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

```
; countstut.bm: a simple clock counter, and a STUTTERING version of it,
;     to test Stutter formalizaton.
;
; . A2PC blows up on stuttering version, but that's been reduced to a
; a very trivial version (3 lines) which also blows up and on which one
; can see that BM loops without any rule: one of the equality hyp. is used
; in the opposite direction as the definition expansion.
; We use ADD-AXIOM on A2-PCs which blow.
;

;;; DEFINITION OF CIRCUITs:
```

```
#|
(setq A '(sy-A (x)
(Yout R 0 Y1)
(Y1 S Inc Yout)
))

(setq B '(sy-B (x)
(Yout R 0 Y1m)
(Y1m S Mux Yst Yout Y1)
(Y1 S Inc Yout)
(Yst R T Yst1)
(Yst1 S Not Yst)
))

(setq C '(sy-C (xst)
(Yout R 0 Y1m)
(Y1m S Mux xst Yout Y1)
(Y1 S Inc Yout)
))

(setq countstut '( |#
; BM DEFINITIONS and A2 LEMMAS, generated by BMSYSD:
; comb_inc.bm: INCrement combinational element
; U7-DONE
```

DEFINITION: $\mathrm{inc}\,(u) = (1 + u)$

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

DEFINITION:
s-inc (x)
= **if** empty (x) **then** E
else a (s-inc (p (x)), inc (l (x))) **endif**

```
;; A2-Begin-S-INC
```

THEOREM: a2-empty-s-inc
empty (s-inc (x)) = empty (x)

THEOREM: a2-e-s-inc
(s-inc (x) = E) = empty (x)

THEOREM: a2-lp-s-inc
$\text{len}(\text{s-inc}(x)) = \text{len}(x)$

THEOREM: a2-lpe-s-inc
$\text{eqlen}(\text{s-inc}(x),\ x)$

THEOREM: a2-ic-s-inc
$\text{s-inc}(\text{i}(c\_x,\ x)) = \text{i}(\text{inc}(c\_x),\ \text{s-inc}(x))$

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

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

THEOREM: a2-hc-s-inc
$(\neg\ \text{empty}(x)) \rightarrow (\text{h}(\text{s-inc}(x)) = \text{inc}(\text{h}(x)))$

THEOREM: a2-bc-s-inc
$\text{b}(\text{s-inc}(x)) = \text{s-inc}(\text{b}(x))$

THEOREM: a2-bnc-s-inc
$\text{bn}(n,\ \text{s-inc}(x)) = \text{s-inc}(\text{bn}(n,\ x))$

```
;; A2-End-S-INC
```

```
; eof:comb_inc.bm
```

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

DEFINITION:
sy-a (*ln*, *x*)
= **if** *ln* = `'yout`
  then if empty (*x*) **then** E
    else i (`0`, sy-a (`'y1`, p (*x*))) **endif**
  elseif *ln* = `'y1` **then** s-inc (sy-a (`'yout`, *x*))
  else sfix (*x*) **endif**

```
;; A2-Begin-SY-A
```

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

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

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

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

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

```
;; A2-End-SY-A
; BM DEFINITIONS and A2 LEMMAS, generated by BMSYSD:
; comb_mux.bm: Mux combinational element, i.e. "if".
; U7-DONE
```

DEFINITION:
mux ($u1$, $u2$, $u3$)
= **if** $u1$ **then** $u2$
else $u3$ **endif**

```
; everything below generated by: (bmcomb 'mux '() '(x1 x2 x3))
; with the EXCEPTIONS/HAND-MODIFICATIONS given below.
```

DEFINITION:
s-mux ($x1$, $x2$, $x3$)
= **if** empty ($x1$) **then** E
else a (s-mux (p ($x1$), p ($x2$), p ($x3$)), mux (l ($x1$), l ($x2$), l ($x3$))) **endif**

```
; SMUX-is-SIF can make things much simpler on occasions:
```

THEOREM: smux-is-sif
s-mux ($x1$, $x2$, $x3$) = s-if ($x1$, $x2$, $x3$)

EVENT: Disable smux-is-sif.

```
; We take advantage of SMUX-is-SIF for all inductive proofs. To do so we
; HAND-MODIFY the code generated by Sugar to replace all the hints by
;   - A2-EMPTY, A2-PC replace hint with: ((enable smux-is-sif))
```

```
;   - A2-LP, A2-IC, A2-HC, A2-BC: ((enable smux-is-sif) (disable len))
;   - A2-BNC: ((enable smux-is-sif) (disable bn len))

;; A2-Begin-S-MUX
```

THEOREM: a2-empty-s-mux
empty (s-mux (*x1*, *x2*, *x3*)) = empty (*x1*)

THEOREM: a2-e-s-mux
(s-mux (*x1*, *x2*, *x3*) = E) = empty (*x1*)

THEOREM: a2-lp-s-mux
len (s-mux (*x1*, *x2*, *x3*)) = len (*x1*)

THEOREM: a2-lpe-s-mux
eqlen (s-mux (*x1*, *x2*, *x3*), *x1*)

THEOREM: a2-ic-s-mux
((len (*x1*) = len (*x2*)) ∧ (len (*x2*) = len (*x3*)))
→ (s-mux (i (*c_x1*, *x1*), i (*c_x2*, *x2*), i (*c_x3*, *x3*))
= i (mux (*c_x1*, *c_x2*, *c_x3*), s-mux (*x1*, *x2*, *x3*)))

THEOREM: a2-lc-s-mux
(¬ empty (*x1*)) → (l (s-mux (*x1*, *x2*, *x3*)) = mux (l (*x1*), l (*x2*), l (*x3*)))

THEOREM: a2-pc-s-mux
p (s-mux (*x1*, *x2*, *x3*)) = s-mux (p (*x1*), p (*x2*), p (*x3*))

THEOREM: a2-hc-s-mux
((¬ empty (*x1*)) ∧ ((len (*x1*) = len (*x2*)) ∧ (len (*x2*) = len (*x3*))))
→ (h (s-mux (*x1*, *x2*, *x3*)) = mux (h (*x1*), h (*x2*), h (*x3*)))

```
;old:    ((DISABLE MUX S-MUX) (ENABLE H LEN) (INDUCT (S-MUX X1 X2 X3)))
```

THEOREM: a2-bc-s-mux
((len (*x1*) = len (*x2*)) ∧ (len (*x2*) = len (*x3*)))
→ (b (s-mux (*x1*, *x2*, *x3*)) = s-mux (b (*x1*), b (*x2*), b (*x3*)))

```
;old:    ((DISABLE MUX) (ENABLE B LEN) (INDUCT (S-MUX X1 X2 X3)))
```

THEOREM: a2-bnc-s-mux
((len (*x1*) = len (*x2*)) ∧ (len (*x2*) = len (*x3*)))
→ (bn (*n*, s-mux (*x1*, *x2*, *x3*)) = s-mux (bn (*n*, *x1*), bn (*n*, *x2*), bn (*n*, *x3*)))

```
;old: ((DISABLE MUX S-MUX))

;; A2-End-S-MUX

; eof:comb_mux.bm

;; already loaded in A: (LOAD "Comb/comb_inc.bm")
```

DEFINITION:
topor-sy-b (ln)
= **if** ln = `'yout` **then** `0`
  elseif ln = `'y1m` **then** `2`
  elseif ln = `'y1` **then** `1`
  elseif ln = `'yst` **then** `0`
  elseif ln = `'yst1` **then** `1`
  else `0` **endif**

DEFINITION:
sy-b (ln, x)
= **if** ln = `'yout`
  then if empty (x) **then** E
    else i (`0`, sy-b (`'y1m`, p (x))) **endif**
  elseif ln = `'y1m`
  then s-mux (sy-b (`'yst`, x), sy-b (`'yout`, x), sy-b (`'y1`, x))
  elseif ln = `'y1` **then** s-inc (sy-b (`'yout`, x))
  elseif ln = `'yst`
  then if empty (x) **then** E
    else i (**t**, sy-b (`'yst1`, p (x))) **endif**
  elseif ln = `'yst1` **then** s-not (sy-b (`'yst`, x))
  else sfix (x) **endif**

```
;; A2-Begin-SY-B
```

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

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

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

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

```
; See note at top of file.
;(PROVE-LEMMA A2-PC-SY-B (REWRITE)
;    (EQUAL (P (SY-B LN X)) (SY-B LN (P X)))
;    ((DISABLE S-MUX S-INC S-NOT A2-IC-S-MUX A2-IC-S-INC A2-IC-S-NOT)))
```

AXIOM: a2-pc-sy-b
p (sy-b (*ln*, *xst*)) = sy-b (*ln*, p (*xst*))

```
;; A2-End-SY-B

; BM DEFINITIONS and A2 LEMMAS, generated by BMSYSD:
;; already loaded: (LOAD "Comb/comb_mux.bm")
;; already loaded: (LOAD "Comb/comb_inc.bm")
```

DEFINITION:
topor-sy-c (*ln*)
= **if** *ln* = 'yout **then** 0
elseif *ln* = 'y1m **then** 2
elseif *ln* = 'y1 **then** 1
else 0 **endif**

DEFINITION:
sy-c (*ln*, *xst*)
= **if** *ln* = 'yout
then if empty (*xst*) **then** E
else i (0, sy-c ('y1m, p (*xst*))) **endif**
elseif *ln* = 'y1m **then** s-mux (*xst*, sy-c ('yout, *xst*), sy-c ('y1, *xst*))
elseif *ln* = 'y1 **then** s-inc (sy-c ('yout, *xst*))
else sfix (*xst*) **endif**

```
;; A2-Begin-SY-C
```

THEOREM: a2-empty-sy-c
empty (sy-c (*ln*, *xst*)) = empty (*xst*)

THEOREM: a2-e-sy-c
(sy-c (*ln*, *xst*) = E) = empty (*xst*)

THEOREM: a2-lp-sy-c
len (sy-c (*ln*, *xst*)) = len (*xst*)

THEOREM: a2-lpe-sy-c
eqlen (sy-c (*ln*, *xst*), *xst*)

```
; blows..
;(PROVE-LEMMA A2-PC-SY-C (REWRITE)
;    (EQUAL (P (SY-C LN XST)) (SY-C LN (P XST)))
;    ((DISABLE S-MUX S-INC A2-IC-S-MUX A2-IC-S-INC)))
; so TEMPORARILY:
```

AXIOM: a2-pc-sy-c
$\mathrm{p}\,(\text{sy-c}\,(ln,\ xst)) = \text{sy-c}\,(ln,\ \mathrm{p}\,(xst))$

```
;; A2-End-SY-C

;;; BEGIN: Circuit CORRECTNESS modulo Stuttering.

;; BEGIN: new 2nd order properties for combinationals.

;; END: new 2nd order properties for combinationals.

;;; Get STUTTER theory:
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  TH_STUTTER.BM
;;;
;;; This file contains Stutter theory for BM.  It is supposed to be
;;; loaded directly when needed (i.e. not general enough to be
;;; stored in Lib/mlp).
;;;

;;; Our current double P-recursive def. of Stutter:
;;; Originally, it came from THETA-PRF-79 (done while babysitting
;;; for Caroline...) followed by MUCH experimentation and fiddling.
```

DEFINITION:
stut-r $(x,\ y)$
= **if** empty (y) **then** x
  elseif empty $(\mathrm{p}\,(y))$ **then** $\mathrm{b}\,(x)$
  elseif $\mathrm{l}\,(\mathrm{p}\,(y))$ **then** stut-r $(x,\ \mathrm{p}\,(y))$
  else $\mathrm{b}\,(\text{stut-r}\,(x,\ \mathrm{p}\,(y)))$ **endif**

DEFINITION:
stut $(x,\ y)$
= **if** empty (y) **then** E
  elseif empty $(\mathrm{p}\,(y))$ **then** $\mathrm{a}\,(\mathrm{E},\ \mathrm{h}\,(x))$
  elseif $\mathrm{l}\,(\mathrm{p}\,(y))$ **then** $\mathrm{a}\,(\text{stut}\,(\mathrm{p}\,(x),\ \mathrm{p}\,(y)),\ \mathrm{l}\,(\text{stut}\,(\mathrm{p}\,(x),\ \mathrm{p}\,(y))))$
  else $\mathrm{a}\,(\text{stut}\,(\mathrm{p}\,(x),\ \mathrm{p}\,(y)),\ \mathrm{h}\,(\text{stut-r}\,(x,\ \mathrm{p}\,(y))))$ **endif**

```
; Stut-Induct inducts like stut, but without the case disjunction
; on LPx which is useless when we stutter on a line rather than an
; input. The resulting induction is not very different from a
; straight P induction, but it takes care of the empty Px case
; separately, and without bringing an elimination.
```

DEFINITION:
stut-induct (x)
= **if** empty (x) **then** 0
elseif empty $(\mathrm{p}(x))$ **then** 1
else stut-induct $(\mathrm{p}(x))$ **endif**

```
;; Properties of Stut:
```

THEOREM: stut-empty
empty (stut (x, y)) = empty (y)

THEOREM: stut-e
(stut $(x, y) = \mathrm{E}$) = empty (y)

THEOREM: stut-p
p (stut (x, y)) = stut (p (x), p (y))

```
;; Properties of Stut-R:
```

```
; Stut-R-E maybe shouldn't be enabled all the time, but when we're
; doing P inductions on Stut-R, this gives the base case.  The
; induction step is given by Stut-R-P.  Note that we don't have a
; full empty x hyp because Stut-R returns x and not sfix x in case
; y is empty...  Maybe we want to fix that at some point.
```

THEOREM: stut-r-e
stut-r $(\mathrm{E}, y) = \mathrm{E}$

THEOREM: stut-r-p
p (stut-r (x, y)) = stut-r (p (x), y)

THEOREM: stut-r-len
len $(x) <$ (1 + (len (y) + len (stut-r (x, y))))

THEOREM: stut-r-not-empty
(len $(y) <$ len (x)) $\rightarrow$ ($\neg$ empty (stut-r (x, y)))

```
; Stut-Rem removes the trailing Ts of y, but ignores Ly (like R)
; and leaves one T: this weird def, so it works like Stut-R needs!
```

DEFINITION:
stut-rem (y)
= **if** empty (y) **then** E
elseif empty $(\mathrm{p}(y))$ **then** y
elseif $\mathrm{l}(\mathrm{p}(y))$ **then** stut-rem $(\mathrm{p}(y))$
else y **endif**

THEOREM: stut-rem-empty
empty (stut-rem (x)) = empty (x)

THEOREM: stut-rem-len
len (stut-rem (x)) $<$ $(1 +$ len $(x))$

THEOREM: stut-rem-len2
$((\neg$ empty $(\mathrm{p}(x))) \wedge \mathrm{l}(\mathrm{p}(x))) \rightarrow$ (len (stut-rem (x)) $<$ len (x))

```
; Stut-Num counts the number of F in y, ignoring Ly, and starts
; w/ 1, like Stut.
```

DEFINITION:
stut-num (y)
= **if** empty (y) **then** 0
elseif empty $(\mathrm{p}(y))$ **then** 1
elseif $\mathrm{l}(\mathrm{p}(y))$ **then** stut-num $(\mathrm{p}(y))$
else $1 +$ stut-num $(\mathrm{p}(y))$ **endif**

THEOREM: stut-num-lessp
stut-num $(x) < (1 +$ len $(x))$

THEOREM: stut-num-eq-0
(stut-num $(x) = 0$) = empty (x)

```
; Requires a small induction.
```

THEOREM: stut-num-rem-len
$(\neg$ empty $(x)) \rightarrow$ (stut-num (p (stut-rem (x))) $<$ len (x))

```
; From Stut-Num and Bn we get a CLOSED FORM for Stut-R !!!
```

THEOREM: stut-r-closed
stut-r $(x,\ y)$ = bn (stut-num (y), x)

```
; Stut-inv is the key invariant property during Stuttering:
```

THEOREM: stut-inv
$((\neg \text{empty}(y)) \land (\text{len}(x) \not< \text{len}(y)))$
$\rightarrow \quad (\text{l}(\text{stut}(x,\ y)) = \text{h}(\text{stut-r}(x,\ \text{p}(\text{stut-rem}(y)))))$

```
; but we only want to use it during the non-stuttering induction
; step, and not in general so:
```

THEOREM: stut-inv0
$((\neg \text{empty}(y)) \land (\text{len}(x) \not< \text{len}(y)) \land (\neg \text{l}(y)))$
$\rightarrow \quad (\text{l}(\text{stut}(x,\ y)) = \text{h}(\text{stut-r}(x,\ \text{p}(\text{stut-rem}(y)))))$

EVENT: Disable stut-inv.

```
; Now we relate Stut-R for Py and P Rem Py, to get the key to the
; induction step on main Stut inductions, in the non-stuttering
; case.
;
; It's a BAD rewrite (i.e. expanding, potentially self-applicable),
; and so are the preliminary lemmas needed to build to it.  This
; is not just an unfortunate construction.  It's inherent, because
; we're essentially giving an alternate definition via a Stut-Rem
; recursion.  And definitions are expanding, self-applicable,
; rewrites.  We get around the problem by lucking out: the
; hypotheses are sufficient to prevent successful self-applic.
```

THEOREM: stut-r-indstep-num
$((\neg \text{empty}(x)) \land \text{l}(x))$
$\rightarrow \quad (\text{stut-num}(x) = (1 + \text{stut-num}(\text{p}(\text{stut-rem}(x)))))$

EVENT: Disable stut-r-indstep-num.

```
; OLD induction step prereq: not needed anymore.
;
;(prove-lemma Stut-R-indstep-Num-Rem (rewrite)
;(implies (and (not (empty y))
;       (not (empty (P y)))
;       (not (L (P y)))
;       )
;  (equal (Stut-Num (P (Stut-Rem (P y))))
```

```
; (sub1 (Stut-Num (P (Stut-Rem y))))
; ))
;((enable Stut-R-indstep-Num))
;)
;(disable Stut-R-indstep-Num-Rem)

; OLD induction step: not needed anymore.
;
;(prove-lemma Stut-indstep (rewrite)
;(implies (and (not (empty y))
;       (not (empty (P y)))
;       (not (L (P y)))
;       )
;  (equal (Stut-R x (P y))
; (B (Stut-R x (P (Stut-Rem (P y)))))
; ))
;((enable Stut-R-indstep-Num-Rem B-Bn-sub1)
; (disable Bn)
; )
;)
;(disable Stut-indstep) ; potentially self-looping...
;                       ; so we enable explicitely.


; NEW & GENERALIZED induction step hack , note: needs just ONE
; prereq! We're getting cleaner...
```

THEOREM: stut-r-indstep
$((\neg\ \text{empty}\,(y)) \wedge (\neg\ \text{l}\,(y)))$
$\rightarrow \quad (\text{stut-r}\,(x,\ y) = \text{b}\,(\text{stut-r}\,(x,\ \text{p}\,(\text{stut-rem}\,(y)))))$

EVENT: Disable stut-r-indstep.

```
; all the internal stuff shouldn't be needed outside:
```

EVENT: Disable stut-r.

EVENT: Disable stut-num.

EVENT: Disable stut-rem.

```
; Note: by leaving Stut, Stut-inv0, Stut-R-closed enabled, we get
;   the effect of an alternate recursive definition of Stut in the
;   most convenient form.  The remaining uncleanliness is that
;   Stut-R-indstep and Stut-R-closed match the same stuff, and need
;   to be used at different places in the main proof.  So far, we
;   survive by extreme cunning: they are in the right order, and
;   the hypothesis on Stut-R-indstep prevents wrong occurences.
;   This is neither clear nor robust...

;; eof: th_stutter.bm


;; BEGIN: ACTUAL Circuit CORRECTNESS modulo Stuttering.

; REVERSAL PROPERTY for sy-a:
```

THEOREM: sy-a-reversal
$(\neg\, \text{empty}\,(\text{bn}\,(n,\, \text{sy-a}\,(\texttt{'yout},\, \text{p}\,(x)))))$
$\rightarrow\quad (\text{h}\,(\text{b}\,(\text{bn}\,(n,\, \text{sy-a}\,(\texttt{'yout},\, x)))) = (1 + \text{h}\,(\text{bn}\,(n,\, \text{sy-a}\,(\texttt{'yout},\, \text{p}\,(x))))))$

THEOREM: count-ac-l
$\text{l}\,(\text{stut}\,(\text{sy-a}\,(\texttt{'yout},\, xst),\, xst)) = \text{l}\,(\text{sy-c}\,(\texttt{'yout},\, xst))$

```
; Now extending to strings.  For some unknow reason, compared to Funacc, we
; need BOTH splits here... Probably because of some weird non-triggering
; phenomenon in equality hyp usage.
```

THEOREM: apl-split-cout
$(\neg\, \text{empty}\,(x))$
$\rightarrow\quad (\text{sy-c}\,(\texttt{'yout},\, x) = \text{a}\,(\text{p}\,(\text{sy-c}\,(\texttt{'yout},\, x)),\, \text{l}\,(\text{sy-c}\,(\texttt{'yout},\, x))))$

THEOREM: apl-split-stuta
$(\neg\, \text{empty}\,(x))$
$\rightarrow\quad (\text{stut}\,(\text{sy-a}\,(\texttt{'yout},\, x),\, x)$
$\quad = \quad \text{a}\,(\text{p}\,(\text{stut}\,(\text{sy-a}\,(\texttt{'yout},\, x),\, x)),\, \text{l}\,(\text{stut}\,(\text{sy-a}\,(\texttt{'yout},\, x),\, x))))$

```
; and finally:
```

THEOREM: count-ac-correct
$\text{stut}\,(\text{sy-a}\,(\texttt{'yout},\, xst),\, xst) = \text{sy-c}\,(\texttt{'yout},\, xst)$

```
;; END: ACTUAL Circuit CORRECTNESS modulo Stuttering for A and C.
```

EVENT: Disable count-ac-l.

EVENT: Disable count-ac-correct.

```
; HCorr properties are the "Hand-Correctness" formulas... They are not
; necessary for the proof of Count-AB-L, but I'm trying to see if they help.
```

THEOREM: hcorr-ab-t
$((\neg \text{ empty}(x)) \wedge (\neg \text{ empty}(\text{p}(x))) \wedge \text{l}(\text{sy-b}(\texttt{'yst}, \text{p}(x))))$
$\rightarrow \quad (\text{l}(\text{sy-b}(\texttt{'yout}, x)) = \text{l}(\text{sy-b}(\texttt{'yout}, \text{p}(x))))$

EVENT: Disable hcorr-ab-t.

```
; obviously..
```

THEOREM: hcorr-ab-f
$((\neg \text{ empty}(x)) \wedge (\neg \text{ empty}(\text{p}(x))) \wedge (\neg \text{ l}(\text{sy-b}(\texttt{'yst}, \text{p}(x)))))$
$\rightarrow \quad (\text{l}(\text{sy-b}(\texttt{'yout}, x)) = (1 + \text{l}(\text{sy-b}(\texttt{'yout}, \text{p}(x)))))$

EVENT: Disable hcorr-ab-f.

```
; Count-AB-L succeeds with either HCorrs enabled, or the expansion hint
; therein.  The costs (time/clarity) seem equal.  In the future, if the
; effort involved in getting HCorrs is greater, the dichotomy may be useful.
```

THEOREM: count-ab-l
$\text{l}(\text{stut}(\text{sy-a}(\texttt{'yout}, x), \text{sy-b}(\texttt{'yst}, x))) = \text{l}(\text{sy-b}(\texttt{'yout}, x))$

THEOREM: apl-split-bout
$(\neg \text{ empty}(x))$
$\rightarrow \quad (\text{sy-b}(\texttt{'yout}, x) = \text{a}(\text{p}(\text{sy-b}(\texttt{'yout}, x)), \text{l}(\text{sy-b}(\texttt{'yout}, x))))$

THEOREM: apl-split-stuta2
$(\neg \text{ empty}(x))$
$\rightarrow \quad (\text{stut}(\text{sy-a}(\texttt{'yout}, x), \text{sy-b}(\texttt{'yst}, x))$
$= \quad \text{a}(\text{p}(\text{stut}(\text{sy-a}(\texttt{'yout}, x), \text{sy-b}(\texttt{'yst}, x))),$
$\text{l}(\text{stut}(\text{sy-a}(\texttt{'yout}, x), \text{sy-b}(\texttt{'yst}, x)))))$

THEOREM: count-ab-correct
$\text{stut}(\text{sy-a}(\texttt{'yout}, x), \text{sy-b}(\texttt{'yst}, x)) = \text{sy-b}(\texttt{'yout}, x)$

```
;; END: ACTUAL Circuit CORRECTNESS modulo Stuttering for A and B.
; eof: countstut.bm
; ))
```

Index