**Title:** *Specification and Verification II*

**Lecturer:** **Mike Gordon**

**Class:** **Computer Science Tripos, Part II**

**Duration:** **Twelve lectures**

# Specification and Verification II

**Mike Gordon**

**Overview**

These lecture notes are for the course entitled *Specification and Verification II*. The topic of this course is the specification and verification of hardware. Knowledge of the contents of the Part II course on the specification and verification of software entitled *Specification and Verification I* is assumed.

**Learning Guide**

These notes contain general and background material for the course.

Some of the material in the notes may not be covered in the course, and some additional details and examples are only presented in the lectures.

The examinable material is what is actually covered in the lectures.

**Acknowledgement**

The section entitled *Temporal Abstraction* contains text derived from an early version of the book *Higher Order Logic and Hardware Verification* by Thomas F Melham (Cambridge Tracts in Theoretical Computer Science 31, Cambridge University Press, 1993). The section entitled *Model Checking* contains examples due to John Herbert.

# Contents

Chapter 1

# Hardware Oriented Programs

*Floyd-Hoare logic can be used to verify hardware algorithms. This will be illustrated with addition and multiplication. Although natural numbers can be used to represent words, this leads to messy details so a type of words is introduced. Proofs depend on laws relating bit and word operations to arithmetical operations.*

Suppose we want to multiply two natural numbers $a$ and $b$ represented as $n$-bit words in the usual way. Writing $ab$ to abbreviate $a \times b$:

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

Here $a_i$ is the $i$-th bit of the binary representation of $a$ ($a_0$ being the least significant bit). Then:

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

Multiplying by 2 corresponds to shifting one place to the left and adding a 0 as the least significant bit. Denote this operation by $b \mapsto b{\frown}0$, then:

$$\begin{aligned}
2^{0}b &= b \\
2^{1}b &= b{\frown}0 \\
2^{2}b &= b{\frown}00 \\
&\;\vdots \\
2^{n}b &= b{\frown}\underbrace{0\cdots0}_{n\ 0s}
\end{aligned}$$

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

$$\begin{array}{rr} & a_0 b \\ + & a_1 b\frown 0 \\ + & a_2 b\frown 00 \\ + & a_3 b\frown 000 \\ & \vdots \\ + & a_{n-1} b\frown 0\cdots 0 \end{array}$$

This corresponds to the standard 'high school' algorithm for doing multiplication. In the binary case, the $i^{\text{th}}$ row is either all zeros (if $a_i$ is 0) or $b$ shifted $i$ places to the left (if $a_i$ is 1). If $a$ and $b$ are both $n$-bit words, then the product needs $2n$ bits.

## 1.1 Hoare logic verification of a multiplier

Let $\mathtt{A}[n]$ denote the $n$-th bit (considered as a number 1 or 0) of the binary representation of $\mathtt{A}$ (with $\mathtt{A}[0]$ being the least significant bit).

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

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

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

Below is an annotated specification of the correctness of 'high school' multiplication.

$\{\mathtt{A} = a\ \wedge\ \mathtt{B} = b\ \wedge\ a < 2^{\mathtt{N}}\ \wedge\ b < 2^{\mathtt{N}}\ \wedge\ \mathtt{N} > 0\}$
    `I := 0; PROD := 0;`
    $\{\mathtt{A} = a\ \wedge\ \mathtt{B} = b\ \wedge\ a < 2^{\mathtt{N}}\ \wedge\ b < 2^{\mathtt{N}}\ \wedge\ \mathtt{N} > 0\ \wedge\ \mathtt{I} = 0\ \wedge\ \mathtt{PROD} = 0\}$
    `WHILE I < N DO` $\{\mathtt{I} \leq \mathtt{N}\ \wedge\ 2^{\mathtt{I}}\mathtt{A}[\mathtt{N-1}:\mathtt{I}]\mathtt{B} + \mathtt{PROD} = ab\}$
     `BEGIN PROD := PROD + A[I]` $\times$ `(`$2^{\mathtt{I}}\times$ `B);`
         `I := I + 1;`
     `END`
$\{\mathtt{PROD} = a \times b\}$

The verification conditions are:

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

(ii) $(\mathtt{A} = a \ \wedge \ \mathtt{B} = b \ \wedge \ a < 2^{\mathtt{N}} \ \wedge \ b < 2^{\mathtt{N}} \ \wedge \ \mathtt{N} > 0 \ \wedge \ \mathtt{I} = 0 \ \wedge \ \mathtt{PROD} = 0)$
$\Rightarrow$
$(\mathtt{I} \leq \mathtt{N} \ \wedge \ 2^{\mathtt{I}}\mathtt{A[N-1:I]B} + \mathtt{PROD} = ab)$

(iii) $\mathtt{I} < \mathtt{N} \ \wedge \ (\mathtt{I} \leq \mathtt{N} \ \wedge \ 2^{\mathtt{I}}\mathtt{A[N-1:I]B} + \mathtt{PROD} = ab)$
$\Rightarrow$
$(\mathtt{I+1} \leq \mathtt{N} \ \wedge \ 2^{\mathtt{I+1}}\mathtt{A[N-1:I+1]B} + (\mathtt{PROD} + \mathtt{A[I]} \times (2^{\mathtt{I}} \times \mathtt{B})) = ab)$

(iv) $(\mathtt{I} \leq \mathtt{N} \ \wedge \ 2^{\mathtt{I}}\mathtt{A[N-1:I]B} + \mathtt{PROD} = ab) \ \wedge \ \neg(\mathtt{I} < \mathtt{N}) \Rightarrow \mathtt{PROD} = a \times b$

The first VC (i) is trivial.

The second VC (ii) is true because if $\mathtt{N} > 0$ and $\mathtt{A} < 2^{\mathtt{N}}$ then $\mathtt{A[N-1:0]} = \mathtt{A}$.

For the third VC (iii), consider separately the cases (a) that $\mathtt{I+1} = \mathtt{N}$ and (b) that $\mathtt{I+1} < \mathtt{N}$. In case (a) $\mathtt{A[N-1:I+1]} = 0$ and $\mathtt{A[N-1:I]} = \mathtt{A[I]}$ and the VC follows easily. In case (b) use $2 \times \mathtt{A[N-1:I+1]} + \mathtt{A[I]} = \mathtt{A[N-1:I]}$.

The final verification condition (iv) follows because if $\mathtt{I} = \mathtt{N}$, then $\mathtt{A[N-1:I]} = 0$.

**Exercise:**
Prove

```
⊢ {A = a  ∧  B = b  ∧  a < 2^N  ∧  b < 2^N  ∧  N > 0}
   BEGIN
    VAR I;
    I := 0; PROD := 0;
    WHILE I < N DO
     BEGIN PROD := PROD + A[I]×B;
           B := 2 × B;
           I := I + 1;
     END
   END
  {PROD = a × b}
```

**Exercise:**
Prove

```
⊢ {A = a  ∧  B = b  ∧  a < 2^N  ∧  b < 2^N  ∧  N > 0}
    PROD := 0;
    FOR I := 0 UNTIL N-1 DO PROD := PROD + A[I]×(2^I×B)
  {PROD = a × b}
```

**Exercise:**
Prove

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

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

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

The program in this last exercise corresponds directly to a possible hardware implementation. One has three registers A, B and PROD. Initially PROD is set to 0 and A and B contain the numbers to be multiplied. The I-th step of the multiplication consists in adding A[I] $\times$ B to PROD and then shifting B one bit to the left (i.e. multiplying it by 2). Register PROD holds the product and so must be 2N-bits wide. However, register B is shifted N times and so it must also be 2N bits wide. This is redundant, because B always just contains the same bits plus some 0s shifted in from the right. An optimisation is to shift PROD to the right instead of shifting B to the left. Then B can be an N-bit register.

The program below exploits this idea.

Note that shifting right corresponds to division by 2.

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

Consider the case when $\texttt{N} = 4$.

Initially $\texttt{PROD} = 0$, so PROD div 2 = 0.

The multiplication according to this program proceeds as follows:

$\texttt{I} = 0$, $\texttt{PROD} = 2^4\texttt{A[0]B}$
$\texttt{PROD} = 2^3\texttt{A[0]B}$

$\texttt{I} = 1$, $\texttt{PROD} = 2^4\texttt{A[1]B} + 2^3\texttt{A[0]B}$
$\texttt{PROD} = 2^3\texttt{A[1]B} + 2^2\texttt{A[0]B}$

$\texttt{I} = 2$, $\texttt{PROD} = 2^4\texttt{A[2]B} + 2^3\texttt{A[1]B} + 2^2\texttt{A[0]B}$
$\texttt{PROD} = 2^3\texttt{A[2]B} + 2^2\texttt{A[1]B} + 2^1\texttt{A[0]B}$

$\texttt{I} = 3$, $\texttt{PROD} = 2^4\texttt{A[3]B} + 2^3\texttt{A[2]B} + 2^2\texttt{A[1]B} + 2^1\texttt{A[0]B}$
$\texttt{PROD} = 2^3\texttt{A[3]B} + 2^2\texttt{A[2]B} + 2^1\texttt{A[1]B} + 2^0\texttt{A[0]B}$

Recall the verification conditions for FOR-commands:

**FOR-commands**

The verification conditions generated from

$$\{P\}\ \texttt{FOR}\ V\texttt{:=}E_1\ \texttt{UNTIL}\ E_2\ \texttt{DO}\ \{R\}\ C\ \{Q\}$$

are

(i) $P \ \Rightarrow\ R\texttt{[}E_1/V\texttt{]}$

(ii) $R\texttt{[}E_2{+}1/V\texttt{]}\ \Rightarrow\ Q$

(iii) $P\ \wedge\ E_2 < E_1\ \Rightarrow\ Q$

(iv) the verification conditions generated by

$$\{R\ \wedge\ E_1 \leq\ V\ \wedge\ V \leq E_2\}\ C\ \{R\texttt{[}V+1/V\texttt{]}\}$$

(v) the syntactic condition that neither $V$, nor any variable occurring in $E_1$ or $E_2$, is assigned to inside $C$.

The program is easily verified by computing the VCs for:

$$
\begin{array}{l}
\vdash\ \{\texttt{A} = a\ \wedge\ \texttt{B} = b\ \wedge\ a < 2^{\texttt{N}}\ \wedge\ b < 2^{\texttt{N}}\ \wedge\ \texttt{N} > 0\}\\
\quad\ \texttt{PROD := 0;}\ \{\texttt{A} = a\ \wedge\ \texttt{B} = b\ \wedge\ a < 2^{\texttt{N}}\ \wedge\ b < 2^{\texttt{N}}\ \wedge\ \texttt{N} > 0\ \wedge\ \texttt{PROD} = 0\}\\
\quad\ \texttt{FOR I := 0 UNTIL N-1 DO}\ \{\texttt{PROD} + 2^{\texttt{N}}\texttt{A[N}{-}1:\texttt{I]B} = 2^{\texttt{N}-\texttt{I}}ab\}\\
\quad\quad \texttt{BEGIN}\\
\quad\quad\ \ \texttt{PROD := }2^{\texttt{N}}\texttt{A[I]B + PROD;}\\
\quad\quad\ \ \texttt{PROD := PROD div 2}\\
\quad\quad \texttt{END}\\
\quad \{\texttt{PROD} = ab\}
\end{array}
$$

These are:

(i) $(\texttt{A} = a\ \wedge\ \texttt{B} = b\ \wedge\ a < 2^{\texttt{N}}\ \wedge\ b < 2^{\texttt{N}}\ \wedge\ \texttt{N} > 0)$
$\Rightarrow$
$(\texttt{A} = a\ \wedge\ \texttt{B} = b\ \wedge\ a < 2^{\texttt{N}}\ \wedge\ b < 2^{\texttt{N}}\ \wedge\ \texttt{N} > 0\ \wedge\ 0 = 0)$

(ii) $(\texttt{A} = a\ \wedge\ \texttt{B} = b\ \wedge\ a < 2^{\texttt{N}}\ \wedge\ b < 2^{\texttt{N}}\ \wedge\ \texttt{N} > 0\ \wedge\ \texttt{PROD} = 0)$
$\Rightarrow$
$(\texttt{PROD} + 2^{\texttt{N}}\texttt{A[N-1}:\texttt{0]B} = 2^{\texttt{N-0}}ab)$

(iii) $(\mathtt{PROD} + 2^{\mathtt{N}}\mathtt{A[N}{-}1 : \mathtt{N]B} = 2^{\mathtt{N}-\mathtt{N}}ab)$
$\Rightarrow$
$(\mathtt{PROD} = ab)$

(iv) $(\mathtt{A} = a \ \wedge\ \mathtt{B} = b \ \wedge\ a < 2^{\mathtt{N}} \ \wedge\ b < 2^{\mathtt{N}} \ \wedge\ \mathtt{N} > 0 \ \wedge\ \mathtt{PROD} = 0) \ \wedge\ \mathtt{N\text{-}1} < 0$
$\Rightarrow$
$(\mathtt{PROD} = ab)$

(v) $(\mathtt{PROD} + 2^{\mathtt{N}}\mathtt{A[N}{-}1 : \mathtt{I]B} = 2^{\mathtt{N}-\mathtt{I}}ab) \ \wedge\ 0 \leq \mathtt{I} \ \wedge\ \mathtt{I} \leq \mathtt{N\text{-}1}$
$\Rightarrow$
$(((2^{\mathtt{N}}\mathtt{A[I]B} + \mathtt{PROD})\ \mathsf{div}\ 2) + 2^{\mathtt{N}}\mathtt{A[N\text{-}1 : I\text{+}1]B} = 2^{\mathtt{N}-(\mathtt{I+1})}ab)$

VCs (i), (ii), (iii) and (iv) are straightforward (note that in the case of (iv) the antecedent $\mathtt{N\text{-}1} < 0$ of the implication is false).

The final VC (v) is also quite easy: assume the antecedent of the implication and then deduce the consequent. Thus assume $0 \leq \mathtt{I} \ \wedge\ \mathtt{I} \leq \mathtt{N\text{-}1}$ and
$\mathtt{PROD} + 2^{\mathtt{N}}\mathtt{A[N}{-}1 : \mathtt{I]B} = 2^{\mathtt{N}-\mathtt{I}}ab$
If this equation holds, then as $\mathtt{N\text{-}I} > 0$, PROD must be divisible by 2 and hence dividing both sides of the equation by 2:
$(\mathtt{PROD}\ \mathsf{div}\ 2) + 2^{\mathtt{N}-1}\mathtt{A[N}{-}1 : \mathtt{I]B} = 2^{\mathtt{N}-(\mathtt{I+1})}ab$
Now as $\mathtt{I} \leq \mathtt{N\text{-}1}$ we have: $\mathtt{A[N}{-}1 : \mathtt{I]} = 2{\times}\mathtt{A[N}{-}1 : \mathtt{I\text{+}1]} + \mathtt{A[I]}$ so:
$(\mathtt{PROD}\ \mathsf{div}\ 2) + 2^{\mathtt{N}-1}(2{\times}\mathtt{A[N}{-}1 : \mathtt{I\text{+}1]} + \mathtt{A[I]})\mathtt{B} = 2^{\mathtt{N}-(\mathtt{I+1})}ab$
which simplifies to:
$(\mathtt{PROD}\ \mathsf{div}\ 2) + 2^{\mathtt{N}}\mathtt{A[N}{-}1 : \mathtt{I\text{+}1]B} + 2^{\mathtt{N}-1}\mathtt{A[I]B} = 2^{\mathtt{N}-(\mathtt{I+1})}ab$
which is equivalent to:
$(\mathtt{PROD}\ \mathsf{div}\ 2) + 2^{\mathtt{N}}\mathtt{A[N}{-}1 : \mathtt{I\text{+}1]B} + ((2^{\mathtt{N}}\mathtt{A[I]B})\ \mathsf{div}\ 2) = 2^{\mathtt{N}-(\mathtt{I+1})}ab$
which can be rearranged to the consequent of the implication of VC (v).
Note that the word representing $2^{\mathtt{N}}\mathtt{A[I]B}$ consists of the word representing A[I]B shifted N places to the left. Thus adding $2^{\mathtt{N}}\mathtt{A[I]B}$ to an 2N-bit word will not change the N least significant bits, so the addition can be done by adding A[I]B to the N most significant bits and then concatenating the result with the N least significant bits. Thus if $\mathtt{PROD} < 2^{(\mathtt{2N})}$ then:
$\mathtt{PROD} = 2^{\mathtt{N}}\mathtt{PROD[2N\text{-}1 : N]} + \mathtt{PROD[N\text{-}1 : 0]}$
hence:
$2^{\mathtt{N}}\mathtt{A[I]B} + \mathtt{PROD} \ = \ 2^{\mathtt{N}}(\mathtt{A[I]B} + \mathtt{PROD[2N\text{-}1 : N]}) + \mathtt{PROD[N\text{-}1 : 0]}$
and the first sum on the right hand side of this equation (the one in the brackets) is the addition of two N-bit numbers, which might require N+1 bits.

Let us split PROD into two N-bit variables PROD1 and PROD2 holding PROD[N-1 : 0] and PROD[2N-1 : N], respectively. Thus:

$\mathtt{PROD} = 2^{\mathtt{N}}\mathtt{PROD2} + \mathtt{PROD1}$

Let SUM be an N+1 bit variable for holding the result of $\mathtt{A[I]B} + \mathtt{PROD2}$.

Then the program above can then be reformulated as:

```
SUM := 0; PROD1 := 0; PROD2 := 0;
FOR I := 0 UNTIL N-1 DO BEGIN
                         SUM := A[I]B + PROD2;
                         PROD1 := ((2^N SUM + PROD1) div 2)[N-1:0];
                         PROD2 := ((2^N SUM + PROD1) div 2)[2N-1:N];
                        END
```

**Exercise:** Write an annotated partial correctness specification of this program that says that it does multiplication, generate the verification conditions and outline why they are true. [**Warning:** I have not formally verified this program – it might have bugs!]

As a final optimisation, the initially unused space in PROD1 can be used to store A. After each right shift (div 2) a bit of A is lost, but we can cunningly arrange that the I-th bit of A is the 0-th bit of PROD1 on iteration I so bits of A are only lost after they have been 'used'.

```
SUM := 0; PROD1 := A; PROD2 := 0;
FOR I := 0 UNTIL N-1 DO BEGIN
                         SUM := PROD1[0]B + PROD2;
                         PROD1 := ((2^N SUM + PROD1) div 2)[N-1:0];
                         PROD2 := ((2^N SUM + PROD1) div 2)[2N-1:N];
                        END
```

**Exercise:** Write an annotated partial correctness specification of this program that says that it does multiplication, generate the verification conditions and outline why they are true. [**Warning:** I have not formally verified this program – it might have bugs!]

The multiplier discussed above modelled $n$-bit words as natural numbers less than $2^n$. Hardware operations are usually specified in terms of bit manipulations like shifts, word concatenations, subword extractions etc. Although these operations can be represented arithmetically (left-shift is multiplication by 2, right-shift is division by 2 etc.) it turns out to be convenient to work with a separate type of words. An important advantage of this is that words can carry their size and so the concatenation of an $m$-bit word $w_1$ with an $n$-bit word $w_2$ can be unambiguously written as $w_1 \frown w_2$. The arithmetical counterpart to concatenation needs to be parameterised on $n$. Thus, if $w_1$ denotes number $a$ and $w_2$

denotes $b$ then $w_1 \frown w_2$ denotes $2^n a + b$ – an expression depending on $n$. Thus $a \frown b$ doesn't make sense and one would need to write something like $a \frown_n b$, – i.e. have a family of operators $\frown_n$ instead of a single $\frown$.

For such reasons, a type of words is useful.

## 1.2 Words

A word is a fixed-size array of bits. Bits are represented by the truth values $\mathsf{T}$ and $\mathsf{F}$ of type *bool*. Following standard practice, the two numerals 1 and 0 will sometimes be used to denote the bits $\mathsf{T}$ and $\mathsf{F}$, respectively.

The 'empty word' has zero length and all other words have length greater than zero. The length of a word is denoted by $|w|$.

Two words are equal if they have both the same length and the same bits:

$$(w_1 = w_2) \quad \equiv \quad (|w_1| = |w_2|) \ \wedge \ \forall i.\ i < |w_1| \Rightarrow w_1\,\mathtt{[}i\mathtt{]} = w_2\,\mathtt{[}i\mathtt{]}$$

Individual words will be written in the stardard way with the least significant bit at the right. For example, here is an 8-bit word: 00011011:

$$|00011011| \;=\; 8$$

If $b$ is a bit and $n > 0$ then $b^n$ is the concatenation of $n$ $b$s, e.g. $0^3 = 000$. Clearly $|b^n| = n$.

## 1.3 Selecting bits and subwords

The $n^{\mathrm{th}}$ bit of a word $w$ will be denoted by $w\,\mathtt{[}n\mathtt{]}$, where the least significant bit is considered to be the $0^{\mathrm{th}}$ bit. Thus:

$$b_{n-1} \cdots b_0\,\mathtt{[}i\mathtt{]} \;=\; b_i$$

When $n \geq |w|$ the convention adopted is that $w\,\mathtt{[}n\mathtt{]} \;=\; 0$. Thus in general:

$$b_{n-1} \cdots b_0\,\mathtt{[}i\mathtt{]} \;=\; (i < n \rightarrow b_i \mid 0)$$

For $m \geq n$ the notation $w\,\mathtt{[}m : n\mathtt{]}$ denotes the word consisting of bits $m$ to $n$ of $w$.

$$b_{n-1} \cdots b_i \cdots b_j \cdots b_0\,\mathtt{[}i : j\mathtt{]} \;=\; b_i \cdots b_j$$

The convention is adopted that if $m < n$ then $w\,\mathtt{[}m : n\mathtt{]}$ denotes the empty word, and if either $m > |w|$ or $n > |w|$, then $w$ is padded (at its left) with 0s. Thus if $m \geq n$, then:

$$|w\,\mathtt{[}m : n\mathtt{]}| \;=\; m - n + 1$$

For example:

$$\begin{array}{rcl} 100101\,[4:1] & = & 0010 \\ 100101\,[7:1] & = & 0010010 \\ 100101\,[8:6] & = & 000 \\ 100101\,[1:2] & = & \text{the empty word} \end{array}$$

$w\,[n:n]$ denotes the 1-bit word consisting of $w\,[n]$.

The type of all words is *word*, thus $w\,[n] : \mathit{bool}$ but $w\,[n:n] : \mathit{word}$.

Thus for $m \geq n$, the subword selection $w\,[m:n]$ is defined by:

$$(w\,[m:n])\,[i] \;=\; (i{+}n \leq m \rightarrow w\,[i{+}n] \mid 0)$$

Obvious facts can be derived from this definition, for example:

$$\begin{array}{lcl} |w| > 0 & \Rightarrow & w\,[|w|{-}1:0] \;=\; w \\ p{+}n \leq m & \Rightarrow & w\,[m:n]\,[p:q] \;=\; w\,[p{+}n:q{+}n] \end{array}$$

Each of these conditional equations are verified by first checking the equated words have the same length (obvious) and then checking that corresponding bits are equal.

In the first example:

$$\begin{array}{rcl} w\,[|w|{-}1:0]\,[i] & = & (i{+}0 \leq |w|{-}1 \rightarrow w\,[i{+}0] \mid 0) \\ & = & (i < |w| \rightarrow w\,[i] \mid 0) \\ & = & w\,[i] \end{array}$$

For the second example:

$$\begin{array}{rcl} (w\,[m:n]\,[p:q])\,[i] & = & (i{+}q \leq p \rightarrow w\,[m:n]\,[i{+}q] \mid 0) \\ & = & (i{+}q \leq p \rightarrow ((i{+}q){+}n \leq m \rightarrow w\,[(i{+}q){+}n] \mid 0) \mid 0) \\ & = & ((i{+}q \leq p) \wedge ((i{+}q){+}n \leq m) \rightarrow w\,[(i{+}q){+}n] \mid 0) \end{array}$$

and

$$\begin{array}{rcl} w\,[p{+}n:q{+}n]\,[i] & = & (i{+}(q{+}n) \leq p{+}n \rightarrow w\,[i{+}(q{+}n)] \mid 0) \\ & = & (i{+}q \leq p \rightarrow w\,[i{+}(q{+}n)] \mid 0) \end{array}$$

Now, if $p + n \leq m$ then clearly:

$$(i{+}q \leq p) \;\Rightarrow\; ((i{+}q){+}n \leq m)$$

hence (as $A{\Rightarrow}B \;\equiv\; A{\wedge}B = A$):

$$(i{+}q \leq p) \wedge ((i{+}q){+}n \leq m) \;\;\equiv\;\; (i{+}q \leq p)$$

and so $(w\,[m:n]\,[p:q])\,[i] \;=\; w\,[p{+}n:q{+}n]\,[i]$.

Bits and 1-bit words are different types. The word corresponding to a bit $b$ is $\mathsf{Bw}(b)$. Thus: $\mathsf{Bw}(b)\,[0] \;=\; b$.

## 1.4 Representing numbers

Words represent natural numbers and integers in a standard way:

- 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[1]: $b_{n-1}\cdots b_0$ represents $-2^{n-1}{\times}b_{n-1} + 2^{n-2}{\times}b_{n-2} + \cdots + 2^0{\times}b_0$

The notation $\mathsf{V}(w)$ will be used to denote the natural number represented by a word, i.e.:

$$\mathsf{V}(b_{n-1}\cdots b_0) \;=\; 2^{n-1}{\times}b_{n-1} + 2^{n-2}{\times}b_{n-2} + \cdots + 2^0{\times}b_0$$

Words can, of course, also represent other numeric or non-numeric values (e.g. floating point numbers or opcodes).

The number represented by a bit $b$ will be denoted by $\mathsf{Bv}(b)$, thus:

$$\mathsf{Bv}(\mathsf{T}) \;=\; 1 \quad \text{and} \quad \mathsf{Bv}(\mathsf{F}) \;=\; 0$$

Word operations are related to arithmetical operations via $\mathsf{div}$ and $\mathsf{mod}$, where

$$\begin{array}{lcl} m \;\mathsf{div}\; n & = & \text{integer part of dividing } m \text{ by } n \\ m \;\mathsf{mod}\; n & = & \text{remainder after dividing } m \text{ by } n \end{array}$$

The fundamental properties relating $\mathsf{div}$ and $\mathsf{mod}$ are:

$$\begin{array}{l} 0 < n \;\Rightarrow\; n \times (m \;\mathsf{div}\; n) + m \;\mathsf{mod}\; n \;=\; m \\ 0 < n \;\Rightarrow\; (m \;\mathsf{mod}\; n) < n \end{array}$$

Note the following:

$$\begin{array}{l} \mathsf{Bv}(w\,[n]\,) \;=\; (\mathsf{V}(w) \;\mathsf{div}\; 2^n) \;\mathsf{mod}\; 2 \\ \mathsf{V}(w\,[m{+}n : n]\,) \;=\; (\mathsf{V}(w) \;\mathsf{div}\; 2^n) \;\mathsf{mod}\; 2^{m+1} \\ \mathsf{V}(w\,[n : 0]\,) \;=\; 2^n \times \mathsf{Bv}(w\,[n]\,) + \mathsf{V}(w\,[n{-}1 : 0]\,) \end{array}$$

Whilst there is a canonical natural number $\mathsf{V}(w)$ associated with a word $w$, there is no canonical inverse function, since e.g.:

$$\begin{array}{lcl} \mathsf{V}(101) & = & 5 \\ \mathsf{V}(0101) & = & 5 \\ \mathsf{V}(00101) & = & 5 \\ & \vdots & \end{array}$$

For any size $n$, the function $\mathsf{W}\ n$ maps a number $m$ to the unique $n$-bit word $w$ such that $\mathsf{V}(w) = m \;\mathsf{mod}\; 2^n$, i.e.:

$$\begin{array}{l} \mathsf{V}(\mathsf{W}\ n\ m) \;=\; m \;\mathsf{mod}\; 2^n \\ \mathsf{W}\ |w|\ (\mathsf{V}(w)) \;=\; w \end{array}$$

For example, $\mathsf{W}\ 5\ 143 \;=\; 01111$.

[1] Two's complement representation.

## 1.5 Operations on bits and words

The standard infixed binary operators of Boolean algebra are $\wedge$ ('and'), $\vee$ ('or'), $\oplus$ ('exclusive or' or 'unequal') and $=$ ('if and only if' or 'equals'). Note that these are all commutative and associative. Negation, $\neg$, is a prefixed unary operator. These operators will be extended (overloaded) to operate bitwise on words.

The concatenation of an $m$-bit word $w_1$ with an $n$-bit word $w_2$ is denoted by $w_1{}^\frown w_2$. This is an $m{+}n$-bit word. Note the following properties:

$$\begin{array}{rcl}
0 < |w_2| & \Rightarrow & (w_1{}^\frown w_2)\,[|w_2|{-}1:0] \;=\; w_2 \\
0 < |w_1| & \Rightarrow & (w_1{}^\frown w_2)\,[|w_1|{+}|w_2|{-}1:|w_2|] \;=\; w_1 \\
 & & \mathsf{V}(w_1{}^\frown w_2) \;=\; 2^{|w_2|}\mathsf{V}(w_1) + \mathsf{V}(w_2) \\
p < |w_2| & \Rightarrow & (w_1{}^\frown w_2)\,[p:q] \;=\; w_2\,[p:q] \\
p \geq q \wedge q \geq |w_2| & \Rightarrow & (w_1{}^\frown w_2)\,[p:q] \;=\; w_1\,[p{-}|w_2|:q{-}|w_2|] \\
p \geq |w_2| \;\wedge\; q < |w_2| & \Rightarrow & (w_1{}^\frown w_2)\,[p:q] \;=\; w_1\,[p{-}|w_2|:0]{}^\frown w_2\,[|w_2|{-}1:q]
\end{array}$$

**Exercise:** are any of the preconditions (antecedents of $\Rightarrow$) redundant?

The 'array update' notation $w\{n{\leftarrow}b\}$ denotes a word identical to $w$, except that $w\,[n] = b$ (when $n \geq |w|$, $w$ is extended with 0s before updating). Note that:

$$\mathsf{V}(w\{n{\leftarrow}b\}\,[n:0]) \;=\; 2^n \times \mathsf{Bv}(b) + \mathsf{V}(w\,[n{-}1:0])$$

## 1.6 Arithmetic on bits and words

The sum of two bits $a$ and $b$ and a carry-in bit $c$ is computed by $a \oplus b \oplus c$ and the carry-out by $(a \wedge b) \vee (c \wedge (a \oplus b))$. This is verified by the equations:

$$\begin{array}{rcl}
\mathsf{Bv}(a \oplus b \oplus c) & = & (\mathsf{Bv}(a) + \mathsf{Bv}(b) + \mathsf{Bv}(c)) \;\mathsf{mod}\; 2 \\
\mathsf{Bv}((a \wedge b) \vee (c \wedge (a \oplus b))) & = & (\mathsf{Bv}(a) + \mathsf{Bv}(b) + \mathsf{Bv}(c)) \;\mathsf{div}\; 2
\end{array}$$

These are most simply verified by exhaustive enumeration. The equation for the sum is verified by:

| $a$ | $b$ | $c$ | $\mathsf{Bv}(a \oplus b \oplus c)$ | $(\mathsf{Bv}(a) + \mathsf{Bv}(b) + \mathsf{Bv}(c))$ 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 |

The equation for the carry is verified by:

| $a$ | $b$ | $c$ | $\mathsf{Bv}((a \wedge b) \vee (c \wedge (a \oplus b)))$ | $(\mathsf{Bv}(a) + \mathsf{Bv}(b) + \mathsf{Bv}(c))$ 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 |

The addition of two non-empty words $w_1$ and $w_2$ is denoted by $w_1 \uplus w_2$ and defined by:

$$w_1 \uplus w_2 \;=\; \mathsf{W}\ (\mathsf{max}(|w_1|, |w_2|) + 1)\ (\mathsf{V}(w_1) + \mathsf{V}(w_2))$$

$w_1 \uplus w_2$ satisfies:

$$\begin{aligned} |w_1 \uplus w_2| &= \mathsf{max}(|w_1|, |w_2|) + 1 \\ \mathsf{V}(w_1 \uplus w_2) &= \mathsf{V}(w_1) + \mathsf{V}(w_2) \\ w \uplus (\mathsf{W}\ |w|\ 0) &= 0 \frown w \end{aligned}$$

## 1.7 Verification of a ripple-carry adder

Let $R$ be:

$$\begin{aligned} &2^{\mathtt{I}}\mathsf{Bv}(\mathtt{CARRY}) + \mathsf{V}(\mathtt{SUM[I}-\mathtt{1:0]}) = \mathsf{V}(\mathtt{A[I}-\mathtt{1:0]}) + \mathsf{V}(\mathtt{B[I}-\mathtt{1:0]}) \\ &\wedge\ \mathtt{A} = w_1\ \wedge\ \mathtt{B} = w_2 \end{aligned}$$

Consider the following annotated specification:

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

```
  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
```

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

Here, `A`, `B` are `SUM` are `N`-bit words, but `CARRY` is a bit (truthvalue) and `I` is an integer. The verification conditions (VCs) are:

(i) $(\mathtt{A} = w_1 \;\wedge\; \mathtt{B} = w_2 \;\wedge\; \mathtt{SUM} = \mathsf{W}\ \mathtt{N}\ 0 \;\wedge\; \mathtt{CARRY} = \mathsf{F}$
$\wedge\; |w_1| \leq \mathtt{N} \;\wedge\; |w_2| \leq \mathtt{N} \;\wedge\; \mathtt{N} > 0)$
$\Rightarrow$
$(2^{\mathtt{0}}\mathsf{Bv}(\mathtt{CARRY}) + \mathsf{V}(\mathtt{SUM[0}{-}\mathtt{1:0]}) = \mathsf{V}(\mathtt{A[0}{-}\mathtt{1:0]}) + \mathsf{V}(\mathtt{B[0}{-}\mathtt{1:0]})$
$\wedge\; \mathtt{A} = w_1 \;\wedge \mathtt{B} = w_2)$

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

(iii) $(\mathtt{A} = w_1 \;\wedge\; \mathtt{B} = w_2 \;\wedge\; \mathtt{SUM} = \mathsf{W}\ \mathtt{N}\ 0 \;\wedge\; \mathtt{CARRY} = \mathsf{F}$
$\wedge\; |w_1| \leq \mathtt{N} \;\wedge\; |w_2| \leq \mathtt{N} \;\wedge\; \mathtt{N} > 0)$
$\wedge\; (\mathtt{N\text{-}1} < 0) \;\Rightarrow\; (\cdots)$

(iv) VCs from

$\{R \;\wedge\; 0 \leq \mathtt{I} \;\wedge\; \mathtt{I} \leq \mathtt{N\text{-}1}\}$
$\mathtt{SUM[I]:=A[I]}\oplus\mathtt{B[I]}\oplus\mathtt{CARRY;}$
$\mathtt{CARRY:=(A[I]}\wedge\mathtt{B[I])}\vee\mathtt{(CARRY}\wedge\mathtt{(A[I]}\oplus\mathtt{B[I]));}$
$\{R\mathtt{[I+1/I]}\}$

The VCs (i), (ii) and (iii) are obviously true.

There is one VC from (iv):

$\big(2^{\mathtt{I}}\times\mathsf{Bv}(\mathtt{CARRY}) + \mathsf{V}(\mathtt{SUM[I\text{-}1:0]}) = \mathsf{V}(\mathtt{A[I\text{-}1:0]}) + \mathsf{V}(\mathtt{B[I\text{-}1:0]})$
$\wedge\; \mathtt{A} = w_1 \;\wedge\; \mathtt{B} = w_2\big) \;\wedge\; 0 \leq \mathtt{I} \;\wedge\; \mathtt{I} \leq \mathtt{N\text{-}1}$
$\Rightarrow$
$\big((2^{\mathtt{I+1}}\times\mathsf{Bv}((\mathtt{A[I]} \wedge \mathtt{B[I]}) \vee (\mathtt{CARRY} \wedge (\mathtt{A[I]} \oplus \mathtt{B[I]})))$
$+\; \mathsf{V}(\mathtt{SUM}\{\mathtt{I}{\leftarrow}\mathtt{A[I]} \oplus \mathtt{B[I]} \oplus \mathtt{CARRY}\}\mathtt{[I:0]})$
$=\; \mathsf{V}(\mathtt{A[I:0]}) + \mathsf{V}(\mathtt{B[I:0]})\big)$
$\wedge\; \mathtt{A} = w_1 \;\wedge\; \mathtt{B} = w_2\big)$

It was previously established by exhaustive enumeration that:

$$\mathsf{Bv}(a \oplus b \oplus c) \;=\; (\mathsf{Bv}(a) + \mathsf{Bv}(b) + \mathsf{Bv}(c)) \;\mathsf{mod}\; 2$$
$$\mathsf{Bv}((a \wedge b) \vee (c \wedge (a \oplus b))) \;=\; (\mathsf{Bv}(a) + \mathsf{Bv}(b) + \mathsf{Bv}(c)) \;\mathsf{div}\; 2$$

Recall also that:

$$\mathsf{V}(w\{n{\leftarrow}b\}[n:0]) \;=\; 2^n \times \mathsf{Bv}(b) + \mathsf{V}(w[n{-}1:0])$$

Thus:

$$\begin{aligned} &2^{\mathtt{I+1}}\times\mathsf{Bv}((\mathtt{A[I]} \wedge \mathtt{B[I]}) \vee (\mathtt{CARRY} \wedge (\mathtt{A[I]} \oplus \mathtt{B[I]}))) \\ &\quad+\; \mathsf{V}(\mathtt{SUM}\{\mathtt{I}{\leftarrow}\mathtt{A[I]} \oplus \mathtt{B[I]} \oplus \mathtt{CARRY}\}\mathtt{[I:0]}) \\ &\quad=\; \mathsf{V}(\mathtt{A[I:0]}) + \mathsf{V}(\mathtt{B[I:0]}) \end{aligned}$$

is equivalent to

$$\begin{aligned}
&2^{\mathtt{I+1}}\times\big((\mathsf{Bv}(\mathtt{A[I]})+\mathsf{Bv}(\mathtt{B[I]})+\mathsf{Bv}(\mathtt{CARRY}))\ \mathsf{div}\ 2\big)\\
&\quad+\ 2^{\mathtt{I}}\times\big((\mathsf{Bv}(\mathtt{A[I]})+\mathsf{Bv}(\mathtt{B[I]})+\mathsf{Bv}(\mathtt{CARRY}))\ \mathsf{mod}\ 2\big)+\mathsf{V}(\mathtt{SUM[I}-\mathtt{1:0]})\\
&\quad=\ \mathsf{V}(\mathtt{A[I:0]})+\mathsf{V}(\mathtt{B[I:0]})
\end{aligned}$$

Using $\mathsf{V}(w\,[n:0]) \ = \ 2^n \times \mathsf{Bv}(w\,[n]) + \mathsf{V}(w\,[n{-}1:0])$ yields:

$$\begin{aligned}
&2^{\mathtt{I+1}}\times\big((\mathsf{Bv}(\mathtt{A[I]})+\mathsf{Bv}(\mathtt{B[I]})+\mathsf{Bv}(\mathtt{CARRY}))\ \mathsf{div}\ 2\big)\\
&\quad+\ 2^{\mathtt{I}}\times\big((\mathsf{Bv}(\mathtt{A[I]})+\mathsf{Bv}(\mathtt{B[I]})+\mathsf{Bv}(\mathtt{CARRY}))\ \mathsf{mod}\ 2\big)+\mathsf{V}(\mathtt{SUM[I}-\mathtt{1:0]})\\
&\quad=\ 2^{\mathtt{I}}\times\mathsf{Bv}(\mathtt{A[I]})+\mathsf{V}(\mathtt{A[I}-\mathtt{1:0]})+2^{\mathtt{I}}\times\mathsf{Bv}(\mathtt{B[I]})+\mathsf{V}(\mathtt{B[I}-\mathtt{1:0]})
\end{aligned}$$

Now, the antecendent of VC (iv) contains:

$$2^{\mathtt{I}}\times\mathsf{Bv}(\mathtt{CARRY})+\mathsf{V}(\mathtt{SUM[I}-\mathtt{1:0]})=\mathsf{V}(\mathtt{A[I}-\mathtt{1:0]})+\mathsf{V}(\mathtt{B[I}-\mathtt{1:0]})$$

hence the first conjunct of the consequent can be simplified to:

$$\begin{aligned}
&2^{\mathtt{I+1}}\times\big((\mathsf{Bv}(\mathtt{A[I]})+\mathsf{Bv}(\mathtt{B[I]})+\mathsf{Bv}(\mathtt{CARRY}))\ \mathsf{div}\ 2\big)\\
&\quad+\ 2^{\mathtt{I}}\times\big((\mathsf{Bv}(\mathtt{A[I]})+\mathsf{Bv}(\mathtt{B[I]})+\mathsf{Bv}(\mathtt{CARRY}))\ \mathsf{mod}\ 2\big)+\mathsf{V}(\mathtt{SUM[I}-\mathtt{1:0]})\\
&\quad=\ 2^{\mathtt{I}}\times\mathsf{Bv}(\mathtt{CARRY})+\mathsf{V}(\mathtt{SUM[I}-\mathtt{1:0]})+2^{\mathtt{I}}\times\mathsf{Bv}(\mathtt{A[I]})+2^{\mathtt{I}}\times\mathsf{Bv}(\mathtt{B[I]})
\end{aligned}$$

Cancelling $\mathsf{V}(\mathtt{SUM[I}-\mathtt{1:0]})$ and then dividing by $2^{\mathtt{I}}$ yields:

$$\begin{aligned}
&2\times\big((\mathsf{Bv}(\mathtt{A[I]})+\mathsf{Bv}(\mathtt{B[I]})+\mathsf{Bv}(\mathtt{CARRY}))\ \mathsf{div}\ 2\big)\\
&\quad+\ \big((\mathsf{Bv}(\mathtt{A[I]})+\mathsf{Bv}(\mathtt{B[I]})+\mathsf{Bv}(\mathtt{CARRY}))\ \mathsf{mod}\ 2\big)\\
&\quad=\ \mathsf{Bv}(\mathtt{CARRY})+\mathsf{Bv}(\mathtt{A[I]})+\mathsf{Bv}(\mathtt{B[I]})
\end{aligned}$$

This is just an instance of $n \times (m\ \mathsf{div}\ n) + m\ \mathsf{mod}\ n \ = \ m$. Thus VC (iv) is established.

## 1.8 Verification of an add-shift multiplier

The notation $b{\bullet}w$, where $b$ is a bit (truth value) and $w$ a word, denotes $w$ if $b = \mathsf{T}$ and $\mathsf{W}\ |w|\ 0$ if $b = \mathsf{F}$, i.e. $b{\bullet}w \ = \ (b \rightarrow w \mid \mathsf{W}\ |w|\ 0)$. Note that:

$$\mathsf{V}(b{\bullet}w) \ = \ \mathsf{Bv}(b) \times \mathsf{V}(w)$$

The product of two natural numbers represented as $n$-bit words $w_1$ and $w_2$ is a word $w$ such that (writing $m \times n$ as $mn$):

$$\begin{aligned}
\mathsf{V}(w) &= \mathsf{V}(w_1)\mathsf{V}(w_2)\\
&= \big(2^{n-1}\mathsf{Bv}(w_1\,[n{-}1]) + 2^{n-2}\mathsf{Bv}(w_1\,[n{-}2]) + \cdots + 2^0\mathsf{Bv}(w_1\,[0])\big)\mathsf{V}(w_2)\\
&= 2^{n-1}\mathsf{Bv}(w_1\,[n{-}1])\mathsf{V}(w_2) + 2^{n-2}\mathsf{Bv}(w_1\,[n{-}2])\mathsf{V}(w_2) + \cdots + 2^0\mathsf{Bv}(w_1\,[0])\mathsf{V}(w_2)\\
&= 2^{n-1}\mathsf{V}(w_1\,[n{-}1]{\bullet}w_2) + 2^{n-2}\mathsf{V}(w_1\,[n{-}2]{\bullet}w_2) + \cdots + 2^0\mathsf{V}(w_1\,[0]{\bullet}w_2)
\end{aligned}$$

Multiplying by 2 corresponds to shifting one place to the left:

$$\begin{aligned}
2^0\mathsf{V}(w) &= \mathsf{V}(w) \\
2^1\mathsf{V}(w) &= \mathsf{V}(w\frown 0) \\
2^2\mathsf{V}(w) &= \mathsf{V}(w\frown 00) \\
&\vdots \\
2^n\mathsf{V}(w) &= \mathsf{V}(w\frown \underbrace{0\cdots 0}_{n\ 0s})
\end{aligned}$$

Thus the product of $w_1$ and $w_2$ is given by adding:

$$\begin{array}{r}
w_1\,\mathtt{[}0\mathtt{]}\bullet w_2 \\
w_1\,\mathtt{[}1\mathtt{]}\bullet w_2\frown 0 \\
w_1\,\mathtt{[}2\mathtt{]}\bullet w_2\frown 00 \\
w_1\,\mathtt{[}3\mathtt{]}\bullet w_2\frown 000 \\
\vdots \\
w_1\,\mathtt{[}n{-}1\mathtt{]}\bullet w_2\frown 0\cdots 0
\end{array}$$

This is the 'high school' algorithm for doing multiplication described earlier.

The following annotated specification expresses the correctness of this algorithm:

$\{\mathsf{V}(\mathtt{A}) = a \ \wedge\ \mathsf{V}(\mathtt{B}) = b \ \wedge\ \mathtt{PROD} = \mathsf{W}(2\mathtt{N})0 \ \wedge\ |\mathtt{A}| \leq \mathtt{N} \ \wedge\ |\mathtt{B}| \leq \mathtt{N} \ \wedge\ \mathtt{N} > 0\}$
 FOR I:=0 UNTIL N-1 DO $\{2^{\mathtt{I}}\mathsf{V}(\mathtt{A[N-1:I]})b + \mathsf{V}(\mathtt{PROD}) = ab \ \wedge\ \mathsf{V}(\mathtt{B}) = 2^{\mathtt{I}}b\}$
  BEGIN
   PROD := PROD $\uplus$ A[I]$\bullet$B;
   B := B$\frown$0
  END
$\{\mathsf{V}(\mathtt{PROD}) = ab\}$

The VCs for the correctness of the 'high school' multiplication algorithm are:

(i) $(\mathsf{V}(\mathtt{A}) = a \ \wedge\ \mathsf{V}(\mathtt{B}) = b \ \wedge\ \mathtt{PROD} = \mathsf{W}(2\mathtt{N})0 \ \wedge\ |\mathtt{A}| \leq \mathtt{N} \ \wedge\ |\mathtt{B}| \leq \mathtt{N} \ \wedge\ \mathtt{N} > 0)$
$\Rightarrow$
$(2^{\mathtt{0}}\mathsf{V}(\mathtt{A[N-1:0]})b + \mathsf{V}(\mathtt{PROD}) = ab \ \wedge\ \mathsf{V}(\mathtt{B}) = 2^{\mathtt{0}}b)$

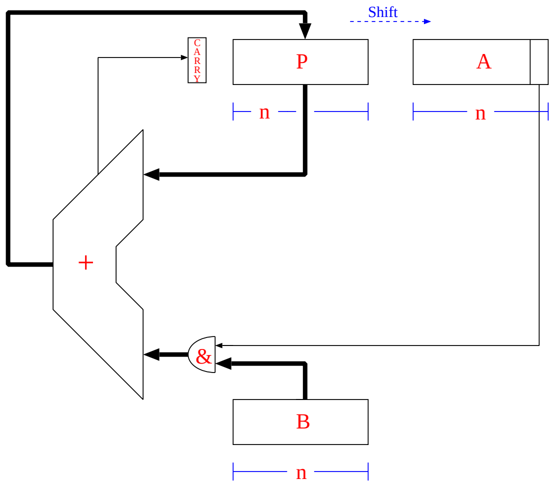
(ii) $(2^{\mathtt{N}}\mathsf{V}(\mathtt{A[N-1:N]})b + \mathsf{V}(\mathtt{PROD}) = ab \ \wedge\ \mathsf{V}(\mathtt{B}) = 2^{\mathtt{N}}b) \Rightarrow (\mathtt{PROD} = ab)$

(iii) $(\mathsf{V}(\mathtt{A}) = a \wedge \mathsf{V}(\mathtt{B}) = b \wedge \mathtt{PROD} = \mathsf{W}(2\mathtt{N})0 \wedge |\mathtt{A}| \leq \mathtt{N} \wedge |\mathtt{B}| \leq \mathtt{N} \wedge \mathtt{N} > 0) \ \wedge\ \mathtt{N}{-}1 < 0$
$\Rightarrow$
$(\mathtt{PROD} = ab)$

(iv) $(2^{\mathtt{I}}\mathsf{V}(\mathtt{A[N-1:I]})b + \mathsf{V}(\mathtt{PROD}) = ab \ \wedge\ \mathsf{V}(\mathtt{B}) = 2^{\mathtt{I}}b) \ \wedge\ 0 \leq \mathtt{I} \ \wedge\ \mathtt{I} \leq \mathtt{N}{-}1$
$\Rightarrow$
$2^{\mathtt{I+1}}\mathsf{V}(\mathtt{A[N-1:I+1]})b + \mathsf{V}(\mathtt{PROD} \uplus \mathtt{A[I]}\bullet\mathtt{B}) = ab \ \wedge\ \mathsf{V}(\mathtt{B}\frown 0) = 2^{\mathtt{I+1}}b$

**Exercise:** Check whether these verification conditions are true.
(**Warning:** quite likely to be wrong!)

Here is a standard textbook multiplier: [2]




In the program below, the product is computed in a single `2N`-bit variable `R`. The `N` high order bits of `R` correspond the register `P` (after `CARRY` has been right-shifted into `P`); the `N` low-order bits of `R` correspond to register `A`.
At the $\mathtt{I}^{\mathrm{th}}$ iteration `R[2N−1 : N−I]` holds the result of adding the first `I` rows of the sum:

$$\begin{array}{r}
w_1\texttt{[0]}\bullet w_2\\
w_1\texttt{[1]}\bullet w_2\frown 0\\
w_1\texttt{[2]}\bullet w_2\frown 00\\
w_1\texttt{[3]}\bullet w_2\frown 000\\
\vdots\\
w_1\texttt{[}n{-}1\texttt{]}\bullet w_2\frown 0\cdots 0
\end{array}$$

and `R[N−I−1 : 0]` holds $w_1$`[N−1 : I]`. Initially $w_1$ is loaded into the bottom half of `R`, but during the computation this is gradually overwritten by the low-order bits of the product. The variable `B` holds $w_2$.
Let

$$\begin{array}{lcl}
\mathcal{I}_1 & \equiv & w_1\texttt{[N}{-}\texttt{1 : I]} = \texttt{R[N}{-}\texttt{1}{-}\texttt{I : 0]}\\
\mathcal{I}_2 & \equiv & \mathsf{V}(\texttt{R[2N}{-}\texttt{1 : N}{-}\texttt{I]}) = \mathsf{V}(w_1\texttt{[I}{-}\texttt{1 : 0]}) \times \mathsf{V}(\texttt{B})
\end{array}$$

Consider:

[2] *Computer Architecture: A Quantitative Approach* by Hennessy and Patterson (ISDBN 1-55860-069-8).

$\{\mathtt{R} = (\mathsf{W}\ \mathtt{N}\ 0)\frown w_1\ \wedge\ \mathtt{B} = w_2\ \wedge\ |w_1| \le \mathtt{N}\ \wedge\ |w_2| \le \mathtt{N}\ \wedge\ \mathtt{N} > 0\}$
  FOR I := 0 UNTIL N-1 DO $\{\mathcal{I}_1\ \wedge\ \mathcal{I}_2\ \wedge\ \mathtt{B} = w_2\}$
   R := (R[0].B $\uplus$ R[2N-1:N])$\frown$R[N-1:1]
$\{\mathsf{V}(\mathtt{R[2N{-}1:0]})\ =\ \mathsf{V}(\mathtt{w_1}) \times \mathsf{V}(\mathtt{w_2})\ \wedge\ \mathtt{B} = \mathtt{w_2}\}$

The verification conditions are:

(i) $(\mathtt{R} = (\mathsf{W}\ \mathtt{N}\ 0)\frown w_1\ \wedge\ \mathtt{B} = w_2\ \wedge\ |w_1| \le \mathtt{N}\ \wedge\ |w_2| \le \mathtt{N}\ \wedge\ \mathtt{N} > 0)$
$\Rightarrow$
$(\mathcal{I}_1\ \wedge\ \mathcal{I}_2\ \wedge\ \mathtt{B} = w_2)[0/\mathtt{I}]$

(ii) $(\mathcal{I}_1\ \wedge\ \mathcal{I}_2\ \wedge\ \mathtt{B} = w_2)[\mathtt{N}/\mathtt{I}]$
$\Rightarrow$
$(\mathsf{V}(\mathtt{R[2N{-}1:0]})\ =\ \mathsf{V}(\mathtt{w_1}) \times \mathsf{V}(\mathtt{w_2})\ \wedge\ \mathtt{B} = \mathtt{w_2})$

(iii) $(\cdots\ \wedge\ \mathtt{N} > 0)\ \wedge\ (\mathtt{N\text{-}1} < 0)\ \Rightarrow\ (\cdots)$

(iv) VCs from
$\{\mathcal{I}_1\ \wedge\ \mathcal{I}_2\ \wedge\ \mathtt{B} = w_2\ \wedge\ 0 \le \mathtt{I}\ \wedge\ \mathtt{I} \le \mathtt{N\text{-}1}\}$
R:=(R[0].B $\uplus$ R[2N-1:N])$\frown$R[N-1:1]
$\{(\mathcal{I}_1\ \wedge\ \mathcal{I}_2\ \wedge\ \mathtt{B} = w_2)\mathtt{[I+1/I]}\}$

To show VC (i) assume
$\mathtt{R} = (\mathsf{W}\ \mathtt{N}\ 0)\frown w_1\ \wedge\ \mathtt{B} = w_2\ \wedge\ |w_1| \le \mathtt{N}\ \wedge\ |w_2| \le \mathtt{N}\ \wedge\ \mathtt{N} > 0$
$\mathcal{I}_1[0/\mathtt{I}]$ is:
$w_1\mathtt{[N{-}1:0]} = ((\mathsf{W}\ \mathtt{N}\ \mathtt{0})\frown w_1\mathtt{[N\text{-}1:0]})\mathtt{[N{-}1:0]}$
which follows from:
$p < |w_2|\ \ \Rightarrow\ \ (w_1\frown w_2)\mathtt{[}p:q\mathtt{]}\ =\ w_2\mathtt{[}p:q\mathtt{]}$
$\mathcal{I}_2[0/\mathtt{I}]$ is:
$\mathsf{V}(((\mathsf{W}\ \mathtt{N}\ \mathtt{0})\frown w_1\mathtt{[N\text{-}1:0]})\mathtt{[2N{-}1:N]}) = \mathsf{V}(w_1\mathtt{[0{-}1:0]}) \times \mathsf{V}(\mathtt{B})$
which is equivalent to $0 = 0$ because of the convention that $\mathsf{V}(\mathtt{A[0\text{-}1:0]})$ is 0, from which it follows that the RHS is 0. The LHS can be simplified using:

$$q \ge |w_2|\ \ \Rightarrow\ \ (w_1\frown w_2)\mathtt{[}p:q\mathtt{]}\ =\ w_1\mathtt{[}p{-}|w_2| : q{-}|w_2|\mathtt{]}$$
$$w\mathtt{[}|w|{-}1:0\mathtt{]}\ =\ w$$
$$\mathsf{V}(\mathsf{W}\ n\ m)\ =\ m\ \mathsf{mod}\ 2^n$$

From this, it follows that:

$$\begin{aligned}\mathsf{V}(((\mathsf{W}\ \mathtt{N}\ \mathtt{0})\frown \mathtt{A[N\text{-}1:0]})\mathtt{[2N{-}1:N]}) &= \mathsf{V}((\mathsf{W}\ \mathtt{N}\ \mathtt{0})\mathtt{[N{-}1:0]})\\ &= \mathsf{V}(\mathsf{W}\ \mathtt{N}\ \mathtt{0})\\ &= \mathtt{0}\end{aligned}$$

To establish the second VC (ii) observe that $\mathcal{I}_2[\mathtt{N}/\mathtt{I}]$ is

$\mathsf{V}(\mathtt{R[2N-1:N-N]}) = \mathsf{V}(w_1\mathtt{[N-1:0]}) \times \mathsf{V}(\mathtt{B})$

so if $|w_1| \leq \mathtt{N}$ and $\mathtt{B} = w_2$ then clearly

$\mathsf{V}(\mathtt{R[2N\text{-}1:0]}) \;=\; \mathsf{V}(w_1) \times \mathsf{V}(w_2)$

The third VC (iii) is trivial (contradictory antecedent of an implication).

The final VC (iv) is the guts of the verification. The invariance of B is obvious. The remaining things to establish are:

**VC4.1**

$$\begin{array}{l}\mathcal{I}_1 \;\wedge\; \mathcal{I}_2 \;\wedge\; \mathtt{B} = w_2 \;\wedge\; \mathtt{0} \leq \mathtt{I} \;\wedge\; \mathtt{I} \leq \mathtt{N} - 1 \\ \Rightarrow \\ \mathtt{w}_1\mathtt{[N-1:I+1]} = ((\mathtt{R[0]}\bullet\mathtt{B} \uplus \mathtt{R[2N-1:N]})^\frown\mathtt{R[N-1:1]})\mathtt{[N-I-2:0]}\end{array}$$

**VC4.2**

$$\begin{array}{l}\mathcal{I}_1 \;\wedge\; \mathcal{I}_2 \;\wedge\; \mathtt{B} = w_2 \;\wedge\; \mathtt{0} \leq \mathtt{I} \;\wedge\; \mathtt{I} \leq \mathtt{N} - 1 \\ \Rightarrow \\ \mathsf{V}(((\mathtt{R[0]}\bullet\mathtt{B} \uplus \mathtt{R[2N-1:N]})^\frown\mathtt{R[N-1:1]})\mathtt{[2N-1:N-I-1]}) = \mathsf{V}(w_1\mathtt{[I:0]}) \times \mathsf{V}(\mathtt{B})\end{array}$$

**VC4.1** follows from:

$$p < |w_2| \;\;\Rightarrow\;\; (w_1{}^\frown w_2)[p:q] \;\;=\;\; w_2[p:q]$$
$$(w[m:n])[p:q] \;\;=\;\; w[p{+}n:q{+}n]$$

Since:

$$\begin{array}{ll}(\mathtt{R[0]}\bullet\mathtt{B} \uplus \mathtt{R[2N-1:N]})^\frown\mathtt{R[N-1:1]})\mathtt{[N-I-2:0]} & = (\mathtt{R[N-1:1]})\mathtt{[N-I-2:0]} \\ & = \mathtt{R[N-I-1:1]} \\ & = \mathtt{R[N-1-I:1]}\end{array}$$

and

$$\begin{array}{ll}\mathcal{I}_1 \;\equiv & w_1\mathtt{[N-1:I]} = \mathtt{R[N-1-I:0]} \\ \;\;\Rightarrow & w_1\mathtt{[N-1:I]}\mathtt{[N-1-I:1]} = \mathtt{R[N-1-I:0]}\mathtt{[N-1-I:1]} \\ \;\;\Rightarrow & w_1\mathtt{[N-1:I+1]} = \mathtt{R[N-1-I:1]}\end{array}$$

To establish **VC4.2** first observe that:

$$\begin{array}{ll}\mathcal{I}_1 \;\equiv & w_1\mathtt{[N-1:I]} = \mathtt{R[N-1-I:0]} \\ \;\;\Rightarrow & w_1\mathtt{[N-1:I]}\mathtt{[0]} = \mathtt{R[N-1-I:0]}\mathtt{[0]} \\ \;\;\Rightarrow & w_1\mathtt{[I]} = \mathtt{R[0]}\end{array}$$

The cases $\mathtt{I} = \mathtt{0}$ and $\mathtt{I} > \mathtt{0}$ need to be considered separately.

If $\mathtt{I} = \mathtt{0}$ then:

$$\begin{array}{ll}\mathcal{I}_2 \;\equiv & \mathsf{V}(\mathtt{R[2N-1:N-I]}) = \mathsf{V}(w_1\mathtt{[I-1:0]}) \times \mathsf{V}(\mathtt{B}) \\ \;\;\Rightarrow & \mathsf{V}(\mathtt{R[2N-1:N]}) = \mathsf{V}(w_1\mathtt{[0-1:0]}) \times \mathsf{V}(\mathtt{B}) \\ \;\;\Rightarrow & \mathsf{V}(\mathtt{R[2N-1:N]}) = \mathtt{0} \times \mathsf{V}(\mathtt{B}) \\ \;\;\Rightarrow & \mathsf{V}(\mathtt{R[2N-1:N]}) = \mathtt{0} \\ \;\;\Rightarrow & \mathtt{R[2N-1:N]} = \mathsf{W}\;\mathtt{N}\;\mathtt{0}\end{array}$$

hence, assuming $\mathcal{I}_1$:

$$
\begin{array}{l}
\mathsf{V}((\texttt{R[0]}\bullet\texttt{B} \uplus \texttt{R[2N}-\texttt{1 : N]})^\frown\texttt{R[N}-\texttt{1 : 1]})\texttt{[2N}-\texttt{1 : N}-\texttt{I}-\texttt{1]}) = \mathsf{V}(w_1\texttt{[I : 0]}) \times \mathsf{V}(\texttt{B}) \\
\Leftrightarrow \\
\mathsf{V}((\texttt{R[0]}\bullet\texttt{B} \uplus \texttt{R[2N}-\texttt{1 : N]})^\frown\texttt{R[N}-\texttt{1 : 1]})\texttt{[2N}-\texttt{1 : N}-\texttt{1]}) = \mathsf{V}(w_1\texttt{[0 : 0]}) \times \mathsf{V}(\texttt{B}) \\
\Leftrightarrow \\
\mathsf{V}((w_1\texttt{[0]}\bullet\texttt{B} \uplus \mathsf{W}\ \texttt{N}\ \texttt{0}))^\frown\texttt{R[N}-\texttt{1 : 1]})\texttt{[2N}-\texttt{1 : N}-\texttt{1]} = \mathsf{Bv}(w_1\texttt{[0]}) \times \mathsf{V}(\texttt{B}) \\
\Leftrightarrow \\
\mathsf{V}(\texttt{0}^\frown(w_1\texttt{[0]}\bullet\texttt{B})^\frown\texttt{R[N}-\texttt{1 : 1]})\texttt{[2N}-\texttt{1 : N}-\texttt{1]}) = \mathsf{Bv}(w_1\texttt{[0]}) \times \mathsf{V}(\texttt{B}) \\
\Leftrightarrow \qquad \big(\text{using: } q \geq |w_2| \;\; \Rightarrow (w_1{}^\frown w_2)[p:q] \;=\; w_1[p-|w_2| : q-|w_2|]\big) \\
\mathsf{V}((\texttt{0}^\frown(w_1\texttt{[0]}\bullet\texttt{B}))\texttt{[N : 0]}) = \mathsf{Bv}(w_1\texttt{[0]}) \times \mathsf{V}(\texttt{B}) \\
\Leftrightarrow \\
\mathsf{V}((w_1\texttt{[0]}\bullet\texttt{B})\texttt{[N}-\texttt{1 : 0]}) = \mathsf{Bv}(w_1\texttt{[0]}) \times \mathsf{V}(\texttt{B}) \\
\Leftrightarrow \qquad \big(\text{Using the assumption that } \texttt{B} \text{ is an } \texttt{N}\text{-bit word}\big) \\
\mathsf{V}(w_1\texttt{[0]}\bullet\texttt{B}) = \mathsf{Bv}(w_1\texttt{[0]}) \times \mathsf{V}(\texttt{B}) \\
\Leftrightarrow \\
\mathsf{Bv}(w_1\texttt{[0]}) \times \mathsf{V}(\texttt{B}) = \mathsf{Bv}(w_1\texttt{[0]}) \times \mathsf{V}(\texttt{B})
\end{array}
$$

If $\texttt{I} > \texttt{0}$ then as $\texttt{N} > \texttt{0}$ it follows that $\texttt{2N}-\texttt{1} \geq \texttt{N}-\texttt{1}$ and $\texttt{N}-\texttt{I}-\texttt{1} < \texttt{N}-\texttt{1}$. Recall:

$$p \geq |w_2| \;\wedge\; q < |w_2| \;\;\Rightarrow (w_1{}^\frown w_2)[p:q] \;=\; w_1[p-|w_2| : 0]^\frown w_2[|w_2|-1 : q]$$

hence, assuming $\mathcal{I}_1$ and $\mathcal{I}_2$:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{V}(((\texttt{R[0]}\bullet\texttt{B} \uplus \texttt{R[2N}-\texttt{1 : N]})^\frown\texttt{R[N}-\texttt{1 : 1]})\texttt{[2N}-\texttt{1 : N}-\texttt{I}-\texttt{1]})} \\
= & \mathsf{V}((\texttt{R[0]}\bullet\texttt{B} \uplus \texttt{R[2N}-\texttt{1 : N]})\texttt{[N : 0]}^\frown\texttt{R[N}-\texttt{1 : 1][N}-\texttt{2 : N}-\texttt{I}-\texttt{1]}) \\
= & \mathsf{V}((\texttt{R[0]}\bullet\texttt{B} \uplus \texttt{R[2N}-\texttt{1 : N]})\texttt{[N : 0]}^\frown\texttt{R[N}-\texttt{1 : N}-\texttt{I]}) \\
= & \mathsf{V}((\texttt{R[0]}\bullet\texttt{B} \uplus \texttt{R[2N}-\texttt{1 : N]})^\frown\texttt{R[N}-\texttt{1 : N}-\texttt{I]}) \\
= & 2^{\texttt{I}}\times\mathsf{V}(\texttt{R[0]}\bullet\texttt{B} \uplus \texttt{R[2N}-\texttt{1 : N]}) \;+\; \mathsf{V}(\texttt{R[N}-\texttt{1 : N}-\texttt{I]}) \\
= & 2^{\texttt{I}}\times\mathsf{V}(\texttt{R[0]}\bullet\texttt{B}) \;+\; \mathsf{V}(\texttt{R[2N}-\texttt{1 : N]}) \;+\; \mathsf{V}(\texttt{R[N}-\texttt{1 : N}-\texttt{I]}) \\
= & 2^{\texttt{I}}\times\mathsf{V}(\texttt{R[0]}\bullet\texttt{B}) \;+\; \big(\mathsf{V}(\texttt{R[2N}-\texttt{1 : N]}) \;+\; \mathsf{V}(\texttt{R[N}-\texttt{1 : N}-\texttt{I]})\big) \\
= & 2^{\texttt{I}}\times\mathsf{V}(\texttt{R[0]}\bullet\texttt{B}) \;+\; \mathsf{V}(\texttt{R[2N}-\texttt{1 : N}-\texttt{I]}) \\
= & 2^{\texttt{I}}\times\mathsf{V}(\texttt{R[0]}\bullet\texttt{B}) \;+\; \mathsf{V}(w_1\texttt{[I}-\texttt{1 : 0]}) \times \mathsf{V}(\texttt{B}) \qquad (\text{by } \mathcal{I}_2) \\
= & 2^{\texttt{I}}\times\mathsf{V}(w_1\texttt{[I]}\bullet\texttt{B}) \;+\; \mathsf{V}(w_1\texttt{[I}-\texttt{1 : 0]}) \times \mathsf{V}(\texttt{B}) \qquad (\text{by } \mathcal{I}_1) \\
= & \big(2^{\texttt{I}}\times\mathsf{Bv}(w_1\texttt{[I]}) \;+\; \mathsf{V}(w_1\texttt{[I}-\texttt{1 : 0]})\big) \times \mathsf{V}(\texttt{B}) \\
= & \mathsf{V}(w_1\texttt{[I : 0]}) \times \mathsf{V}(\texttt{B})
\end{array}
$$

This establishes **VC4.2**.

## 1.9 From programs to hardware

In the previous sections a number of programs that did hardware-like things were verified. However, programs are not hardware. We now consider how programs can be interpreted as hardware.

One possible interpretation is to 'unroll' a program into combinational logic.

Consider the ripple-carry adder, for example.

```
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 `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
```

Assuming initially `CARRY` $=$ `F`, the `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 this (and doing some Boolean algebra simplification) yields:

```
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 now independent assignments that give explicit 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]`.

Clearly this process can be used to get the logic equations for adders of arbitrary bit-widths. The verification done earlier shows that any adder generated this way is correct. [3]

[3]This assumes that the original program is equivalent to the unrolled version. I hope this is intuitively clear. We will not attempt a more formal justification.

This interpretation of commands as hardware is sensible for the adder, but less so for the multipliers we considered. It is straightforward to unroll a multiplier into combinational logic, but the resulting Boolean expressions will be huge. Trying to evaluate them in one clock cycle is likely to make the cycle time too slow. The usual approach is to implement multipliers as sequential machines that compute the product over a number of cycles. For example, one might do the add and shift in a single cycle which would take `N` cycles. Alternatively, one might add and shift on separate cycles, taking `2N`, presumably shorter, cycles. The 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.

The view of a multiplier as computing a single state change from the initial values of the registers to the final values is quite abstract, but adequate for showing pure functional correctness (i.e. it does multiplication). Less abstract views are needed for timing analysis and other kinds of issues.

Modern HDLs like Verilog allow the designer to indicate how operations are scheduled into clock cycles. To illustrate this, we allow statements to be prefixed by `@`, which is called an *event control*. The multiplier that takes `N` cycles is:

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

and a multiplier that takes `2N` cycles is:

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

Note that `SUM` here needs to be `2N+1` bits wide.

In Verilog, for example, event controls can be more detailed, specifying that the subsequent commands are only triggered by particular kinds of events (e.g. rising edges or falling edges). However, at least for the time being, our toy HDL only has an abstract 'clock tick' events `@`.

It is clear that programs with added event controls can still be reasoned about using Floyd Hoare logic, since the transformation from initial to final state is unchanged. The `@`'s merely serve to determine the intermediate states that occur at clock ticks. However, there are properties that one might want to hold but which cannot be expressed using Hoare-style correctness specifications, for example the fact that the variable `B` is not changed during the computation. 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
```

Here `¬B` denotes the negation of every bit of `B`. This program takes `2N` cycles and (as `¬¬B = B`) it has the same overall effect. However, `B` keeps changing throughout the computation and this cannot be expressed using Hoare triples $\{P\}C\{Q\}$. Often, e.g. in connection with handshake protocols, one wants to ensure variables stay constant throughout parts of the computation, and this cannot be verified using Floyd-Hoare logic.

Later in this course we will meet *temporal logic* which enables properties of intermediate states to be specified.

HDL synthesis tools can compile imperative programs to state machines that can be directly implemented in hardware. One can thus verify hardware by verifying HDL code segments using Floyd-Hoare logic prior to synthesis. Of course, this assumes hardware synthesis is correct!

In the next chapter we look at describing hardware structure and behaviour directly in higher order logic. We will look at the ripple-carry adder and add-shift multiplier again and compare their verification as source HDL with their verification as directly modelled hardware.

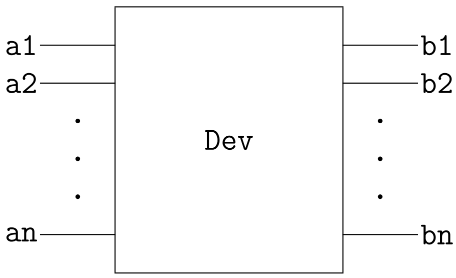
Chapter 2

# Describing Hardware Directly in Higher Order Logic

*It is shown how higher order logic can be used both as a hardware description language and as a formalism for proving that designs meet their specifications. Examples are given which illustrate various specification and verification techniques. These include a CMOS inverter, a CMOS full adder, an $n$-bit ripple-carry adder, a sequential multiplier and an edge-triggered Dtype register.*

## 2.1 Representing behaviour with predicates

A device is a 'black box' with a specified behaviour; for example:




This device is called `Dev` and has external lines `a1`, `a2`, ... , `am`, `b1`, `b2`, ... , `bn`. These lines correspond to the 'pins' of an integrated circuit. When the device is in operation each line has a value drawn from some set of possible values. Different kinds of device are modelled with different sets of values. The behaviour of device `Dev` is specified by defining a predicate Dev (with $m+n$ arguments) such that $\mathsf{Dev}(a_1, a_2, \ldots, a_m, b_1, b_2, \ldots, b_n)$ holds if and only if $a_1$, $a_2$, ... , $a_m$, $b_1$, $b_2$, ... , $b_n$ are allowable values on the corresponding lines of `Dev`.

The following font conventions will be used:

- Physical objects like devices and lines will be written in `typewriter` font.

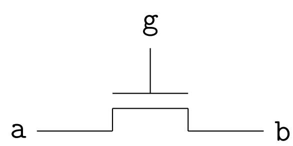
- Mathematical variables will be written in *italic* font.
- Mathematical constants, (e.g. predicate and function constants) will be written in $\mathsf{sans\ serif}$ font.

The same letter will be used for a physical object and its mathematical representation. Thus, for example, $l$ will range over the values allowed at line `l`, and $\mathsf{Dev}$ denotes the predicate describing the behaviour of device `Dev`.

We now describe two examples that illustrate the use of predicates to specify behaviour. In the first of these examples the values on lines are modelled with truth-values. In the second example the values on lines are modelled with functions, and consequently the predicate used to specify the behaviour of the device is higher-order.

### 2.1.1 A delayless switch

Zero-delay combinational devices can be modelled by taking the boolean values $\mathsf{T}$ and $\mathsf{F}$ as the allowed values on their lines. An example is a switch:




The intended behaviour of this is that `a` is connected to `b` if `g` has the value $\mathsf{T}$ and `a` and `b` are not connected if `g` has the value $\mathsf{F}$. This behaviour can be represented by the predicate $\mathsf{Switch}$ defined by:

$$\mathsf{Switch}(g, a, b) \ \equiv \ (g \Rightarrow (a = b))$$

The condition $\mathsf{Switch}(g, a, b)$ holds if and only if whenever $g$ is true then $a$ and $b$ are equal. For example, $\mathsf{Switch}(\mathsf{T}, \mathsf{F}, \mathsf{F})$ holds because $\mathsf{T} \Rightarrow (\mathsf{F}{=}\mathsf{F})$ is true, and $\mathsf{Switch}(\mathsf{F}, \mathsf{T}, \mathsf{F})$ holds because $\mathsf{F} \Rightarrow (\mathsf{T}{=}\mathsf{F})$ is true, but $\mathsf{Switch}(\mathsf{T}, \mathsf{T}, \mathsf{F})$ does not hold because $\mathsf{T} \Rightarrow (\mathsf{T}{=}\mathsf{F})$ is false.

### 2.1.2 An inverter with delay

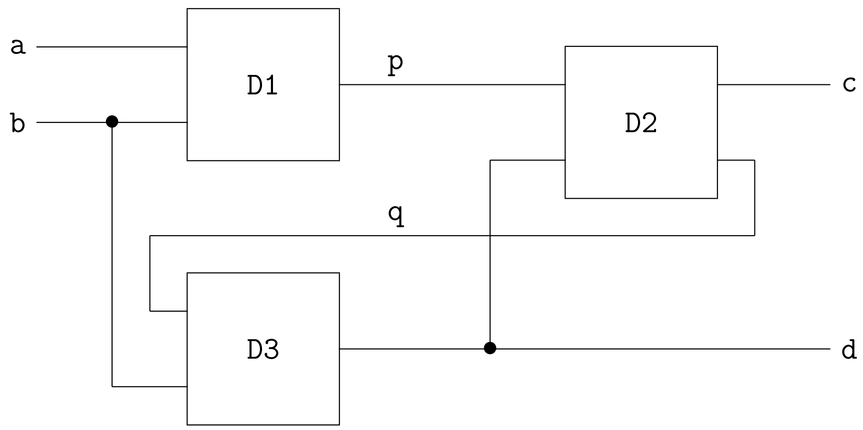
The values occurring on the lines of devices may vary over time. When this happens, their behaviour must be represented by predicates whose arguments are 'time varying values'. Such values correspond to 'waveforms' and can be modelled by functions of time. For example, the behaviour of an inverter with a delay of $\delta$ units of time can be specified with a predicate $\mathsf{Invert}$ defined by:

$$\mathsf{Invert}(i, o) \ \equiv \ \forall t.\ o(t{+}\delta) = \neg i(t)$$

Here the values on lines `i` and `o` are functions $i$ and $o$ which map times (represented by numbers) to values (represented by booleans). These functions are in the $\mathsf{Invert}$ relation if and only if for all times $t$, the value of $o$ at time $t+\delta$ equals the negation of the value of $i$ at time $t$.

## 2.2 Representing circuit structure with predicates

Consider the following structure (called `D`):




This device is built by connecting together three component devices `D1`, `D2` and `D3`. The external lines of `D` are `a`, `b`, `c` and `d`. The lines `p` and `q` are internal and are not connected to the 'outside world'. (External lines might correspond to the pins of an integrated circuit, and internal lines to tracks.)

Suppose the behaviours of `D1`, `D2` and `D3` are specified by predicates $\mathsf{D}_1$, $\mathsf{D}_2$ and $\mathsf{D}_3$ respectively. How can we derive the behaviour of the system `D` shown above? Each device constrains the values on its lines. If $a$, $b$ and $p$ denote the values on the lines `a`, `b` and `p`, then `D1` constrains these values so that $\mathsf{D}_1(a,b,p)$ holds. To get the constraint imposed by the whole device `D` we just conjoin (i.e. $\wedge$-together) the constraints imposed by `D1`, `D2` and `D3`; the combined constraint is thus:

$$\mathsf{D}_1(a,b,p) \ \wedge \ \mathsf{D}_2(p,d,c,q) \ \wedge \ \mathsf{D}_3(q,b,d)$$

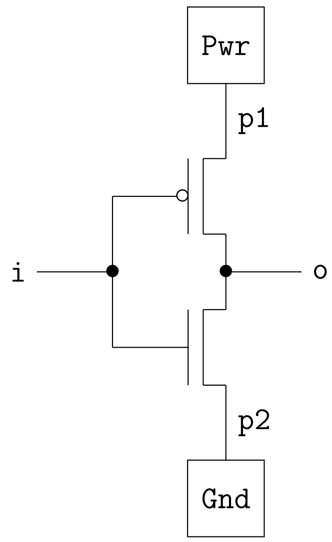
This expression constrains the values on both the external lines `a`, `b`, `c` and `d` and the internal lines `p` and `q`. If we regard `D` as a 'black box' with the internal lines invisible, then we are really only interested in what constraints are imposed on its external lines. The variables $a$, $b$, $c$ and $d$ will denote possible values at the external lines `a`, `b`, `c` and `d` if and only if the conjunction above holds *for some* values $p$ and $q$. We can therefore define a predicate $\mathsf{D}$ representing the behaviour of `D` by:

$$\mathsf{D}(a,b,c,d) \ \equiv \ \exists p\ q.\ \mathsf{D}_1(a,b,p) \ \wedge \ \mathsf{D}_2(p,d,c,q) \ \wedge \ \mathsf{D}_3(q,b,d)$$

Thus we see that the behaviour corresponding to a circuit is got by conjoining the constraints corresponding to the components, and then existentially quantifying the variables corresponding to the internal lines. This technique of representing circuit diagrams in logic is fairly well known. Other ways of representing structure in logic are also possible. There is a nice paper on this by William Clocksin entitled *Logic Programming and the Specification of Circuits* (Computer Laboratory Technical Report No. 72, 1985).

## 2.3 A CMOS inverter

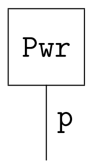
The standard CMOS implementation of an inverter is:




### 2.3.1 Specification of the components

The inverter shown above can be viewed as a structure built out of four components: a power source, a ground, an $n$-transistor and a $p$-transistor.

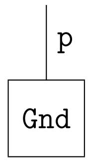
#### 2.3.1.1 Power



This is a power source (sometimes called $\mathsf{V_{dd}}$) and can be modelled by a predicate $\mathsf{Pwr}$ that constrains the value on the line $\mathtt{p}$ to be $\mathsf{T}$.

$$\mathsf{Pwr}(p) \equiv (p = \mathsf{T})$$

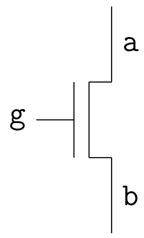
#### 2.3.1.2 Ground




This represents 'ground' and can be modelled by a predicate $\mathsf{Gnd}$ that constrains the value on the line $\mathtt{p}$ to be $\mathsf{F}$.

$$\mathsf{Gnd}(p) \equiv (p = \mathsf{F})$$

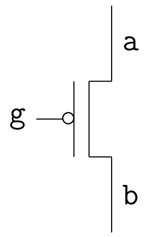
#### 2.3.1.3 *n*-transistor




This represents an $n$-transistor. It can be modelled as a switch.

$$\mathsf{Ntran}(g, a, b) \equiv (g \Rightarrow (a = b))$$

#### 2.3.1.4 *p*-transistor




This represents a $p$-transistor. It can be modelled as a switch which conducts when its gate (i.e. line $\mathtt{g}$) is low.

$$\mathsf{Ptran}(g, a, b) \equiv (\neg g \Rightarrow (a = b))$$

### 2.3.2 Logic representation of the inverter circuit

Conjoining together the constraints from the four components and existentially quantifying the internal line variables yields the following definition of a predicate $\mathsf{Inv}$:

$$\mathsf{Inv}(i, o) \;\equiv\; \exists p_1\ p_2.\ \mathsf{Pwr}(p_1) \wedge \mathsf{Ptran}(i, p_1, o) \wedge \mathsf{Ntran}(i, o, p_2) \wedge \mathsf{Gnd}(p_2)$$

If $\mathsf{Inv}(i, o)$ holds then the values $i$ and $o$ are constrained to be in the relation determined by the inverter circuit above.

### 2.3.3 Verification by proof

It follows by standard logical reasoning that if $\mathsf{Inv}$ is defined as above, then

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

This shows that the constraint on $i$ and $o$ imposed by the inverter circuit is exactly what we want: $o$ is the inverse of $i$.

An outline of the formal proof of this is as follows:

1. By definition of $\mathsf{Inv}$:

$$\begin{aligned}\mathsf{Inv}(i, o) \;\equiv\; \exists p_1\ p_2.\ & \mathsf{Pwr}(p_1) \wedge \\ & \mathsf{Ptran}(i, p_1, o) \wedge \\ & \mathsf{Ntran}(i, o, p_2) \wedge \\ & \mathsf{Gnd}(p_2)\end{aligned}$$

2. Substituting in the definitions of $\mathsf{Pwr}$ and $\mathsf{Gnd}$ yields:

$$\begin{aligned}\mathsf{Inv}(i,o) \ \equiv \ \exists p_1\ p_2.\ & (p_1 = \mathsf{T}) \wedge \\ & \mathsf{Ptran}(i,p_1,o) \wedge \\ & \mathsf{Ntran}(i,o,p_2) \wedge \\ & (p_2 = \mathsf{F})\end{aligned}$$

3. Substituting with the equations $p_1$=$\mathsf{T}$ and $p_2$=$\mathsf{F}$ yields:

$$\begin{aligned}\mathsf{Inv}(i,o) \ \equiv \ \exists p_1\ p_2.\ & (p_1 = \mathsf{T}) \wedge \\ & \mathsf{Ptran}(i,\mathsf{T},o) \wedge \\ & \mathsf{Ntran}(i,o,\mathsf{F}) \wedge \\ & (p_2 = \mathsf{F})\end{aligned}$$

4. In general, if $t_1$ and $t_2$ are any terms such that $t_2$ doesn't contain $x$ then:

$$(\exists x.\ t_1 \ \wedge \ t_2) \ = ((\exists x.\ t_1) \ \wedge \ t_2)$$

and

$$(\exists x.\ t_2 \ \wedge \ t_1) \ = (t_2 \ \wedge \ (\exists x.\ t_1))$$

are both true. Using these properties we can move the existential quantifiers inwards to derive:

$$\begin{aligned}\mathsf{Inv}(i,o) \ \equiv \ & (\exists p_1.\ p_1 = \mathsf{T}) \wedge \\ & \mathsf{Ptran}(i,\mathsf{T},o) \wedge \\ & \mathsf{Ntran}(i,o,\mathsf{F}) \wedge \\ & (\exists p_2.\ p_2 = \mathsf{F})\end{aligned}$$

5. Both $(\exists p_1.\ p_1$=$\mathsf{T})$ and $(\exists p_2.\ p_2$=$\mathsf{F})$ are logical truths and so can be deleted from conjunctions; hence:

$$\mathsf{Inv}(i,o) \ \equiv \ \mathsf{Ptran}(i,\mathsf{T},o) \ \wedge \ \mathsf{Ntran}(i,o,\mathsf{F})$$

6. Next we substitute in the definitions of $\mathsf{Ptran}$ and $\mathsf{Ntran}$ to get:

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

7. From this we can derive

$$\mathsf{Inv}(\mathsf{T},o) \ \equiv \ (o = \mathsf{F})$$

and

$$\mathsf{Inv}(\mathsf{F},o) \ \equiv \ (o = \mathsf{T})$$

from which it follows by case analysis that:

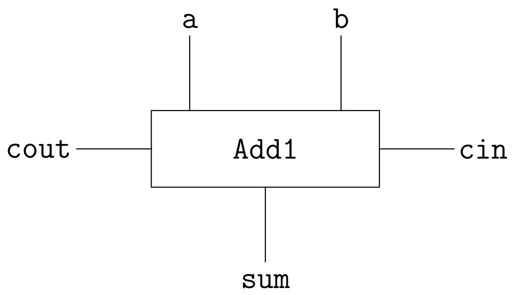
$$\mathsf{Inv}(i,o) \ \equiv \ (o = \neg i)$$

# 2.4 A 1-bit CMOS full adder

The full adder described in this section illustrates the use of bidirectional transistors in CMOS[1]. The transistor models Ptran and Ntran can be used to prove the circuit correct. Such a proof would be difficult with the usual representation of combinational circuits as boolean functions. Relations rather than functions are needed to model bidirectionality.

## 2.4.1 Specification

Here is a diagram of a full adder:




The lines `a`, `b`, `cin`, `sum` and `cout` carry the boolean values T or F. The specification of the adder is:

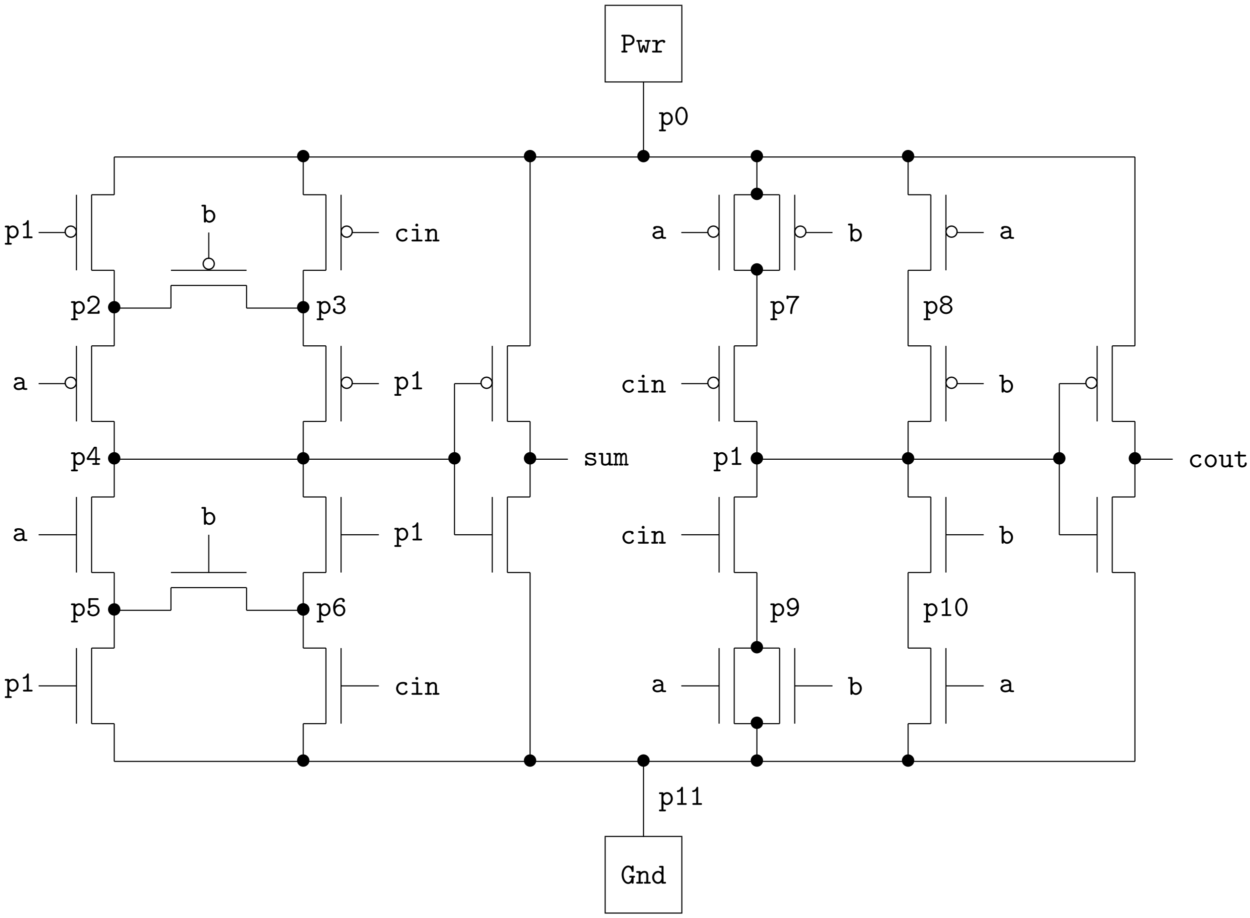
$$\mathsf{Add1}(a, b, cin, sum, cout) \quad \equiv \quad (2\times\mathsf{Bv}(cout) + \mathsf{Bv}(sum) = \mathsf{Bv}(a) + \mathsf{Bv}(b) + \mathsf{Bv}(cin))$$

A correct implementation of this specification is a circuit with lines `a`, `b`, `cin`, `sum` and `cout` such that the constraints imposed on the values $a$, $b$, $cin$, $sum$ and, $cout$ that can exist on these lines imply that $\mathsf{Add1}(a, b, cin, sum, cout)$ always holds.

## 2.4.2 Implementation

A CMOS implementation of the adder is given below. Lines with the same name are connected. The lines `p0`, ... , `p11` are internal. The two transistors drawn horizontally function bidirectionally.

[1] I got the example from Inder Dhingra.




This circuit can be represented in logic by defining:

$$
\begin{array}{l}
\mathsf{Add1\_Imp}(a, b, cin, sum, cout) \;\equiv \\
\exists p_0\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6\ p_7\ p_8\ p_9\ p_{10}\ p_{11}. \\
\quad \mathsf{Ptran}(p_1, p_0, p_2) \wedge \mathsf{Ptran}(cin, p_0, p_3) \wedge \mathsf{Ptran}(b, p_2, p_3) \wedge \mathsf{Ptran}(a, p_2, p_4) \wedge \\
\quad \mathsf{Ptran}(p_1, p_3, p_4) \wedge \mathsf{Ntran}(a, p_4, p_5) \wedge \mathsf{Ntran}(p_1, p_4, p_6) \wedge \mathsf{Ntran}(b, p_5, p_6) \wedge \\
\quad \mathsf{Ntran}(p_1, p_5, p_{11}) \wedge \mathsf{Ntran}(cin, p_6, p_{11}) \wedge \mathsf{Ptran}(a, p_0, p_7) \wedge \mathsf{Ptran}(b, p_0, p_7) \wedge \\
\quad \mathsf{Ptran}(a, p_0, p_8) \wedge \mathsf{Ptran}(cin, p_7, p_1) \wedge \mathsf{Ptran}(b, p_8, p_1) \wedge \mathsf{Ntran}(cin, p_1, p_9) \wedge \\
\quad \mathsf{Ntran}(b, p_1, p_{10}) \wedge \mathsf{Ntran}(a, p_9, p_{11}) \wedge \mathsf{Ntran}(b, p_9, p_{11}) \wedge \mathsf{Ntran}(a, p_{10}, p_{11}) \wedge \\
\quad \mathsf{Pwr}(p_0) \wedge \mathsf{Ptran}(p_4, p_0, sum) \wedge \mathsf{Ntran}(p_4, sum, p_{11}) \wedge \\
\quad \mathsf{Gnd}(p_{11}) \wedge \mathsf{Ptran}(p_1, p_0, cout) \wedge \mathsf{Ntran}(p_1, cout, p_{11})
\end{array}
$$

### 2.4.3 Verification

To verify that the implementation Add1_Imp correctly implements the specification Add1, it can be shown that:

$$\mathsf{Add1\_Imp}(a, b, cin, sum, cout) \;\equiv\; \mathsf{Add1}(a, b, cin, sum, cout)$$

The most straightforward way to prove this implication is to consider separately the eight possible input combinations.

Manipulations like those given for the inverter yield the following eight facts:

$$\begin{array}{lcl}
\mathsf{Add1_Imp}(\mathsf{T},\mathsf{T},\mathsf{T},sum,cout) & \equiv & (sum{=}\mathsf{T}) \ \wedge \ (cout{=}\mathsf{T}) \\
\mathsf{Add1_Imp}(\mathsf{T},\mathsf{T},\mathsf{F},sum,cout) & \equiv & (sum{=}\mathsf{F}) \ \wedge \ (cout{=}\mathsf{T}) \\
\mathsf{Add1_Imp}(\mathsf{T},\mathsf{F},\mathsf{T},sum,cout) & \equiv & (sum{=}\mathsf{F}) \ \wedge \ (cout{=}\mathsf{T}) \\
\mathsf{Add1_Imp}(\mathsf{T},\mathsf{F},\mathsf{F},sum,cout) & \equiv & (sum{=}\mathsf{T}) \ \wedge \ (cout{=}\mathsf{F}) \\
\mathsf{Add1_Imp}(\mathsf{F},\mathsf{T},\mathsf{T},sum,cout) & \equiv & (sum{=}\mathsf{F}) \ \wedge \ (cout{=}\mathsf{T}) \\
\mathsf{Add1_Imp}(\mathsf{F},\mathsf{T},\mathsf{F},sum,cout) & \equiv & (sum{=}\mathsf{T}) \ \wedge \ (cout{=}\mathsf{F}) \\
\mathsf{Add1_Imp}(\mathsf{F},\mathsf{F},\mathsf{T},sum,cout) & \equiv & (sum{=}\mathsf{T}) \ \wedge \ (cout{=}\mathsf{F}) \\
\mathsf{Add1_Imp}(\mathsf{F},\mathsf{F},\mathsf{F},sum,cout) & \equiv & (sum{=}\mathsf{F}) \ \wedge \ (cout{=}\mathsf{F})
\end{array}$$

Deriving these equations is equivalent to exhaustive simulation for all input values and is best done by computer. It follows from these eight equations that:

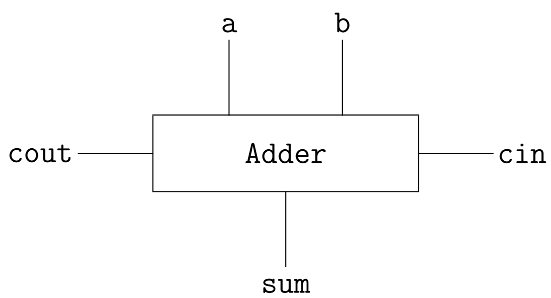
$$\mathsf{Add1\_Imp}(a,b,cin,sum,cout) \ \equiv \ \mathsf{Add1}(a,b,cin,sum,cout)$$

## 2.5 An $n$-bit adder

The example in this section is intended to illustrate the use of higher-order logic to represent parameterized systems. An $n$-bit adder computes an $n$-bit sum and 1-bit carry-out from two $n$-bit inputs and a 1-bit carry-in.

### 2.5.1 Specification

Here is a diagram of an $n$-bit adder:



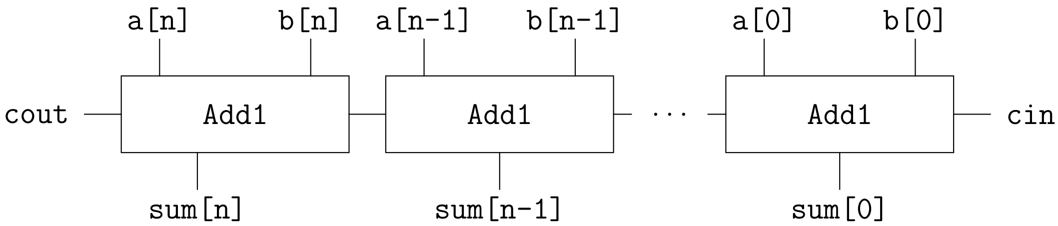

The lines `cin` and `cout` carry single bits, i.e. Booleans, and the lines `a`, `b` and `sum` carry $n$-bit words.

A function Adder is defined which when applied to the number $n$ yields a predicate specifying an $n$+1-bit adder. Thus, for example, Adder(3) is a predicate specifying a 4-bit adder. The definition of Adder is:

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

### 2.5.2 Implementation

An $n$-bit adder can be built by connecting together $n$ full adders. The diagram below shows an $(n{+}1)$-bit adder. The inputs are a single bit carry-in `cin` and two $(n{+}1)$-bit words `a` and `b`. The outputs are an $(n{+}1)$-bit sum `sum` and a 1-bit carry-out `cout`.




To express this diagram in logic we define $\mathsf{Adder\_Imp}(n)(a,b,cin,out,cout)$ which uses a higher order function $\mathsf{Adder\_Imp}$ which when applied to a number $n$ yields the predicate specifying the implementation of an $n$+1-bit adder.

A primitive-recursive definition of $\mathsf{Adder\_Imp}$ corresponding to the above diagram has the following basis:

$$\mathsf{Adder\_Imp}(0)(a,b,cin,sum,cout)\ \equiv\ \mathsf{Add1}(a\texttt{[0]},b\texttt{[0]},cin,sum\texttt{[0]},cout)$$

The recursive part of the definition says that an $(n{+}2)$-bit adder is built by first building an $n$+1-bit adder and then connecting its carry-out to the carry-in of a 1-bit adder.

$$\begin{aligned}&\mathsf{Adder\_Imp}(n{+}1)(a,b,cin,sum,cout)\ \equiv\\ &\quad\exists c.\ \mathsf{Adder\_Imp}(n)(a,b,cin,sum,c)\ \wedge\\ &\qquad\mathsf{Add1}(a\texttt{[}n{+}1\texttt{]},b\texttt{[}n{+}1\texttt{]},c,sum\texttt{[}n{+}1\texttt{]},cout)\end{aligned}$$

To verify the adder one proves by induction on $n$ that:

$$\mathsf{Adder\_Imp}(n)(a,b,cin,sum,cout)\Rightarrow\mathsf{Adder}(n)(a,b,cin,sum,cout)$$

The basis of the induction is:

$$\mathsf{Adder\_Imp}(0)(a,b,cin,sum,cout)\Rightarrow\mathsf{Adder}(0)(a,b,cin,sum,cout)$$

This is proved by substituting the definitions of $\mathsf{Adder\_Imp}$ and $\mathsf{Adder}$ into the above implication and showing the result is true. This is easy using $\mathsf{V}(w\texttt{[}0:0\texttt{]})=\mathsf{Bv}(w\texttt{[0]})$. The induction step is:

$$\begin{aligned}&(\mathsf{Adder\_Imp}(n)(a,b,cin,sum,cout)\Rightarrow\mathsf{Adder}(n)(a,b,cin,sum,cout))\\ &\Rightarrow\\ &(\mathsf{Adder\_Imp}(n{+}1)(a,b,cin,sum,cout)\Rightarrow\mathsf{Adder}(n{+}1)(a,b,cin,sum,cout))\end{aligned}$$

This can be proved by simple arithmetical reasoning. Assume:

$$(\mathsf{Adder\_Imp}(n)(a, b, cin, sum, cout) \Rightarrow \mathsf{Adder}(n)(a, b, cin, sum, cout))$$

Then

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

If $(A = B) \wedge (C = D)$ then it follows that $(A + 2^{n+1}C) = (B + 2^{n+1}D)$. Hence:

$$\begin{array}{l}
\qquad \exists c.\ (2^{n+1}\mathsf{Bv}(c) + \mathsf{V}(sum\,[n:0]\,) = \mathsf{V}(a\,[n:0]\,) + \mathsf{V}(b\,[n:0]\,) + \mathsf{Bv}(cin)) \\
\qquad\qquad \wedge \\
\qquad\qquad (2\mathsf{Bv}(cout) + \mathsf{Bv}(sum\,[n{+}1]\,) = \mathsf{Bv}(a\,[n{+}1]\,) + \mathsf{Bv}(b\,[n{+}1]\,) + \mathsf{Bv}(c)) \\
\quad \Rightarrow \exists c.\ (2^{n+1}\mathsf{Bv}(c) + \mathsf{V}(sum\,[n:0]\,) \\
\qquad\qquad\quad + 2^{n+1}2\mathsf{Bv}(cout) + 2^{n+1}\mathsf{Bv}(sum\,[n{+}1]\,) \\
\qquad\qquad = \mathsf{V}(a\,[n:0]\,) + \mathsf{V}(b\,[n:0]\,) + \mathsf{Bv}(cin) \\
\qquad\qquad\quad + 2^{n+1}\mathsf{Bv}(a\,[n{+}1]\,) + 2^{n+1}\mathsf{Bv}(b\,[n{+}1]\,) + 2^{n+1}\mathsf{Bv}(c)) \\
\quad = \ \exists c.\ (\mathsf{V}(sum\,[n{+}1:0]\,) + 2^{n+2}\mathsf{Bv}(cout) \\
\qquad\qquad = \mathsf{V}(a\,[n{+}1:0]\,) + \mathsf{V}(b\,[n{+}1:0]\,) + \mathsf{Bv}(cin)) \\
\quad = \ (\mathsf{V}(sum\,[n{+}1:0]\,) + 2^{n+2}\mathsf{Bv}(cout) \\
\qquad\ = \mathsf{V}(a\,[n{+}1:0]\,) + \mathsf{V}(b\,[n{+}1:0]\,) + \mathsf{Bv}(cin)) \\
\quad = \ \mathsf{Adder}(n{+}1)(a, b, cin, sum, cout))
\end{array}$$

## 2.6 Sequential Devices

All of the examples so far have been combinational; i.e. the values on the outputs have only depended (via combinational logic) on the current input values, not on values stored in registers (which might be input values latched at previous clock cycles). Sequential devices can be modelled by taking the values on lines to be sequences of values (modelled by functions of time, where time measures number of elapsed clock cycles). For example, a unit-delay element `Del`, with input line `i` and output line `o`, is modelled by specifying that the value output at time $t{+}1$ (i.e. on cycle $t{+}1$) equals the value input at time $t$ (i.e. on cycle $t$). This is expressed in higher-order logic by:

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

Combinational devices can be modelled as sequential devices having no delay.[2] To illustrate this, recall the specification of the adder:

$$
\begin{array}{l}
\mathsf{Adder}(n)(a, b, cin, sum, cout) \;\equiv \\
\quad (2^{n+1} \times \mathsf{Bv}(cout) + \mathsf{V}(sum[n:0]) = \;\; \mathsf{V}(a[n:0]) + \mathsf{V}(b[n:0]) + \mathsf{Bv}(cin))
\end{array}
$$

Here the variables $a$, $b$ and $sum$ range over words and the variables $cin$ and $cout$ range over bits (Booleans). To model the adder as a zero-delay sequential device we must represent its behaviour with a predicate whose arguments are functions of time. In the definition below, the variables $a$, $b$ and $sum$ range over functions from time to words, and the variables $cin$ and $cout$ range over functions from time to bits. Thus, for example, $out(7)[5]$ is bit 5 of the word output at time 7.

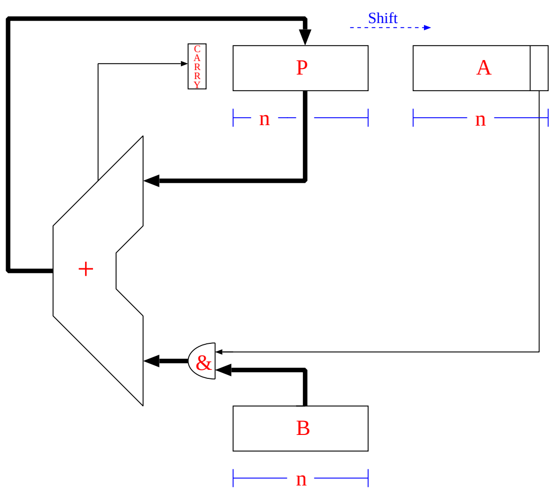
$$
\begin{array}{l}
\mathsf{Combinational\_Adder}(n)(a, b, cin, sum, cout) \;\equiv \\
\quad \forall t.\ \mathsf{Adder}(n)(a(t), b(t), cin(t), sum(t), cout(t))
\end{array}
$$

If we wanted to specify the adder as having a unit-delay then we could define:

$$
\begin{array}{l}
\mathsf{Unit\_Delay\_Adder}(n)(a, b, cin, sum, cout) \;\equiv \\
\quad \forall t.\ \mathsf{Adder}(n)(a(t), b(t), cin(t), sum(t{+}1), cout(t{+}1))
\end{array}
$$

## 2.7 The add-shift multiplier

Recall the add-shift multiplier:




As in Section 1.8 we will model `CARRY`, `P` and `A` as a single register device PA. We assume PA has an $n{+}1$-bit input $in$ and a $2n$-bit outputs $r$. On each cycle PA first shifts the $n$

[2] In reality all devices have some physical delay. Modelling combinational devices with 'zero-delay' is an abstraction that only works if the physical delays of all combinational logic is small compared to the time between clock ticks.

least significant bits of $r$ to the right (losing the rightmost bit) and then reads in $n{+}1$ bits into the $n{+}1$ most significant bits of $r$.

For simplicity, we ignore (fudge) the details of the loading in of the values to be multiplied, $w_1$ and $w_2$ say.

The atomic components of the multiplier are.

1. The $2n$-bit autoshifting register $\mathsf{PA}$, which we parameterise on $n$ and the initial value $w$ of its $n$ least significant bits.

$$\begin{array}{l}\mathsf{PA}(n,w)(in,r) \;\equiv\\ \quad (r(0) \;=\; (\mathsf{W}\; n\; 0)^\frown w\,\texttt{[}n{-}1:0\texttt{]}\,) \wedge\\ \quad (\forall t.\; r(t{+}1) \;=\; in(t) \frown (r(t)\,\texttt{[}n{-}1:1\texttt{]}\,))\end{array}$$

2. A component $\mathsf{Seg}$ to select a subword of a word..

$$\mathsf{Seg}(m,n)(in,out) \;\equiv\; \forall t.\; out(t) \;=\; in(t)\,\texttt{[}m:n\texttt{]}$$

3. A component $\mathsf{Lsb}$ to select the least significant bit of a word.

$$\mathsf{Lsb}(in,out) \;\equiv\; \forall t.\; out(t) \;=\; in(t)\,\texttt{[0]}$$

4. A combinational $n$-bit adder.

$$\mathsf{Add}(n)(in_1,in_2,sum) \;\equiv\; \forall t.\; sum(t) \;=\; in_1(t) \uplus in_2(t)$$

Note that instead of the separate sum and carry-out shown in the diagram, we have a single $n{+}1$-bit output $sum$.

5. A constant register $\mathsf{B}$ parameterised on the value it stores.

$$\mathsf{B}(w)(b) \;\equiv\; \forall t.\; b(t) \;=\; w$$

6. A combinational device $\mathsf{And}$ that has a 1-bit input $bin$ and an $n$-bit input $win$ and outputs the word obtained by $\wedge$-ing together $bin$ with each bit of $win$.

$$\mathsf{And}(bin,win,out) \;\equiv\; \forall t.\; out(t) \;=\; bin(t)_\bullet win(t)$$

With these components, the multiplier is represented by:

$$\begin{array}{l}\mathsf{AddShift}(n,w_1,w_2)(r) \;\equiv\\ \quad \exists sum\; a_0\; b\; p\; q.\\ \qquad \mathsf{PA}(n,w_1)(sum,r) \;\wedge\; \mathsf{Seg}(2n{-}1,n)(r,p) \;\wedge\; \mathsf{Lsb}(r,a_0) \wedge\\ \qquad \mathsf{Add}(n)(p,q,sum) \;\wedge\; \mathsf{B}(w_2)(b) \;\wedge\; \mathsf{And}(a_0,b,q)\end{array}$$

Previously, we specified the multiplier by the Hoare triple of the form:

$$\{\mathtt{R} = (\mathsf{W}\ \mathtt{N}\ 0)\frown w_1 \ \wedge\ \mathtt{B} = w_2 \ \wedge\ |w_1| \leq \mathtt{N} \ \wedge\ |w_2| \leq \mathtt{N} \ \wedge\ \mathtt{N>0}\}$$
$$\vdots$$
$$\{\mathsf{V}(\mathtt{R[2N-1:0]}) = \mathsf{V}(w_1)\times\mathsf{V}(w_2)\}$$

A corresponding specification directly in higher order logic is:

$$\mathsf{AddShift}(n, w_1, w_2)(r) \ \wedge\ |w_1| \leq n \ \wedge\ |w_2| \leq n \ \Rightarrow\ \mathsf{V}(r(n)) = \mathsf{V}(w_1)\times\mathsf{V}(w_2)$$

To prove this we show by induction on $t$ that:

$$\begin{array}{l}
\mathsf{AddShift}(n, w_1, w_2)(r) \ \wedge\ |w_1| \leq n \ \wedge\ |w_2| \leq n \\
\Rightarrow \\
\forall t.\ t \leq n \ \Rightarrow\ \begin{array}[t]{l} |r(t)| \leq 2n \\ \wedge \\ w_1\,[n{-}1 : t] = r(t)\,[n{-}1{-}t : 0] \\ \wedge \\ \mathsf{V}(r(t)\,[2n{-}1 : n{-}t]) = \mathsf{V}(w_1\,[t{-}1 : 0])\times\mathsf{V}(w_2) \end{array}
\end{array}$$

This clearly entails the desired specification (take $t = n$). The first step is to expand the definitions of the components of $\mathsf{AddShift}(n, w_1, w_2)(r)$ and then unwind the resulting equations.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{AddShift}(n, w_1, w_2)(r)} \\
= & \exists sum\ a_0\ b\ p\ q. \\
& \quad \mathsf{PA}(n, w_1)(sum, r) \ \wedge \\
& \quad \mathsf{Seg}(2n{-}1, n)(r, p) \ \wedge \\
& \quad \mathsf{Lsb}(r, a_0) \wedge \\
& \quad \mathsf{Add}(n)(p, q, sum) \ \wedge \\
& \quad \mathsf{B}(w_2)(b) \ \wedge \\
& \quad \mathsf{And}(a_0, b, q) \\
= & \exists sum\ a_0\ b\ p\ q. \\
& \quad (r(0) \ = \ (\mathsf{W}\ n\ 0)\frown w_1\,[n{-}1 : 0]) \wedge \\
& \quad (\forall t.\ r(t{+}1) \ = \ sum(t) \frown (r(t)\,[n{-}1 : 1])) \ \wedge \\
& \quad (\forall t.\ p(t) \ = \ r(t)\,[2n{-}1 : n]) \ \wedge \\
& \quad (\forall t.\ a_0(t) \ = \ r(t)\,[0]) \\
& \quad (\forall t.\ sum(t) \ = \ p(t) \uplus q(t)) \ \wedge \\
& \quad (\forall t.\ b(t) \ = \ w_2) \ \wedge \\
& \quad (\forall t.\ q(t) \ = \ a_0(t)\bullet b(t)) \\
= & \quad (r(0) \ = \ (\mathsf{W}\ n\ 0)\frown w_1\,[n{-}1 : 0]) \wedge \\
& \quad (\forall t.\ r(t{+}1) \ = \ (r(t)\,[2n{-}1 : n] \uplus\ r(t)\,[0]\bullet w_2) \frown (r(t)\,[n{-}1 : 1])) \\
= & \quad (r(0) \ = \ (\mathsf{W}\ n\ 0)\frown w_1) \wedge \\
& \quad (\forall t.\ r(t{+}1) \ = \ (r(t)\,[0]\bullet w_2 \uplus r(t)\,[2n{-}1 : n]) \frown (r(t)\,[n{-}1 : 1]))
\end{array}$$

Let

$$\begin{array}{lll} P(t) \;=\; t \leq n \;\Rightarrow & |r(t)| \leq 2n \\ & \wedge \\ & w_1\,[n{-}1 : t] = r(t)\,[n{-}1{-}t : 0] \\ & \wedge \\ & \mathsf{V}(r(t)\,[2n{-}1 : n{-}t]\,) = \mathsf{V}(w_1\,[t{-}1 : 0]\,) {\times} \mathsf{V}(w_2) \end{array}$$

then we need to show :

$$\mathsf{AddShift}(n, w_1, w_2)(r) \;\wedge\; |w_1| \leq n \;\wedge\; |w_2| \leq n \;\Rightarrow\; \forall t.\ P(t)$$

Assume $\mathsf{AddShift}(n, w_1, w_2)(r) \;\wedge\; |w_1| \leq n \;\wedge\; |w_2| \leq n$, i.e.:

$$\begin{array}{l} (r(0) \;=\; (\mathsf{W}\ n\ 0) {\frown} w_1\,[n{-}1 : 0]\,) \wedge \\ (\forall t.\ r(t{+}1) \;=\; (r(t)\,[0]\,{\bullet} w_2 \uplus r(t)\,[2n{-}1 : n]\,) \frown (r(t)\,[n{-}1 : 1]\,)) \wedge \\ |w_1| \leq n \;\wedge\; |w_2| \leq n \end{array}$$

We show $\forall t.\ P(t)$ by induction on $t$.

**Basis:** $P(0)$
Clearly $0 \leq n$. Need to show:

$$\begin{array}{l} |r(0)| \leq 2n \\ \wedge \\ w_1\,[n{-}1 : 0] = r(0)\,[n{-}1 : 0] \\ \wedge \\ \mathsf{V}(r(0)\,[2n{-}1 : n]\,) = \mathsf{V}(w_1\,[-1 : 0]\,) {\times} \mathsf{V}(w_2) \end{array}$$

The first conjunct follows from $r(0) \;=\; (\mathsf{W}\ n\ 0){\frown} w_1\,[n{-}1 : 0]$. The second conjunct follows from this also. The third conjunct is $0 = 0$, because $r(0) \;=\; (\mathsf{W}\ n\ 0){\frown} w_1\,[n{-}1 : 0]$.

**Step:** $P(t) \;\vdash\; P(t{+}1)$
Note that $t{+}1 \leq n \;\Rightarrow\; t \leq n$. Assume A1–A6, where:

A1: $(\forall t.\ r(t{+}1) \;=\; (r(t)\,[0]\,{\bullet} w_2 \uplus r(t)\,[2n{-}1 : n]\,) \frown (r(t)\,[n{-}1 : 1]\,))$

A2: $|w_1| \leq n$

A3: $|w_2| \leq n$

A4: $t{+}1 \leq n$

A5: $w_1\,[n{-}1 : t] = r(t)\,[n{-}1{-}t : 0]$

A6: $\mathsf{V}(r(t)\,[2n{-}1 : n{-}t]\,) = \mathsf{V}(w_1\,[t{-}1 : 0]\,) {\times} \mathsf{V}(w_2)$

Must show:

$$\begin{array}{l} w_1\,[n{-}1 : t{+}1] \;=\; r(t{+}1)\,[n{-}1{-}(t{+}1) : 0] \\ \wedge \\ \mathsf{V}(r(t{+}1)\,[2n{-}1 : n{-}(t{+}1)]\,) \;=\; \mathsf{V}(w_1\,[(t{+}1){-}1 : 0]\,) {\times} \mathsf{V}(w_2) \end{array}$$

i.e.:

$$\begin{array}{l} w_1\,[n{-}1 : t{+}1] \;=\; r(t{+}1)\,[n{-}t{-}2 : 0] \\ \wedge \\ \mathsf{V}(r(t{+}1)\,[2n{-}1 : n{-}t{-}1]\,) \;=\; \mathsf{V}(w_1\,[t : 0]\,)\times\mathsf{V}(w_2) \end{array}$$

i.e. by A1:

$$\begin{array}{l} w_1\,[n{-}1 : t{+}1] \;=\; ((r(t)\,[0]\,{\bullet}w_2 \uplus r(t)\,[2n{-}1 : n]\,) \frown (r(t)\,[n{-}1 : 1]\,))\,[n{-}t{-}2 : 0] \\ \wedge \\ \mathsf{V}(((r(t)\,[0]\,{\bullet}w_2 \uplus r(t)\,[2n{-}1 : n]\,) \frown (r(t)\,[n{-}1 : 1]\,))\,[2n{-}1 : n{-}t{-}1]\,) \\ \quad = \; \mathsf{V}(w_1\,[t : 0]\,)\times\mathsf{V}(w_2) \end{array}$$

Recall VC4.1 and VC4.2 on page 18: take `R`$=r(t)$ and `I`$=t$, then the two conjuncts above follow from the same reasoning used to show VC4.1 and VC4,2, respectively. Thus the proof by induction on $t$ is essentially the same as the proof using verification conditions!

## 2.8 Another multiplier

The multiplier that follows is not a realistic hardware implementation, but is designed to further illustrate the use of higher order logic.

**Exercise**
Devise a proof rule for a command

`REPEAT` *command* `UNTIL` *statement*

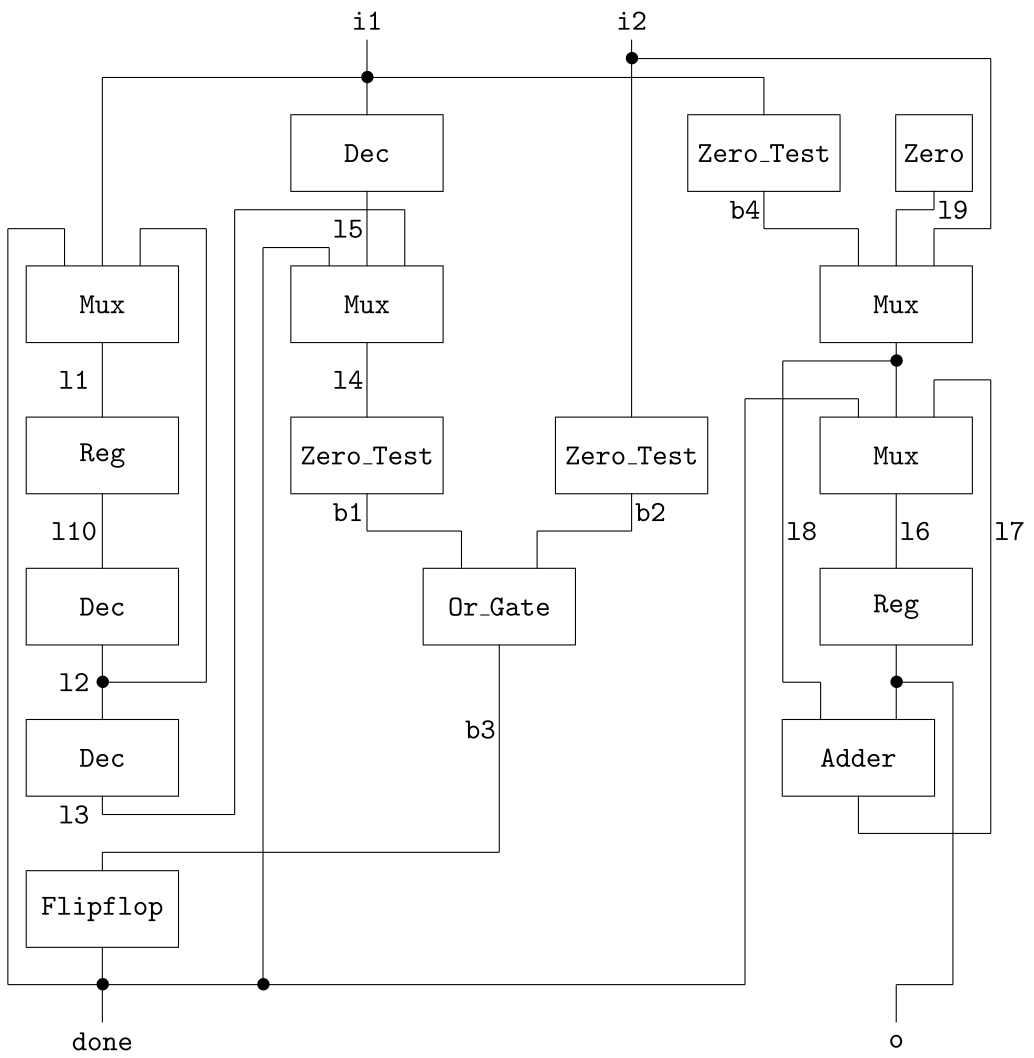
The meaning of `REPEAT C UNTIL S` is that `C` is executed and then `S` is tested; if the result is true, then nothing more is done, otherwise the whole `REPEAT` command is repeated. Thus `REPEAT C UNTIL S` is equivalent to `C; WHILE ¬S DO C`.

Assume that variables `I1`, `I2`, `O1` and `O2` range over natural numbers (i.e. integers $\geq 0$) and `DONE` is a Boolean. Assume also that if $m \leq n$ then $m{-}n = 0$ (this ensures that if $m \geq 0$ and $n \geq 0$ then $m{-}n \geq 0$). Verify the following partial correctness specification:

```
{DONE = T}
 REPEAT
  BEGIN
   IF DONE THEN O1 := I1 ELSE O1 := O1-1;
   IF I1=0 THEN O2 := 0 ELSE (IF DONE THEN O2 := I2 ELSE O2 := I2+O2);
   IF DONE THEN DONE := (I1-1=0 ∨ I2=0) ELSE DONE := (O1-2=0 ∨ I2=0)
  END
 UNTIL DONE
{O2 = I1×I2}
```

**End of exercise**

This program corresponds to the algorithm embodied in the following multiplier circuit:




where the various components are defined by:

$\mathsf{Mux}(ctl, i_1, i_2, o) \;\equiv\; \forall t.\ o(t) = (ctl(t) \rightarrow i_1(t) \mid i_2(t))$
$\mathsf{Reg}(i, o) \;\equiv\; \forall t.\ o(t{+}1) = i(t)$
$\mathsf{Flipflop}(i, o) \;\equiv\; \forall t.\ o(t{+}1) = i(t)$
$\mathsf{Dec}(i, o) \;\equiv\; \forall t.\ o(t) = i(t){-}1$
$\mathsf{Add}(i_1, i_2, o) \;\equiv\; \forall t.\ o(t) = i_1(t) + i_2(t)$
$\mathsf{Zero\_Test}(i, o) \;\equiv\; \forall t.\ o(t) = (i(t){=}0)$
$\mathsf{Or\_Gate}(i_1, i_2, o) \;\equiv\; \forall t.\ o(t) = i_1(t) \vee i_2(t)$
$\mathsf{Zero}(o) \;\equiv\; \forall t.\ o(t) = 0$

However, there is more that can be said about this circuit than that it just does multiplication. Specifically:

**If**

- `done` has value $\mathsf{T}$ at time $t_1$, and
- $t_2$ is the first time after $t_1$ that `done` again has value $\mathsf{T}$, and
- the values at `i1` and `i2` are stable from $t_1$ to $t_2$,

**then**

- the value at `o` at $t_2$ is the product of the values at `i1` and `i2` at $t_1$.

In order to formalize this in logic various temporal notions like "the first time after" and "stable" must be represented.

### 2.8.1 Some temporal predicates

The predicate Stable is defined so that $\mathsf{Stable}(t_1, t_2)(f)$ is true if and only if the value of $f$ is constant from $t_1$ until just before time $t_2$. Formally:

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

The predicate Next is defined so that $\mathsf{Next}(t_1, t_2)(f)$ is true if and only if $t_2$ is the first time after $t_1$ that $f(t_2){=}\mathsf{T}$. Formally:

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

Using Stable and Next, the specification of `Mult` can be represented with the predicate Mult defined by:

$$\begin{array}{l}
\mathsf{Mult}(i_1, i_2, o, done) \;\equiv \\
\;done(t_1) \wedge \\
\;\mathsf{Next}(t_1, t_2)(done) \wedge \\
\;\mathsf{Stable}(t_1, t_2)(i_1) \wedge \\
\;\mathsf{Stable}(t_1, t_2)(i_2) \wedge \\
\;\Rightarrow \\
\;(o(t_2) = i_1(t_1) \times i_2(t_1))
\end{array}$$

### 2.8.2 Verification

The circuit diagram above is captured in logic as follows:

$$
\begin{array}{l}
\mathsf{Mult\_Imp}(i_1, i_2, o, done) \;\equiv \\
\;\exists b_1\; b_2\; b_3\; b_4\; l_1\; l_2\; l_3\; l_4\; l_5\; l_6\; l_7\; l_8\; l_9\; l_{10}. \\
\quad \mathsf{Mux}(done, l_8, l_7, l_6) \;\wedge\; \mathsf{Reg}(l_6, o) \;\wedge\; \mathsf{Add}(l_8, o, l_7) \;\wedge\; \mathsf{Dec}(i_1, l_5) \;\wedge \\
\quad \mathsf{Mux}(done, l_5, l_3, l_4) \;\wedge\; \mathsf{Mux}(done, i_1, l_2, l_1) \;\wedge\; \mathsf{Reg}(l_1, l_{10}) \;\wedge \\
\quad \mathsf{Dec}(l_{10}, l_2) \;\wedge\; \mathsf{Dec}(l_2, l_3) \;\wedge\; \mathsf{Zero}(l_9) \;\wedge\; \mathsf{Mux}(b_4, l_9, i_2, l_8) \;\wedge \\
\quad \mathsf{Zero\_Test}(i_1, b_4) \;\wedge\; \mathsf{Zero\_Test}(l_4, b_1) \;\wedge\; \mathsf{Zero\_Test}(i_2, b_2) \;\wedge \\
\quad \mathsf{Or\_Gate}(b_1, b_2, b_3) \;\wedge\; \mathsf{Flipflop}(b_3, done)
\end{array}
$$

The correctness of the multiplier implementation is established by proving that for all values of $i_1$, $i_2$, $o$ and $done$:

$$\mathsf{Mult\_Imp}(i_1, i_2, o, done) \Rightarrow \mathsf{Mult}(i_1, i_2, o, done)$$

Expanding the definition of Mult and then slightly rearranging the result yields:

$$
\begin{array}{l}
\forall t_1\; t_2.\; \mathsf{Mult\_Imp}(i_1, i_2, o, done) \wedge \\
\qquad done(t_1) \wedge \\
\qquad \mathsf{Next}(t_1, t_2)(done) \wedge \\
\qquad \mathsf{Stable}(t_1, t_2)(i_1) \wedge \\
\qquad \mathsf{Stable}(t_1, t_2)(i_2) \wedge \\
\qquad \;\Rightarrow \\
\qquad (o(t_2) = i_1(t_1) \times i_2(t_1))
\end{array}
$$

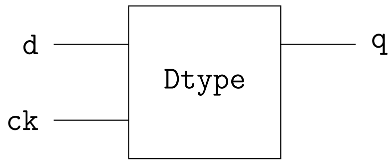
This can be proved by mathematical induction on $t_2-t_1$. The proof is mostly routine, but there are a few slightly tricky bits. Some elementary results concerning $+$ and $\times$ are required, together with the following lemmas about time:

$$
\begin{array}{l}
f(t{+}1) \Rightarrow \mathsf{Next}(t, t{+}1)(f) \\
\mathsf{Next}(t_1, t_2)(f) \;\wedge\; \neg f(t_1{+}1) \Rightarrow \mathsf{Next}(t_1{+}1, t_2)(f) \\
\mathsf{Next}(t_1, t_2)(f) \;\wedge\; \mathsf{Next}(t_1, t_3)(f) \Rightarrow (t_2{=}t_3) \\
\mathsf{Next}(t, (t{+}1){+}d)(f) \;\wedge\; \neg f(t{+}1) \Rightarrow \neg(d{=}0) \\
\mathsf{Stable}(t_1, t_2)(f) \Rightarrow \mathsf{Stable}(t_1{+}1, t_2)(f) \\
\mathsf{Stable}(t, (t{+}1) + d)(f) \;\wedge\; \neg(d{=}0) \Rightarrow (f(t) = f(t{+}1))
\end{array}
$$

We do not give details of the proof here.

## 2.9 An edge-triggered Dtype

The implementation of the multiplier described in the preceding section was described at the register-transfer level. This is an abstract level in which devices are viewed as sequential machines. At this level registers are modelled as unit-delay elements without explicit clock lines. To implement such a register using actual hardware, something like a Dtype device must be used:




### 2.9.1 Specification

An informal behavioural specification of `Dtype` is:

**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 $\mathsf{c}_1$ units of time before $t_1$ ($\mathsf{c}_1$ is the setup time), and
- there are at least $\mathsf{c}_2$ units of time between $t_1$ and $t_2$ ($\mathsf{c}_2$ is the minimum clock period)

**then**

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

To formalize this we need to define what a "rising edge" is. We will continue to use a discrete model of time, but the grain of time will be finer than before. A function from time to truth-values is defined to rise at time $t$ if it is $\mathsf{F}$ at time $t{-}1$ and $\mathsf{T}$ at $t$. Formally:

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

If the function $\mathsf{Rise}$ is applied to a single argument $f$, then the resulting expression $\mathsf{Rise}(f)$ denotes a predicate that is true of $t$ if and only if $f$ rises at $t$. The specification of the Dtype below illustrates the use of this kind of 'partial application'.

The informal behavioural specification of a Dtype can now be formalized in logic by:

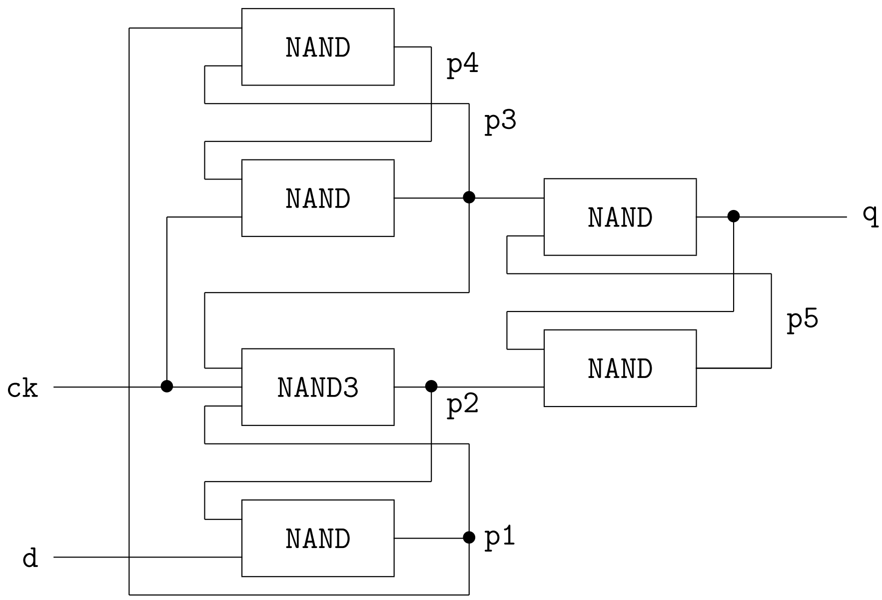
$$
\begin{array}{l}
\mathsf{Dtype}(\mathsf{c}_1,\mathsf{c}_2,\mathsf{c}_3,\mathsf{c}_4)(d,ck,q) \;\equiv \\
\forall t_1\ t_2.\ \mathsf{Rise}(ck)(t_1) \wedge \\
\qquad \mathsf{Next}(t_1,t_2)(\mathsf{Rise}(ck)) \wedge \\
\qquad (t_2{-}t_1 > \mathsf{c}_2) \wedge \\
\qquad \mathsf{Stable}(t_1{-}\mathsf{c}_1,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

The parameters $\mathsf{c}_1$, $\mathsf{c}_2$, $\mathsf{c}_3$ and $\mathsf{c}_4$ are the timing constants of the Dtype; their value depends on how the device is fabricated. Note that $\mathsf{Next}(t_1,t_2)(\mathsf{Rise}(ck))$ is an expression formed by applying $\mathsf{Next}(t_1,t_2)$ to the predicate $\mathsf{Rise}(ck)$.

A Dtype becomes a unit-delay if we abstract signals to the sequence of values occurring at rising edges of the clock. For more details of this kind of example, see the technical report *Formal Verification of Basic Memory Devices* by John Herbert, which is available from the Computer Laboratory (e.g. in the library).

### 2.9.2 Implementation

A common implementation of a Dtype uses NAND-gates:



### 2.9.3 Verification

To show that this implementation works we must use a model in which the NAND-gates have delay, since it is the delay in feedback loops that provides memory. The simplest such model is one in which each gate has unit-delay:

$$
\begin{aligned}
\mathsf{NAND}(i_1, i_2, o) &\equiv \forall t.\ o(t{+}1) = \neg(i_1(t) \wedge i_2(t)) \\
\mathsf{NAND3}(i_1, i_2, i_3, o) &\equiv \forall t.\ o(t{+}1) = \neg(i_1(t) \wedge i_2(t) \wedge i_3(t))
\end{aligned}
$$

The Dtype implementation can be represented in logic by defining:

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

One can then attempt to prove for suitable timing constants $\delta_1$, $\delta_2$, $\delta_3$ and $\delta_4$ that:

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

This would show that if each NAND-gate has unit-delay then the Dtype has a setup time of $\delta_1$, a minimum clock period of $\delta_2$, a start time of $\delta_3$ and a finish time of $\delta_4$. Formal proofs of this sort of result are fairly complicated and are not considered here. They are hard to get right: for example, my formulation *and proof* of a correctness statement was originally wrong. A recent automatic theorem proving tool called Mona[3] discovered a counterexample to this proof!

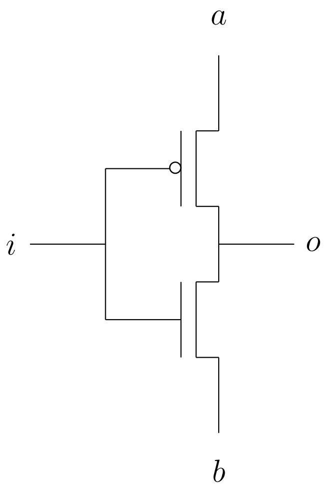
## 2.10 The simple switch model of transistors

This model was described in Section 2.3. It represents the two binary digital values with the logical truth values $\mathsf{T}$ and $\mathsf{F}$. An N-type transistor is modelled with a predicate $\mathsf{Ntran}$ that asserts that if the value on the gate is $\mathsf{T}$ then the values at the source and drain are equal; a P-type transistor is modelled by a predicate $\mathsf{Ptran}$ that asserts that if the value on the gate is $\mathsf{F}$ then the values at the source and drain are equal. The formal definitions are:

$$
\begin{aligned}
\mathsf{Ntran}(g, a, b) &= (g \Rightarrow (a = b)) \\[1ex]
\mathsf{Ptran}(g, a, b) &= (\neg g \Rightarrow (a = b))
\end{aligned}
$$

[3] `http://www.brics.dk/~mona/`

Then, for example, the behaviour of the circuit:




would be represented by defining the predicate $\mathsf{Circ}$ representing this circuit by:

$$\mathsf{Circ}(i, o, a, b) \;=\; \mathsf{Ptran}(i, a, o) \;\wedge\; \mathsf{Ntran}(i, o, b)$$

From this definition it follows by standard logical deductions (See Sections 2.4.3) that:

$$\mathsf{Circ}(i, o, \mathsf{T}, \mathsf{F}) \;=\; (o = \neg i)$$

which shows that if $a$ is connected to power and $b$ to ground, then the circuit behaves like an inverter.

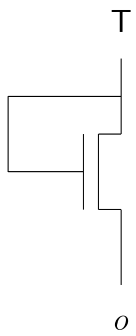
### 2.10.1 Inadequacies of the Simple Switch Model

Unfortunately, many analog features of circuits are missing in the simple switch model. As Tony Hoare has pointed out, it also can suggest behaviour that is not found in reality. For example:

$$\mathsf{Circ}(i, o, \mathsf{T}, \mathsf{F}) \;=\; (i = \neg o)$$

which suggests that if the circuit can also be used 'backwards' (i.e. with $o$ as input and $i$ as output). This is certainly not the case in practice.

Another problem with the model is that it assumes transistors conduct $\mathsf{T}$ and $\mathsf{F}$ equally well. For example, consider the following pull-up circuit:

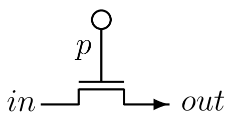



In the simple switch model, it follows that the output at $o$ is always $\mathsf{T}$, but in reality the value at $o$ will be weakened by the N-type transistor.

In the simple switch model, transistors are modelled as ideal switches which are controlled by the boolean logic level present on their gates. For example, the formal specification for an N-type transistor in this model describes this device as an ideal switch which is closed when its gate has the value $\mathsf{T}$ and open when its gate has the value $\mathsf{F}$. Although this very simple *switch model* of transistor behaviour can be useful for some purposes, it clearly fails to capture many important aspects of the way real CMOS devices behave.

One of these aspects is the fact that the switching behaviour of a real CMOS transistor does not depend simply on the 'logic level' present on its gate, but on the magnitude of the gate-to-source voltage V$gs$, compared to some non-zero threshold voltage V$t$. This means that a transistor does not behave like an ideal switch which can transmit both logic levels equally well. An N-type transistor, for example, transmits logic *low* well, but transmits logic *high* poorly. In the switch model, however, the specifications for N-type and P-type transistors do not reflect this important aspect of transistor behaviour—transistors are modelled as if they can transmit both logic levels equally well.

This simplification makes it possible to prove, using the switch model, the 'correctness' of certain CMOS circuits which do not work in practice. An example is the simple device shown below, where the value on the input *in* is transmitted through an N-type transistor to drive a capacitative load at the output *out*:




$$\vdash (\exists p.\, \mathsf{Pwr}\ p \land \mathsf{Ntran}(p, in, out)) = (out = in)$$

This circuit is simply an N-type transistor with its gate connected directly to power. In the switch model, this circuit is equivalent to a *wire* which connects the output directly to the input. This is stated formally by the correctness theorem shown on the right, which asserts that a formal model of this circuit, constructed using the switch model primitives, is logically equivalent to the specification '$out = in$'. In reality, however, the circuit shown above does not behave like a direct connection between *out* and *in*. If the output drives a capacitative load, and the input is at logic level *high*, then the voltage at *out* will only reach a level which is the threshold voltage V$t$ less than VDD. This voltage may be too low to drive the gate of another transistor, so it must be treated as distinct from the logic level *high*. The switch model correctness statement shown above is therefore misleading, for it asserts that an N-type transistor with its gate wired to VDD provides a completely transparent connection between *out* and *in*.

This threshold problem can also be illustrated with Circ:

$$\mathsf{Circ}(i, o, \mathsf{F}, \mathsf{T}) \;=\; (o = i)$$

$$\mathsf{Circ}(i, o, \mathsf{T}, \mathsf{T}) \;=\; (o = \mathsf{T})$$

In the first case, the actual value at $o$ will in reality be 'weak' due to lack of thresholds. In the second case, the output will be strong when $i$ is F but the output will be weak when $i$ is T (as in that case there will be no switching threshold); this behaviour is not represented at all in the simple switch model.

## 2.11 Fourman's switch model

The transistor model discussed in this section is based on a suggestion made by M. Fourman at the workshop on *Theoretical Aspects of VLSI Architectures* at the University of Leeds in 1986.

The fundamental problem with the switch model is that it specifies the behaviour of transistors using a logical type with only two values. In this very simple model, each wire in a circuit must have either the value *high* (modelled by T) or the value *low* (modelled by F). The physical phenomenon of a 'degraded' logic level—a logic level which is distinct from both these values, and which cannot be used to drive the gate of a transistor—is not even a *possibility* in this model.

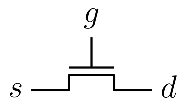
To overcome this problem, a type with more than two values is needed. The simplest solution is to use a defined logical type with exactly three distinct values.

$$tri \;\; ::= \;\; \mathsf{Hi} \;\mid\; \mathsf{Lo} \;\mid\; \mathsf{X}$$

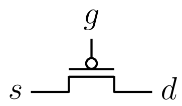
This informal definition of the type *tri* states that it denotes a set which contains exactly three distinct values—namely Hi, Lo, and X. This three-valued logical type can be used as the basis for a transistor model which at least partly captures the threshold switching behaviour of real CMOS devices. The basic idea of this model is to represent the strongly-driven logic levels *high* and *low* by the values Hi and Lo, and to represent all degraded logic levels, which cannot reliably drive the gates of transistors, by the value X.


$\vdash \mathsf{Gnd}\ g = (g = \mathsf{Lo})$


$\vdash \mathsf{Pwr}\ p = (p = \mathsf{Hi})$




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




$\vdash \mathsf{Ptran}(g, s, d) = ((g = \mathsf{Lo}) \Rightarrow ((d = \mathsf{Hi}) = (s = \mathsf{Hi})))$

Figure 2.1: A Threshold Switching Model.

The formal specifications shown in Figure 2.1 constitute a *threshold switching* model of CMOS transistor behaviour based on this representation of logic levels. The specifications $\mathsf{Pwr}\ p$ and $\mathsf{Gnd}\ g$ model VDD and VSS as constant sources of $\mathsf{Hi}$ and $\mathsf{Lo}$ respectively. The specifications for N-type and P-type transistors are intended to reflect the fact that these devices do not transmit both logic levels equally well. For example, it follows from the specification for an N-type transistor $\mathsf{Ntran}(g, s, d)$ that when the gate $g$ has the value $\mathsf{Hi}$ and the source $s$ has the value $\mathsf{Lo}$ (i.e. when the gate-to-source voltage is large) then the drain $d$ must also have value $\mathsf{Lo}$. This reflects the fact that the logic level modelled by $\mathsf{Lo}$ is transmitted unchanged through an N-type transistor. But when both $g$ and $s$ have the value $\mathsf{Hi}$, then the value of $d$ may be either $\mathsf{Hi}$ or $\mathsf{X}$. The specification '$\mathsf{Ntran}(g, s, d)$' is satisfied in both cases. This reflects the fact that the value $\mathsf{Hi}$ can be degraded to $\mathsf{X}$ when it is transmitted through an N-type transistor. The specification for a $\mathsf{P}$-type transistor is similar. In this case, when $g$ and $s$ are both $\mathsf{Lo}$ the value of $d$ can be either $\mathsf{Lo}$ or $\mathsf{X}$, reflecting the fact that the logic level modelled by $\mathsf{Lo}$ is only imperfectly transmitted through a P-type transistor.

With Fourman's model, the undesirable symmetry between inputs and outputs in $\mathsf{Circ}$ disappears because, for example, it can be shown that:

$$\mathsf{Circ}(i, o, \mathsf{Hi}, \mathsf{Lo}) \;=\; ((i = \mathsf{Hi}) \Rightarrow (o = \mathsf{Lo})) \quad \wedge \quad ((i = \mathsf{Lo}) \Rightarrow (o = \mathsf{Hi}))$$

but *not* that:

$$\mathsf{Circ}(i, o, \mathsf{Hi}, \mathsf{Lo}) \;=\; ((o = \mathsf{Hi}) \Rightarrow (i = \mathsf{Lo})) \quad \wedge \quad ((o = \mathsf{Lo}) \Rightarrow (i = \mathsf{Hi}))$$

The threshhold problem disappears, because all that follows about the output $o$ of the pull-up circuit is that $\neg(o = \mathsf{Lo})$. It does not follow that $o = \mathsf{Hi}$. Also:

$$\begin{array}{lcl} \mathsf{Circ}(i, o, \mathsf{Lo}, \mathsf{Hi}) & = & ((i = \mathsf{Hi}) \Rightarrow \neg(o = \mathsf{Lo})) \quad \wedge \quad ((i = \mathsf{Lo}) \Rightarrow \neg(o = \mathsf{Hi})) \\ \mathsf{Circ}(i, o, \mathsf{Hi}, \mathsf{Hi}) & = & ((i = \mathsf{Hi}) \Rightarrow \neg(o = \mathsf{Lo})) \quad \wedge \quad ((i = \mathsf{Lo}) \Rightarrow (o = \mathsf{Hi})) \end{array}$$

It *does not*, however, follow that:

$$\begin{array}{lcl} \mathsf{Circ}(i, o, \mathsf{Lo}, \mathsf{Hi}) & = & ((i = \mathsf{Hi}) \Rightarrow (o = \mathsf{Hi})) \quad \wedge \quad ((i = \mathsf{Lo}) \Rightarrow (o = \mathsf{Lo})) \\ \mathsf{Circ}(i, o, \mathsf{Hi}, \mathsf{Hi}) & = & ((i = \mathsf{Hi}) \Rightarrow (o = \mathsf{Hi})) \quad \wedge \quad ((i = \mathsf{Lo}) \Rightarrow (o = \mathsf{Hi})) \end{array}$$

## 2.12 Hoare's switch model

Hoare has proposed another approach to incorporating 'weak values' into a logical switch model. His idea is that each point $\mathsf{p}$ in a circuit be associated with both a value $p$, which is $\mathsf{T}$ or $\mathsf{F}$ as in the simple switch model, and a strength $\delta p$. Strengths are also represented by truth values: $\mathsf{T}$ represents a driven value and $\mathsf{F}$ a weak one. This idea is related to Bryant's model in which values-strength pairs are also used, but Bryant deals with the case where there may be more than two values and/or strengths, and his strengths model capacitive drive rather than transistor attenuation. In Hoare's model, the behaviour of transistors is defined by combining the simple switch model with rules that determine whether a point is driven.

$$\mathsf{Ntran}((g, \delta g),\ (s, \delta s),\ (d, \delta d)) \ = \ (g \Rightarrow (s = d)) \ \wedge \ (g \ \wedge \ \neg s \ \wedge \ \neg d \Rightarrow (\delta s = \delta d))$$

$$\mathsf{Ptran}((g, \delta g),\ (s, \delta s),\ (d, \delta d)) \ = \ (\neg g \Rightarrow (s = d)) \ \wedge \ (\neg g \ \wedge \ s \ \wedge \ d \Rightarrow (\delta s = \delta d))$$

These definitions are slightly different from the ones Hoare gives, but they are closely based on his ideas. The main difference is that Hoare makes the conservative assumption that the gates of transistors must be strongly driven for them to conduct.

With these definitions, it follows that $o = \mathsf{T}$ in the pull-up circuit on page 47, but not that $\delta o = \mathsf{T}$, i.e. it can be proved that the output $o$ has the value high, but not that this value is driven. This is an improvement over Fourman's model in which all that follows is that $o$ is 'not low'. Applying Hoare's model to $\mathsf{Circ}$ results in:

$$\mathsf{Circ}((i, \delta i), (o, \delta o), (\mathsf{T}, \mathsf{T}), (\mathsf{F}, \mathsf{T})) \ = \ (i = \neg o) \ \wedge \ \delta o$$

which says that $i$ and $o$ are inverses of each other and that the output is always driven. It does not follow that driving the output will drive the input, so the model is correctly asymmetrical. It also follows that:

$$\begin{array}{lcl} \mathsf{Circ}((i, \delta i), (o, \delta o), (\mathsf{F}, \mathsf{T}), (\mathsf{T}, \mathsf{T})) & = & (o = i) \\ \mathsf{Circ}((i, \delta i), (o, \delta o), (\mathsf{T}, \mathsf{T}), (\mathsf{T}, \mathsf{T})) & = & (o = \mathsf{T}) \ \wedge \ ((i = \mathsf{F}) \Rightarrow \ \delta o) \end{array}$$

The first of these equations shows that when $a$ and $b$ are driven low and high respectively (i.e. with the opposite values used in an inverter), then $i$ and $o$ will be equal, but neither will be driven. The second shows that when $a$ and $b$ are both driven high, then the output is always high, but it is only driven when the input is low.

The extra $\delta$-variables make Hoare's switch model more complex than Fourman's, but it appears to approximate the electrical intuition of circuits a little better. It also nicely separates the purely logical relations between values from considerations as to whether these values are driven. Hoare's model retains the logical purity of the simple switch model and handles threshhold effects by a separate additional mechanism. Thus a proof in the simple switch model can be extended to a proof in Hoare's model just by doing some extra calculations about $\delta$-variables.

A much-discussed problem with all three switch models is that they can assert that two values are equal when in practice they cannot possibly be. This means that the assertions derived from circuits by $\wedge$-ing together the individual transistor predicates may be false. For example, taking the simple switch model, if $i$ and $o$ are driven $\mathsf{T}$ and $b$ is driven $\mathsf{F}$ in $\mathsf{Circ}$, then it will follow from the simple switch model that $\mathsf{T} = \mathsf{F}$, which is just not true. The switch models are useful only when there are no shorts; this must be proved separately.

## 2.13 Summary

The examples presented here demonstrate that higher-order logic is a formalism in which a wide variety of behaviour and structure can be specified.

Hardware verification requires various kinds of reasoning.

- The adder example shows the need for mathematical induction (both to deal with iterated structures and for proving arithmetic lemmas).
- The multiplier example shows the need for reasoning about temporal concepts ($\mathsf{Next}$, $\mathsf{Stable}$ etc.).
- The Dtype and unit-delay show the need for reasoning about abstractions between different time scales.

All these kinds of reasoning can be done using the standard inference rules of logic.

## Chapter 3

# Temporal Abstraction



*This chapter is condensed by Mike Gordon from Tom Melham's PhD Thesis, which was subsequently revised and published as the book "Higher Order Logic and Hardware Verification" by Thomas F Melham, Cambridge Tracts in Theoretical Computer Science 31, Cambridge University Press, 1993.*

The formal specification of required behaviour for a large device—a microprocessor, for example—is unlikely to include as much information about how the device behaves over time as will be given by a detailed model of its design. A realistic model of a microprocessor might, for example, describe its behaviour at a level of temporal detail which includes information about system timing and propagation delay. But an abstract specification for this device is more likely to describe it as a finite state machine, in which the emphasis is on the sequence of operations carried out by the device, rather than the exact times at which these operations occur. This specification would then represent a *temporal abstraction* of the more detailed behaviour given by the model. It may, for example, state the behaviour a device is expected to exhibit at only certain significant or 'interesting' points of time, and leave unspecified the details of any intermediate states through which the device must pass to realize this behaviour. In this case, the specification will employ a more abstract notion of time than would be used in a more detailed design model—i.e. a model that *does* describe the device's behaviour at these intermediate states. A correctness condition based on temporal abstraction must therefore establish a relationship between two different formal representations of time: an 'abstract' representation of time in the specification, and a 'concrete' representation of time in the model.

In the general case, each *unit* of discrete time in the specification corresponds to an *interval* of discrete time in the more detailed design model. In this case, the specification describes the values that appear on the external wires of the device at fewer points of 'real' time than the model does. Each point of 'abstract time' in the specification corresponds to a particular point of 'concrete time' in the model. And, at these corresponding points in time, the model and the specification impose the same constraint on the values that can appear on the external wires of the device. But the model also constrains these values at points of concrete time which lie *between* what are considered to be adjacent points of time at the more abstract level of the specification. A correctness statement that relates this model to the more abstract specification must therefore establish a correspondence between two different time-scales: a 'fine-grained' time-scale in the model and a 'coarse-grained' time-scale in the specification.

This correspondence can be described formally by a function that maps each point of abstract time in the specification to a corresponding point of concrete time in the model. Such a function is a *time mapping* that describes the precise relationship between the abstract time-scale used in the specification and the concrete time-scale used in the more detailed model. A simple example is shown in Figure 3.1. The solid lines in this diagram represent continuous or real time. The dots represent the points of real time which constitute the two discrete time-scales involved: the concrete time-scale $t_c$ used in the model, and the abstract time-scale $t_a$ used in the specification. The mapping $\mathsf{f}$ describes the relationship between these two time-scales. To every point of time $t$ on the abstract time-scale, the function $\mathsf{f}$ assigns a corresponding point of concrete time '$\mathsf{f}\ t$' such that the order of time is preserved:

$$\vdash \forall t_1\, t_2.\ (t_1 < t_2) \Rightarrow (\mathsf{f}\ t_1 < \mathsf{f}\ t_2)$$

This establishes a correspondence between units of time on the abstract time-scale and intervals of time on the concrete time-scale by mapping successive points of abstract time to *selected* points of concrete time.

Any correspondence between successive units of abstract time and contiguous intervals of concrete time can be described formally in logic by a time mapping of this kind. The particular point of concrete time assigned by such a function to each point of abstract time will, of course, depend on the exact relationship between the model and the specification involved. For example, each unit of abstract time in the specification for a clocked synchronous device might correspond to an interval of concrete time between two rising edges of a clock signal in the model. In this case, the function $\mathsf{f}$ would map points of time on the abstract time-scale to the points of concrete time at which these rising edges of the clock occur. A detailed account of how such a function can be defined is given in Section 3.2.

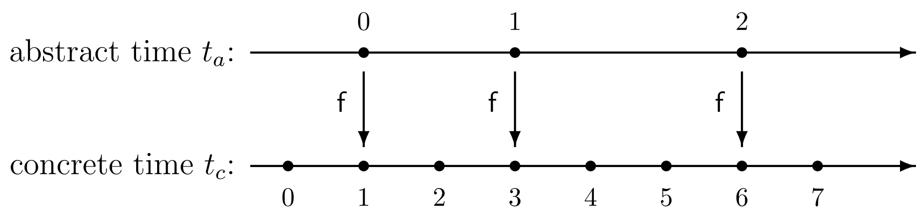



Figure 3.1: A Mapping between Time-scales.

Given an appropriately-defined mapping $\mathsf{f}$ from abstract time to concrete time, a correctness statement that relates a model to a specification at a higher level of temporal abstraction can be formulated in logic as follows. Suppose that the two logical terms

$$M[c_1, \ldots, c_n] \qquad \text{and} \qquad S[a_1, \ldots, a_n]$$

are the model of a device and an abstract specification of required behaviour, respectively. To simplify matters, assume that the free variables in both the model and the specification are functions of type *num→bool*, and that $c_i$ corresponds to $a_i$ for $1 \leq i \leq n$. If the specification is a temporal abstraction of the model (in the sense discussed above) and if the device is correct, then each sequence of values $a_i$ in the specification will correspond to a *subsequence* of the values given by the variable $c_i$ in the model. Each function $a_i$ in the specification will represent a sequence of values which could be obtained by 'sampling' the function $c_i$ at only those points of concrete time which are significant at the abstract level of the specification, and therefore correspond to points of discrete time on the abstract time-scale used in the specification.

Given a function $\mathsf{f}$ that describes this correspondence, a correctness statement that relates the model to the specification can be formulated as shown below.[1]

$$\vdash \forall c_1 \; \cdots \; c_n.\; M[c_1, \ldots, c_n] \Rightarrow S[c_1 \circ \mathsf{f}, \ldots, c_n \circ \mathsf{f}]$$

This theorem states that if the functions $c_1, \ldots, c_n$ satisfy the temporally detailed constraint imposed by the model, then the functions $c_1 \circ \mathsf{f}, \ldots, c_n \circ \mathsf{f}$ will satisfy the temporally abstract specification of required behaviour. Here, the model describes the values that appear on each external wire $c_i$ at points of fine-grained or concrete time. The function $\mathsf{f}$ specifies which of these points of concrete time correspond to points of time on the abstract time-scale. Composition on the right with $\mathsf{f}$ constructs an abstract sequence of values '$c_i \circ \mathsf{f}$' from each detailed sequence of values $c_i$ by sampling the function $c_i$ at these selected points of concrete time. The combination of abstract sequences obtained in this way must satisfy the abstract specification of required behaviour.

The resulting correctness statement asserts that the combinations of values present on the external wires of the device satisfy the specification of required behaviour at each point in time that is regarded as significant or important at the abstract level of description. That is, the behaviour of the device when observed at only those selected points of concrete time specified by the function $\mathsf{f}$ satisfies the temporally abstract specification of its required behaviour.

The advantage of temporal abstraction is that it hides irrelevant details about intermediate state transitions from the abstract specification of required behaviour for a device. Points

[1] $f \circ g$ denotes the functional composition of $f$ and $g$, i.e. $(f \circ g)\; x = f(g\; x)$.

of time on the abstract time-scale in a correctness theorem based on temporal abstraction correspond to *selected* points of time on the more detailed concrete time-scale, and it is only at these selected points of time that the device's behaviour is stipulated by the abstract specification. Furthermore, the use of a time mapping to relate abstract and concrete time-scales not only allows the behaviour of the device at *other* points of time to be left unconstrained by the specification, but also makes intermediate states transitions completely invisible to the specification. Intermediate states represented by points of time on the concrete time-scale used in the model simply do not exist on the abstract time-scale used in the specification. This allows the specification to describe the required behaviour of a device at only significant points of time, without also having to indicate precisely *which* points of time are in fact of interest.

## 3.1 Two Problems

Correctness is stated by an implication having the form $M \Rightarrow S$, in which the model is the antecedent and a substitution instance of the specification is the consequent. There are two problems that can arise when correctness is stated by an implication of this form. These are discussed briefly in the two sections that follow.

### 3.1.1 Underspecification

Whenever the behaviour which a device is required to exhibit is stated formally by a partial specification, there is the possibility that this partial specification in fact *under*specifies the intended behaviour of the device. That is, a partial specification may inadvertently fail to stipulate some important aspect of the device's intended behaviour, and therefore be satisfied by a wider range of values than is actually intended by the designer. In this case, the constraint expressed by the specification will be satisfied by some combinations of values which in fact ought *not* to appear on the external wires of the device. But when correctness is formulated as logical implication, a model which is satisfied by these undesirable combinations of values (and therefore represents an *incorrect* design) will, according to this formal notion of correctness, be considered correct with respect to this specification.

This is much less likely to happen when correctness is stated formally by logical equivalence. If the specification and the model are required to express the *same* constraint on the free variables which they contain, then any weakness in the specification must either: (1) be matched exactly by a corresponding degree of 'nondeterminism' in the model, or (2) make it impossible to complete the proof of correctness. But if the criterion of correctness is relaxed to logical implication, then the specification is allowed to express a strictly weaker constraint than the model. An inadequate specification is therefore less likely to be detected during the course of a proof, since the behaviour given by the model is required only to lie somewhere within the range of behaviour stipulated by the specification.

There is no complete solution to this problem, since it is a problem of inaccuracy in the specification of intended behaviour for a device. It is not possible to *prove* that a partial specification in fact covers all the essential aspects of a device's intended behaviour. Whenever it is possible to leave *something* unspecified, it is also possible to leave something *essential* unspecified.

### 3.1.2 Inconsistent Models

A second problem with using logical implication to express correctness is that an *inconsistent* model then trivially satisfies any specification. An inconsistent model is one which cannot be satisfied by any assignment of values to its free variables. A simple example is the term '$\mathsf{Pwr}\ x \wedge \mathsf{Gnd}\ x$', where $\mathsf{Pwr}\ x$ and $\mathsf{Gnd}\ x$ are instances of the specifications for power and ground defined in Section 2.3. This term is logically equivalent to $\mathsf{F}$ (i.e. falsity), since no boolean value $x$ can satisfy both $\mathsf{Pwr}\ x$ and $\mathsf{Gnd}\ x$. If satisfaction is formulated as logical implication, then this inconsistent model satisfies (i.e. implies) *any* specification. In general, if the model on the left hand side of the implication:

$$M[v_1, \ldots, v_n] \Rightarrow S[v_1, \ldots, v_n]$$

is false for all values of the variables $v_1, \ldots, v_n$, then this implication is a *theorem*, no matter what constraint is imposed on these variables by the term on the right hand side of the implication. This is clearly unsatisfactory, since a formal theorem of this kind provides no meaningful assurance of functional correctness.

The ideal solution to this problem would be to have a collection of specifications for the primitive components used in designs that always yields a consistent model, no matter how this model is constructed from these primitives using the syntactic operations of composition ('$\wedge$') and hiding ('$\exists$'). This, however, may require the specifications for primitive components to be of considerable complexity. A more pragmatic solution is to check the consistency of the particular design model on which proof of correctness is based. This can be done by proving a consistency theorem of the form:

$$\vdash \exists v_1 \ldots v_n.\, M[v_1, \ldots, v_n]$$

in addition to proving a correctness statement of the general form illustrated by the implication shown above. Proving this extra consistency theorem ensures that the model shown above can be satisfied by at least one combination of values for the variables $v_1, \ldots, v_n$. It therefore shows that this model does not satisfy a specification merely because it is inconsistent. If none of the external wires of a device are bidirectional (i.e.

every wire of the device is either an input or an output), then a stronger consistency theorem can be formulated:

$$\vdash \forall i_1 \ldots i_n.\ \exists o_1 \ldots o_m.\ M[i_1, \ldots, i_n, o_1, \ldots, o_m]$$

This theorem states that for any collection of input values $i_1, \ldots, i_n$, there are output values $o_1, \ldots, o_m$ which, according to the model, are consistent with them. Again, this shows that the model does not satisfy a specification of required behaviour merely because it is inconsistent.

In general, it is necessary to prove a consistency theorem of one of these two kinds, in addition to a correctness theorem, whenever a satisfaction relation based on implication is used. Consistency theorems are usually not proved in most of the examples presented in the literature, since the models which are used are generally simple enough to be easily seen to be consistent. But when formal verification is applied to much larger examples, it may be necessary to consider more explicitly the possibility that the models involved might be inconsistent.

## 3.2 More on Temporal Abstraction

With temporal abstraction, the abstract specification for a device simply describes its externally observable behaviour at fewer points of time than the formal model of its design. The grain of discrete time used in the specification is 'coarser' than the grain of discrete time used in the model, and each *unit* of discrete time at the abstract level of description corresponds to an *interval* of time at the more detailed level of description.

To express this abstraction relationship formally in logic, the idea of a mapping between time-scales was introduced. A mapping of this kind specifies a correspondence between successive points of time on an abstract time-scale and selected points of time on a concrete time-scale. Given an appropriate time mapping $\mathsf{f}$, a correctness statement based on temporal abstraction by sampling is formulated in logic as shown below:

$$\vdash M[c_1, \ldots, c_n] \Rightarrow S[c_1 \circ \mathsf{f}, \ldots, c_n \circ \mathsf{f}]$$

The model $M[c_1, \ldots, c_n]$ in this correctness theorem describes the values that appear on each external wire $c_i$ at points of fine-grained or concrete time. The abstract specification is a constraint of the form $S[a_1, \ldots, a_n]$, and specifies the desired behaviour in terms of the values allowed on its external wires at points of coarse-grained or abstract time. The time mapping $\mathsf{f}$ defines the intervals of concrete time that correspond to each unit of abstract time. The correctness theorem states that whenever a sequences of values denoted by $c_i$ satisfies the temporally detailed model, the subsequence $c_i \circ \mathsf{f}$, obtained by sampling $c_i$ at the points of concrete time specified by $\mathsf{f}$, will satisfy the temporally abstract specification of required behaviour. Proving a correctness statement of this form involves showing that if the sequences $c_1$, …, $c_n$ take on the intermediate values defined by model, then the values of these sequences the selected points of time specified by the time mapping $\mathsf{f}$ will satisfy the abstract specification.

This correctness relationship is formulated as an implication, rather than an equivalence, because there may be several non-equivalent ways of implementing behaviour specified by the abstract constraint $S[a_1, \ldots, a_n]$. The implementation in which the sequences $c_1$, …, $c_n$ take on the intermediate values defined by the model $M[c_1, \ldots, c_n]$ is only *one* such method. In the example given above, every sequence $c_i$ in the model (every '*signal*') is sampled using the same time mapping $\mathsf{f}$. In general, however, it is not necessary that the same time mapping be used for every signal. For example, some signals may be sampled at points of time corresponding to the rising edges of a clock, while others are sampled at points of time corresponding to the falling edges of a clock.

Any correspondence between successive units of abstract time and contiguous intervals of concrete time can be specified formally in logic by a time mapping of the kind used in the correctness statement shown above. Such a mapping is just a function $\mathsf{f}$ of type *num*→*num* that assigns a particular point of concrete time to each point of abstract time, as shown in Figure 3.2. Not every function of logical type *num*→*num*, however, specifies a valid correspondence between time-scales. A mapping from abstract to concrete time must be a strictly *increasing* function on the natural numbers. This requirement on a time mapping $f$ can be expressed formally by the predicate $\mathsf{Incr}$ defined as follows.

$$\vdash \mathsf{Incr}\ f = \forall\, t_1\ t_2.\ (t_1 < t_2) \Rightarrow (f\ t_1 < f\ t_2)$$

This ensures that if time $t_1$ comes before time $t_2$ on the abstract time-scale, then this relationship also holds between the corresponding points of time $\mathsf{f}\ t_1$ and $\mathsf{f}\ t_2$ on the concrete time-scale.

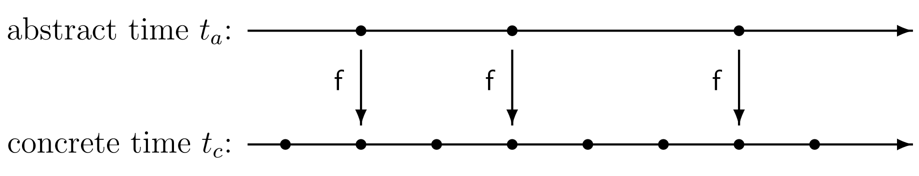



Figure 3.2: A Typical Mapping between Time-scales.

### 3.2.1 Constructing Mappings between Time Scales

The first step in the formulation of a correctness statement that involves temporal abstraction by sampling is to define an appropriate mapping from the abstract time-scale used in the specification to the concrete time-scale used in the model. In general, the points of concrete time that correspond to points of abstract time may depend on the behaviour of the device itself. In this case, a fixed mapping from abstract time to concrete time—for example a function that maps successive points of abstract time to every tenth point of concrete time—is not possible.

Consider, for example the correspondence between time-scales shown below in Figure 3.3. Here, successive points of abstract time correspond to the points of concrete time at which there is a rising edge of the clock signal $ck$. The precise correspondence between concrete time and abstract time depends on the behaviour of this clock signal, and the mapping $\mathsf{f}$ must be defined in such a way as to reflect this dependence. This can be done formally by *constructing* the function $\mathsf{f}$ shown in Figure 3.3 from the predicate '$\mathsf{Rise}\ ck$', which identifies those points of time on the concrete time-scale at which the rising edges of the clock $ck$ occur.

In general, any time mapping can be defined formally by means of a predicate that specifies which points of time on the concrete time-scale are to correspond to points of time on the abstract time-scale. The idea is to define this predicate such that it is true of precisely those selected points of concrete time which are to be in the image of the mapping from abstract time to concrete time. The free variables in the model can themselves be used as parameters to this predicate. In synchronous systems, for example, the appropriate points of concrete time can often be identified by the value of a clock signal $ck$. (In asynchronous systems, handshaking signals might be used for the same purpose.) The required time mapping can then be constructed from the values given by this predicate on concrete time. This allows the mapping from abstract time to concrete time used in a correctness statement to reflect the time-dependent behaviour of the device itself (as described by the model). The time mapping does not assign a fixed point of concrete

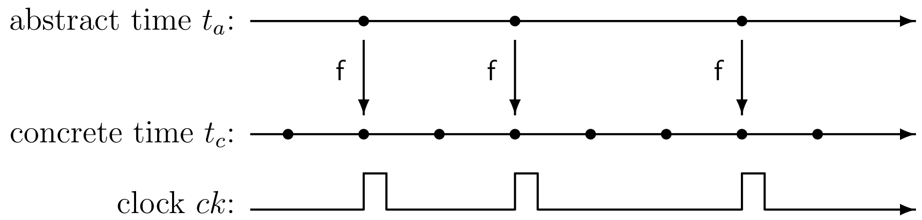



Figure 3.3: A Time Mapping which Depends on a Clock $ck$.

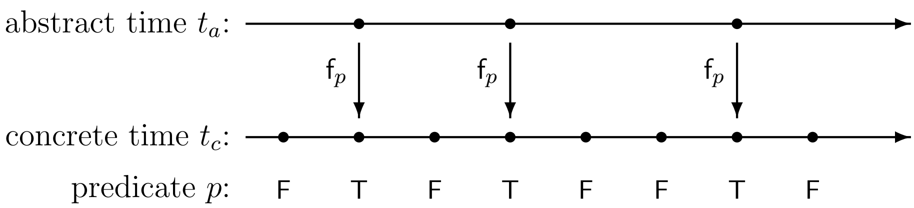



Figure 3.4: Constructing a Time Mapping $\mathsf{f}_p$ from a Predicate $p$.

time to each point of abstract time, but establishes a correspondence between time-scales that covers the entire range of time-dependent behaviour described by the model.

To construct a time mapping in this way, it sufficient to define a predicate $p{:}num{\rightarrow}bool$ that is true of precisely those points of concrete time which are to correspond to points of abstract time. Given such a predicate $p$, it is possible to construct a mapping $\mathsf{f}_p$ which assigns each point of abstract time $n$ to the $n$th point of concrete time at which $p$ is true, as shown in Figure 3.4 above. If for any predicate $p$ and abstract time $n$ the term '$\mathsf{Timeof}\ p\ n$' denotes the point of concrete time at which $p$ is true for the $n$th time, then the mapping between time-scales $\mathsf{f}_p$ shown in this diagram can be defined by:

$$\vdash \mathsf{f}_p\ n = \mathsf{Timeof}\ p\ n \qquad\qquad (\text{i.e.}\quad \vdash \mathsf{f}_p = \mathsf{Timeof}\ p)$$

It remains to define the function $\mathsf{Timeof}$ formally in higher order logic.

### 3.2.2 Defining the Function Timeof

The term '$\mathsf{Timeof}\ p\ n$', as informally described above, may in fact be undefined for some values of $p$ and $n$. If the predicate $p$ true at only a finite number of points of concrete time, then there will be some number $N$ such that for all $n > N$ there is no concrete time at which $p$ is true for the $n$th time. The value of $\mathsf{Timeof}\ p\ n$ is therefore 'undefined' for these values of $n$, and the function $\mathsf{Timeof}\ p$ is itself a partial function.

In higher order logic, however, all functions must be total functions. The higher order function $\mathsf{Timeof}$ will therefore be defined to be a total function whose value is only partially specified. This will be done by using the primitive constant $\varepsilon$ to define $\mathsf{Timeof}$ such that '$\mathsf{Timeof}\ p$' denotes the required mapping between time-scales when the predicate $p$ is true infinitely often, and denotes a mapping about which nothing can be proved when $p$ is true only finitely often.

#### 3.2.2.1 The Relation $\mathsf{Istimeof}$

The formal definition of $\mathsf{Timeof}$ is based on a relation $\mathsf{Istimeof}$, defined such that the term '$\mathsf{Istimeof}\ p\ n\ t$' has the meaning '$p$ is true for the $n$th time at time $t$'. The definition of this relation is done by primitive recursion on the natural number $n$. When $n$ is zero, the defining equation is:

$$\vdash \mathsf{Istimeof}\ p\ 0\ t = p\ t \wedge \forall t'.\, t' < t \Rightarrow \neg(p\ t')$$

That is, the predicate $p$ is true for the first (i.e. the 0th) time at concrete time $t$ if $p$ is true at time $t$ and false at every point of time prior to time $t$. For the $(n{+}1)$th time at which $p$ is true, the defining equation is:

$$\vdash \mathsf{Istimeof}\ p\ (\mathsf{Suc}\ n)\ t = \exists t'.\, \mathsf{Istimeof}\ p\ n\ t' \wedge \mathsf{Next}\ t'\ t\ p$$

where the auxiliary predicate $\mathsf{Next}$ is defined by:

$$\vdash \mathsf{Next}\ t_1\ t_2\ p = t_1 < t_2\ \wedge\ p\ t_2\ \wedge\ \forall t.\, (t_1 < t \wedge t < t_2) \Rightarrow \neg p\ t$$

In this case, the defining equation for $\mathsf{Istimeof}$ states that $p$ is true for the $(n{+}1)$th time at concrete time $t$ if there exists a point of concrete time $t'$ prior to time $t$ at which $p$ is true for the $n$th time, and $t$ is the next time after $t'$ at which $p$ is true. To summarize, the primitive recursive definition of the relation $\mathsf{Istimeof}$ is given by the two theorems shown below.

$$\begin{array}{lcl}\vdash \mathsf{Istimeof}\ p\ 0\ t & = & p\ t\ \wedge\ \forall t'.\, t' < t \Rightarrow \neg p\ t' \\ \vdash \mathsf{Istimeof}\ p\ (\mathsf{Suc}\ n)\ t & = & \exists t'.\, \mathsf{Istimeof}\ n\ p\ t' \wedge \mathsf{Next}\ t'\ t\ p\end{array}$$

#### 3.2.2.2 A Theorem about $\mathsf{Istimeof}$

This primitive recursive definition of $\mathsf{Istimeof}\ p\ n\ t$ captures the idea that the predicate $p$ is true for the $n$th time at concrete time $t$. There is no guarantee, however, that such a time $t$ exists for all values of $p$ and $n$. In order to use the relation $\mathsf{Istimeof}$ to define the function $\mathsf{Timeof}$, it is necessary to show that if the predicate $p$ is true infinitely often, then for all $n$ there is a unique time $t$ at which $p$ is true for the $n$th time. That is, if $p$ is true infinitely often, then the relation $\mathsf{Istimeof}\ p\ n\ t$ defines a unique value $t$ for each value of $n$, and therefore in fact represents well-defined total function that maps $p$ and $n$ to $t$.

The condition that $p$ must be true at an infinite number of points of concrete time is stated formally by the predicate $\mathsf{Inf}$ defined below.

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

Given this predicate, it is straightforward to show that if $p$ is true infinitely often, then for all $n$ there exists a unique time $t$ at which $p$ is true for the $n$th time:

$$\vdash \mathsf{Inf}\ p \Rightarrow \forall n.\ \exists!\, t.\ \mathsf{Istimeof}\ p\ n\ t \tag{3.1}$$

The formal proof of this theorem proceeds by proving the existence and uniqueness parts separately. The existence of $t$ follows by induction on $n$, using the well ordering property of natural numbers:

$$\vdash \forall p.\ \exists t. p\ t \Rightarrow \exists t.\ p\ t \wedge \forall t'.t' < t \Rightarrow \neg p\ t'$$

to infer from the assumption $\mathsf{Inf}\ p$ that there is always a smallest *next* time at which the predicate $p$ is true. The uniqueness of $t$ also follows by induction on $n$.

#### 3.2.2.3 Using Istimeof to Define Timeof

Given theorem (3.1), the relation Istimeof can be used to define the function Timeof as follows. Using the primitive constant $\varepsilon$, the function Timeof can be defined formally by the equation shown below.

$$\vdash \mathsf{Timeof}\ p\ n = \varepsilon\ (\mathsf{Istimeof}\ p\ n)$$

This equation defines the term $\mathsf{Timeof}\ p\ n$ to denote some time, $t$ say, such that $\mathsf{Istimeof}\ p\ n\ t$ is true. If no such time exists, then $\mathsf{Timeof}\ p\ n$ denotes an arbitrary natural number. Using the primitive constant $\varepsilon$, this definition makes the term '$\mathsf{Timeof}\ p$' always denote a total function. The term '$\mathsf{Timeof}\ p\ n$' denotes *some* natural number for all values of $n$ and $p$, even when the predicate $p$ is true at only a finite number of points of concrete time.

If, however, the predicate $p$ is true infinitely often, then for all $n$ there will exist a unique time $t$ such that $\mathsf{Istimeof}\ p\ n\ t$ is true. Thus, if $\mathsf{Inf}\ p$ holds, then $\mathsf{Timeof}\ p\ n$ will in fact denote the unique time at which $p$ is true for the $n$th time, as desired. More formally, an immediate consequence of the existence part of theorem (3.1) is:

$$\vdash \mathsf{Inf}\ p \Rightarrow \mathsf{Istimeof}\ p\ n\ (\mathsf{Timeof}\ p\ n)$$

from which it follows immediately that:

$$\vdash \mathsf{Inf}\ p \Rightarrow p(\mathsf{Timeof}\ p\ n)$$
$$\vdash \mathsf{Inf}\ p \Rightarrow \forall t.\ (t < (\mathsf{Timeof}\ p\ 0)) \Rightarrow \neg p\ t$$

That is, if the predicate $p$ is true infinitely often, then $\mathsf{Timeof}\ p\ n$ always denotes a point of concrete time at which $p$ is in fact true, and $\mathsf{Timeof}\ p$ maps 0 to the first time at which

$p$ is true. From the uniqueness part of theorem (3.1) it also follows that $\mathsf{Timeof}\ p$ denotes an increasing function from abstract to concrete time, and that this function does not skip any points of concrete time identified by the predicate $p$:

$$\vdash \mathsf{Inf}\ p \Rightarrow \forall n.\, (\mathsf{Timeof}\ p\ n) < (\mathsf{Timeof}\ p\ (n{+}1))$$
$$\vdash \mathsf{Inf}\ p \Rightarrow \forall n\, t.\, (\mathsf{Timeof}\ p\ n) < t \wedge t < (\mathsf{Timeof}\ p\ (n{+}1)) \Rightarrow \neg p\ t$$

These lemmas about Timeof show that if the predicate $p$ is true infinitely often, then the term '$\mathsf{Timeof}\ p$' is a well defined total function and denotes the desired mapping from abstract time to selected points of concrete time. The function $\mathsf{Timeof}\ p$ maps each point of abstract time $n$ to the point of concrete time at which $p$ is true for the $n$th time, as required.

### 3.2.3 Using Timeof to Formulate Correctness

Having formally defined the function Timeof, and shown that it constructs a well defined time mapping for any predicate $p$ that is true at an infinite number of points of concrete time, it is possible to use Timeof to formulate correctness theorems based on temporal abstraction by sampling. The time mapping required for such a correctness theorem just an increasing function f of type $num{\rightarrow}num$. Any such function can be defined using Timeof and an appropriate predicate $p$ which indicates the points of concrete time that are to correspond to points of abstract time. Formally, the property that a function $f$ is strictly increasing is logically equivalent to the assertion that $f$ can be constructed from a predicate $p$ for which $\mathsf{Inf}\ p$ holds:

$$\vdash \forall f.\, \mathsf{Incr}\ f = \exists p.\, \mathsf{Inf}\ p \wedge f{=}\mathsf{Timeof}\ p$$

This theorem follows from the definition of the constant Incr and the properties of Timeof discussed above in Section 3.2.2.3.

If $p$ is an appropriate predicate that indicates which points of concrete time correspond to points of abstract time, then a correctness theorem that relates a detailed design model $M[c_1,\ldots,c_n]$ to an abstract specification $S[a_1,\ldots,a_n]$ can be formulated in logic as shown below:

$$\vdash M[c_1,\ldots,c_n] \Rightarrow S[c_1 \circ (\mathsf{Timeof}\ p),\ldots,c_n \circ (\mathsf{Timeof}\ p)]$$

This correctness theorem states that whenever the signals $c_1$ …, $c_n$ satisfy the model, the abstract signals constructed by sampling $c_1$, …, $c_n$ when the predicate $p$ is true will satisfy the temporally abstract specification. In the general case, the predicate $p$ can be defined in terms of the variables $c_1$, …, $c_n$, in order to make the times at which the values in the model are sampled depend on the behaviour of the device itself.

If when is an infix constant defined formally as follows:

$$\vdash s \;\mathsf{when}\; p = s \circ (\mathsf{Timeof}\; p)$$

then this correctness statement can be written:

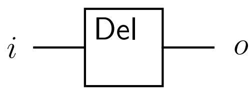
$$\vdash M[c_1, \ldots, c_n] \Rightarrow S[c_1 \;\mathsf{when}\; p, \ldots, c_n \;\mathsf{when}\; p]$$

Since every mapping from abstract to concrete time can be constructed using ‘Timeof’ from an appropriate predicate $p$, any correctness relationship based on temporal abstraction by sampling can be expressed in this form.

The example given in the next section shows how the when operator defined above can be used to formulate the correctness of a D-type flip flop with respect to the abstract specification of one-bit unit delay register.

## 3.3 A Simple Example

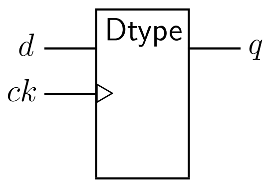
A commonly used register-transfer level device is the unit delay, described formally by the specification shown below.




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

This specification simply states that the value on the output $o$ is equal to the value on the input $i$ delayed by one unit of discrete time.

The unit delay device described by this specification is an abstraction—there are many circuits that can implement the abstract behaviour described by the specification $\mathsf{Del}(i, o)$. An implementation is a simplification of the rising edge triggered D-type flip flop discussed in Section 2.9. The sequential behaviour of this device is modelled in logic by the term $\mathsf{Dtype}(ck, d, q)$ defined below:




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

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

Informally, the D-type device shown above implements a unit delay by sampling the input value $d$ when the clock rises and holding this value on the output $q$ until the next rise of the clock. In this way the D-type delays by one clock period the sequence of values consisting of the values present on the input $d$ at successive rising edges of the clock $ck$. This suggests that the time mapping used to relate the model $\mathsf{Dtype}(ck, d, q)$ to the abstract specification $\mathsf{Del}(i, o)$ should map successive points of abstract time to the points of concrete time at which the clock rises.

Using the constant Rise, the required time mapping is given by the term '$\mathsf{Timeof}\ (\mathsf{Rise}\ ck)$'. Given the mapping between time-scales denoted by this term, a correctness statement that relates the D-type model to the unit delay specification can then formulated as shown below:

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

This correctness theorem states that if the sequences given by the variables $ck$, $d$, and $q$ satisfy the model, then the abstract sequences obtained by sampling $d$ and $q$ at successive rising edges of the clock $ck$ will satisfy the abstract specification for a unit delay device. Here, there is a *family* of sampling functions used to relate the model to the abstract specification. For each value of the clock $ck$, the term $\mathsf{Rise}\ ck$ denotes an appropriate predicate that identifies the points of concrete time which at which the clock rises. The infix when operator is then used to sample the sequences $d$ and $q$ whenever this predicate is true.

The assumption that the clock rises infinitely often is a *validity condition* on the abstraction relationship expressed by this correctness statement. The theorem asserts that the specification represents a valid abstract view of the behaviour of a D-type device only if the validity condition '$\mathsf{Inf}(\mathsf{Rise}\ ck)$' is satisfied. This validity condition on the clock must be met by the environment in which the D-type flip flop is placed. The condition itself is as unrestrictive as possible: the clock $ck$ is not required to be regular or have a minimum period. The liveness condition on the clock input expressed by $\mathsf{Inf}(\mathsf{Rise}\ ck)$ is sufficient for the D-type device to function correctly as specified by $\mathsf{Del}(i, o)$

The proof of this correctness theorem is straightforward. The main step is an induction on the number of time steps between adjacent rises of the clock, showing that the value on the input $d$ that is sampled at each rising edge of the clock $ck$ is held on the output $q$ until the next rising edge. The correctness theorem then follows easily from this result and the properties of Timeof proved above in Section 3.2.2.3.

This proof provides a very simple example of a common type of temporal abstraction, where contiguous intervals of concrete time correspond to successive units of abstract time. Examples involving detailed timing information or several different time mappings in the same correctness statement are typically much more complex than the simple example given here. But—as far as the abstraction relationship itself is concerned—more complex examples of temporal abstraction by sampling involve the same general approach as illustrated by this example.

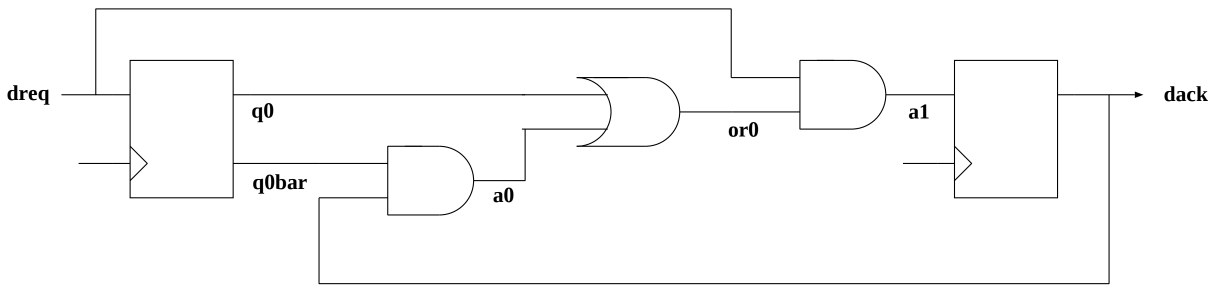
Chapter 4

# Model checking

*Temporal logic has already been introduced as a way of extending Floyd-Hoare logic to deal with continuously-running programs and hardware specified in an HDL. Temporal logic can also be used to reason directly about hardware structures. An important technique is Model checking, that allows certain properties to be established completely automatically. Particularly efficient model checking algorithms exist for branching-time logics, as opposed to the linear time logics introduced earlier.*

## 4.1 Transition systems

Consider the following circuit (which could be used to generate acknowledgements from a handshake request):[1]




This is represented in higher order logic by a formula like:

[1]This example was supplied by John Herbert.

```
 ∀dreq dack.
   RECEIVER(dreq,dack) =
   (∃q0 qbar0 a0 or0 a1.
     DTYPE_BAR(dreq,q0,qbar0) ∧
     AND(qbar0,dack,a0) ∧
     OR(q0,a0,or0) ∧
     AND(dreq,or0,a1) ∧
     DTYPE(a1,dack))
```

Using the usual methods, this simplifies to:

```
∀dreq dack.
RECEIVER(dreq,dack) =
(∃q0.
  (∀t. q0(t + 1) = dreq t) ∧
  (∀t. dack(t + 1) =
        dreq t ∧ (q0 t ∨ (¬q0 t ∧ dack t))))
```

If the state is represented by a triple `(dreq,q0,dack)` consisting of the value being input and the values stored in the two registers, then the implementation `RECEIVER(dreq,dack)` defines a *transition system*:

```
(dreq, q0, dack)  --->  (dreq', dreq, dreq ∧ (q0 ∨ (¬q0 ∧ dack)))
```

For some temporal logics there are relatively efficient algorithms for automaticaly checking whether certain kinds of temporal formulae hold of a transition system. This is a very active and rapidly-changing research area, and it probably provides the most successful applications of formal methods for hardware to real world problems.

General principles tell us that fully automatic methods are not applicable to all problems, so attempts are in progress to combine human-guided formal verification with model checking and other decision procedure based techniques.

## 4.2 Computation Tree Logic (CTL)

Model checking was introduced by Clarke and Emerson in the early 1980s. The logic they proposed, which is still widely used, is called Computation Tree Logic (CTL). This has the following well-formed formulae (wffs):

$$
\begin{array}{lll}
\mathit{wff} & ::= & \mathit{constant} & \text{(Constant)} \\
 & | & \neg \mathit{wff} & \text{(Negation)} \\
 & | & \mathit{wff}_1 \wedge \mathit{wff}_2 & \text{(Conjunction)} \\
 & | & \mathit{wff}_1 \vee \mathit{wff}_2 & \text{(Disjunction)} \\
 & | & \mathit{wff}_1 \Rightarrow \mathit{wff}_2 & \text{(Implication)} \\
 & | & \mathbf{AX}\mathit{wff} & \text{(All successors)} \\
 & | & \mathbf{EX}\mathit{wff} & \text{(Some successors)} \\
 & | & \mathbf{A}[\mathit{wff}_1\ \mathbf{U}\ \mathit{wff}_2] & \text{(Until -- along all paths)} \\
 & | & \mathbf{E}[\mathit{wff}_1\ \mathbf{U}\ \mathit{wff}_2] & \text{(Until -- along some path)}
\end{array}
$$

CTL is a *branching time* logic: it allows one to specify that a property hold along all paths (**A**) through a transition system or just some paths (**E**). Time may branch as a result of non-determinism in the model. This can arise in several ways: for example as a result of a partially specified environment (different 'futures' resulting from different inputs) or due to 'internal' non-determinism in the system (e.g. the next-state of the system is not fully specified).

If $\mathcal{R} : \alpha \times \alpha \to \mathit{bool}$ then an $\mathcal{R}$-path is an infinite sequence of states represented as a function $\sigma : \mathit{num} \to \alpha$ such that: $\forall t.\ \mathcal{R}(\sigma(t), \sigma(t{+}1))$ (i.e. successive state are related by $R$).

Let $\mathsf{Path}(\mathcal{R}, s)\sigma$ be true if $\sigma$ is an $\mathcal{R}$-path starting from $s$, i.e.:

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

CTL wffs denote predicates on pairs $(\mathcal{R}, s)$, where $\mathcal{R}$ is a transition relation and $s$ a state. Thus define:

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

## 4.3 Using CTL

Operators **AF** and **EF** ("F" is mnemonic for "Future") are defined using the constant $\mathsf{T}$ ('true') by:

$$\mathbf{AF}P\ = \mathbf{A}[\mathsf{T}\ \mathbf{U}\ P]$$

$$\mathbf{EF}P\ = \mathbf{E}[\mathsf{T}\ \mathbf{U}\ P]$$

$\mathbf{AF}P$ is true if $P$ holds somewhere along every $\mathcal{R}$-path – $P$ is *inevitable*.

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

$\mathbf{EF}P$ is true if $P$ holds somewhere along some $\mathcal{R}$-path – $P$ *potentially* holds.

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

Operators $\mathbf{AG}$ and $\mathbf{EG}$ (“G” is mnemonic for “Global”) are defined by:

$$\mathbf{AG}P \;= \neg\mathbf{EF}(\neg P)$$

$$\mathbf{EG}P \;= \neg\mathbf{AF}(\neg P)$$

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

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

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

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

$\mathbf{A}[P\mathbf{W}Q]$ is true if along every path either $P$ always holds, or $Q$ holds sometime, and until it does $P$ holds.[2]

[2]The notation $\mathbf{A}[P\mathbf{W}Q]$ is not standard.

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

$\mathbf{E}\,\texttt{[}P\mathbf{W}Q\texttt{]}$ is true if along some path either $P$ always holds, or $Q$ holds sometime, and until it does $P$ holds.[3]

[3] The notation $\mathbf{E}\,\texttt{[}P\mathbf{W}Q\texttt{]}$ is not standard.

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

## 4.4 Examples of CTL formulae

Here are some typical CTL formulae:

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

  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*.

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

  No acknowledgement *Ack* if no request *Req* on previous cycle.

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

  *DeviceEnabled* holds infinitely often.

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

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

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

  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.

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

  Whenever *Req* is true and *Ack* is false then *Ack* will eventually become true and until it does *Req* will remain true.

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

  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 $\mathit{Req}_1$ and $\mathit{Req}_2$ are false, they remain false until *Start* becomes true with $\mathit{Req}_2$ 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.

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

  $P$ 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.

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

  One cycle after *Req*, $\mathit{Ack}_1$ should become true, and then $\mathit{Ack}_2$ becomes true 1 to 6 cycles later and then eventually *Respond* becomes true, but until it does *Waiting* holds from the time of $\mathit{Ack}_2$.

## 4.5 CTL model checking algorithm

Call $\mathcal{R}$ computable if for any particular $s$ the set of successor states $\{s' \mid \mathcal{R}(s, s')\}$ is finite and computable.

Call $P$ *checkable* for $\mathcal{R}$ if for any particular state $s$ it is decidable whether $P(\mathcal{R}, s)$ is true.

Clearly if $P$ and $Q$ are checkable for $\mathcal{R}$, then so are $\neg P$, $P \wedge Q$, $P \vee Q$ and $P \Rightarrow Q$. If $\mathcal{R}$ is computable then $\mathbf{AX}P$ and $\mathbf{EX}P$ are also checkable for $\mathcal{R}$.

If the set of states is finite, $\mathcal{R}$ is computable and $P$ and $Q$ are checkable for $\mathcal{R}$, then Clarke and Emerson showed that $\mathbf{A}[P\ \mathbf{U}\ Q]$ and $\mathbf{E}[P\ \mathbf{U}\ Q]$ are also checkable for $\mathcal{R}$.

Consider first $\mathbf{A}[P\ \mathbf{U}\ Q]$. Define $\mathbf{A}[P\ \mathbf{U}\ Q]_i$ inductively by:

$$
\begin{array}{ll}
\mathbf{A}[P\ \mathbf{U}\ Q]_0 & = Q \\
\mathbf{A}[P\ \mathbf{U}\ Q]_{i+1} & = \mathbf{A}[P\ \mathbf{U}\ Q]_i \vee (P \wedge \mathbf{AX}(\mathbf{A}[P\ \mathbf{U}\ Q]_i))
\end{array}
$$

then $\mathbf{A}[P\ \mathbf{U}\ Q]_i(\mathcal{R}, s)$ is true if along all $R$-paths of length $i{+}1$ or less there is a point at which $Q$ is true and at all earlier points of the path $P$ is true.

Thus $\mathbf{A}[P\ \mathbf{U}\ Q](\mathcal{R}, s) \ = \ \exists i.\ \mathbf{A}[P\ \mathbf{U}\ Q]_i(\mathcal{R}, s)$.

If $Q$ is is checkable for $\mathcal{R}$, then $\{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_0(\mathcal{R}, s)\} \ = \ \{s \mid Q(R, s)\}$ can be computed.

Suppose $\{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_i(\mathcal{R}, s)\}$ has been computed, then $\{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_{i+1}(\mathcal{R}, s)\}$ consists of all states $s$ in $\{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_i(\mathcal{R}, s)\}$ plus all states $s$ such that $P(\mathcal{R}, s)$ and all of whose successors are in $\{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_i(\mathcal{R}, s)\}$. This set can be computed if $P$ is checkable for $\mathcal{R}$ and $\mathcal{R}$ is computable.

Thus an increasing sequence of sets of states can be computed:

$$
\{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_0(\mathcal{R}, s)\} \ \subseteq \ \{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_1(\mathcal{R}, s)\} \ \subseteq \ \{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_2(\mathcal{R}, s)\} \ \subseteq \ \cdots
$$

If the set of all states is finite then this sequence cannot continue to increase indefinitely.

Thus there is some $i$ such that $\{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_i(\mathcal{R}, s)\} \ = \ \{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_{i+1}(\mathcal{R}, s)\}$, and then:

$$\begin{aligned}
&\{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_{i+2}(\mathcal{R},s)\}\\
&\quad = \{s \mid (\mathbf{A}[P\ \mathbf{U}\ Q]_{i+1} \lor (P \land \mathbf{AX}(\mathbf{A}[P\ \mathbf{U}\ Q]_{i+1}))(\mathcal{R},s))\}\\
&\quad = \{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_{i+1}(\mathcal{R},s)\ \lor\ (P(\mathcal{R},s)\ \land\ \mathbf{AX}(\mathbf{A}[P\ \mathbf{U}\ Q]_{i+1})(\mathcal{R},s))\}\\
&\quad = \{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_{i+1}(\mathcal{R},s)\}\ \cup\ (\{s \mid P(\mathcal{R},s)\}\ \cap\ \{s \mid \mathbf{AX}(\mathbf{A}[P\ \mathbf{U}\ Q]_{i+1})(\mathcal{R},s)\}\\
&\quad = \{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_{i+1}(\mathcal{R},s)\}\\
&\qquad \cup\ (\{s \mid P(\mathcal{R},s)\}\ \cap\ \{s \mid \forall s'.\ \mathcal{R}(s,s')\ \Rightarrow\ \mathbf{A}[P\ \mathbf{U}\ Q]_{i+1}(\mathcal{R},s')\}\\
&\quad = \{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_{i+1}(\mathcal{R},s)\}\\
&\qquad \cup\ (\{s \mid P(\mathcal{R},s)\}\ \cap\ \{s \mid \forall s'.\ \mathcal{R}(s,s')\ \Rightarrow\ s' \in \{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_{i+1}(\mathcal{R},s)\}\}\\
&\quad = \{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_{i}(\mathcal{R},s)\}\\
&\qquad \cup\ (\{s \mid P(\mathcal{R},s)\}\ \cap\ \{s \mid \forall s'.\ \mathcal{R}(s,s')\ \Rightarrow\ s' \in \{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_{i}(\mathcal{R},s)\}\}\\
&\quad = \{s \mid (\mathbf{A}[P\ \mathbf{U}\ Q]_{i} \lor (P \land \mathbf{AX}(\mathbf{A}[P\ \mathbf{U}\ Q]_{i}))(\mathcal{R},s))\}
\end{aligned}$$

and hence $\{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_{j}(\mathcal{R},s)\}\ =\ \{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_{i}(\mathcal{R},s)\}$ for all $j \geq i$.
Thus

$$\{s \mid \mathbf{A}[P\ \mathbf{U}\ Q](\mathcal{R},s)\}\ =\ \{s \mid \mathbf{A}[P\ \mathbf{U}\ Q]_{i}(\mathcal{R},s)\}$$

hence $\mathbf{A}[P\ \mathbf{U}\ Q]$ is checkable for $\mathcal{R}$.
A similar argument shows that $\mathbf{E}[P\ \mathbf{U}\ Q]$ is checkable for $\mathcal{R}$ too. Define:

$$\begin{aligned}
\mathbf{E}[P\ \mathbf{U}\ Q]_0 &= Q\\
\mathbf{E}[P\ \mathbf{U}\ Q]_{i+1} &= \mathbf{E}[P\ \mathbf{U}\ Q]_i \lor (P \land \mathbf{EX}(\mathbf{E}[P\ \mathbf{U}\ Q]_i))
\end{aligned}$$

and proceed as before.

## 4.6 An example

Recall that:

```
∀dreq dack.
RECEIVER(dreq,dack) =
(∃q0.
  (∀t. q0(t + 1) = dreq t) ∧
  (∀t. dack(t + 1) =
        dreq t ∧ (q0 t ∨ (¬q0 t ∧ dack t))))
```

This corresponds to the transition system:

```
(dreq, q0, dack)  --->  (dreq', dreq, dreq ∧ (q0 ∨ (¬q0 ∧ dack)))
```

The transition relation, $\mathcal{R}_{\texttt{RECEIVER}}$ say, corresponding to this transition system is:

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

which is equivalent to:

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

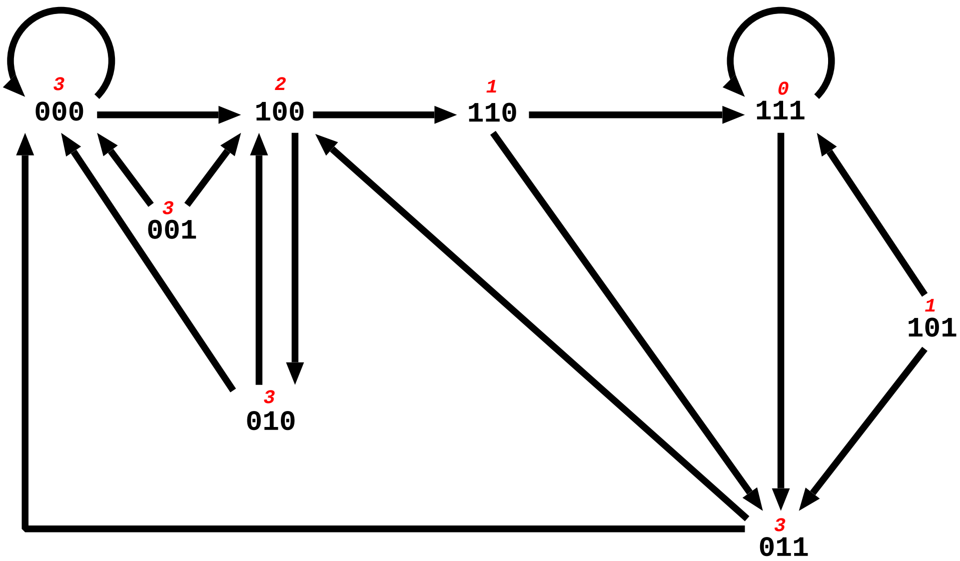
The vector of primed variables `(dreq',q0',dack')` represent the 'next state'. Note that this relation is non-deterministic, because the value of `dreq'` is not specified.
As an example formula to check consider $\mathbf{EF}(\texttt{dreq} \wedge \texttt{q0} \wedge \texttt{dack})$ with respect to $\mathcal{R}_{\texttt{RECEIVER}}$.
First recall that in general:

$$\mathbf{EF}P \quad = \mathbf{E}[\mathsf{T}\ \mathbf{U}\ P]$$

so $\{s \mid \mathbf{EF}P(\mathcal{R}, s)\}$ is computed by iteratively generating:

$$\mathcal{S}_0 \quad = \{s \mid P(\mathcal{R}, s)\}$$
$$\mathcal{S}_{i+1} \quad = \mathcal{S}_i\ \cup\ \{s \mid \exists s'.\ \mathcal{R}(s, s')\ \wedge\ s' \in \mathcal{S}_i\}$$

For the `RECEIVER` example the set of possible states is:
$\{000, 001, 010, 011, 100, 101, 110, 111\}$
where $b_2b_1b_0$ denotes the state $\texttt{dreq} = b_2\ \wedge\ \texttt{q0} = b_1\ \wedge\ \texttt{dack} = b_0$.
The graph of the transition relation is:




A number $i$ above a state indicates that the state is in the set $\mathcal{S}_i$ defined below.

To check $\mathbf{EF}(\mathtt{dreq} \land \mathtt{q0} \land \mathtt{dack})$ take
$P(\mathcal{R}_{\mathtt{RECEIVER}}, b_2b_1b_0) \ = \ b_2 \ \land \ b_1 \ \land \ b_0$
and hence:

$$
\begin{aligned}
\mathcal{S}_0 \quad &= \{b_2b_1b_0 \mid P(\mathcal{R}_{\mathtt{RECEIVER}}, b_2b_1b_0)\} \\
\mathcal{S}_{i+1} \quad &= \mathcal{S}_i \ \cup \ \{s \mid \exists b_2'b_1'b_0'.\ \mathcal{R}(b_2b_1b_0, b_2'b_1'b_0') \ \land \ b_2'b_1'b_0' \in \mathcal{S}_i\} \\
&= \mathcal{S}_i \ \cup \ \{b_2b_1b_0 \mid \exists b_2'b_1'b_0'.\ (b_1' = b_2) \ \land \ (b_0' = b_2 \land (b_1 \lor b_0)) \ \land \ b_2'b_1'b_0' \in \mathcal{S}_i\}
\end{aligned}
$$

Thus:

$$
\begin{aligned}
\mathcal{S}_0 \ &= \{111\} \\
\mathcal{S}_1 \ &= \{111\} \ \cup \ \{101, 110\} \\
&= \{111, 101, 110\} \\
\mathcal{S}_2 \ &= \{111, 101, 110\} \ \cup \ \{100\} \\
&= \{111, 101, 110, 110\} \\
\mathcal{S}_3 \ &= \{111, 101, 110, 100\} \ \cup \ \{000, 001, 010, 011\} \\
&= \{111, 101, 110, 100, 000, 001, 010, 011\} \\
\mathcal{S}_i \ &= \mathcal{S}_3 \quad (i > 3)
\end{aligned}
$$

This shows that $\forall s.\ \mathbf{EF}(\mathtt{dreq} \land \mathtt{q0} \land \mathtt{dack})(\mathcal{R}_{\mathtt{RECEIVER}}, s)$.

## 4.7 Implementing model checking

The algorithm outlined in Section 4.5 computes the set of states for which a CTL wff holds.

A good representation for this set is a BDD. For example, a set of states

$\{(x_0, y_0, z_0), (x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)\}$

of `RECEIVER` would be represented as the BDD of the formula:

$$
\begin{array}{c}
(\mathtt{dreq} = x_0 \land \mathtt{q0} = y_0 \land \mathtt{dack} = z_0) \lor \\
(\mathtt{dreq} = x_1 \land \mathtt{q0} = y_1 \land \mathtt{dack} = z_1) \lor \\
\vdots \\
(\mathtt{dreq} = x_n \land \mathtt{q0} = y_n \land \mathtt{dack} = z_n)
\end{array}
$$

where `dreq`, `q0` and `dack` are the three state variables and $x_i$, $y_i$ and $z_i$ $(i = 0, 1, \ldots, n)$ are truth value constants (i.e. $\mathsf{T}$ or $\mathsf{F}$).

The transition relation $\mathcal{R}_{\mathtt{RECEIVER}}$ is:

$\mathcal{R}_{\texttt{RECEIVER}}$`((dreq,q0,dack),(dreq',q0',dack')) =`
  `(q0'` $\Leftrightarrow$ `dreq)` $\wedge$ `(dack'` $\Leftrightarrow$ `dreq` $\wedge$ `(q0` $\vee$ `dack))`

This formula can be represented by a BDD containing variables `dreq`, `q0`, `dack`, `dreq'`, `q0'`, `dack'`. In general, if there are $n$ state variables, then the transition relation is represented as the BDD of a formula with $2n$ variables (each variable occurs primed and unprimed).

Suppose $B$ is a BDD representing a set of states for `RECEIVER`, then the formula representing the set of states satisfying **AX** $B$ is:

$\forall$`dreq' q0' dack'`. $\mathcal{R}_{\texttt{RECEIVER}} \Rightarrow B$`[dreq',q0',dack'/dreq,q0,dack]`

The BDD corresponding to $\mathcal{R}_{\texttt{RECEIVER}} \Rightarrow B$`[dreq',q0',dack'/dreq,q0,dack]` is computed using standard BDD methods. Call this $B_1$.

The BDD corresponding to $\forall$`dack'`. $B_1$ is the BDD of

$B_1$`[T/dack']` $\wedge$ $B_1$`[F/dack']`

This is also computed by standard methods. Call this $B_2$. Similarly the BDD, $B_3$ say, corresponding to $\forall$`q0'`. $B_2$ and then the BDD, $B_4$ say, corresponding to $\forall$`dreq'`. $B_3$ can be computed. $B_4$ is then the BDD representing **AX** $B$.

The formula representing the set of states satisfying **EX** $B$ is:

$\exists$`dreq' q0' dack'`. $\mathcal{R}_{\texttt{RECEIVER}} \wedge B$`[dreq',q0',dack'/dreq,q0,dack]`

The BDD corresponding to $\mathcal{R}_{\texttt{RECEIVER}} \wedge B$`[dreq',q0',dack'/dreq,q0,dack]` is computed using standard BDD methods. Call this $B_1$.

The BDD corresponding to $\exists$`dack'`. $B_1$ is the BDD of

$B_1$`[T/dack']` $\vee$ $B_1$`[F/dack']`

This is also computed by standard methods. Call this $B_2$. Similarly the BDD, $B_3$ say, corresponding to $\exists$`q0'`. $B_2$ and then the BDD, $B_4$ say, corresponding to $\exists$`dreq'`. $B_3$ can be computed. $B_4$ is then the BDD representing **EX** $B$.

This method is easily generalised from this example to provide a way of computing the BDDs of **AX** $P$ and **EX** $P$ from the BDDs of $P$ for an arbitrary finite set of states and transition relation $\mathcal{R}$.

The BDD of $\mathbf{A}\texttt{[}P\ \mathbf{U}\ Q\texttt{]}$ is computed by computing the BDDs for $\mathbf{A}\texttt{[}P\ \mathbf{U}\ Q\texttt{]}_i$ for $i = 0, 1, 2, \ldots$. The equality of BDDs is efficient to check (constant time using a hash table), so it can be detected when $\{s \mid \mathbf{A}\texttt{[}P\ \mathbf{U}\ Q\texttt{]}_i(\mathcal{R}, s)\} = \{s \mid \mathbf{A}\texttt{[}P\ \mathbf{U}\ Q\texttt{]}_{i+1}(\mathcal{R}, s)\}$.

The BDD of $\mathbf{E}\texttt{[}P\ \mathbf{U}\ Q\texttt{]}$ is computed analogously.

This method of using BDDs to represent and compute sets of states is called *symbolic model checking* and was independently discovered by several people. The widely used system SMV (by Ken McMillan) uses it.

Note also that BDDs have other uses for hardware verification. They can be used for efficient Boolean equivalence checking of circuits (much more efficient than naive brute-force enumeration). BDDs can also be used to represent transition relations and compute the next state in cycle-based simulation.

## 4.8 State transition systems and reachability

Assume a type of states: type *states* and an initial set of states given by a predicate $\mathcal{B}$:

- $\mathcal{B} \;:\; \mathit{states}{\rightarrow}\mathit{bool}$
- $\mathcal{B}\ s$ means $s$ is an initial state

A state transition relation $\mathcal{R}$ is such that

- $\mathcal{R} \;:\; \mathit{states} \times \mathit{states}{\rightarrow}\mathit{bool}$
- $\mathcal{R}(s, s')$ means $s'$ a successor to $s$

An example of a state transition system is a single machine:

- State transition function: $\delta$

  $\delta \;:\; \mathit{states} \times \mathit{inputs}{\rightarrow}\mathit{states}$
- Define state transition relation:

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

Note that a deterministic machine gives rise to non-deterministic transition relation via existential quantification over inputs.
A more interesting class of state transition systems corresponds to $n$ machines running in parallel.
Assume $n$ state variables:

- $\mathit{states} \;=\; \mathit{states}_1 \times \cdots \times \mathit{states}_n$
- $\vec{v} \;=\; (v_1, \ldots, v_n)$

Assume $n$ transition functions:

$$\delta_i \;:\; \mathit{states} \times \mathit{inputs}{\rightarrow}\mathit{states}_i \qquad (1 \le i \le n)$$

The transition relation of the asynchronous parallel composition of the $n$ machines is defined by:

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

Note that an $\mathcal{R}$-step is one or more $\delta_i$-steps.

The condition for a state $s$ to be reachable in one $\mathcal{R}$-step from a state in $\mathcal{B}$ is:

$$\exists u.\ \mathcal{B}\ u \ \wedge \ \mathcal{R}(u,s)$$

The set of states reachable in at most $n$ steps is defined by primitive recursion by:

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

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

The set of reachable states is then defined as the set of states reachable in some number of steps:

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

A key fixedpoint property is (equality between predicates represents set equality):

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

**Exercise:** Sketch a proof of this fixedpoint property.

### 4.8.1 Using BDDs

Reduced ordered binary decision diagrams (ROBDDs or just BDDs) canonically represent of boolean expressions (Bryant 1986).

Boolean formulae $\mathcal{R}(s,s')$ and $\mathcal{B}\ s$ can be represented as BDDs.

Then one can then iteratively compute:

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

Given BDDs for $\mathcal{S}_i\ s$ and $\mathcal{R}(s,s')$ the BDD of $\exists u.\ \mathcal{S}_i\ u\ \wedge\ \mathcal{R}(u,s)$ is computed by:

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

this can be done efficiently using standard BDD algorithms for substitution, conjunction and existential quantification.

By the fixedpoint property, when $\mathcal{S}_{n+1}\ s = \mathcal{S}_n\ s$ it follows that:

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

hence one can compute the BDD of $\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ \ s$

A typical verification problem is to show some property $\mathcal{Q}$ is true in all reachable states. This can be verified by building the BDD of $\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s\ \Rightarrow\ \mathcal{Q}\ s$ and seeing if it is $\mathsf{T}$.

Suppose $\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s\ \Rightarrow\ \mathcal{Q}\ s$ is false. Then BDDs can be used to find a shortest sequence of state transitions that lead to a reachable state in which $\mathcal{Q}$ fails to hold. This can be very useful for debugging. Perhaps even more useful than verification.

The first step is to find a shortest path to a counterexample. One successively generates BDDs of $\mathsf{ReachBy}\ i\ \mathcal{R}\ \mathcal{B}\ s$ $(i = 0, 1, \ldots)$ checking for each $i$ whether $\mathcal{Q}\ s$ holds, i.e. whether the BDD of $\mathsf{ReachBy}\ i\ \mathcal{R}\ \mathcal{B}\ s\ \Rightarrow\ \mathcal{Q}\ s$ is true.

Eventually one will find a smallest $n$ such that $\mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s\ \Rightarrow\ \mathcal{Q}\ s$ is not true.

Using standard BDD algorithms one can then find a state $s_n$ such that $\mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s_n\ \Rightarrow\ \mathcal{Q}\ s_n$ is false, i.e. $\mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s_n\ \wedge\ \neg(\mathcal{Q}\ s_n)$ is true.

To get a path to $s_n$ we can trace backwards. Define:

$$\begin{array}{l}
\mathsf{Pre}\ \mathcal{R}\ \mathcal{Q}\ s\ =\ \exists s'.\ \mathcal{R}(s,s')\ \wedge\ \mathcal{Q}\ s' \\
\mathsf{Eq}\ s_1\ s_2\ =\ (s_1 = s_2)
\end{array}$$

then use BDD algorithms to get a sequence of states $s_n, \ldots, s_0$, where, given $s_i$, the state $s_{i-1}$ is obtained by using a BDD algorithm to find an $s$ such that:

$$\mathsf{ReachBy}\ (i{-}1)\ \mathcal{R}\ \mathcal{B}\ s\ \wedge\ \mathsf{Pre}\ \mathcal{R}\ (\mathsf{Eq}\ s_i)\ s$$

Note that we have already computed the BDDs for $\mathsf{ReachBy}\ \ i\ \ \mathcal{R}\ \mathcal{B}\ s$ (for $i = 0, 1, \ldots, n$) when searching for $s_n$, so they are already available.

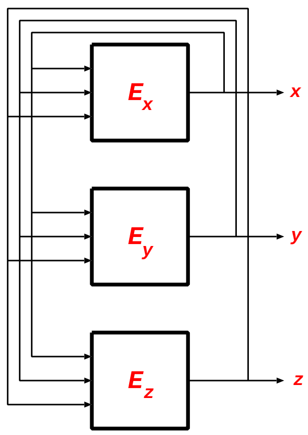
The BDD for $\mathsf{Pre}\ \mathcal{R}\ (\mathsf{Eq}\ s_i)\ s$ is easily computed from the BDDs for $\mathcal{R}$ and $\mathsf{Eq}$ using the BDD algorithms for substitution, conjunction and existential quantification.

### 4.8.2 Early quantification

The BDD of the Boolean formula representing a transition relation $\mathcal{R}(s, s')$ can be very big. It is sometimes possible to construct the BDD of the set of reachable states without having to ever construct the BDD of the transition relation.

In such circumstances, finding paths to counterexamples can also sometimes be done without having to compute the BDD of the transition relation.

Consider three machines running asynchronously in parallel:




This can be modelled by:

$$
\begin{array}{l}
\mathcal{R}((x,y,z),(x',y',z')) = \\
\quad (x' = E_x(x,y,z) \;\wedge\; y' = y \;\wedge\; z' = z) \;\vee \\
\quad (x' = x \;\wedge\; y' = E_y(x,y,z) \;\wedge\; z' = z) \;\vee \\
\quad (x' = x \;\wedge\; y' = y \;\wedge\; z' = E_z(x,y,z))
\end{array}
$$

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

$$
\begin{array}{l}
\exists\overline{x}\ \overline{y}\ \overline{z}.\ \mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}(\overline{x},\overline{y},\overline{z}) \;\wedge\; \mathcal{R}((\overline{x},\overline{y},\overline{z}),(x,y,z)) \\
\quad = \exists\overline{x}\ \overline{y}\ \overline{z}.\ \mathcal{S}(\overline{x},\overline{y},\overline{z}) \;\wedge\; \mathcal{R}((\overline{x},\overline{y},\overline{z}),(x,y,z)) \\
\quad = \exists\overline{x}\ \overline{y}\ \overline{z}.\ \mathcal{S}(\overline{x},\overline{y},\overline{z}) \;\wedge\; ((x = E_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 = E_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 = E_z(\overline{x},\overline{y},\overline{z}))) \\
\quad = (\exists\overline{x}\ \overline{y}\ \overline{z}.\ \mathcal{S}(\overline{x},\overline{y},\overline{z}) \;\wedge\; x = E_x(\overline{x},\overline{y},\overline{z}) \;\wedge\; y = \overline{y} \;\wedge\; z = \overline{z}) \;\vee \\
\qquad (\exists\overline{x}\ \overline{y}\ \overline{z}.\ \mathcal{S}(\overline{x},\overline{y},\overline{z}) \;\wedge\; x = \overline{x} \;\wedge\; y = E_y(\overline{x},\overline{y},\overline{z}) \;\wedge\; z = \overline{z}) \;\vee \\
\qquad (\exists\overline{x}\ \overline{y}\ \overline{z}.\ \mathcal{S}(\overline{x},\overline{y},\overline{z}) \;\wedge\; x = \overline{x} \;\wedge\; y = \overline{y} \;\wedge\; z = E_z(\overline{x},\overline{y},\overline{z})) \\
\quad = ((\exists\overline{x}.\ \mathcal{S}(\overline{x},y,z) \wedge x{=}E_x(\overline{x},y,z)) \;\wedge\; (\exists\overline{y}.\ y{=}\overline{y}) \;\wedge\; (\exists\overline{z}.\ z{=}\overline{z})) \;\vee \\
\qquad ((\exists\overline{x}.\ x{=}\overline{x}) \;\wedge\; (\exists\overline{y}.\ \mathcal{S}(x,\overline{y},z) \wedge y{=}E_y(x,\overline{y},z)) \;\wedge\; (\exists\overline{z}.\ z{=}\overline{z})) \;\vee \\
\qquad ((\exists\overline{x}.\ x{=}\overline{x}) \;\wedge\; (\exists\overline{y}.\ y{=}\overline{y}) \;\wedge\; (\exists\overline{z}.\ \mathcal{S}(x,y,\overline{z}) \wedge z{=}E_z(x,y,\overline{z}))) \\
\quad = (\exists\overline{x}.\ \mathcal{S}(\overline{x},y,z) \;\wedge\; x = E_x(\overline{x},y,z)) \;\vee \\
\qquad (\exists\overline{y}.\ \mathcal{S}(x,\overline{y},z) \;\wedge\; y = E_y(x,\overline{y},z)) \;\vee \\
\qquad (\exists\overline{z}.\ \mathcal{S}(x,y,\overline{z}) \;\wedge\; z = E_z(x,y,\overline{z}))
\end{array}
$$

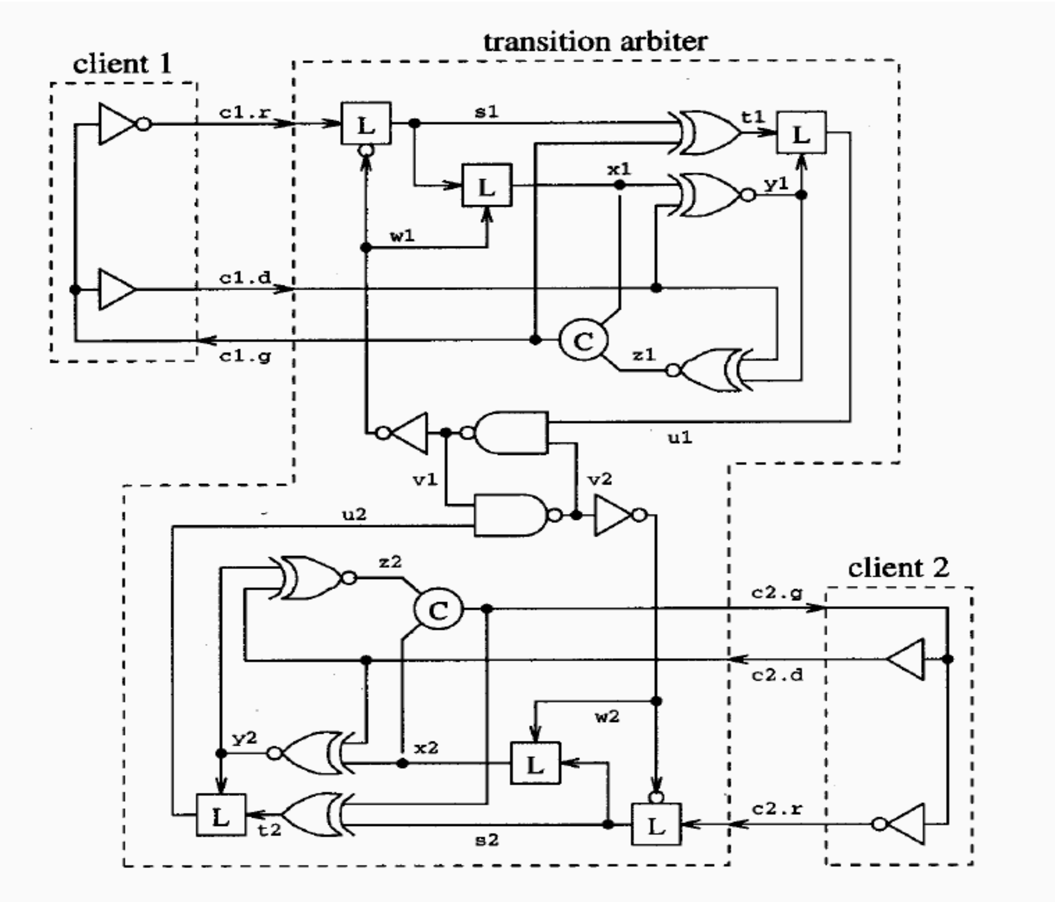
Thus the BDD of $\exists \overline{x}\ \overline{y}\ \overline{z}$. ReachBy $n\ \mathcal{R}\ \mathcal{B}(\overline{x},\overline{y},\overline{z})\ \wedge\ \mathcal{R}((\overline{x},\overline{y},\overline{z}),(x,y,z))$ can be computed without ever computing the BDD of $\mathcal{R}((\overline{x},\overline{y},\overline{z}),(x,y,z))$. This technique is called *disjunctive partitioning* and is an example of *early quantification*.

Disjunctive partitioning can also be used to find paths to counterexamples: Pre $\mathcal{R}\ \mathcal{Q}\ s$ can be deductively simplified, so that its BDD can be computed without having to compute the BDD of the transition relation $\mathcal{R}$.

### 4.8.3 Example

The example here is from:

> Trevor W. S. Lee, Mark R. Greenstreet, Carl-Johan Seger, *Automatic Verification of Asynchronous Circuits*, UBC TR 93-40, November 1993.




This is an asynchronous circuit for mutual exclusion and illustrates the kind of transition relation that benefits from disjunctive partitioning.

It can be in the following states:

$$\begin{array}{lllll} r_i & = g_i & = d_i & \text{Client } i \text{ Idle} \\ \neg r_i & = g_i & = d_i & \text{Client } i \text{ Requesting} \\ \neg r_i & = \neg g_i & = d_i & \text{Client } i \text{ Privileged} \\ r_i & = \neg g_i & = d_i & \text{Client } i \text{ PendingDone} \end{array}$$

The circuit is intended to prevent both clients from being simultaneously Privileged.

Lee/Greenstreet/Seger give two circuits: one correct (mutual exclusion is verified) and one incorrect (a counterexample is generated). I forget whether the circuit above is the correct or incorrect one!

The arbiter can be modelled as parallel compostion of 22 machines. The state (incorrect machine, I think) has 22 components:

$$\begin{array}{l} (r_1, g_1, d_1, s_1, t_1, u_1, v_1, w_1, x_1, y_1, z_1, \\ \ r_2, g_2, d_2, s_2, t_2, u_2, v_2, w_2, x_2, y_2, z_2) \end{array}$$

Representative models of some of the component machines are:

- XOR machine: inputs $s_1$, $g_1$; output $t_1$

  $$\begin{array}{l} (r_1' = r_1) \wedge (g_1' = g_1) \wedge (d_1' = d_1) \wedge (s_1' = s_1) \wedge \\ \boxed{(t_1' \ = \ \neg(s_1 = g_1))} \wedge (u_1' = u_1) \wedge (v_1' = v_1) \wedge (w_1' = w_1) \wedge \\ (x_1' = x_1) \wedge (r_2' = r_2) \wedge (g_2' = g_2) \wedge (d_2' = d_2) \wedge (s_2' = s_2) \\ \wedge (t_2' = t_2) \wedge (u_2' = u_2) \wedge (v_2' = v_2) \wedge (w_2' = w_2) \wedge (x_2' = x_2) \end{array}$$

- LATCH machine: inputs $w_1$, $s_1$; output $x_1$

  $$\cdots \wedge \boxed{(x_1' \ = \ \texttt{IF } w_1 \texttt{ THEN } s_1 \texttt{ ELSE } x_1)} \wedge \cdots$$

- MULLER machine: inputs $x_1$, $z_1$; output $g_1$

  $$\cdots \wedge \boxed{(g_1' \ = \ \texttt{IF } x_1 = z_1 \texttt{ THEN } x_1 \texttt{ ELSE } g_1)} \wedge \cdots$$

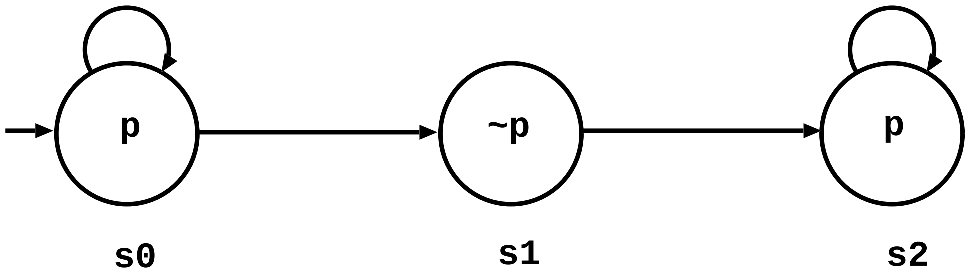
The arbiter is 22 machines $\vee$-ed together.

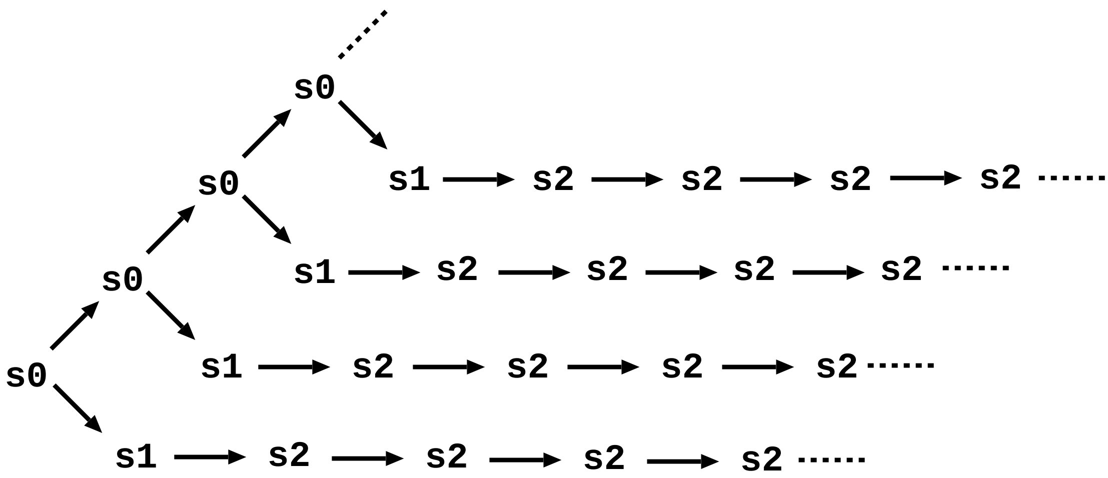
## 4.9 Expressibility of CTL

Consider the property

> *on every path there is a point after which* **p** *is always true*

This cannot be expressed in CTL. We would need something like **AF** $P$ where $P$ is something like " property **p** *true from now on*" but $P$ must start with a path quantifier **A** or **E** so cannot talk about current path, only about all or some paths. Consider the following model:




The property is true, but **AF AG p** is false!

### 4.9.1 Linear Temporal Logic (LTL)

A CTL property is a predicate on a tree: $P(\mathcal{R}, s)$, but an LTL property is a predicate on a path: $P(\sigma)$.

The syntax of LTL well-formed formulae is:

$$
\begin{array}{lcll}
\mathit{wff} & ::= & \mathsf{Atom}(p) & \text{(Atomic formula)} \\
 & | & \neg \mathit{wff} & \text{(Negation)} \\
 & | & \mathit{wff}_1 \vee \mathit{wff}_2 & \text{(Disjunction)} \\
 & | & \mathbf{X}\mathit{wff} & \text{(successor)} \\
 & | & \mathbf{F}\mathit{wff} & \text{(sometimes)} \\
 & | & \mathbf{G}\mathit{wff} & \text{(always)} \\
 & | & [\mathit{wff}_1\ \mathbf{U}\ \mathit{wff}_2] & \text{(Until)}
\end{array}
$$

Define

$$\text{Tail } m\ (\sigma)\ =\ \lambda n.\ \sigma(n{+}m)$$

then a semantics of LTL via a shallow embedding in higher order logic is given by.

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

**Example:**

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

The formula $\mathbf{FG}P$ is true if there is a point after which $P$ is always true

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

Thus $\mathbf{FG}P$ holds for our example, and hence LTL can express things that CTL can't express. However, CTL can express things that LTL can't express. For example,

$\mathbf{AG}(\mathbf{EF}\ P)$ says:

> *from every state it is possible to get to a state for which $P$ holds*

but we can't say this in LTL.

Consider disjunction like $\mathbf{FG}P \vee \mathbf{AG}(\mathbf{EF}\ P)$, for example:

*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*

One can't express this in either CTL or LTL! The temporal logic CTL* combines CTL and LTL and can express this property. CTL* has two kinds of formulas: state formulas (*swff*) and path formulas (*pwff*): state formulas are true of a tree $(\mathcal{R}, s)$ and path formulas are true of a path $\sigma$.

The following syntax is mutually recursive

| | | | |
|---|---|---|---|
| *swff* | ::= | $\mathsf{Atom}(p)$ | (Atomic formula) |
| | \| | $\neg swff$ | (Negation) |
| | \| | $swff_1 \vee swff_2$ | (Disjunction) |
| | \| | $\mathbf{A}pwff$ | (All paths) |
| | \| | $\mathbf{E}pwff$ | (Some paths) |

| | | | |
|---|---|---|---|
| *pwff* | ::= | $\mathsf{PathForm}(swff)$ | (Every state formula is a path formula) |
| | \| | $\neg pwff$ | (Negation) |
| | \| | $pwff_1 \vee pwff_2$ | (Disjunction) |
| | \| | $\mathbf{X}pwff$ | (Successor) |
| | \| | $\mathbf{F}pwff$ | (Sometimes) |
| | \| | $\mathbf{G}pwff$ | (Always) |
| | \| | $[pwff_1\ \mathbf{U}\ pwff_2]$ | (Until) |

CTL is CTL* restricted with $\mathbf{X}$, $\mathbf{F}$, $\mathbf{G}$, $[-\mathbf{U}-]$ preceded by $\mathbf{A}$ or $\mathbf{E}$.

LTL consists of CTL* formulas of form $\mathbf{A}pwff$, where the only state formulas in *pwff* are atomic. The selection of primitives above is arbitrary: $\vee$, $\neg$, $\mathbf{X}$, $\mathbf{U}$, $\mathbf{E}$ would be enough.

The semantics of CTL* combines the state semantics of CTL with the path semantics of LTL:

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

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

Note that the semantics of state and path formulas have different types.

The semantics is more readable if we assume $\mathcal{R}$ fixed. Let $\mathsf{Path}\ s\ \sigma$ abbreviate $\mathsf{Path}(\mathcal{R}, s)\sigma$, then:

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

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

**Fairness:** Often one wants to assume a component or the environment is 'fair'.

**Example 1:** A fair arbiter

- the arbiter doesn't ignore one of its requests forever;
- not every request need be granted, but we don't want an infinite number of requests from one source and no grant to that source.

**Example 2:** A reliable channel

- no message continuously transmitted but never received;
- not every message need be received, but we don't want an infinite number of sends and no receive.

To ensure fairness we may want to require, for example, that on all paths if a property, say $P$, holds infinitely often then some other property, say $Q$, also holds infinitely often on the path. In LTL (or CTL*) this is expressible as $\mathbf{A}(\mathbf{G}(\mathbf{F}\ P)\ \Rightarrow\ \mathbf{G}(\mathbf{F}\ Q))$, but we can't say this in CTL (why not – what's wrong with $\mathbf{AG}(\mathbf{AF}\ P)\ \Rightarrow\ \mathbf{AG}(\mathbf{AF}\ Q)$ ?).

Fair CTL model checking is implemented in the model checking algorithm, but with LTL fairness can be expressed in the logic. This is an argument in favour of LTL. Fairness is a tricky and subtle subject: there are several notions or fairness: 'weak fairness', 'strong fairness' etc. There are whole books on fairness (`www.amazon.com/exec/obidos/ISBN=0387962352/`)!

## 4.10 Propositional modal $\mu$-calculus

The $\mu$-calculus has fixed-point operators for both maximal and minimal fixed points. Model checking consists of calculating fixed points, as we have seen with CTL. Many logics (e.g. CTL*) can be translated into the $\mu$-calculus, however the $\mu$-calculus is very non-intuitive to use! It is used as 'intermediate code' rather than as a practical property language.

The $\mu$-calculus is strictly stronger than CTL*: to represent CTL* in it one needs fixed point operators nested two deep. However, the expressibility of the $\mu$-calculus strictly increases as allowed nesting increases.

## 4.11 Interval Temporal Logic (ITL)

Interval Temporal Logic (ITL) specifies properties of intervals, where an interval is a sequence of states with a beginning and an end. Intervals are useful for talking about 'transactions'. ITL has an executable subset called *Tempura* suitable for simulation. It was developed by Ben Moszkowski and others at Stanford then here at Cambridge. Moszkowski is now at De Montford University.

The syntax of ITL (simplified and with expressions omitted) is:

$$
\begin{array}{lcll}
\mathit{wff} & ::= & \mathsf{Atom}(p) & \text{(Atomic formula)} \\
 & \mid & \mathbf{true} & \text{(Truth)} \\
 & \mid & \neg \mathit{wff} & \text{(Negation)} \\
 & \mid & \mathit{wff}_1 \vee \mathit{wff}_2 & \text{(Disjunction)} \\
 & \mid & \mathbf{skip} & \text{(interval with exactly two states)} \\
 & \mid & \mathit{wff}_1 \ \boldsymbol{;}\ \mathit{wff}_2 & \text{(Chop)} \\
 & \mid & \mathit{wff}* & \text{(Repeat)}
\end{array}
$$

The semantics is:

$$
\begin{array}{ll}
\mathsf{Atom}(p) & = \lambda\langle s_0 \cdots s_n\rangle.\ p(s_0) \ \wedge\ n = 0 \\
\mathbf{true} & = \lambda\langle s_0 \cdots s_n\rangle.\ T \\
\neg P & = \lambda\langle s_0 \cdots s_n\rangle.\ \neg(P\langle s_0 \cdots s_n\rangle) \\
P \vee Q & = \lambda\langle s_0 \cdots s_n\rangle.\ P\langle s_0 \cdots s_n\rangle \ \vee\ Q\langle s_0 \cdots s_n\rangle \\
\mathbf{skip} & = \lambda\langle s_0 \cdots s_n\rangle.\ n = 1 \\
P \ \boldsymbol{;}\ Q & = \lambda\langle s_0 \cdots s_n\rangle.\ \exists k.\ k \leq n \ \wedge\ P\langle s_0 \cdots s_k\rangle \ \wedge\ Q\langle s_k \cdots s_n\rangle \\
P* & = \lambda\langle s_0 \cdots s_n\rangle. \\
 & \qquad \exists w_1 \ \cdots\ w_l.\ \langle s_0 \cdots s_n\rangle \ = w_1 \cdots w_l \ \wedge\ P\ w_1 \ \wedge\ \cdots\ \wedge\ P\ w_l
\end{array}
$$

Here are some examples of ITL.

| **Formula** | **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 |
| $\neg(\mathbf{true}\,;\,\neg\mathbf{P})$ | $P$ always true |

## 4.12 Accellera Property Specification Language (PSL)

*Sugar 1* is the property language of IBM's RuleBase model checker. It is CTL plus *Sugar Extended Regular Expressions* (SEREs). SEREs are ITL-like constructs. The *Accellera* organisation ran a competition to select a 'standard' property language. The finalists were IBM's *Sugar 2* and Motorola's *CBV*. *Sugar 2* won. It is based on LTL rather than CTL but has (optional) CTL constructs and clocking constructs for temporal abstraction. Recently Sugar was renamed by Accellera to PSL (Property Specification Language).

Here is some PSL/*Sugar* notation and the corresponding standard notation.

| **Standard notation** | PSL/*Sugar* **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 not negation)` |
| $\mathbf{G}P$ | `always` $P$ | |
| $[P\ \mathbf{U}\ Q]$ | $P$ `until!` $Q$ | |
| $[P\ \mathbf{W}\ Q]$ | $P$ `until` $Q$ | |
| $\mathbf{skip}$ | `true` | |
| $R*$ | $R$`[*]` | |
| $R_1\ ;\ R_2$ | $R_1\ :\ R_2$ | |
| $R_1\ ;\mathbf{skip}\,;\ R_2$ | $R_1\ ;\ R_2$ | |

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 |

The main innovation of Sugar is the Sugar Extended Regular Expressions (SEREs). These provide facilities similar to to ITL – but weaker (FSM decidable).

Following a 'deIBMisation' of Sugar by the Accellera organisation, "SERE" now stands for "Sequential Extended Regular Expression".

PSL/Sugar provides a rich set of notations for expressing properties compactly. Here is an example taken from an IBM document on Sugar. Compare:

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

with the following that uses "`r1 |-> r2`" which means that "whenever SERE `r1` is matched then, starting on the last state that matched `r1`, the SERE `r2` should match".

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

The following shows two ways to modify this so that the two cycles of `ackin` start the cycle after `!abortin`.

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

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

PSL/Sugar has many definitional extensions, for example:

```
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}
```

Here is a version of this example with a fixed number of non-consecutive repetitions:

> *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`

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

With another extension:

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

we can write a nicer representation:

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

Here's a version of our example with a variable number of non-consecutive repetitions:

> *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`

With yet another extension:

```
b[= i..j] = {b[= i]} | {b[= (i+1)]} | ... | {b[= j]}
```

this becomes:

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

The syntax of core SEREs is:

| | | | |
|---|---|---|---|
| $r$ | $::=$ | $\mathsf{Atom}(p)$ | (Atomic formula) |
| | $\mid$ | $r_1$ `\|` $r_2$ | (Disjunction) |
| | $\mid$ | $r_1$ `;` $r_2$ | (Concatenation) |
| | $\mid$ | $r_1$ `:` $r_2$ | (Fusion: ITL's chop) |
| | $\mid$ | $r_1$ `&&` $r_2$ | (Length matching conjunction) |
| | $\mid$ | $r_1$ `&` $r_2$ | (Flexible matching conjunction) |
| | $\mid$ | $r[*]$ | (Repeat) |

The numerous extensions are "syntactic sugar" (i.e. definitional extensions) of the core.

The semantics is straightforward ($s$ ranges over states; $w$ ranges over finite lists of states; "**head**" denotes head of a list; $|w|$ denotes the length; infix "." denotes concatenation).

$$
\begin{array}{lll}
\mathsf{Atom}(p) & = \lambda w.\ p(\mathbf{head}\ w) \wedge |w| = 1 \\
r_1 \mid r_2 & = \lambda w.\ r_1\ w\ \vee\ r_2\ w \\
r_1\ ;\ r_2 & = \lambda w.\ \exists w_1\ w_2.\ w = w_1.w_2\ \wedge\ r_1\ w_1\ \wedge\ r_2\ w_2 \\
r_1\ :\ r_2 & = \lambda w.\ \exists w_1\ s\ w_2.\ w = w_1.s.w_2\ \wedge\ r_1(w_1.s)\ \wedge\ r_2(s.w_2) \\
r_1\ \&\&\ r_2 & = \lambda w.\ r_1\ w\ \wedge\ r_2\ w \\
r_1\ \&\ r_2 & = \lambda w.\ \exists w_1\ w_2.\ w = w_1.w_2\ \wedge\ (r_1\ w\ \wedge\ r_2\ w1)\ \vee\ (r_2\ w\ \wedge\ r_1\ w_1) \\
r[*] & = \lambda w.\ w = \langle\rangle \vee \exists w_1\ \cdots\ w_l.\ w\ = w_1.\cdots.w_l\ \wedge\ r\ w_1\ \wedge\ \cdots\ \wedge\ r\ w_l
\end{array}
$$

SEREs can be combined with LTL formulas in various ways. For example, formula $\{r\}f$ means LTL formula $f$ true after SERE $r$. Here's an 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.*

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

The syntax of PSL/Sugar formulas is below:

$$
\begin{array}{lll}
f ::= & \mathsf{Atom}(p) & \text{(Atomic formula)} \\
\mid & \neg f & \text{(Negation)} \\
\mid & f_1 \vee f_2 & \text{(Disjunction)} \\
\mid & \mathbf{next} f & \text{(successor)} \\
\mid & \{r\}(f) & \text{(Suffix implication)} \\
\mid & \{r_1\}\ \texttt{|->}\ \{r_2\}! & \text{(Strong suffix next implication)} \\
\mid & [f_1\ \texttt{until}\ f_2] & \text{(Until)}
\end{array}
$$

The semantics (simplified by omitting clocking) is:

$$
\begin{array}{ll}
\mathsf{Atom}(p) & = \lambda\sigma.\ p(\sigma(0)) \\
\neg f & = \lambda\sigma.\ \neg(f\ \sigma) \\
f_1 \vee f_2 & = \lambda\sigma.\ f_1\ \sigma\ \vee\ f_2\ \sigma \\
\texttt{next} f & = \lambda\sigma.\ f(\mathrm{Tail}\ 1\ (\sigma)) \\
\{r\}(f) & = \lambda\sigma.\ \exists w\ \sigma'.\ \sigma = w.\sigma'\ \wedge\ r\ w\ \wedge\ f\ \sigma' \\
\{r_1\}\ \texttt{|->}\ \{r_2\}! & = \lambda\sigma.\ \exists w_1\ \sigma'.\ \sigma = w_1.\sigma'\ \wedge\ r_1\ w_1\ \Rightarrow\ \exists w_2\ \sigma''.\ \sigma' = w_2.\sigma''\ \wedge\ r_2\ w_2 \\
[f_1\ \texttt{until}\ f_2] & = \lambda\sigma.\ \exists i.\ f_2(\mathrm{Tail}\ i\ \sigma)\ \wedge\ \forall j.\ j < i\ \Rightarrow\ f_1(\mathrm{Tail}\ j\ \sigma)
\end{array}
$$

There is also an Optional Branching Extension (OBE), which is completely standard CTL: **EX**, **E**[–**U**–], **EG**

PSL/Sugar allows SEREs and formulas to be 'clocked'. The basic idea is that $r$`@clk`, $f$`@clk` abstracts $r$, $f$ on rising edges of `clk`, respectively. There can be several clocks. The official semantics is complex due to clocking, and is still evolving. There is an alternative equivalent semantics that 'translates away' clocks by pushing `@clk` inwards, e.g. $b$`@clk = {!clk[*];clk & `$b$`}`. Thus one really only need consider unclocked PSL.

PSL/Sugar is intended for both model checking (static verification) and use in simulation (dynamic verification).

For both these SEREs are checked by generating finite automata called "satellites".

For model checking, standard LTL and CTL methods can be used, linked to the satellite automata (details omitted). For dynamic checking, HDL checkers are generated and added to the model being simulated. IBM have a product called FoCs that does this.

## 4.13 Examples of CTL formulae revisited

The CTL examples in Section 4.4 can be formulated in PSL/Sugar. Some examples are essentially branching time properties and can't be reformulated in linear time LTL, however, PSL/Sugar has CTL in its Optional Branching Extension (OBE) and includes the usual operators like **EF**.

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

  **CTL:** $\mathbf{EF}(\mathit{Started} \wedge \neg \mathit{Ready})$
  **PSL:** $\mathbf{EF}(\mathit{Started} \wedge \neg \mathit{Ready})$

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

  **CTL:** $\mathbf{AG}(\mathit{Req} \Rightarrow \mathbf{AF}\mathit{Ack})$
  **PSL:** `always(`$\mathit{Req}$ `->` `eventually!` $\mathit{Ack}$`)`

- "No acknowledgement *Ack* if no request *Req* on previous cycle."

  **CTL:** $\mathbf{AG}(\neg \mathit{Req} \Rightarrow \mathbf{AX}\neg \mathit{Ack})$
  **PSL:** `always(!`$\mathit{Req}$ `->` `next(!`$\mathit{Ack}$`))`

- "*DeviceEnabled* holds infinitely often."

  **CTL:** $\mathbf{AG}(\mathbf{AF}\ \mathit{DeviceEnabled})$
  **PSL:** `always(eventually!` $\mathit{DeviceEnabled}$`)`

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

  **CTL:** $\mathbf{AG}(\mathbf{EF}\mathit{Restart})$
  **PSL:** $\mathbf{AG}(\mathbf{EF}\mathit{Restart})$

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

  **CTL:** $\mathbf{AG}(Req \Rightarrow \mathbf{A}\,\mathtt{[}Req\ \mathbf{U}\ Ack\mathtt{]})$
  **PSL:** $\mathtt{always(}Req\ \mathtt{->}\ \mathtt{(}Req\ \mathtt{until!}\ Ack\mathtt{))}$

- "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."

  **CTL:** $\mathbf{AG}(Req \Rightarrow \mathbf{AX}(\mathbf{A}\,\mathtt{[}\neg Req\ \mathbf{U}\ Ack\mathtt{]}))$
  **PSL:** $\mathtt{always(}Req\ \mathtt{->}\ \mathtt{next(!}Req\ \mathtt{until!}\ Ack\mathtt{))}$

- "Whenever *Req* is true and *Ack* is false then *Ack* will eventually become true and until it does *Req* will remain true."

  **CTL:** $\mathbf{AG}(Req \Rightarrow (\neg Ack \Rightarrow \mathbf{AX}(\mathbf{A}\,\mathtt{[}Req\ \mathbf{U}\ Ack\mathtt{]})))$
  **PSL:** $\mathtt{always(}Req\ \mathtt{->}\ \mathtt{(!}Ack\ \mathtt{->}\ \mathtt{next(!}Req\ \mathtt{until!}\ Ack\mathtt{)))}$

- "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."

  **CTL:** $\mathbf{AG}\,\mathtt{[}Enabled \Rightarrow \mathbf{AG}\,\mathtt{[}Start \Rightarrow \mathbf{A}\,\mathtt{[}\neg Waiting\ \mathbf{U}\ Ack\mathtt{]]]}$
  **PSL:** $\mathtt{always(}Enabled\ \mathtt{->}\ \mathtt{always(}Start\ \mathtt{->}\ \mathtt{(!}Waiting\ \mathtt{until!}\ Ack\mathtt{)))}$

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

  **CTL:** $\mathbf{AG}\,\mathtt{[}\neg Req_1 \wedge \neg Req_2 \Rightarrow \mathbf{A}\,\mathtt{[}\neg Req_1 \wedge \neg Req_2\ \mathbf{U}\ (Start \wedge \neg Req_2)\mathtt{]]}$
  **PSL:** $\mathtt{always(!}Req_1\ \mathtt{\&}\ \mathtt{!}Req_2\ \mathtt{->}\ \mathtt{((!}Req_1\ \mathtt{\&}\ \mathtt{!}Req_2\mathtt{)}\ \mathtt{until!}\ \mathtt{(}Start\ \mathtt{\&}\ \mathtt{!}Req_2\mathtt{)))}$

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

  **CTL:** $\mathbf{AG}\,\mathtt{[}Req \Rightarrow \mathbf{AX}(Ack \Rightarrow \mathbf{AF}\ \neg Req)\mathtt{]}$
  **PSL:** $\mathtt{always(}Req\ \mathtt{->}\ \mathtt{next(}Ack\ \mathtt{->}\ \mathtt{eventually!}\ \mathtt{!}Req\mathtt{))}$
  **PSL:** $\mathtt{always(}\{Req\mathtt{;}Ack\}\ \mathtt{|->}\ \mathtt{eventually!}\ \mathtt{!}Req\mathtt{)}$

- "$P$ is true on all paths $i$ units of time later."

  **CTL:** $\mathbf{AX}_i\ P \ \equiv\ \underbrace{\mathbf{AX}(\mathbf{AX}(\cdots(\mathbf{AX}\ P)\cdots))}_{i \text{ instances of } \mathbf{AX}}$
  **PSL:** $\mathtt{next[}i\mathtt{]}\,P$

- "$P$ is true on all paths sometime between $i$ units of time later and $j$ units of time later."

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

  **PSL:** `next_e[`$i$`..`$j$`]`$P$

- "One cycle after $Req$, $Ack_1$ should become true, and then $Ack_2$ becomes true 1 to 6 cycles later and then eventually $Respond$ becomes true, but until it does $Waiting$ holds from the time of $Ack_2$."

  **CTL:** $\mathbf{AG}\,[Req \Rightarrow \mathbf{AX}\,[Ack_1 \wedge \mathbf{ABF}_{1..6}(Ack_2 \wedge \mathbf{A}\,[Waiting\ \mathbf{U}\ Respond]\,)]]$

  **PSL:** `always`
  `(`$Req$ `->` `next(`$Ack_1$ `&` `next_e[1..6](`$Ack_2$ `&` `(`$Waiting$ `until!` $Respond$`))))`

When I asked one of the PSL/Sugar designers to 'sanity check' a first version of these examples, I got back some errors (corrected above) and also the comment:

```
... these examples don't really illustrate the strengths of PSL all that
well.  To do so, you can invite your students to think about how you
would say the following in CTL:

 always (a -> next_event(b)(c))
 always {a;b[*];c} -> {d;e[*];f}
 always (a -> next b)@(posedge clk)
```

## Chapter 5

# HDLs Semantics

*Hardware description languages (HDLs) are used to specify hardware. HDL programs can be simulated, compiled to circuits and input to formal verifiers. These different activities use different semantics and it is a current research area to try to relate them. In this chapter, which is close to the current research frontier, an approach to formalising and relating different HDL semantics is described.*

*I am not sure how much of the detail that follows will make it into the lectures. As always, only material covered in the lectures is examinable!*

## 5.1 Introduction

Hardware description languages (HDLs) like VHDL and Verilog have a simulation oriented semantics based on events, i.e. changes to the values of wires and registers. This *event semantics* can accurately model detailed asynchronous behaviour, but is very fine-grained and does not easily support formal verification.

Most practical formal methods (e.g. model checking and theorem proving) are oriented towards descriptions of systems in terms of their execution traces, which are sequences of states. One might characterise simulation semantics as 'edge-oriented' and trace semantics as 'level-oriented'. The relationship between the two views is obtained by accumulating the changes (events) during a simulation cycle to obtain the state holding at the end of the cycle. The sequence of states that the simulation cycle quiesces to at successive instants of simulation time is an abstraction called the *trace semantics*. If there are race conditions, then there may be several possible successor states to a given state (i.e. branching time). Trace semantics has the same timescale as event (simulation) semantics – namely simulation time – but abstracts away from the individual events within a single simulation cycle (delta-time).

Clocked sequential systems can also be viewed more abstractly in terms of the sequence of states held in registers during successive clock cycles. This view is called the *cycle semantics*. Certain kinds of hardware (e.g. transparent level sensitive latches) are rather badly approximated if only the states latched at clock edges are considered, so equivalences between such hardware is best done with trace semantics.

A cycle semantics determines the exact clock cycle that each register transfer occurs on. A more abstract view is the *behavioural semantics* in which certain state transitions are regarded as not observable. With respect to behavioural semantics, functional equivalence is preserved by moving operations across certain sequences of clock cyles (e.g. within the same 'super state').

## 5.2 Syntax

The language described below is intended to embody the essence of the core of synthesisable Verilog, but with a lighter syntax.

A program consists of an initialising assignment to all the variable together with a set of concurrently executing non-terminating threads:

$$
\begin{array}{l}
\texttt{initial}\ V_1, \ldots, V_m\ \texttt{:=}\ E_1, \ldots, E_m \\
\texttt{always}\ S_1 \\
\texttt{always}\ S_2 \\
\quad\vdots \\
\texttt{always}\ S_n
\end{array}
$$

where $V_1$, $V_2$, ... , $V_m$ are *variables*, $E_1$, $E_2$, ... , $E_m$ are *expressions* and $S_1$, $S_2$, ... , $S_n$ are *statements*. Statements are built out of variables, expressions and event controls.

Each thread models a piece of hardware, with interconnection specified by coincidence of variable names.

If the variables represent registers, then an initialising assignment can be used to specify the 'power up' values of the registers. In practice, one often uses an 'unknown' value. In Verilog each register variable has `x` as its initial value (which can be overwritten by an explicit `initial`-statement).

### 5.2.1 Expressions

Expressions are built up from variables and constants using operators (unary, binary, conditional etc.). The metatavariables $V$ and $E$ will range over variables and expressions, respectively.

The structure of expressions is not specified in detail here. It is assumed that there are two Boolean constants `T` (alternatively `1`) and and `F` (alternatively `0`), an infixed binary equality operator `=` and a conditional $E\texttt{?}E_1\texttt{:}E_2$ (*if* $E$ *then* $E_1$ *else* $E_2$). In examples, a mixture of standard logical notation and the C-like expression syntax of Verilog will be used.

It is assumed that each $E$ expression has a *value* with respect to an assignment of values to each variable occurring in it. If $s$ is such an assignment of values to variables, then the value of $E$ is denoted by $[\![E]\!]s$.

### 5.2.2 Event expressions

Event expressions $T$ only occur as components of event controls `@(`$T$`)`. They can be used both to delimit cycle boundaries and to specify combinational logic.

$$
\begin{array}{lcll}
T & ::= & V & \text{(Change of value)} \\
 & \mid & \texttt{posedge}\ V & \text{(Positive edge)} \\
 & \mid & \texttt{negedge}\ V & \text{(Negative edge)} \\
 & \mid & T_1\ \texttt{or}\ \cdots\ \texttt{or}\ T_n & \text{(Compound sensitivity list)}
\end{array}
$$

### 5.2.3 Statements

The syntax of statements $S$ is given by the BNF below.

$$
\begin{array}{lcll}
S & ::= & V_1, \ldots, V_n\ \texttt{:=}\ E_1, \ldots, E_n & \text{(Assignment)} \\
 & \mid & S_1\texttt{;}\ \cdots\ \texttt{;}\ S_n & \text{(Sequencing block)} \\
 & \mid & \texttt{if}\ E\ \texttt{then}\ S_1\ \{\texttt{else}\ S_2\} & \text{(Conditional)} \\
 & \mid & \texttt{@(}T\texttt{)}\ S & \text{(Event control)}
\end{array}
$$

In an assignment $V_1, \ldots, V_n$ `:=` $E_1, \ldots, E_n$, the variables $V_1, \ldots, V_n$ must be distinct. Each pair $(V_i, E_i)$ is called an *update*. The assignment is executed by updating in parallel each $V_i$ with the value of $E_i$.

Sometimes the notation $\overline{\mathcal{V}}$ will be used for the vector of variables $V_1, \ldots, V_n$ and $\overline{\mathcal{E}}$ for the vector of expressions $E_1, \ldots, E_n$. An important special case is a single assignment $V$ `:=` $E$. Note that in Verilog all assignments are single assignments with "`=`" (or "$\Leftarrow$") being used instead of "`:=`".

Parallel assignments in Verilog are achieved using non-blocking assignment, i.e.:

$V_1, \ldots, V_n$ `:=` $E_1, \ldots, E_n$

is written as:

`begin` $V_1 \Leftarrow E_1$`;` $\cdots$ $V_n \Leftarrow E_n$`;` `end`

Note also that in Verilog "`;`" is a terminator (not a separator), sequences have to be surrounded by "`begin`" and "`end`" and "`if` $E$ `then`" is written "`if(`$E$`)`".

### 5.2.4 Abbreviations

The following constructs are derived in terms of the syntax given above.

#### 5.2.4.1 Case statements

```
case (E)
    E1: S1
    E2: S2
     ⋮
    En: Sn
   default: Sn+1
endcase
```

$\texttt{case}\ (E)$
$\quad E_1\texttt{:}\ S_1$
$\quad E_2\texttt{:}\ S_2$
$\quad\ \vdots$
$\quad E_n\texttt{:}\ S_n$
$\ \ \texttt{default:}\ S_{n+1}$
$\texttt{endcase}$

abbreviates

$\texttt{if (}E\texttt{=}E_1\texttt{)}\ S_1\ \texttt{else if (}E\texttt{=}E_2\texttt{)}\ S_2\ \cdots\ \texttt{else if (}E\texttt{=}E_n\texttt{)}\ S_n\ \texttt{else}\ S_{n+1}$

#### 5.2.4.2 Continuous assignments

$\texttt{assign}\ \overline{\mathcal{V}}\ \texttt{=}\ \overline{\mathcal{E}}$

abbreviates

$\texttt{always}\ \ \texttt{@(}V_1\ \texttt{or}\ \cdots\ \texttt{or}\ \ V_r\texttt{)}\ \overline{\mathcal{V}}\ \texttt{:=}\ \overline{\mathcal{E}}$

where $V_1$, ... , $V_r$ are the variables occurring in any component expression of the vector $\overline{\mathcal{E}}$.

Note there there is a subtle difference between these continuous assignments in the idealised Verilog presented here and the ones in real Verilog. However, the HDL used here is not rich enough for this difference to be significant.

## 5.3 Semantic Pseudo-Code

To simplify the presentation of the semantics, each thread is first compiled to a sequence of assembler-like pseudo instructions.

### 5.3.1 Pseudo-code instructions

Statements are compiled to pseudo-code consisting of sequences of instructions from the following instruction set:

| | |
|---|---|
| $\overline{\mathcal{V}}\ \texttt{:=}\ \overline{\mathcal{E}}$ | assignment |
| $\texttt{@(}T\texttt{)}$ | event control |
| $\texttt{go}\ n$ | unconditional jump to instruction $n$ |
| $\texttt{ifnot}\ E\ \texttt{go}\ n$ | jump to instruction $n$ if $E$ is not true |

### 5.3.2 The size of a statement

The size function defined in this section is used in the translation algorithm described in 5.3.3. Let the size $\mid S \mid$ of $S$ be as defined below inductively on the structure of $S$. It will turn out that $\mid S \mid$ is the number of instructions that $S$ is translated to.

$$
\begin{array}{lcl}
\mid \overline{\mathcal{V}} \texttt{ := } \overline{\mathcal{E}} \mid & = & 1 \\
\mid S_1 \texttt{;} \cdots \texttt{;}\ S_n \mid & = & \mid S_1 \mid + \ \cdots \ + \ \mid S_n \mid \\
\mid \texttt{if}\ E\ \texttt{then}\ S \mid & = & \mid S \mid +1 \\
\mid \texttt{if}\ E\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid & = & \mid S_1 \mid + \mid S_2 \mid +2 \\
\mid \texttt{@(}T\texttt{)} \mid & = & 1
\end{array}
$$

### 5.3.3 Translation algorithm

The sequence $\langle \texttt{i}_0, \ldots, \texttt{i}_n \rangle$ of instructions that statement $S$ is translated to is denoted by $[\![S]\!]p$, where $p$ is the position of the first instruction (e.g. `go` $p$ jumps to the start of the program).

To handle sequential blocks, it is convenient to define in parallel the translation of a sequence $\langle S_1, \ldots, S_N \rangle$ of statements (see the third and fourth clauses of the definition below). The symbol $\frown$ denotes sequence concatenation.

$$
\begin{array}{lcl}
[\![\overline{\mathcal{V}} \texttt{ := } \overline{\mathcal{E}}]\!]p & = & \langle \overline{\mathcal{V}} \texttt{ := } \overline{\mathcal{E}} \rangle \\
[\![S_1\texttt{;}\ S_2\texttt{;} \ldots \texttt{;}S_n]\!]p & = & [\![S_1]\!]p \frown [\![S_2\texttt{;} \ldots \texttt{;}S_n]\!](p+ \mid S_1 \mid) \\
[\![\texttt{if}\ E\ \texttt{then}\ S]\!]p & = & \langle \texttt{ifnot}\ E\ \texttt{go}\ p+ \mid S \mid +1 \rangle \frown [\![S]\!](p+1) \\
[\![\texttt{if}\ E\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]p & = & \langle \texttt{ifnot}\ E\ \texttt{go}\ p+ \mid S_1 \mid +2 \rangle \\
 & & \quad \frown [\![S_1]\!](p+1) \\
 & & \quad \frown \langle \texttt{go}\ p+ \mid S_1 \mid + \mid S_2 \mid +2 \rangle \\
 & & \quad \frown [\![S_2]\!](p+ \mid S_1 \mid +2) \\
[\![\texttt{@(}T\texttt{)}\ S]\!]\ p & = & \langle \texttt{@(}T\texttt{)} \rangle \frown [\![S]\!](p+1)
\end{array}
$$

A thread `always` $S$ is translated into the infinite loop:

$$[\![S]\!]0 \frown \langle \texttt{go}\ 0 \rangle$$

### 5.3.4 Example translations

| | |
|---|---|
| if $E$<br> then<br>  a := b;<br>  @(posedge clk) b := a;<br> else<br>  a := b;<br>  b := a; | 0: ifnot $E$ go 5<br>1: a := b<br>2: @(posedge clk)<br>3: b := a<br>4: go 7<br>5: a := b<br>6: b := a |

```
always
 @(b or c) a := b + c;
```

```
0:   @(b or c)
1:   a := b + c
2:   go 0
```

```
always
  @(posedge clk)
   total := data;
  @(posedge clk)
   total := total + data;
  @(posedge clk)
   total := total + data;
```

```
0:   @(posedge clk)
1:   total := data
2:   @(posedge clk)
3:   total := total + data
4:   @(posedge clk)
5:   total := total + data
6:   go 0
```

```
always
 @(posedge clk)
  case state of
  0:
   total := data;
   state := 1;
  1:
   total := total + data;
   state := 2;
  default:
   total := total + data;
   state := 0;
```

```
 0:   @(posedge clk)
 1:   ifnot state = 0 go 5
 2:   total := data
 3:   state := 1
 4:   go 11
 5:   ifnot state = 1 go 9
 6:   total := total + data
 7:   state := 2
 8:   go 11
 9:   total := total + data
10:   state := 0
11:   go 0
```

## 5.4 Event Semantics

The event semantics is specified via a simulation cycle that non-deterministically executes threads according to the event controls present.

Consider the following program:

always $S_1$
always $S_2$
$\vdots$
always $S_n$

An *input* is a variable that occurs in one of $S_1, \ldots, S_n$, but does not occur on the left hand side of an assignment.

A *state variable* is a variable that occurs on the left hand side of an assignment. The values of the state variables are the *outputs* of the program.

It is assumed that each input is driven by the environment with a sequence of values – the elements of the sequence being the values at successive instants of simulation time. When time increases, the values being input at the new time may change from their previous value. This change may cause an event expression $T$ to 'fire' and any threads waiting on `@(`$T$`)` will then be enabled for execution. The simulation cycle (see 5.4.1 below) consists of repeatedly choosing an enabled thread, executing it until an event control is reached (an atomic step), and then enabling any new threads that fire. If several threads are simultaneously enabled, then the choice of which thread to advance is non-deterministic. If a state is reached in which there are no enabled threads, then the simulation cycle is said to have *quiesced*. When this happens the simulation time is incremented, the next set of inputs is assigned to the input variables and the whole process repeats.

Thus sequences of values on the inputs non-deterministically generate sequences of states (and hence a sequence of outputs). Such a sequence is called a *trace* and has the form:

$$\langle i_0, s_0\rangle\langle i_1, s_1\rangle\langle i_2, s_2\rangle \;\cdots$$

where $s_0$ is an initial state and $i_0 i_1 i_2 \ldots$ are inputs.

### 5.4.1 Simulation algorithm

The state of a thread during simulation consists of:

- the simulation time (a non-negative integer);
- the values of all variables;
- the value of the program counter (denoted by `pc`).

It is assumed that the environment to the program supplies a value for each input at each instant of simulation time.

The simulation algorithm can be succinctly described if some terminology is introduced.

The instruction pointed to by the program counter of a thread is called the *current instruction*.

A thread is *enabled* if its current instruction is not an event control `@(`$T$`)`. Enabled threads can be executed immediately. Once a thread starts executing it continues until an event control is reached. This execution is atomic and is called *advancing* the thread (see below)

A thread is *listening* if its current instruction is an event control `@(`$T$`)`. Note that if a thread is not enabled then it is listening.

When a variable or input changes, an event expression may be *triggered*:

- $V$ is triggered if the new value of $V$ differs from the previous one;

- `posedge` $V$ is triggered if the new value of $V$ is `1` and the previous value was not `1`;
- `negedge` $V$ is triggered if the new value of $V$ is `0` and the previous value was not `0`;
- $T_1$ `or` $\cdots$ `or` $T_n$ is triggered if any of the $T_i$ are triggered ($1 \leq i \leq n$).

A thread is *triggered* if it is listening and the event control of the current instruction is triggered. A triggered thread is *fired* by incrementing its program counter.

If a thread is not executing, but is enabled, then its current instruction cannot be an unconditional jump, but must be an assignment or a conditional jump. This is because, by the definition of $[\![S]\!]$, unconditional jumps cannot occur immediately after event controls, so when a thread is triggered the current instruction cannot become an unconditional jump.

Threads are executed concurrently. When an enabled thread is run it advances without interruption until an event control is reached. Thus a thread never stops running at an unconditional jump `go` $n$. To see the need for this consider:

| Thread 1 | Thread 2 |
|---|---|
| `always @(inp) w := !inp;` | `always @(w or inp) outp := w & inp;` |
| `0: @(inp)`<br>`1: w := !inp`<br>`2: go 0` | `0: @(w or inp)`<br>`1: outp := w & inp`<br>`2: go 0` |

If Thread 2 could stop at instruction `2`, then Thread 1 could interleave at that point and update to `w` whilst Thread 2 is not listening. By forcing all jumps to be taken immediately, this possibility is precluded.

The execution of a single instruction is performed as follows:

## Instruction Execution

- $V_1, \ldots, V_n$ `:=` $E_1, \ldots, E_n$
  For $1 \leq i \leq n$ change the $V_i$ component of the state to the value of $E_i$ in the current state, note all variables whose value changes, then increment the program counter;

- `go` $n$
  Set the program counter to $n$;

- `ifnot` $E$ `go` $n$
  If $E$ is true in the current state then increment the program counter, otherwise set it to $n$.

When a thread fires these operations are repeated until an event control is reached and then any triggered threads (including possibly the one just executed) are fired. This process is called advancing a thread.

### Advancing a Thread

- A thread is advanced by repeatedly executing the current instruction until an event control is reached.
- This execution is considered to be an atomic step and is not interruptable.
- As a thread is advancing, the variables that are modified by assignments are noted.
- When the thread stops advancing any listening threads that are guarded by triggered event controls are fired.

Note that states at which threads stop advancing are those for which the current instruction is an event control. Such states are called *stopping states*. When reasoning about the execution of programs, it is sufficient to just consider the transitions of threads between stopping states, since these are the uninterruptible atomic actions of the thread. The decomposition of these actions into smaller ones, like unconditional jumps, is just an artifact of the way the event semantics is presented.

Inititially simulation time is 0, each program counter is set to 0 and each variable has some initial value (which is not specified here, but would typically be an 'undefined' value, like `x` in Verilog). The simulation algorithm consists of iterating forever the simulation step shown in the box below.

### Simulation Step

- **There is an enabled thread:**

  Non-deterministically choose an enabled thread and advance it.
- **No enabled threads – i.e. all threads listening:**

  Increment the simulation time, read in new input values and fire any triggered threads.

## 5.5 Trace semantics

For each program, initial state and sequence of inputs, the simulation algorithm determines a set of traces. Each member of such a trace consists of the inputs at each time together with a state the simulation cycle quiesces to. Note that since simulation is non-determinishtic, there may be several possible quiescent states (or none). Thus time is branching.

The simulation algorithm is hard to work with. In this section it is shown how the traces of a number of common kinds of programs can be derived more directly.

### 5.5.1 Combinational programs

The idea of a combinational program is that it computes its outputs instantaneously from its inputs. A syntactic condition that ensures this will be defined. In preparation, consider a thread having the form `always @(`$T$`)`$S$, where $S$ contains no event controls. This translates to pseudo code of the form:

0: @($T$)
$\vdots$
$n$: go 0

where the pseudo instruction 1, 2, ... , $n{-}1$ are assignments or jumps (i.e. not event controls).

Since threads execute atomically from event control to event control, it is clear that such a pseudo program is equivalent to one of the form:

0: @($T$)
1: $\overline{\mathcal{V}}$ := $\overline{\mathcal{E}}$
2: go 0

For example,

```
always @(y or z) x := 1;
                 if x>y then x,y := y,x else y,z := 3,y
                 z := x+y
```

translates to:

```
0: @(y or z)
1: x := 1
2: ifnot x>y go 5
3: x,y := y,x
4: go 6
5: y,z := 3,y
6: z := x+y
7: go 0
```

which is equivalent to:

```
0: @(y or z)
1: x,y,z :=
     x>y ? y : 1,
     x>y ? 1 : 3,
     (x>y ? y : 1) + (x>y ? 1 : 3)
2: go 0
```

A formal description of the straightforward conversion of combinational threads into a single parallel assignment, as illustrated by this example, is not given here.

A *weakly combinational* program has the form:

$\texttt{always @(}T_1\texttt{)}\ \overline{\mathcal{V}}_1\texttt{:=}\overline{\mathcal{E}}_1 \quad \cdots \quad \texttt{always @(}T_n\texttt{)}\ \overline{\mathcal{V}}_n\texttt{:=}\overline{\mathcal{E}}_n$

and satisfies three conditions:

1. $T_i$ consists of all the variables occurring in $\overline{\mathcal{E}}_i$ `or`-ed together;
2. no variable occurs more than once on the left of an assignment (no clashing assignments);
3. after performing any update $(V, E)$, the value of $V$ in the resulting state equals the value of $E$.

Updates satisfying condition 3 above are called idempotent. More formally, $(V, E)$ is *idempotent* if

$[\![V]\!](s[V \leftarrow [\![E]\!]s]) = [\![E]\!]s = [\![E]\!](s[V \leftarrow [\![E]\!]s])$

A sufficient condition that $(V, E)$ be idempotent is that $V$ not occur in $E$.

If the simulation cycle of a weakly combinational program quiesces then it does do with a a unique value for all the variables. Showing quiescence requires an additional condition (loop-free) which is defined later.

A weakly combinational program generates $n$ threads:

| <u>**Thread 1**</u> | | | <u>**Thread $n$**</u> | |
|---|---|---|---|---|
| 0: | $\texttt{@(}T_1\texttt{)}$ | | 0: | $\texttt{@(}T_n\texttt{)}$ |
| 1: | $\overline{\mathcal{V}}_1\ \texttt{:=}\ \overline{\mathcal{E}}_1$ | $\cdots$ | 1: | $\overline{\mathcal{V}}_n\ \texttt{:=}\ \overline{\mathcal{E}}_n$ |
| 2: | go 0 | | 2: | go 0 |

Suppose thread $i$ has program counter $\texttt{pc}_i$. Let $\mathsf{Inv}$ be the $n$-ary conjunction $\bigwedge_{i=1}^{n} \mathsf{Inv}_i$, where $\mathsf{Inv}_i$ is the statement:

$$(\texttt{pc}_i = 0 \ \Rightarrow\ \overline{\mathcal{V}}_i = \overline{\mathcal{E}}_i)\ \wedge\ (\overline{\mathcal{V}}_i \neq \overline{\mathcal{E}}_i \ \Rightarrow\ \texttt{pc}_i = 1)$$

$\mathsf{Inv}_i$ asserts two things: (i) if the $i$-th thread is at instruction 0 then the value of the vector of variables $\overline{\mathcal{V}}_i$ equals the value of the vector of expressions $\overline{\mathcal{E}}_i$ and (ii) if any of these values are not equal then the thread is at instruction 1.

Note that $\mathsf{Inv}$ is a predicate on states. This is made explicit by defining:

$$\begin{array}{lll}\mathsf{Inv}(s) \equiv_{Def} & \bigwedge_{i=1}^{n} & ([\![\mathsf{pc}_i]\!]s = 0 \;\Rightarrow\; [\![\overline{\mathcal{V}}_i]\!]s = [\![\overline{\mathcal{E}}_i]\!]s) \\ & & \wedge \\ & & ([\![\overline{\mathcal{V}}_i]\!]s \neq [\![\overline{\mathcal{E}}_i]\!]s \;\Rightarrow\; [\![\mathsf{pc}_i]\!]s = 1)\end{array}$$

However, where the state is clear from context the less formal implicit-state notation will be used.

We want to show that $\mathsf{Inv}$ is an invariant of each simulation step, and that simulation quiesces.

To show $\mathsf{Inv}$ is an invariant suppose $\mathsf{Inv}$ holds. There are two cases to consider: there are some enabled threads, or there are no enabled threads.

1. If there are some enabled threads, then one of them, say thread $i$, is chosen and executed. Since all updates are idempotent, the execution will cause the value of $\overline{\mathcal{V}}_i$ to become equal the value of $\overline{\mathcal{E}}_i$, hence $\mathsf{Inv}_i$ will hold immediately after the assignment is executed. Next, other threads may be fired (including, possibly, thread $i$ itself). The following case analysis establishes $\mathsf{Inv}$ continues to hold after this.
   (a) If thread $j$ is enabled but not selected for execution, then $\mathtt{pc}_j = 1$, so $\mathsf{Inv}_j$ remains vacuously true.
   (b) If a thread $j$ is triggered by the execution of thread $i$ (the case $j{=}i$ is possible), then thread $j$ becomes enabled with $\mathtt{pc}_j = 1$ and so $\mathsf{Inv}_j$ becomes vacuously true.
   (c) If a thread $j$ is listening but not triggered then it must be the case that no variable changed by executing thread $i$ occurs in $\overline{\mathcal{E}}_j$, so the value of $\overline{\mathcal{E}}_j$ is unchanged. Also $\mathtt{pc}_j$ remains equal to 0.
      i. If $j = i$ then as all updates are idempotent the execution of $\overline{\mathcal{V}}_i \;\mathtt{:=}\; \overline{\mathcal{E}}_i$ makes the value of each variable in $\overline{\mathcal{V}}_i$ equal to the value of the corresponding expression in $\overline{\mathcal{E}}_i$ and hence $\mathsf{Inv}_i$ holds.
      ii. If $j \neq i$, the variables in $\overline{\mathcal{V}}_j$ are distinct from those in $\overline{\mathcal{V}}_i$, so the value of $\overline{\mathcal{V}}_j$ is unchanged and thus $\mathsf{Inv}_j$ remains true.
2. If there are no enabled threads, then time will advance and if the inputs changes some threads may be fired. If no threads are fired, then the values of all variables occurring in any $\overline{\mathcal{E}}_i$ $(1 \leq i \leq n)$ will be unchanged. Since the environment only changes inputs, the value of $\overline{\mathcal{V}}_i$ is unchanged. Thus $\mathsf{Inv}$ continues to hold.

   If thread $i$ is fired then $\mathtt{pc}_i = 1$ and $\mathsf{Inv}_i$ becomes vacuously true.

Thus $\mathsf{Inv}$ is an invariant of each simulation step. If simulation quiesces, then all program counters will be 0 and so for each $i$ the value of $\overline{\mathcal{V}}_i$ will equal the value of $\overline{\mathcal{E}}_i$.

To establish that the simulation of combinational programs always quiesces – i.e. doesn't go into an infinite loop without advancing time – it is sufficient to exhibit a 'variant' that decreases on each iteration of the simulation algorithm for which there is an enabled thread. Such a variant shows that eventually a simulation state must be reached in which there are no enabled threads, and hence time advances. A suitable variant uses the 'Dershowitz-Manna multiset ordering'. Under this ordering multiset $M_2$ is less than multiset $M_1$ if $M_2$ can be obtained from $M_1$ by removing a finite number of elements and replacing each of them by a finite number of strictly smaller elements. This ordering is well-founded if the ordering on the elements is.

The idea of the termination proof is to show that each chain of events – one causing the next – is finite. Define $(V, E)\nearrow(V', E')$ to mean, roughly, that doing the update $(V, E)$ will immediately trigger a thread containing the update $(V', E')$.

$$(V, E)\nearrow(V', E') \quad \equiv_{Def} \quad \exists s. \quad [\![E']\!]s \neq [\![E']\!](s[V \leftarrow [\![E]\!]s])$$

where $s[V \leftarrow [\![E]\!]s]$ denotes the state optained from $s$ by changing the value of variable $V$ to the value of $E$ and leaving the values of all other variables unchanged.

The relation $\nearrow$ is defined semantically, but for weakly combinational programs it implies a syntactic relation $\nearrow\!\!\!\nearrow$ defined by:

$$(V, E)\nearrow\!\!\!\nearrow(V', E') \quad \equiv_{Def} \qquad V \textit{ occurs in } E' \textit{ and } V \neq V'$$

If $(V, E)\nearrow(V', E')$ then $(V, E)\nearrow\!\!\!\nearrow(V', E')$. Assume that $(V, E)\nearrow\!\!\!\nearrow(V', E')$ doesn't hold, then there are two cases:

**Case 1:** $V$ does not occur in $E'$.

In this case it is clear that for all $s$:

$$[\![E']\!]s \; = \; [\![E']\!](s[V \leftarrow [\![E]\!]s])$$

which contradicts

$$\exists s. \quad [\![E']\!]s \neq [\![E']\!](s[V \leftarrow [\![E]\!]s])$$

hence it can't be the case that $(V, E)\nearrow(V', E')$.

**Case 2:** $V = V'$.

There are no clashing assignments, thus $E = E'$, so if $(V, E)\nearrow(V', E')$ then $(V, E)\nearrow(V, E)$, i.e. $\exists s.\ [\![E]\!]s \neq [\![E]\!](s[V \leftarrow [\![E]\!]s])$, which contradicts the requirement that $(V, E)$ be idempotent.

In both cases it follows that it is not the case that $(V, E)\nearrow(V', E')$.

Thus by contradiction $(V, E)\nearrow(V', E')$ entails $(V, E)\nearrow\!\!\!\nearrow(V', E')$.

Define a $\nearrow$*-chain* in a set of assignments to be a sequence $(V_1, E_1)$, $\ldots$ , $(V_l, E_l)$ of updates such that for $1 \leq p < l$:

(i) $V_p$ is updated to $E_p$ by one of the assignments in the set;

(ii) $(V_p, E_p)\nearrow(V_{p+1}, E_{p+1})$.

Call a set of assignments *loop-free* if the lengths of all $\nearrow$-chains are bounded.

Define a program to be *combinational* if it is weakly combinational and loop free.

Note that since every $\nearrow$-chain is also a $\nearrow\!\!\!\!\nearrow$-chain, if there is an infinite $\nearrow$-chain then there is also an infinite $\nearrow\!\!\!\!\nearrow$-chain. So checking that all $\nearrow\!\!\!\!\nearrow$-chains are finite ensures the program is loop-free and hence combinational.

Define the *height* of $V$ to be the length of the longest $\nearrow$-chain in the assignments $\overline{\mathcal{V}}_1 := \overline{\mathcal{E}}_1$, ... , $\overline{\mathcal{V}}_n := \overline{\mathcal{E}}_n$, whose first element is $(V, E)$, for some $E$.

To show each simulation cycle quiesces, consider the sequence of steps that happen after the advance of simulation time. If no inputs change then the cycle is already quiescent and there is nothing to show. If some inputs change, then some threads will fire and become enabled with their program counters at 1.

The subsequent quiescence (termination) of the simulation algorithm can be verified by taking as variant the multiset of the heights of those variables $V$ such that the value of $V$ differs from the value of $E$. Note that by $\mathsf{Inv}$ all such updates must be part of an enabled thread.

Suppose some threads are enabled and one of them, thread $i$ say, is selected for execution. All the variables in $\overline{\mathcal{V}}_i$ are updated with the corresponding variables in $\overline{\mathcal{E}}_i$ by the assignment and the thread advances to instruction 0. Next, every thread $j$ such that $T_j$ contains a variable modified by executing thread $i$ is fired and so becomes enabled.

If an update $(V, E)$ was done by executing thread $i$ then, since it is idempotent, the value of $V$ will equal the value of $E$ after the update and so the height of $V$ will be removed from the multiset. This deleted height will be replaced with the heights of other variables $V'$ such that there is an update $(V', E')$ and the value of $V'$ becomes different from the value of $E'$. Suppose $s$ is the state in which $(V, E)$ was done. The height of $V'$ will be added if $[\![V']\!]s = [\![E']\!]s$ but:

$$[\![V']\!](s[V \leftarrow [\![E]\!]s]) \neq [\![E']\!](s[V \leftarrow [\![E]\!]s])$$

There are two cases to consider:

**Case 1:** If $V \neq V'$ this equation is:

$$[\![V']\!]s \neq [\![E']\!](s[V \leftarrow [\![E]\!]s])$$

i.e.

$$[\![E']\!]s \neq [\![E']\!](s[V \leftarrow [\![E]\!]s])$$

hence $(V, E)\nearrow(V', E')$.

**Case 2:** If $V = V'$ then (as each variable can only be on the left of at most one update) $E = E'$ and hence:

$$[\![V]\!](s[V \leftarrow [\![E]\!]s]) \neq [\![E]\!](s[V \leftarrow [\![E]\!]s])$$

i.e.

$$[\![E]\!]s \neq [\![E]\!](s[V \leftarrow [\![E]\!]s])$$

but this is impossible as $(V, E)$ is assumed idempotent.

Thus each added height – i.e. the heights of each such $V'$ – will be less than the height of $V$, so the multiset decreases in the Dershowitz-Manna ordering.

Each step of the simulation algorithm decreases the multiset. As long as there are enabled threads the simulation cycle continues, so by the invariant $\mathsf{Inv}$ and the well-foundedness of the multiset ordering eventually there must be no enabled threads, i.e. the simulation quiesces.

#### 5.5.1.1 Transparent latches

Consider:

```
always @(clk or d) if clk then q := d
```

If this is converted to an equivalent assignment, the result is:

```
always @(clk or d) q := clk ? d : q
```

The update $(\mathtt{q}, \mathtt{clk\ ?\ d\ :\ q})$ is idempotent, since clearly:

$$\forall s.\ [\![\mathtt{clk\ ?\ d\ :\ q}]\!]s \ = \ [\![\mathtt{clk\ ?\ d\ :\ q}]\!](s[\mathtt{q} \leftarrow [\![\mathtt{clk\ ?\ d\ :\ q}]\!]s])$$

It is not the case that:

$$(\mathtt{q}, \mathtt{clk\ ?\ d\ :\ q}) \not\nearrow\!\!\!\nearrow (\mathtt{q}, \mathtt{clk\ ?\ d\ :\ q})$$

hence the latch is loop-free.

Define an update $(V, E)$ to be *latch-like* if $E$ is either $V$ or a nested conditional expression in which the only occurrences of $V$ are in the then or else part of conditionals. This is a static (syntactically checkable) property that obviously implies simplicity.

Define a thread $\mathtt{always\ @(}T\mathtt{)}\ \overline{\mathcal{V}}\mathtt{:=}\overline{\mathcal{E}}$ to be latch-like if every update $(V, E)$ in the assignment is either latch-like or has $V$ not occurring in $E$. Note that all such updates will be idempotent.

Consider a weakly combinational program:

$$\mathtt{always\ @(}T_1\mathtt{)}\ \overline{\mathcal{V}}_1\mathtt{:=}\overline{\mathcal{E}}_1 \quad \cdots \quad \mathtt{always\ @(}T_n\mathtt{)}\ \overline{\mathcal{V}}_n\mathtt{:=}\overline{\mathcal{E}}_n$$

where each thread is latch-like. A sufficient conditional for this to be combinational is that every chain:

$(V_1, E_1) \nearrow\!\!\!\nearrow (V_2, E_2) \nearrow\!\!\!\nearrow (V_3, E_3)\ \ldots$
is finite.

The fact that loop-free latch-like threads are combinational shows that the use here of the term is a bit non-standard. One might expect that "combinational" would denote circuits with no memory.

#### 5.5.1.2 Summary

If $S_1$, ... , $S_n$ contain no event controls then any program of the form:

$$\texttt{always @(}T_1\texttt{)}\ S_1 \quad \cdots \quad \texttt{always @(}T_n\texttt{)}\ S_n$$

is equivalent to a program of the form:

$$\texttt{always @(}T_1\texttt{)}\ \overline{\mathcal{V}}_1\texttt{:=}\overline{\mathcal{E}}_1 \quad \cdots \quad \texttt{always @(}T_n\texttt{)}\ \overline{\mathcal{V}}_n\texttt{:=}\overline{\mathcal{E}}_n$$

Such a program is combinational if it is weakly combinational and loop-free. A sufficient condition is that every thread is latch-like and there are no infinite $\nearrow\!\!\!\nearrow$-chains.

Each simulation step of a combinational program preserves the invariant:

$$\bigwedge_{i=1}^{n}\ (\texttt{pc}_i = 0\ \Rightarrow\ \overline{\mathcal{V}}_i = \overline{\mathcal{E}}_i)\ \wedge\ (\overline{\mathcal{V}}_i \neq \overline{\mathcal{E}}_i\ \Rightarrow\ \texttt{pc}_i = 1)$$

and as long as inputs do not change, each step reduces the size of the variant and thus the simulation cycle must quiesce with all threads listening at pseudo-instruction 0 and hence with $\overline{\mathcal{V}}_i = \overline{\mathcal{E}}_i$.

Thus if $s_0$ is an initial state and $i_0 i_1 i_2 \ldots$ are the inputs, then the traces of this combinational program are sequences:

$$\langle i_0, s_0\rangle\langle i_1, s_1\rangle\langle i_2, s_2\rangle\ \cdots$$

where the value of each variable in $\overline{\mathcal{V}}_i$ in state $s_{t+1}$ is the value of the corresponding expression in $\overline{\mathcal{E}}_i$ evaluated with inputs $i_{t+1}$ and state $s_t$.

### 5.5.2 Flip-flops

A flip-flop is a single register with input `d` and output `q` that is clocked on the positive edge of a clock `clk` (which is assumed distinct from `d` and `q`). This is modelled by:

```
always @(posedge clk) q := d
```

It is clear that the effect of the simulation algorithm is to update the value of `q` with the value input on `d` at each simulation time that triggers `posedge clk` (i.e. at each time that `clk` is high, but was not high at the preceding time).

If $f(t)$ denotes the value of input or variable $f$ at simulation time $t$, then the trace semantics of the register above is:

$$\mathtt{q}(t{+}1) \;=\; \mathit{if}\ \mathtt{clk}(t{+}1) = \mathtt{1} \,\wedge\, \mathtt{clk}(t) \neq \mathtt{1}\ \mathit{then}\ \mathtt{d}(t)\ \mathit{else}\ \mathtt{q}(t)$$

Define:

$$(\mathtt{posedge}\ \mathit{clk})(t+1) \;\equiv_{\mathit{Def}}\; \mathit{if}\ \mathtt{clk}(t{+}1) = \mathtt{1} \,\wedge\, \mathtt{clk}(t) \neq \mathtt{1}$$

then the trace semantics can be written:

$$\mathtt{q}(t{+}1) \;=\; \mathit{if}\ (\mathtt{posedge\ clk})(t+1)\ \mathit{then}\ \mathtt{d}(t)\ \mathit{else}\ \mathtt{q}(t)$$

A bank of $n$ registers can be represented by a single parallel assignment:

```
always @(posedge clk) q̄ := d̄
```

where $\overline{\mathtt{q}} = (\mathtt{q}_1, \ldots, \mathtt{q}_n)$ and $\overline{\mathtt{d}} = (\mathtt{q}_1, \ldots, \mathtt{d}_n)$ and $\mathtt{q}_i \neq \mathtt{q}_j$ if $i \neq j$. The trace semantics is:

$$\bigwedge_{i=1}^{n} \mathtt{q}_i(t{+}1) \;=\; \mathit{if}\ (\mathtt{posedge\ clk})(t+1)\ \mathit{then}\ \mathtt{d}_i(t)\ \mathit{else}\ \mathtt{q}(t)$$

An alternative representation, with the same trace semantics, is to have $n$ separate threads; one for each individual register.

```
always @(posedge clk) q1 := d1
  ⋮
always @(posedge clk) qn := dn
```

The trace equivalence of the two representations is assumed clear.

Consider now some combinational logic, say an inverter, connected to the input of a flip-flop. This could be represented with one thread by:

```
always @(posedge clk) q := !d
```

or with the inverter and flip-flop in different threads:

```
always @(d) a := !d
always @(posedge clk) q := a
```

If `d` changes at the same time as a positive edge on `clk` then the second representation, but not the first, might store the negation of the unchanged (i.e. previous) value of `d`. This would happen if the flip-flop thread was executed first. The subsequent execution of the inverter thread will set `a` to the negation of the new value of `d`, but since the flip-flop is not sensitive to changes in `a` it won't update the value of `q`.

Thus the single thread representation is more robust and generally to be preferred.

The trace semantics of the single thread version is:

$$\mathtt{q}(t{+}1) \;=\; \mathit{if}\ (\mathtt{posedge\ clk})(t+1)\ \mathit{then}\ \mathtt{!d}(t)\ \mathit{else}\ \mathtt{q}(t)$$

and of the two threads is:

$$\begin{array}{l}(\forall t.\ \mathtt{a}(t) \;=\; \mathtt{!d}(t))\\ \wedge\\ (\forall t.\ \mathtt{q}(t{+}1) \;=\; \mathit{if}\ (\mathtt{posedge\ clk})(t+1)\ \mathit{then}\ \mathtt{a}(t)\ \mathit{else}\ \mathtt{q}(t))\end{array}$$

If it is assumed that inputs never change on clock edges[1] then the two representations are produce the same signal trace on `q`.

If the combinational logic is connected to the output of the flip-flop the problem doesn't arise. Thus no matter what order the threads in:

```
always @(posedge clk) a := d
always @(a) q := !a
```

are executed, the final quiescent state will be the same – namely `q` will equal the negation of the value of `d`.

$$\begin{array}{l}(\forall t.\ \mathtt{a}(t{+}1) \;=\; \mathit{if}\ (\mathtt{posedge\ clk})(t+1)\ \mathit{then}\ \mathtt{d}(t)\ \mathit{else}\ \mathtt{q}(t))\\ \wedge\\ (\forall t.\ \mathtt{q}(t) \;=\; \mathtt{!a}(t))\end{array}$$

However, with this example there is a hazard: if the inverter executes first then `q` will be set to the negation of the old value of `a`. The subsequent execution of the flip-flop will update `a` and might then change the value of `q`, since the inverter is sensitive to changes in `a`. The hazard is only visible at the event semantics level; it is invisiable with trace semantics.

[1] This assumption on the environment can be hard or impossible to enforce if the inputs are asynchronous – however clever analogue engineering can make it true with high probablility.

### 5.5.3 RTL programs

Register Transfer Level (RTL) hardware consists of combinational logic and flip-flops. An HDL representation of this is a program that can be partitioned into a set of threads that are a combinational program plus a set of flip-flops of the form:

always @(posedge clk) $\overline{\mathcal{V}}$ := $\overline{\mathcal{E}}$

It is assumed that there is a single input clock `clk` and that the environment is such that no other inputs change when `clk` has a positive edge. Also it is assumed no two distinct flip-flops drive the same ourput (no clashing assignments). Thus an RTL program has the form:

always @($T_1$) $\overline{\mathcal{V}}_1$ := $\overline{\mathcal{E}}_1$
$\vdots$
always @($T_m$) $\overline{\mathcal{V}}_m$ := $\overline{\mathcal{E}}_m$

always @(posedge clk) $\overline{\mathcal{V}}_{m+1}$ := $\overline{\mathcal{E}}_{m+1}$
$\vdots$
always @(posedge clk) $\overline{\mathcal{V}}_{m+n}$ := $\overline{\mathcal{E}}_{m+n}$

where the first $m$ threads constitute a combinational program and the last $n$ threads are flip-flops.

A simulation cycle that starts with a positive edge on `clk` will (by assumption) be one for which none of the other inputs change. Thus the positive edge will enable all the flip-flops and none of the other threads. After $m$ simulation steps (i.e. one to execute each flip-flop) the output variables will have been updated and the cycle will quiesce.

A simulation cycle that that doesn't start with a positive edge on `clk` will not enable any of the flip-flops. The rest of the threads will simulate as combinational logic and, by the analysis in 5.5.1, will quiesce with each variable set to the value of the expression that updates it.

Thus the trace semantics of the RTL program above is:

$$
\begin{array}{l}
(\forall t.\ \overline{\mathcal{V}}_1(t)\ =\ \overline{\mathcal{E}}_1(t)) \\
\wedge \\
\vdots \\
\wedge \\
(\forall t.\ \overline{\mathcal{V}}_m(t)\ =\ \overline{\mathcal{E}}_m(t)) \\
\wedge \\
(\forall t.\ \overline{\mathcal{V}}_{m+1}(t{+}1)\ =\ \mathit{if}\ (\texttt{posedge clk})(t+1)\ \mathit{then}\ \overline{\mathcal{E}}_{m+1}(t)\ \mathit{else}\ \overline{\mathcal{V}}_{m+1}(t)) \\
\wedge \\
\vdots \\
\wedge \\
(\forall t.\ \overline{\mathcal{V}}_{m+n}(t{+}1)\ =\ \mathit{if}\ (\texttt{posedge clk})(t+1)\ \mathit{then}\ \overline{\mathcal{E}}_{m+n}(t)\ \mathit{else}\ \overline{\mathcal{V}}_{m+n}(t))
\end{array}
$$

## 5.6 Cycle semantics

As described in the previous chapter, the trace semantics of a program can be abstracted at clock edges to get the sequence of values at clock edges. This is called the cycle semantics and corresponds to the abstract state machine level.

Consider, for example, the program:

$$\begin{array}{l} \texttt{initial}\ out\ \texttt{:=}\ \mathsf{F} \\ \texttt{always @(posedge}\ clk\texttt{)}\ out\ \texttt{:=}\ in\texttt{?}\neg out\texttt{:}out \end{array}$$

This has one register $out$ that's initialized to $\mathsf{F}$. On each rising edge of $clk$ the value of $out$ is complemented.

The trace semantics is:

$$\begin{array}{rcl} out(0) & = & \mathsf{F} \wedge \\ \forall t.\ out(t{+}1) & = & \mathit{if}\ (\texttt{posedge clk})(t{+}1)\ \mathit{then}\ (in(t) \rightarrow \neg out(t) \mid out(t))\ \mathit{else}\ out(t) \end{array}$$

Note that the inner conditional has been written using the "$(b \rightarrow e_1 \mid e_2)$" notation rather than using the equivalent "$\mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2$" notation. This is just a matter of style.

Expanding with the definition of $(\texttt{posedge clk})(t{+}1)$:

$$\begin{array}{rcl} out(0) & = & \mathsf{F} \wedge \\ \forall t.\ out(t{+}1) & = & \mathit{if}\ clk(t{+}1) \wedge \neg clk(t)\ \mathit{then}\ (in(t) \rightarrow \neg out(t) \mid out(t))\ \mathit{else}\ out(t) \end{array}$$

folding in the definition of $\mathsf{Rise}$ and switching to a uniform notation for conditionals:

$$out(0)\ =\ \mathsf{F}\ \wedge\ \forall t.\ out(t{+}1)\ =\ (\mathsf{Rise}\ clk\ t\ \rightarrow\ (in(t) \rightarrow \neg out(t)\ \mid\ out(t))\ \mid\ out(t))$$

Recall the definition of $\mathsf{Dtype}$:

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

The trace semantics of the program can thus be written:

$$out(0)\ =\ \mathsf{F}\ \wedge\ \mathsf{Dtype}(clk, (\lambda t.\ (in(t) \rightarrow \neg out(t)\ \mid\ out(t))), out)$$

Define:

$$\vdash f{\uparrow}c\ =\ f\ \mathsf{when}\ (\mathsf{Rise}\ c)$$

then

$$(f{\uparrow}c)\ n\ =\ (f\ \mathsf{when}\ (\mathsf{Rise}\ c))\ n\ =\ f(\mathsf{Timeof}\ (\mathsf{Rise}\ c)\ n)$$

so

$$\begin{array}{l} ((\lambda t.\ (in(t) \rightarrow \neg out(t)\ \mid\ out(t))){\uparrow}clk)\ n \\ \quad =\ (\lambda t.\ (in(t) \rightarrow \neg out(t)\ \mid\ out(t)))(\mathsf{Timeof}(\mathsf{Rise}\ clk)n) \\ \quad =\ (in(\mathsf{Timeof}(\mathsf{Rise}\ clk)n) \rightarrow \neg out(\mathsf{Timeof}(\mathsf{Rise}\ clk)n)\ \mid\ out(\mathsf{Timeof}(\mathsf{Rise}\ clk)n)) \\ \quad =\ ((in{\uparrow}clk)n \rightarrow \neg((out{\uparrow}clk)\ n)\ \mid\ (out{\uparrow}clk)\ n) \end{array}$$

Thus if we define a 'lifted' conditional:

$$\vdash (f \longrightarrow e_1 \mid e_2) \;=\; \lambda t.\ (f\ t \;\rightarrow\; e_1\ t \mid e_2\ t)$$

then

$$(\lambda t.\ (in(t) \rightarrow \neg out(t) \mid out(t)))\!\uparrow\! clk \;=\; (in\!\uparrow\! clk \;\longrightarrow\; \neg \circ out\!\uparrow\! clk \mid out\!\uparrow\! clk)$$

Instantiating the $\mathsf{Dtype}$-theorem yields:

$$\begin{array}{l}
\mathsf{Inf}(\mathsf{Rise}\ clk) \wedge \\
out(0) \;=\; \mathsf{F} \wedge \\
\mathsf{Dtype}(clk, (\lambda t.\ (in(t) \rightarrow \neg out(t) \mid out(t))), out) \\
\Rightarrow \\
\mathsf{Del}((in\!\uparrow\! clk \;\longrightarrow\; \neg \circ out\!\uparrow\! clk \mid out\!\uparrow\! clk),\ out\!\uparrow\! clk)
\end{array}$$

Thus at the abstracted cycle level:

$$\mathsf{Del}((in\!\uparrow\! clk \;\longrightarrow\; \neg \circ out\!\uparrow\! clk \mid out\!\uparrow\! clk),\ out\!\uparrow\! clk)$$

i.e.:

$$\forall n.\ out\!\uparrow\! clk(n{+}1) \;=\; (in\!\uparrow\! clk\ n \;\rightarrow\; \neg(out\!\uparrow\! clk\ n) \mid out\!\uparrow\! clk\ n)$$

This confirms the obvious – namely, that the cycle bahaviour of the program:

```
initial out := F
always @(posedge clk) out := in?¬out:out
```

is to conditionally complement the stored value depending on the value being input.