Interrupt handler design with Petri nets, STGs and Petrify

Vintage 1996

Alex Yakovlev

(slides made by copying old slides from pdfs)

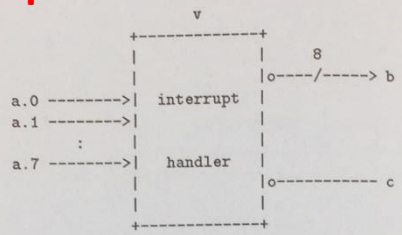
July 2019

Interrupt handler problem from Philips - 1996

Interrupt Handler

Our second problem is a simplified version of an interrupt handler as is found in, amongst other applications, microcontrollers.

Problem Specification



Interface:

* r is an input (reset);
* $a.0$ .. $a.7$ are inputs;
* b is an 8-bit output, passive handshake channel:
  - the data is single-rail encoded in output wires $b.j$, $(0 \leq j < 8)$;
  - the handshake is 4-phase along wire pair (br, bk);
  - wires $b.j$ must be stable ("data valid") when bk is high;
* $c = (cr, ck)$ is an undirected ("nonput"), passive, 4-phase handshake channel.

The interrupt handler is reset into its initial state when r is high, and starts handling interrupts after r has gone low. (Inputs br and cr are low when r is high.) Then it must be prepared for two kinds of request:

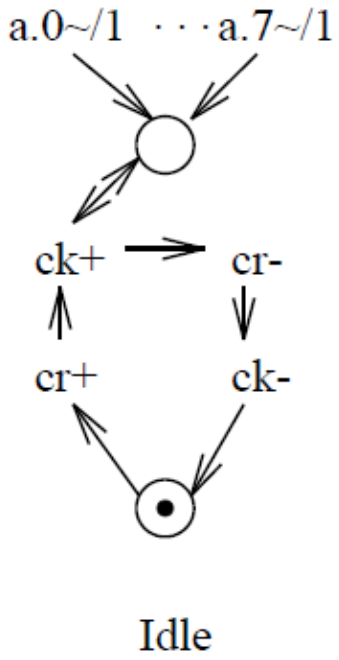
1. b_r : Return an 8-bit interrupt vector: bit $b.j$ indicates that at least one transition has occurred on $a.j$ since reset or since the previous request along b. Note that during the data-valid interval of channel b the data wires of b must be stable whereas the input wires $a.j$ may change.
2. c_r : Wait until a transition has occurred on one of the input wires $a.0..a.7$ (since reset or since previous request along b); then complete the handshake along c.

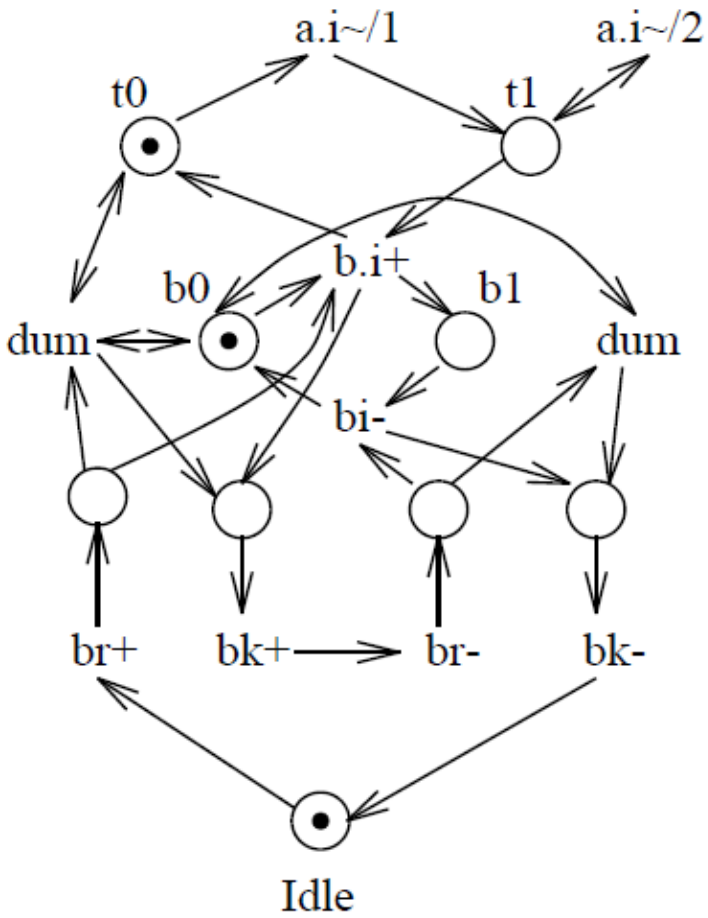
For simplicity we assume that handshakes along b and c are mutually exclusive. We are interested in a low-power solution. Hence, there should be no internal activity when the handler is in one of its quiescent states.

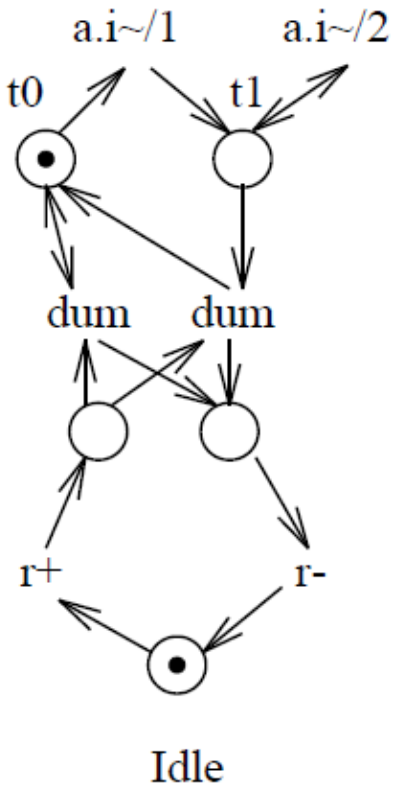
My solution: abstract

Interrupt handler. This design is based on the decomposition of the handler into 8 one-Bit Handlers $BH[i], i = 0..7$ and overall Control Circuit (CC). Each $BH[i]$ registers events on input $a.i$. Such an event is latched as a bit value 1 on the basis of "at least one transition occurs on $a.i$ since reset or since the previous request on b". The latching process is arbitrated (using a mutual exclusion logic) against a polling signal b_{in} or r_{in} generated by CC upon arrival of either b_r or r. The polling by b_{in} results in forwarding the registered event values onto the $b.i$ (vector) wires. The CC is responsible for overall synchronisation of the BH's and producing the ack-outputs b_k and c_k for commands b and c. The design satisfies the major requirements, such as (1) being a "low power" (command c produces no polling activity - it is just a simple "non-busy wait"), (2) keeping the $b.i$ stable while $a.i$ may change. The STG models of the $BH[i]$ and CC are implemented into logic gates by synthesis tool petrify.

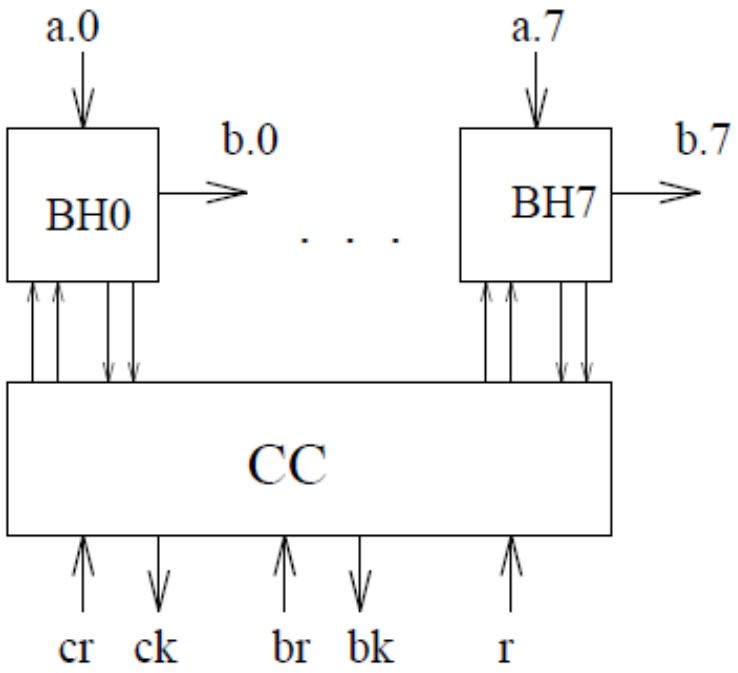
Towards Petri net specification





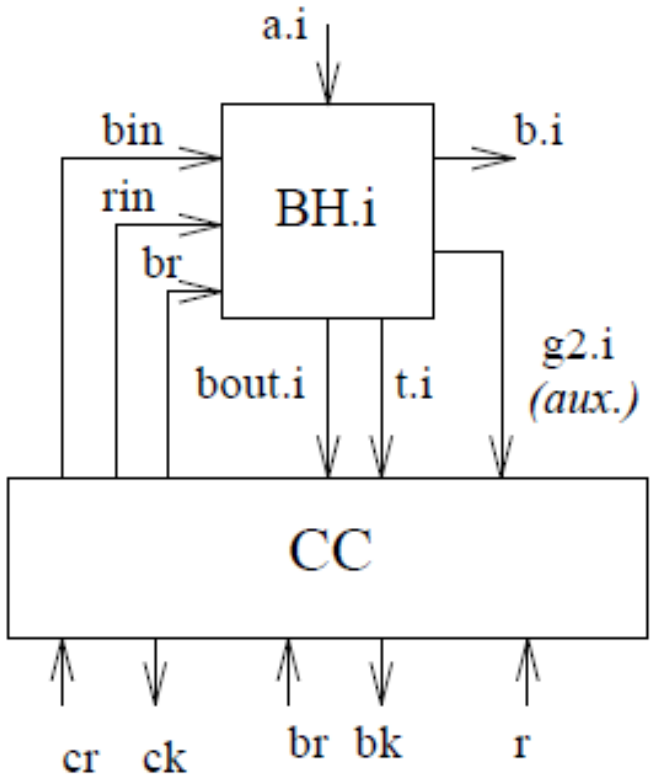


Overall structure



- $BH0, \ldots, BH7$ are one-bit handlers
- CC is a coordination-synchronisation circuit

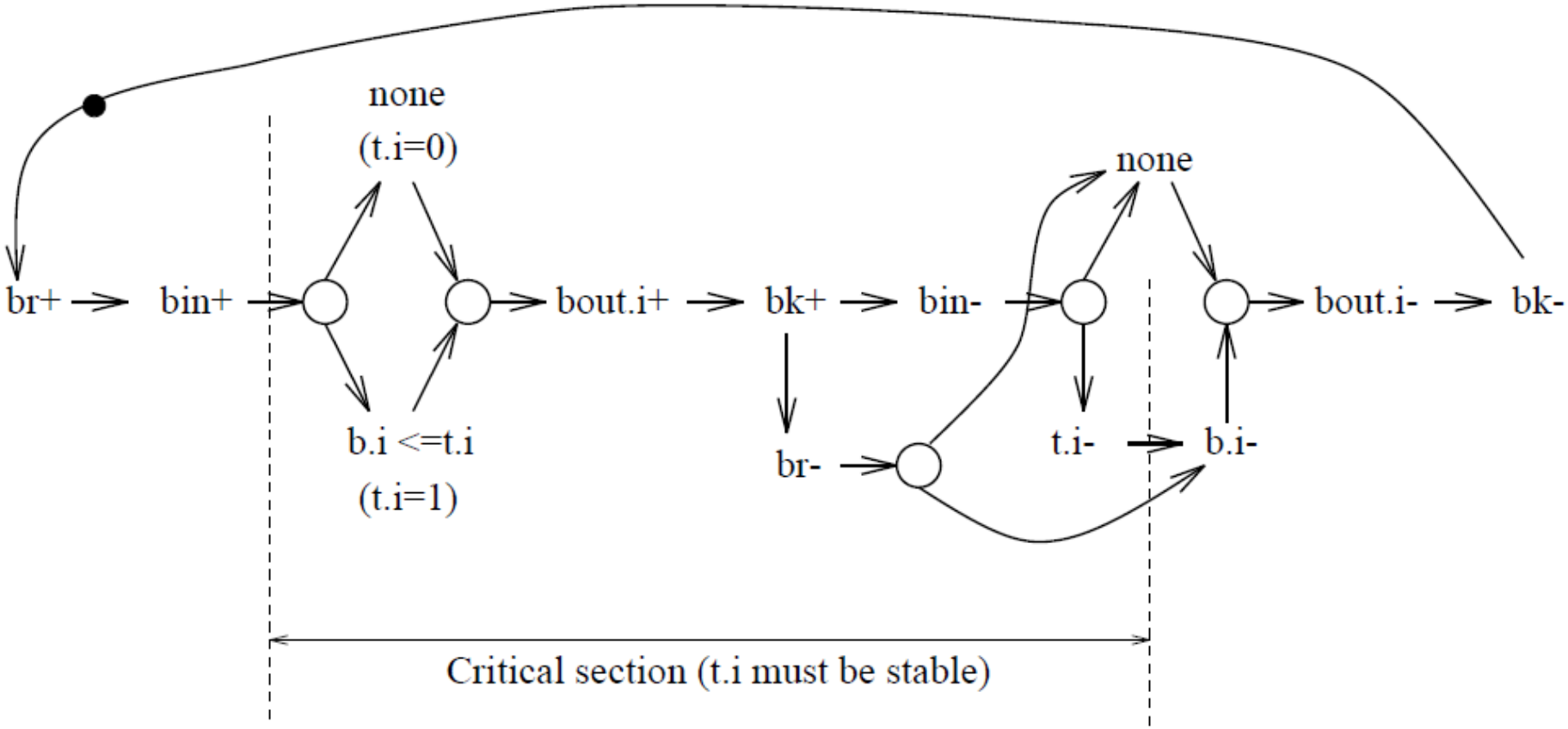
One-bit handler: Basic Structure



Basic Action:

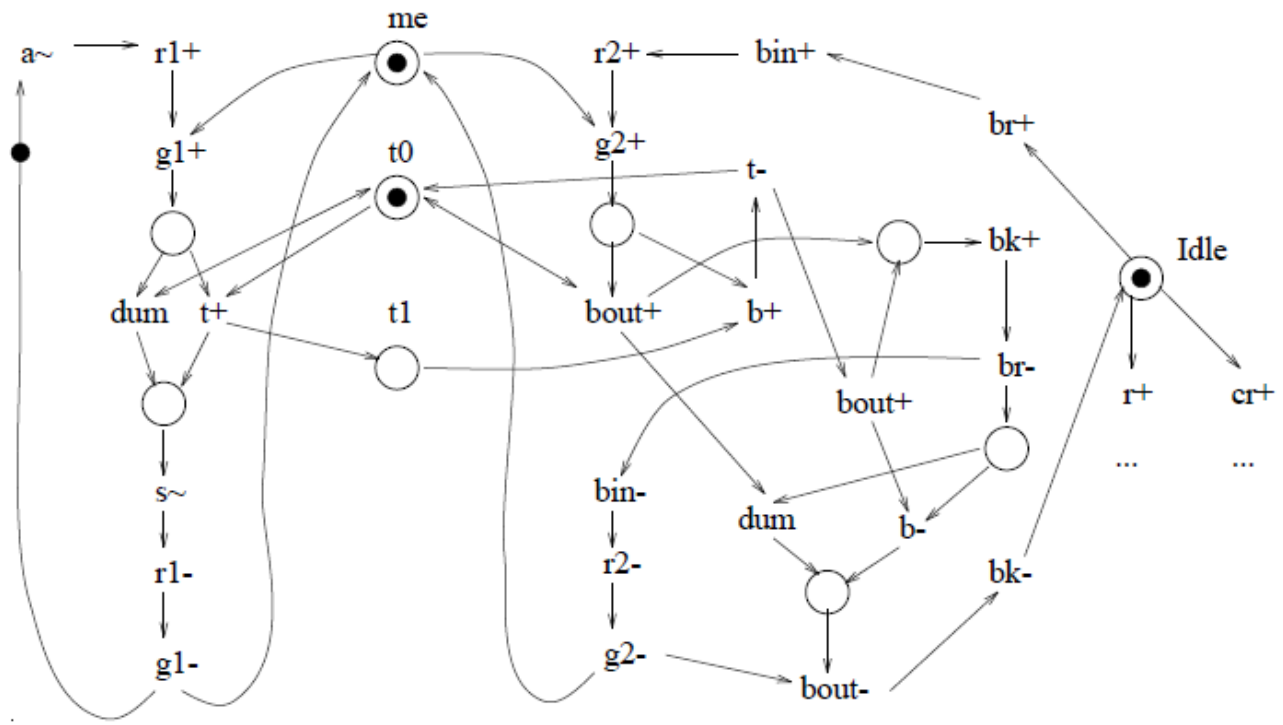
- Latch event on $a.i$ into $t.i$ before br or r
- Collect $t.i$ via OR logic to produce ck upon cr
- Rewrite $t.i$ into $b.i$ upon br, reset $t.i$

Basic schedule for the execution of command *b*



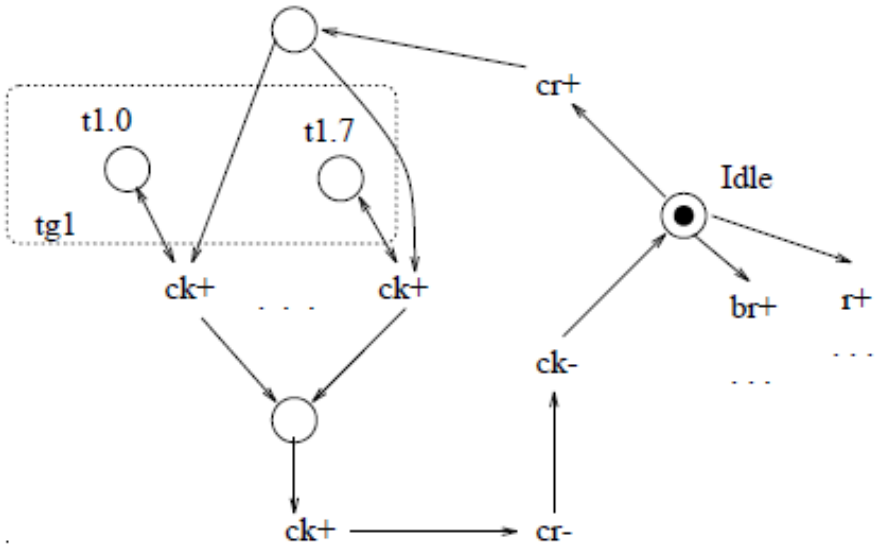
The critical section must be organised via *mutual exclusion* between latching events on $a.i$ into $t.i$ and processing b

Basic STG for command *b*



Timing assumption: The minimal interval between two adjacent changes of interupt signal $a.i$ should be at least equal to the delay of the *Mutex* element and latch for $t.i$

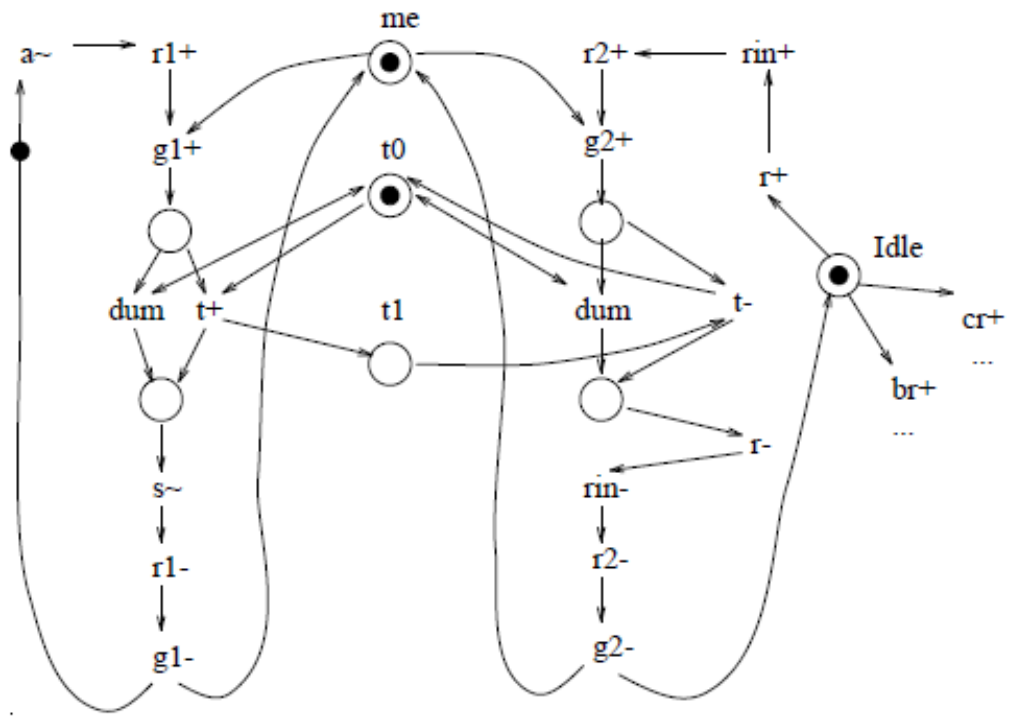
Basic STG for command *c*



Important:

- Signal $c.k$ is produced as soon as at least one of $t.i$ is set to 1 – this requires the gloabal signal $tg = t.0 + \ldots + t.7$.
- Ack-ing signal tg may be a problem for pure speed-independence (tg must be included into the STG specification).

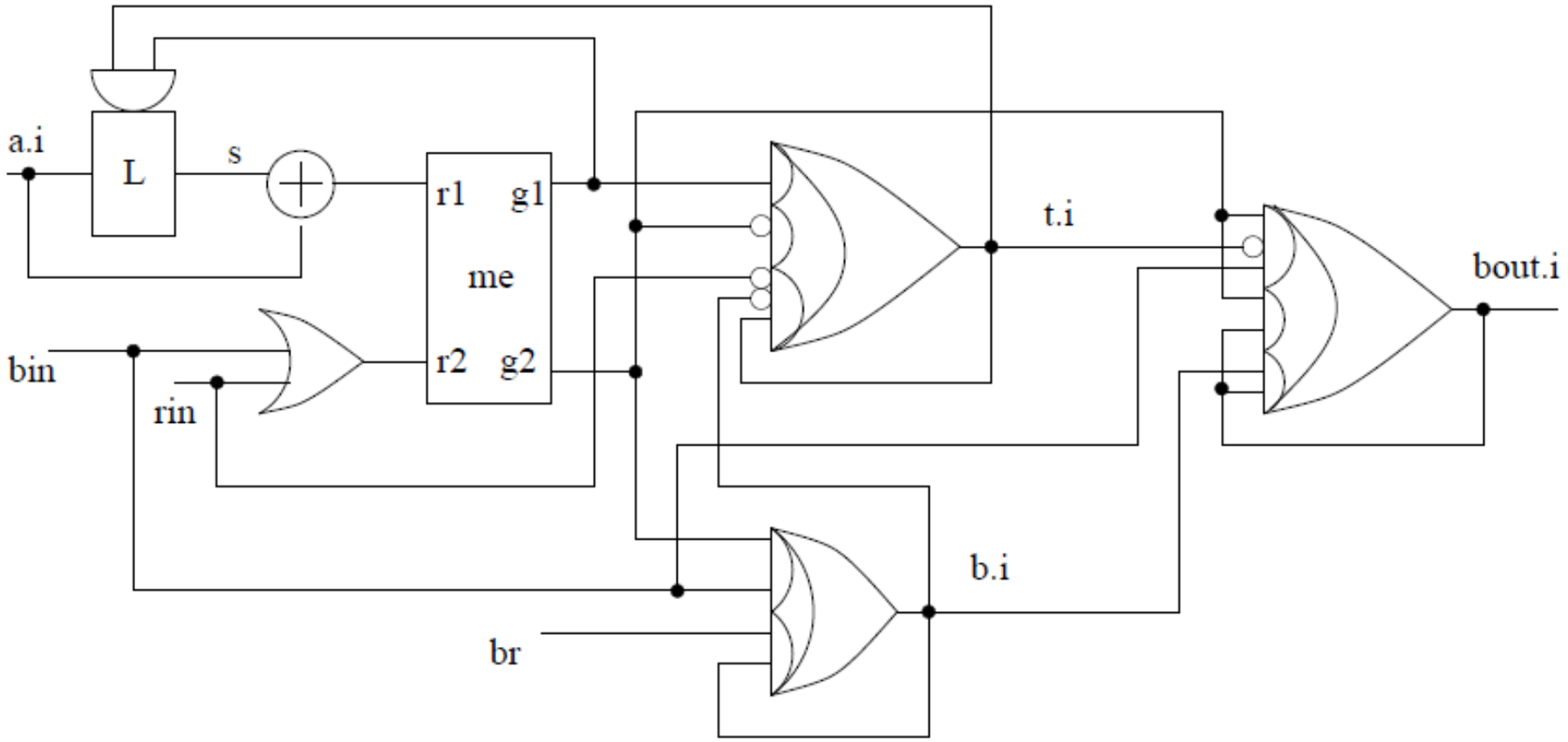
Basic STG for command *r*



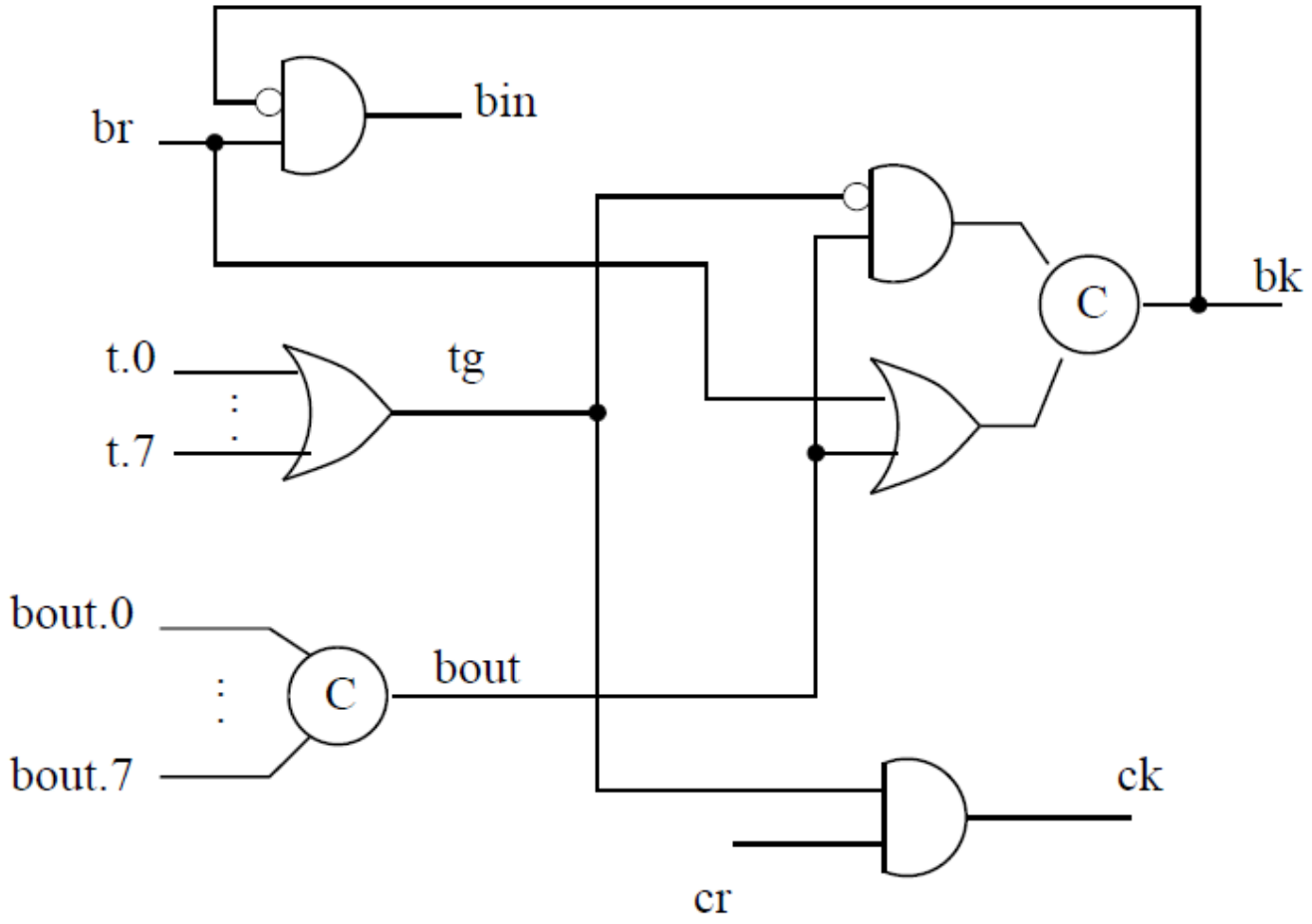
Timing assumption:

- The minimal interval between $r+$ and $r-$ should be sufficient for the circuit to reset its latches $t.i$ (including *Mutex* delays.
- Internally we may implement handshake rin and $rout$ (similar to bin and $bout$)

Implementation of one-bit handler



Implementation of Coordination Circuit



This circuit is not speed-independent wrt OR gate for tg

Speed-independent design (for two bit handlers)

Output from petrify:

```
[bk] =  bout*g2t + br*bk;
[ck] =  cr*tg;
[t1] =  !g12*t1 + g11 + t1*bin;
[t2] =  !g22*t2 + g21 + t2*bin;
[tg] =  t1 + t2;
[b1] =  br*b1 + tg*b1 + !r*g12*t1;
[b2] =  br*b2 + tg*b2 + !r*g22*t2;
[bin] =  !bk*bin + br*!bk*!g2t;
[bout1] =  g12*!t1*bin + b1;
[r11] =  a1*!s1 + !a1*s1 = XOR(a1,s1);
[r21] =  a2*!s2 + !a2*s2 = XOR(a2,s2);
[r12] =  r + bout + bin;
[r22] =  r + bout + bin;
[bout2] =  g22*!t2*bin + b2;
[bout] =  bout1*bout + bout1*bout2 + bout2*bout;
[g2t] =  g22*g2t + g12*g22 + g12*g2t = C(g12,g22);
[s1] =  a1*g11*t1*tg + !tg*s1 + !g11*s1 + !t1*s1;
[s2] =  a2*g21*t2*tg + !tg*s2 + !g21*s2 + !t2*s2;
```

Speed-independent design with C-gates (for two bit handlers)

Output from petrify:

```
[bk_C1] =  bout*g2t; [bk_C2] =  br;
[ck] =  cr*tg;
[t1_C1] =  g11; [t1_C2] =  !t1 + !r12 + !g2t + bin;
[t2_C1] =  g21; [t2_C2] =  !t2 + !r22 + !g2t + bin;
[tg] =  t1 + t2;
[b1_C1] =  g12*t1*bin; [b1_C2] =  !b1 + br + t1;
[b2_C1] =  g22*t2*bin; [b2_C2] =  !b2 + br + t2;
[bin_C1] =  br*!bk*!g2t; [bin_C2] =  !bk;
[bout1] =  b1 + g12*!t1*bin;
[r11] =  !a1*s1 + a1*!s1;
[r12] =  r + bout + bin;
[bout2] =  b2 + g22*!t2*bin;
[bout_aux_C1] = bout1; [bout_aux_C2] = bout2
[bout] = bout_aux + bout * tg;
[r21] =  !a2*s2 + a2*!s2;
[r22] =  r + bout + bin;
[g2t_C1] =  g22; [g2t_C2] =  g12;
[s1_C1] =  a1*g11*t1*tg; [s1_C2] =  !g11 + !t1 + !tg + a1;
[s2_C1] =  a2*g21*t2*tg; [s2_C2] =  !g21 + !t2 + !tg + a2;
```

Remarks on speed-independence

- Transitions of the OR-gate for tg must be ack-ed in both b and r commands

- Neither of BIH is allowed to leave the critical region until ALL of them completed it – hence synchronisation of $g2$ on $g2t$

- The last solution allows resetting of the Mutex'es on $(r2, g2)$ only after all $b.i$ are reset – this means a slight delay in latching events on $a.i$ into $t.i$ (the latter waits for release of Mutex).