Control Circuit Design

Kenneth S. Stevens

Electrical and Computer Engineering
University of Utah
Control Circuit Design

- Asynchronous control circuits interface between channels
- Each channel will have a particular protocol
  - Four-phase on one channel, two phase on another
  - 4-phase early narrow to 4-phase late narrow
- Controller dictates concurrency between channels
  - More concurrency usually means more complicated circuit
- Channels may have fan-in or fan-out

![Control Circuit Diagram](image)
Control Circuit Design

- These two controllers have fan-out and fan-in
  - May control latches/flops
  - May just steer control logic
Control Circuit Design

- There is an unbelievable richness to behaviors
- The controllers are usually small ($< 25$ gates)
- The behavior of these blocks require *formal methods* to specify behavior
  - Graph formalisms are common
    - State graphs, Petri Nets
    - Work well for the small size
    - Add visual richness
  - Process languages
    - Structural and behavioral components
    - Better for larger systems
Burst Mode Control

- State Graph and Burst-Mode model:

Mealy type state diagram

Primitive flow table

<table>
<thead>
<tr>
<th>Inputs a,b</th>
<th>Output c</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 01 11 10</td>
<td></td>
</tr>
<tr>
<td>s0  s2  -  s1</td>
<td>0</td>
</tr>
<tr>
<td>-  -  s3  s1</td>
<td>0</td>
</tr>
<tr>
<td>-  s2  s3  -</td>
<td>0</td>
</tr>
<tr>
<td>-  s4  s3  s5</td>
<td>1</td>
</tr>
<tr>
<td>s0  s4  -  -</td>
<td>1</td>
</tr>
<tr>
<td>s0  -  -  s5</td>
<td>1</td>
</tr>
</tbody>
</table>

Burst mode specification

a+b+/c+

- C-Element Specification
Burst Mode Control

1. Burst-mode operates in *fundamental mode*
   - Circuit internally stable before new inputs arrive

2. Allows restricted form of multiple input and output changes
   a. In stable state, an input set changes
   b. At completion, output(s) and state variables change
   c. Circuit must stabilize before new input(s) arrive

3. State graph specification similar to clocked machines

4. Input behavior describes possible environment behaviors
   - unreachable behavior defined

5. Output defines circuit response
Burst Mode Control

- Burst-mode state machines may change outputs concurrently or sequentially with outputs.

- In MEAT and earlier tools, outputs do not feed back.
- Later tools allow $Z$ to feed back as input.
Burst Mode Control
A Labeled Directed Graph \(\langle V, E, I, O, v_0, in, out \rangle\) with two labeling functions \(trans_i\) and \(trans_o\) where

- \(V\) is a finite set of vertices (states)
- \(E \subseteq V \times V\) is the set of edges (or transitions)
- \(I = \{x_0, \ldots, x_m\}\) is the set of inputs
- \(O = \{z_0, \ldots, z_n\}\) is the set of outputs
- \(in : V \rightarrow \{0, 1, *\}^m\) defines input value entering each state
- \(out : V \rightarrow \{0, 1\}^n\) defines output values entering each state
- \(trans_i : E \rightarrow 2^I\) defines input set changes on edges
- \(trans_o : E \rightarrow 2^O\) defines output set changes on edges
Burst Mode State Machine

- Burst description
  - For any edge $e = (u, v) \in E$, an input $x_i$ is in the transition if it must switch
    - $x_i \in trans_i(e)$ iff $in_i(u) \neq in_i(v)$
  - For any edge $e = (u, v) \in E$, an input is an extended don’t care if it might switch
    - $x_i \in trans_i(e) \land val(x_i, in_i(u)) = *$
  - For any edge $e = (u, v) \in E$, an output $z_i$ is in the transition if it switches
    - $z_i \in trans_o(e)$ iff $out_i(u) \neq out_i(v)$
Burst Mode State Machine

- Input bursts must obey **maximal set property**
  - No input burst leaving a given state can be a subset of another leaving the same state
  - $\forall (u, v), (u, w) \in E : trans_i(u, v) \subseteq trans_i(u, w) \Rightarrow v = w$

- Each state must contain an **essential transition** that must switch
  - $\forall (u, v) \in E : trans_i(u, v) \neq \emptyset \land \exists x_i \in trans_i(u, v) \land val(x_i, in_i(u)) \neq ^*$

- **Compatible Entry property**: Multiple entry values don’t violate other properties (maximal set, essential transition)
  - Simplified with **unique entry property** in presentation:
    - $\forall v \in V \exists$ unique $in_i(v)$
    - Otherwise you would split the state to create unique entry property
      - MEAT implemented with **compatible entry property**
      - 3D, Minimalist implement **unique entry property**
Burst Mode State Machine

- (a) violates the maximal set property, (b) is not a BM machine, violates state consistency, (c) a BM state for diagram b.

From Myer’s *Asynchronous Circuit Design* Fig. 4.6
Burst Mode State Machine

Sbuf Send Control specification:
: fsm sbuf-send-ctl
: init-in () ; value of inputs in initial state (optional)
: init-out () ; value of outputs in initial state (optional)
: init-state 0 ; initial state (optional)

;;:mex (a1 a2) ; sets of mutually exclusive inputs (optional)
: in (Deliver Begin-Send Ack-Send) ; list of input variables
: out (Latch-Addr IdleBAR Send-Pkt) ; list of output variables

: state 0 (Deliver) 1 (IdleBAR * Latch-Addr)
: state 1 (Deliver~) 2 ()
: state 2 (Begin-Send) 3 (Latc-Addr~)
: state 3 (Begin-Send~) 4 (Send-Pkt)
: state 4 (Ack-Send) 5 (Send-Pkt~)
: state 5 (Ack-Send~) 0 (IdleBAR~)
: state 4 (Deliver) 6 ()
: state 6 (Deliver~ * Ack-Send) 7 (Send-Pkt~ * Latch-Addr)
: state 7 (Ack-Send~) 2 ()
Burst Mode State Machine

lisp
> (load "load-meat")
> (meat "sbuf-send-ctl.data")
Max Compatibles: ((0 5) (1 2 7) (3 4) (6))
Enter State Set: '((0 5) (1 2 7) (3 4) (6))

SOP for "Y1":
  18: DELIVER + Y1*BEGIN-SEND~

SOP for "Y0":
  28: BEGIN_SEND + Y0*ACK-SEND~ + Y0*DELIVER

SOP for LATCH-ADDR:
  12 Y1*Y0~

SOP for IDLEBAR:
  30 ACK-SEND + BEGIN-SEND + Y0 + Y1

SOP for SEND-PKT:
  12 Y0*BEGIN-SEND~

HEURISTIC TOTAL FOR THIS ASSIGNMENT: 100
Burst Mode State Machine

Sbuf Read Control specification:

“Directed Don’t care” – Req will change across states 3 to 5

:fsm sbuf-ram-write
:in (Req Precharged Done WENin WSLDin)
:out (Ack PRBAR WSEN WEN WSLD)
:init-in ()
:init-out (WSEN)
:init-state 0
:state 0 (Req * Precharged * Done~) 1 (PRBAR)
:state 1 (Precharged~) 2 (WEN)
:state 2 (Done * WENin) 3 (WEN~ * WSEN~ * Ack)
:state 3 (WENin~ * Req~ + WENin~ * Req) 4 (WSLD)
:state 4 (WSLDin * Req~ + WSLDin * Req) 5 (WSLD~)
:state 5 (WSLDin~ * Req~) 0 (PRBAR~ * WSEN * Ack~)
Burst Mode Synthesis: 3D

- A second tool we will use is 3D
  - From Ken Yun and Steve Nowick, Stanford
- Similar to MEAT

To run:

3D filename.nounc

(If you run with -f argument it will synthesize to dynamic gates)
Burst Mode Synthesis

- 3D files MUST end with .nounc (???)

- following is a c-element specification:

```plaintext
;;; c-element design
input a 0 ; input signal and voltage
input b 0

output c 0 ; output signal and voltage

;;; signals labeled with * are directed don’t cares
0 1 a+ b+ | c+
1 0 a- b- | c-

Result is in file <filename>.eqn
```
Petri Nets

- Current methods – flow tables and state machines – describe state transitions
- Newer methods describe interface behavior across states
  - Describe interface transitions
  - Internal state is *synthesized*
- Petri-Nets is one of the primary specifications
  - Signal Transition Graph (STG) variant used for async circuits
Petri Nets

A directed graph $\langle P, T, F, M_0 \rangle$ where

- $P$ is a set of places
- $T$ is a set of transitions
- $F$ is the Flow Relationship pair
  - $F \subseteq (P \times T) \cup (T \times P)$
- $M$ is a marking, mapping to natural numbers $M : P \rightarrow N$
  - $M_0$ is the initial state

A preset of transition $t \in T$ denoted $\bullet t$ is set of places connected to $t$:

- $\bullet t = \{ p \in P | (p, t) \in F \}$

A transition $t \in T$ is enabled if all preset places are marked

A transition fires by removing a token in preset and placing one in each postset.
Petri Nets

Formal representation of a C-Element

The marked graph above happens to be conflict free, no cases of:
Petri Nets

Formal representation of a C-Element

The marked graph above happens to be conflict free, no cases of:
Petri Nets

Formal representation of a C-Element

The marked graph above happens to be *conflict free*, no cases of:
Petri Nets

Formal representation of a C-Element

The marked graph above happens to be *conflict free*, no cases of:
Petri Nets

Formal representation of a C-Element

The marked graph above happens to be conflict free, no cases of:
Petri Nets

Formal representation of a C-Element

The marked graph above happens to be conflict free, no cases of:
Petri Nets

Formal representation of a C-Element

The marked graph above happens to be conflict free, no cases of:

\[
\text{UofU 26}
\]
Petrify

- Petrify synthesizes asynchronous control

```
.model  c-element
.inputs a b
.outputs c
.graph
#
 a+ c+
b+ c+
c+ a- b-
a- c-
b- c-
c- a+ b+
# Initial token markings
.marking {<c-,a+>  <c-,b+>}
#.capacity <a1-,ack+>=2  <req+,r1+>=2
.end
```
Petrify

- Can look at petri net graphically:

xastg c-element.g
Petrify

- Synthesize with the following command:

```bash
petrify -cg c-element.g -o c-element.gg \
   -log c-element.log -eqn c-element.eqn
```

- The output graph is placed in `c-element.gg`
- The `-log` file contains synthesis results and some choices
- The `-eqn` file contains the selected circuit
- Petrify has *many* options - see the manual
Petrify

SBuf Send Control
Petrify
Synthesized SBuf Send Control (.slowenv timing assumption)
Petrify

- Some inputs in parallel, others in series

- $csc0$ - must occur before input $acksend$

Resulting logic:

\[
[latchaddr] = csc0' csc1 (beginsend' sendpkt' \\
\quad + deliver' acksend); \\
[idlebar] = csc1 + acksend; \\
[sendpkt] = beginsend' acksend' csc0 \\
\quad + sendpkt (deliver + acksend'); \\
[csc0] = deliver' csc0 (csc1 + sendpkt) + beginsend; \\
[csc1] = csc1 (csc0' + acksend') + deliver;
\]
Petrify vs MEAT

- MEAT solution:

\[
\begin{align*}
\text{idlebar} &= \text{acksend} + \text{beginsend} + y_0 + y_1 \\
\text{sendpkt} &= y_0 \cdot \text{beginsend} \\
\text{latchaddr} &= y_1 \cdot y_0 \\
y_0 &= \text{beginsend} + (y_0 \cdot (\text{acksend} + \text{deliver})) \\
y_1 &= \text{deliver} + (y_1 \cdot \text{beginsend})
\end{align*}
\]

- Petrify solution:

\[
\begin{align*}
\text{idlebar} &= \text{csc1} + \text{acksend} \\
\text{sendpkt} &= (\text{beginsend} \cdot \text{acksend} \cdot \text{csc0}) \\
&\quad + (\text{sendpkt} \cdot (\text{deliver} + \text{acksend})); \\
\text{latchaddr} &= \text{csc0} \cdot \text{csc1} \cdot ((\text{beginsend} \cdot \text{sendpkt}) \\
&\quad + (\text{deliver} \cdot \text{acksend})); \\
\text{csc0} &= (\text{deliver} \cdot \text{csc0} \cdot (\text{csc1} + \text{sendpkt})) + \text{beginsend}; \\
\text{csc1} &= (\text{csc1} \cdot (\text{csc0} + \text{acksend})) + \text{deliver};
\end{align*}
\]
Synthesis Results

Which solution is better – MEAT or Petrify?
Synthesis Results

Which solution is better – MEAT or Petrify?

MEAT is certainly smaller, probably faster

What about hazards and timing constraints?
Synthesis Results

Which solution is better – MEAT or Petrify?

**MEAT is certainly smaller, probably faster**

What about hazards and timing constraints?

- Both require fundamental mode assumption
- Does logic decomposition create new hazards?
- Do the inverters add new hazards?

Relative Timing and Formal Verification will give us that answer.
Synthesis Summary

- We will be using three tools: meat, 3D, petrify
- These all synthesize asynchronous controllers
- Tools are in:
  /uusoc/facility/cad_tools/Async/bin/
- See the documentation for various tools in:
  /uusoc/facility/cad_tools/Async/doc/