

Dealing with logic loops in CSGE

Joe English

1 Aug 2003

Abstract

Feedback loops in logic diagrams indicate system states. This paper demonstrates how to isolate such states so that they can be implemented in CSGE.

An example

Consider the the following snippet of Fortran code (part of a control system logic block):

```
S3 = Not (Or (RR, S5))  
S4 = Not (Or (SwPmd, S3))  
S5 = And (SwYsas, S4)  
Rcysa = S5
```

Translating this into CSGE in the “obvious” way gives a diagram something like Fig. 1 (In fact, I suspect the original Fortran code was originally based on a circuit diagram — composed mostly of NOR gates — that looked a bit like that.)

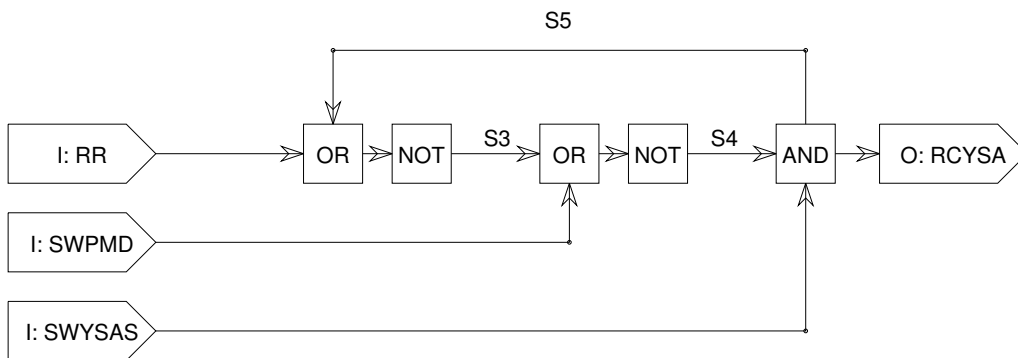


Figure 1: A logic loop

But there's a problem there! The output of the final AND block feeds back into itself as an input, leading to the dreaded *cyclic data dependency* error.

At this point, one may be tempted to just stick a 1-step delay in there and have done with it. But there's a better way.

The basic problem is that **S5** is defined in terms of itself. Well, not really. Actually, the *new* value of **S5** is defined in terms of the *old* value of **S5**. Whenever this situation occurs, the value in question must be treated as an internal state in CSGE. (That's exactly what happens when you stick a delay in.)

The best way to represent logic-valued internal states in CSGE is with the **LATCH** component. A **LATCH** has an internal state Q and takes two input signals: S (for "set", connected to the top input) and R (for "reset," connected at the bottom). The output equations are:

$$\begin{aligned} Q' &= (Q \vee S) \wedge \neg R \\ Y &= Q' \end{aligned}$$

and the truth table is:

S	R	Y
0	0	No change
1	0	1
0	1	0
1	1	0

Going back to the original Fortran code and rewriting it as a set of simultaneous equations (see p. 3 for a summary of the notation used), we have:

$$\begin{aligned} S'_5 &= YSAS \wedge S_4 \\ S_4 &= \neg(\text{SWPMD} \vee S_3) \\ S_3 &= \neg(\text{RR} \vee S_5) \end{aligned}$$

Then performing some simple algebra:

$$\begin{aligned} &S'_5 \\ = &YSAS \wedge S_4 && \text{(definition } S'_5) \\ = &YSAS \wedge \neg(\text{SWPMD} \vee S_3) && \text{(definition } S_4) \\ = &YSAS \wedge \neg(\text{SWPMD} \vee \neg(\text{RR} \vee S_5)) && \text{(definition } S_3) \\ = &YSAS \wedge (\neg\text{SWPMD} \wedge (\text{RR} \vee S_5)) && \text{(de Morgan)} \\ = &(\text{YSAS} \wedge \neg\text{SWPMD}) \wedge (\text{RR} \vee S_5) && \text{(associativity)} \\ = &\neg(\neg YSAS \vee \text{SWPMD}) \wedge (\text{RR} \vee S_5) && \text{(de Morgan)} \\ = &(S_5 \vee \text{RR}) \wedge \neg(\neg YSAS \vee \text{SWPMD}) && \text{(commutativity, twice)} \end{aligned}$$

This is the latch equation, once you substitute $Q = S_5$, $S = \text{RR}$, and $R = \neg YSAS \vee \text{SWPMD}$. Translating this into CSGE gives something like Fig. 2

Not only does this version have fewer blocks, it's easier to see what's going on: **RR** turns the latch on, and either of **SWPMD** or $\neg YSAS$ turn it off.

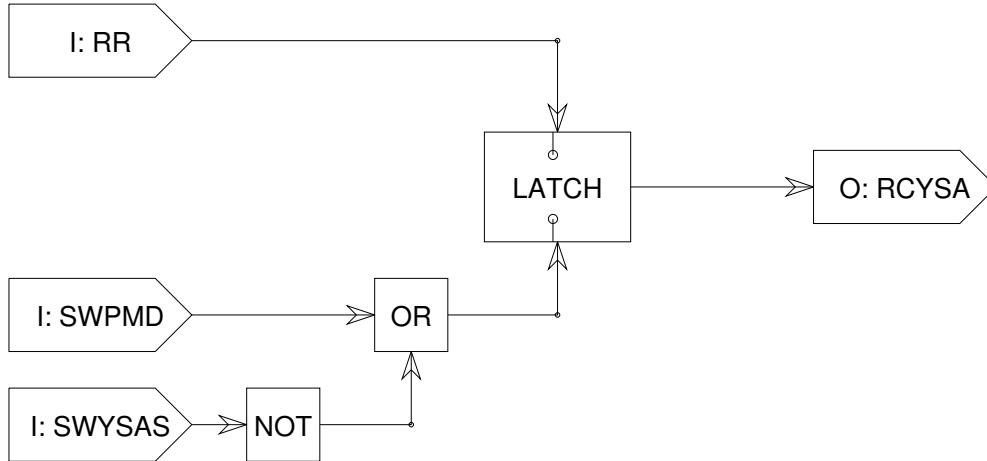


Figure 2: The logic loop fixed

Variations

What's not obvious from the diagram is what happens when the SET and RESET inputs are both true. For that you just have to know the rule: in the CSGE LATCH component, the RESET input takes precedence.

If you have a case where this makes a difference, but you want SET to take precedence instead, this can be achieved by swapping the roles of S and R and inverting the output (see Fig. 3).

We can prove that this is correct, again with simple algebra:

$$\begin{aligned}
 & \text{CSGE latch equation} \\
 \equiv & Q' = (Q \vee S) \wedge \neg R && \text{(definition)} \\
 \equiv & \neg q' = (\neg q \vee r) \wedge \neg s && \text{(substitute } Q = \neg q, S = r, R = s) \\
 \equiv & q' = \neg((\neg q \vee r) \wedge \neg s) && \text{(involution)} \\
 \equiv & q' = \neg(\neg(q \wedge \neg r) \wedge \neg s) && \text{(de Morgan)} \\
 \equiv & q' = \neg\neg((q \wedge \neg r) \vee s) && \text{(de Morgan again)} \\
 \equiv & q' = (q \wedge \neg r) \vee s && \text{(involution)}
 \end{aligned}$$

The last line is our desired latch equation, where s takes precedence over r .

More hidden states

Earlier on in the same Fortran code was the following:

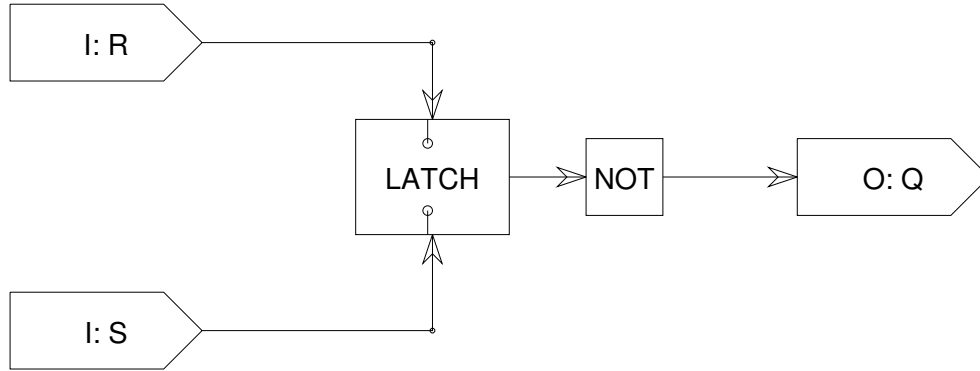


Figure 3: Alternate LATCH

```

IF( ABS(RSENSOR) .GT. YRLIM+EPS ) RR = 0
IF( ABS(RSENSOR) .LT. YRLIM-EPS ) RR = 1

```

Now at first glance there doesn't appear to be a cyclic dependency here: RR is defined solely in terms of RSENSOR, YRLIM, and EPS (and the latter two are presumably constants).

But there is a cycle there! Consider what happens if neither of the two IF statements are executed (i.e., if $yrlim - \epsilon < |rsensor| < yrlim + \epsilon$). Then RR is not changed; its new value depends (directly!) on its previous value.

This becomes apparent when you write RR as a single equation:

$$\begin{aligned}
 RR &= \begin{cases} 0, & |rsensor| > yrlim + \epsilon \\ 1, & |rsensor| < yrlim - \epsilon \\ RR_{prev}, & \text{otherwise} \end{cases} \\
 &= (RR_{prev} \vee (|rsensor| < yrlim - \epsilon)) \wedge \neg(|rsensor| > yrlim + \epsilon)
 \end{aligned}$$

This is a simple hysteresis. The current version of CSGE doesn't provide one out of the box, but you can build one with a LATCH (see Fig. 4)

Summary of notation

\vee is the logical *or* operator (disjunction), \wedge is the logical *and* operator (conjunction), and \neg is the logical *not* operator (negation).

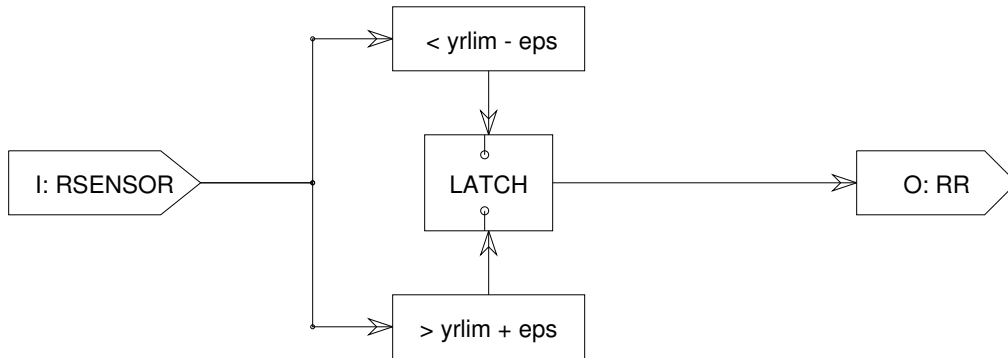


Figure 4: Hysteresis

The unary \neg operator has higher precedence than the two binary operators, so $\neg A \vee B$ means $((\neg A) \vee B)$, *not* $\neg(A \vee B)$. Often \wedge is treated as having higher precedence than \vee , so $A \wedge B \vee C$ would mean $(A \wedge B) \vee C$, but it's usually best to parenthesize all expressions that mix \wedge and \vee to avoid any confusion.

Lastly, here are some useful algebraic laws:

$x \wedge y = y \wedge x$	(\wedge -commutativity)
$(x \wedge y) \wedge z = x \wedge (y \wedge z)$	(\wedge -associativity)
$x \wedge x = x$	(\wedge -idempotency)
$x \wedge 0 = 0$	(zero)
$x \vee y = y \vee x$	(\vee -commutativity)
$(x \vee y) \vee z = x \vee (y \vee z)$	(\vee -associativity)
$x \vee x = x$	(\vee -idempotency)
$x \vee 1 = 1$	(unit)
$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$	(\wedge - \vee -distributivity)
$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	(\vee - \wedge -distributivity)
$\neg\neg x = x$	(involution)
$\neg(x \vee y) = \neg x \wedge \neg y$	(de Morgan)
$\neg(x \wedge y) = \neg x \vee \neg y$	(de Morgan)