{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s\ \vee \\ \qquad \exists u.\ \texttt{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ u\ \wedge\ \mathcal{R}(u,s) \end{array}$$

- With $s=(x,y,z)$ it can be shown (see next slide):

$$\begin{array}{l} \texttt{ReachBy}\ (n{+}1)\ \mathcal{R}\ \mathcal{B}\ (x,y,z) \\ \quad = \texttt{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ (x,y,z)\ \vee \\ \qquad (\exists\overline{x}.\ \texttt{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ (\overline{x},y,z)\ \wedge\ x=\delta_x(\overline{x},y,z)) \vee \\ \qquad (\exists\overline{y}.\ \texttt{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ (x,\overline{y},z)\ \wedge\ y=\delta_y(x,\overline{y},z)) \vee \\ \qquad (\exists\overline{z}.\ \texttt{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ (x,y,\overline{z})\ \wedge\ z=\delta_z(x,y,\overline{z})) \end{array}$$

- $\mathcal{R}(u,s)$ not a subterm: 'early quantification', 'disjunctive partitioning'

More Details (Exercise: check the logic below)

Let $\texttt{Ry}(\overline{x},\overline{y},\overline{z})$ abbreviate $\texttt{ReachBy}\,n\,\mathcal{R}\,\mathcal{B}(\overline{x},\overline{y},\overline{z})$ then:

$$\begin{array}{l} \exists\overline{x}\,\overline{y}\,\overline{z}.\,\texttt{ReachBy}\,n\,\mathcal{R}\,\mathcal{B}(\overline{x},\overline{y},\overline{z}) \wedge \mathcal{R}((\overline{x},\overline{y},\overline{z}),(x,y,z)) \\ = \exists\overline{x}\,\overline{y}\,\overline{z}.\,\texttt{Ry}(\overline{x},\overline{y},\overline{z}) \wedge \mathcal{R}((\overline{x},\overline{y},\overline{z}),(x,y,z)) \\ = \exists\overline{x}\,\overline{y}\,\overline{z}.\,\texttt{Ry}(\overline{x},\overline{y},\overline{z}) \wedge ((x=\delta_x(\overline{x},\overline{y},\overline{z}) \wedge y=\overline{y} \wedge z=\overline{z}) \vee \\ \qquad\qquad (x=\overline{x} \wedge y=\delta_y(\overline{x},\overline{y},\overline{z}) \wedge z=\overline{z}) \vee \\ \qquad\qquad (x=\overline{x} \wedge y=\overline{y} \wedge z=\delta_z(\overline{x},\overline{y},\overline{z}))) \\ = (\exists\overline{x}\,\overline{y}\,\overline{z}.\,\texttt{Ry}(\overline{x},\overline{y},\overline{z}) \wedge x=\delta_x(\overline{x},\overline{y},\overline{z}) \wedge y=\overline{y} \wedge z=\overline{z}) \vee \\ \quad (\exists\overline{x}\,\overline{y}\,\overline{z}.\,\texttt{Ry}(\overline{x},\overline{y},\overline{z}) \wedge x=\overline{x} \wedge y=\delta_y(\overline{x},\overline{y},\overline{z}) \wedge z=\overline{z}) \vee \\ \quad (\exists\overline{x}\,\overline{y}\,\overline{z}.\,\texttt{Ry}(\overline{x},\overline{y},\overline{z}) \wedge x=\overline{x} \wedge y=\overline{y} \wedge z=\delta_z(\overline{x},\overline{y},\overline{z})) \\ = (\exists\overline{x}\,\overline{y}\,\overline{z}.\,\texttt{Ry}(\overline{x},y,z) \wedge x=\delta_x(\overline{x},y,z) \wedge y=\overline{y} \wedge z=\overline{z}) \vee \\ \quad (\exists\overline{x}\,\overline{y}\,\overline{z}.\,\texttt{Ry}(x,\overline{y},z) \wedge x=\overline{x} \wedge y=\delta_y(x,\overline{y},z) \wedge z=\overline{z}) \vee \\ \quad (\exists\overline{x}\,\overline{y}\,\overline{z}.\,\texttt{Ry}(x,y,\overline{z}) \wedge x=\overline{x} \wedge y=\overline{y} \wedge z=\delta_z(x,y,\overline{z})) \\ = ((\exists\overline{x}.\,\texttt{Ry}(\overline{x},y,z) \wedge x{=}\delta_x(\overline{x},y,z)) \wedge (\exists\overline{y}.\,y{=}\overline{y}) \wedge (\exists\overline{z}.\,z{=}\overline{z})) \vee \\ \quad ((\exists\overline{x}.\,x{=}\overline{x}) \wedge (\exists\overline{y}.\,\texttt{Ry}(x,\overline{y},z) \wedge y{=}\delta_y(x,\overline{y},z)) \wedge (\exists\overline{z}.\,z{=}\overline{z})) \vee \\ \quad ((\exists\overline{x}.\,x{=}\overline{x}) \wedge (\exists\overline{y}.\,y{=}\overline{y}) \wedge (\exists\overline{z}.\,\texttt{Ry}(x,y,\overline{z}) \wedge z{=}\delta_z(x,y,\overline{z}))) \\ = (\exists\overline{x}.\,\texttt{Ry}(\overline{x},y,z) \wedge x=\delta_x(\overline{x},y,z)) \vee \\ \quad (\exists\overline{y}.\,\texttt{Ry}(x,\overline{y},z) \wedge y=\delta_y(x,\overline{y},z)) \vee \\ \quad (\exists\overline{z}.\,\texttt{Ry}(x,y,\overline{z}) \wedge z=\delta_z(x,y,\overline{z})) \end{array}$$

Verification and Counterexamples

- Typical safety question:
  - is $\mathcal{Q}$ true in all reachable states?
  - i.e. is `Reach` $\mathcal{R}\ \mathcal{B}\ s \ \Rightarrow\ \mathcal{Q}\ s$ true?
- Compute BDD of `Reach` $\mathcal{R}\ \mathcal{B}\ s \ \Rightarrow\ \mathcal{Q}\ s$
- Formula is true if BDD is the single node $\boxed{1}$
  - because `T` represented by a unique BDD (canonical property)
- If BDD is not $\boxed{1}$ can get counterexample

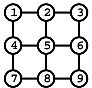
Generating Counterexample Traces

BDD algorithms can find satisfying assignments (SAT)

- Suppose `Reach` $\mathcal{R}\ \mathcal{B}\ s \ \Rightarrow\ \mathcal{Q}\ s$ is not true
- Must exist s satisfying `Reach` $\mathcal{R}\ \mathcal{B}\ s\ \wedge\ \neg\mathcal{Q}\ s$
- Find counterexample algorithm:
  - iteratively generate BDDs of `ReachBy` $i\ \mathcal{R}\ \mathcal{B}\ s$ ($i = 0, 1, \ldots$)
  - at each stage check if `ReachBy` $i\ \mathcal{R}\ \mathcal{B}$ s $\wedge\ \neg(\mathcal{Q}$ s$)$ satisfiable
  - hence find first n and, using SAT, a state $\mathtt{s}_n$ such that
    (`ReachBy` n $\mathcal{R}\ \mathcal{B}\ s\ \wedge\ \neg(\mathcal{Q}\ s))\ [\mathtt{s}_n/s]$
    i.e.
    `ReachBy` n $\mathcal{R}\ \mathcal{B}\ \mathtt{s}_n\ \wedge\ \neg(\mathcal{Q}\ \mathtt{s}_n)$
- Then use BDD SAT to get $\mathtt{s}_{n-1}$ where
  (`ReachBy` $(\mathtt{n}-1)\,\mathcal{R}\,\mathcal{B}\,s \wedge \mathcal{R}(s, \mathtt{s}_n))\ [\mathtt{s}_{n-1}/s]$
  i.e.
  `ReachBy` $(n-1)\,\mathcal{R}\,\mathcal{B}\,\mathtt{s}_{n-1} \wedge \mathcal{R}(\mathtt{s}_{n-1}, \mathtt{s}_n)$
- Iteratively trace backwards to get $\mathtt{s}_n, \ldots, \mathtt{s}_0$ where for $0 < i \leq \mathtt{n}$:
  `ReachBy` $(i-1)\,\mathcal{R}\,\mathcal{B}\,\mathtt{s}_{i-1} \wedge \mathcal{R}(\mathtt{s}_{i-1}, \mathtt{s}_i)$
- Can sometimes apply partitioning, so BDD of $\mathcal{R}$ not needed

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.

Solution

The state space can consist of the set of vectors
`(v0,v1,v2,v3,v4,v5,v6,v7,v8)`
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 `Trans` is then defined by:

```
Trans((v0,v1,v2,v3,v4,v5,v6,v7,v8),(v0',v1',v2',v3',v4',v5',v6',v7',v8'))
 = ((v0'=¬v0)∧(v1'=¬v1)∧(v2'=v2)∧(v3'=¬v3)∧(v4'=v4)∧
     (v5'=v5)∧(v6'=v6)∧(v7'=v7)∧(v8'=v8))          (toggle switch 1)
 ∨ ((v0'=¬v0)∧(v1'=¬v1)∧(v2'=¬v2)∧(v3'=v3)∧(v4'=¬v4)∧
     (v5'=v5)∧(v6'=v6)∧(v7'=v7)∧(v8'=v8))          (toggle switch 2)
 ∨ ((v0'=v0)∧(v1'=¬v1)∧(v2'=¬v2)∧(v3'=v3)∧(v4'=v4)∧
     (v5'=¬v5)∧(v6'=v6)∧(v7'=v7)∧(v8'=v8))         (toggle switch 3)
 ∨ ((v0'=¬v0)∧(v1'=v1)∧(v2'=v2)∧(v3'=¬v3)∧(v4'=¬v4)∧
     (v5'=v5)∧(v6'=¬v6)∧(v7'=v7)∧(v8'=v8))         (toggle switch 4)
 ∨ ((v0'=v0)∧(v1'=¬v1)∧(v2'=v2)∧(v3'=¬v3)∧(v4'=¬v4)∧
     (v5'=¬v5)∧(v6'=v6)∧(v7'=¬v7)∧(v8'=v8))        (toggle switch 5)
 ∨ ((v0'=v0)∧(v1'=v1)∧(v2'=¬v2)∧(v3'=v3)∧(v4'=¬v4)∧
     (v5'=¬v5)∧(v6'=v6)∧(v7'=v7)∧(v8'=¬v8))        (toggle switch 6)
 ∨ ((v0'=v0)∧(v1'=v1)∧(v2'=v2)∧(v3'=¬v3)∧(v4'=v4)∧
     (v5'=v5)∧(v6'=¬v6)∧(v7'=¬v7)∧(v8'=v8))        (toggle switch 7)
 ∨ ((v0'=v0)∧(v1'=v1)∧(v2'=v2)∧(v3'=v3)∧(v4'=¬v4)∧
     (v5'=v5)∧(v6'=¬v6)∧(v7'=¬v7)∧(v8'=¬v8))       (toggle switch 8)
 ∨ ((v0'=v0)∧(v1'=v1)∧(v2'=v2)∧(v3'=v3)∧(v4'=v4)∧
     (v5'=¬v5)∧(v6'=v6)∧(v7'=¬v7)∧(v8'=¬v8))       (toggle switch 9)
```

Predicates `Init`, `Final` characterising the initial and final states, respectively, are defined by:

`Init(v0,v1,v2,v3,v4,v5,v6,v7,v8) =`
 `¬v0 ∧ v1 ∧ ¬v2 ∧ v3 ∧ ¬v4 ∧ v5 ∧ ¬v6 ∧ v7 ∧ ¬v8`

`Final(v0,v1,v2,v3,v4,v5,v6,v7,v8) =`
 `¬v0 ∧ ¬v1 ∧ ¬v2 ∧ ¬v3 ∧ ¬v4 ∧ ¬v5 ∧ ¬v6 ∧ ¬v7 ∧ ¬v8`

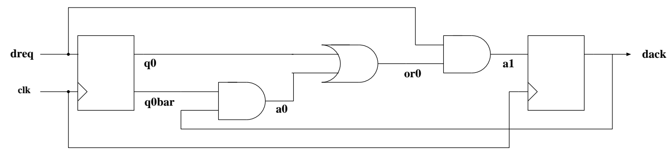
Model checkers can find counter-examples to properties, and sequences of transitions from an initial state to a counter-example state. Thus we could use a model checker to find a trace to a counter-example to the property that `¬Final(v0,v1,v2,v3,v4,v5,v6,v7,v8)`.

Properties

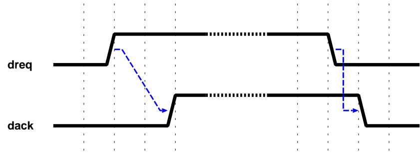
- `Reach` $\mathcal{R}\ \mathcal{B}\ s \Rightarrow \mathcal{Q}\ s$ means $\mathcal{Q}$ true in all reachable states
- Might want to verify other properties, e.g:
  1. *DeviceEnabled* is always true somewhere along every path starting anywhere (i.e. it holds infinitely often along every path)
  2. From any state it is possible to get to a state for which *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:



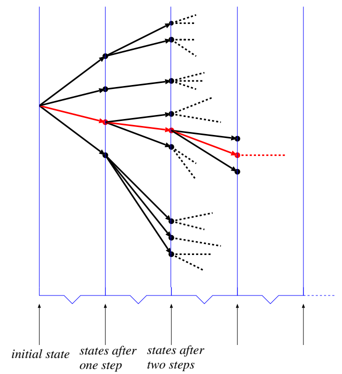
- Input: `dreq`, registers: `q0`, `dack`
- Timing Diagram:



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 $\alpha \times \alpha \to bool$**
  - $\mathcal{R}$ is a transition relation
  - α ranges (intuitively) over states
- **An $\mathcal{R}$-path is a function $\sigma : num \to \alpha$ such that: $\forall t.\ \mathcal{R}(\sigma(t), \sigma(t{+}1))$**
- **$\mathtt{Path}(\mathcal{R}, s)\sigma$ means σ is an $\mathcal{R}$-path from s**

$$\mathtt{Path}(\mathcal{R}, s)\sigma \ = \ (\sigma(0){=}s) \ \wedge \ \forall t.\ \mathcal{R}(\sigma(t), \sigma(t{+}1))$$

. . . .

- **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

Computation Tree Logic (CTL)

- **Syntax of CTL well-formed formulas:**

wff	$::= \mathtt{Atom}(p)$	(Atomic formula)
	$\mid \neg wff$	(Negation)
	$\mid wff_1 \wedge wff_2$	(Conjunction)
	$\mid wff_1 \vee wff_2$	(Disjunction)
	$\mid wff_1 \Rightarrow wff_2$	(Implication)
	$\mid \mathbf{AX}wff$	(All successors)
	$\mid \mathbf{EX}wff$	(Some successors)
	$\mid \mathbf{A}[wff_1\ \mathbf{U}\ wff_2]$	(Until – along all paths)
	$\mid \mathbf{E}[wff_1\ \mathbf{U}\ wff_2]$	(Until – along some path)

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

Semantics of CTL (shallow embedding)

- **A model is a pair $(\mathcal{R}, s)$ — a transition relation and an initial state**
- **Define:**

$$
\begin{array}{ll}
\mathtt{Atom}(p) & = \lambda(\mathcal{R}, s).\ p(s) \\
\neg P & = \lambda(\mathcal{R}, s).\ \neg(P(\mathcal{R}, s)) \\
P \wedge Q & = \lambda(\mathcal{R}, s).\ P(\mathcal{R}, s) \ \wedge \ Q(\mathcal{R}, s) \\
P \vee Q & = \lambda(\mathcal{R}, s).\ P(\mathcal{R}, s) \ \vee \ Q(\mathcal{R}, s) \\
P \Rightarrow Q & = \lambda(\mathcal{R}, s).\ P(\mathcal{R}, s) \ \Rightarrow \ Q(\mathcal{R}, s) \\
\mathbf{AX}P & = \lambda(\mathcal{R}, s).\ \forall s'.\ \mathcal{R}(s, s') \ \Rightarrow \ P(\mathcal{R}, s') \\
\mathbf{EX}P & = \lambda(\mathcal{R}, s).\ \exists s'.\ \mathcal{R}(s, s') \ \wedge \ P(\mathcal{R}, s') \\
\mathbf{A}[P\ \mathbf{U}\ Q] & = \lambda(\mathcal{R}, s).\ \forall \sigma.\ \mathtt{Path}(\mathcal{R}, s)\sigma \Rightarrow \exists i.\ Q(\mathcal{R}, \sigma(i)) \wedge \forall j.\ j < i \ \Rightarrow \ P(\mathcal{R}, \sigma(j)) \\
\mathbf{E}[P\ \mathbf{U}\ Q] & = \lambda(\mathcal{R}, s).\ \exists \sigma.\ \mathtt{Path}(\mathcal{R}, s)\sigma \wedge \exists i.\ Q(\mathcal{R}, \sigma(i)) \wedge \forall j.\ j < i \ \Rightarrow \ P(\mathcal{R}, \sigma(j))
\end{array}
$$

The defined operator AF

- **Define $\mathbf{AF}P = \mathbf{A}[\mathtt{T}\ \mathbf{U}\ P]$**
- **$\mathbf{AF}P$ is true if P holds somewhere along every $\mathcal{R}$-path – P is *inevitable***

$$
\begin{array}{l}
\mathbf{AF}P \\
\quad = \mathbf{A}[\mathtt{T}\ \mathbf{U}\ P] \\
\quad = \lambda(\mathcal{R}, s).\ \forall \sigma.\ \mathtt{Path}(\mathcal{R}, s)\sigma \Rightarrow \exists i.\ P(\mathcal{R}, \sigma(i)) \ \wedge \ \forall j.\ j < i \ \Rightarrow \ \mathtt{T}(\mathcal{R}, \sigma(j)) \\
\quad = \lambda(\mathcal{R}, s).\ \forall \sigma.\ \mathtt{Path}(\mathcal{R}, s)\sigma \Rightarrow \exists i.\ P(\mathcal{R}, \sigma(i))
\end{array}
$$

The defined operator EF

- Define $\mathbf{EF}P = \mathbf{E}[\mathtt{T}\ \mathbf{U}\ P]$
- $\mathbf{EF}P$ is true if P holds somewhere along some $\mathcal{R}$-path – i.e. P *potentially* holds

$$
\begin{aligned}
&\mathbf{EF}P\\
&\quad = \mathbf{E}[\mathtt{T}\ \mathbf{U}\ P]\\
&\quad = \lambda(\mathcal{R},s).\ \exists\sigma.\ \mathtt{Path}(\mathcal{R},s)\sigma \wedge \exists i.\ P(\mathcal{R},\sigma(i)) \wedge \forall j.\ j<i \Rightarrow \mathtt{T}(\mathcal{R},\sigma(j))\\
&\quad = \lambda(\mathcal{R},s).\ \exists\sigma.\ \mathtt{Path}(\mathcal{R},s)\sigma \wedge \exists i.\ P(\mathcal{R},\sigma(i))
\end{aligned}
$$

The defined operator AG

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

$$
\begin{aligned}
\mathbf{AG}P &= \neg\mathbf{EF}(\neg P)\\
&= \lambda(\mathcal{R},s).\ (\neg\mathbf{EF}(\neg P))(\mathcal{R},s)\\
&= \lambda(\mathcal{R},s).\ \neg(\exists\sigma.\ \mathtt{Path}(\mathcal{R},s)\sigma \wedge \exists i.\ (\neg P)(\mathcal{R},\sigma(i)))\\
&= \lambda(\mathcal{R},s).\ \neg(\exists\sigma.\ \mathtt{Path}(\mathcal{R},s)\sigma \wedge \exists i.\ \neg P(\mathcal{R},\sigma(i)))\\
&= \lambda(\mathcal{R},s).\ \forall\sigma.\ \neg(\mathtt{Path}(\mathcal{R},s)\sigma \wedge \exists i.\ \neg P(\mathcal{R},\sigma(i)))\\
&= \lambda(\mathcal{R},s).\ \forall\sigma.\ \neg\mathtt{Path}(\mathcal{R},s)\sigma \vee \neg(\exists i.\ \neg P(\mathcal{R},\sigma(i)))\\
&= \lambda(\mathcal{R},s).\ \forall\sigma.\ \neg\mathtt{Path}(\mathcal{R},s)\sigma \vee \forall i.\ \neg\neg P(\mathcal{R},\sigma(i))\\
&= \lambda(\mathcal{R},s).\ \forall\sigma.\ \neg\mathtt{Path}(\mathcal{R},s)\sigma \vee \forall i.\ P(\mathcal{R},\sigma(i))\\
&= \lambda(\mathcal{R},s).\ \forall\sigma.\ \mathtt{Path}(\mathcal{R},s)\sigma \Rightarrow \forall i.\ P(\mathcal{R},\sigma(i))
\end{aligned}
$$

- $\mathbf{AG}P$ means P true at all reachable states
- $\mathbf{AG}(\mathtt{Atom}\ p)(\mathcal{R},s) \equiv \forall s'.\ \mathtt{Reach}\ \mathcal{R}\ (\lambda x.\ x{=}s)\ s' \Rightarrow p(s')$

The defined operator EG

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

$$
\begin{aligned}
\mathbf{EG}P &= \neg\mathbf{AF}(\neg P)\\
&= \lambda(\mathcal{R},s).\ (\neg\mathbf{AF}(\neg P))(\mathcal{R},s)\\
&= \lambda(\mathcal{R},s).\ \neg(\forall\sigma.\ \mathtt{Path}(\mathcal{R},s)\sigma \Rightarrow \exists i.\ (\neg P)(\mathcal{R},\sigma(i)))\\
&= \lambda(\mathcal{R},s).\ \neg(\forall\sigma.\ \mathtt{Path}(\mathcal{R},s)\sigma \Rightarrow \exists i.\ \neg P(\mathcal{R},\sigma(i)))\\
&= \lambda(\mathcal{R},s).\ \exists\sigma.\ \neg(\mathtt{Path}(\mathcal{R},s)\sigma \Rightarrow \exists i.\ \neg P(\mathcal{R},\sigma(i)))\\
&= \lambda(\mathcal{R},s).\ \exists\sigma.\ \mathtt{Path}(\mathcal{R},s)\sigma \wedge \neg(\exists i.\ \neg P(\mathcal{R},\sigma(i)))\\
&= \lambda(\mathcal{R},s).\ \exists\sigma.\ \mathtt{Path}(\mathcal{R},s)\sigma \wedge \forall i.\ \neg\neg P(\mathcal{R},\sigma(i))\\
&= \lambda(\mathcal{R},s).\ \exists\sigma.\ \mathtt{Path}(\mathcal{R},s)\sigma \wedge \forall i.\ P(\mathcal{R},\sigma(i))
\end{aligned}
$$

The defined operator $\mathbf{A}[P\mathbf{W}Q]$

- $\mathbf{A}[P\mathbf{W}Q]$ is a 'partial correctness' version of $\mathbf{A}[P\mathbf{U}Q]$
- 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

$$
\begin{aligned}
&\mathbf{A}[P\mathbf{W}Q]\\
&\quad = \neg\mathbf{E}[(P\wedge\neg Q)\mathbf{U}(\neg P\wedge\neg Q)]\\
&\quad = \lambda(\mathcal{R},s).\ (\neg\mathbf{E}[(P\wedge\neg Q)\mathbf{U}(\neg P\wedge\neg Q)])(\mathcal{R},s)\\
&\quad = \lambda(\mathcal{R},s).\ \neg(\mathbf{E}[(P\wedge\neg Q)\mathbf{U}(\neg P\wedge\neg Q)])(\mathcal{R},s)\\
&\quad = \lambda(\mathcal{R},s).\ \neg(\exists\sigma.\ \mathtt{Path}(\mathcal{R},s)\sigma \wedge \exists i.\ (\neg P\wedge\neg Q)(\mathcal{R},\sigma(i)) \wedge \forall j.\ j<i \Rightarrow (P\wedge\neg Q)(\mathcal{R},\sigma(j)))
\end{aligned}
$$

- Exercise: understand the next three slides

$\mathbf{A}[P\mathbf{W}Q]$ continued (1)

- **Continuing:**

$$
\begin{array}{l}
\lambda(\mathcal{R},s).\\
\quad\neg(\exists\sigma.\ \texttt{Path}(\mathcal{R},s)\sigma\\
\qquad\qquad\wedge\\
\qquad\quad\exists i.\ (\neg P{\wedge}\neg Q)(\mathcal{R},\sigma(i))\ \wedge\ \forall j.\ j<i\ \Rightarrow\ (P{\wedge}\neg Q)(\mathcal{R},\sigma(j)))\\
=\lambda(\mathcal{R},s).\\
\quad\forall\sigma.\ \neg(\texttt{Path}(\mathcal{R},s)\sigma\\
\qquad\qquad\wedge\\
\qquad\quad\exists i.\ (\neg P{\wedge}\neg Q)(\mathcal{R},\sigma(i))\ \wedge\ \forall j.\ j<i\ \Rightarrow\ (P{\wedge}\neg Q)(\mathcal{R},\sigma(j)))\\
=\lambda(\mathcal{R},s).\\
\quad\forall\sigma.\ \texttt{Path}(\mathcal{R},s)\sigma\\
\qquad\quad\Rightarrow\\
\qquad\neg(\exists i.\ (\neg P{\wedge}\neg Q)(\mathcal{R},\sigma(i))\ \wedge\ \forall j.\ j<i\ \Rightarrow\ (P{\wedge}\neg Q)(\mathcal{R},\sigma(j)))\\
=\lambda(\mathcal{R},s).\\
\quad\forall\sigma.\ \texttt{Path}(\mathcal{R},s)\sigma\\
\qquad\quad\Rightarrow\\
\qquad\forall i.\ \neg(\neg P{\wedge}\neg Q)(\mathcal{R},\sigma(i))\ \vee\ \neg(\forall j.\ j<i\ \Rightarrow\ (P{\wedge}\neg Q)(\mathcal{R},\sigma(j)))
\end{array}
$$

$\mathbf{A}[P\mathbf{W}Q]$ continued (2)

- **Continuing:**

$$
\begin{array}{l}
\lambda(\mathcal{R},s).\\
\quad\forall\sigma.\ \texttt{Path}(\mathcal{R},s)\sigma\\
\qquad\quad\Rightarrow\\
\qquad\forall i.\ \neg(\neg P{\wedge}\neg Q)(\mathcal{R},\sigma(i))\ \vee\ \neg(\forall j.\ j<i\ \Rightarrow\ (P{\wedge}\neg Q)(\mathcal{R},\sigma(j)))\\
=\lambda(\mathcal{R},s).\\
\quad\forall\sigma.\ \texttt{Path}(\mathcal{R},s)\sigma\\
\qquad\quad\Rightarrow\\
\qquad\forall i.\ \neg(\forall j.\ j<i\ \Rightarrow\ (P{\wedge}\neg Q)(\mathcal{R},\sigma(j)))\\
\qquad\qquad\vee\\
\qquad\qquad\neg(\neg P{\wedge}\neg Q)(\mathcal{R},\sigma(i))\\
=\lambda(\mathcal{R},s).\\
\quad\forall\sigma.\ \texttt{Path}(\mathcal{R},s)\sigma\\
\qquad\quad\Rightarrow\\
\qquad\forall i.\ (\forall j.\ j<i\ \Rightarrow\ P(\mathcal{R},\sigma(j))\ \wedge\ \neg Q(\mathcal{R},\sigma(j)))\\
\qquad\qquad\Rightarrow\\
\qquad\qquad P(\mathcal{R},\sigma(i))\vee Q(\mathcal{R},\sigma(i))
\end{array}
$$

- **Exercise: does this correspond to earlier description of $\mathbf{A}[P\mathbf{W}Q]$?**
  - **this exercise illustrates the subtlety of writing CTL!**

$\mathbf{A}[P\mathbf{W}\texttt{F}]\ =\ \mathbf{AG}\ P$

- **From last slide:**

$$
\begin{array}{l}
\mathbf{A}[P\mathbf{W}Q]\\
=\lambda(\mathcal{R},s).\\
\quad\forall\sigma.\ \texttt{Path}(\mathcal{R},s)\sigma\\
\qquad\quad\Rightarrow\\
\qquad\forall i.\ (\forall j.\ j<i\ \Rightarrow\ P(\mathcal{R},\sigma(j))\ \wedge\ \neg Q(\mathcal{R},\sigma(j)))\\
\qquad\qquad\Rightarrow\\
\qquad\qquad P(\mathcal{R},\sigma(i))\vee Q(\mathcal{R},\sigma(i))
\end{array}
$$

- **Set Q to be $\texttt{F}$:**

$$
\begin{array}{l}
\mathbf{A}[P\mathbf{W}\texttt{F}]\\
=\lambda(\mathcal{R},s).\\
\quad\forall\sigma.\ \texttt{Path}(\mathcal{R},s)\sigma\\
\qquad\quad\Rightarrow\\
\qquad\forall i.\ (\forall j.\ j<i\ \Rightarrow\ P(\mathcal{R},\sigma(j))\ \wedge\ \neg\texttt{F}(\mathcal{R},\sigma(j)))\\
\qquad\qquad\Rightarrow\\
\qquad\qquad P(\mathcal{R},\sigma(i))\vee \texttt{F}(\mathcal{R},\sigma(i))
\end{array}
$$

- **Simplify:**

$$
\begin{array}{l}
\mathbf{A}[P\mathbf{W}\texttt{F}]\\
=\lambda(\mathcal{R},s).\ \forall\sigma.\ \texttt{Path}(\mathcal{R},s)\sigma\ \Rightarrow\ \forall i.\ (\forall j.\ j<i\ \Rightarrow\ P(\mathcal{R},\sigma(j)))\ \Rightarrow\ P(\mathcal{R},\sigma(i))
\end{array}
$$

- **By induction on i:**

$$\mathbf{A}[P\mathbf{W}\texttt{F}]\ =\ \lambda(\mathcal{R},s).\ \forall\sigma.\ \texttt{Path}(\mathcal{R},s)\sigma\ \Rightarrow\ \forall i.\ P(\mathcal{R},\sigma(i))$$

- **Exercise: describe the property specified by $\mathbf{A}[\texttt{T}\mathbf{W}Q]$**

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
```


Summary of CTL operators (primitive + defined)

- **CTL formulas:**

$\texttt{Atom}(p)$	(Atomic formula - $p : \mathit{states}\rightarrow\mathit{bool}$)
$\neg P$	(Negation)
$P \wedge Q$	(Conjunction)
$P \vee Q$	(Disjunction)
$P \Rightarrow Q$	(Implication)
$\mathbf{AX}P$	(All successors)
$\mathbf{EX}P$	(Some successors)
$\mathbf{AF}P$	(Somewhere – along all paths)
$\mathbf{EF}P$	(Somewhere – along some path)
$\mathbf{AG}P$	(Everywhere – along all paths)
$\mathbf{EG}P$	(Everywhere – along some path)
$\mathbf{A}[P\ \mathbf{U}\ Q]$	(Until – along all paths)
$\mathbf{E}[P\ \mathbf{U}\ Q]$	(Until – along some path)
$\mathbf{A}[P\ \mathbf{W}\ Q]$	(Unless – along all paths)
$\mathbf{E}[P\ \mathbf{W}\ Q]$	(Unless – along some path)

- **Say 'P holds' if $P(\mathcal{R}, s)$ for all initial states s**

Example CTL formulas

- $\mathbf{EF}(\mathit{Started} \wedge \neg\mathit{Ready})$

  It is possible to get to a state where *Started* holds but *Ready* does not hold

- $\mathbf{AG}(\mathit{Req} \Rightarrow \mathbf{AF}\mathit{Ack})$

  If a request *Req* occurs, then it will eventually be acknowledged by *Ack*

- $\mathbf{AG}(\mathbf{AF}\mathit{DeviceEnabled})$

  ***DeviceEnabled* is always true somewhere along every path starting anywhere: i.e. *DeviceEnabled* holds infinitely often along every path**

- $\mathbf{AG}(\mathbf{EF}\mathit{Restart})$

  From any state it is possible to get to a state for which *Restart* holds

More CTL examples (1)

- $\mathbf{AG}(\mathit{Req} \Rightarrow \mathbf{A}[\mathit{Req}\ \mathbf{U}\ \mathit{Ack}])$

  If a request *Req* occurs, then it continues to hold, until it is eventually acknowledged

- $\mathbf{AG}(\mathit{Req} \Rightarrow \mathbf{AX}(\mathbf{A}[\neg\mathit{Req}\ \mathbf{U}\ \mathit{Ack}]))$

  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?

- $\mathbf{AG}(\mathit{Req} \Rightarrow (\neg\mathit{Ack} \Rightarrow \mathbf{AX}(\mathbf{A}[\mathit{Req}\ \mathbf{U}\ \mathit{Ack}])))$

  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)

- $\mathbf{AG}[\mathit{Enabled} \Rightarrow \mathbf{AG}[\mathit{Start} \Rightarrow \mathbf{A}[\neg \mathit{Waiting}\ \mathbf{U}\ \mathit{Ack}]]]$

  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

- $\mathbf{AG}[\neg \mathit{Req}_1 \wedge \neg \mathit{Req}_2$
  $\Rightarrow$
  $\mathbf{A}[\neg \mathit{Req}_1 \wedge \neg \mathit{Req}_2\ \mathbf{U}\ (\mathit{Start} \wedge \neg \mathit{Req}_2)]]$

  Whenever Req_1 and Req_2 are false, they remain false until *Start* becomes true with Req_2 still false

- $\mathbf{AG}[\mathit{Req} \Rightarrow \mathbf{AX}(\mathit{Ack} \Rightarrow \mathbf{AF}\ \neg \mathit{Req})]$

  If *Req* is true and *Ack* becomes true one cycle later, then eventually *Req* will become false

Some abbreviations

- $\mathbf{AX}_i\ P \ \equiv\ \underbrace{\mathbf{AX}(\mathbf{AX}(\cdots(\mathbf{AX}\ P)\cdots))}_{i \text{ instances of } \mathbf{AX}}$

  P is true on all paths i units of time later

- $\mathbf{ABF}_{i..j}\ P \ \equiv$
  $\mathbf{AX}_i\underbrace{(P \vee \mathbf{AX}(P \vee \cdots\ \mathbf{AX}(P \vee \mathbf{AX}\ P)\cdots))}_{j-i \text{ instances of } \mathbf{AX}}$

  P is true on all paths sometime between i units of time later and j units of time later

- $\mathbf{AG}[\mathit{Req} \Rightarrow \mathbf{AX}[\mathit{Ack}_1 \wedge \mathbf{ABF}_{1..6}(\mathit{Ack}_2 \wedge \mathbf{A}[\mathit{Wait}\ \mathbf{U}\ \mathit{Reply}])]]$

  One cycle after *Req*, Ack_1 should become true, and then Ack_2 becomes true 1 to 6 cycles later and then eventually *Reply* becomes true, but until it does *Wait* holds from the time of Ack_2

- More abbreviations in the 'Industry Standard' language PSL

CTL model checking algorithm

- A model is a relation $\mathcal{R}$
- A property is a CTL formula P
- Model checking: given CTL formula P compute $\{s \mid P(\mathcal{R}, s)\}$
- $P(\mathcal{R}, s_0)$ true if and only if $s_0 \in \{s \mid P(\mathcal{R}, s)\}$
- Assume set of states to be finite
  (infinite state model checking possible for some models)
- Already seen how to model check reachability
  $\mathbf{AG}(\mathtt{Atom}\ p)(\mathcal{R}, s)\ \equiv\ \forall s'.\ \mathtt{Reach}\ \mathcal{R}\ (\mathtt{Eq}\ s)\ s'\ \Rightarrow\ p(s')$
  so can model check AG of atomic properties – compute:
  $\{s' \mid\ \mathtt{Reach}\ \mathcal{R}\ (\mathtt{Eq}\ s)\ s'\ \Rightarrow\ p(s')\}$,
  e.g. via BDD of
  $\mathtt{Reach}\ \mathcal{R}\ (\mathtt{Eq}\ s)\ s'\ \Rightarrow\ p(s')$

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

$\mathbf{EF}(\mathtt{Atom}\ p)(\mathcal{R}, s)$ if p holds along some path starting at s

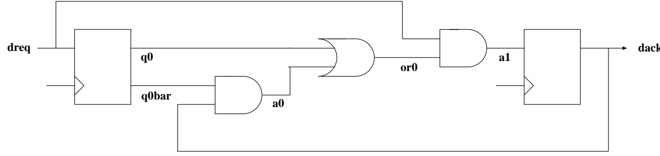
- Mark all the states satisfying p
- Repeatedly mark all the states which have at least one marked successor until no change
- $\{s \mid \mathbf{EF}(\mathtt{Atom}\ p)(\mathcal{R}, s)\}$ computed by generating:

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

  $\mathcal{S}_{i+1} = \mathcal{S}_i\ \cup\ \{s \mid \exists s'.\ \mathcal{R}(s, s')\ \wedge\ s' \in \mathcal{S}_i\}$
- $\mathbf{EF}(\mathtt{Atom}\ p)$ is true in marked states and false in unmarked states
- Algorithm similar for $\mathbf{AF}(\mathtt{Atom}\ p)$:
  repeatedly mark all the states which have all successors marked
- To check $\mathbf{AF}\ \mathbf{EF}\ (\mathtt{Atom}\ p)$:
  - apply EF algorithm
  - starting with resulting marking apply AF algorithm

Recall handshake example

- Part of a handshake circuit



- Transition relation:

```
(q0' = dreq) ∧ (dack' = dreq ∧ (q0 ∨ dack))
```

- Define $\mathcal{R}_{\texttt{RECEIVER}}$ by:

$\mathcal{R}_{\texttt{RECEIVER}}$`((dreq,q0,dack),(dreq',q0',dack')) =`
`(q0' ⇔ dreq) ∧ (dack' ⇔ dreq ∧ (q0 ∨ dack))`

- Primed variables (`dreq'`,`q0'`,`dack'`) represent 'next state'
- `dreq'` unconstrained, hence non-determinism

Model checking RECEIVER

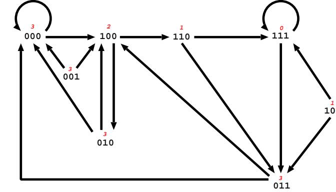
- Possible states for RECEIVER:

  $\{000, 001, 010, 011, 100, 101, 110, 111\}$

  where $b_2b_1b_0$ denotes state

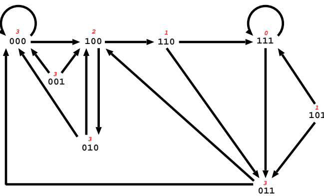
  $\texttt{dreq} = b_2 \;\wedge\; \texttt{q0} = b_1 \;\wedge\; \texttt{dack} = b_0$

- Graph of the transition relation:



- i above a state indicates membership of $\mathcal{S}_i$ (defined below)

Example: **EF**(dreq ∧ q0 ∧ dack)



- Define:

  $P \;=\; \texttt{Atom}(\lambda b_2b_1b_0.\; b_2 \;\wedge\; b_1 \;\wedge\; b_0)$

  $P(\mathcal{R}_{\texttt{RECEIVER}}, b_2b_1b_0) \;=\; b_2 \;\wedge\; b_1 \;\wedge\; b_0$

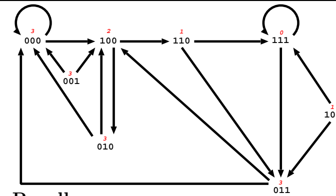
- Define:

  $\mathcal{S}_0 \;= \{b_2b_1b_0 \mid P(\mathcal{R}_{\texttt{RECEIVER}}, b_2b_1b_0)\}$

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

  $\quad = \mathcal{S}_i \;\cup\; \{b_2b_1b_0 \mid \exists b_2'b_1'b_0'.\; (b_1' = b_2) \;\wedge\; (b_0' = b_2 \wedge (b_1 \vee b_0)) \;\wedge\; b_2'b_1'b_0' \in \mathcal{S}_i\}$

Checking **EF**(dreq ∧ q0 ∧ dack)



- Recall:

  $\mathcal{S}_0 \;= \{b_2b_1b_0 \mid P(\mathcal{R}_{\texttt{RECEIVER}}, b_2b_1b_0)\}$

  $\mathcal{S}_{i+1} = \mathcal{S}_i \;\cup\; \{b_2b_1b_0 \mid \exists b_2'b_1'b_0'.\; (b_1' = b_2) \;\wedge\; (b_0' = b_2 \wedge (b_1 \vee b_0)) \;\wedge\; b_2'b_1'b_0' \in \mathcal{S}_i\}$

- Compute:

  $\mathcal{S}_0 = \{111\}$

  $\mathcal{S}_1 = \{111\} \;\cup\; \{101, 110\}$
  $\quad = \{111, 101, 110\}$

  $\mathcal{S}_2 = \{111, 101, 110\} \;\cup\; \{100\}$
  $\quad = \{111, 101, 110, 100\}$

  $\mathcal{S}_3 = \{111, 101, 110, 100\} \;\cup\; \{000, 001, 010, 011\}$
  $\quad = \{111, 101, 110, 100, 000, 001, 010, 011\}$

  $\mathcal{S}_i = \mathcal{S}_3 \qquad (i > 3)$

- Hence $\forall s.$ **EF**$(\texttt{Atom}(\lambda(\texttt{dreq},\texttt{q0},\texttt{dack}).\; \texttt{dreq} \wedge \texttt{q0} \wedge \texttt{dack}))(\mathcal{R}_{\texttt{RECEIVER}}, s)$

Symbolic model checking

- Represent sets of states with BDDs
- Represent Transition relation with a BDD
- If BDDs of $P(\mathcal{R}, s)$, $Q(\mathcal{R}, s)$ are known, then BDDs of
  $\neg P(\mathcal{R}, s)$
  $P(\mathcal{R}, s) \wedge Q(\mathcal{R}, s)$
  $P(\mathcal{R}, s) \vee Q(\mathcal{R}, s)$
  $P(\mathcal{R}, s) \Rightarrow Q(\mathcal{R}, s)$
  can be computed using standard BDD algorithms
- If BDDs of $P(\mathcal{R}, s)$, $Q(\mathcal{R}, s)$ are known, then BDDs of
  $\mathbf{AX}P(\mathcal{R}, s)$, $\mathbf{EX}P(\mathcal{R}, s)$, $\mathbf{A}[P\ \mathbf{U}\ Q](\mathcal{R}, s)$, $\mathbf{E}[P\ \mathbf{U}\ Q](\mathcal{R}, s)$
  computed using fairly straightforward algorithms (see textbooks)
- Model checking CTL generalises iteration for reachable states (`AG`)

History of Model checking

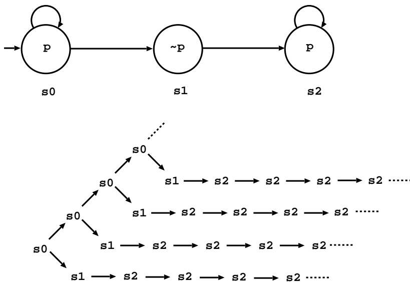
- CTL model checking invented by Emerson, Clarke and Sifakis
- Use of BDDs to represent and compute sets of states is called *symbolic model checking*
- Independently discovered by several people:
  Clarke & McMillan
  Coudert, Berthet & Madre
  Pixley
- SMV (McMillan) is a popular symbolic model checker

```
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 **p** *is always true on that path"*
- Consider



- Property true, but cannot be expressed in CTL
  - would need something like AF P
  - where P is something like "**property 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 (`Atom` p) is false (consider path `s0s0s0`$\cdots$)

Linear Temporal Logic (LTL)

- CTL property is a predicate on a state in a tree: $P(\mathcal{R}, s)$
- LTL property is a predicate on a path: $P(\sigma)$
- Syntax of LTL well-formed formulae:

wff ::=	`Atom`(p)	(Atomic formula)
|	$\neg$*wff*	(Negation)
|	$wff_1 \vee wff_2$	(Disjunction)
|	**X***wff*	(successor)
|	**F***wff*	(sometimes)
|	**G***wff*	(always)
|	$[wff_1\ \mathbf{U}\ wff_2]$	(Until)

- **Note:** no path quantifiers A or E

Semantics of LTL (shallow embedding)

- **Define** $\text{Tail}\ m\ \sigma\ =\ \lambda n.\ \sigma(n{+}m)$
- **Define:**

$$
\begin{array}{ll}
\texttt{Atom}(p) & = \lambda\sigma.\ p(\sigma(0)) \\
\neg P & = \lambda\sigma.\ \neg(P\ \sigma) \\
P \vee Q & = \lambda\sigma.\ P\ \sigma\ \vee\ Q\ \sigma \\
\mathbf{X}P & = \lambda\sigma.\ P(\text{Tail}\ 1\ \sigma) \\
\mathbf{F}P & = \lambda\sigma.\ \exists m.\ P(\text{Tail}\ m\ \sigma) \\
\mathbf{G}P & = \lambda\sigma.\ \forall m.\ P(\text{Tail}\ m\ \sigma) \\
[P\ \mathbf{U}\ Q] & = \lambda\sigma.\ \exists i.\ Q(\text{Tail}\ i\ \sigma)\ \wedge\ \forall j.\ j < i\ \Rightarrow\ P(\text{Tail}\ j\ \sigma)
\end{array}
$$

- **Example:**

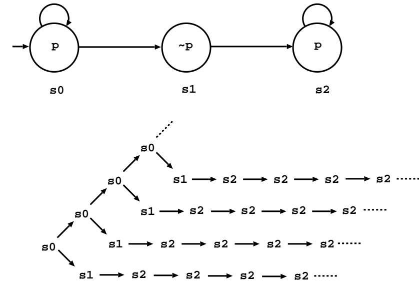
$$\mathbf{X}(\texttt{Atom}(p))(\sigma)\ =\ \texttt{Atom}(p)(\text{Tail}\ 1\ \sigma)\ =\ p(\text{Tail}\ 1\ \sigma\ 0)\ =\ p(\sigma(0{+}1)) = p(\sigma(1))$$

FG

- **FG**P **is true if there is a point after which** P **is always true**

$$
\begin{array}{l}
\mathbf{FG}P(\sigma) \\
\quad = \mathbf{F}(\mathbf{G}(P))(\sigma) \\
\quad = \exists m_1.\ (\mathbf{G}(P))(\text{Tail}\ m_1\ \sigma) \\
\quad = \exists m_1.\ \forall m_2.\ P(\text{Tail}\ m_2\ (\text{Tail}\ m_1\ \sigma)) \\
\quad = \exists m_1.\ \forall m_2.\ P(\text{Tail}\ (m_1{+}m_2)\ \sigma)
\end{array}
$$

- **Recall:**



- **LTL can express things that CTL can't express**

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*

- **Two kinds of formulas: state formulas** (*swff*) **& path formulas** (*pwff*)
  - **state formulas are true of a state** s **in a tree** $\mathcal{R}$ .... $\lambda(\mathcal{R}, s)$ **like CTL**
  - **path formulas are true of a path** σ **through a tree** $\mathcal{R}$ .... $\lambda(\mathcal{R}, \sigma)$ **like LTL**
- **Defined mutually recursively**

swff ::=	$\texttt{Atom}(p)$	(Atomic formula)
|	$\neg$*swff*	(Negation)
|	$swff_1 \vee swff_2$	(Disjunction)
|	**A***pwff*	(All paths)
|	**E***pwff*	(Some paths)
pwff ::=	$\texttt{PathForm}(swff)$	(Every state formula is a path formula)
|	$\neg$*pwff*	(Negation)
|	$pwff_1 \vee pwff_2$	(Disjunction)
|	**X***pwff*	(Successor)
|	**F***pwff*	(Sometimes)
|	**G***pwff*	(Always)
|	$[pwff_1\ \mathbf{U}\ pwff_2]$	(Until)

- **CTL is CTL* restricted with X, F, G, [–U–] preceded by A or E**
- **LTL consists of CTL* formulas of form A***pwff***,**
  where the only state formulas in *pwff* **are atomic**
- **Selection of primitives above arbitrary:** $\vee$, $\neg$, **X, U, E enough**

CTL* semantics

- Combining state semantics of CTL with path semantics of LTL:

$$
\begin{aligned}
\texttt{Atom}(p) &= \lambda(\mathcal{R}, s).\ p(s) \\
\neg S &= \lambda(\mathcal{R}, s).\ \neg(S(\mathcal{R}, s)) \\
S_1 \vee S_2 &= \lambda(\mathcal{R}, s).\ S_1(\mathcal{R}, s) \ \vee\ S_2(\mathcal{R}, s) \\
\mathbf{A}P &= \lambda(\mathcal{R}, s).\ \forall\sigma.\ \texttt{Path}(\mathcal{R}, s)\sigma \ \Rightarrow\ P(\mathcal{R}, \sigma) \\
\mathbf{E}P &= \lambda(\mathcal{R}, s).\ \exists\sigma.\ \texttt{Path}(\mathcal{R}, s)\sigma \ \wedge\ P(\mathcal{R}, \sigma)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{PathForm}(S) &= \lambda(\mathcal{R}, \sigma).\ S(\mathcal{R}, \sigma(0)) \\
\neg P &= \lambda(\mathcal{R}, \sigma).\ \neg(P(\mathcal{R}, \sigma)) \\
P_1 \vee P_2 &= \lambda(\mathcal{R}, \sigma).\ P_1(\mathcal{R}, \sigma) \ \vee\ P_2(\mathcal{R}, \sigma) \\
\mathbf{X}P &= \lambda(\mathcal{R}, \sigma).\ P(\mathcal{R}, \text{Tail } 1\ \sigma) \\
\mathbf{F}P &= \lambda(\mathcal{R}, \sigma).\ \exists m.\ P(\mathcal{R}, \text{Tail } m\ \sigma) \\
\mathbf{G}P &= \lambda(\mathcal{R}, \sigma).\ \forall m.\ P(\mathcal{R}, \text{Tail } m\ \sigma) \\
[P_1\ \mathbf{U}\ P_2] &= \lambda(\mathcal{R}, \sigma).\ \exists i.\ P_2(\mathcal{R}, \text{Tail } i\ \sigma) \ \wedge\ \forall j.\ j < i \ \Rightarrow\ P_1(\mathcal{R}, \text{Tail } j\ \sigma)
\end{aligned}
$$

- Note semantics of state and path formulas have different types
  - $\lambda(\mathcal{R}, s)$ versus $\lambda(\mathcal{R}, \sigma)$
- Semantics looks simpler if we assume $\mathcal{R}$ fixed

Simplified CTL* semantics (textbook semantics)

- Let $\texttt{Path}\ s\ \sigma$ abbreviate $\texttt{Path}(\mathcal{R}, s)\sigma$, then:

$$
\begin{aligned}
\texttt{Atom}(p) &= \lambda s.\ p(s) \\
\neg S &= \lambda s.\ \neg(S\ s) \\
S_1 \vee S_2 &= \lambda s.\ S_1\ s \ \vee\ S_2\ s \\
\mathbf{A}P &= \lambda s.\ \forall\sigma.\ \texttt{Path}\ s\ \sigma \ \Rightarrow\ P\ \sigma \\
\mathbf{E}P &= \lambda s.\ \exists\sigma.\ \texttt{Path}\ s\ \sigma \ \wedge\ P\ \sigma
\end{aligned}
$$

$$
\begin{aligned}
\texttt{PathForm}(S) &= \lambda\sigma.\ S(p(0)) \\
\neg P &= \lambda\sigma.\ \neg(P\sigma) \\
P_1 \vee P_2 &= \lambda\sigma.\ P_1\ \sigma \ \vee\ P_2\ \sigma \\
\mathbf{X}P &= \lambda\sigma.\ P(\text{Tail } 1\ \sigma) \\
\mathbf{F}P &= \lambda\sigma.\ \exists m.\ P(\text{Tail } m\ \sigma) \\
\mathbf{G}P &= \lambda\sigma.\ \forall m.\ P(\text{Tail } m\ \sigma) \\
[P_1\ \mathbf{U}\ P_2] &= \lambda\sigma.\ \exists i.\ P_2(\text{Tail } i\ \sigma) \ \wedge\ \forall j.\ j < i \ \Rightarrow\ P_1(\text{Tail } j\ \sigma)
\end{aligned}
$$

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 $\mathbf{G}(\mathbf{F}\ P) \ \Rightarrow\ \mathbf{G}(\mathbf{F}\ Q)$
- Can't say this in CTL
  - why not – what's wrong with $\mathbf{AG}(\mathbf{AF}\ P) \ \Rightarrow\ \mathbf{AG}(\mathbf{AF}\ Q)$?
  - in CTL* expressible as $\mathbf{A}(\mathbf{G}(\mathbf{F}\ P) \ \Rightarrow\ \mathbf{G}(\mathbf{F}\ Q))$
  - fair CTL model checking is implemented in the model checking algorithm
  - fair LTL just needs a fairness assumption like $\mathbf{G}(\mathbf{F}\ P) \ \Rightarrow\ \cdots$
- Fairness is a tricky and subtle subject
  - several notions or fairness: 'weak fairness', 'strong fairness' etc
  - exist whole books on fairness

Propositional modal μ-calculus

- Modal μ-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 μ-calculus
- Strictly stronger than CTL*
  - expressibility in μ-calculus strictly increases as allowed nesting increases
  - need fixed point operators nested 2 deep for CTL*
- The μ-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 infinte 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**

ITL (simplified and with expressions omitted)

- **Syntax of ITL well-formed formulae:**

wff ::=	$\texttt{Atom}(p)$	(Atomic formula)
|	**true**	(Truth)
|	$\neg wff$	(Negation)
|	$wff_1 \lor wff_2$	(Disjunction)
|	**skip**	(interval with exactly two states)
|	$wff_1 \ ; \ wff_2$	(Chop)
|	$wff*$	(Repeat)

- **Semantics (properties are predicates on intervals):**

$\texttt{Atom}(p) = \lambda\langle s_0 \cdots s_n\rangle.\ p(s_0)$

$\mathbf{true} = \lambda\langle s_0 \cdots s_n\rangle.\ T$

$\neg P = \lambda\langle s_0 \cdots s_n\rangle.\ \neg(P\langle s_0 \cdots s_n\rangle)$

$P \lor Q = \lambda\langle s_0 \cdots s_n\rangle.\ P\langle s_0 \cdots s_n\rangle \ \lor \ Q\langle s_0 \cdots s_n\rangle$

$\mathbf{skip} = \lambda\langle s_0 \cdots s_n\rangle.\ n = 1$

$P \ ; \ Q = \lambda\langle s_0 \cdots s_n\rangle.\ \exists k.\ k \leq n \ \land \ P\langle s_0 \cdots s_k\rangle \ \land \ Q\langle s_k \cdots s_n\rangle$

$P* = \lambda\langle s_0 \cdots s_n\rangle.$
$\exists w_1 \ \cdots \ w_l.\ \langle s_0 \cdots s_n\rangle \ = w_1 \cdots w_l \ \land \ P\ w_1 \ \land \ \cdots \ \land \ P\ w_l$

Examples of ITL

Abbreviation	**Meaning**
$P_1 ; P_2$	P_1 holds then P_2 holds (overlapping state)
$P_1 ; \mathbf{skip} ; P_2$	P_1 holds then P_2 holds (no overlapping state)
$\mathbf{skip} ; P$	P true on the next state
$\mathbf{true} ; P$	P sometimes true
$\neg\mathbf{true} ; \neg P$	P always true

Too many logics: CTL, LTL, 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 Commitee
  - 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

Previous notation	PSL ASCII notation	
$P \wedge Q$	P `&` Q	
$P \Rightarrow Q$	P `->` Q	
$\neg P$	`!`P	(exclamation mark is negation)
$\mathbf{X}P$	`next` P	
$\mathbf{F}P$	`eventually!` P	(exclamation mark is not negation)
$\mathbf{G}P$	`always` P	
$[P\ \mathbf{U}\ Q]$	P `until!` Q	
$[P\ \mathbf{W}\ Q]$	P `until` Q	
skip	`true`	
$R*$	R`[*]`	
R_1 ; R_2	R_1 `:` R_2	
R_1 ;**skip**; R_2	R_1 `;` R_2	

Sequential Extended Regular Expressions (SEREs)

- **Similar to ITL – but weaker**
- **On earlier slide:** R`[*]`, R_1`:`R_2, R_1`;`R_2
- **Other SERE operators include**

  R_1 `|` R_2 — either R_1 or R_2 holds

  R_1 `&&` R_2 — both R_1 and R_2 hold for same number of cycles

  R_1 `&` R_2 — both R_1 and R_2 hold, but one may finish before the other
- **Actually `&` is not primitive (braces { and } used for grouping)**

```
{r1} & {r2}  =  {{r1} && {r2;true[*]}} | {{r1;true[*]} ** {r2}}
```

- **SEREs can be used to improve readability of formulas, compare:**

```
always
 (reqin -> next(ackout -> next(!abortin -> (ackin & next ackin))))
```

with

```
always {reqin;ackout;!abortin} |-> {ackin;ackin}
```

where PSL formulas r_1 `|->` r_2 **defined later**

SEREs in HOL

- **Syntax :**

r ::=	`Atom`(p)	(Atomic formula)
|	r_1 `|` r_2	(Disjunction)
|	r_1 `;` r_2	(Concatenation)
|	r_1 `:` r_2	(Fusion: ITL's chop)
|	r_1 `&&` r_2	(Length matching conjunction)
|	r`[*]`	(Repeat)

- **Semantics:**
  (s ranges over states; w ranges over finite lists of states; "head" denotes head of a list; infix "." denotes concatenation)

`Atom`(p) $= \lambda w.\ p(\mathbf{head}\ w)$

r_1 `|` r_2 $= \lambda w.\ r_1\ w \ \vee\ r_2\ w$

r_1 `;` r_2 $= \lambda w.\ \exists w_1\ w_2.\ w = w_1.w_2 \ \wedge\ r_1\ w_1 \ \wedge\ r_2\ w_2$

r_1 `:` r_2 $= \lambda w.\ \exists w_1\ s\ w_2.\ w = w_1.s.w_2 \ \wedge\ r_1(w_1.s) \ \wedge\ r_2(s.w_2)$

r_1 `&&` r_2 $= \lambda w.\ r_1\ w \ \wedge\ r_2\ w$

r`[*]` $= \lambda w.\ w = \langle\rangle \vee \exists w_1 \cdots w_l.\ w = w_1.\cdots.w_l \ \wedge\ r\ w_1 \ \wedge\ \cdots \ \wedge\ r\ w_l$

PSL Foundation Language (FL)

- **Syntax:**

f ::=	`Atom`(p)	(Atomic formula)
|	$\neg f$	(Negation)
|	$f_1 \vee f_2$	(Disjunction)
|	`next` f	(successor)
|	$\{r\}(f)$	(Suffix implication)
|	$\{r_1\}$ `|->` $\{r_2\}$	(Suffix next implication)
|	$[f_1$ `until` $f_2]$	(Until)

- **Semantics**
  (simplified – no clocking, weak/strong distinction omitted):

`Atom`(p) $= \lambda\sigma.\ p(\sigma(0))$

$\neg f$ $= \lambda\sigma.\ \neg(f\ \sigma)$

$f_1 \vee f_2$ $= \lambda\sigma.\ f_1\ \sigma \ \vee\ f_2\ \sigma$

`next` f $= \lambda\sigma.\ f(\mathrm{Tail}\ 1\ (\sigma))$

$\{r\}(f)$ $= \lambda\sigma.\ \exists w\ \sigma'.\ \sigma = w.\sigma' \ \wedge\ r\ w \ \wedge\ f\ \sigma'$

$\{r_1\}$ `|->` $\{r_2\}$ $= \lambda\sigma.\ \exists w_1\ \sigma'.\ \sigma = w_1.\sigma' \ \wedge\ r_1\ w_1 \ \Rightarrow\ \exists w_2\ \sigma''.\ \sigma' = w_2.\sigma'' \ \wedge\ r_2\ w_2$

$[f_1$ `until` $f_2]$ $= \lambda\sigma.\ \exists i.\ f_2(\mathrm{Tail}\ i\ \sigma) \ \wedge\ \forall j.\ j < i \ \Rightarrow\ f_1(\mathrm{Tail}\ j\ \sigma)$

- **There is also an Optional Branching Extension (OBE)**
  - **completely standard CTL: EX, E[–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 `req` *is asserted, followed four cycles later by an assertion of* `grant`*, followed by a cycle in which* `abortin` *is not asserted, we expect to see an assertion of* `ack` *some time in the future.*
- **Can represent by**

```
always {req;[*3];grant;!abortin}(eventually! ack)
```

- **where** `eventually!` **is LTL future operator** `F`**, so:**

```
eventually! f  =  [T U f]  =  [true until! f]
```

- **N.B. suffix "`!`" denotes "strong"**
  - strong/weak distinction not covered here – important for dynamic checking
  - gives semantics when simulator halts before an expected event occurs

SERE examples

- **How can we modify**

```
always {reqin;ackout;!abortin} |-> {ackin;ackin}
```

  so that the two cycles of `ackin` **start the cycle after** `!abortin`**?**

- **Two ways of doing this**

```
always {reqin;ackout;!abortin} |-> {true;ackin;ackin}

always {reqin;ackout;!abortin} |=> {ackin;ackin}
```

- `|=>` **is a defined operator**

```
{r1} |=> {r2}  =  {r1} |-> {true;r2}
```

- **Note:** `true` **and** `T` **are synonyms**

Examples of defined notations: consecutive repetition

- **Define**

```
r[+]       = {r;r[*]}
             ┌ false[*]      if i=0
r[*i]      = │
             └ {r;r;...;r}  otherwise (i repetitions of r)
r[*i..j]   = {r[*i]} | {r[*(i+1)]} | ... | {r[*j]}
[+]        = true[+]
[*]        = true[*]
```

- **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 consecutive data transfers, followed by the assertion of signal* `end_trans`*. A data transfer is indicated by the assertion of signal* `data`

```
always {req;ack} |=> {start_trans;data[*1..8];end_trans}
```


Fixed 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 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`
- **Can represent by**

```
always {req;ack} |=>
       {start_trans;{{!data[*];data}[*8];!data[*]};end_trans}
```

- **Define**

```
b[= i] = {!b[*];b}[*i];!b[*]
```

- **Then have a nicer representation**

```
always {req;ack} |=> {start_trans;data[= 8];end_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]} | {b[= (i+1)]} | ... | {b[= j]}
```

- Then

```
always{req;ack} |=> {start_trans;data[= 1..8];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 (f`@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` $\longrightarrow$ `{!clk[*];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

```
assert always (!en1 & en2))
   |      |         |
   |      |         |
   |      |         |--- Boolean layer
   |      |
   |      |-------------- 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:

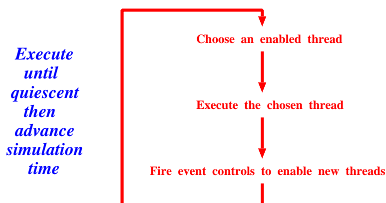
  `always @(`v_1 `or` $\cdots$ `or` v_n`)` v `:=` E
  - enabled by any change to $v_1, \ldots, v_n$
- Positive edge triggered sequential threads:

  `always @(posedge` clk`)` v `:=` E
  - enabled by clk changing to `T`
- Negative edge triggered sequential threads:

  `always @(negedge` clk`)` v `:=` E
  - enabled by clk changing to `F`

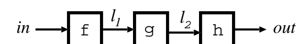
Simulation

- 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



- HDL-like specification:

  `always @(`in`)` l_1 `:= f(`in`)` ..... thread T1
  `always @(`l_1`)` l_2 `:= g(`l_1`)` ..... thread T2
  `always @(`l_2`)` out `:= h(`l_2`)` ..... thread T3

- Suppose in **changes** to v at simulation time t
  - T1 will become enabled and assign `f`(v) to l_1
  - if l_1's value changes then T2 will become enabled
    (still simulation time t)
  - T2 will assign `g(f`(v)) to l_2
  - if l_2's value changes then T will become enabled
    (still simulation time t)
  - T3 will assign `h(g(f`(v))) to out
  - simulation quiesces
    (still simulation time t)
- Steps at same simulation time happen in δ-*time*
  (VHDL jargon)

Semantic gap

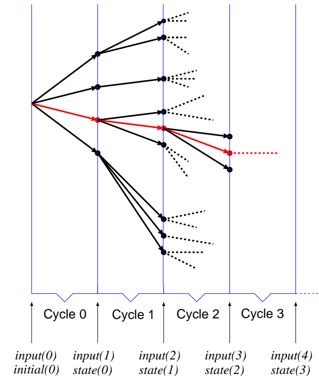
- Designers use HDLs and verify via simulation
  - event semantics
- Formal verifiers use logic and verify via proof
  - trace semantics
- **Problem:** show consistency between semantics
- Goal:

traces = 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 ⇒ trace semantics (partial correctness)
  - simulation always termininates"quiesces" (total correctness)
  - extend to sequential threads

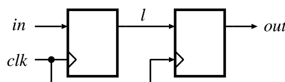
Trace defined by a simulation run

- Simulation defines a tree of states



- Inputs read at start of cycle
- State computed at end of cycle
- Traces = sequences of end-of-cycle states (example shown in red)
- Branching time

Sequential threads – event semantics

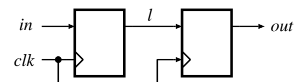


- Consider two Dtypes in series:

```
always @(posedge clk) l := in
always @(posedge clk) out := l
```

- If `posedge` *clk*:
  - both threads become enabled
  - race condition
- Right thread executed first:
  - *out* gets previous value of *l*
  - then left thread executed
  - so *l* gets value input at *in*
- Left thread executed first:
  - *l* gets input value at *in*
  - then right thread executed
  - so *out* gets input value at *in*

Sequential threads – trace semantics



- Trace semantics:

$(\forall t.\ l(t{+}1) \;=\; (\texttt{Rise}\ clk\ t\ \rightarrow\ in\ t\ \mid\ l\ t)) \wedge$

$(\forall t.\ out(t{+}1) \;=\; (\texttt{Rise}\ clk\ t\ \rightarrow\ l\ t\ \mid\ out\ t))$

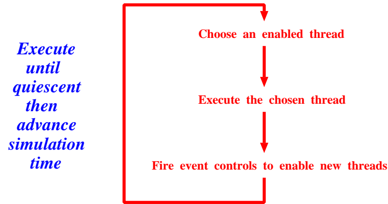
- Corresponds to right thread executed first
- How to ensure event and trace semantics agree?
- **Method 1:** use non-blocking assignments:

```
always @(posedge clk) l <= in;
always @(posedge clk) out <= l;
```

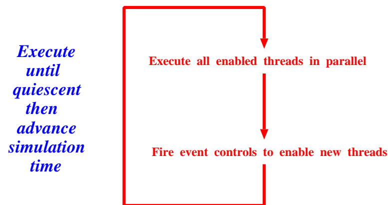
  - non-blocking assignments (<=) in Verilog
  - RHS of all non-blocking assignments first computed
  - assignments done at end of simulation cycle
- **Method 2:** make simulation cycle VHDL-like

Verilog versus VHDL simulation cycles

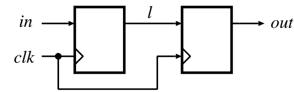
- Verilog-like simulation cycle:



- VHDL-like simulation cycle:



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 ($\exists$, $\wedge$)
  - 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!