Generalised Asynchronous Arbiter

Stanislavs Golubcovs, Andrey Mokhov, Alex Bystrov, Danil Sokolov, Alex Yakovlev

27 June 2019, Aachen
Generalised Asynchronous Arbiter

Stanislavs Golubcovs, Andrey Mokhov, Alex Bystrov, Danil Sokolov, Alex Yakovlev

27 June 2019, Aachen
Why a new arbiter?
Why arbiters?

**Arbiters** orchestrate access to shared resources:

– Memory, where multiple processors meet
– Road intersections, where multiple cars meet
– Ice-cream shops, where multiple overheated people of Aachen meet
Why arbiters?

Arbiters orchestrate access to shared resources:
- Memory, where multiple processors meet
- Road intersections, where multiple cars meet
- Ice-cream shops, where multiple overheated people of Aachen meet

Good properties that we want from arbiters:
- Low latency: “I need an ice-cream as soon as possible, please!”
Why arbiters?

**Arbiters** orchestrate access to shared resources:
- Memory, where multiple processors meet
- Road intersections, where multiple cars meet
- Ice-cream shops, where multiple overheated people of Aachen meet

Good properties that we want from arbiters:
- Low latency: “I need an ice-cream *as soon as possible*, please!”
- Deadlock freedom: “What do you mean you *run out* of ice-cream?!”
Why arbiters?

Arbiters orchestrate access to shared resources:
- Memory, where multiple processors meet
- Road intersections, where multiple cars meet
- Ice-cream shops, where multiple overheated people of Aachen meet

Good properties that we want from arbiters:
- Low latency: “I need an ice-cream as soon as possible, please!”
- Deadlock freedom: “What do you mean you run out of ice-cream?!"
- Fairness: “Hey, that’s my ice-cream!”
Why arbiters?

**Arbiters** orchestrate access to shared resources:

– Memory, where multiple processors meet
– Road intersections, where multiple cars meet
– Ice-cream shops, where multiple overheated people of Aachen meet

Good properties that we want from arbiters:

– Low latency: “I need an ice-cream *as soon as possible*, please!”
– Deadlock freedom: “What do you mean you *run out* of ice-cream?!”
– Fairness: “Hey, that’s *my* ice-cream!”
– Constraints: “Can I have *one pistachio and one non-vanilla* scoop?”
Why a new arbiter?

Arbiters orchestrate access to shared resources:
- Memory, where multiple processors meet
- Road intersections, where multiple cars meet
- Ice-cream shops, where multiple overheated people of Aachen meet

Good properties that we want from arbiters:
- Low latency: asynchronous request capture, event-driven
- Deadlock freedom: formally verified using Petri nets
- Fairness:
- Constraints: generalised decision making by combinational logic
What’s the main challenge?
What is the main challenge?

What we want
What is the main challenge?

What we have

What we want
What is the main challenge?

What we have: Design, synthesis

What we want:
What is the main challenge?

What we have

Design, synthesis

What we want
What is the main challenge?

Mutex gate

What we have

What we want
What is the main challenge?

Mutex gate

What we have

Complex arbiter

What we want
Mutual exclusion (mutex) element

Circuit and PN specification

Standard implementation
The main idea
The main idea

What we want

Input Channel 1

... Input Channel n

Grant Controller

Lock Controller

Top-level structure

req1

req2

req3

... req

ack1

ack2

ack3

... ack
Initial state
Request 1 arrives, activates the Lock
Request 2 arrives, just in time to go through
Request 3 arrives too late
Make decision, grant request 2
Release the Lock, allowing Request 3 in
Request 2 is released, but too late
Grant controller decides there is nothing to do
Lock is released; falling Request 2 goes through
Request 1 is (finally) granted
The main idea

What we want

Top-level structure

Input Channel 1

Input Channel n

Lock Controller

Grant Controller
Input channels

The **request interface** of the arbiter:

– Accepts arbitration request changes
– Activates the Lock controller to start the new arbitration round
– Provides the current request state to the Grant controller
Lock controller

**Locks all input channels** to create the locked request state

- Initialised by one (or more) input channels
- Activates grant controller when all request states are ready
- Unlocks all input channels when the grant controller has finished
Grant controller

Grant requests subject to constraints
– initialised by one (or more) input channels
– Activated when the state of all requests is ready
– Unlocks all input channels once the decision has been made
Mutex elements in action
Input Channel in Action (Case 1)

Initial state
Input Channel in Action (Case 1)

XOR element registers the change of input request
Request change has won the arbitration
Initialise the lock controller with \textbf{init}
Input Channel in Action (Case 1)

The second d-latch propagates request state
MUTEX is now ready to accept lock signal, rstate is stable
Input Channel in Action (Case 1)

MUTEX accepts lock, initialises ready to compute
Some other channel has initialised the arbitration
Input Channel in Action (Case 2)

Any req changes will not affect rstate until the end of computation
Design issues
Verification flow

Convert to Petri Net

Verify with Punf, Mpsat

Hazard and deadlock traces
Scaling the Lock Controller

Timing assumption: all OR-gates should settle faster than the delay of the C-element tree and the grant controller.
Performance

Design scales linearly with the increased number of input channels. Latency scales logarithmically, because of the tree structures.
Example: 1-of-3 Arbiter

Arbitration: 1 resource is shared among 3 users.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>g1'</th>
<th>g2'</th>
<th>g3'</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Other combinations: 0 0 0

\[
g1' = r_1 \cdot (\overline{g_3} \cdot (\overline{g_2} + r_2) + \overline{r_3} \cdot g_3)
\]
\[
g2' = r_2 \cdot (\overline{r_1} \cdot (g_3 + r_3) + g_2)
\]
\[
g3' = r_3 \cdot (\overline{r_1} \cdot \overline{r_2} + g_3)
\]
Example: 2-of-3 Arbiter

Arbitration: 2 resources are shared among 3 users.

<table>
<thead>
<tr>
<th></th>
<th>$r_1$</th>
<th>$g_1$</th>
<th>$r_2$</th>
<th>$g_2$</th>
<th>$r_3$</th>
<th>$g_3$</th>
<th>$g_1'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Other combinations: 0

<table>
<thead>
<tr>
<th></th>
<th>$r_1$</th>
<th>$g_1$</th>
<th>$r_2$</th>
<th>$g_2$</th>
<th>$r_3$</th>
<th>$g_3$</th>
<th>$g_2'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Other combinations: 0

$g_1' = r_1 \cdot r_2 \cdot g_2 \cdot r_3 \cdot g_3$

$g_2' = r_2 \cdot (r_1 \cdot g_3 \cdot r_3 + g_2)$

$g_3' = r_3 \cdot (r_1 \cdot r_2 + g_3)$

<table>
<thead>
<tr>
<th></th>
<th>$r_1$</th>
<th>$g_1$</th>
<th>$r_2$</th>
<th>$g_2$</th>
<th>$r_3$</th>
<th>$g_3$</th>
<th>$g_3'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Other combinations: 0
Example: Nacking Arbiter

Nacking actively acknowledges that the resource is occupied, the user can attempt to take some other action rather than waiting for grant.

\[
\begin{array}{cccc|cccc}
  r1 & g1 & n1 & r2 & g2 & n2 & g1' & n1' & g2' & n2' \\
  1 & X & X & 0 & X & X & 1 &  &  &  \\
  1 & X & X & X & 0 & X & 1 &  &  &  \\
  1 & X & X & 1 & 1 & X & 1 &  &  &  \\
  0 & X & X & 1 & X & X & 1 &  &  &  \\
  X & X & X & 1 & 1 & X & 1 &  &  &  \\
  1 & X & X & 1 & 0 & X & 1 &  &  &  \\
  \text{other combinations} & 0 & 0 & 0 & 0 & 0 &  &  &  &  \\
\end{array}
\]

\[
g1' = r1 \cdot \overline{r2} \cdot g2 \cdot r3 \cdot g3
\]

\[
g2' = r2 \cdot (r1 \cdot g3 \cdot r3 + g2)
\]

\[
g3' = r3 \cdot (r1 \cdot r2 + g3)
\]
Main results

Generalised asynchronous arbiter:

– Low latency, deadlock-free, arbitrary decision logic
– Clean top-level decomposition: Input Channels, Lock and Grant
– Same general structure can be used to design different arbiters
– Grant controller can be synthesised automatically

Problem solved?

– In practical terms, the answer is “Probably Yes”
– If you are a theoretician then the answer is “No”: still no synthesis!