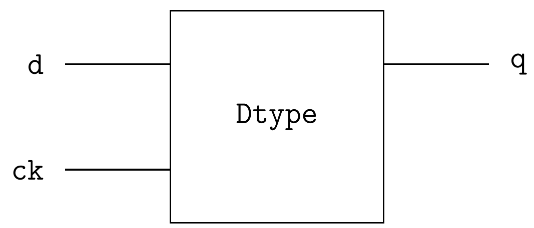
An edge-triggered Dtype

- **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**



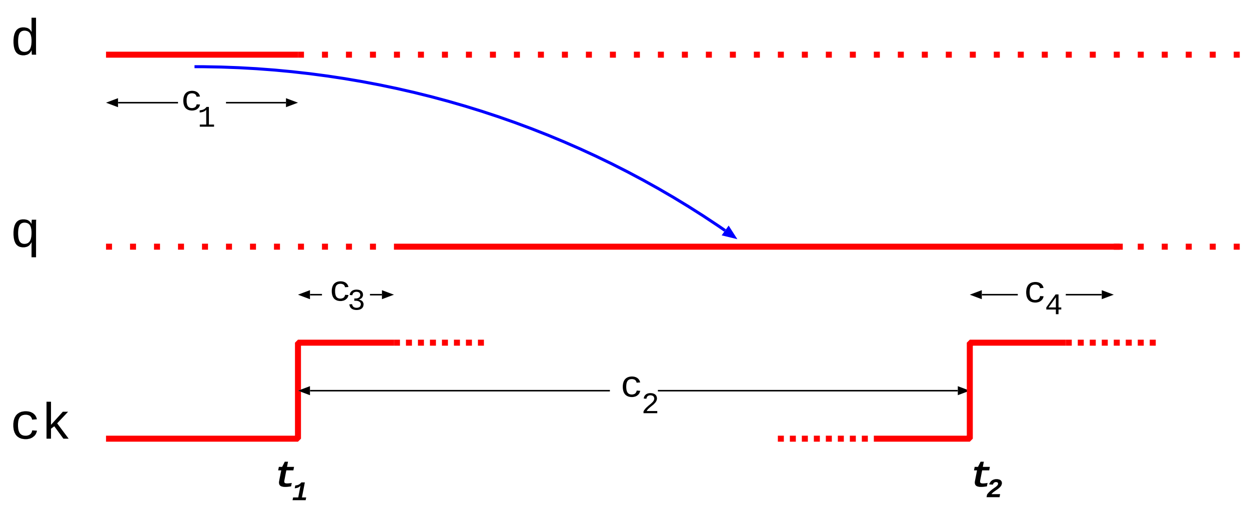
Specification of `Dtype`

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:**

  $\texttt{Rise}_1(f)(t) \ \equiv \ (f(t-1) = \texttt{F}) \ \wedge \ (f(t) = \texttt{T})$

- **Page 65:**

  $\texttt{Rise}_2(f)(t) = \neg f(t) \wedge f(t+1)$

- **However:**

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

  $\forall f \ t.\ t \geq 0 \Rightarrow \ (\texttt{Rise}_2(f)(t) = \texttt{Rise}_1(f)(t+1))$

- **In Accellera standard language PSL function** $\texttt{Rise}_1$ **is called** `Rose`

Some temporal operators in Higher Order Logic

- **Define:**

  $$\texttt{Next}(t_1, t_2)(f) \;\equiv\; t_1 < t_2 \quad \wedge \quad f(t_2) \quad \wedge \quad \forall t.\ t_1 < t \ \wedge\ t < t_2 \Rightarrow \neg f(t)$$

- **Define:**

  $$\texttt{Stable}(t_1, t_2)(f) \;\equiv\; \forall t.\ t_1 \leq t \ \wedge\ t < t_2 \Rightarrow (f(t){=}f(t_1))$$

- **These are raw higher order logic not temporal logic**
  - **various temporal logics are described later**

`Dtype` specification

- **Logic specification:**

$$
\begin{array}{l}
\mathtt{Dtype}(\mathtt{c}_1,\mathtt{c}_2,\mathtt{c}_3,\mathtt{c}_4)(d,ck,q) \;\equiv \\
\;\forall t_1\ t_2.\ \mathtt{Rise}_1(ck)(t_1) \wedge \\
\qquad\qquad \mathtt{Next}(t_1,t_2)(\mathtt{Rise}_1(ck)) \wedge \\
\qquad\qquad (t_2 - t_1 > \mathtt{c}_2) \wedge \\
\qquad\qquad \mathtt{Stable}(t_1 - \mathtt{c}_1, t_1 + 1)(d) \\
\qquad\qquad \Rightarrow \\
\qquad\qquad (\mathtt{Stable}(t_1 + \mathtt{c}_3, t_2 + \mathtt{c}_4)(q) \;\wedge\; (q(t_2) = d(t_1)))
\end{array}
$$

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

- **Note that**

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

  formed by applying

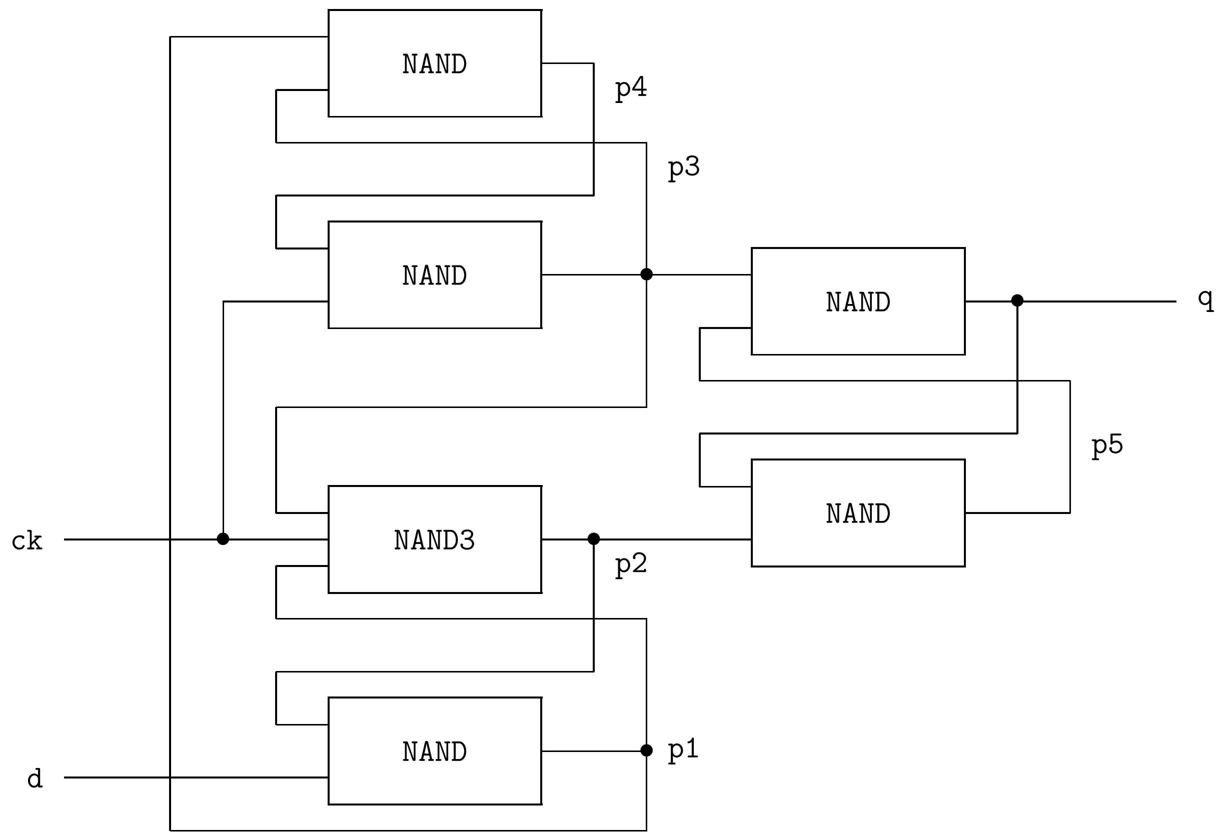
  $\mathtt{Next}(t_1,t_2)$

  to the predicate

  $\mathtt{Rise}_1(ck)$

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:**

$$\begin{array}{l}\texttt{Dtype\_Imp}(d, ck, q) \;\equiv \\ \;\exists p_1\ p_2\ p_3\ p_4\ p_5. \\ \qquad \texttt{NAND}(p_2, d, p_1) \;\wedge\; \texttt{NAND3}(p_3, ck, p_1, p_2) \;\wedge \\ \qquad \texttt{NAND}(p_4, ck, p_3) \;\wedge\; \texttt{NAND}(p_1, p_3, p_4) \;\wedge \\ \qquad \texttt{NAND}(p_3, p_5, q) \;\wedge\; \texttt{NAND}(q, p_2, p_5)\end{array}$$

- **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** n**th 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**

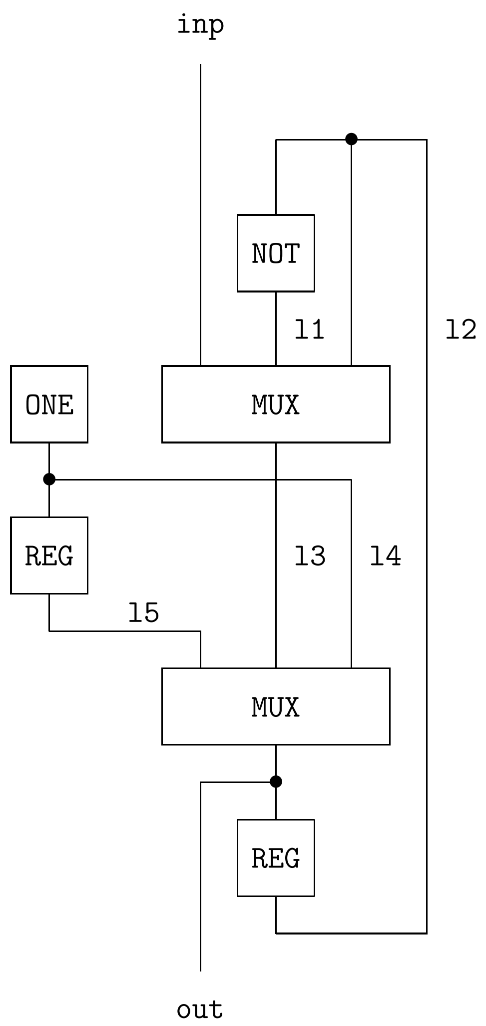
```
out 0 = T
------------------------------------
  0.  out 0 = T
  1.  ∀t. out(t+1) = if inp(t+1) then ¬(out t) else out t

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

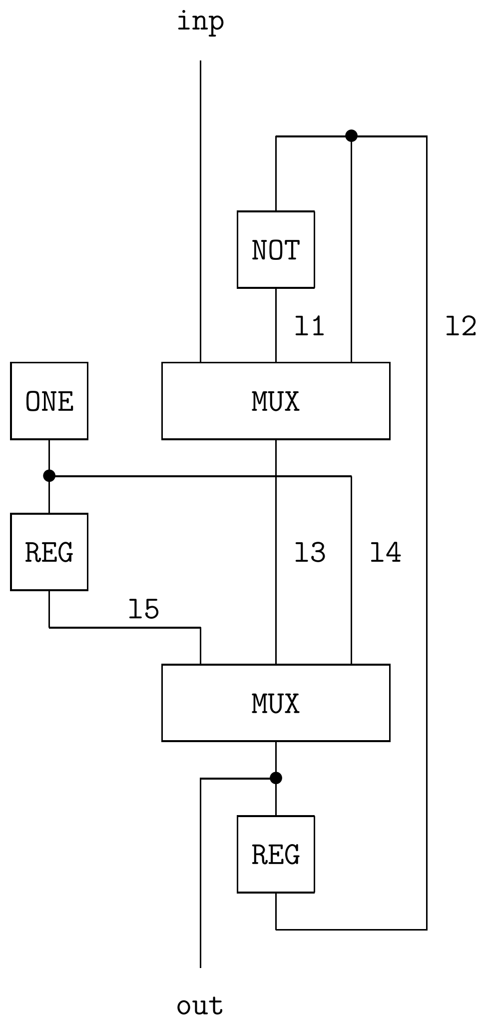
- **Assume registers 'power up' storing F**
- **Thus the output at time 0 cannot be taken directly from a register**
  - **because the output of the parity checker at time 0 is specified to be T**



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
```

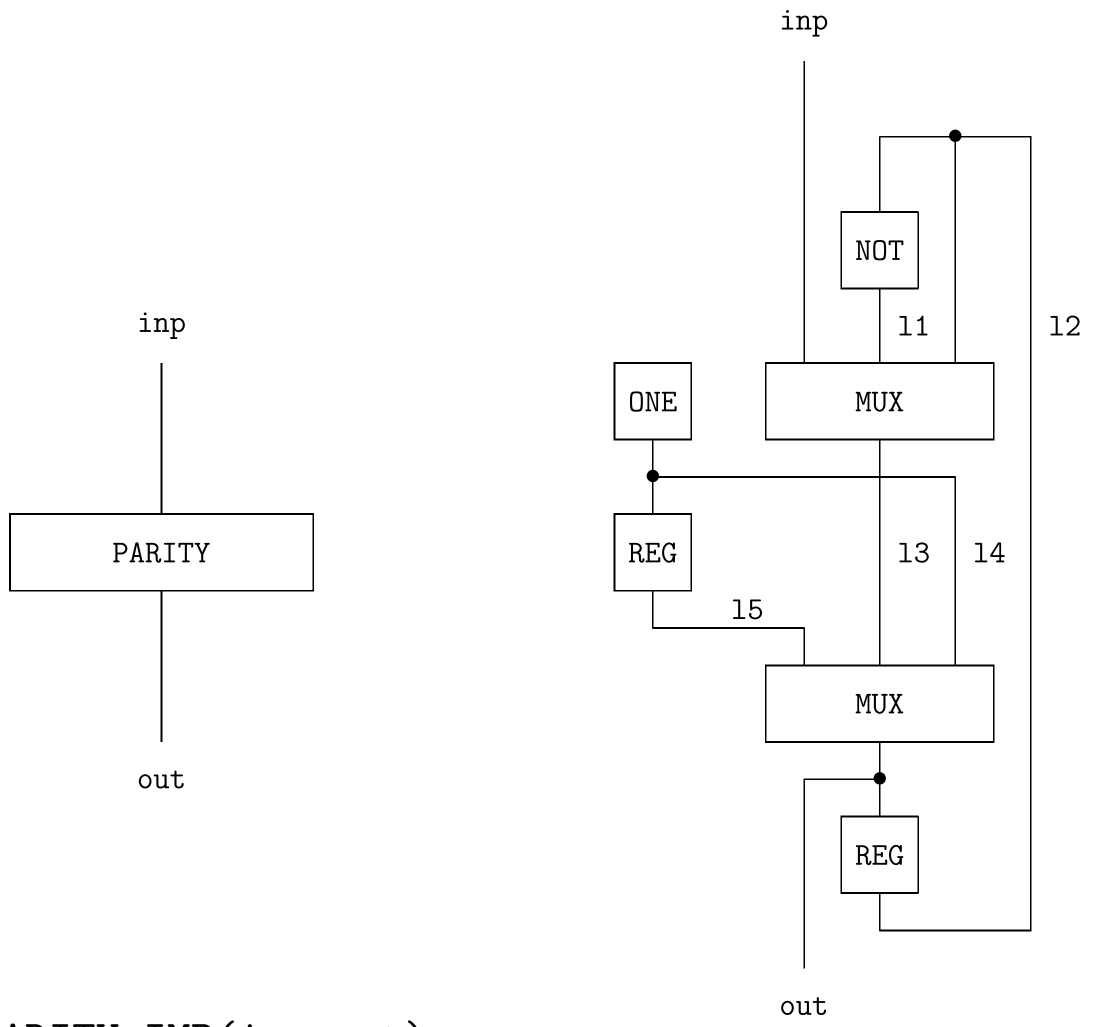
- **Hence by transitivity of ⇒**

```
 |- ∀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**

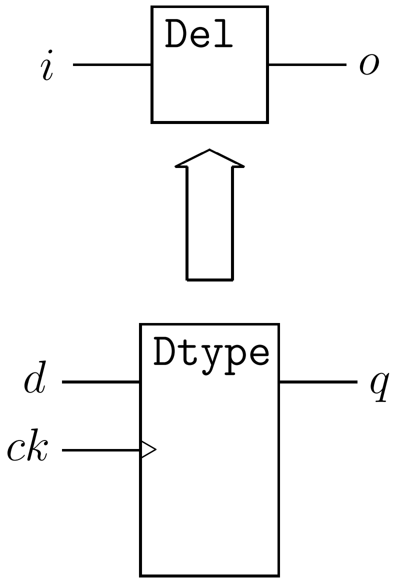
```
       inp
        |
        |    |--------|
        |    |        |
     |---------|      |
     |   XOR   |      |
     |---------|      |
          |           |
 l2 |-----|      l1   |
    |     |           |
    |  |-----|        |
    |  | REG |        |
    |  |-----|        |
    |     |           |
    |     |-----------|
    |
    |
 |-----|
 | NOT |
 |-----|
    |
   out
```

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

```
inp        =  T T F F T . . .
l1         =  F T F F F . . .
l2         =  T F F F T . . .
out        =  F T T T F . . .
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`**:**



$\vdash \texttt{Del}(i,o) = \forall t.\, o(t{+}1) = i\ t$

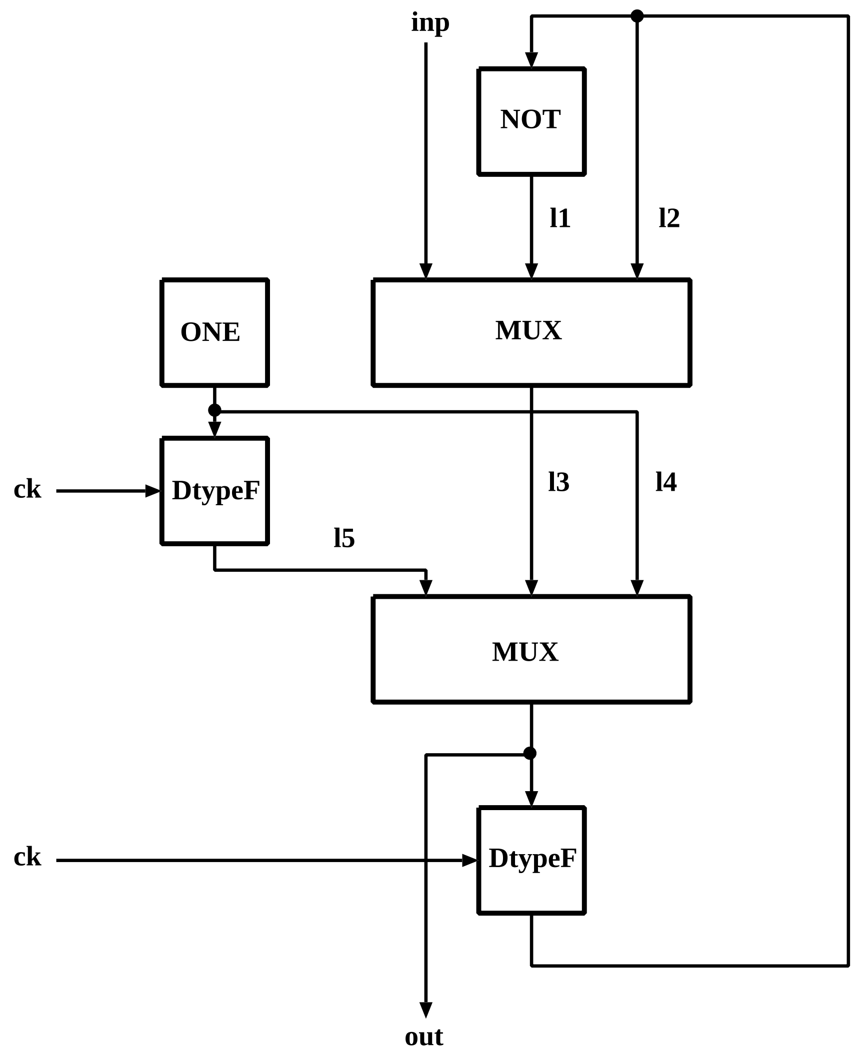
$\vdash \texttt{Rise}\ ck\ t \ = \ \neg ck(t) \wedge ck(t{+}1)$

$\vdash \texttt{Dtype}(ck,d,q) \ = \ \forall t.\, q(t{+}1) = (\texttt{Rise}\ ck\ t \rightarrow d\ t \mid q\ t)$

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

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

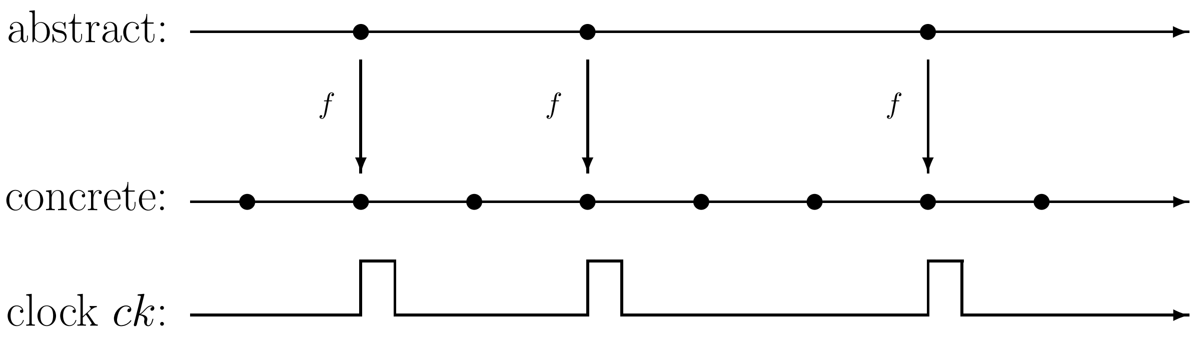
Trace level version of the Parity device




```
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:**

$(s\ \texttt{when}\ P)(n) = \textit{value of } s \textit{ at the concrete time } t \textit{ when } \texttt{P} \textit{ true for } n\textit{th time}$

$\vdash \texttt{Timeof P}\ n = \textit{the concrete time } t \textit{ when } \texttt{P} \textit{ true for } n\textit{th time}$

$\vdash s\ \texttt{when}\ P = s \circ (\texttt{Timeof}\ P)$

- **From Melham's Theorem:**

$$\begin{array}{l} \vdash \forall ck.\ \texttt{Inf}(\texttt{Rise}\ ck) \Rightarrow \\ \qquad \forall d\, q.\ \texttt{DtypeF}(ck, d, q) \Rightarrow \texttt{REG}(d\ \texttt{when}\ (\texttt{Rise}\ ck), q\ \texttt{when}\ (\texttt{Rise}\ ck)) \end{array}$$

- $\texttt{Inf}\ P$ **means "P true infinitely often"**

$$\texttt{Inf}\ P = \forall t.\, \exists\, t'.\ t' > t\ \wedge\ P\ t'$$

Digression on defining `Timeof`

- **How do we define the temporal abstraction function:**

  $\vdash$ `Timeof P` $n =$ *the concrete time* t_c *such that* `P` *true for* n*th 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:**

  $\vdash$ `Timeof P` $n =$ *the time* t_c *such that* `P` *true for* n*th 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` $\wedge$ `sig t2` $\wedge$ $\forall$`t. t1<t` $\wedge$ `t<t2` $\Rightarrow$ $\neg$`(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` $\wedge$ $\forall$`t'. t'<t` $\Rightarrow$ $\neg$`(sig t')))`
  $\wedge$
  `(IsTimeof (n+1) sig t =` $\exists$`t'. IsTimeof n sig t'` $\wedge$ `Next t' t sig)`

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

  `Timeof sig n =` ϵ`t. 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)
```

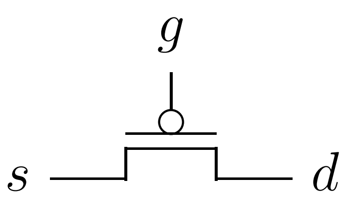
- **Hence, by transitivity of** ⇒

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

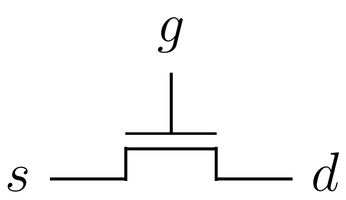
- **This is a typical correctness result using temporal abstraction**

NEW TOPIC: modelling transistors

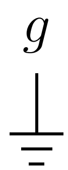
- **Recall simple switch model of CMOS**



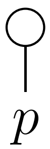
$\vdash \texttt{Ptran}(g, s, d) = (\neg g \Rightarrow (d = s))$



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



$\vdash \texttt{Gnd}\ g = (g = \texttt{F})$

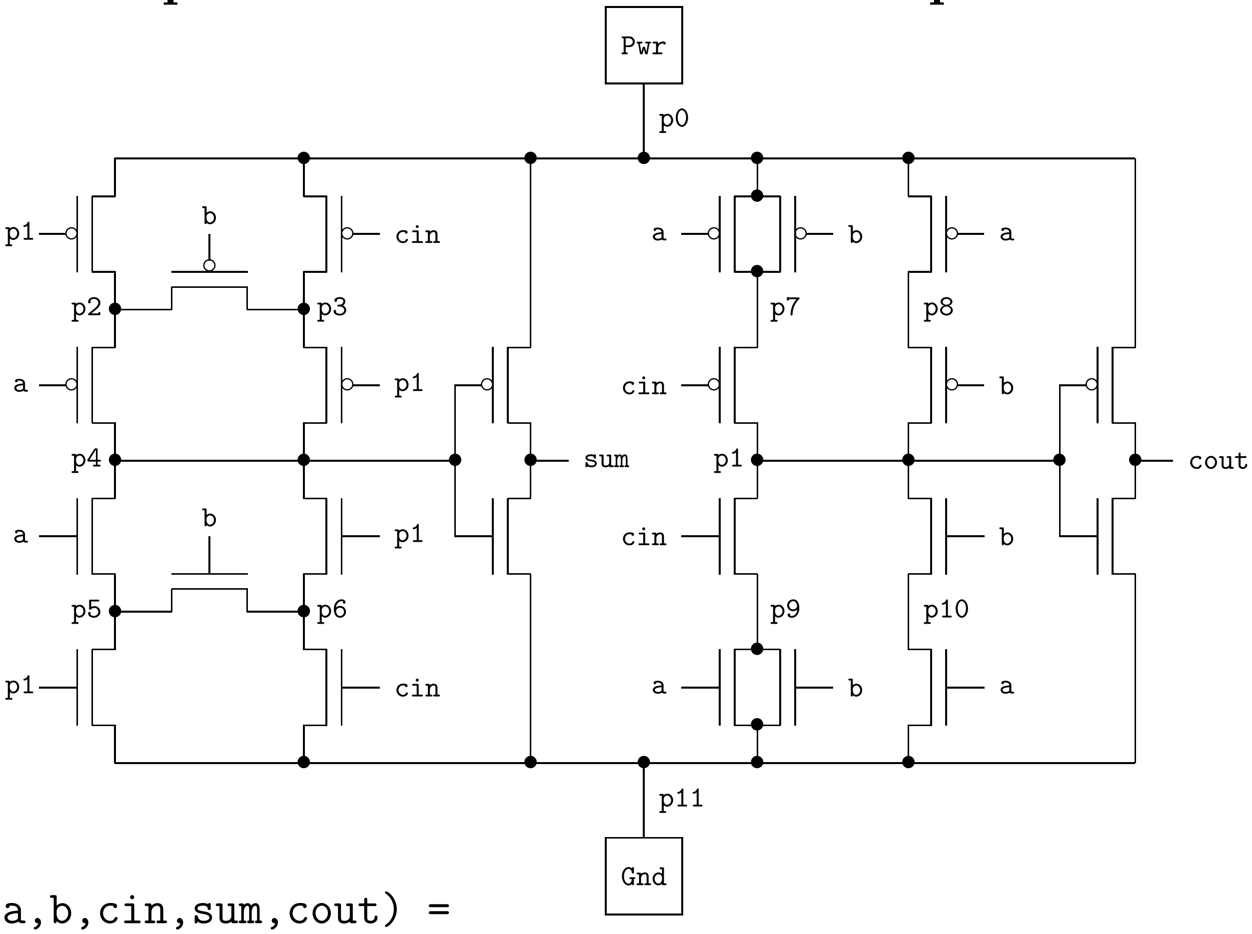


$\vdash \texttt{Pwr}\ p = (p = \texttt{T})$

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

The simple adder example

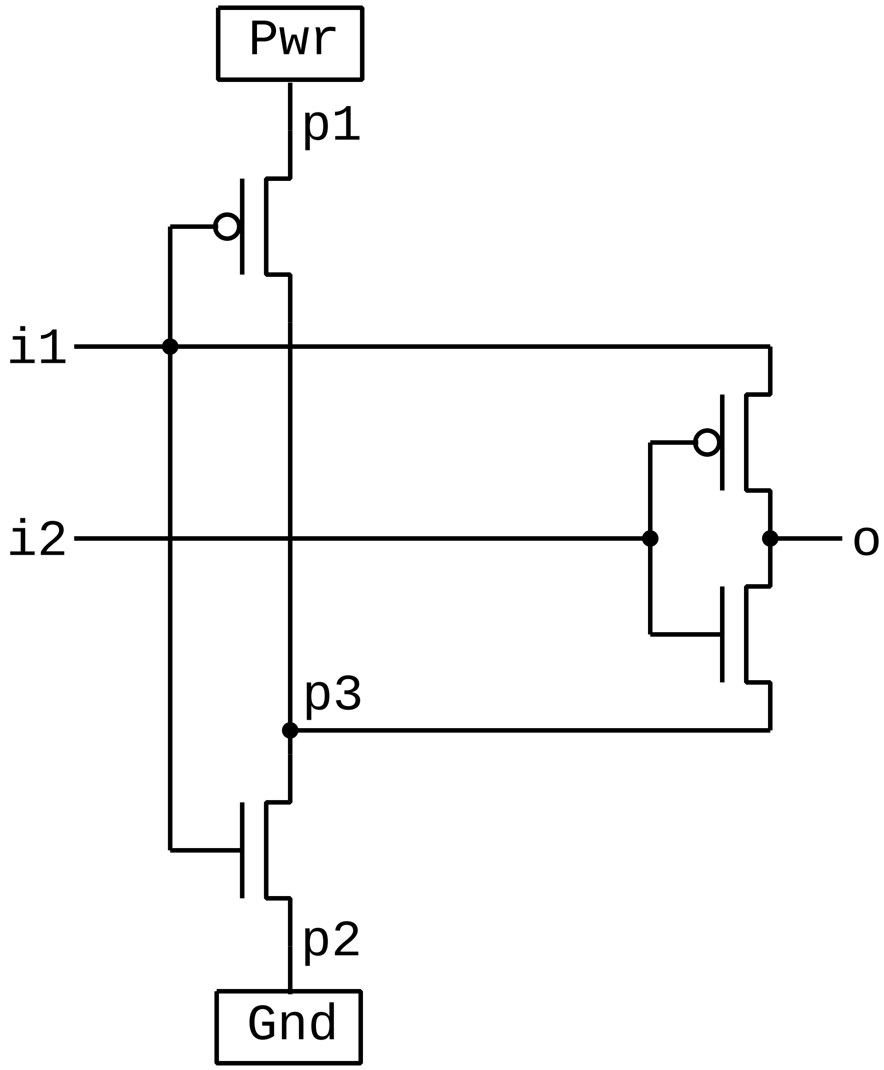
- **This example shows non-obvious examples can be analysed**

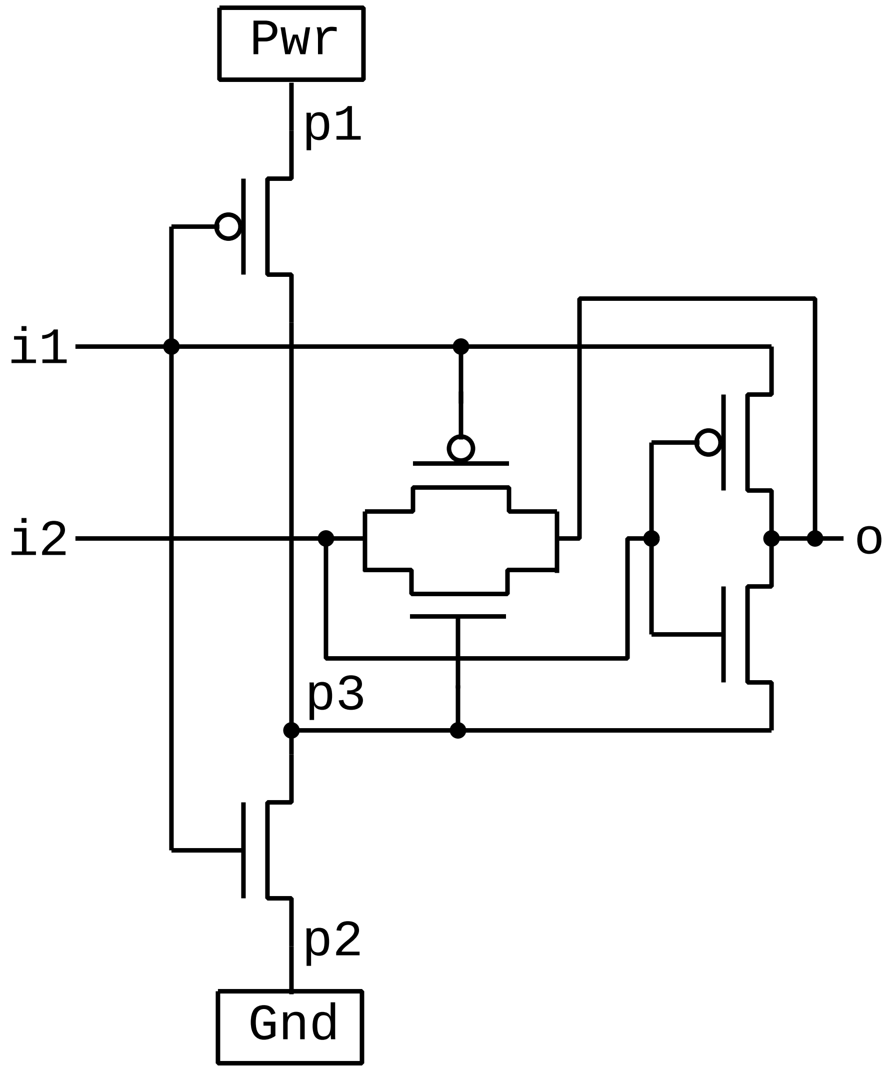



```
Add1(a,b,cin,sum,cout) =
      ∃p0 p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11.
        Ptran(p1,p0,p2)    ∧ Ptran(cin,p0,p3)  ∧ Ptran(b,p2,p3)    ∧
        Ptran(a,p2,p4)     ∧ Ptran(p1,p3,p4)   ∧ Ntran(a,p4,p5)    ∧
        Ntran(p1,p4,p6)    ∧ Ntran(b,p5,p6)    ∧ Ntran(p1,p5,p11)  ∧
        Ntran(cin,p6,p11)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

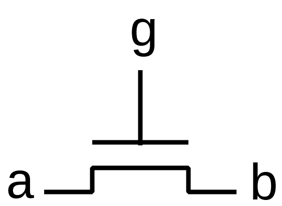




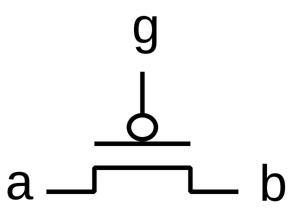
- **Equivalent in simple switch model!**

How transistors work

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

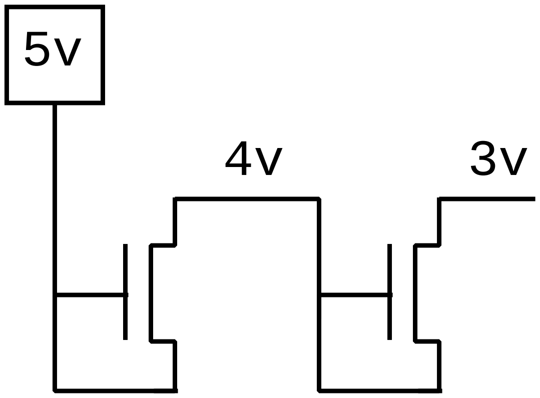


only conducts well if $V_g - V_a \geq V_{TH}$ or $V_g - V_b \geq V_{TH}$



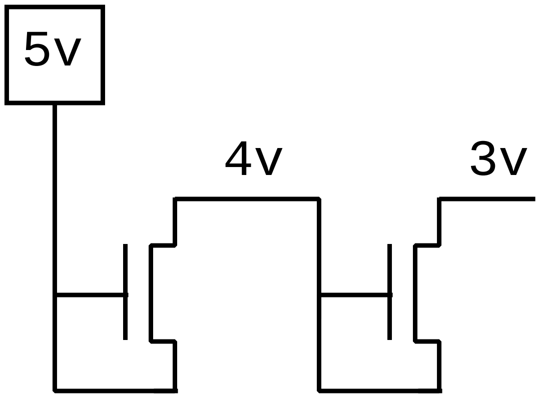
only conducts well if $V_a - V_g \geq V_{TH}$ or $V_b - V_g \geq V_{TH}$

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



- **Weak output may not be able to switch transistors**

What happens in the Simple Switch Model



- **From the definitions**

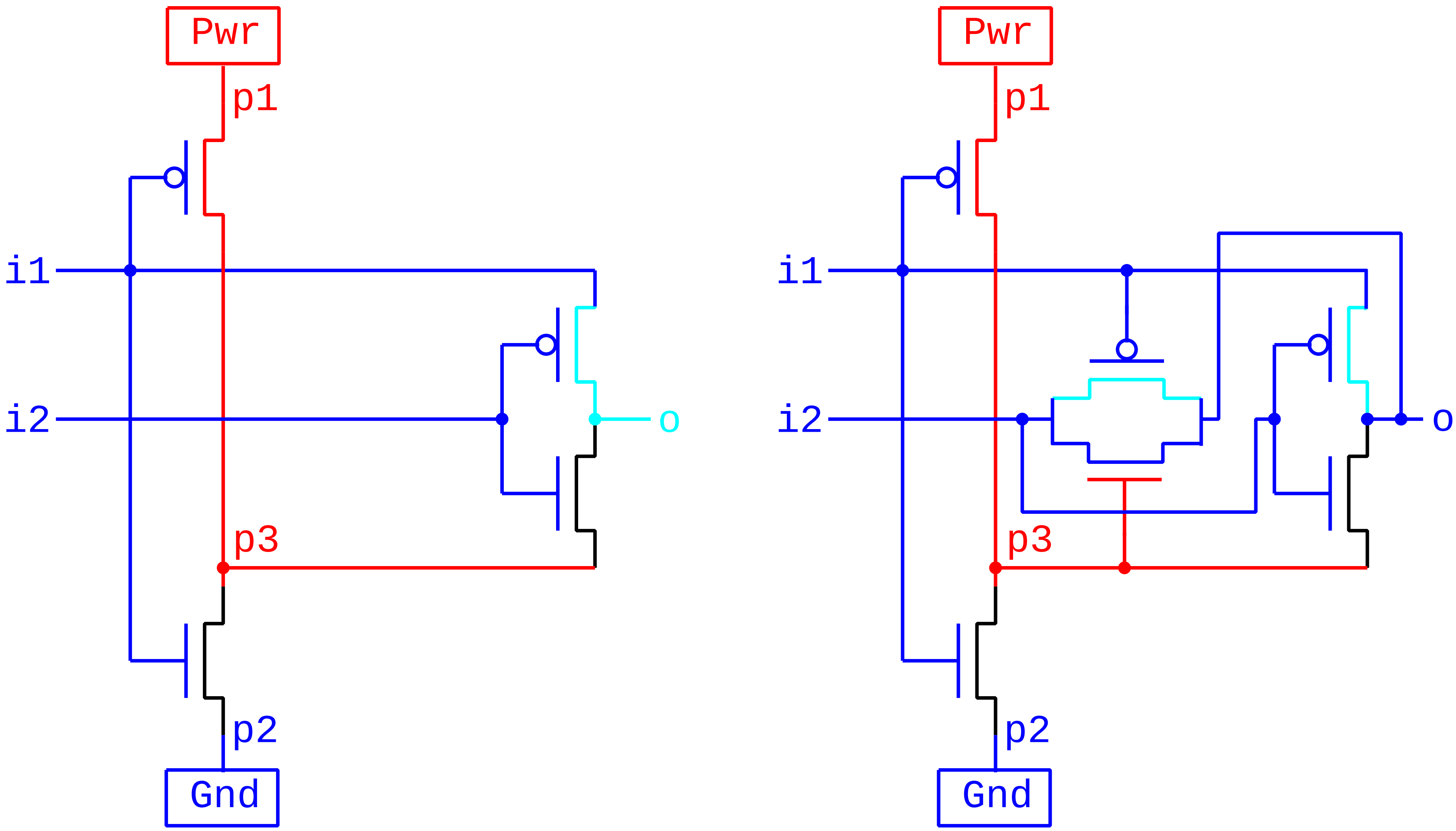
```
|- ∀p. Pwr p = (p = T)

|- ∀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

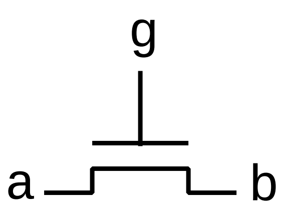
- Compare



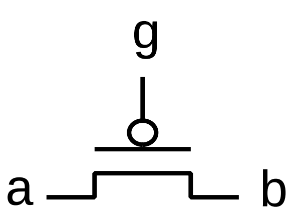
- Bad design has weak output
- Good design has strong output
- Need a better model to distinguish the designs

Difference switching model (Mike Fourman)

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

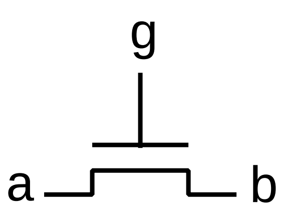



```
Ntran(g,a,b) = ((g=Hi) ∧ (a=Lo) ⇒ (b=Lo))
               ∧
               ((g=Hi) ∧ (b=Lo) ⇒ (a=Lo))
```

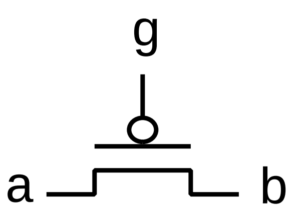



```
Ptran(g,a,b) = ((g=Lo) ∧ (a=Hi) ⇒ (b=Hi))
                ∧
               ((g=Lo) ∧ (b=Hi) ⇒ (a=Hi))
```

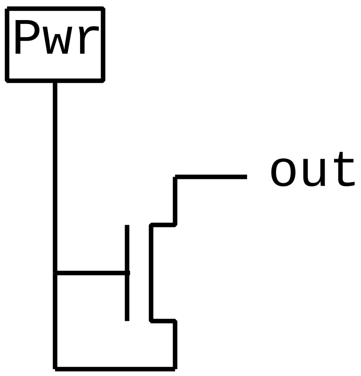
More compact definitions



$$\texttt{Ntran(g,a,b) = (g=Hi)} \Rightarrow \texttt{((a=Lo) = (b=Lo))}$$

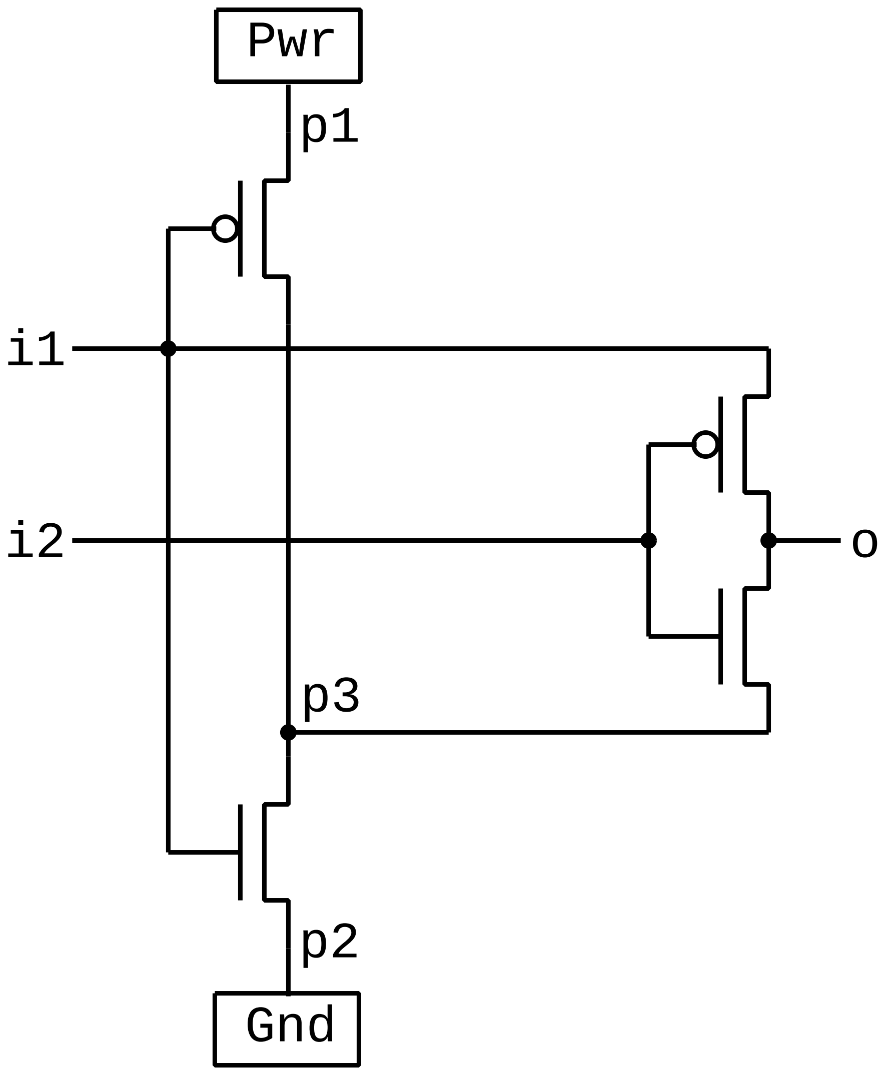


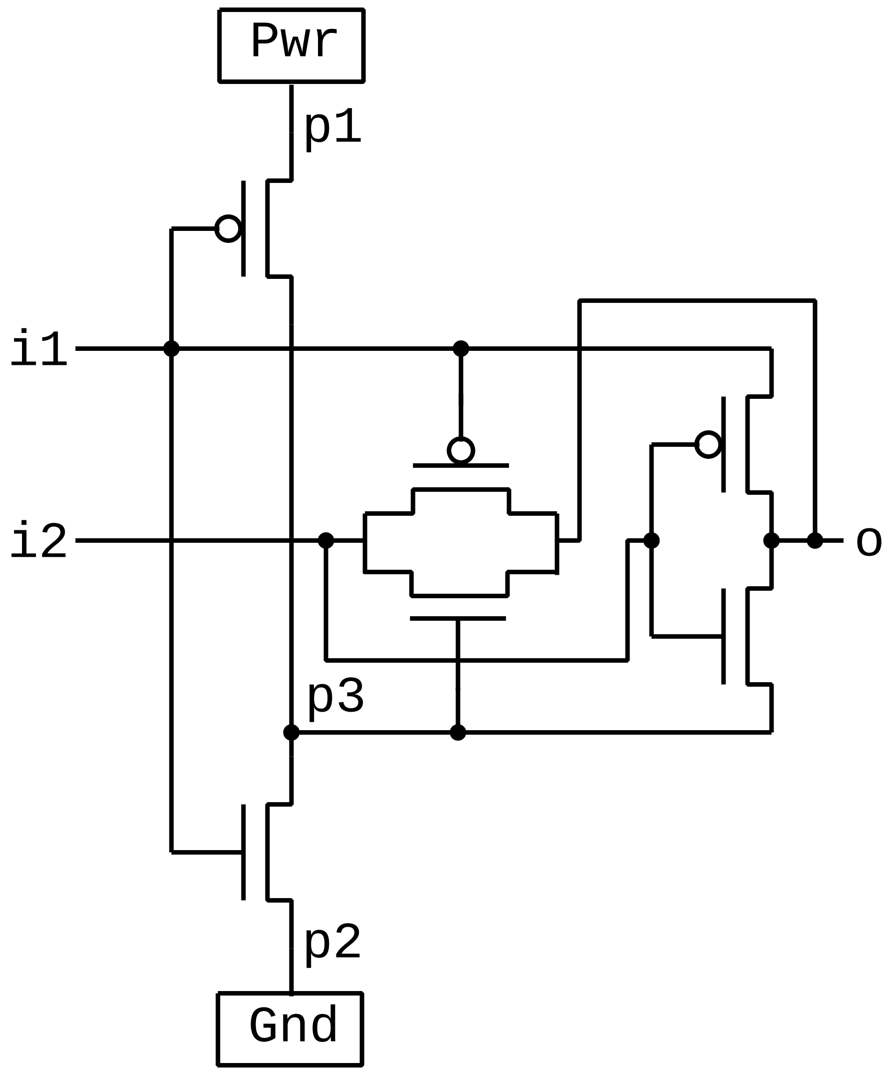
$$\texttt{Ptran(g,a,b) = (g=Lo)} \Rightarrow \texttt{((a=Hi) = (b=Hi))}$$



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

Good and bad Xors now distinguished

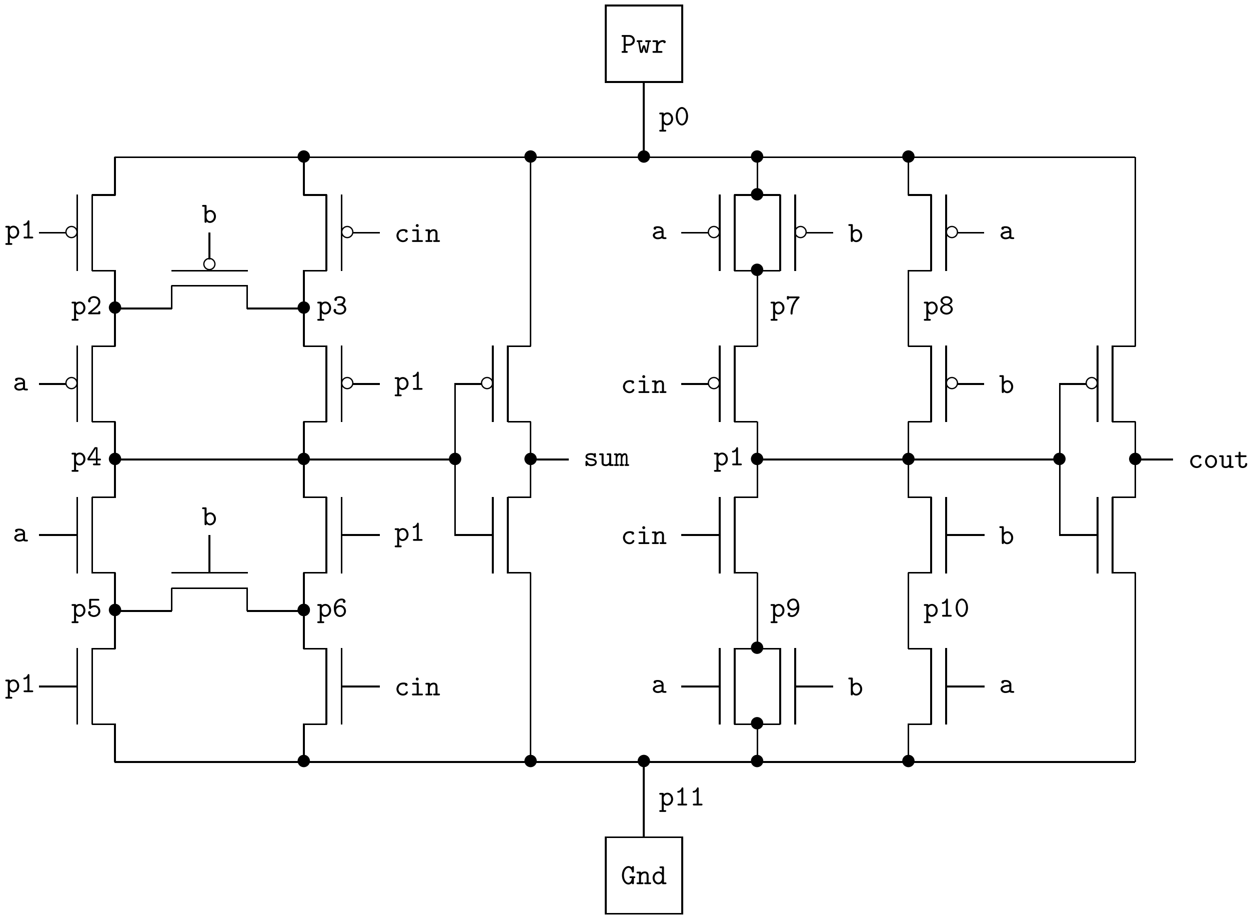




((i1=Hi) $\wedge$ (i2=Hi) $\Rightarrow$ (out = Lo)) $\wedge$
((i1=Hi) $\wedge$ (i2=Lo) $\Rightarrow$ (out = Hi)) $\wedge$
((i1=Lo) $\wedge$ (i2=Hi) $\Rightarrow$ $\neg$(out = Lo)) $\wedge$
((i1=Lo) $\wedge$ (i2=Lo) $\Rightarrow$ $\neg$(out = Hi))

((i1=Hi) $\wedge$ (i2=Hi) $\Rightarrow$ (out = Lo)) $\wedge$
((i1=Hi) $\wedge$ (i2=Lo) $\Rightarrow$ (out = Hi)) $\wedge$
((i1=Lo) $\wedge$ (i2=Hi) $\Rightarrow$ (out = Hi)) $\wedge$
((i1=Lo) $\wedge$ (i2=Lo) $\Rightarrow$ (out = Lo))

Earlier examples still work



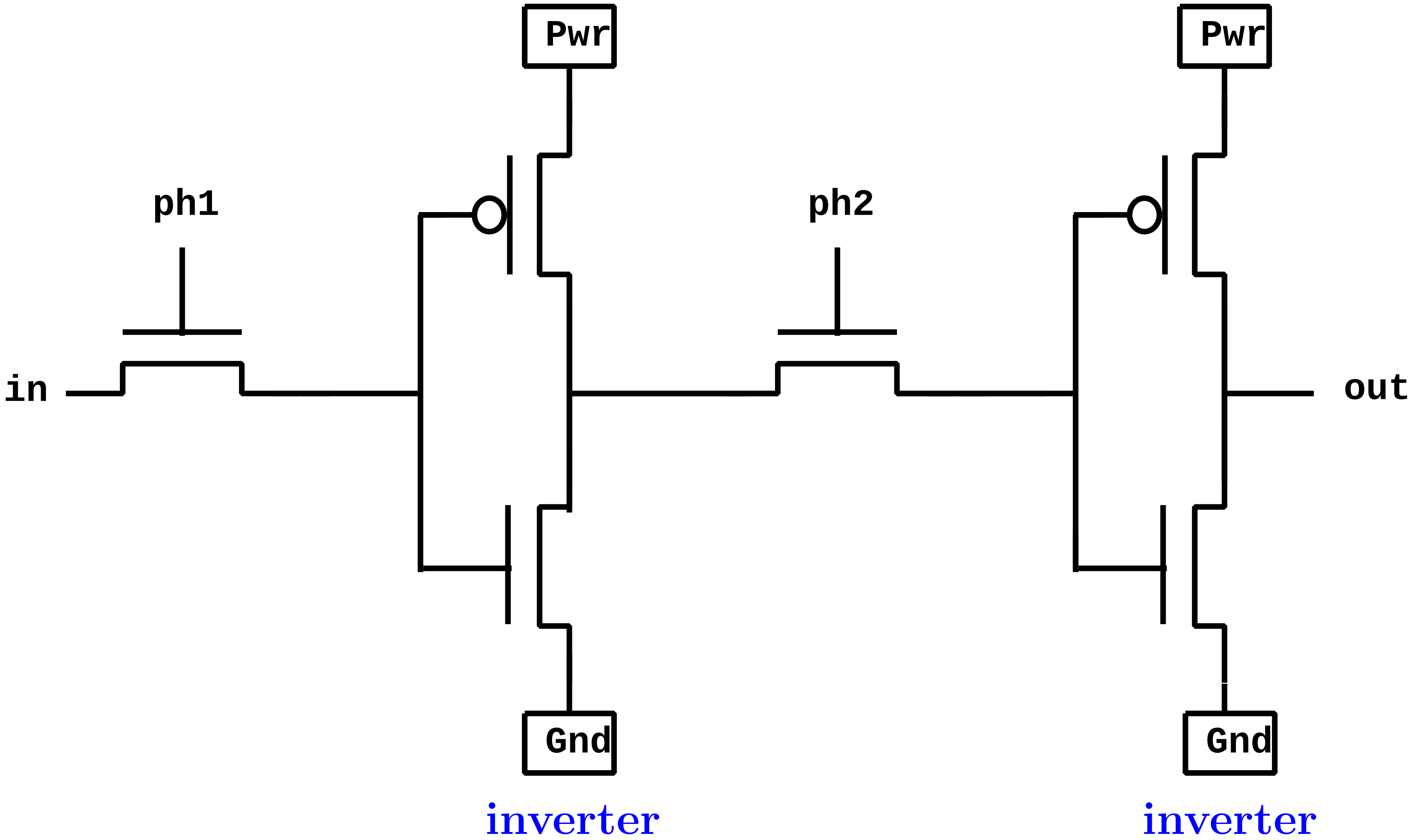
- **Define**

```
|- 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

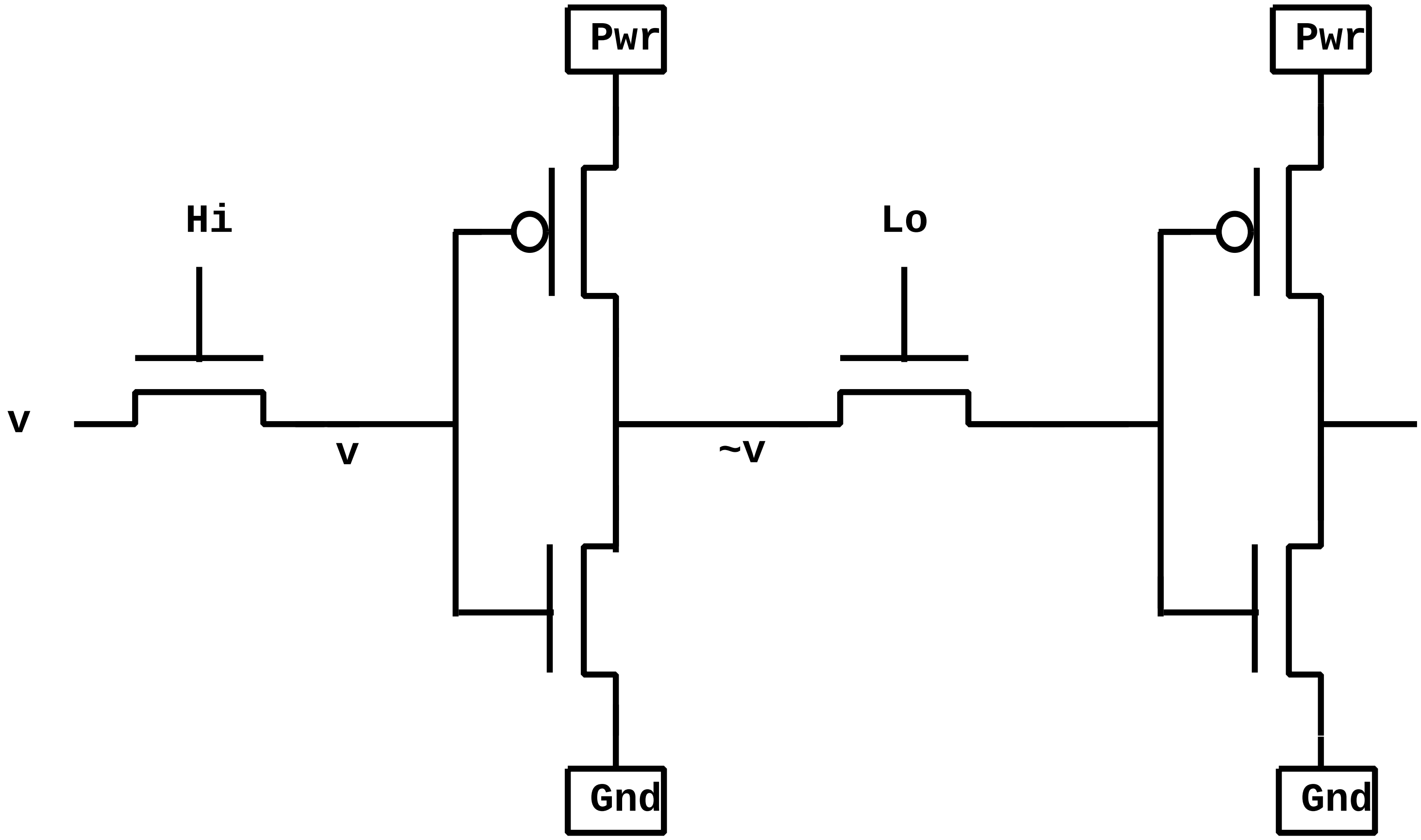


- **Switch models only allow us to deduce**

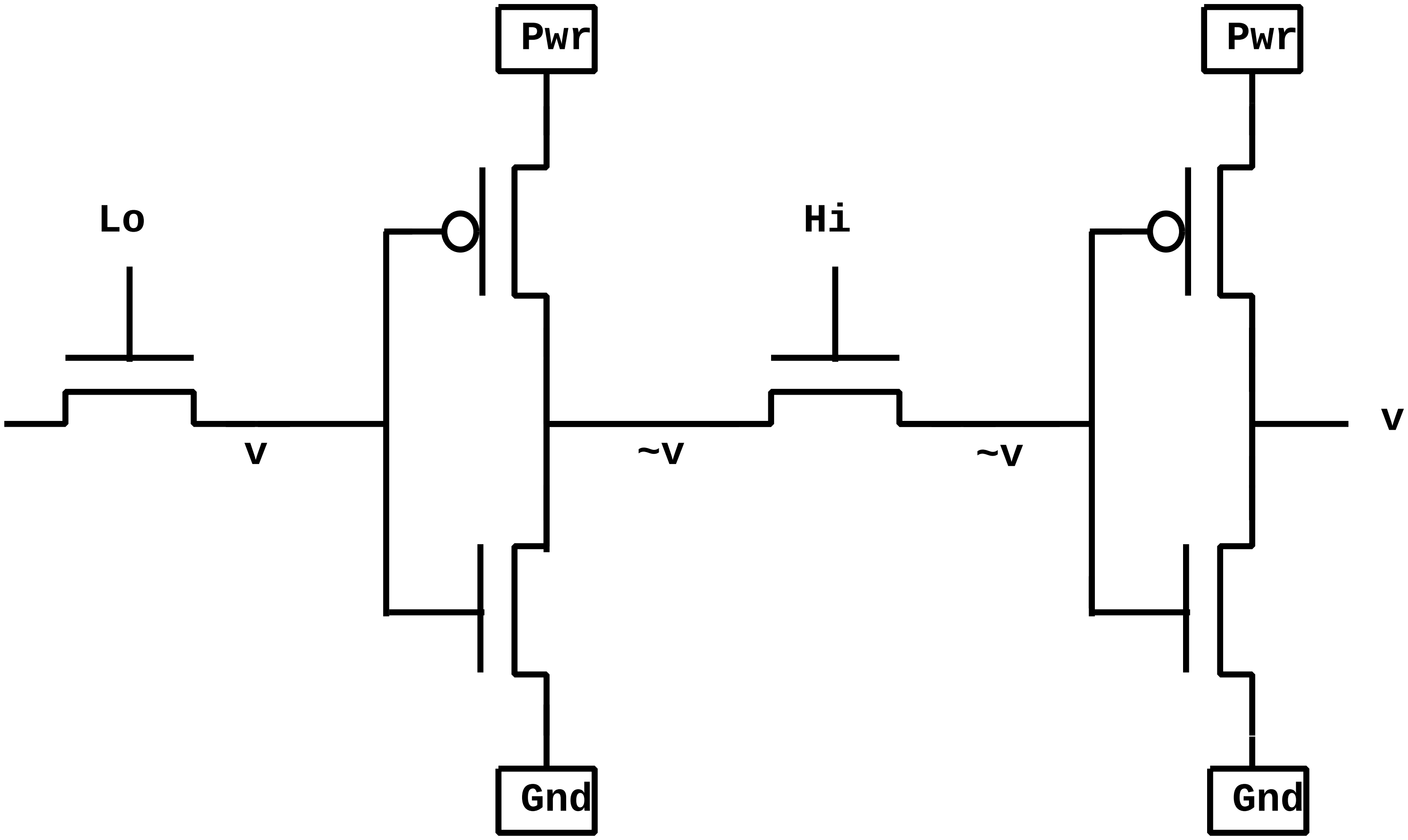
  (ph1=Hi) $\wedge$ (ph2=Hi) $\Rightarrow$ ((in=Hi) $\Rightarrow$ (out=Hi)) $\wedge$ ((in=Lo) $\Rightarrow$ (out=Lo))

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

Phase 1: `ph1=Hi` **and** `ph2=Lo`

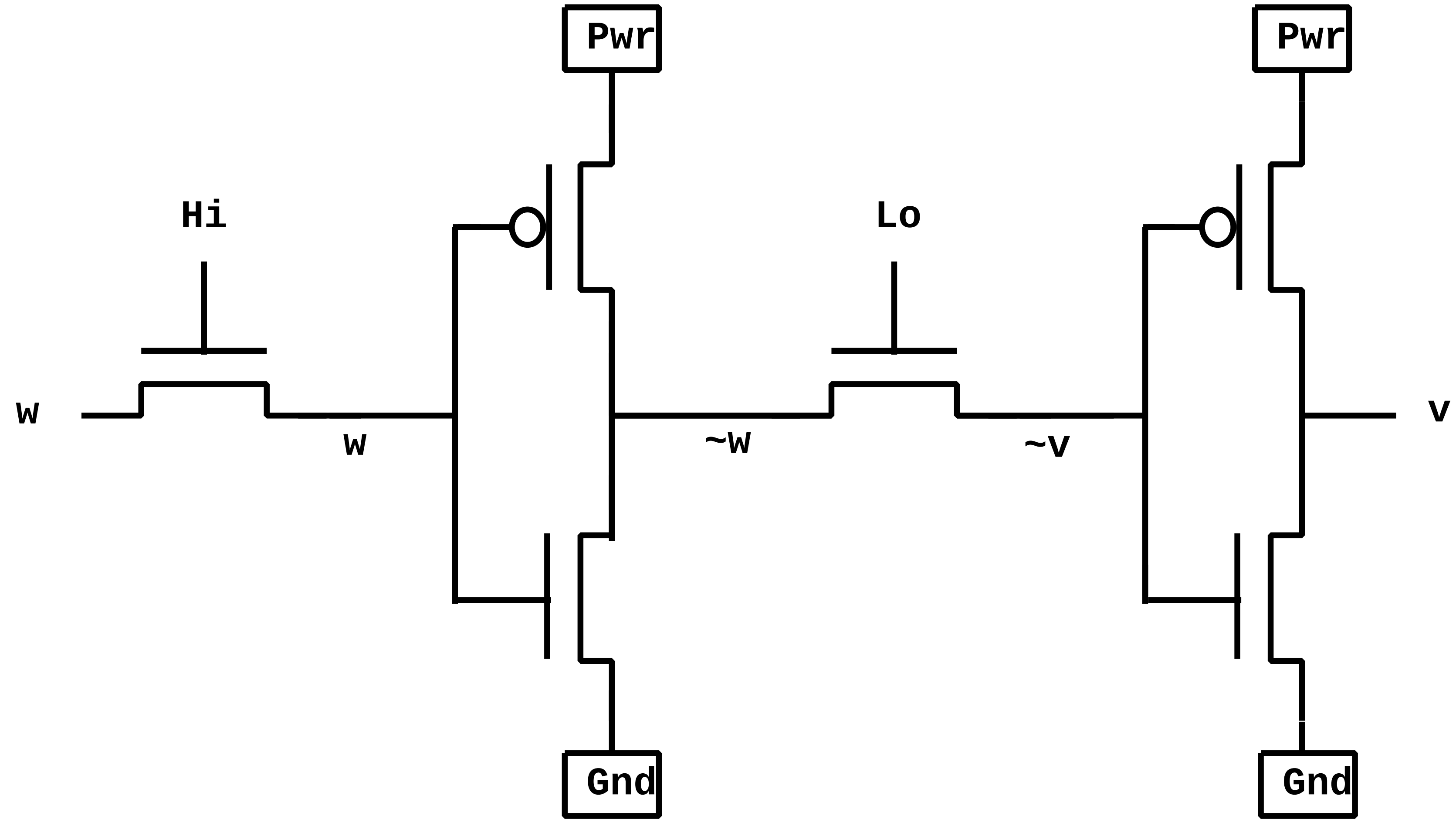


Phase 2: `ph1=Lo` **and** `ph2=Hi`



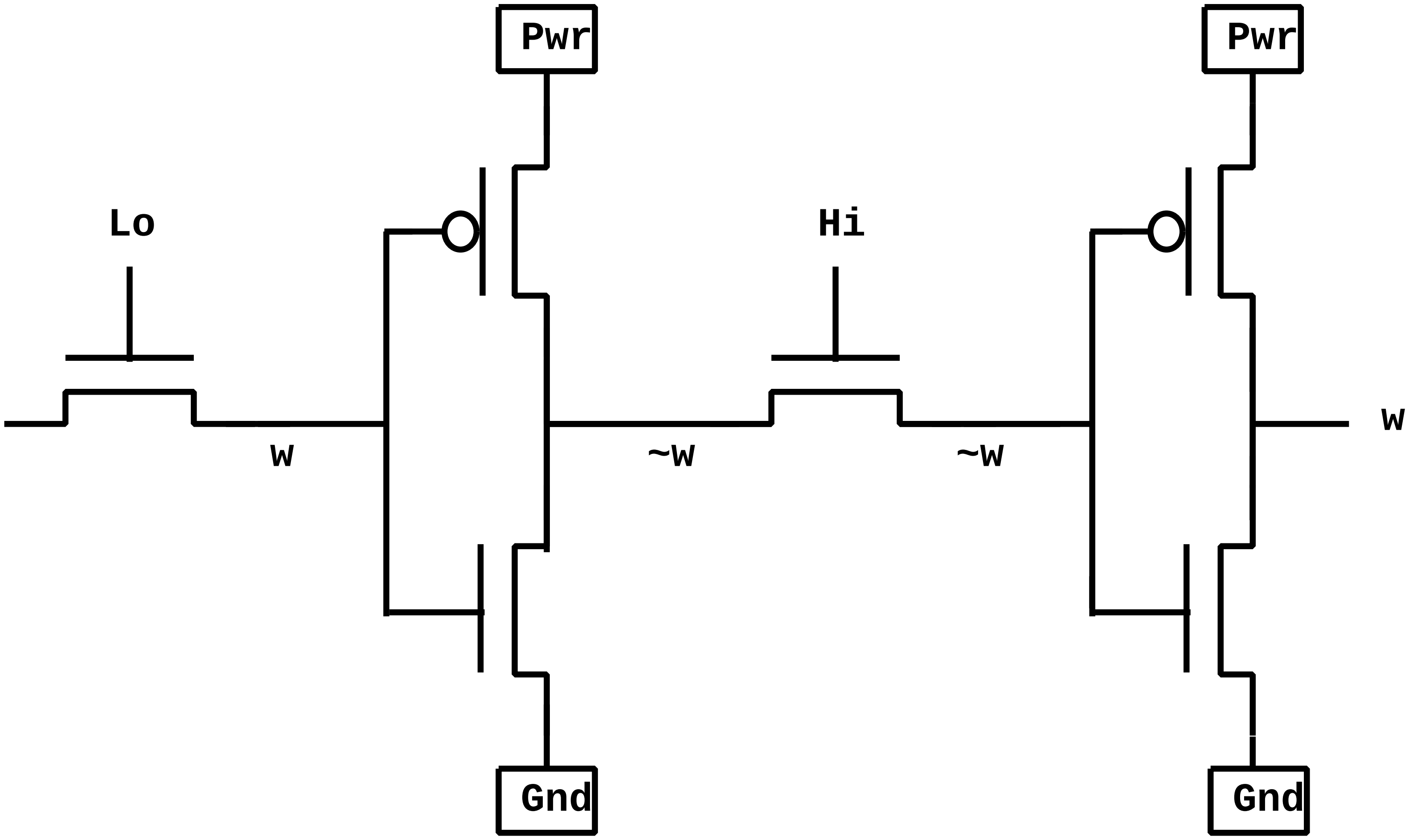

```
(ph1 t    = Hi) ∧ (ph2 t    = Lo) ∧
(ph1(t+1) = Lo) ∧ (ph2(t+1) = Hi)
⇒
(out(t+1) = in t)
```

Phase 3: ph1=Hi and ph2=Lo



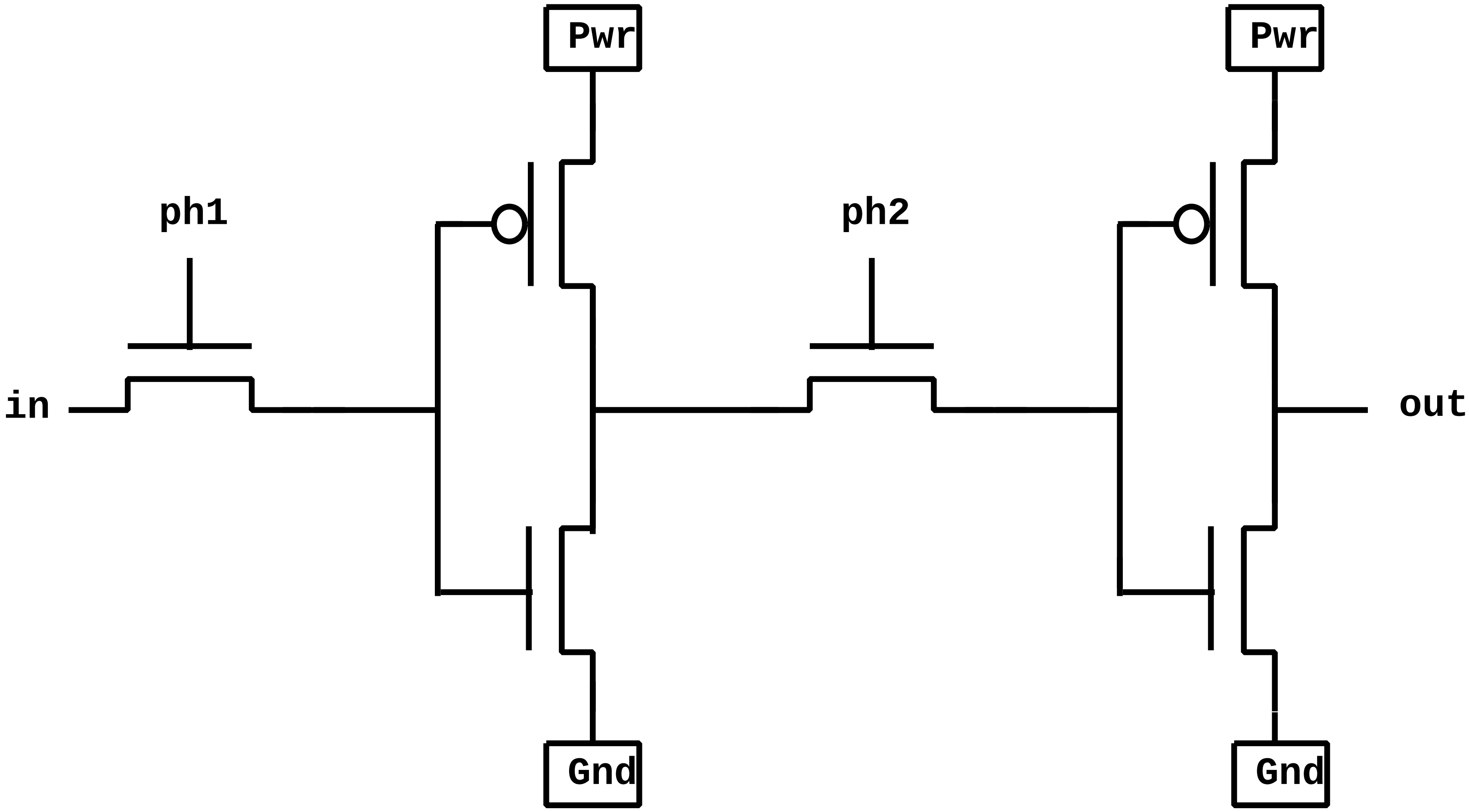

```
(ph1 t    = Hi) ∧ (ph2 t    = Lo) ∧
(ph1(t+1) = Lo) ∧ (ph2(t+1) = Hi) ∧
(ph1(t+2) = Hi) ∧ (ph2(t+2) = Lo)
⇒
(out(t+2) = in t)
```

Phase 4: `ph1=Lo` **and** `ph2=Hi`



(ph1(t+2) = Hi) $\wedge$ (ph2(t+2) = Lo) $\wedge$
(ph1(t+3) = Lo) $\wedge$ (ph2(t+3) = Hi)
$\Rightarrow$
(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` $\mapsto$ `t+2` **in first property**

Unidirectional sequential model

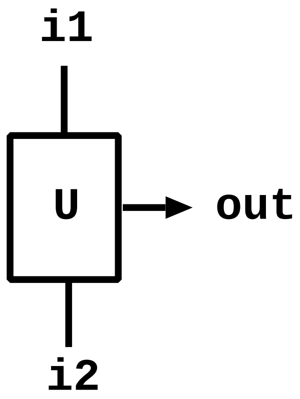
- **Four values:** `Hi`, `Lo`, `Fl` (**'floating'**), `X` (**unknown/error**)

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

|- Strong v = ((v = Hi) ∨ (v = Lo))

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

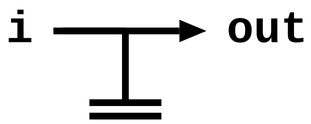
- **Join operator:** `U`

```
|- 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

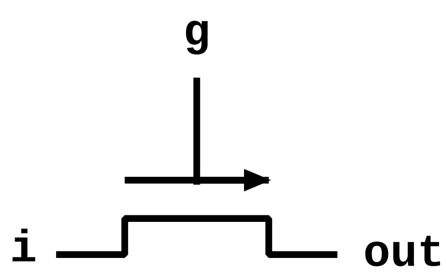



```
|- Join(i1,i2,out) = ∀t. out t = (i1 t) U (i2 t)

|- Pwr out = ∀t. out t = Hi

|- Gnd out = ∀t. out t = Lo
```

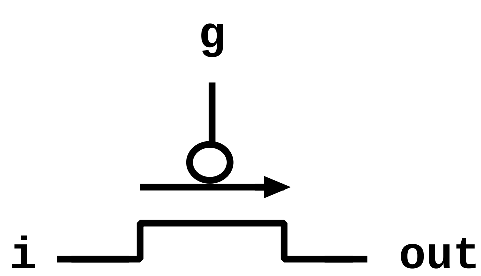



```
|- Cap(i,out) = ∀t. out t = if Strong(i t) then i t else
                            if t=0 then X else
                            if Float(i t) ∧ Strong(i(t-1)) then i(t-1)
                            else Fl
```

Unidirectional sequential transistor models

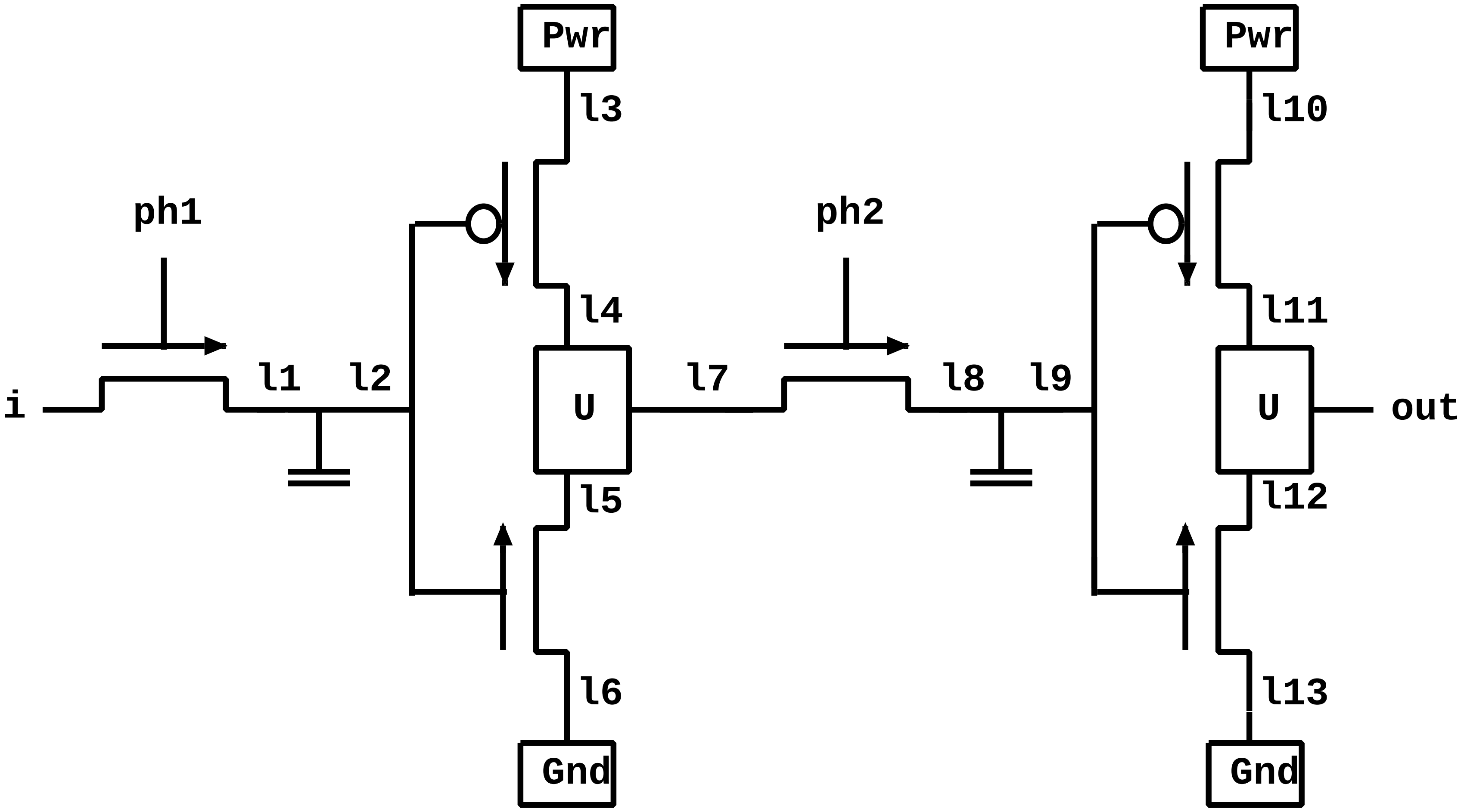



```
|- Nswitch(g,i,out) = ∀t. out t = if g t = Hi then i t else
                                  if (g t = Lo ) ∨ (i t = Fl) then Fl
                                  else X
```




```
|- 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
```

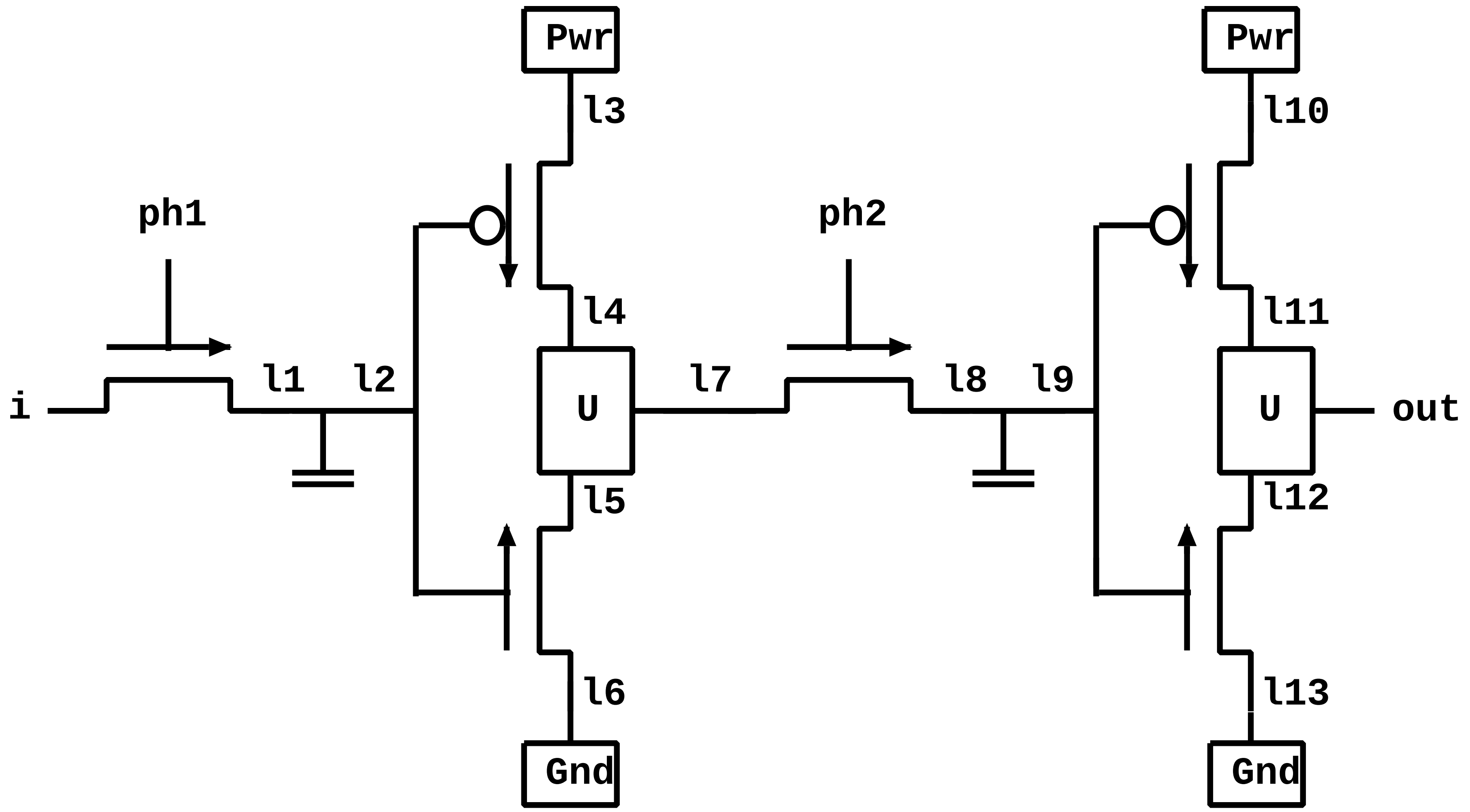
Sequential shift register model




```
|- 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!**

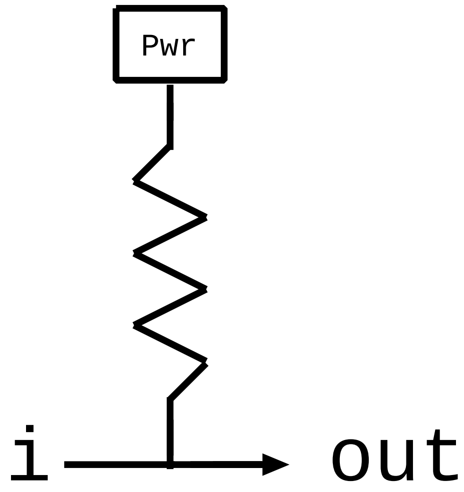
Correctness of sequential shift register model




```
|- 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

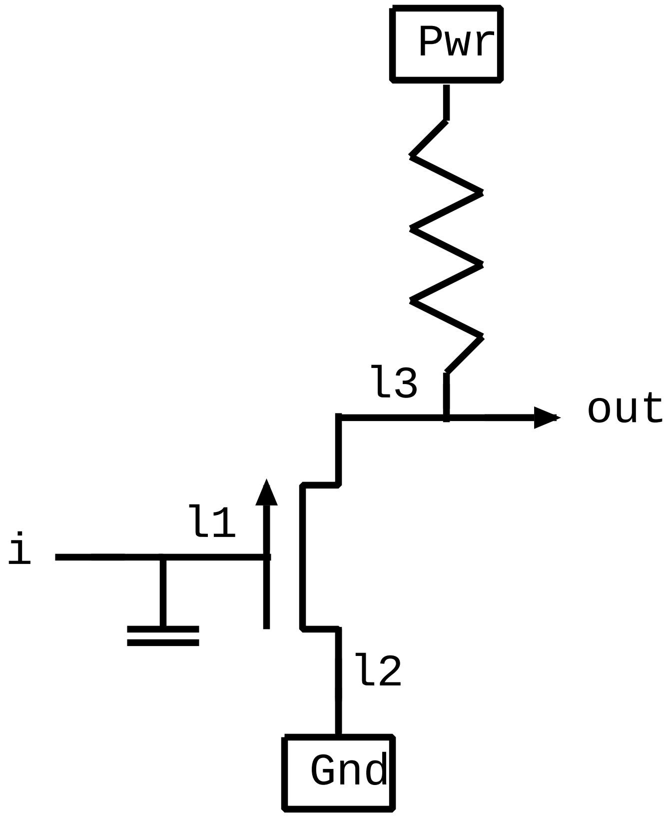
- **Need a new component: pullup**



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

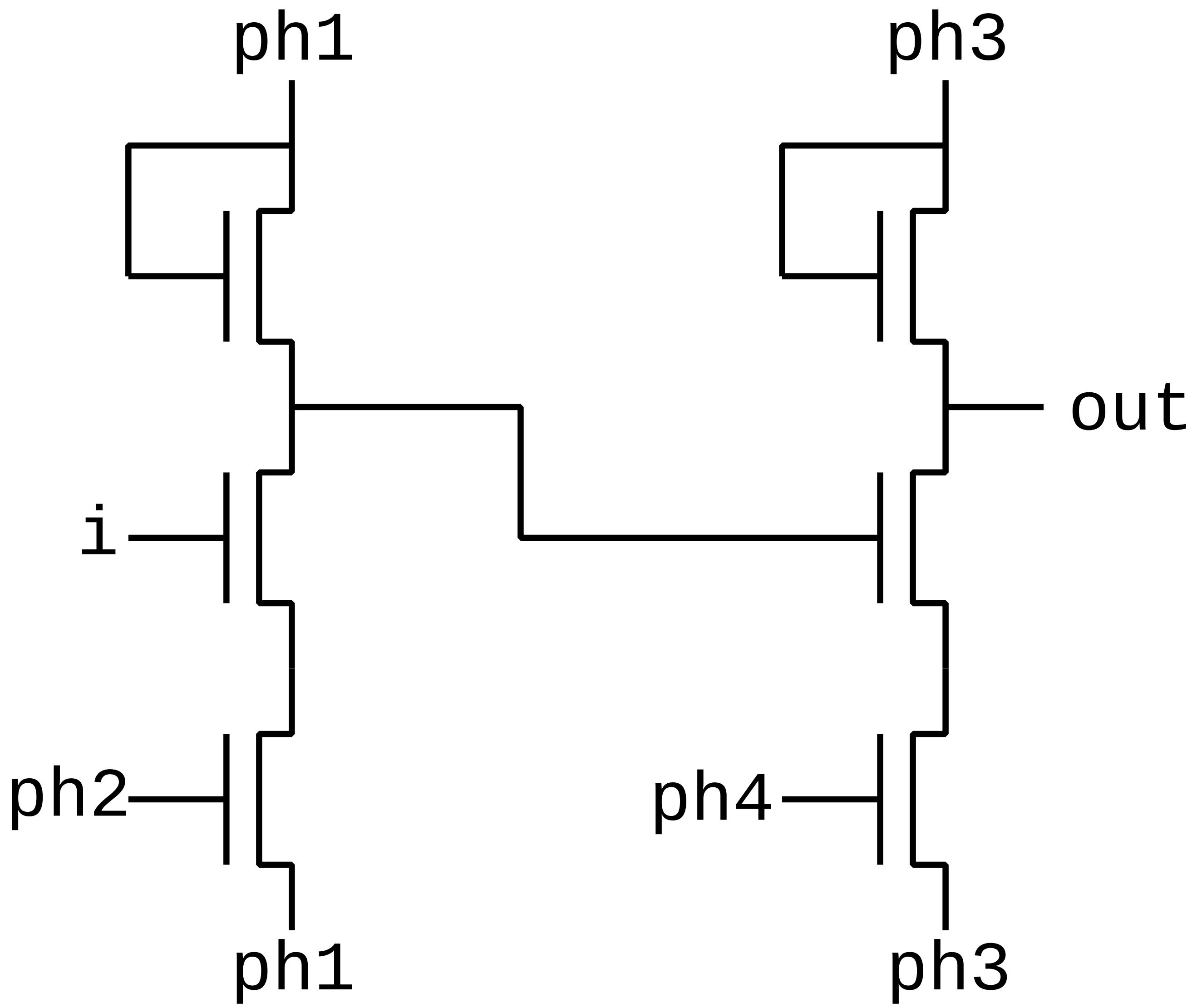
- **If** `i` **is strong then** `out = i`
- **If** `i` **is floating then** `out = Hi`

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))))
```

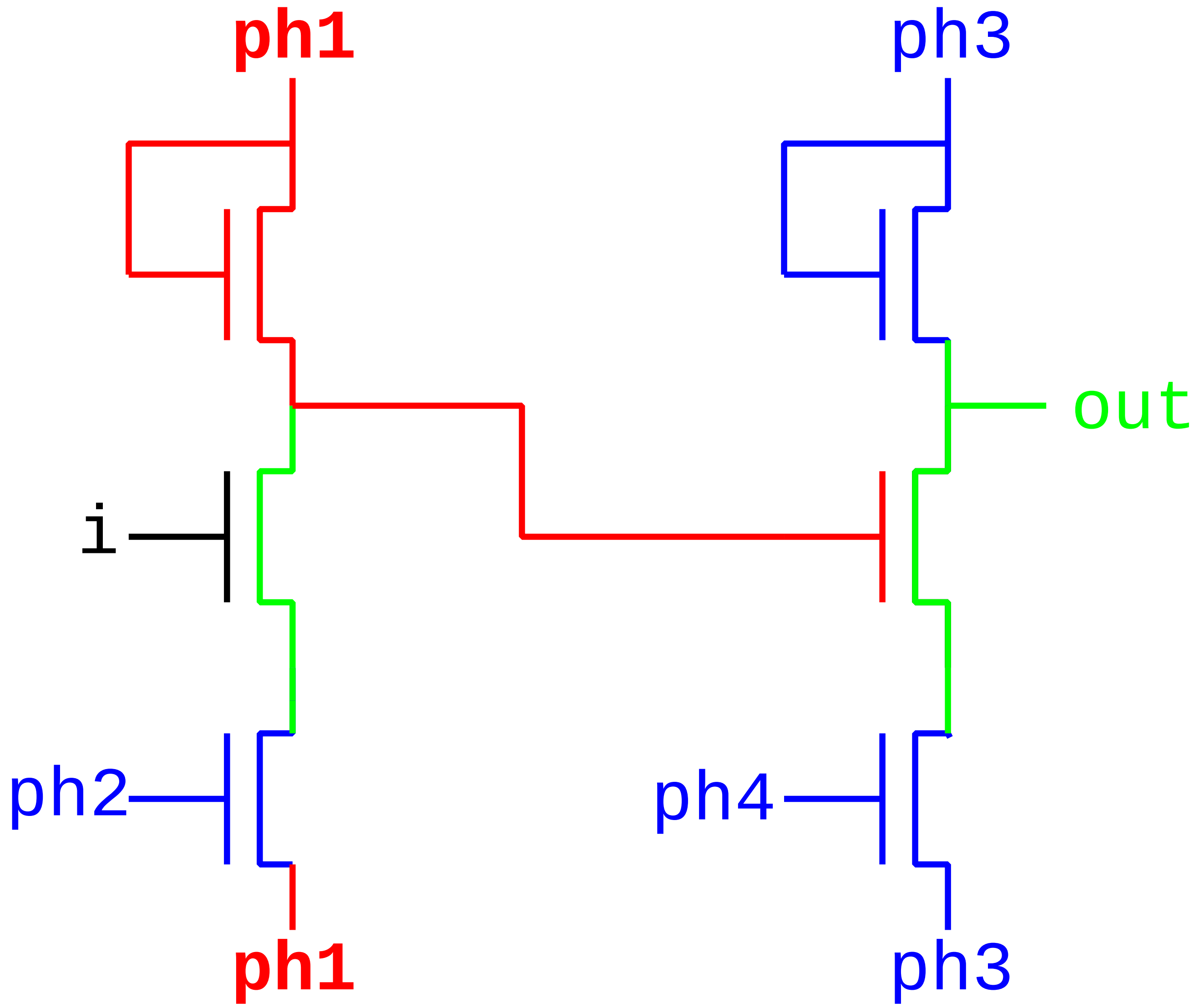
Four phase NMOS shift register




```
|- 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)

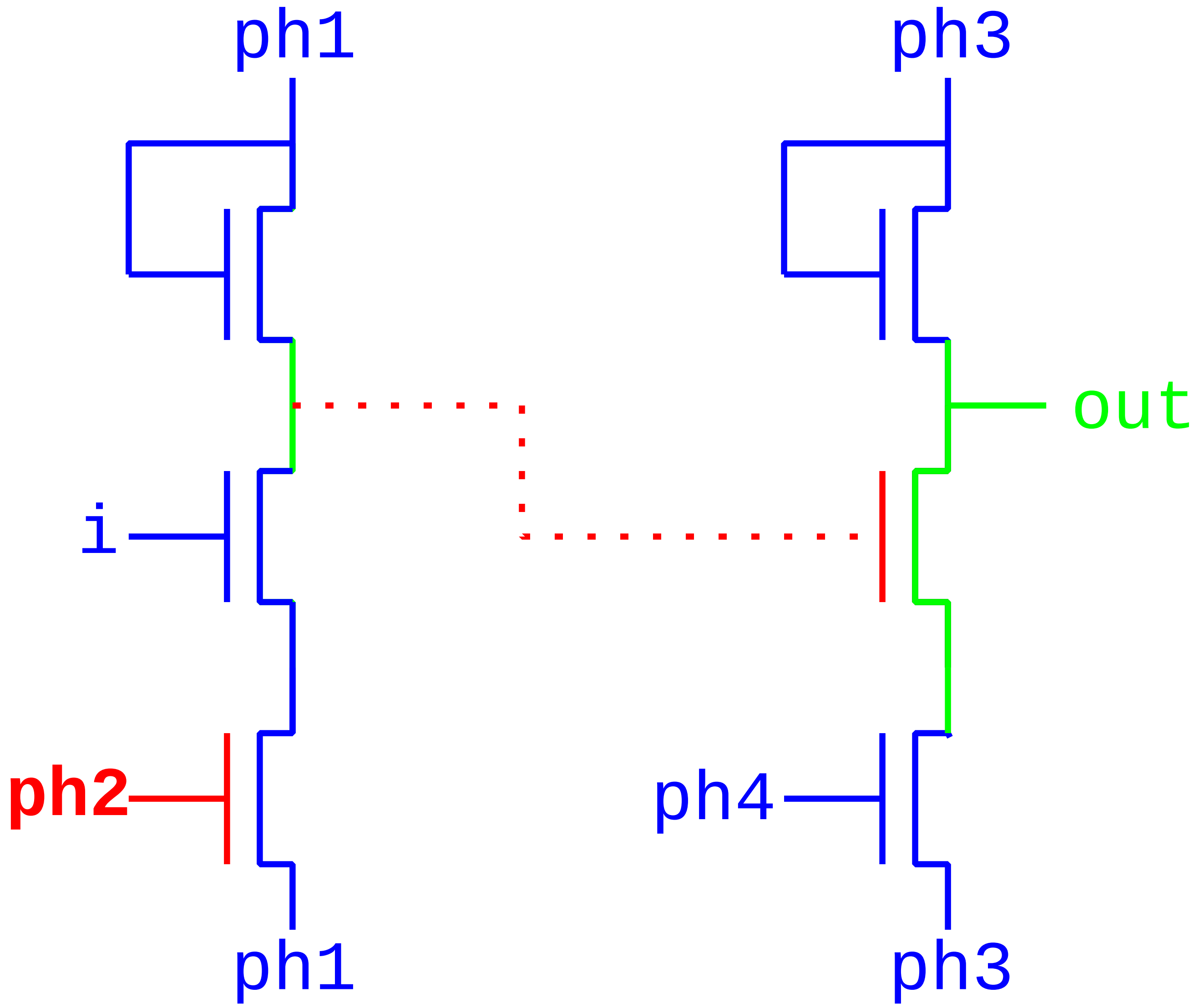
Colour scheme: Hi, Lo, Fl; threshold effects ignored



`(ph1 t = Hi)` $\wedge$ `(ph2 t = Lo)` $\wedge$ `(ph3 t = Lo)` $\wedge$ `(ph4 t = Lo)`

Phase 2 (input `Lo`, retain precharge)

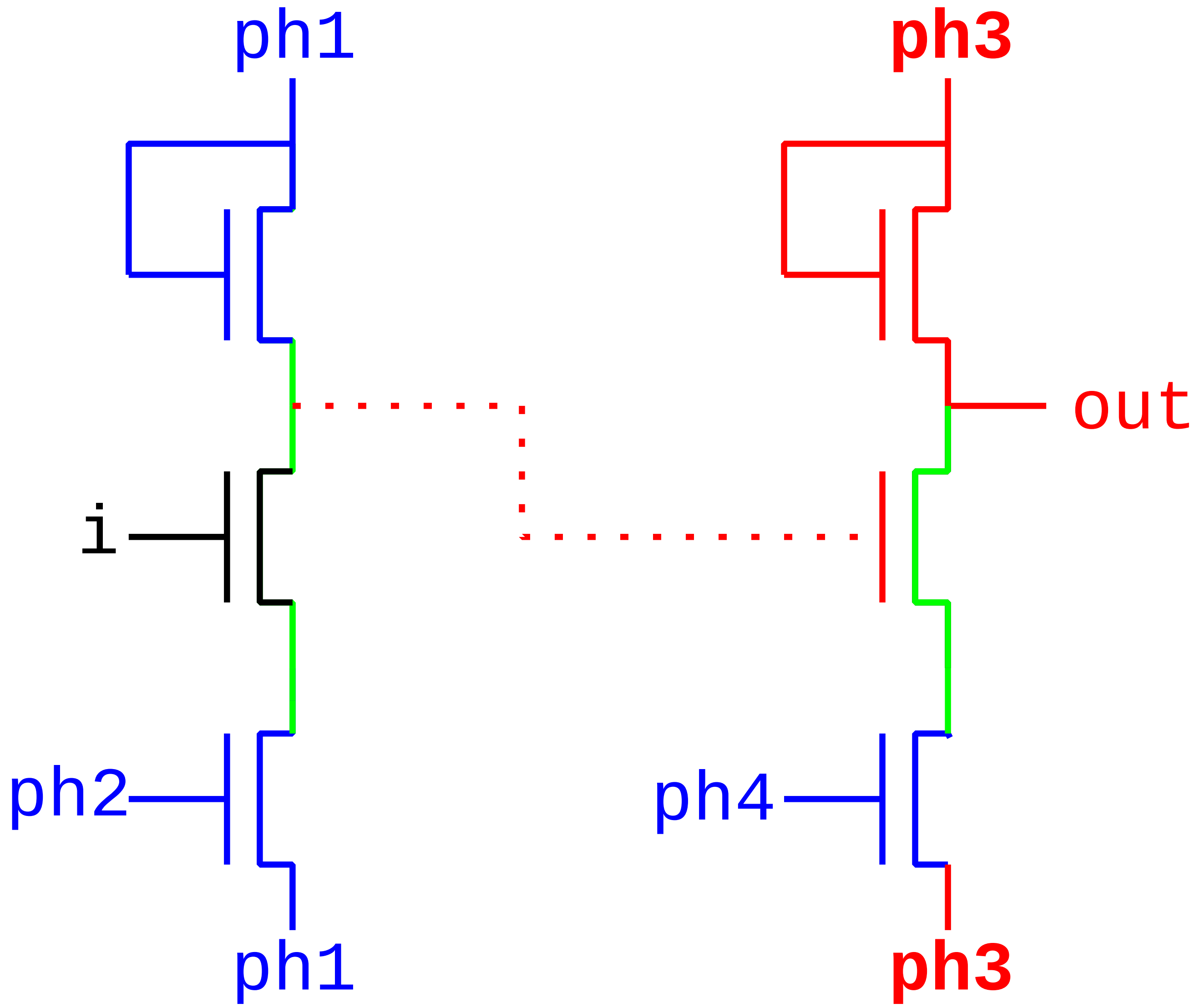
Colour scheme: Hi, Lo, Fl and dotted means precharge



`(ph1(t+1)=Lo)` $\wedge$ `(ph2(t+1)=Hi)` $\wedge$ `(ph3(t+1)=Lo)` $\wedge$ `(ph4(t+1)=Lo)`

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

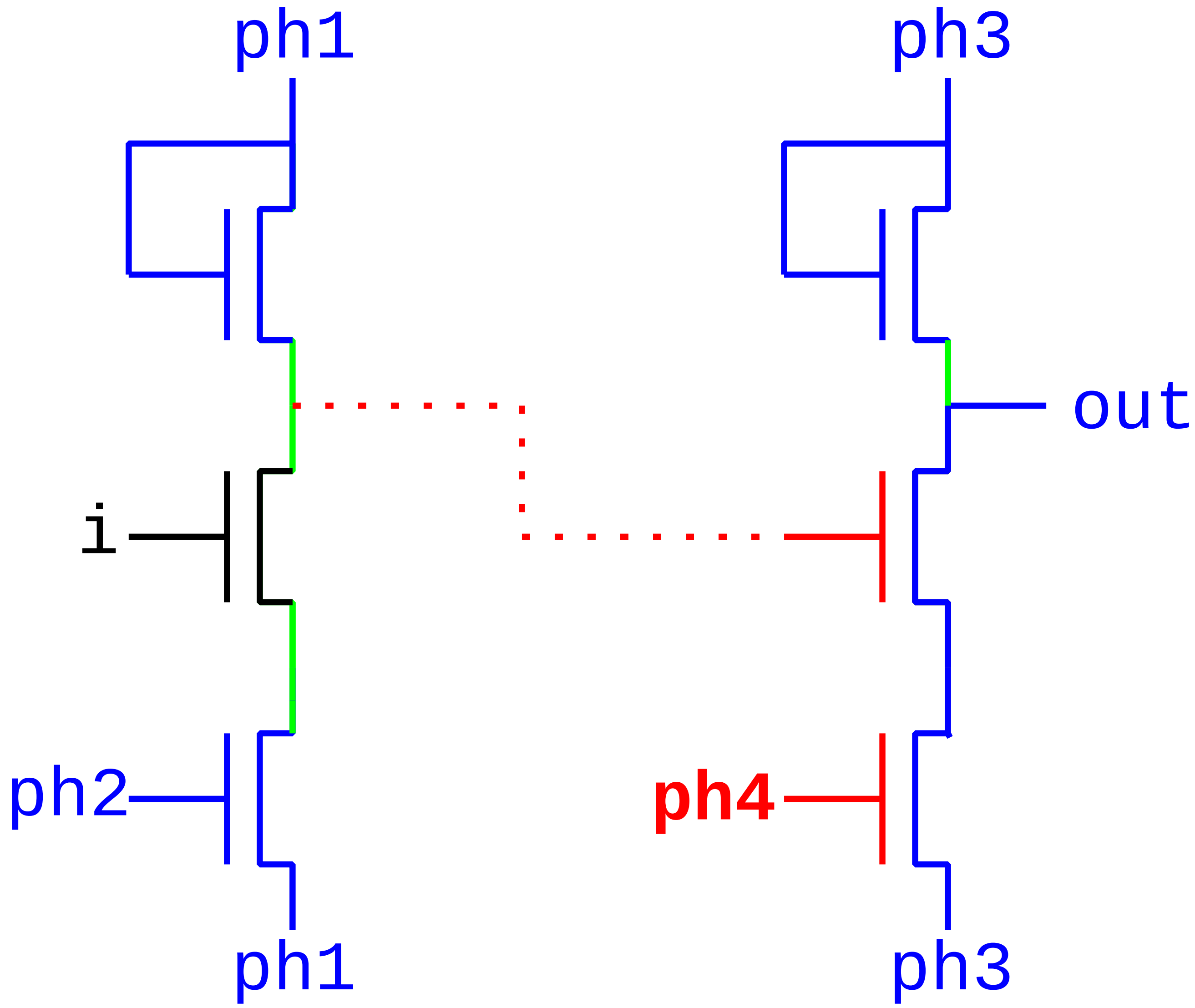
Phase 3 (precharge `out`, internal node retains value)



`(ph1(t+2)=Lo)` $\wedge$ `(ph2(t+2)=Lo)` $\wedge$ `(ph3(t+2)=Hi)` $\wedge$ `(ph4(t+2)=Lo)`

`(out(t+2) = Hi)`

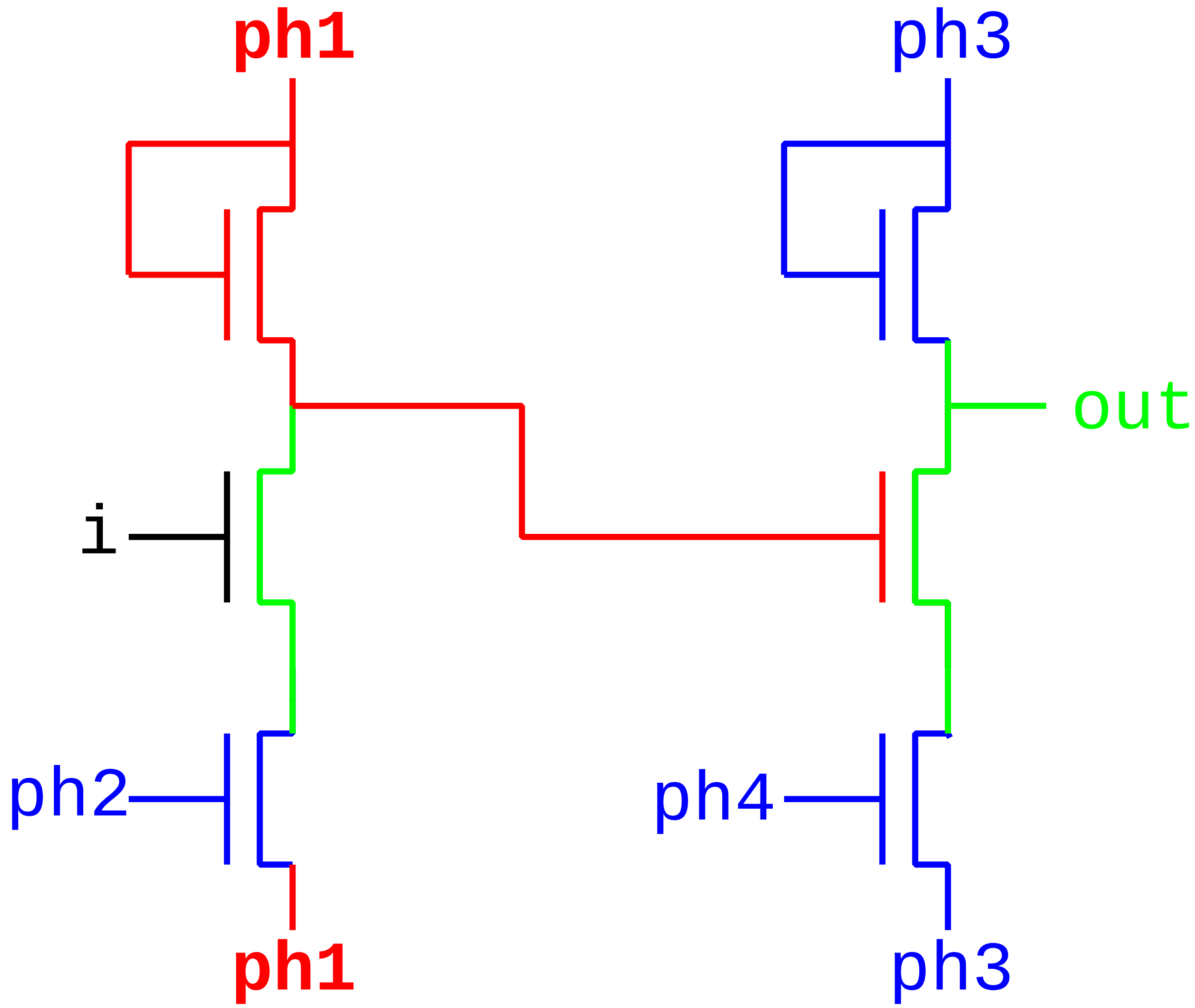
Phase 4 (kill precharge)



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

(out(t+3) = Lo)

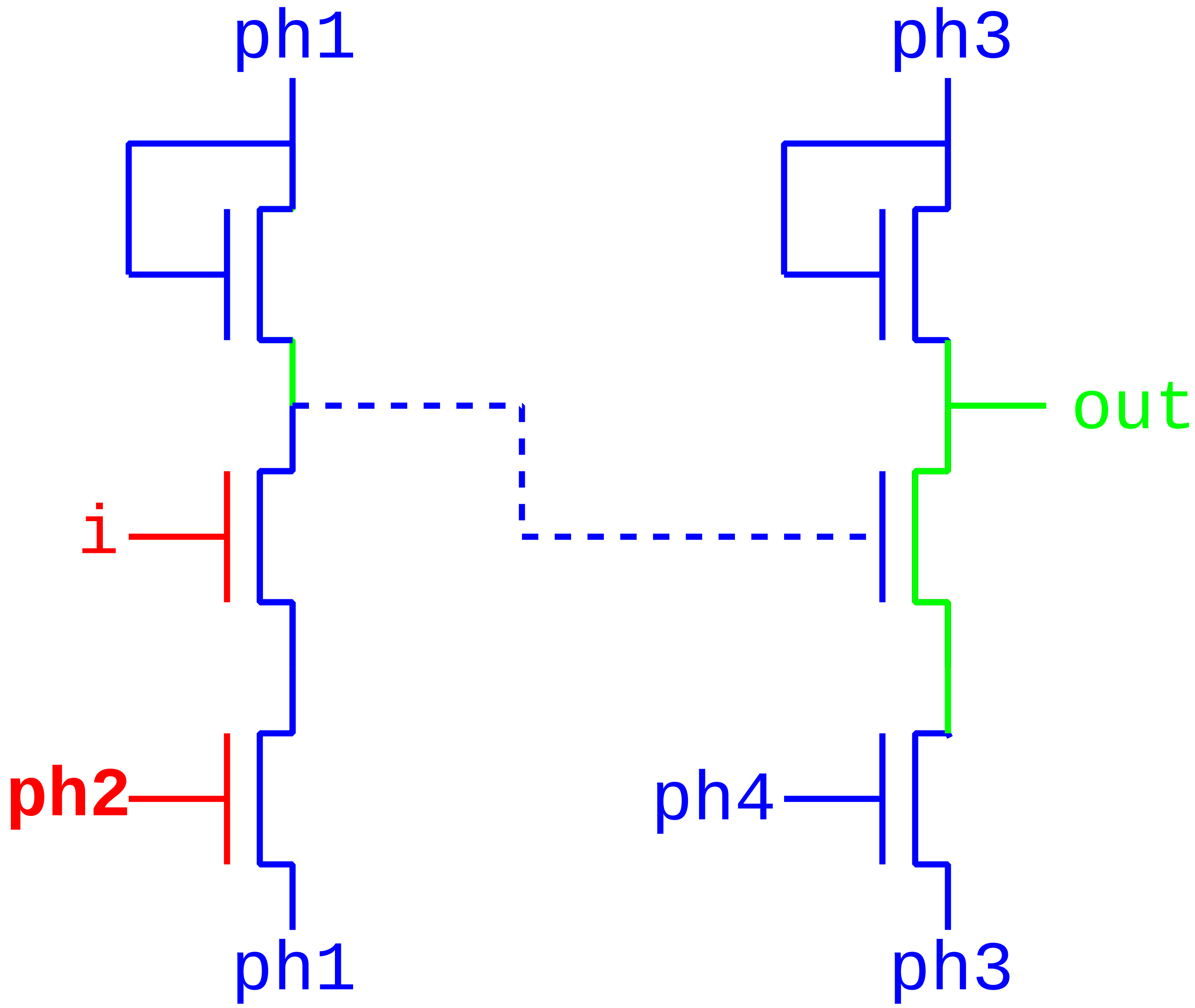
Phase 1 (precharge internal node)



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

- **out retains previous value**

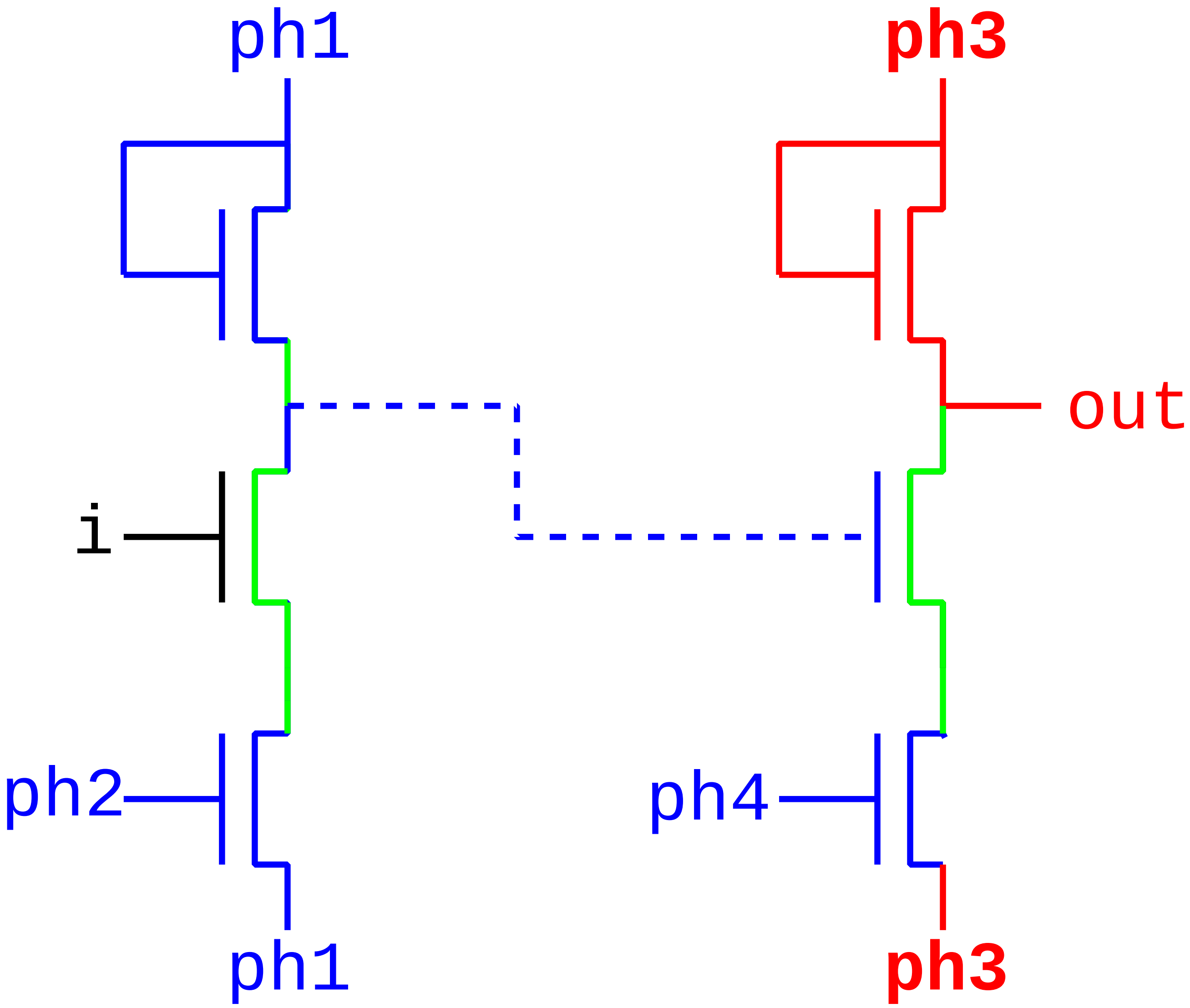
Phase 2 (input Hi, kill precharge)



(ph1(t+1)=Lo) $\wedge$ (ph2(t+1)=Hi) $\wedge$ (ph3(t+1)=Lo) $\wedge$ (ph4(t+1)=Lo)

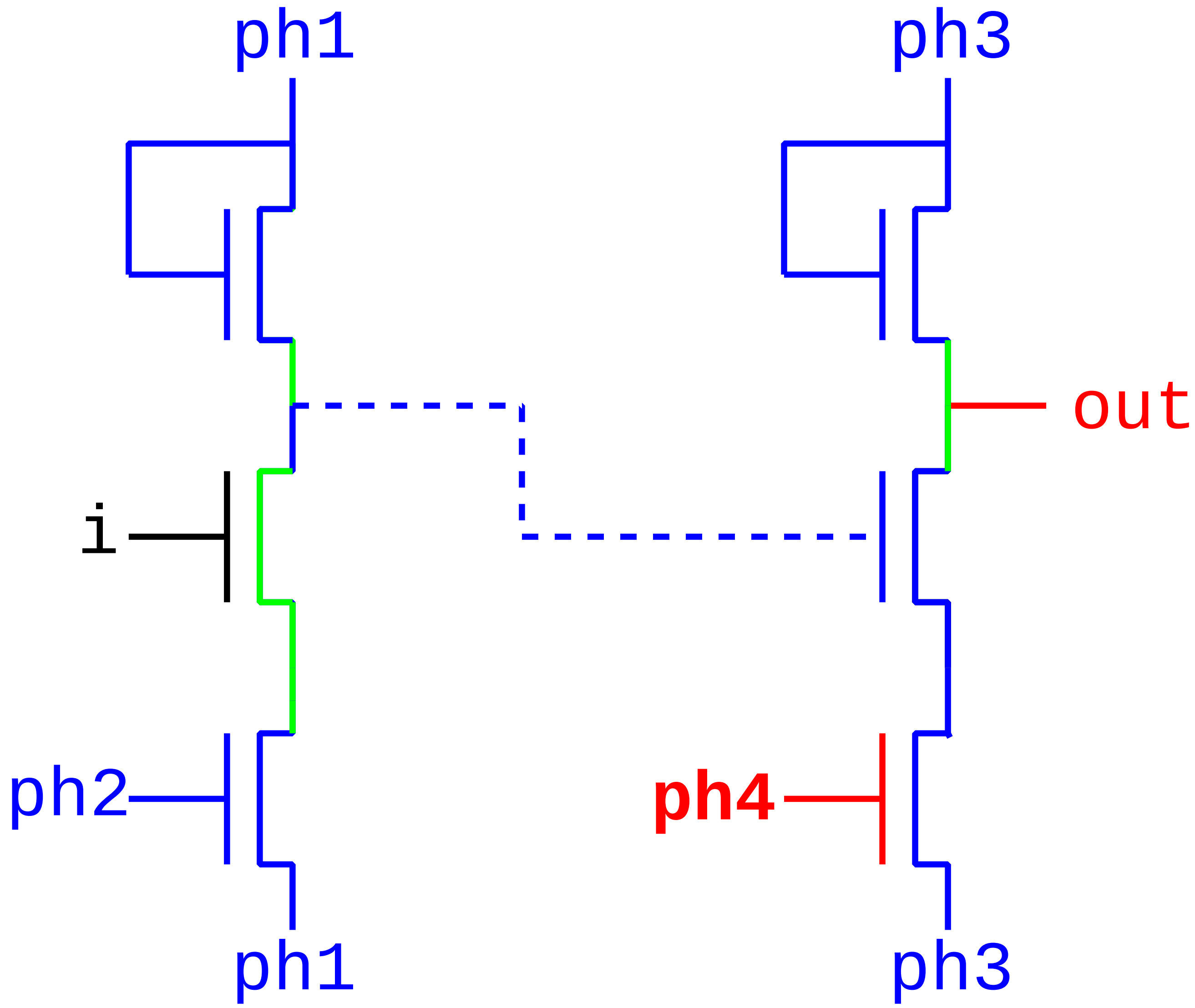
(i(t+1) = Hi)

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



`(ph1(t+2)=Lo)` $\wedge$ `(ph2(t+2)=Lo)` $\wedge$ `(ph3(t+2)=Hi)` $\wedge$ `(ph4(t+2)=Lo)`

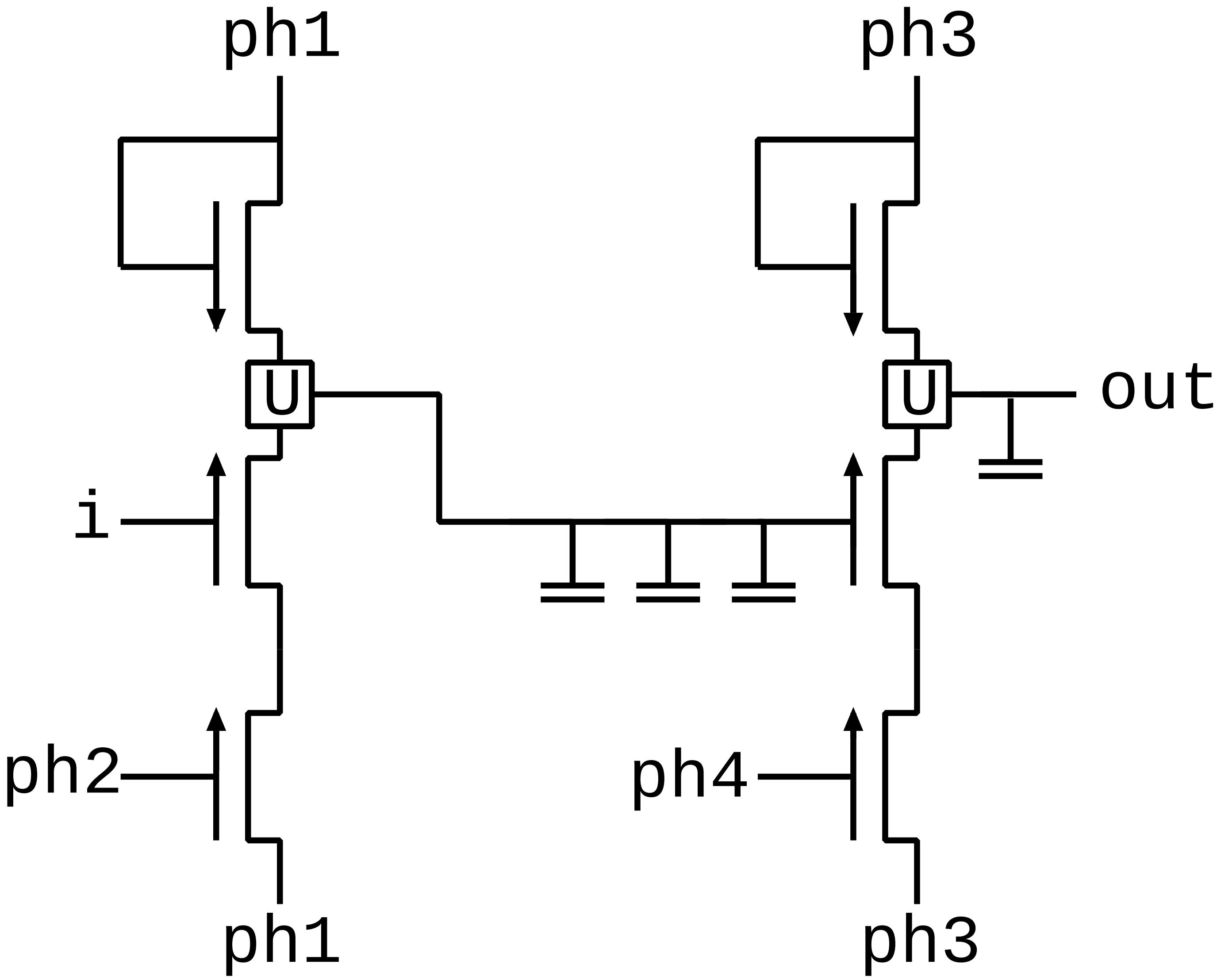
Phase 4 (out retains precharge)



(ph1(t+3)=Lo) $\wedge$ (ph2(t+3)=Lo) $\wedge$ (ph3(t+3)=Lo) $\wedge$ (ph4(t+3)=Hi)

out(t+3) = Hi

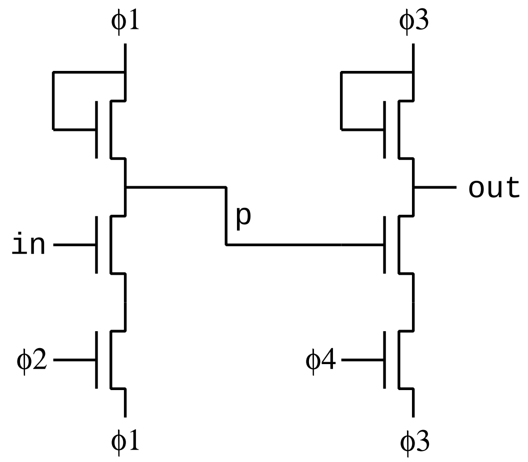
Four phase NMOS shift register model




```
|- 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:**

$\vdash$ {*If environment ensures always that:* `DONE`=`0` $\Rightarrow$ `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**

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 $\Rightarrow$ bugs	proof success $\Rightarrow$ 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**

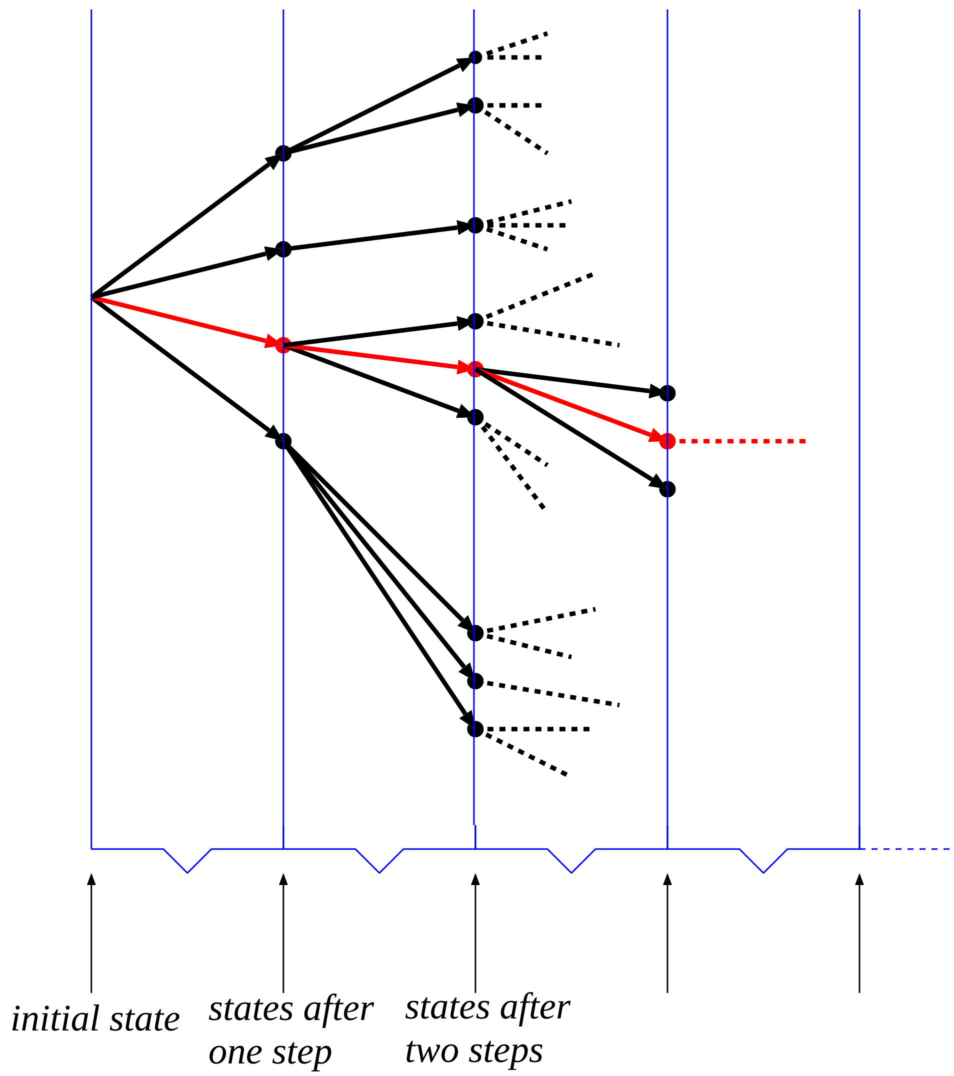
NEW TOPIC: Model Checking

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

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



Example: single state machine

- **State transition function:** δ

  $\delta \;:\; \mathit{states} \times \mathit{inputs} {\rightarrow} \mathit{states}$

- **Define state transition relation:**

  $\mathcal{R}(s, s') \;=\; \exists \mathit{inp}.\; s' \;=\; \delta(s, \mathit{inp})$

- **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**

  $\delta_i \;:\; states \times inputs \rightarrow states_i \qquad\qquad (1 \leq i \leq n)$
- **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** $v = b_1 \lor v = b_2 \lor \ldots \lor v = b_n$
  - $b_1, b_2, \ldots, b_n$ **are truth-values (i.e.** $\mathtt{T}$ **or** $\mathtt{F}$**)**
  - v **is a boolean variable**
  - $b \in \{b_1, b_2, \ldots, b_n\}$ **if and only if** $\vdash\ (v = b_1 \lor v = b_2 \lor \ldots \lor v = b_n)\mathtt{[}b/v\mathtt{]}$

- **A set of states**

  $\{(b_{11}, \ldots, b_{1m}), \ldots, (b_{n1}, \ldots, b_{nm})\}$

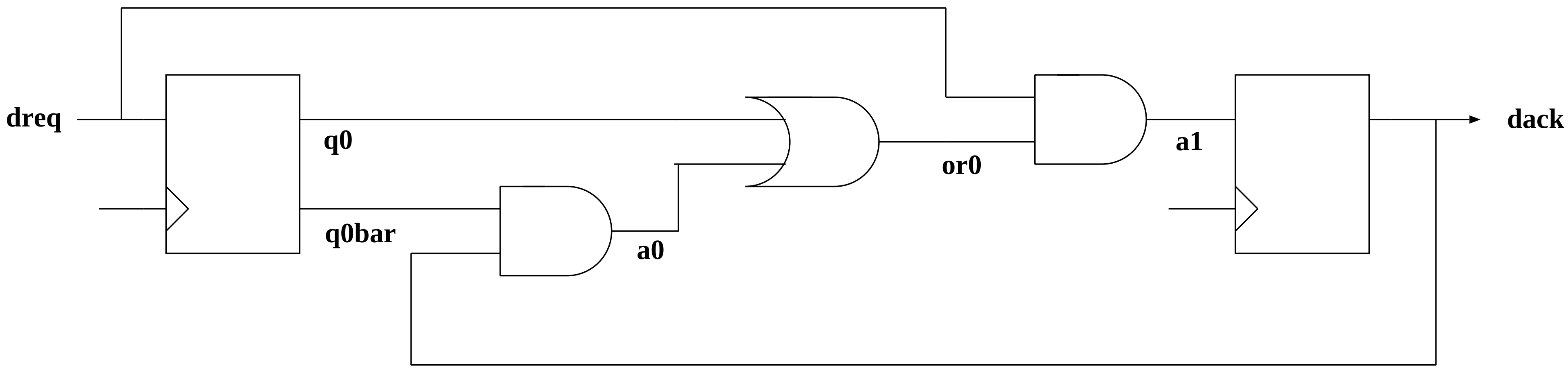
  is represented by a formula with m **boolean variables:**

  $(v_1 = b_{11} \land \ldots \land v_m = b_{1m}) \lor \ldots \lor (v_1 = b_{n1} \land \ldots \land v_m = b_{nm})$

- **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:**
  $((v_1 = b_{11} \land \ldots \land v_m = b_{1m}) \lor \ldots \lor (v_1 = b_{n1} \land \ldots \land v_m = b_{nm}))\mathtt{[}(b_1, \ldots, b_m)/(v_1, \ldots, v_m)\mathtt{]}$

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)` $\wedge$ `(dack' = dreq` $\wedge$ `(q0` $\vee$ `(`$\neg$`q0` $\wedge$ `dack)))`

- **Transition relation equivalent to:**

  `(q0' = dreq)` $\wedge$ `(dack' = dreq` $\wedge$ `(q0` $\vee$ `dack))`

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

  $\mathcal{R}_{\texttt{RECEIVER}}$`((dreq,q0,dack),(dreq',q0',dack')) =`
  `(q0'` $\Leftrightarrow$ `dreq)` $\wedge$ `(dack'` $\Leftrightarrow$ `dreq` $\wedge$ `(q0` $\vee$ `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 $\texttt{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}$ to be set of states reachable in at most n steps:**

  $\vdash\ \ \texttt{ReachBy}\ 0\ \mathcal{R}\ \mathcal{B}\ s\ =\ \mathcal{B}\ s$

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

- **Reachable states are states reachable in a finite number of steps:**

  $\vdash\ \ \texttt{Reach}\ \mathcal{R}\ \mathcal{B}\ s\ =\ \exists n.\ \texttt{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s$

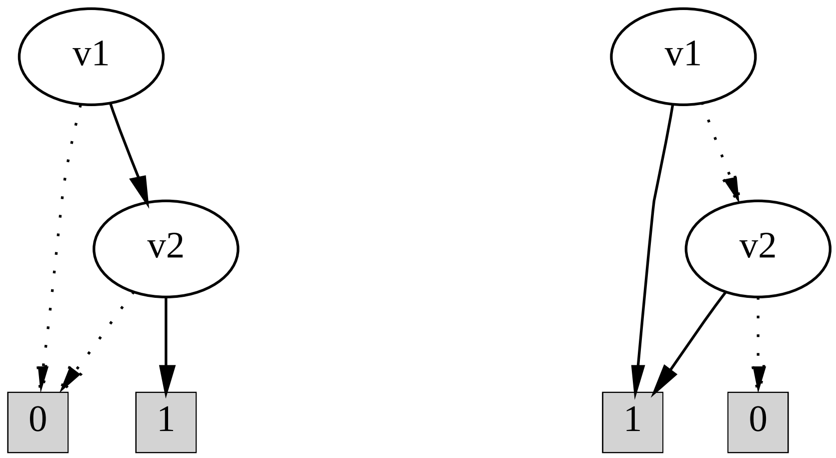
- **Key property (equality between predicates represents set equality):**

  $$\begin{array}{ll}\vdash & (\texttt{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ =\ \texttt{ReachBy}\ (n{+}1)\ \mathcal{R}\ \mathcal{B}) \\ & \Rightarrow \\ & (\texttt{Reach}\ \mathcal{R}\ \mathcal{B}\ =\ \texttt{ReachBy}\ n\ \mathcal{R}\ \mathcal{B})\end{array}$$

Represent formulas as Binary Decision Diagrams

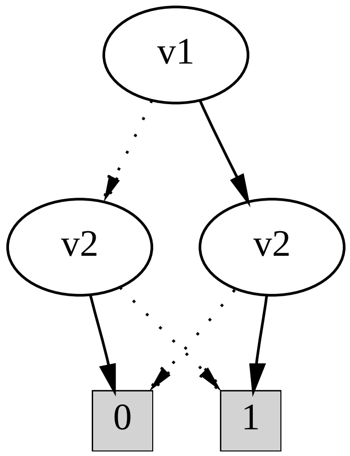
- Reduced Ordered Binary Decision Diagrams (ROBDDs or BDDs for short) are a data-structure for representing Boolean formulas
- Key features:
  - canonical (given a variable ordering)
  - efficient to manipulate
- Variables: `v = if v then 1 else 0` and `¬v = if v then 0 else 1`
- Example: BDDs of variable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

- Example: BDDs of `v1` $\wedge$ `v2` and `v1` $\vee$ `v2`

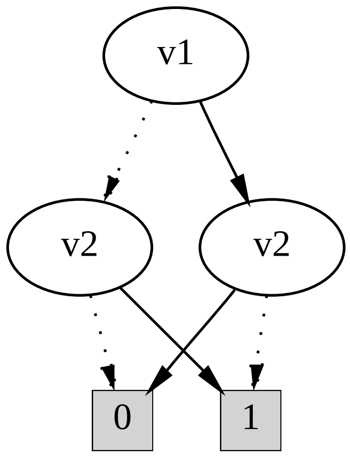


More BDD examples

- **BDD of** `v1` $=$ `v2`



- **BDD of** `v1` $\neq$ `v2`

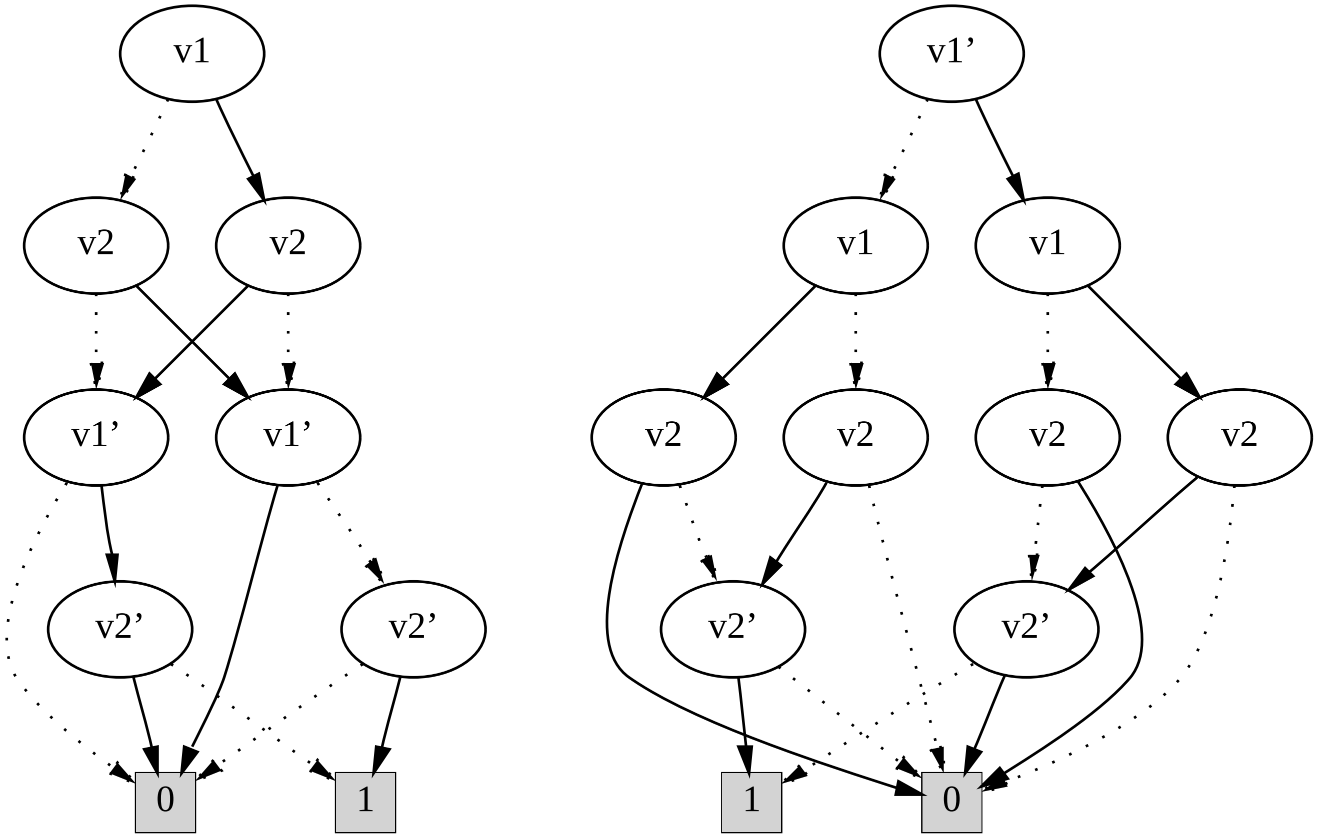


BDD of a transition relation

- **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 $\mathtt{T}$**

  - **b_1, ..., b_n is a satisfying assignment (solution to SAT problem)**

  - **$f\,\mathtt{[}(b_1, \ldots, b_n)/(v_1, \ldots, v_n)\mathtt{]}$ evaluates to $\mathtt{T}$**

  - **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:**

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

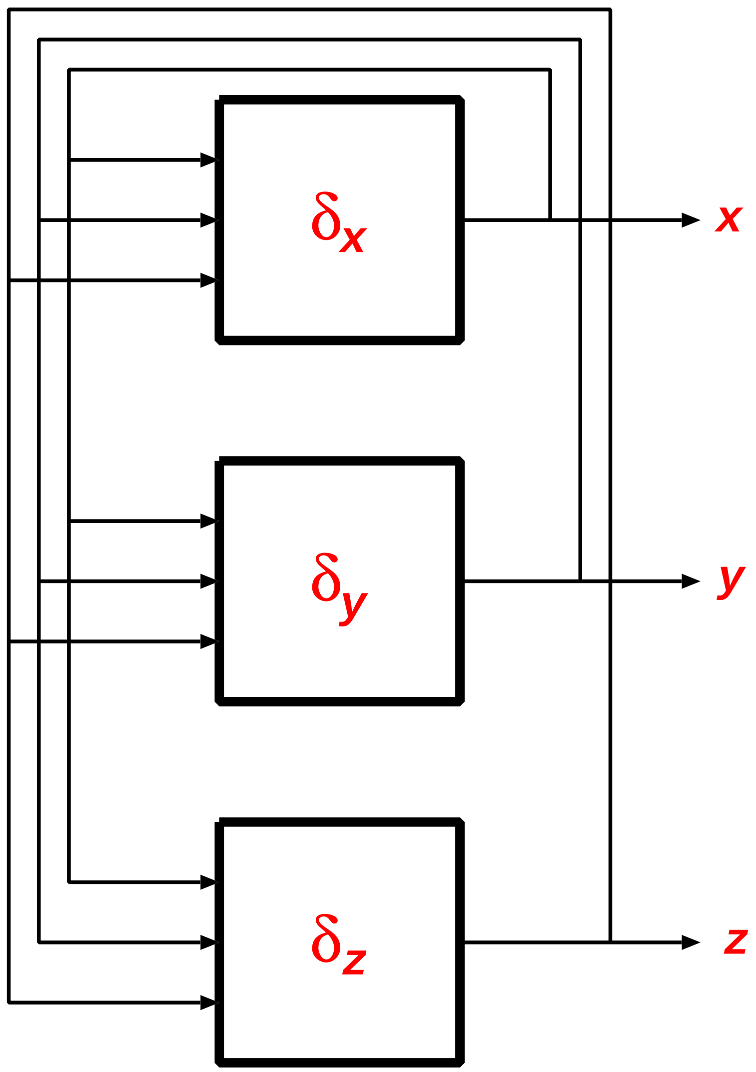
efficient using standard BDD algorithms
(renaming, then conjuction, then existential quantification)

- **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):**

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

Avoiding building big BDDs

- **Transition relation for three machines in parallel**

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

- **Recall:**

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

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

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

- $\mathcal{R}(u,s)$ **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)) \\[6pt]
\;= \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)) \\[6pt]
\;= \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\qquad\qquad\qquad\quad (x=\overline{x} \wedge y=\delta_y(\overline{x},\overline{y},\overline{z}) \wedge z=\overline{z}) \vee \\
\qquad\qquad\qquad\qquad\qquad\quad (x=\overline{x} \wedge y=\overline{y} \wedge z=\delta_z(\overline{x},\overline{y},\overline{z}))) \\[6pt]
\;= (\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})) \\[6pt]
\;= (\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})) \\[6pt]
\;= ((\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}))) \\[6pt]
\;= (\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** $\texttt{Reach}\ \mathcal{R}\ \mathcal{B}\ s\ \Rightarrow\ \mathcal{Q}\ s$ **true?**
- **Compute BDD of** $\texttt{Reach}\ \mathcal{R}\ \mathcal{B}\ s\ \Rightarrow\ \mathcal{Q}\ s$
- **Formula is true if BDD is the single node** $\boxed{1}$
  - **because** $\texttt{T}$ **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 $\texttt{Reach}\ \mathcal{R}\ \mathcal{B}\ s\ \Rightarrow\ \mathcal{Q}\ s$ is not true
- Must exist s satisfying $\texttt{Reach}\ \mathcal{R}\ \mathcal{B}\ s\ \wedge\ \neg\mathcal{Q}\ s$
- Find counterexample algorithm:
  - iteratively generate BDDs of $\texttt{ReachBy}\ i\ \mathcal{R}\ \mathcal{B}\ s$ ($i = 0, 1, \ldots$)
  - at each stage check if $\texttt{ReachBy}\ i\ \mathcal{R}\ \mathcal{B}\ \texttt{s}\ \wedge\ \neg(Q\ \texttt{s})$ satisfiable
  - hence find first $\texttt{n}$ and, using SAT, a state $\texttt{s}_n$ such that

    $$(\texttt{ReachBy}\ \texttt{n}\ \mathcal{R}\ \mathcal{B}\ s\ \wedge\ \neg(Q\ s))\ [\texttt{s}_n/s]$$

    i.e.

    $$\texttt{ReachBy}\ \texttt{n}\ \mathcal{R}\ \mathcal{B}\ \texttt{s}_n\ \wedge\ \neg(Q\ \texttt{s}_n)$$
- Then use BDD SAT to get $\texttt{s}_{n-1}$ where

  $$(\texttt{ReachBy}\,(\texttt{n}{-}1)\,\mathcal{R}\,\mathcal{B}\,s \wedge \mathcal{R}\,(s, \texttt{s}_n))\ [\texttt{s}_{n-1}/s]$$

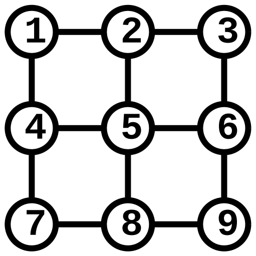
  i.e.

  $$\texttt{ReachBy}\,(n{-}1)\,\mathcal{R}\,\mathcal{B}\,\texttt{s}_{n-1} \wedge \mathcal{R}\,(\texttt{s}_{n-1}, \texttt{s}_n)$$
- Iteratively trace backwards to get $\texttt{s}_n, \ldots, \texttt{s}_0$ where for $0 < i \leq \texttt{n}$:

  $$\texttt{ReachBy}\,(i{-}1)\,\mathcal{R}\,\mathcal{B}\,\texttt{s}_{i-1} \wedge \mathcal{R}\,(\texttt{s}_{i-1}, \texttt{s}_i)$$
- 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 vi **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
```

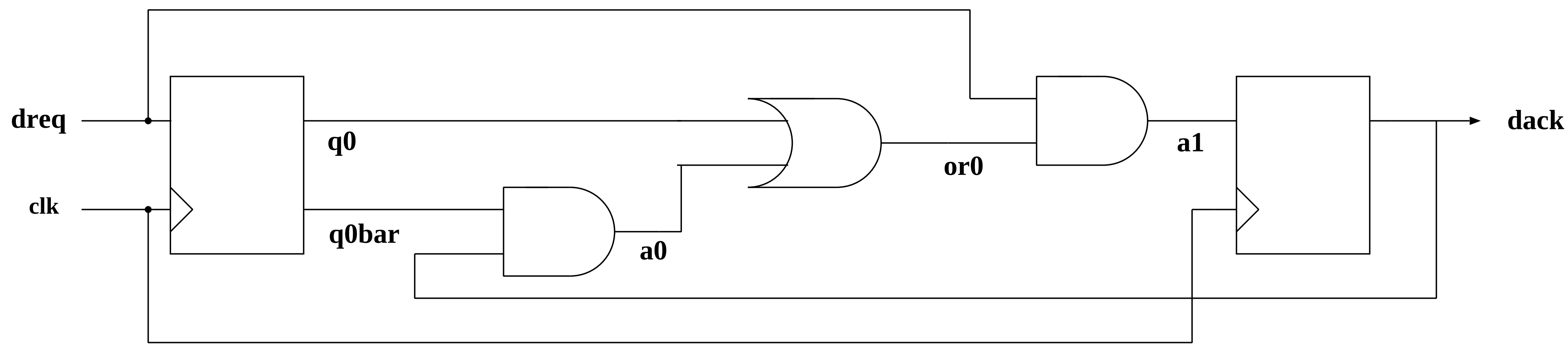
Model checkers can find counter-examples to properties, and sequences of transitions from an initial state to a counter-example state. Thus we could use a model checker to find a trace to a counter-example to the property that `¬Final(v0,v1,v2,v3,v4,v5,v6,v7,v8)`.

Properties

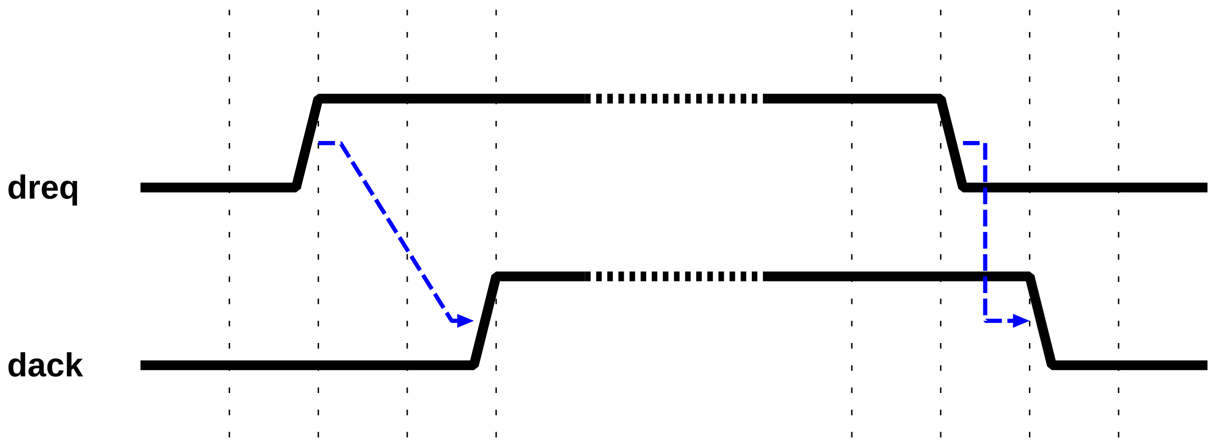
- `Reach` $\mathcal{R}\ \mathcal{B}\ s \ \Rightarrow\ \mathcal{Q}\ s$ means $\mathcal{Q}$ true in all reachable states
- Might want to verify other properties, e.g:
  1. *DeviceEnabled* is always true somewhere along every path starting anywhere (i.e. it holds infinitely often along every path)
  2.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



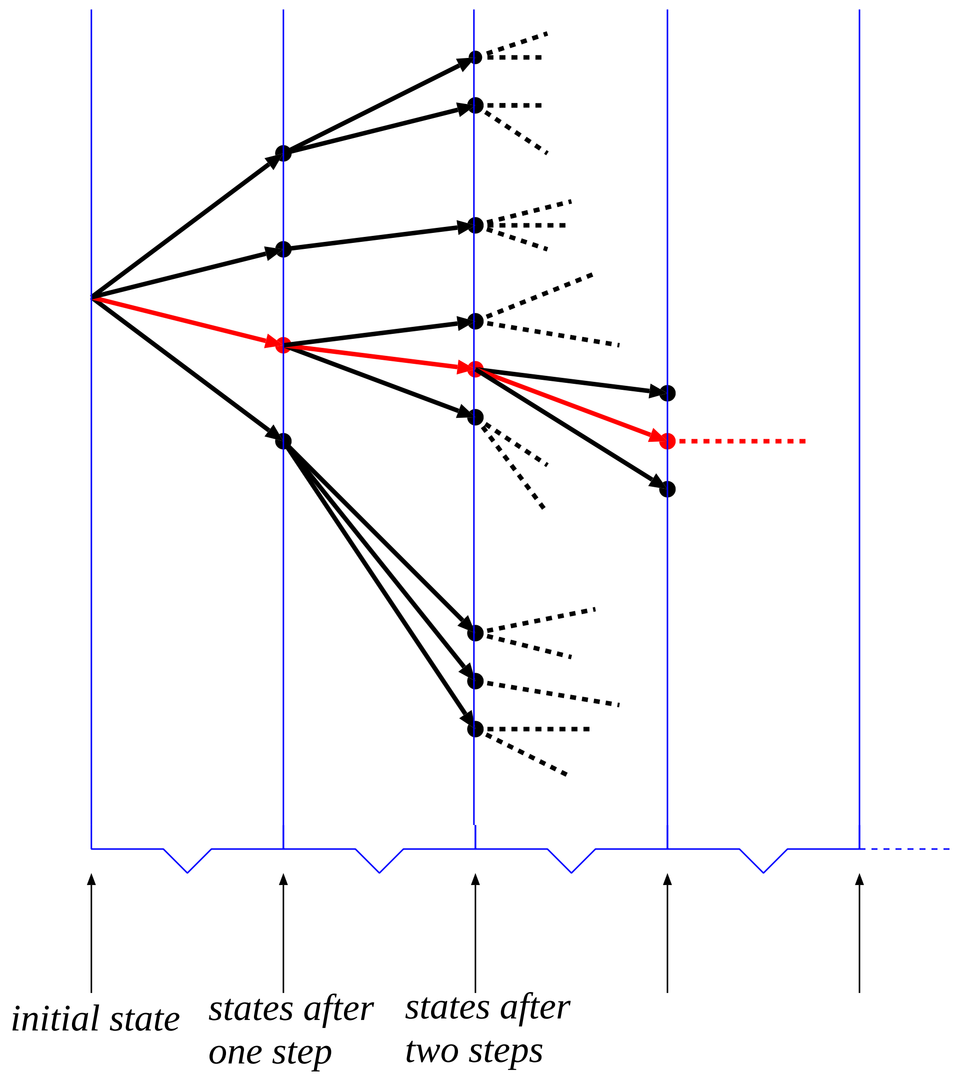
- **Input:** `dreq`, **registers:** `q0, dack`
- **Timing Diagram:**



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 $\alpha \times \alpha \rightarrow bool$**
  - **$\mathcal{R}$ is a transition relation**
  - **α ranges (intuitively) over states**

- **An $\mathcal{R}$-path is a function $\sigma : num \rightarrow \alpha$ 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:**

$$
\begin{array}{rcll}
\mathit{wff} & ::= & \mathtt{Atom}(p) & \text{(Atomic formula)} \\
& \mid & \neg \mathit{wff} & \text{(Negation)} \\
& \mid & \mathit{wff}_1 \wedge \mathit{wff}_2 & \text{(Conjunction)} \\
& \mid & \mathit{wff}_1 \vee \mathit{wff}_2 & \text{(Disjunction)} \\
& \mid & \mathit{wff}_1 \Rightarrow \mathit{wff}_2 & \text{(Implication)} \\
& \mid & \mathbf{AX}\mathit{wff} & \text{(All successors)} \\
& \mid & \mathbf{EX}\mathit{wff} & \text{(Some successors)} \\
& \mid & \mathbf{A}[\mathit{wff}_1\ \mathbf{U}\ \mathit{wff}_2] & \text{(Until – along all paths)} \\
& \mid & \mathbf{E}[\mathit{wff}_1\ \mathbf{U}\ \mathit{wff}_2] & \text{(Until – along some path)}
\end{array}
$$

- **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:**

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

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*

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

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

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

The defined operator AG

- **Define AG**$P = \neg\mathbf{EF}(\neg P)$

- **AG**P **is true if** P **holds everywhere along every** $\mathcal{R}$**-path**

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

- **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

- **EGP 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**

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

- **Exercise: understand the next three slides**

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

- **Continuing:**

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

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

- **Continuing:**

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

- **Exercise: does this correspond to earlier description of $\mathbf{A}[P\mathbf{W}Q]$?**

  - **this exercise illustrates the subtlety of writing CTL!**

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

- **From last slide:**

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

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

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

- **Simplify:**

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

- **By induction on i:**

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

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