EVENT: Start with the library "mlp" using the compiled version.

```
; bcd.bm: a BCD code checker, with serial (bit) input.
;  . we have proved correctness w/ registers init to 0 (i.e.
;   reality) AND for ARBITRARY init values ('r0,'r1); although it
;   doesn't REALLY MEAN anything since the combinationals interpret
;   that as ones... 2/6/89
;  . we have also expressed the correctness hypothesis in various
;   ways, to analyze idiom trade-offs for expressing that the
;   "first 3 chars don't matter".
;

;;; CIRCUIT in SUGARED form:
```

```
#|
(setq sysd '(sy-BCD (x)
(Y0 R 'r0 x)
(Y1 R 'r1 Y0)
(Y2 S bor Y0 Y1)
(Yout S bnand x Y2)
))

(setq bcd '( |#
; BM DEFINITIONS and A2 LEMMAS, generated by BMSYSD:
; comb_bor.bm: Binary Or combinational element
; U7-DONE
```

DEFINITION:
bor $(u,\ v)$
$=$ **if** $(u = \texttt{0}) \wedge (v = \texttt{0})$ **then** `0`
else `1` **endif**

```
; Everything below generated by: (bmcomb 'bor '() '(x y))
```

DEFINITION:
s-bor $(x,\ y)$
$=$ **if** empty (x) **then** E
else a (s-bor (p (x), p (y)), bor (l (x), l (y))) **endif**

```
;; A2-Begin-S-BOR
```

THEOREM: a2-empty-s-bor
empty (s-bor $(x,\ y)$) $=$ empty (x)

THEOREM: a2-e-s-bor
(s-bor $(x,\ y) =$ E) $=$ empty (x)

THEOREM: a2-lp-s-bor
len (s-bor $(x,\ y)$) $=$ len (x)

THEOREM: a2-lpe-s-bor
eqlen (s-bor $(x,\ y)$, x)

THEOREM: a2-ic-s-bor
(len (x) $=$ len (y))
$\rightarrow$ (s-bor (i $(c\_x,\ x)$, i $(c\_y,\ y)$) $=$ i (bor $(c\_x,\ c\_y)$, s-bor $(x,\ y)$))

THEOREM: a2-lc-s-bor
$(\neg \text{empty}(x)) \rightarrow (\text{l}(\text{s-bor}(x,\ y)) = \text{bor}(\text{l}(x),\ \text{l}(y)))$

THEOREM: a2-pc-s-bor
$\text{p}(\text{s-bor}(x,\ y)) = \text{s-bor}(\text{p}(x),\ \text{p}(y))$

THEOREM: a2-hc-s-bor
$((\neg \text{empty}(x)) \wedge (\text{len}(x) = \text{len}(y)))$
$\rightarrow \quad (\text{h}(\text{s-bor}(x,\ y)) = \text{bor}(\text{h}(x),\ \text{h}(y)))$

THEOREM: a2-bc-s-bor
$(\text{len}(x) = \text{len}(y)) \rightarrow (\text{b}(\text{s-bor}(x,\ y)) = \text{s-bor}(\text{b}(x),\ \text{b}(y)))$

THEOREM: a2-bnc-s-bor
$(\text{len}(x) = \text{len}(y)) \rightarrow (\text{bn}(n,\ \text{s-bor}(x,\ y)) = \text{s-bor}(\text{bn}(n,\ x),\ \text{bn}(n,\ y)))$

```
;; A2-End-S-BOR

; eof:comb_bor.bm

; comb_bnand.bm: Binary Nand combinational element
; U7-DONE
```

DEFINITION:
$\text{bnand}(u,\ v)$
$= \quad$ **if** $(u = 0) \vee (v = 0)$ **then** 1
else 0 **endif**

```
; Everything below generated by: (bmcomb 'bnand '() '(x y))
```

DEFINITION:
$\text{s-bnand}(x,\ y)$
$= \quad$ **if** $\text{empty}(x)$ **then** E
else $\text{a}(\text{s-bnand}(\text{p}(x),\ \text{p}(y)),\ \text{bnand}(\text{l}(x),\ \text{l}(y)))$ **endif**

```
;; A2-Begin-S-BNAND
```

THEOREM: a2-empty-s-bnand
$\text{empty}(\text{s-bnand}(x,\ y)) = \text{empty}(x)$

THEOREM: a2-e-s-bnand
$(\text{s-bnand}(x,\ y) = \text{E}) = \text{empty}(x)$

THEOREM: a2-lp-s-bnand
len (s-bnand (x, y)) = len (x)

THEOREM: a2-lpe-s-bnand
eqlen (s-bnand (x, y), x)

THEOREM: a2-ic-s-bnand
(len (x) = len (y))
$\rightarrow$ (s-bnand (i (c_x, x), i (c_y, y)) = i (bnand (c_x, c_y), s-bnand (x, y)))

THEOREM: a2-lc-s-bnand
($\neg$ empty (x)) $\rightarrow$ (l (s-bnand (x, y)) = bnand (l (x), l (y)))

THEOREM: a2-pc-s-bnand
p (s-bnand (x, y)) = s-bnand (p (x), p (y))

THEOREM: a2-hc-s-bnand
(($\neg$ empty (x)) $\wedge$ (len (x) = len (y)))
$\rightarrow$ (h (s-bnand (x, y)) = bnand (h (x), h (y)))

THEOREM: a2-bc-s-bnand
(len (x) = len (y)) $\rightarrow$ (b (s-bnand (x, y)) = s-bnand (b (x), b (y)))

THEOREM: a2-bnc-s-bnand
(len (x) = len (y)) $\rightarrow$ (bn (n, s-bnand (x, y)) = s-bnand (bn (n, x), bn (n, y)))

```
;; A2-End-S-BNAND

; eof:comb_bnand.bm
```

DEFINITION:
topor-sy-bcd (ln)
= **if** ln = `'y0` **then** `0`
elseif ln = `'y1` **then** `0`
elseif ln = `'y2` **then** `1`
elseif ln = `'yout` **then** `2`
else `0` **endif**

DEFINITION:
sy-bcd (ln, x)
= **if** ln = `'y0`
then if empty (x) **then** E
else i (`'r0`, p (x)) **endif**
elseif ln = `'y1`

then if empty (x) **then** E
else i ('r1, sy-bcd ('y0, p (x))) **endif**
elseif ln = 'y2 **then** s-bor (sy-bcd ('y0, x), sy-bcd ('y1, x))
elseif ln = 'yout **then** s-bnand (x, sy-bcd ('y2, x))
else sfix (x) **endif**

```
;; A2-Begin-SY-BCD
```

THEOREM: a2-empty-sy-bcd
empty (sy-bcd (ln, x)) = empty (x)

THEOREM: a2-e-sy-bcd
(sy-bcd (ln, x) = E) = empty (x)

THEOREM: a2-lp-sy-bcd
len (sy-bcd (ln, x)) = len (x)

THEOREM: a2-lpe-sy-bcd
eqlen (sy-bcd (ln, x), x)

THEOREM: a2-pc-sy-bcd
p (sy-bcd (ln, x)) = sy-bcd (ln, p (x))

```
;; A2-End-SY-BCD

;;; Circuit CORRECTNESS /Paillet:

; BCD-bits defines a correct binary coded decimal,
;    b0 is most-significant.
```

DEFINITION:
bcd-bits ($b0$, $b1$, $b2$, $b3$) = (($b0$ = 0) $\lor$ (($b1$ = 0) $\land$ ($b2$ = 0)))

```
; BCD-Spec defines what the circuit is supposed to compute, by its
; last char.
```

DEFINITION:
bcd-spec (x) = bcd-bits (l (x), l (p (x)), l (p (p (x))), l (p (p (p (x)))))

```
; CORRECTNESS:
; original correctness statement:
```

THEOREM: bcd-correct
(($\neg$ empty (x))
$\land$ ($\neg$ empty (p (x)))
$\land$ ($\neg$ empty (p (p (x))))
$\land$ ($\neg$ empty (p (p (p (x)))))
$\to$ (bibo (l (sy-bcd ('yout, x))) = bcd-spec (x))

```
; alternative: implicitely using P(n+1) x /=e => P(n) x,..,Px,x /=e
;   ALSO works, with many more time and more CASES because BM
;   explores all the "obviously impossible" cases and takes some
;   time reducing the hyp to F.
```

THEOREM: bcd-correct2
$(\neg\ \mathrm{empty}\,(\mathrm{p}\,(\mathrm{p}\,(\mathrm{p}\,(x))))) \rightarrow (\mathrm{bibo}\,(\mathrm{l}\,(\text{sy-bcd}\,(\texttt{'yout},\ x))) = \text{bcd-spec}\,(x))$

```
; alternative: use the Pn operator instead of explicit P's in hyp:
;   ALSO works, with same number of CASES as explicit P's, but a
;   little bit more time due to the expansions of Pn required.
```

THEOREM: bcd-correct3
$(\neg\ \mathrm{empty}\,(\mathrm{pn}\,(\texttt{3},\ x))) \rightarrow (\mathrm{bibo}\,(\mathrm{l}\,(\text{sy-bcd}\,(\texttt{'yout},\ x))) = \text{bcd-spec}\,(x))$

```
; alternative: use LENgth > 3 instead of Pn or explicit P's.
;   ALSO works, with same number of CASES and time as Pn
```

THEOREM: bcd-correct4
$(\texttt{3} < \mathrm{len}\,(x)) \rightarrow (\mathrm{bibo}\,(\mathrm{l}\,(\text{sy-bcd}\,(\texttt{'yout},\ x))) = \text{bcd-spec}\,(x))$

```
;; one can give a CONTRIVED reading of Paillet's as pure streams:
```

DEFINITION:
bcd-bits-s $(b0,\ b1,\ b2,\ b3)$
$=\quad$ s-or (s-equal $(b0,$ s-const $(\texttt{0},\ b0)),$
s-and (s-equal $(b1,$ s-const $(\texttt{0},\ b1)),$ s-equal $(b2,$ s-const $(\texttt{0},\ b2))))$

DEFINITION:
bcd-spec-s $(x) =$ bcd-bits-s $(x,\ \mathrm{p}\,(x),\ \mathrm{p}\,(\mathrm{p}\,(x)),\ \mathrm{p}\,(\mathrm{p}\,(\mathrm{p}\,(x))))$

```
; but do NOT have the equality Yout = BCD-Spec-S because of the
; initial 3 values output by the circuit.  To get around that, we
; could:
;    - redefine BCD-Spec-S to be a big IF statement on the length
;      of x, and put in the explicit values computed by the
;      circuit, obtained by evaluating SY-BCD in rloop.
;      [ A major pain ]
;    - somehow express that we're not looking at the first 3 values
;      output.
;      We could use (RST (RST (RST ... ) = (RST (RST (RST ... ) but
;      clearly that would not work, since we currently have nothing
;      about RST.
; Note that this is essentially the PIPELINE problem.
; So far, the best approach seems to talk about L(sysd) which is
```

```
; most natural.
; NOTE: now that we've defined B (instead of RST) to deal with
; pipelines, and entered its theory, the above discussioon is moot.


; eof: bcd.bm
;))
```

Index