Towards High-Level Specification & Synthesis of Dynamic Process Logic

Oliver Diessel
Usama Malik
Keith So

School of Computer Science and Engineering,
The University of New South Wales, Australia

George Milne

School of Computer Science & Software Engineering,
The University of Western Australia, Australia

Overview

- Goals & challenges of dynamic reconfiguration
- Formal modelling of dynamic reconfiguration
  - A Circal primer
  - FPGA implementation of Circal
  - An FPGA interpreter for Circal
- Ongoing work
- Conclusion

Architectural reconfiguration

- **Definition:***The ability of a device or system architecture to change its structure over time*
  - Which structural aspects?
  - What time scale?
  - How controlled?
- **NB:** some structural changes may result in behavioural changes

Dynamic reconfiguration

- Aims to exploit architectural reconfiguration at run time in order to:
  - Adapt to changing algorithmic needs as a computation progresses
  - Improve application/system performance
  - Reuse computational resources

Dynamic reconfiguration

- Facilitates and supports
  - Adaptive processes
  - Dynamic environments
  - Hardware independence
  - Multitasking

Reconfigurable computing challenges

- How to design efficient, cost-effective architectural mixes at device and system level
- How to exploit operating niche
- How to support systems and application design
- Killer apps: finding appeal and acceptance

The design challenge

- Designing with short lead-times for short-lived, highly-customized applications
- Skill base needs to span many layers and dimensions of abstraction: from logic circuit to application layer
  - E.g. conceiving high-performance hard-wired algorithms
- Lack of integrated tools that exploit hardware capabilities

Formal modelling of dynamic reconfiguration

Goal of project: Basic language research

- Discover semantic operators needed to model static and dynamic FPGA circuits
- Learn how to compile down to those operators – how much can be automated?
- Determine what aspects need to be expressed explicitly in the design language in order to guide the compiler

Not another language…

The goal is NOT to design yet another language for reconfigurable computing.

Rather, the goal is to identify the key requirements of such a language and its compiler.

Our approach

- Investigate the problem from a formal modelling perspective — we use a process algebra called “Circal” to model circuits, their structure and behaviour

How can Circal help?

- Circal provides a means of describing concurrent systems in an uncluttered, abstract fashion
  - Facilitates discovery of fundamental operations, semantics, syntax
- Circal offers the possibility of producing circuits and translation schemes that are verifiably correct (equivalent to their specification)
- Hope to make use of formal methods literature…

Circal background

Circal background

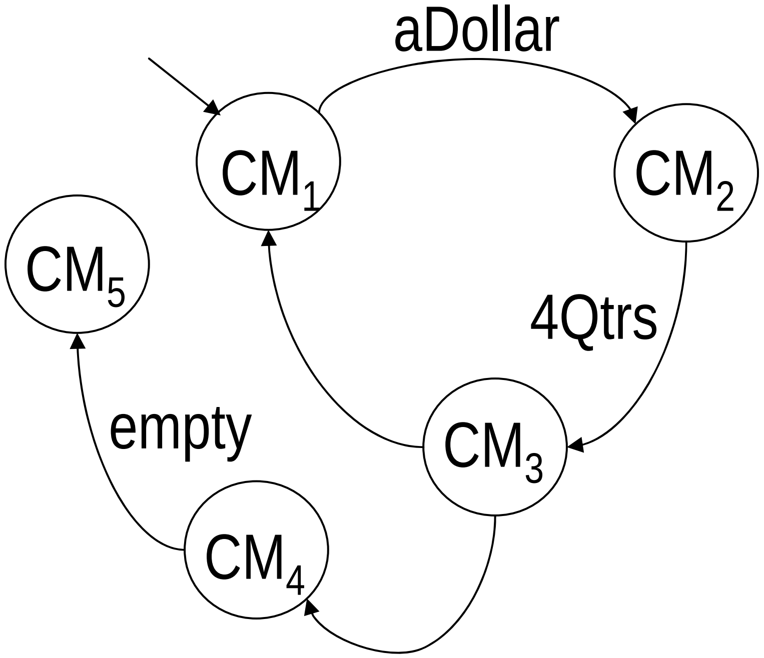
- Process algebras such as CCS, CSP, and Circal (CIRcuit CALculus) appeared mid- to late-1970s
  - Mathematical formalisms for describing & analyzing the behaviour of concurrent systems
    - Allow behavioural specification, property checking, equivalence checking, formal verification

What is Circal?

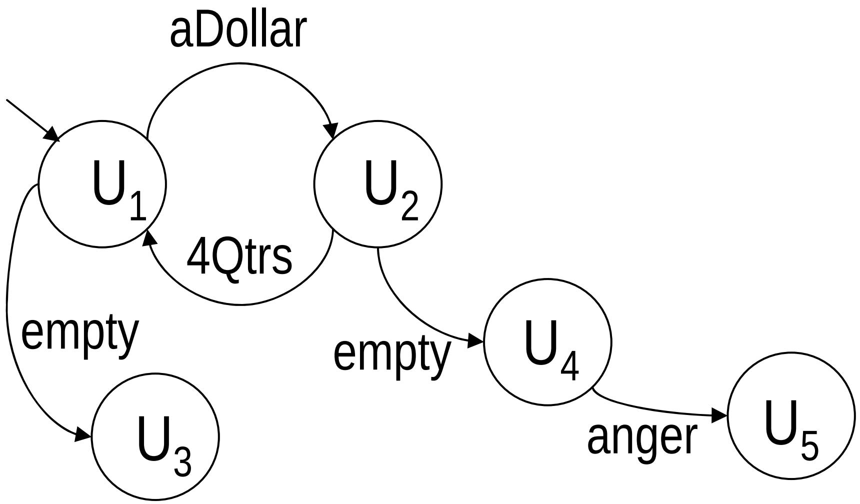
- Allows us to reason about *processes* that have *state*, and that perform or respond to *actions*

What is Circal?

- Allows us to reason about *processes* that have *state*, and that perform or respond to *actions*
- For example, we might model a change machine using a state diagram

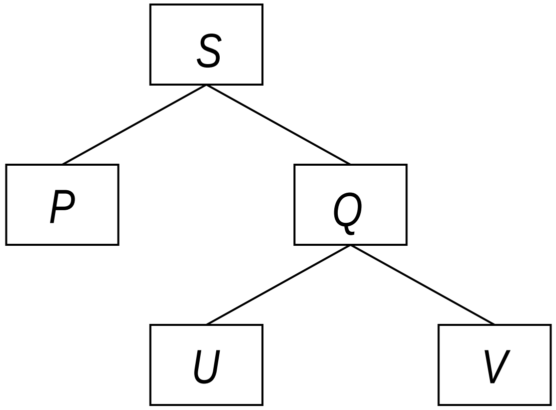


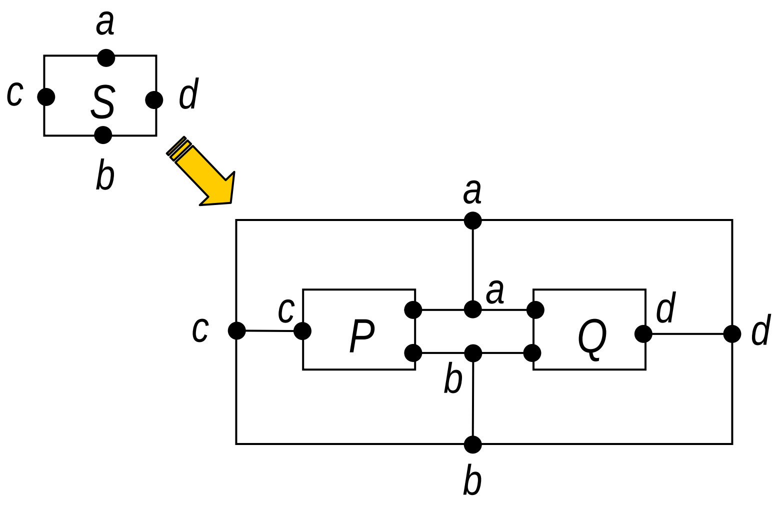
Modelling the user



- We may be interested to know:
  - What is the composed behaviour of the cash machine and the user?
  - Will the user ever get angry?
- PAs define the rules that allow these and other questions to be answered

Circal basics





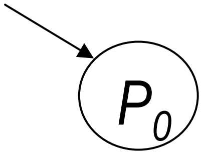
- The Circal process algebra supports hierarchical, modular, and constructive description of interacting processes
- *Processes* are behavioural objects that interact based on the occurrence of events

Behavioural modelling in Circal

- Behavioural operators

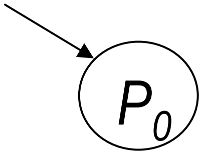
Behavioural modelling in Circal

- Behavioural operators
  - Process definition



$$P \leftarrow P_0$$

Behavioural modelling in Circal

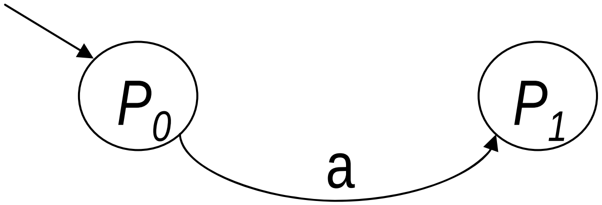


- Behavioural operators
  - Process definition
  - Process termination

$$P \leftarrow P_0$$

$$P_0 \leftarrow \Delta$$

Behavioural modelling in Circal

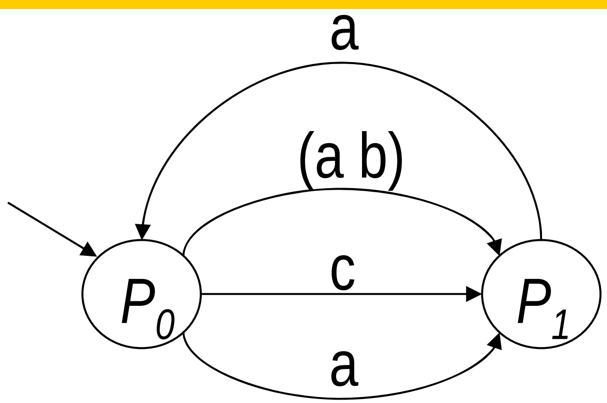


- Behavioural operators
  - Process definition
  - Process termination
  - Process evolution

$$P \leftarrow P_0$$

$$P_0 \leftarrow a\,P_1$$

Behavioural modelling in Circal



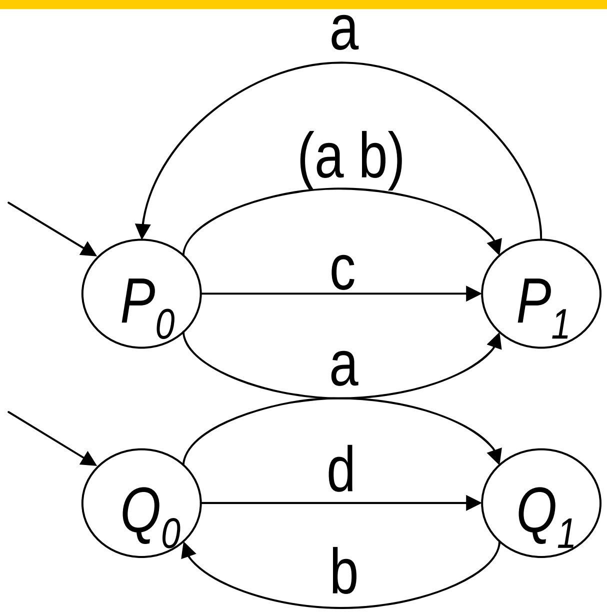
- Behavioural operators
  - Process definition
  - Process termination
  - Process evolution
  - Deterministic choice

$$P \leftarrow P_0$$

$$P_0 \leftarrow a\ P_1 + c\ P_1 + (a\ b)\ P_1$$

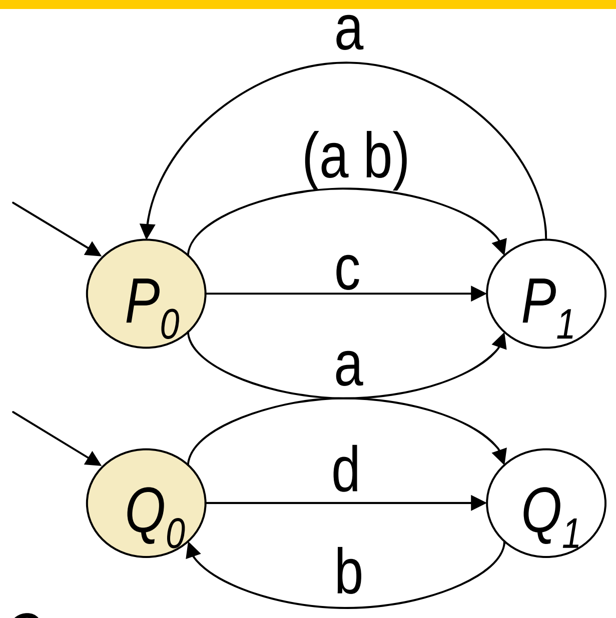
$$P_1 \leftarrow a\ P_0$$

Structural modelling in Circal



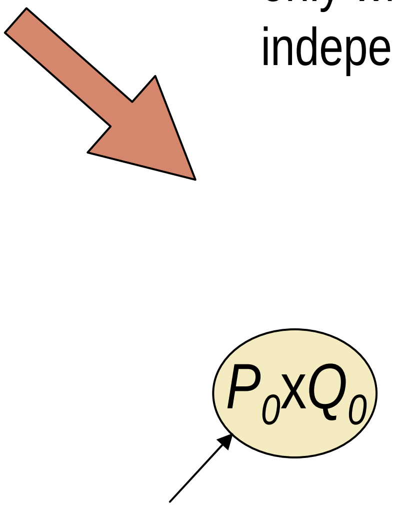
- Structural operators
  - Composition

Structural modelling in Circal

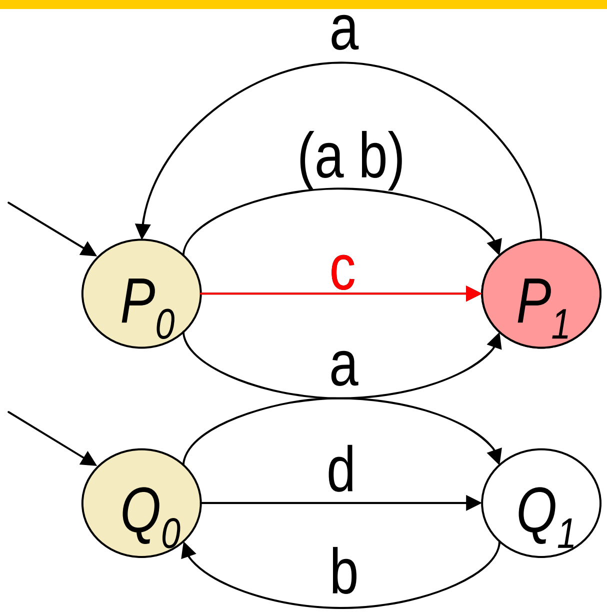


$S \leftarrow P_0 * Q_0$

- Structural operators
  - Composition
    - Evolve on shared events only when each is independently able to



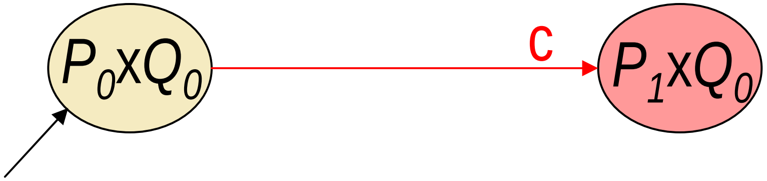
Structural modelling in Circal



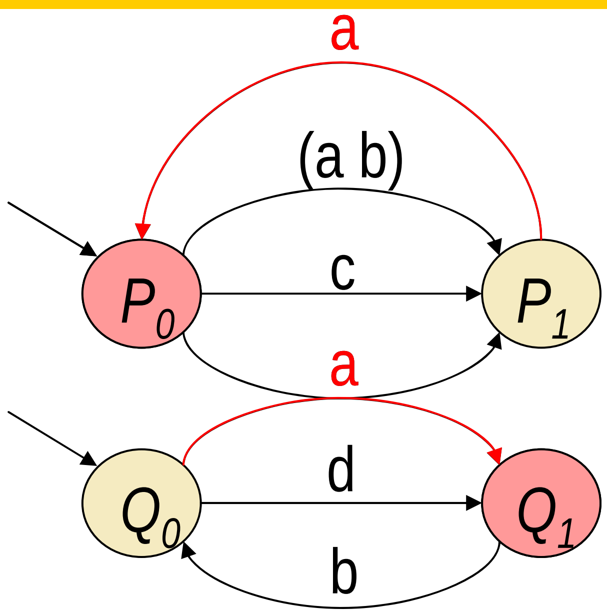
- Structural operators
  - Composition
    - Evolve on shared events only when each is independently able to

$S \leftarrow P_0 * Q_0$

$P_0 * Q_0 \leftarrow c\, P_1 * Q_0$



Structural modelling in Circal

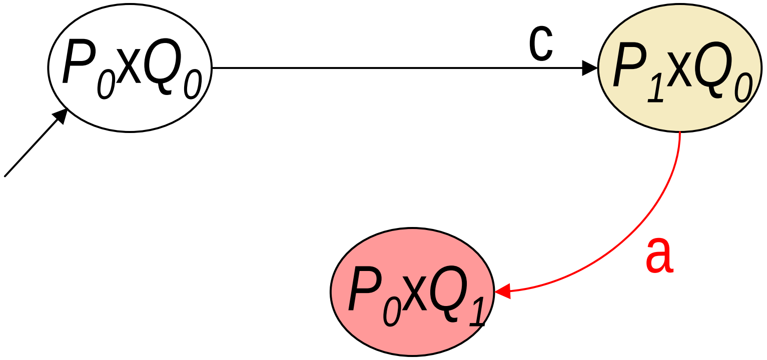


- Structural operators
  - Composition
    - Evolve on shared events only when each is independently able to

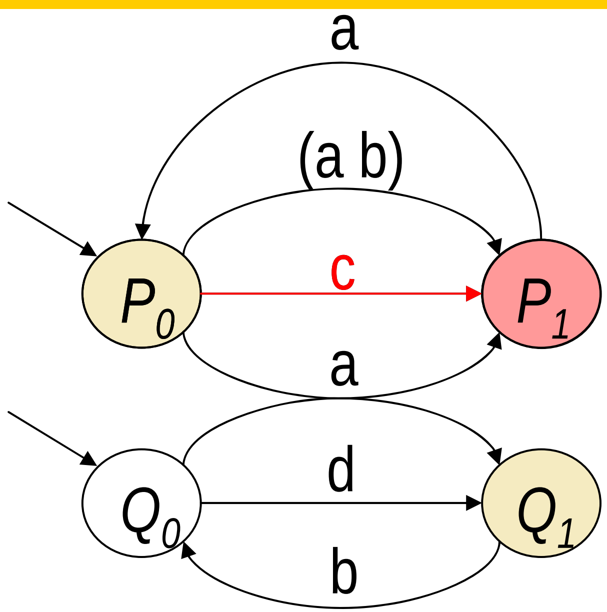
$$S \leftarrow P_0 * Q_0$$

$$P_0 * Q_0 \leftarrow c\; P_1 * Q_0$$

$$P_1 * Q_0 \leftarrow a\; P_0 * Q_1$$



Structural modelling in Circal



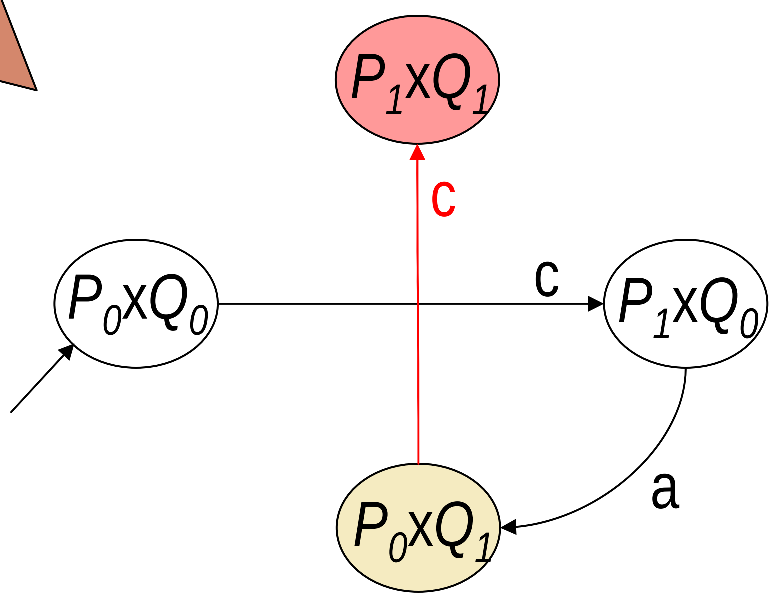
- Structural operators
  - Composition
    - Evolve on shared events only when each is independently able to

$$S \leftarrow P_0 * Q_0$$

$$P_0 * Q_0 \leftarrow c\ P_1 * Q_0$$

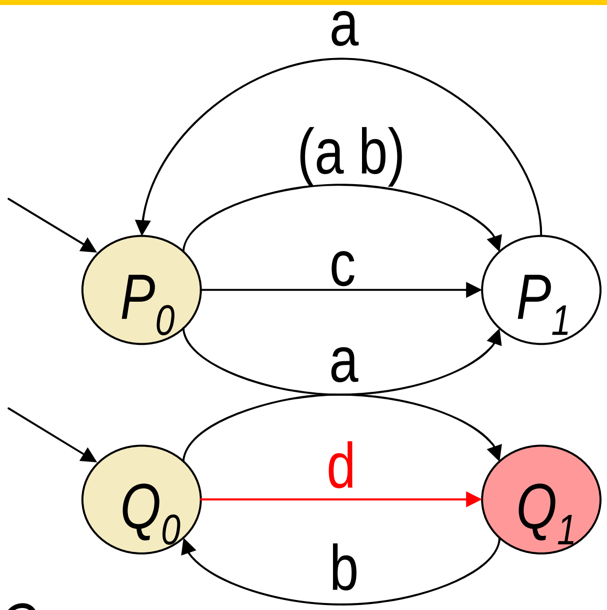
$$P_1 * Q_0 \leftarrow a\ P_0 * Q_1$$

$$P_0 * Q_1 \leftarrow c\ P_1 * Q_1$$



Structural modelling in Circal

- Structural operators
  - Composition
    - Evolve on shared events only when each is independently able to



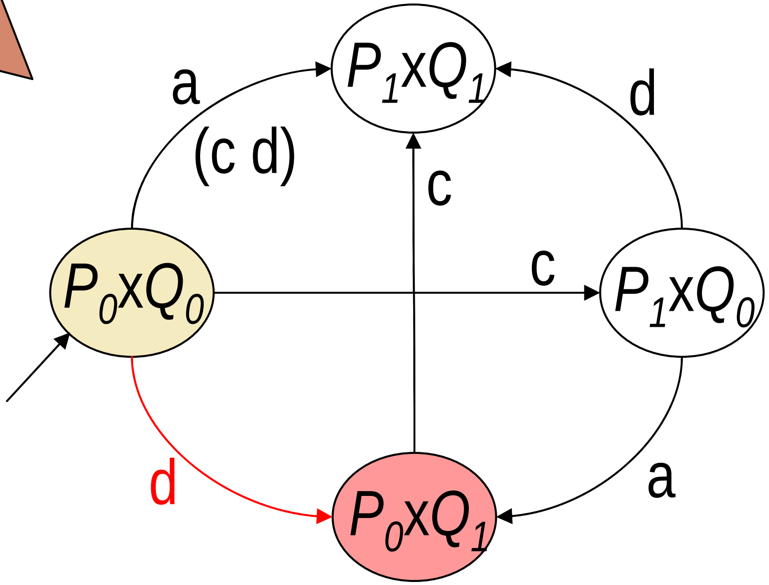
$$S \leftarrow P_0 * Q_0$$

$$P_0 * Q_0 \leftarrow \text{c } P_1 * Q_0 + \text{d } P_0 * Q_1 + \text{a } P_1 * Q_1 + \text{(c d) } P_1 * Q_1$$

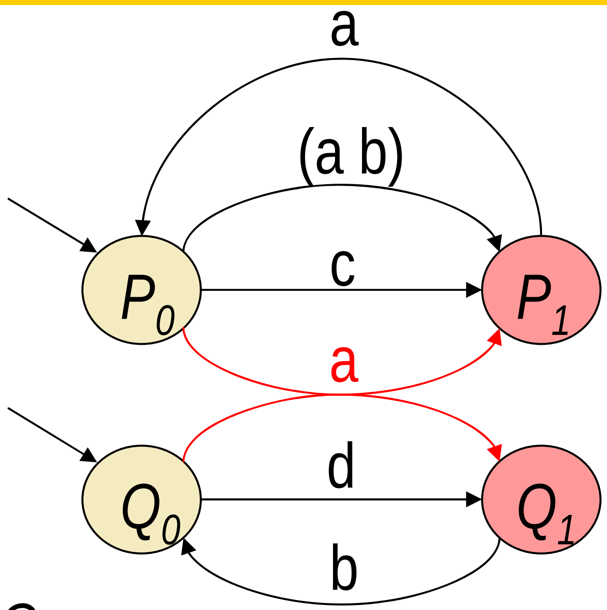
$$P_1 * Q_0 \leftarrow \text{a } P_0 * Q_1 + \text{d } P_1 * Q_1$$

$$P_0 * Q_1 \leftarrow \text{c } P_1 * Q_1$$

$$P_1 * Q_1 \leftarrow \Delta$$



Structural modelling in Circal



$$S \leftarrow P_0 \ast Q_0$$

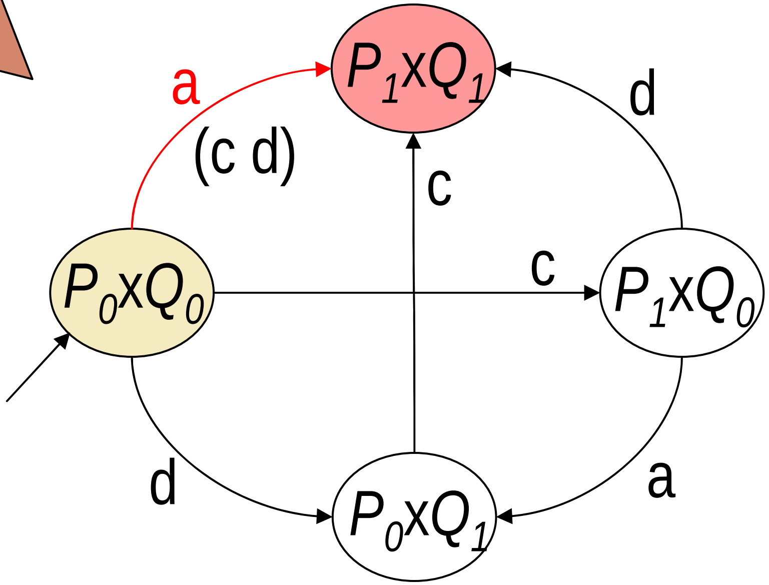
$$P_0 \ast Q_0 \leftarrow c\, P_1 \ast Q_0 + d\, P_0 \ast Q_1 + a\, P_1 \ast Q_1 + (c\ d)\, P_1 \ast Q_1$$

$$P_1 \ast Q_0 \leftarrow a\, P_0 \ast Q_1 + d\, P_1 \ast Q_1$$

$$P_0 \ast Q_1 \leftarrow c\, P_1 \ast Q_1$$

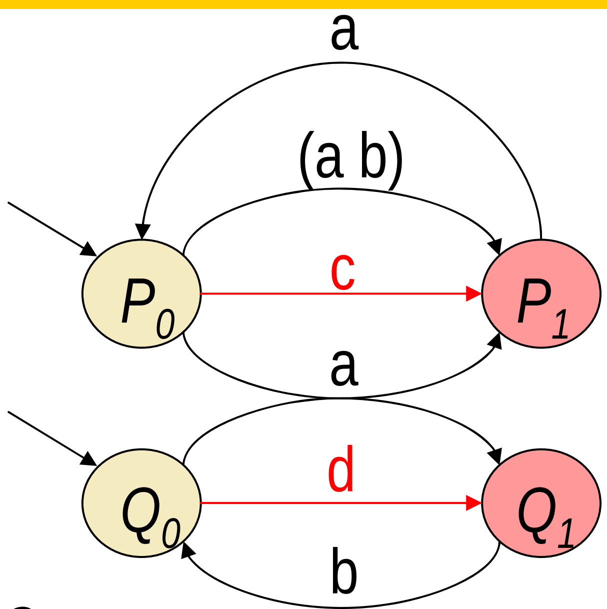
$$P_1 \ast Q_1 \leftarrow \Delta$$

- Structural operators
  - – Composition
    - Evolve on shared events only when each is independently able to



Structural modelling in Circal

- Structural operators
  - Composition
    - Evolve on shared events only when each is independently able to



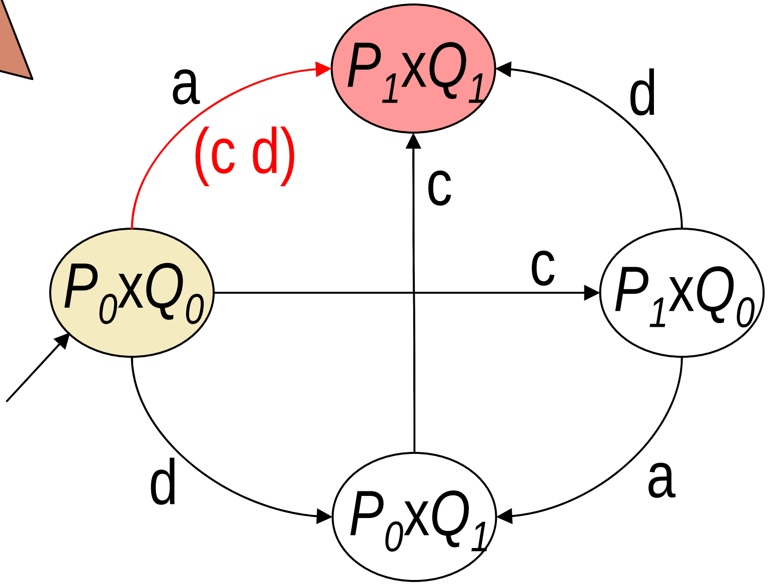
$S \leftarrow P_0 \ast Q_0$

$P_0 \ast Q_0 \leftarrow c\ P_1 \ast Q_0 + d\ P_0 \ast Q_1 + a\ P_1 \ast Q_1$
$+ (c\ d)\ P_1 \ast Q_1$

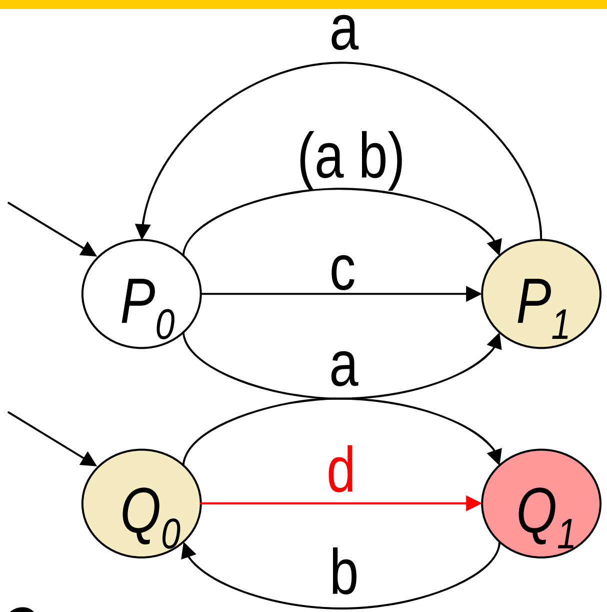
$P_1 \ast Q_0 \leftarrow a\ P_0 \ast Q_1 + d\ P_1 \ast Q_1$

$P_0 \ast Q_1 \leftarrow c\ P_1 \ast Q_1$

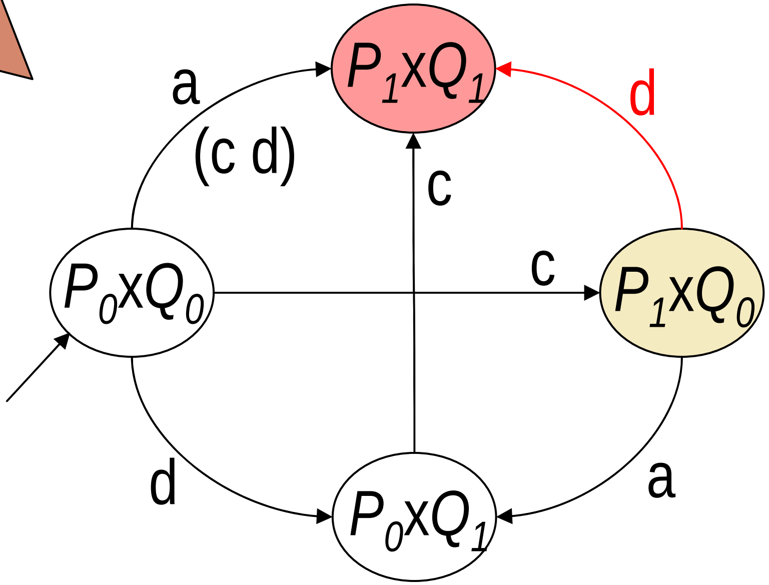
$P_1 \ast Q_1 \leftarrow \Delta$



Structural modelling in Circal



- Structural operators
  - Composition
    - Evolve on shared events only when each is independently able to



$$S \leftarrow P_0 * Q_0$$

$$P_0 * Q_0 \leftarrow c\ P_1 * Q_0 + d\ P_0 * Q_1 + a\ P_1 * Q_1 + (c\ d)\ P_1 * Q_1$$

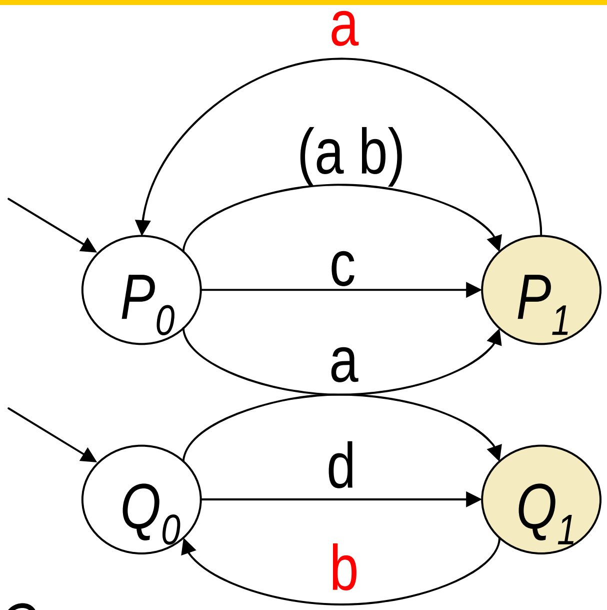
$$P_1 * Q_0 \leftarrow a\ P_0 * Q_1 + d\ P_1 * Q_1$$

$$P_0 * Q_1 \leftarrow c\ P_1 * Q_1$$

$$P_1 * Q_1 \leftarrow \Delta$$

Structural modelling in Circal

- Structural operators
  - Composition
    - Evolve on shared events only when each is independently able to



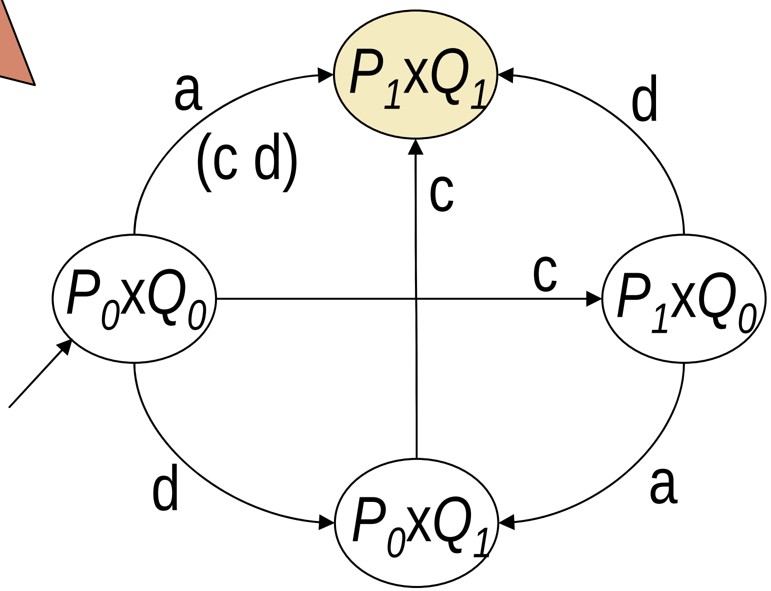
$S \leftarrow P_0 * Q_0$

$P_0 * Q_0 \leftarrow c\, P_1 * Q_0 + d\, P_0 * Q_1 + a\, P_1 * Q_1$
$\quad + (c\ d)\, P_1 * Q_1$

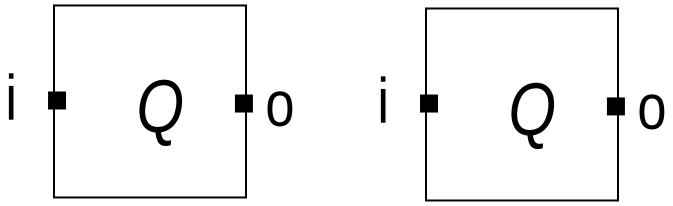
$P_1 * Q_0 \leftarrow a\, P_0 * Q_1 + d\, P_1 * Q_1$

$P_0 * Q_1 \leftarrow c\, P_1 * Q_1$

$P_1 * Q_1 \leftarrow \Delta$ (DEADLOCK)

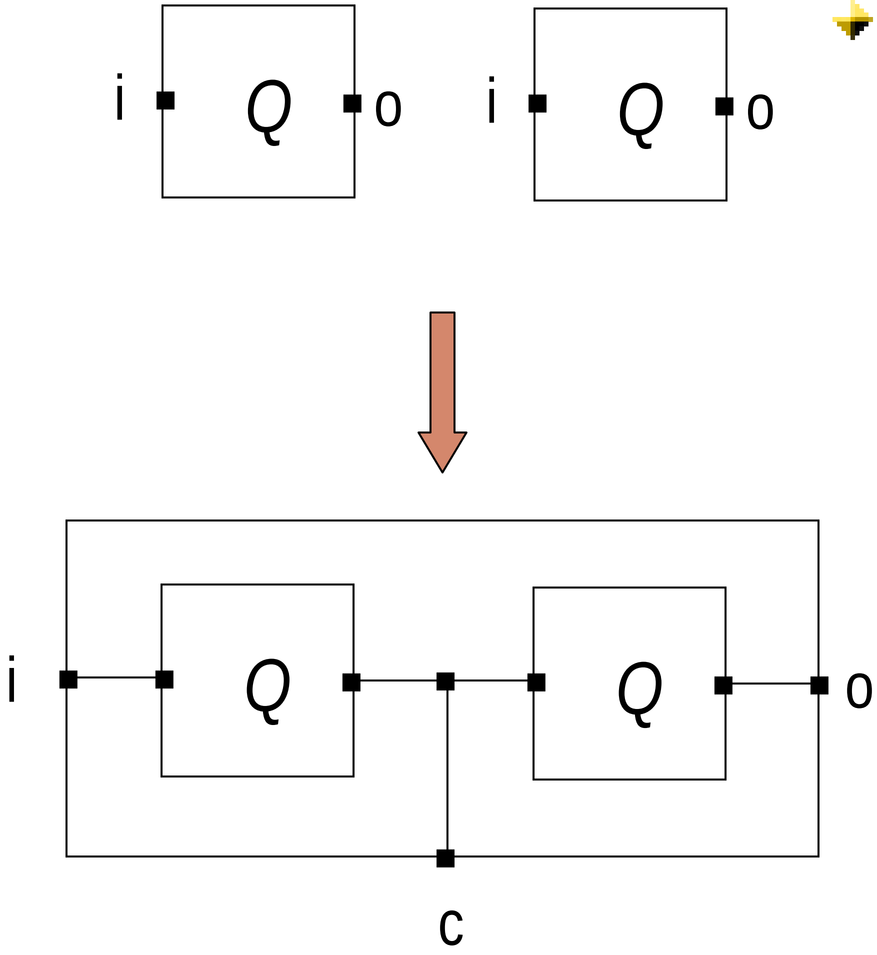


Structural modelling in Circal



- Structural operators
  - Composition
  - Relabelling
    - Similar to parameterization
    - Supports reuse

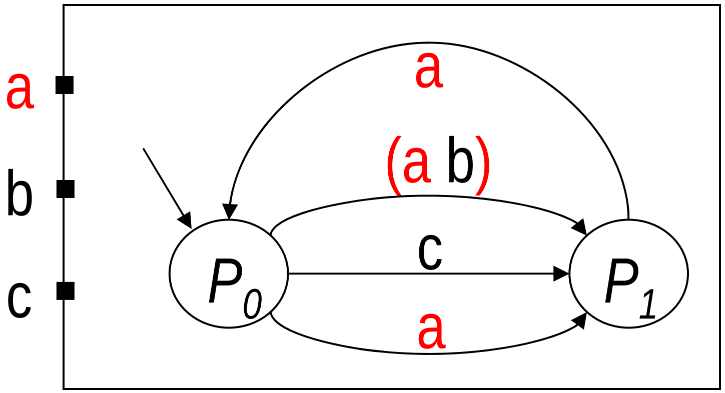
Structural modelling in Circal



- Structural operators
  - Composition
  - Relabelling
    - Similar to parameterization
    - Supports reuse

$S \leftarrow Q\ [c/o] * Q\ [c/i]$

Structural modelling in Circal



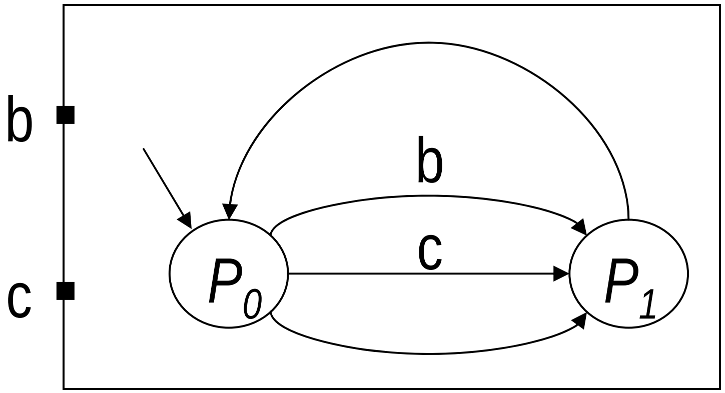
- Structural operators
  - – Composition
  - – Relabelling
  - – Abstraction
    - Hides events from observer

$$P \leftarrow P_0$$

$$P_0 \leftarrow a\, P_1 + c\, P_1 + (a\ b)\, P_1$$

$$P_1 \leftarrow a\, P_0$$

Structural modelling in Circal



$$P - a \leftarrow P_0$$

$$P_0 \leftarrow P_1 \;\&\; c\,P_1 \;\&\; b\,P_1$$

$$P_1 \leftarrow P_0$$

- Structural operators
  - Composition
  - Relabelling
  - Abstraction
    - Hides events from observer
    - Introduces non-deterministic behaviour
    - Limited use in HW description

What Circal has been used for

- Modelling & verifying digital (CMOS) circuits and asynchronous (micropipeline) systems
- Verifying the timing, performance, and correctness of concurrent systems
- Describing complex systems such as traffic networks using a CA framework

Circal as a specification language

- Success with using Circal for digital design & verification led us to wonder:

Is Circal suitable as the basis of
a specification language for FPGAs?

Question has led to work on

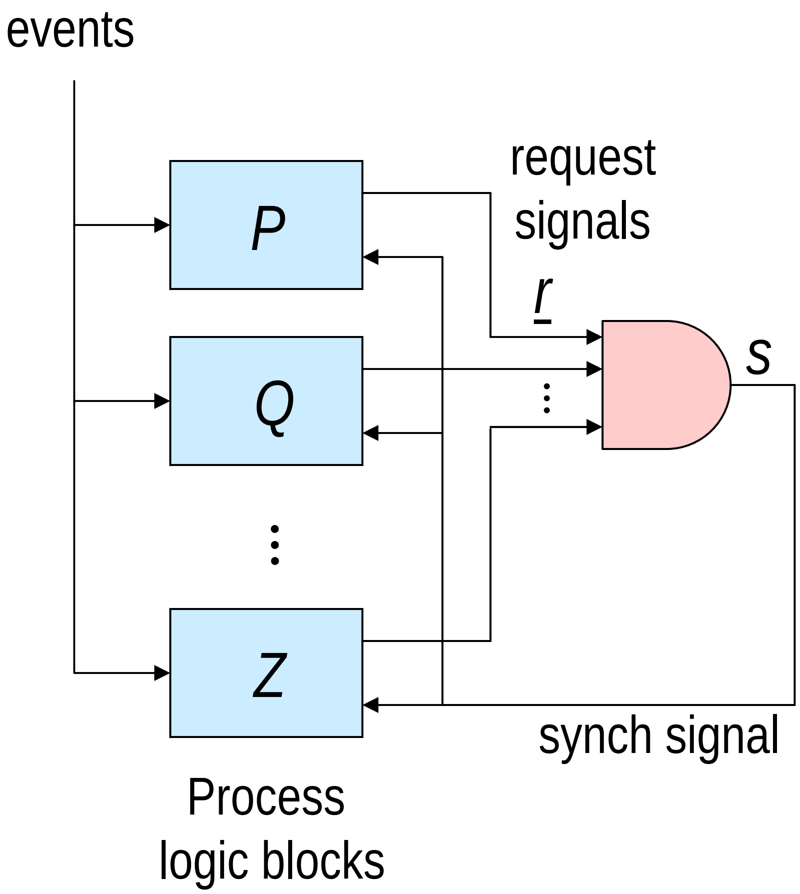
- Mapping Circal specifications to RL
- Automatic support for virtualized circuit designs
- Modelling DRL using a PA formalism
- Determining what that provides us with

FPGA implementation of Circal

Mapping goals

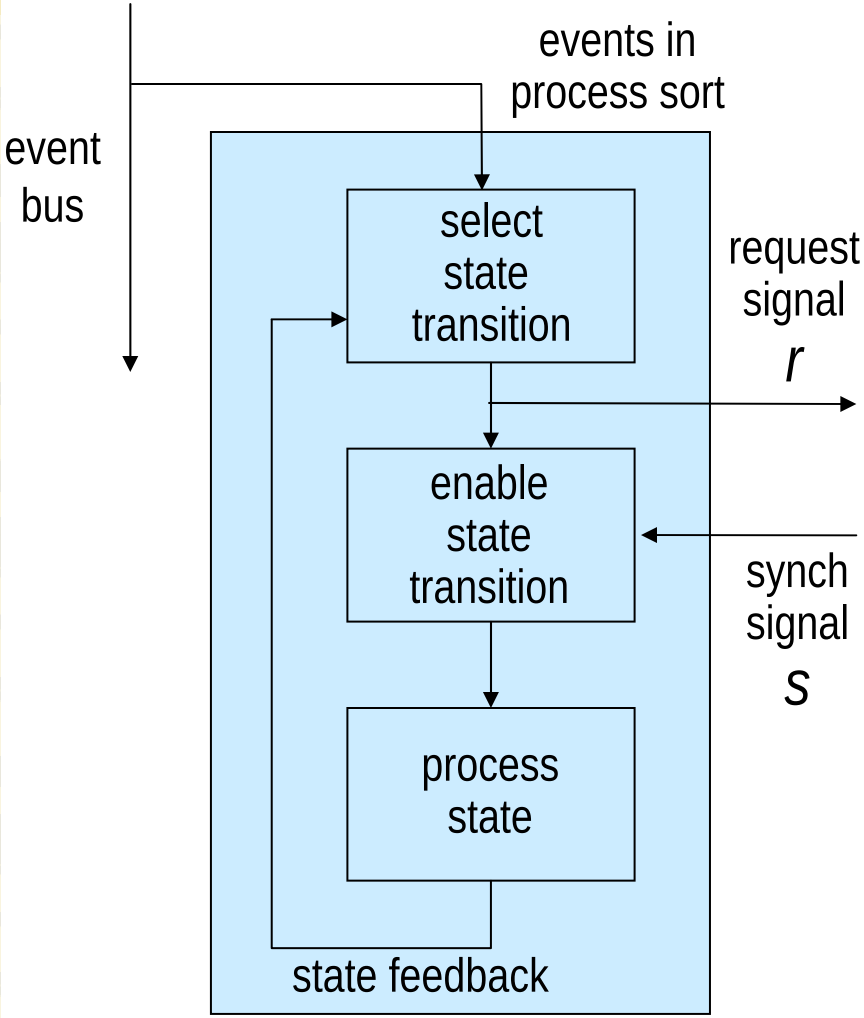
- Quick and easy instantiation of circuits
- Distribute computation for scalability
- Speedup through concurrent execution
- Use dynamic reconfiguration to overcome resource limitations
- Provide scope for implementing dynamic specifications

Circuit realization of Circal



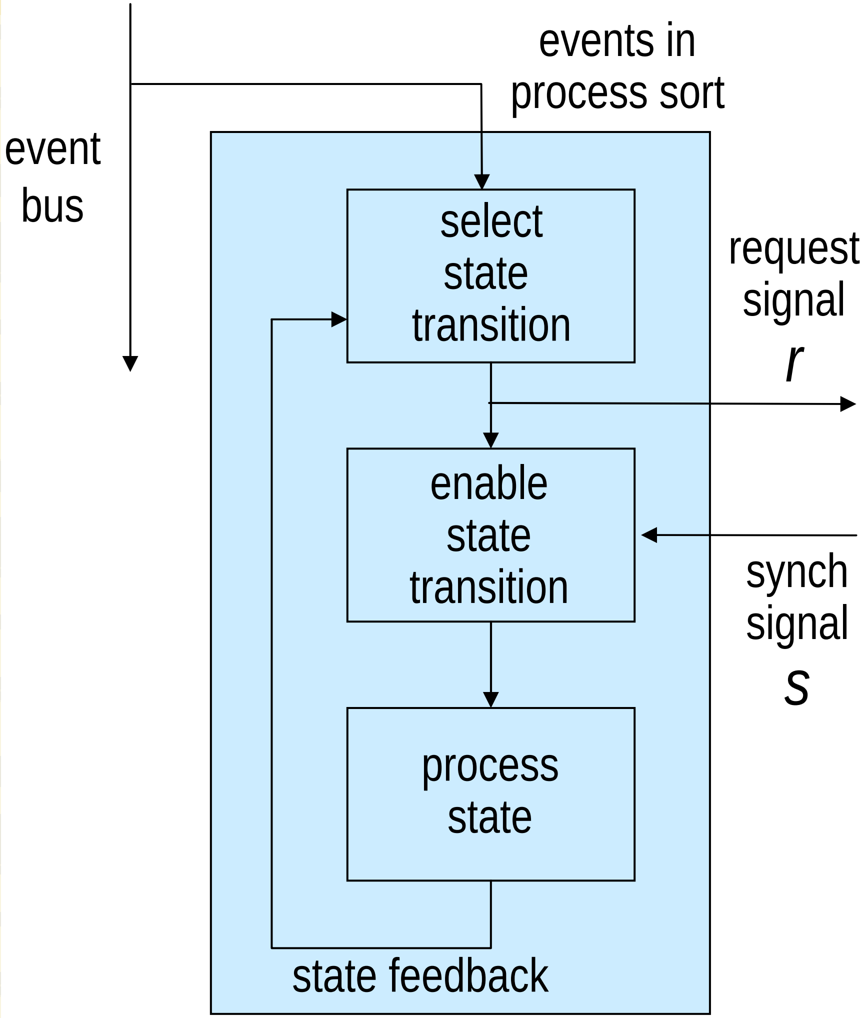
- At the system level, a Circal specification is realized as an interconnection of independent, concurrently active *process logic blocks* and *synchronisation logic*

Circuit realization of Circal (cont)



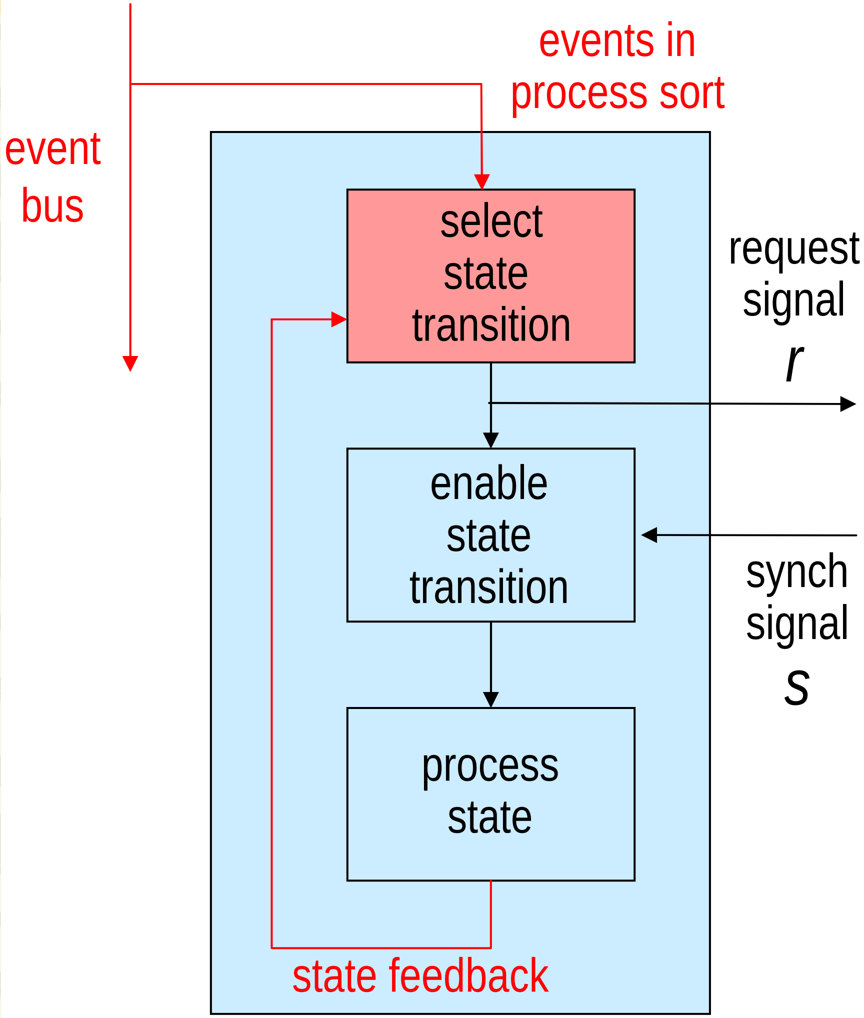
- Each process logic block implements the behaviour specified by its process definitions
- Of course the Circal composition law constrains process state transitions to those that are globally acceptable

Circuit activation



- Processes respond to input events each cycle
- State renewal consists of three phases:

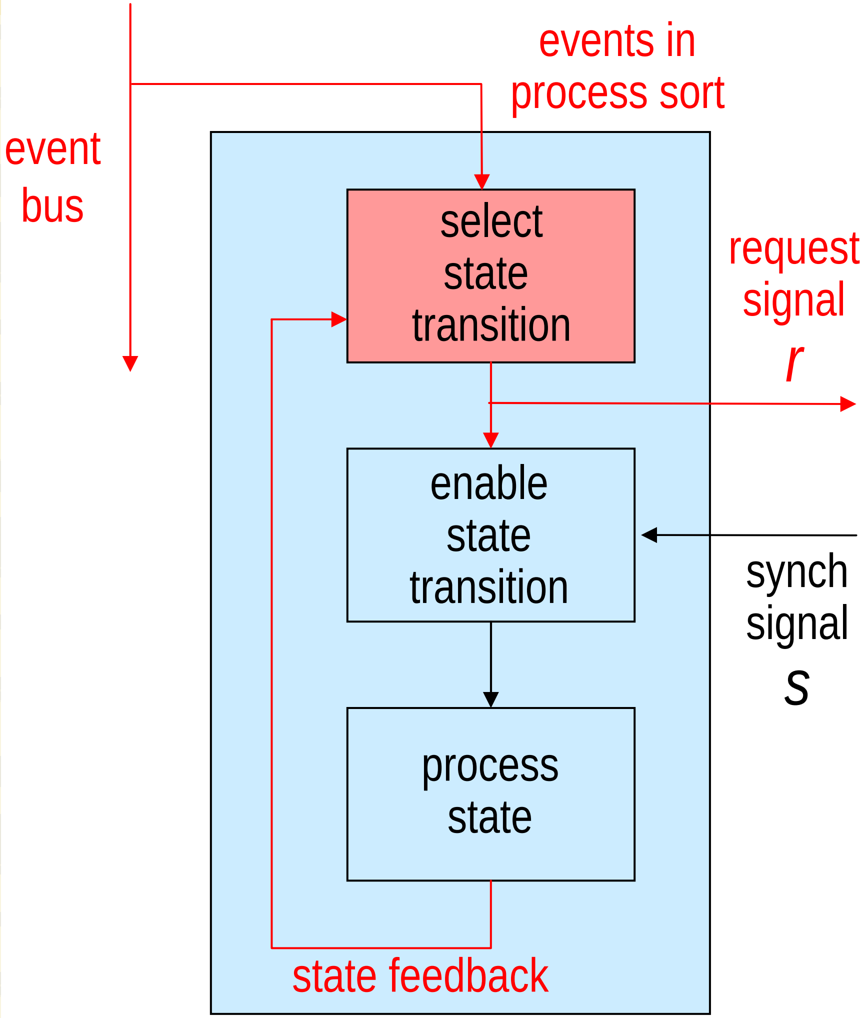
Circuit activation (cont)



- Processes respond to input events each cycle
- State renewal consists of three phases:

1. Each process checks whether the event is acceptable to itself

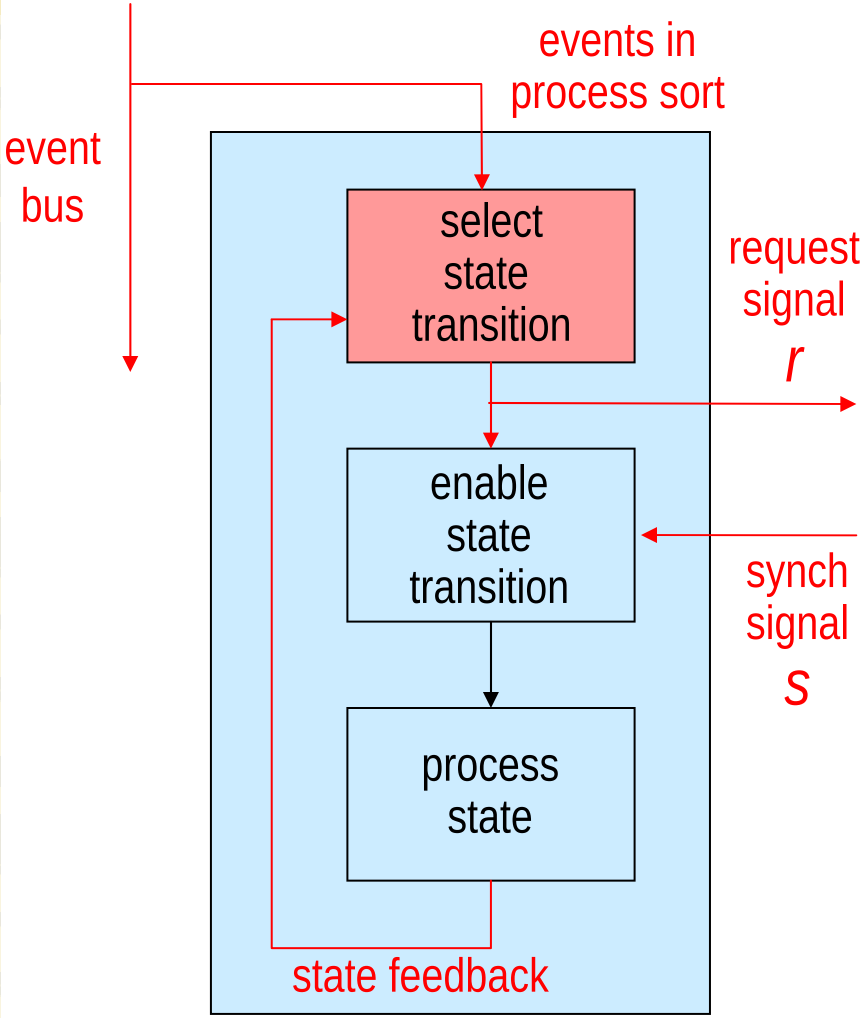
Circuit activation (cont)



- Processes respond to input events each cycle
- State renewal consists of three phases:

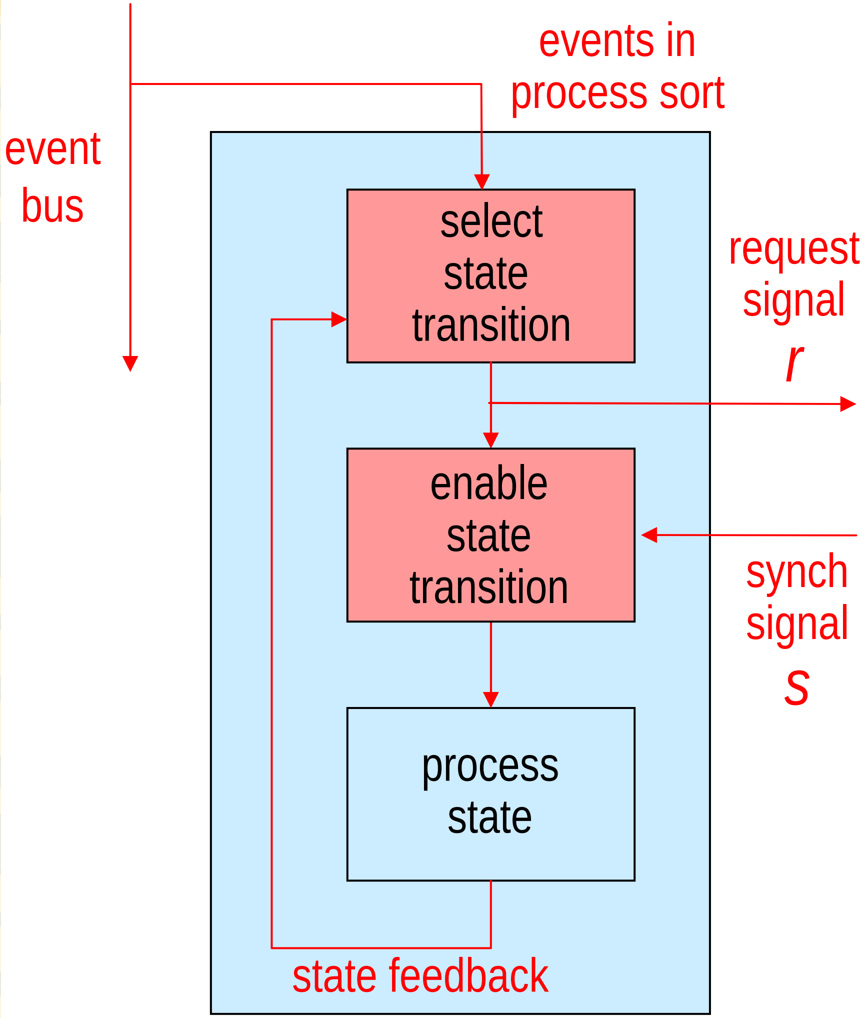
1. Each process checks whether the event is acceptable to itself and raises a synchronization request signal if it is;

Circuit activation (cont)



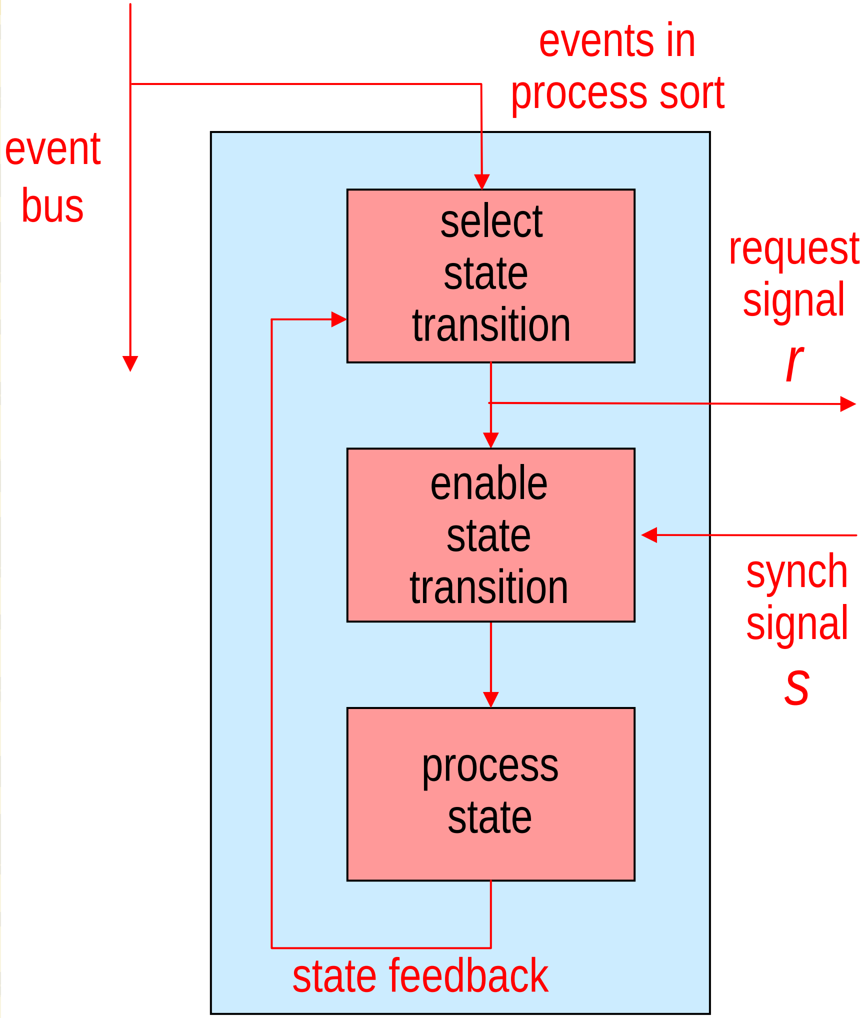
- 3 phase state renewal:
  1. Check event acceptability;
  2. Synchronization logic asserts a synchronization signal if all processes find the event acceptable; and

Circuit activation (cont)



- 3 phase state renewal:
  1. Check event acceptability;
  2. Synchronization signal asserted; and
  3. Each process enables the state transition guarded by the input event if synchronization is asserted.

Circuit activation (cont)

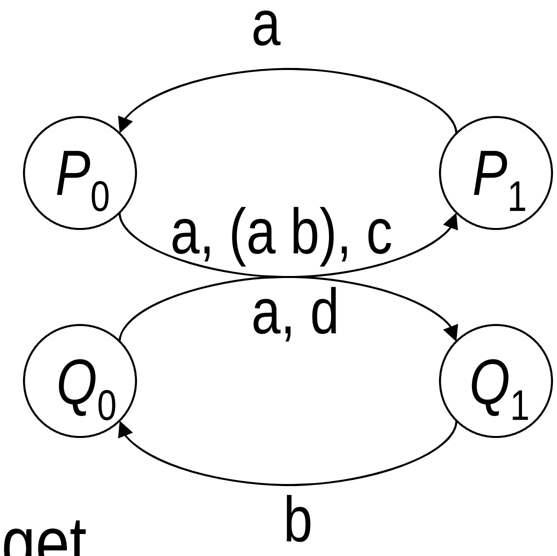


- 3 phase state renewal:
  1. Check event acceptability;
  2. Synchronization signal asserted; and
  3. Each process enables the state transition guarded by the input event if synchronization is asserted.
- State is updated at the next clock edge

Design example

Consider
$$P_0 \leftarrow aP_1 + (ab)P_1 + cP_1$$
$$P_1 \leftarrow aP_0$$

and
$$Q_0 \leftarrow aQ_1 + dQ_1$$
$$Q_1 \leftarrow bQ_0$$



Applying the Circal composition law we get

$$P_0 * Q_0 \leftarrow aP_1 * Q_1 + cP_1 * Q_0 + (cd)P_1 * Q_1 + dP_0 * Q_1$$
$$P_1 * Q_0 \leftarrow aP_0 * Q_1 + dP_1 * Q_1$$
$$P_0 * Q_1 \leftarrow cP_1 * Q_1$$
$$P_1 * Q_1 \leftarrow \Delta$$

State renewal phase 1:
Determining whether an event combination is valid

Consider just the logic for process *P:* $P_0 \leftarrow aP_1 + (ab\ )P_1 + cP_1$

$P_1 \leftarrow aP_0$

In state P_0 the process
responds to events in the set . . . . $\{a\bar{b}\bar{c}, ab\bar{c}, \bar{a}\bar{b}c, \bar{a}\bar{b}\bar{c}\}$.

Hence process *P* in state P_0 accepts
the boolean expression of events . . . . $(\bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + a\bar{b}\bar{c} + ab\bar{c})$.

Similarly, in state P_1, the process accepts . . . . $(\bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c})$.

The synchronization request signal can thus be expressed as . . .

$$r_P = (\bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + a\bar{b}\bar{c} + ab\bar{c}).P_0 + (\bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c}).P_1.$$

State renewal phase 2:
Checking global acceptability of an event

Done by forming the global conjunction of process synchronization request signals: $s = \prod_i r_i$

State renewal phase 3:
Allowing state transitions

Let D_{P_0} and D_{P_1} denote the boolean input functions for the P_0 and P_1 state flip-flops

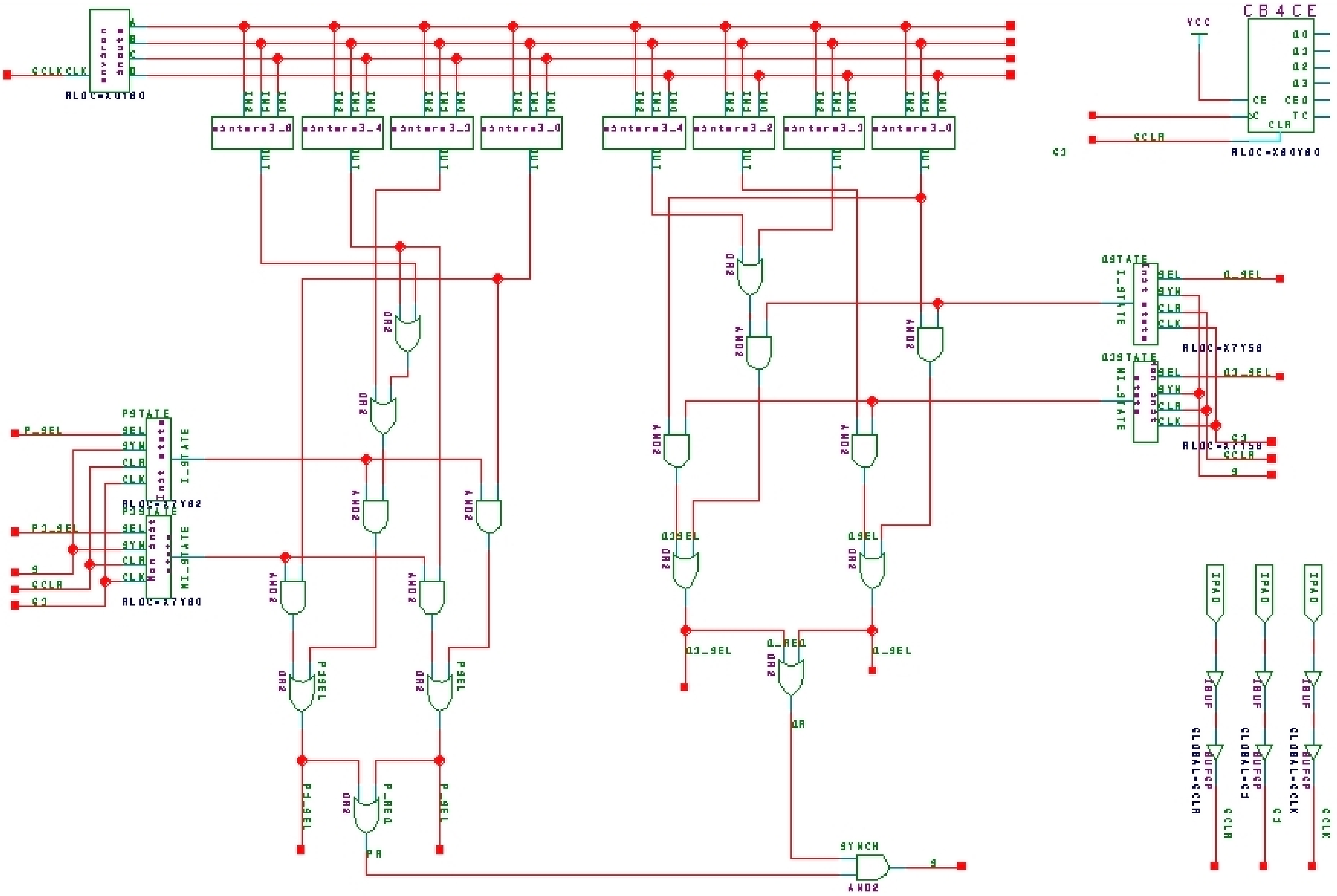
Then from

$$P_0 \leftarrow aP_1 + (ab)P_1 + cP_1$$
$$P_1 \leftarrow aP_0$$

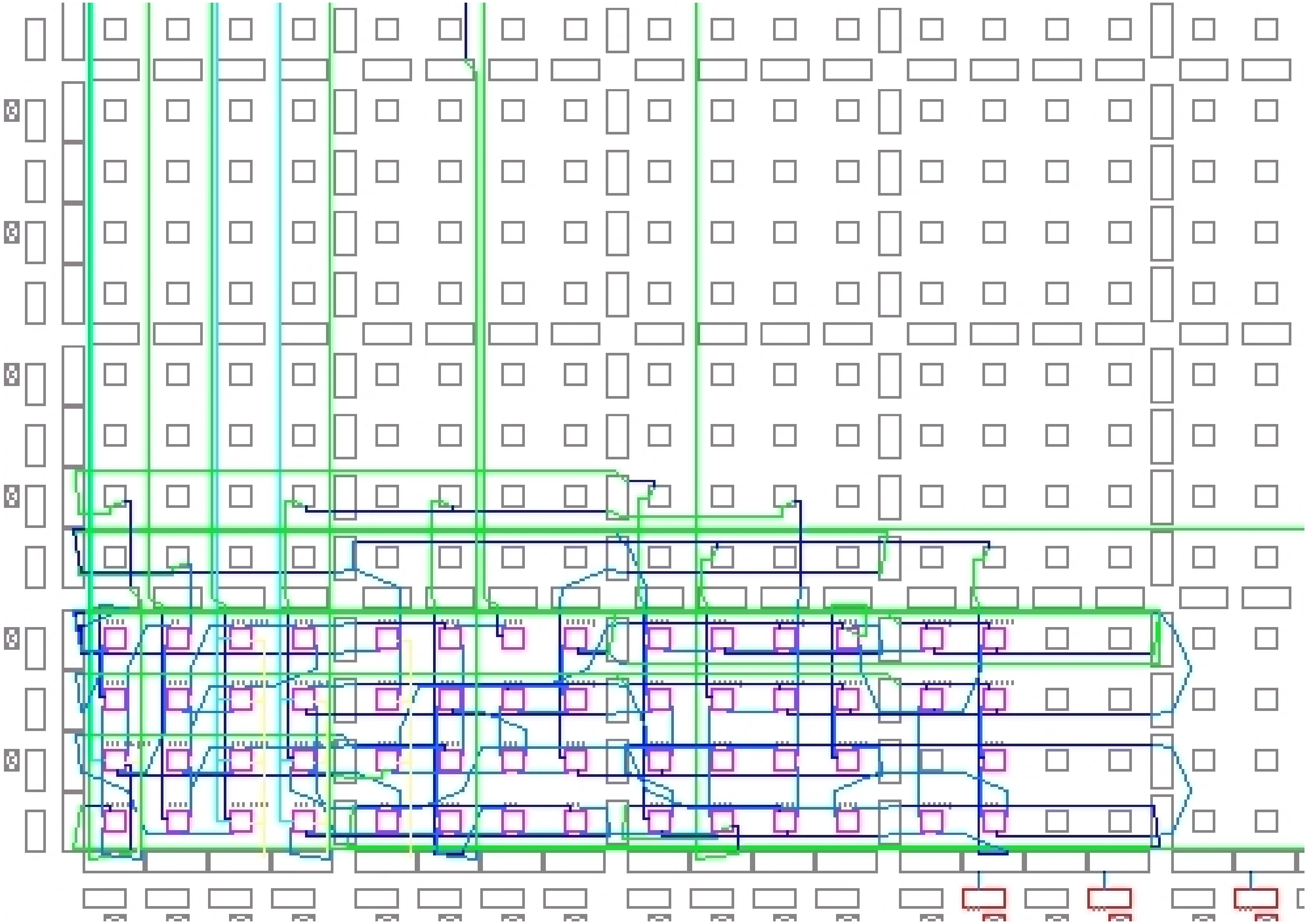
we can derive

$$D_{P_0} = s.(a\overline{b}\,\overline{c}.P_1 + \overline{a}\,\overline{b}\,\overline{c}.P_0) + \overline{s}.P_0$$
$$D_{P_1} = s.([a\overline{b}\,\overline{c} + ab\overline{c} + \overline{a}\,\overline{b}c].P_0 + \overline{a}\,\overline{b}\,\overline{c}.P_1) + \overline{s}.P_1$$

Schematic for *P*Q*



Automatic place & route of flat design



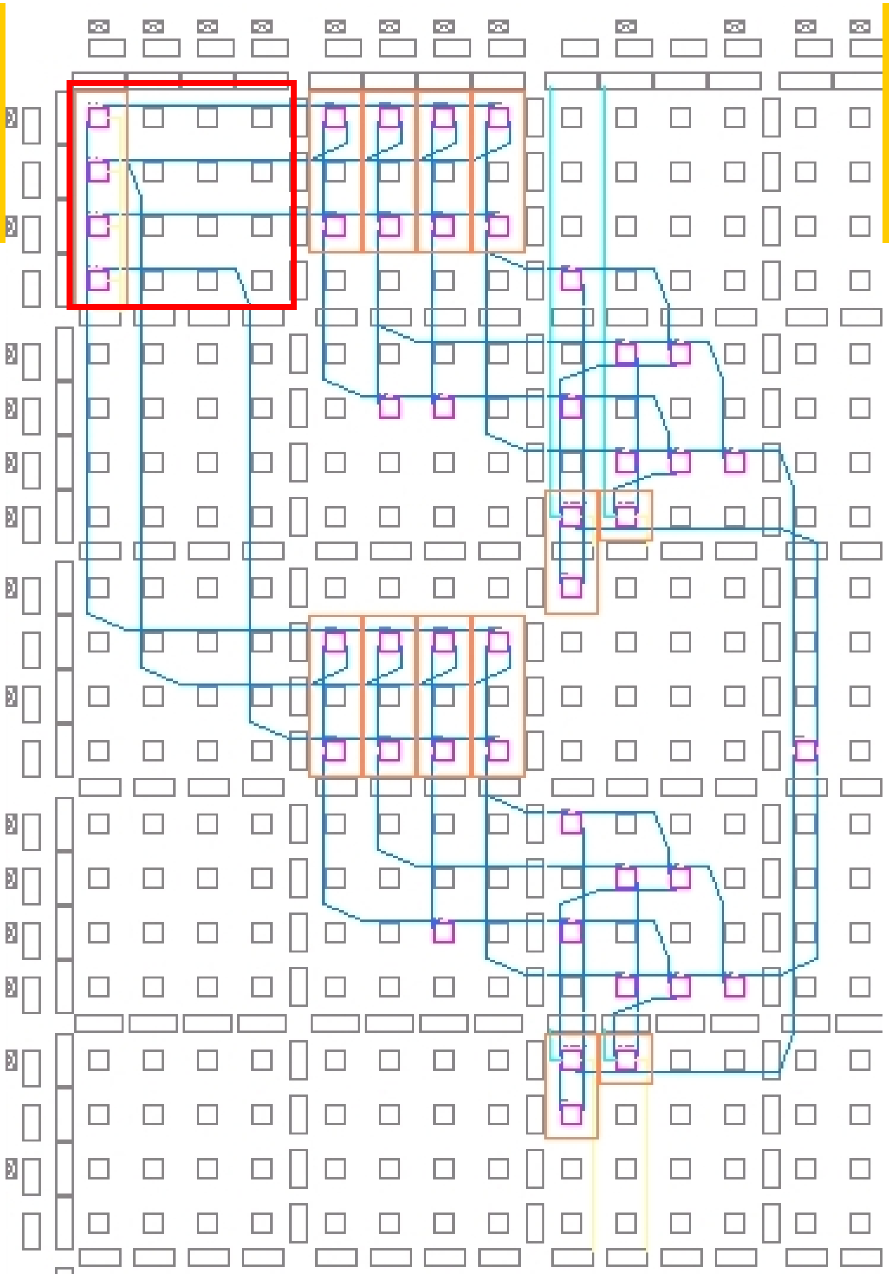
Module Generators

1. Environmental Inputs:

   EI(tlc, tlr, ni, or, ob);

   - consists of input register and wires to right and bottom

   - specified by coords of top left corner, number of inputs, and vector of wires to right & bottom

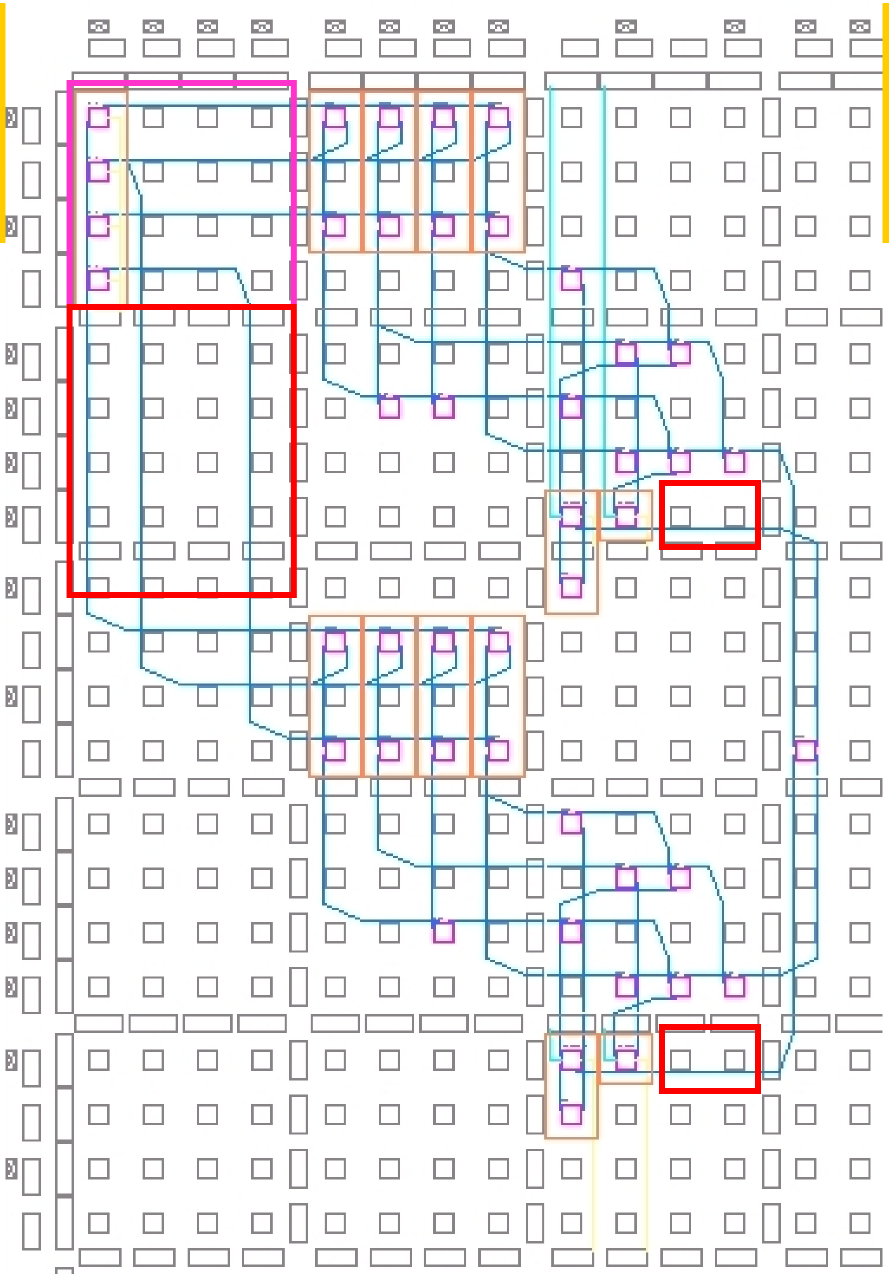


Module Generators

2. Buses:

B(tlc, tlr, wi, hi, or, aw);

- specified by coords of top left corner, width and height, orientation, and vector of active wires



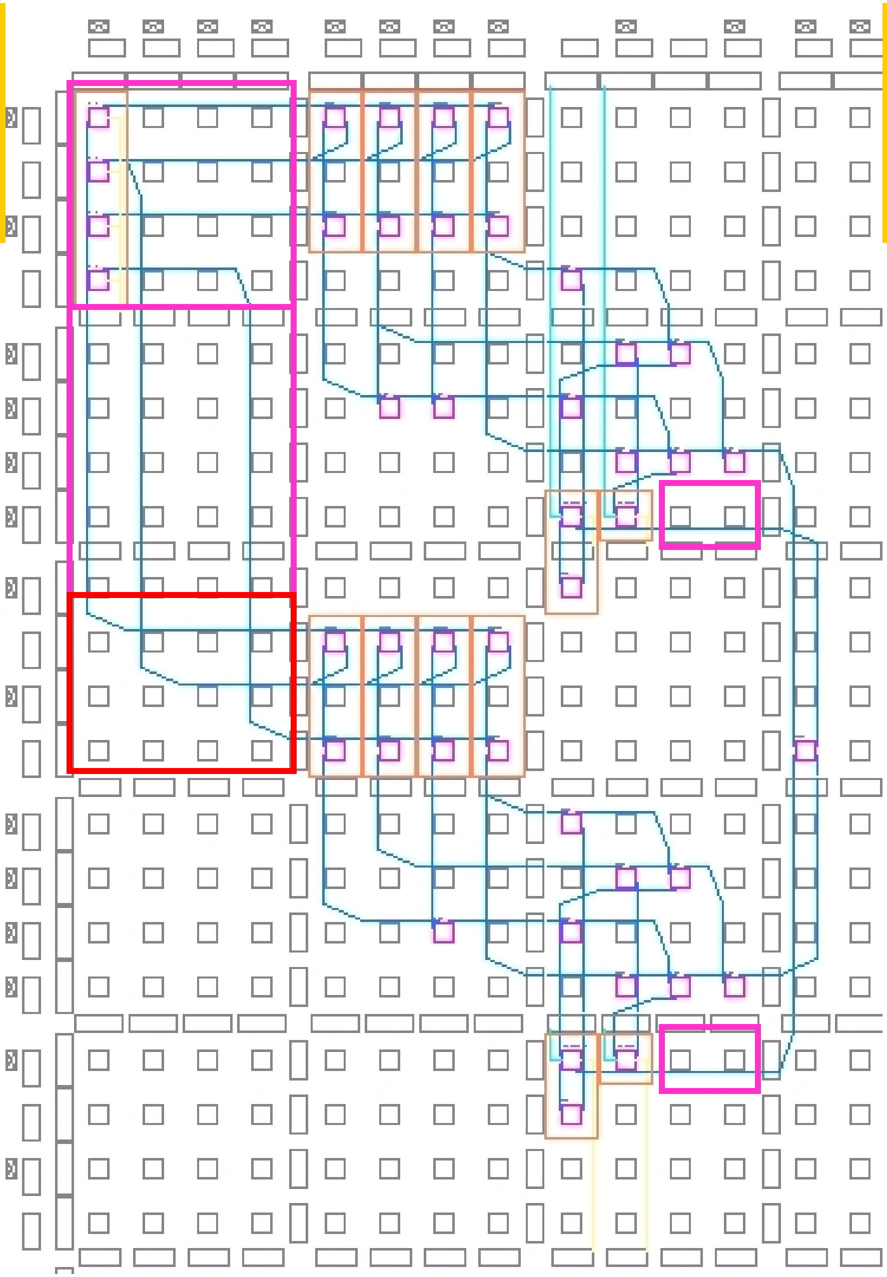
Module Generators

3. Input Junctions:

IJ(tlc, tlr, wi, or, ob);

- allows another process to be added

- specified by coords of top left corner, width, and vector of outputs to right and bottom



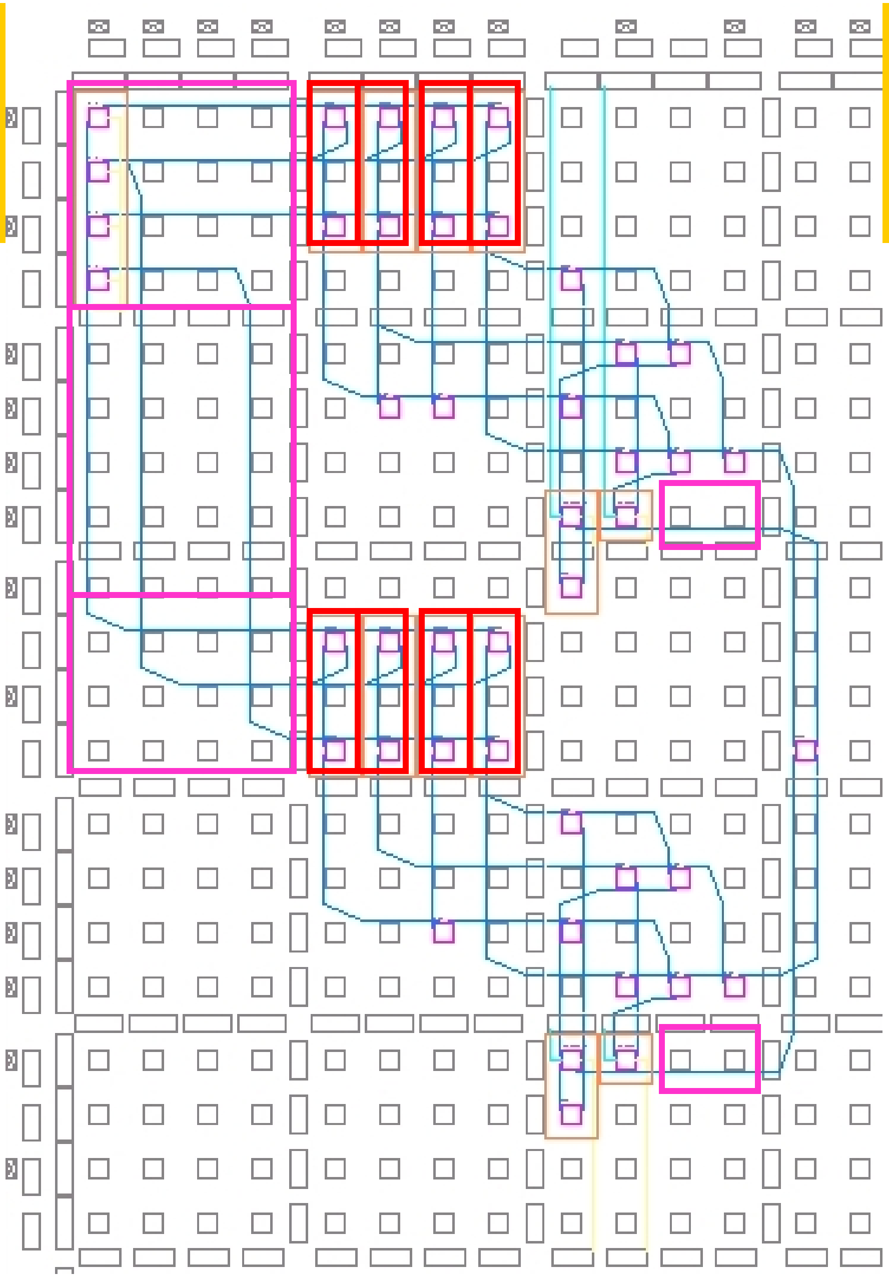
Module Generators

4. Minterms:

M(tlc, tlr, ni, mn);

- computes minterm, passes input to right and output to bottom

- specified by coords of top left corner, number of inputs, and minterm number



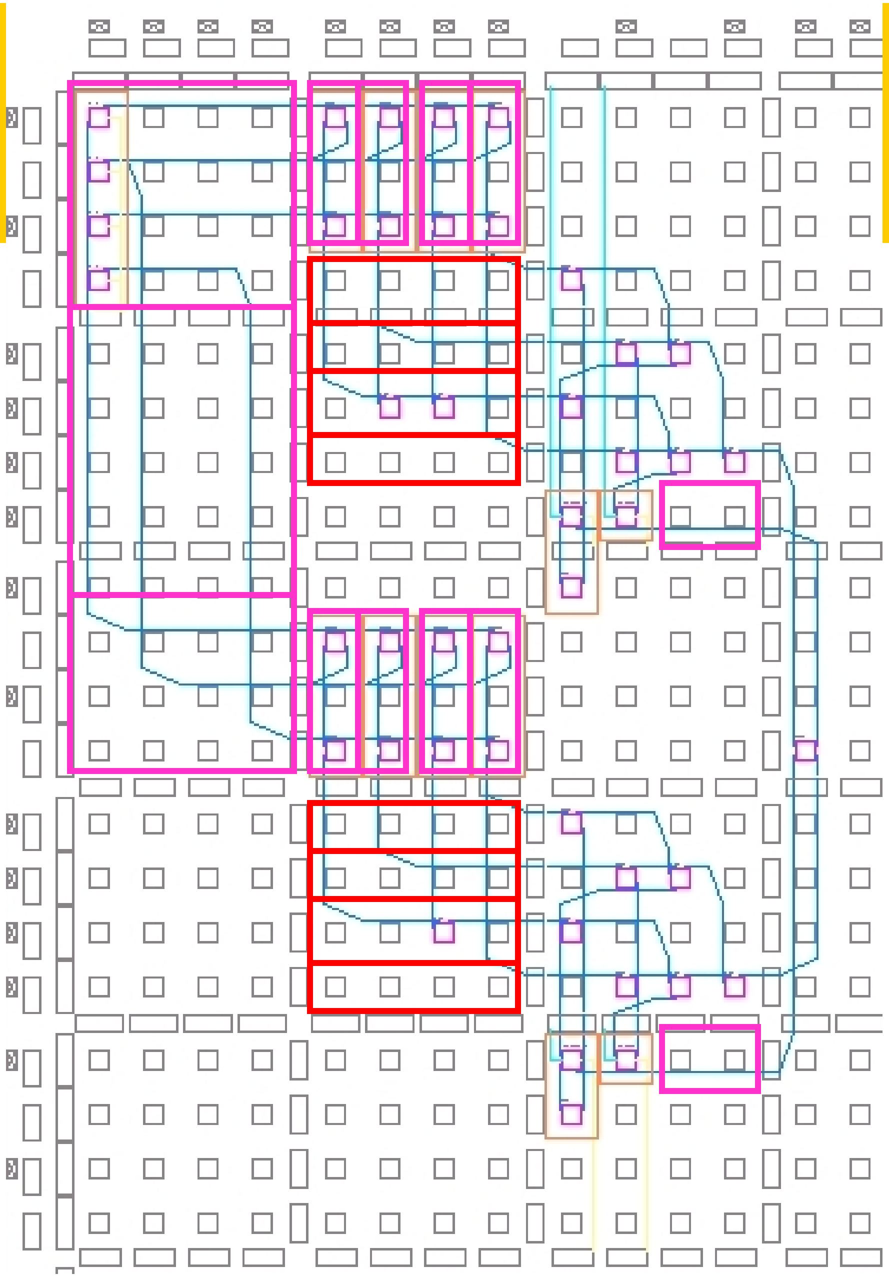
Module Generators

5. Guards:

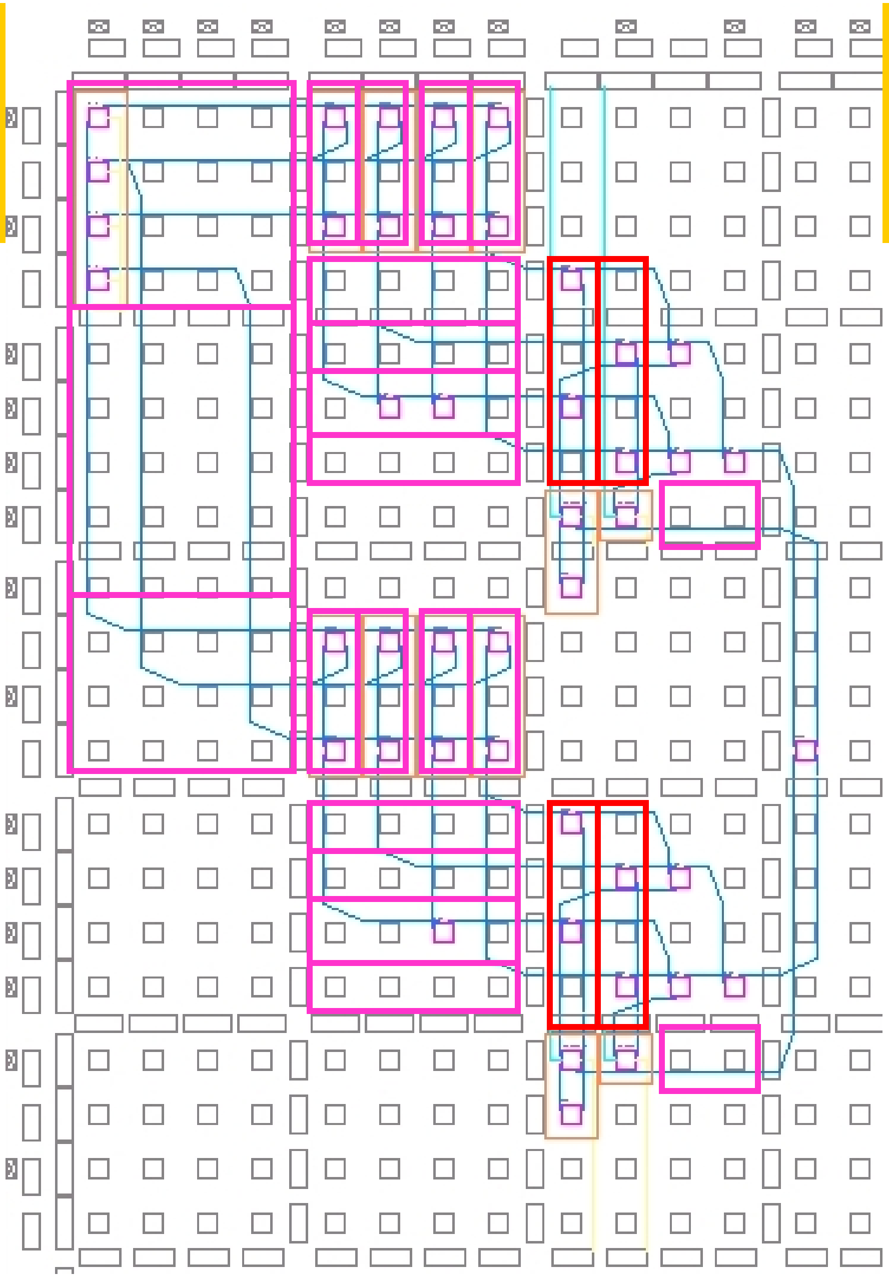
G(tlc, tlr, wi, ai, g, o);

- forms OR of selected minterm wires with output to right

- specified by coords of top left corner, width, and vectors of active inputs, inputs to be ORed, and inputs to be output below



Module Generators



6. Requestors:

R(tlc, tlr, hi, tp, ss, r);

- combine guards with current state and pass signals through their body

- specified top left corner, height, and vectors of throughputs (→), state selectors (←), and requestors

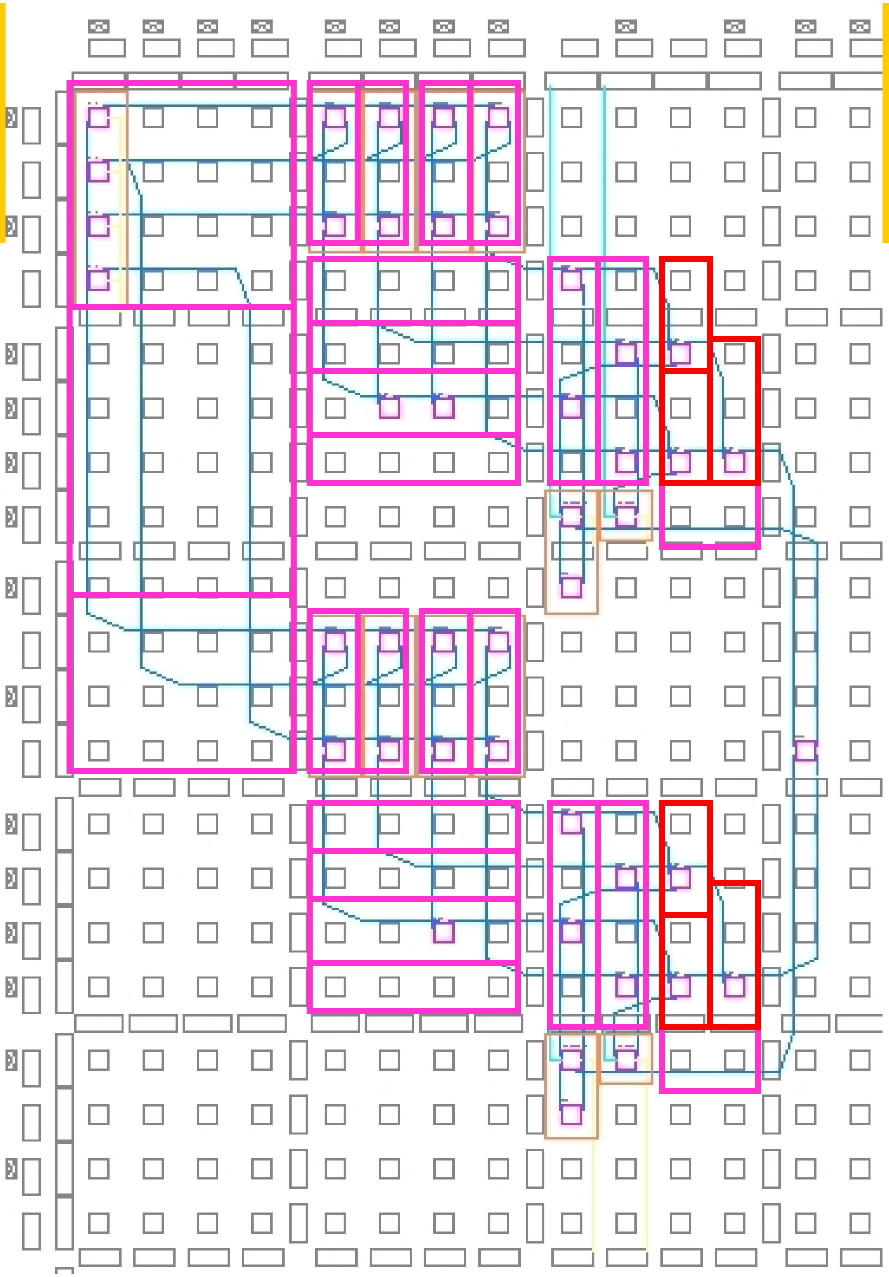
Module Generators

7. OR gate trees:

OR(tlc, tlr, hi, ai, of);

- forms tree to right and bottom, output to right, or both left and right

- specified by coords of top left corner, height, active input vector, and output direction flag



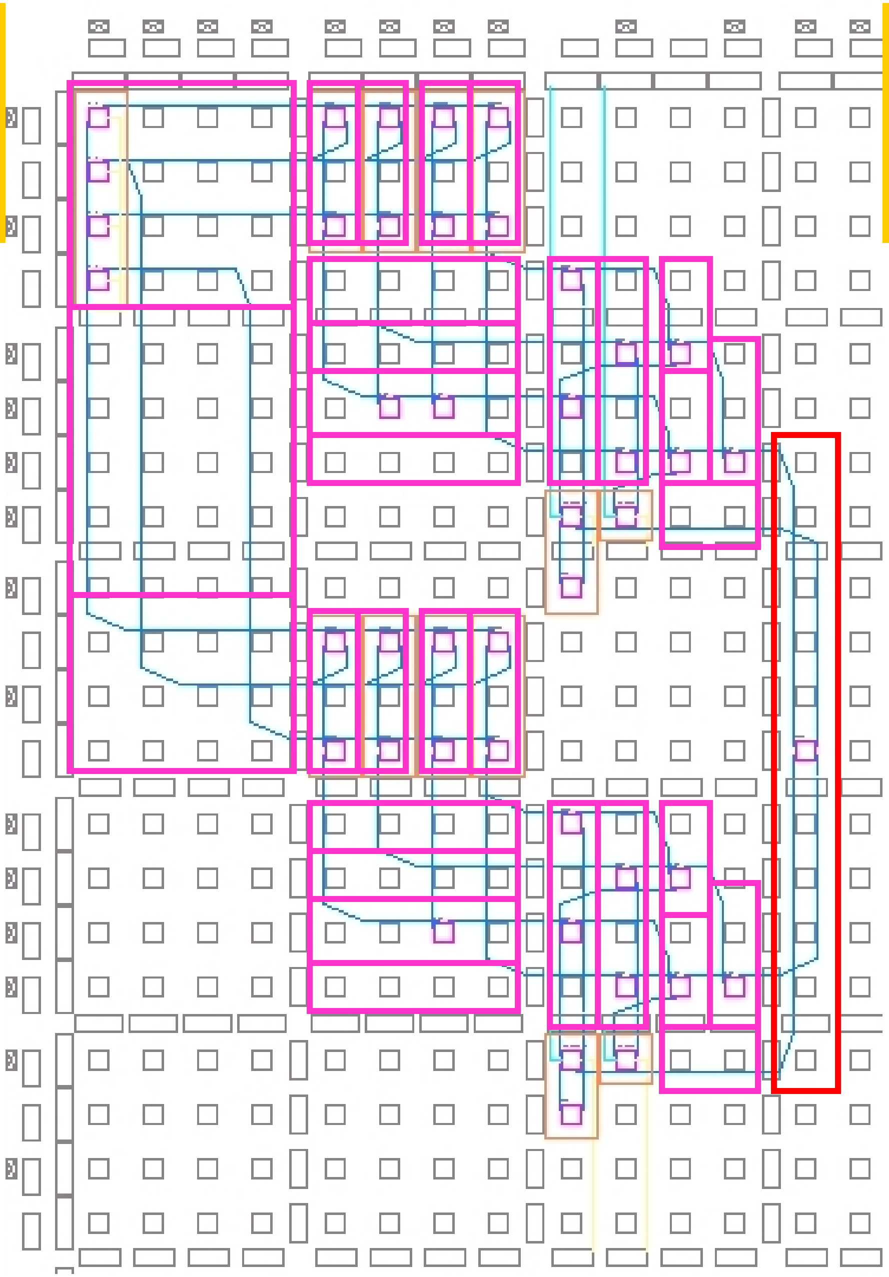
Module Generators

8. Synch logic:

SL(tlc, tlr, hi, ai);

- forms AND tree of inputs and distributes outputs to rows below inputs

- specified by coords of top left corner, height, and active input vector

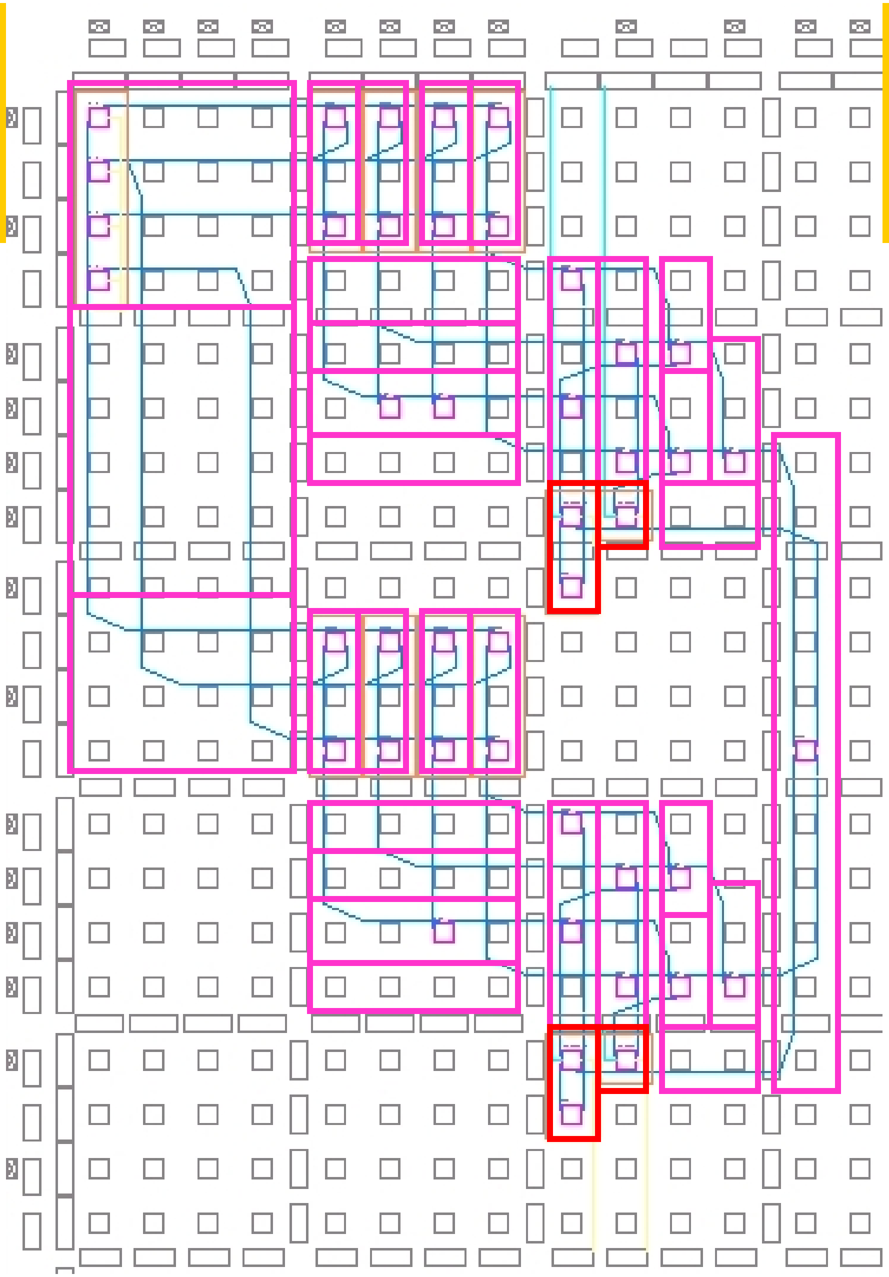


Module Generators

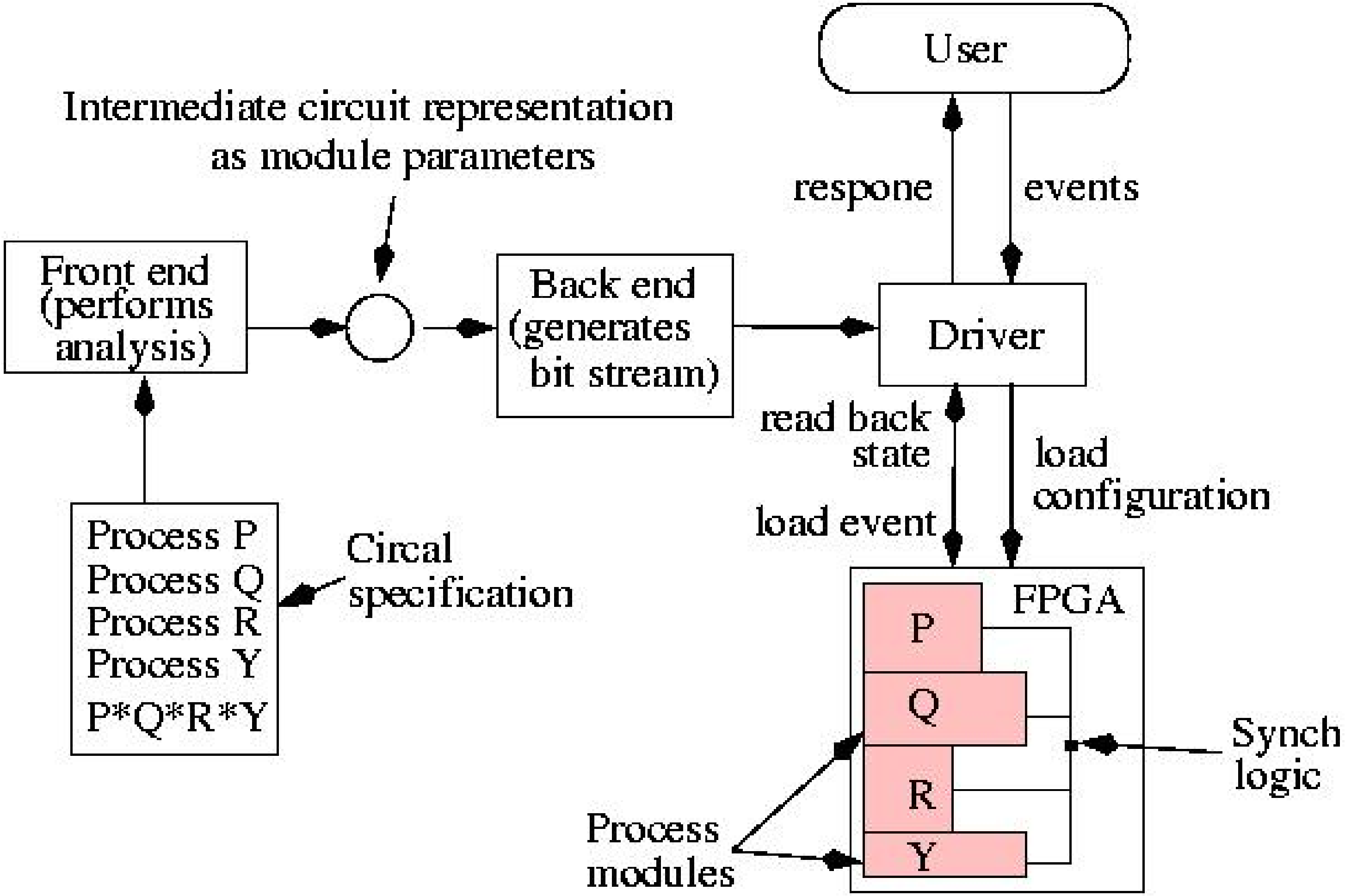
9. State registers:

SR(tlc, tlr, tf);

- implements initial or non-initial state using selector from N and enable from E
- specified by coords of top left corner, height, and state type flag



Overview of compiler operation



Drawbacks of compilation approach

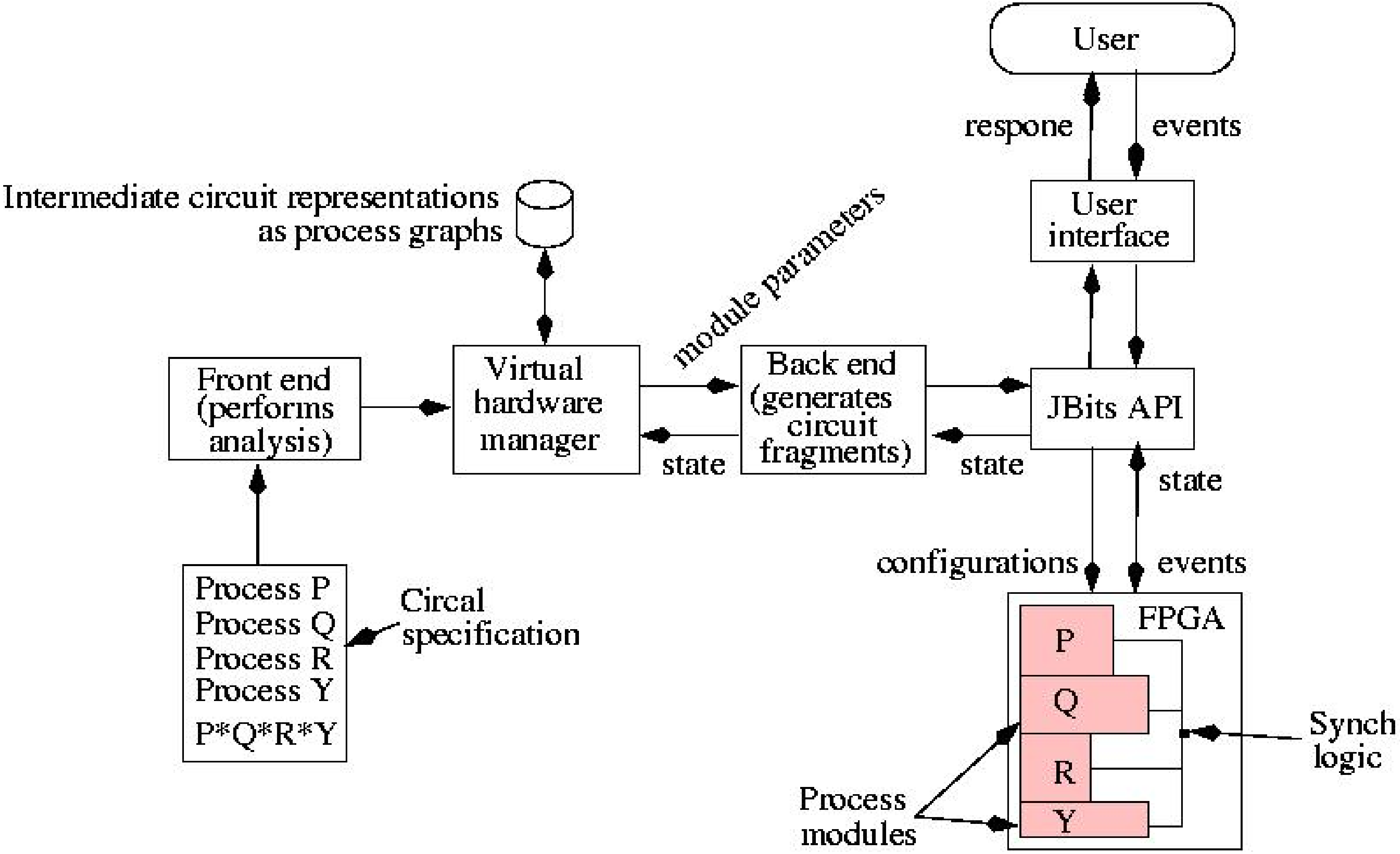
- Compile-time partitioning does not consider run-time *need* for resources
  - Which processes need to be concurrently active?
- Static allocations do not adapt to run-time *availability* of resources
  - Distributed & multitasked environments
- Static circuits do not readily support dynamic circuit behaviour or structure
  - Power of reconfiguration remains untapped

An FPGA interpreter for Circal

Interpreter concept

- We pre-process a spec until the functional parameters of modules are known
- At run time, the interpreter looks after loading modules on an as needs basis
  - Involves module placement, bitstream gen & config
- The amount of logic loaded depends upon resource availability
  - Currently load logic for state, but could dynamically load fragment of a new process hierarchy

Overview of interpreter operation

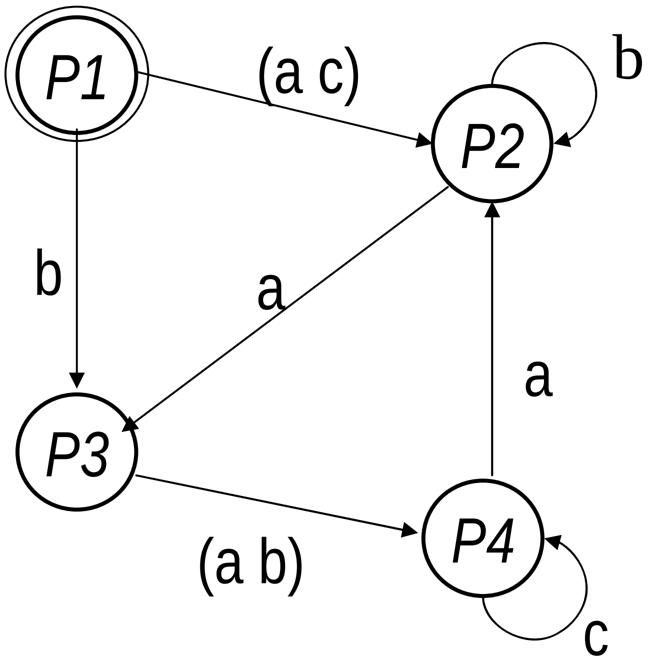


The Circal interpreter

- Extends the design flow to run-time management & ongoing FPGA configuration — circuit design does not complete until execution has finished
- Finalizes partitioning, logical, and physical mapping of circuits at run time
- Determines through feedback which components to implement next
- Elaborates & loads parts of the circuit as they are needed

Interpretation example

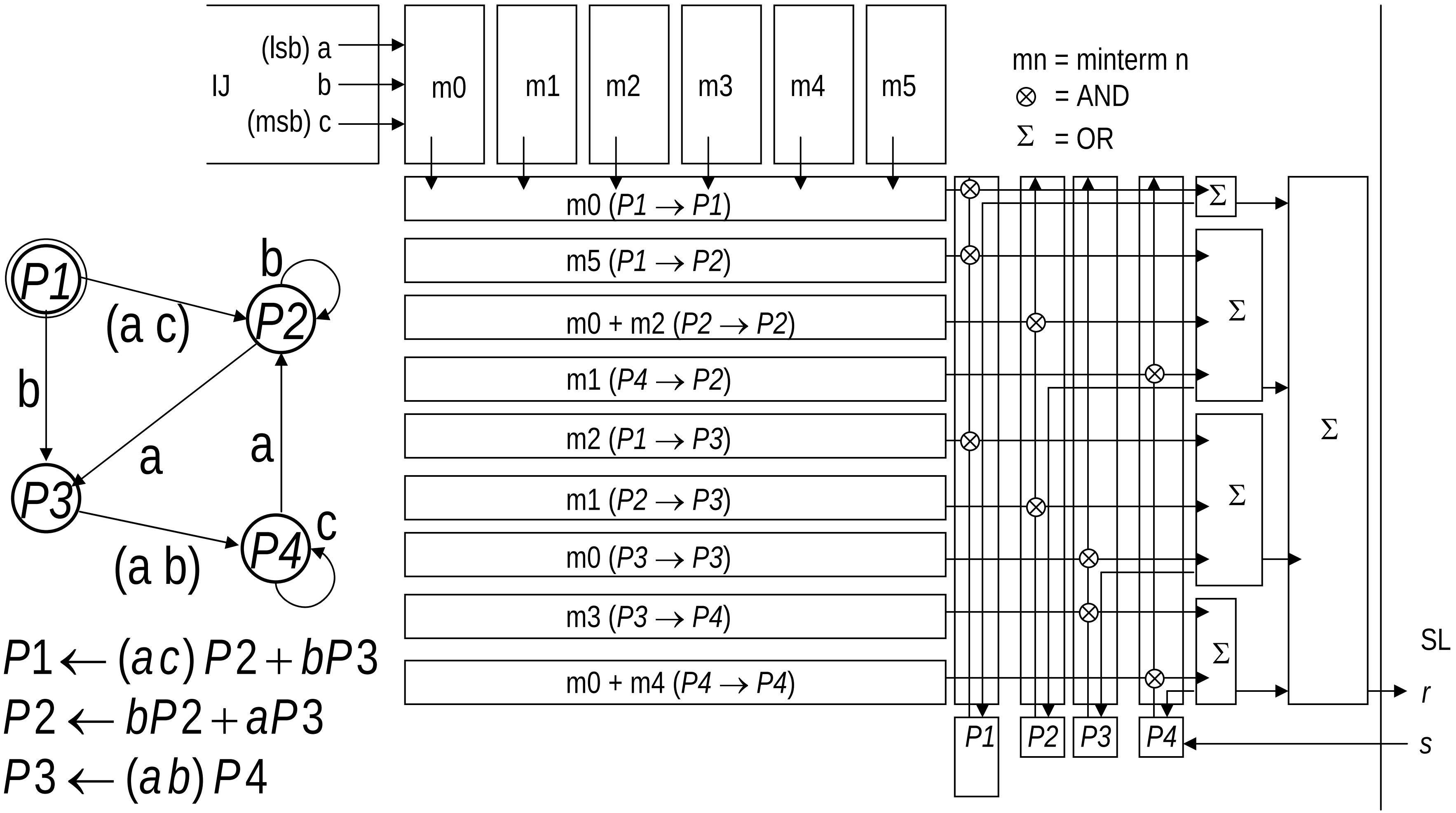
- Consider the FSM for process P with 4 states



- This process has the following Circal spec

$$
\begin{aligned}
P1 &\leftarrow (a\,c)\,P2 + bP3 \\
P2 &\leftarrow bP2 + aP3 \\
P3 &\leftarrow (a\,b)\,P4 \\
P4 &\leftarrow cP4 + aP2
\end{aligned}
$$

Static circuit implementation



mn = minterm n
⊗ = AND
Σ = OR

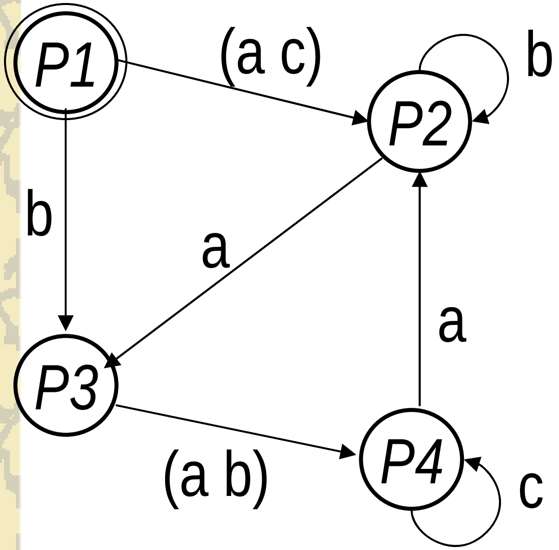
$$P1 \leftarrow (a\,c)\,P2 + bP3$$

$$P2 \leftarrow bP2 + aP3$$

$$P3 \leftarrow (a\,b)\,P4$$

$$P4 \leftarrow cP4 + aP2$$

Circuit modelling & partitioning



$$P1 \leftarrow (ac)\,P2 + bP3$$

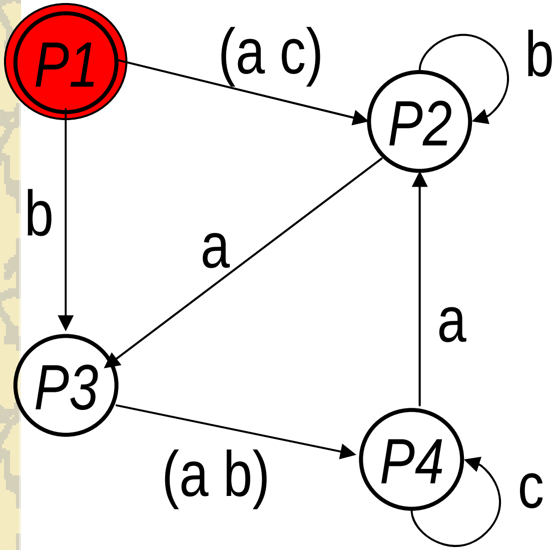
$$P2 \leftarrow bP2 + aP3$$

$$P3 \leftarrow (ab)\,P4$$

$$P4 \leftarrow cP4 + aP2$$

- Processes are modelled as state transition graphs and processes are partitioned according to their definitions

Circuit modelling & partitioning



$$P1 \leftarrow (ac)P2 + bP3$$

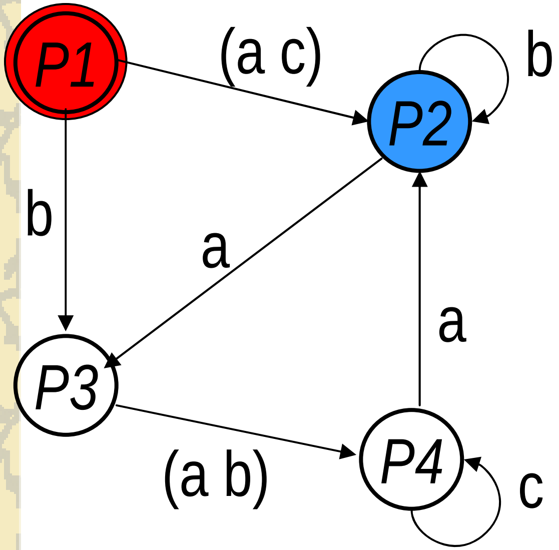
$$P2 \leftarrow bP2 + aP3$$

$$P3 \leftarrow (ab)P4$$

$$P4 \leftarrow cP4 + aP2$$

- Processes are modelled as state transition graphs and processes are partitioned according to their definitions
- Initially, the interpreter implements a sub-graph rooted at the initial state

Circuit modelling & partitioning



$$P1 \leftarrow (ac)P2 + bP3$$

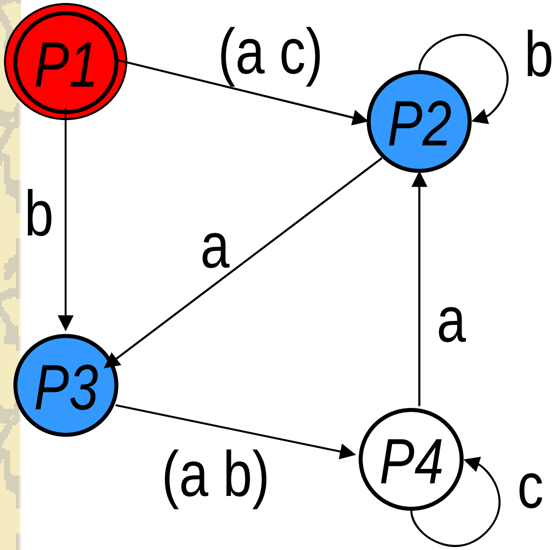
$$P2 \leftarrow bP2 + aP3$$

$$P3 \leftarrow (ab)P4$$

$$P4 \leftarrow cP4 + aP2$$

- Processes are modelled as state transition graphs and processes are partitioned according to their definitions
- Initially, the interpreter implements a sub-graph rooted at the initial state
- Nodes are included breadth-first until it is not possible to fit the transition logic for the next state

Circuit modelling & partitioning



$$P1 \leftarrow (a\,c)\,P2 + bP3$$

$$P2 \leftarrow bP2 + aP3$$

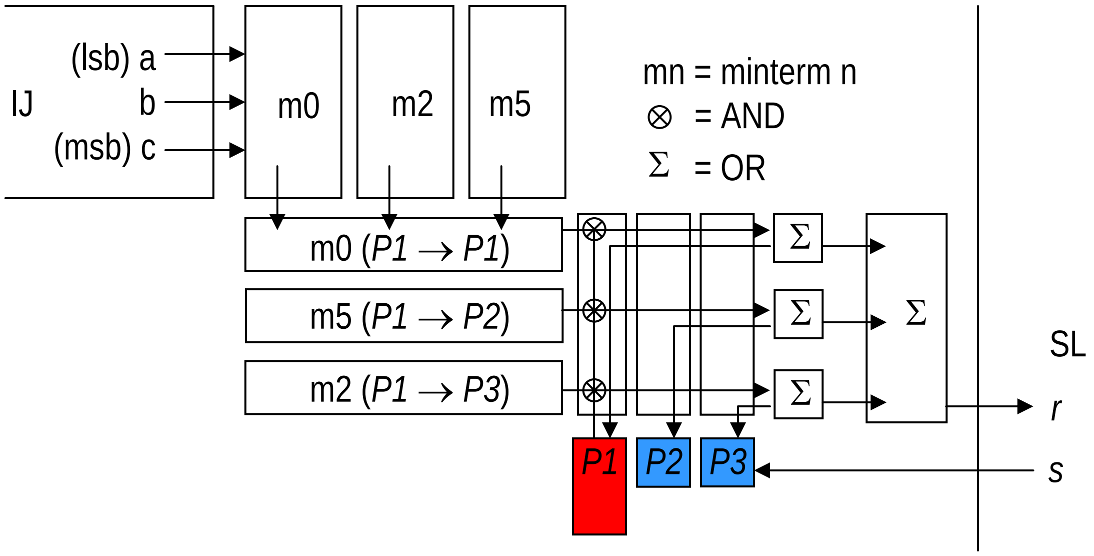
$$P3 \leftarrow (a\,b)\,P4$$

$$P4 \leftarrow cP4 + aP2$$

- Processes are modelled as state transition graphs and processes are partitioned according to their definitions
- Initially, the interpreter implements a sub-graph rooted at the initial state
- Nodes are included breadth-first until it is not possible to fit the transition logic for the next state

Example

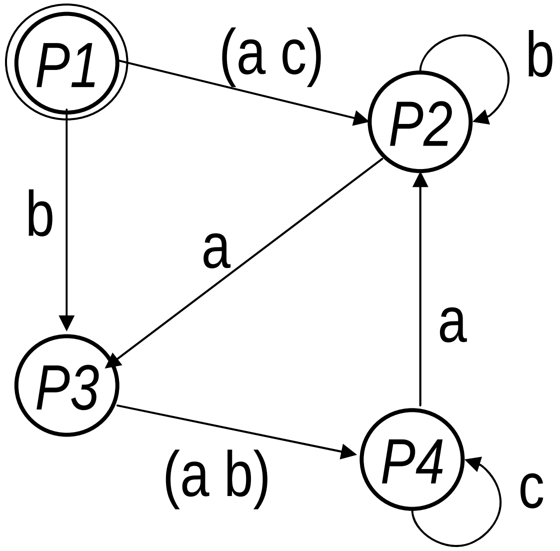
- Suppose the array area for process P can only accommodate the behaviour for state P1
- To determine which transition occurred, *boundary state registers* for P2 and P3 are needed as well



Determining circuitry to load next

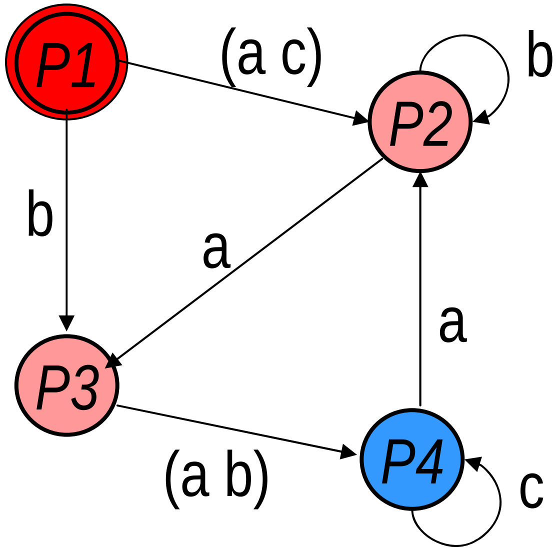
- When the boundary of the implemented sub-graph is reached, the interpreter *builds a new sub-graph* rooted at the boundary state that has become active
- We use a quick estimate of the additional space needed by a state and its transition logic
  - Estimator based on number of transitions from state

Constructing the new sub-graph



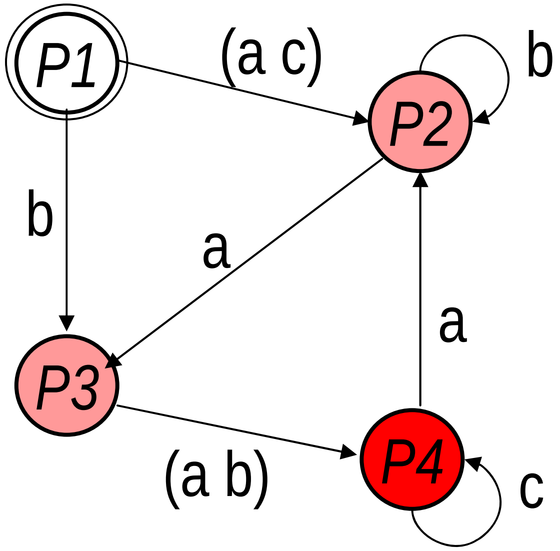
- Suppose we can support the logic for T=6 transitions in total

Constructing the new sub-graph



- Suppose we can support the logic for T=6 transitions in total
  - P1, P2, & P3 can be implemented with T=5, but the inclusion of P4 with t4=2 is deemed infeasible

Constructing the new sub-graph



- Suppose we can support the logic for T=6 transitions in total
  - P1, P2, & P3 can be implemented with T=5, but the inclusion of P4 with t4=2 is deemed infeasible
  - When P4 becomes active, P2, P3, & P4 form a stable configuration

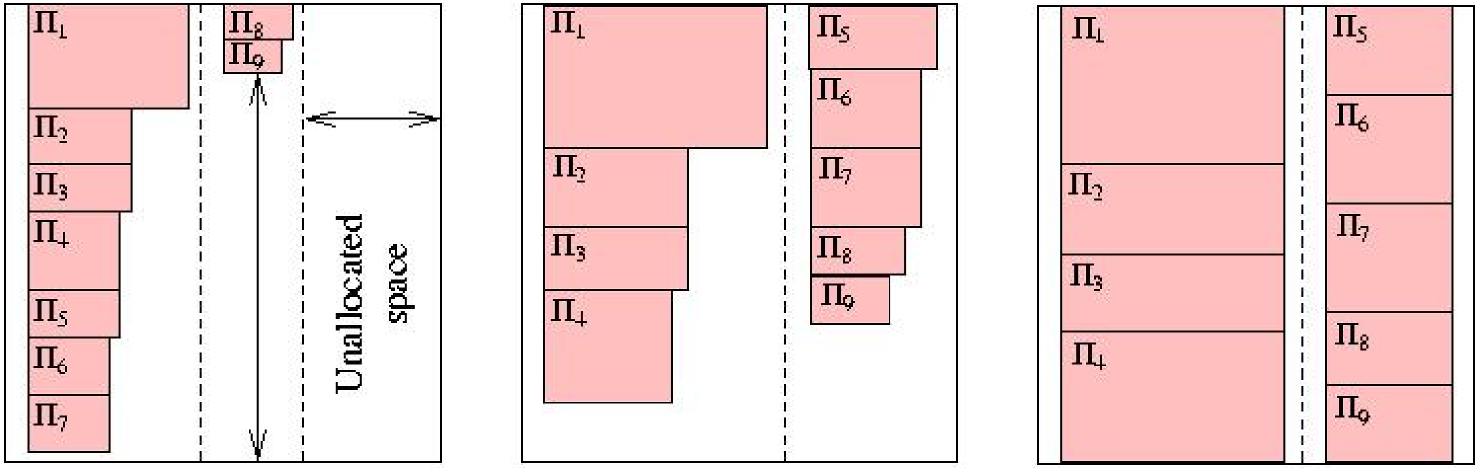
Detecting the need for reconfiguration

- We support two modes of operation:

1. *Observed mode*: all process states are polled each cycle
2. *Unobserved mode* (such as in an embedded application): some small circuitry is added to each process in order to interrupt the VHM when a boundary state is reached

FPGA partitioning

- FPGA area is statically partitioned to simplify run-time reconfiguration
- Initially, space to accommodate the largest state for each process is allocated — this is then expanded to provide more space for additional states when possible



After Step 3

After Step 5

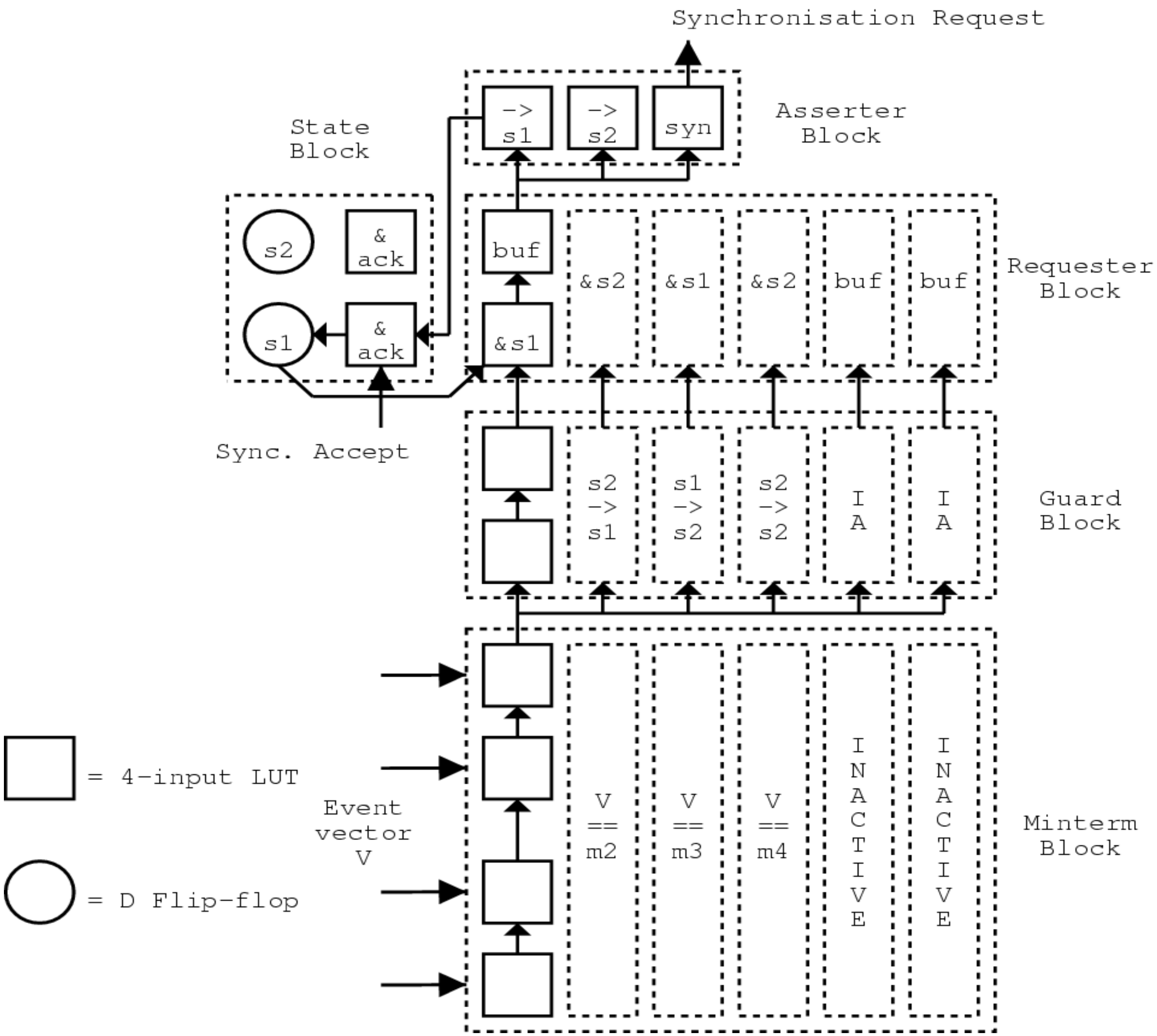
After Step 6

Experimental assessment

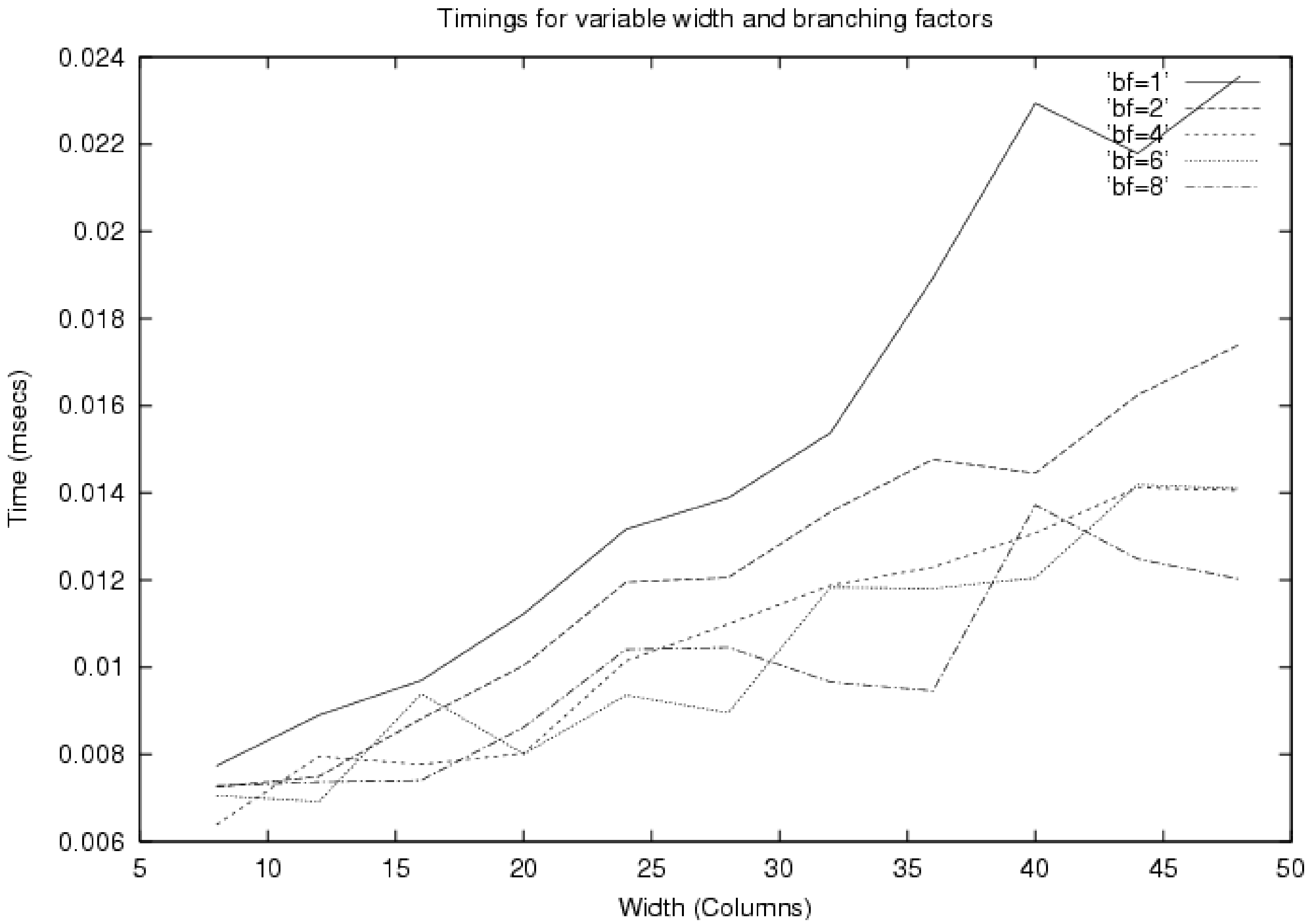
- Random process generation
- Timing of sub-graph selection in front end with varied branching factors and circuit widths
- Worst-case timing measurements for circuit initialisation and update in the back end
- Used 500Mhz, 512MB PIII, Java, JBits, and Celoxica XCV1000 board

Virtex layout

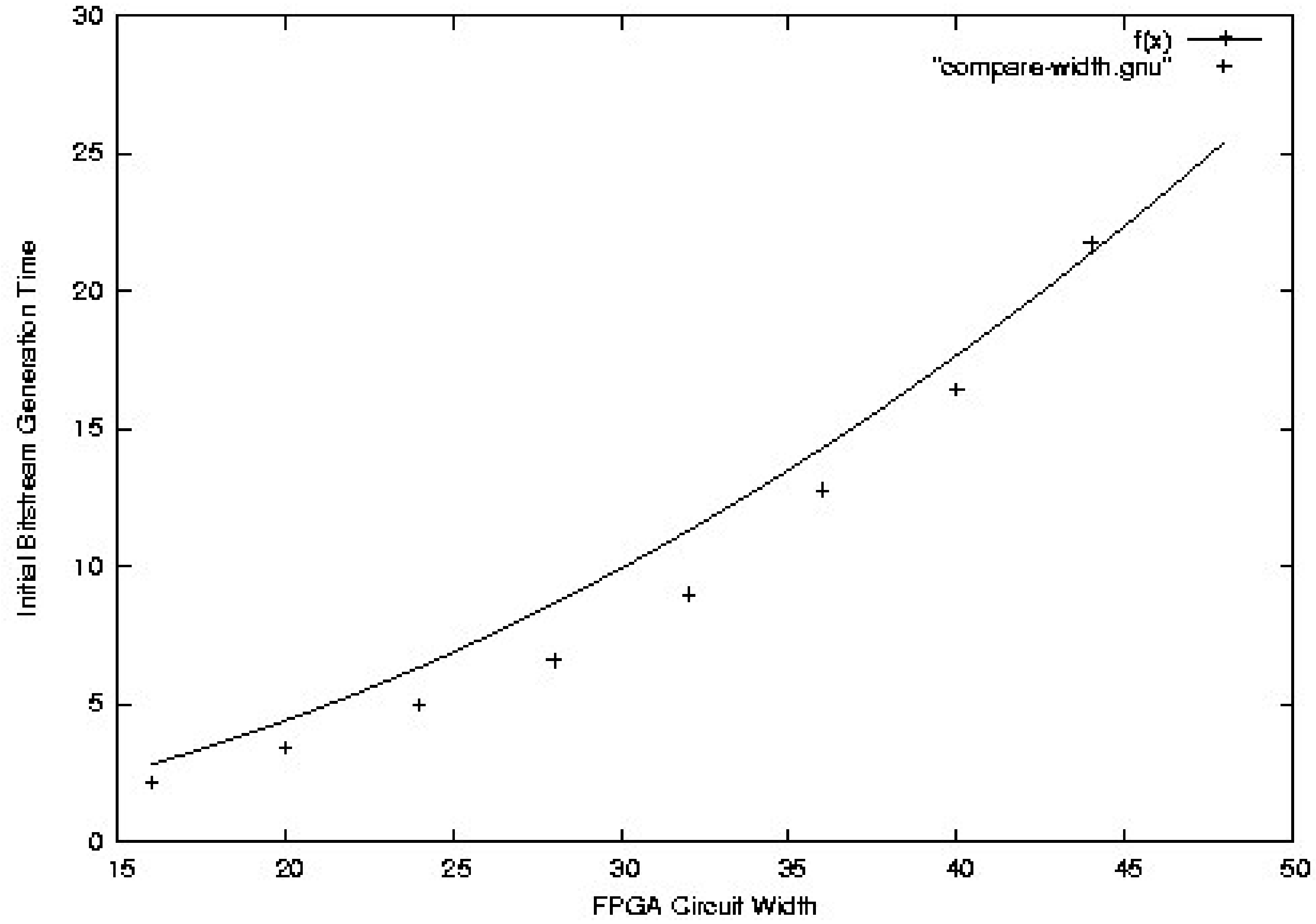
- Exploits fast carry chains and minimizes number of reconfiguration frames
- Deploys routing framework to reduce rerouting overheads



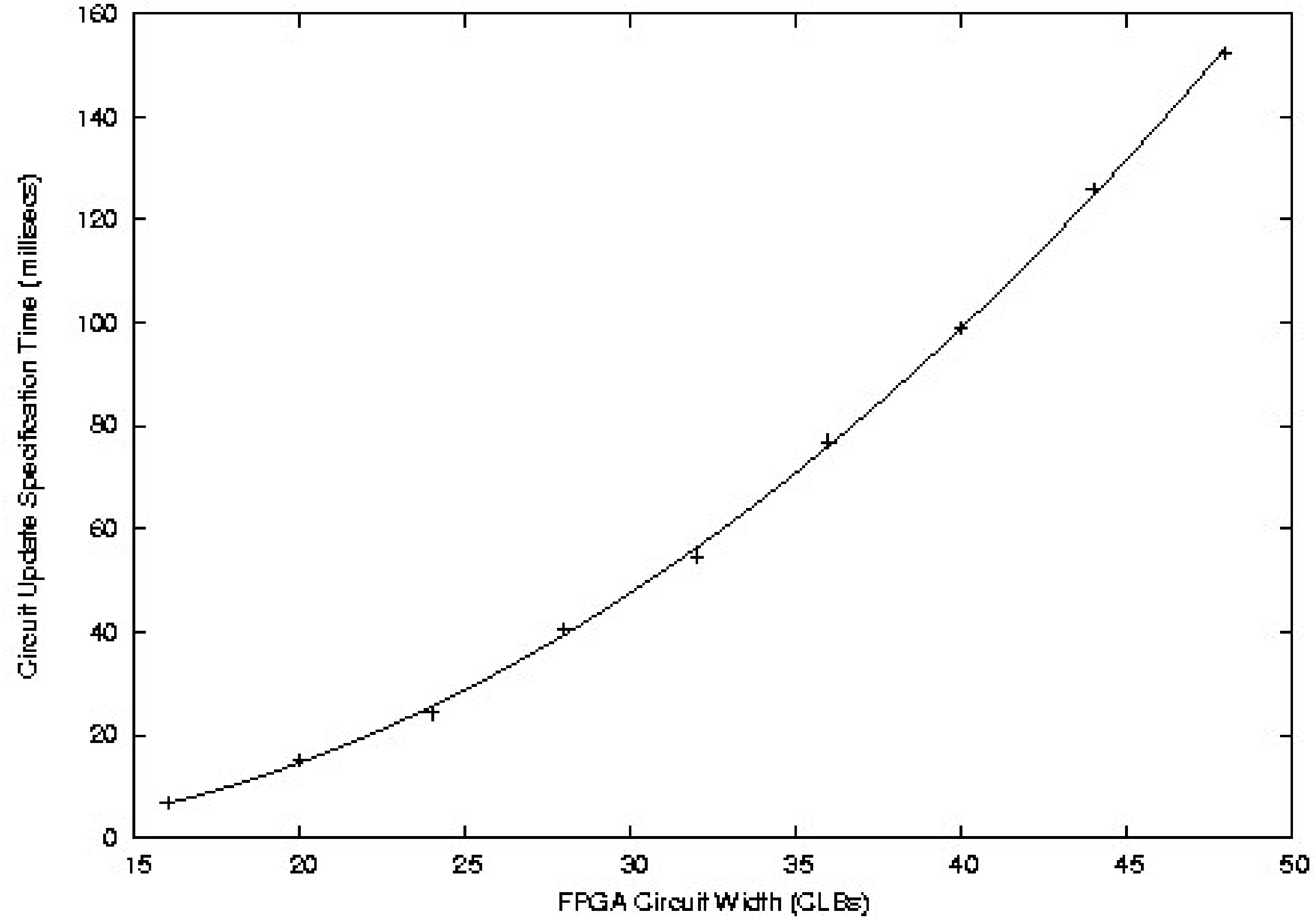
Sub-graph selection time (ms) vs Process width (CLB cols)



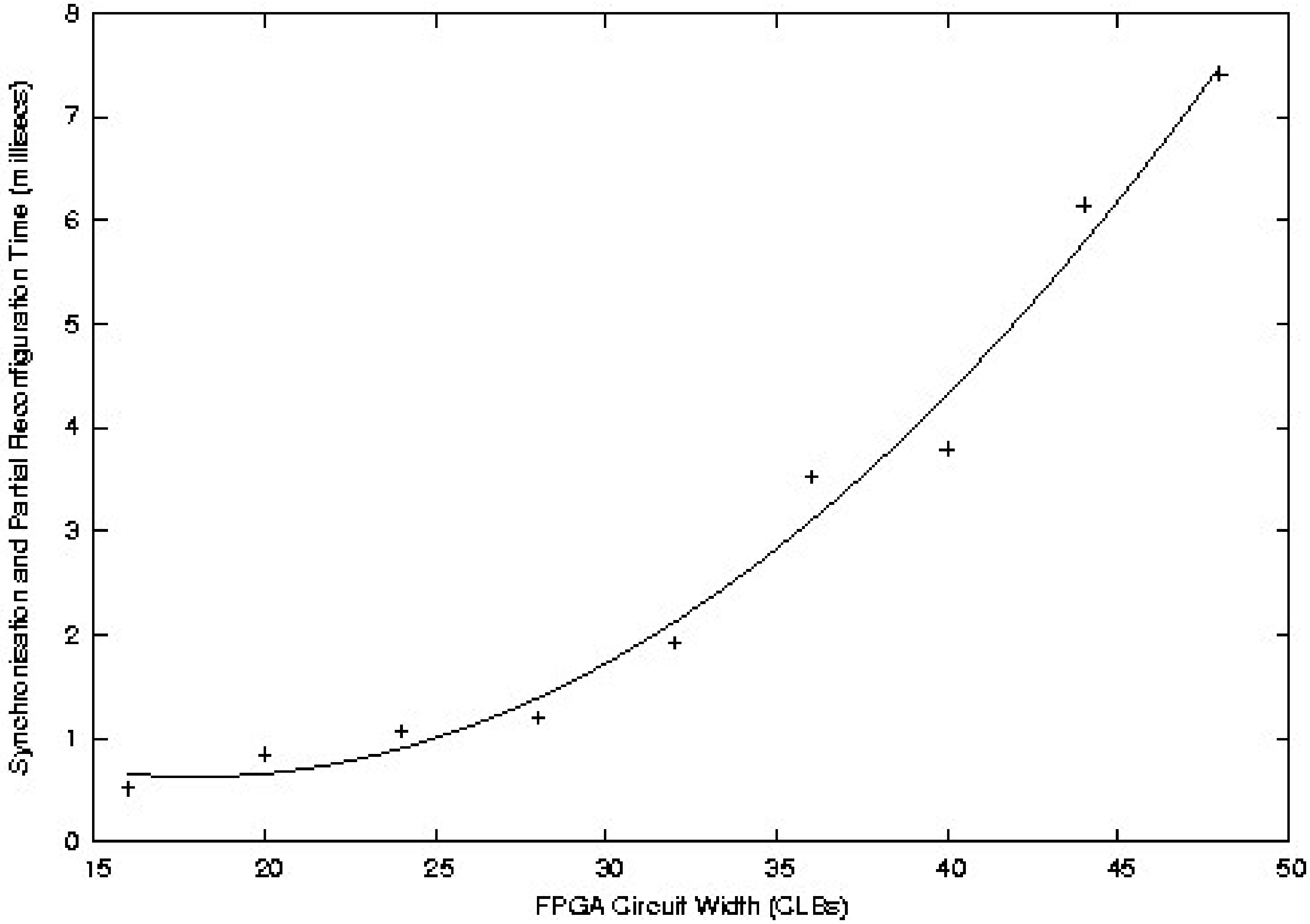
Initial bitstream generation time (s) vs Process width (CLB cols)



Generating circuit updates (ms) vs Process width (CLB cols)



Reconfiguration time (ms) vs Process width(CLB cols)



Analysis

- High routing costs with JBits, even for highly structured circuits
  - Constrains the designs that can be run-time reconfigured
- Capable of reconfiguring within 100ms
  - The current methodology is fine for control applications that can tolerate these delays
- Lower bound on implementation delays in the order of 10ms
  - More hardware required for applications that cannot tolerate this

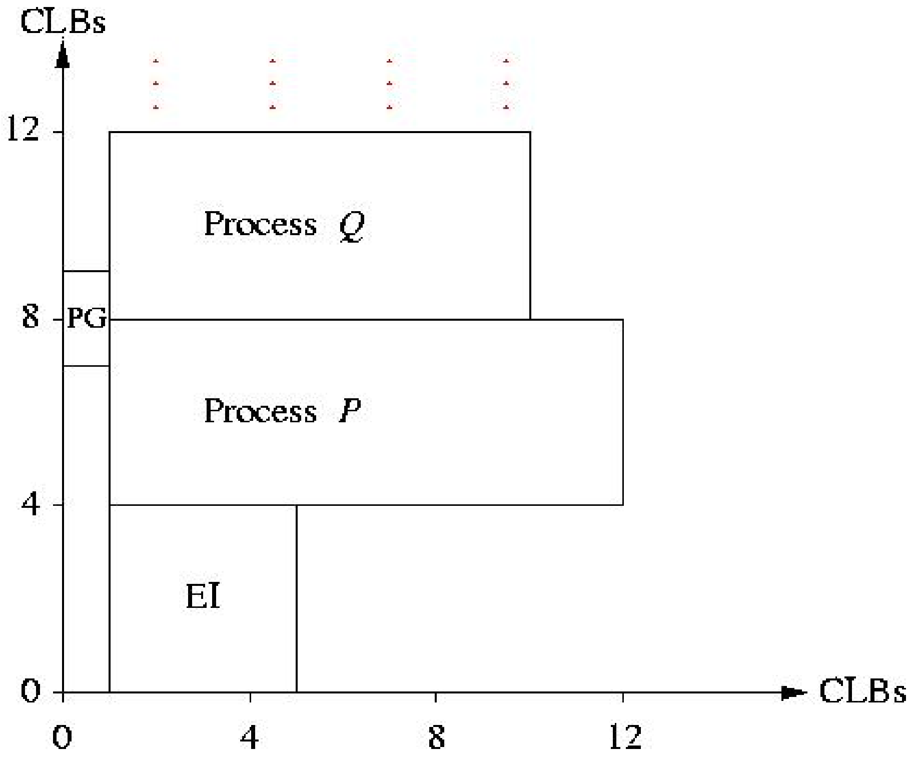
Further interpreter work

- Performance improvement
  - Better router
  - Caching loops
  - Better partitioning strategies?
- Determining how to efficiently adapt to dynamic partition sizes
- Incorporating data flow into the Circal interpreter
- Taking user’s performance objectives into account

Ongoing work

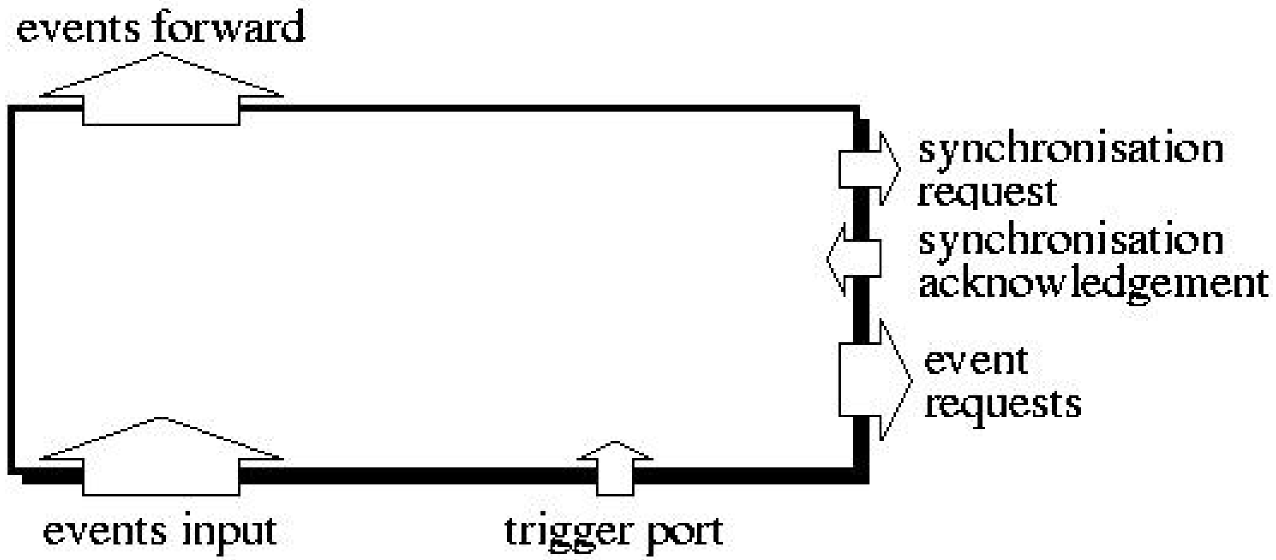
Generalized processes

- Work done by Jérémie Detrey to implement *hierarchy, abstraction* and *process creation*
- Developed on Wildcard XCV300 implementation of compiler



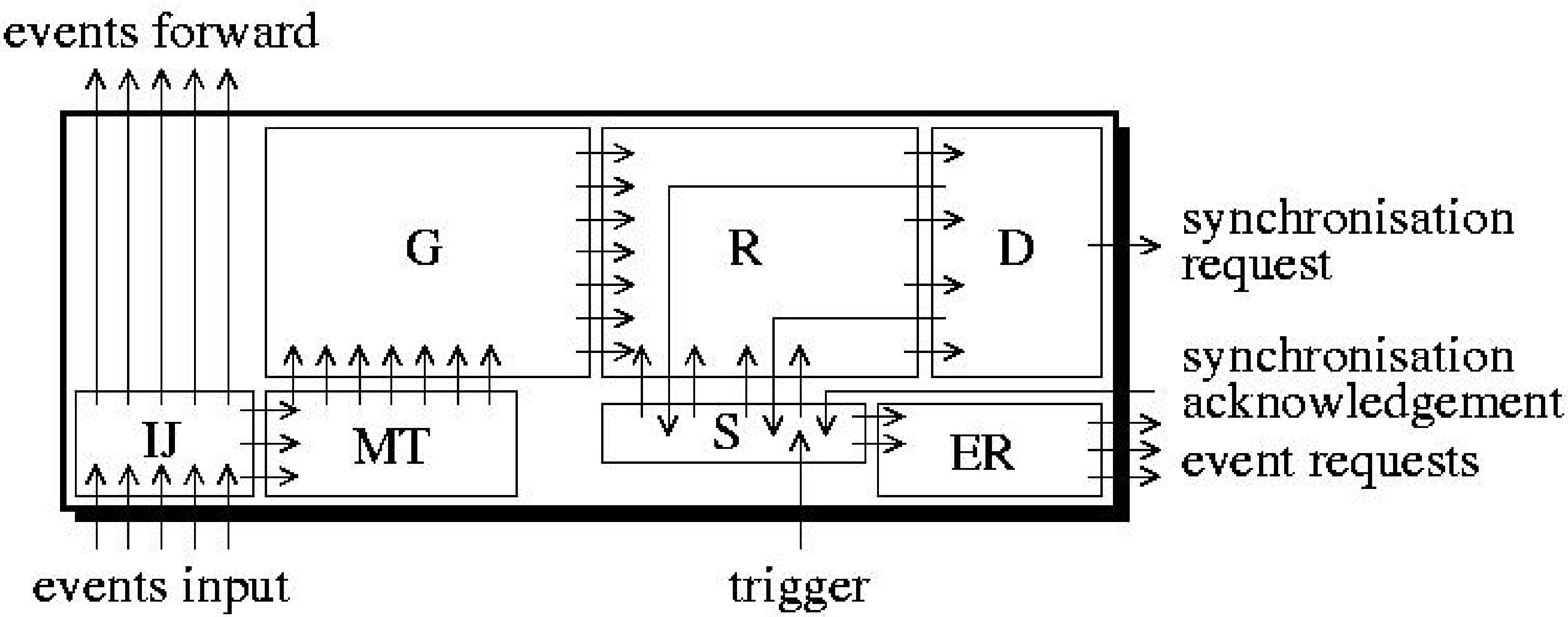
Abstract process interface

- For the variety of process blocks required, a *common process interface* has been defined

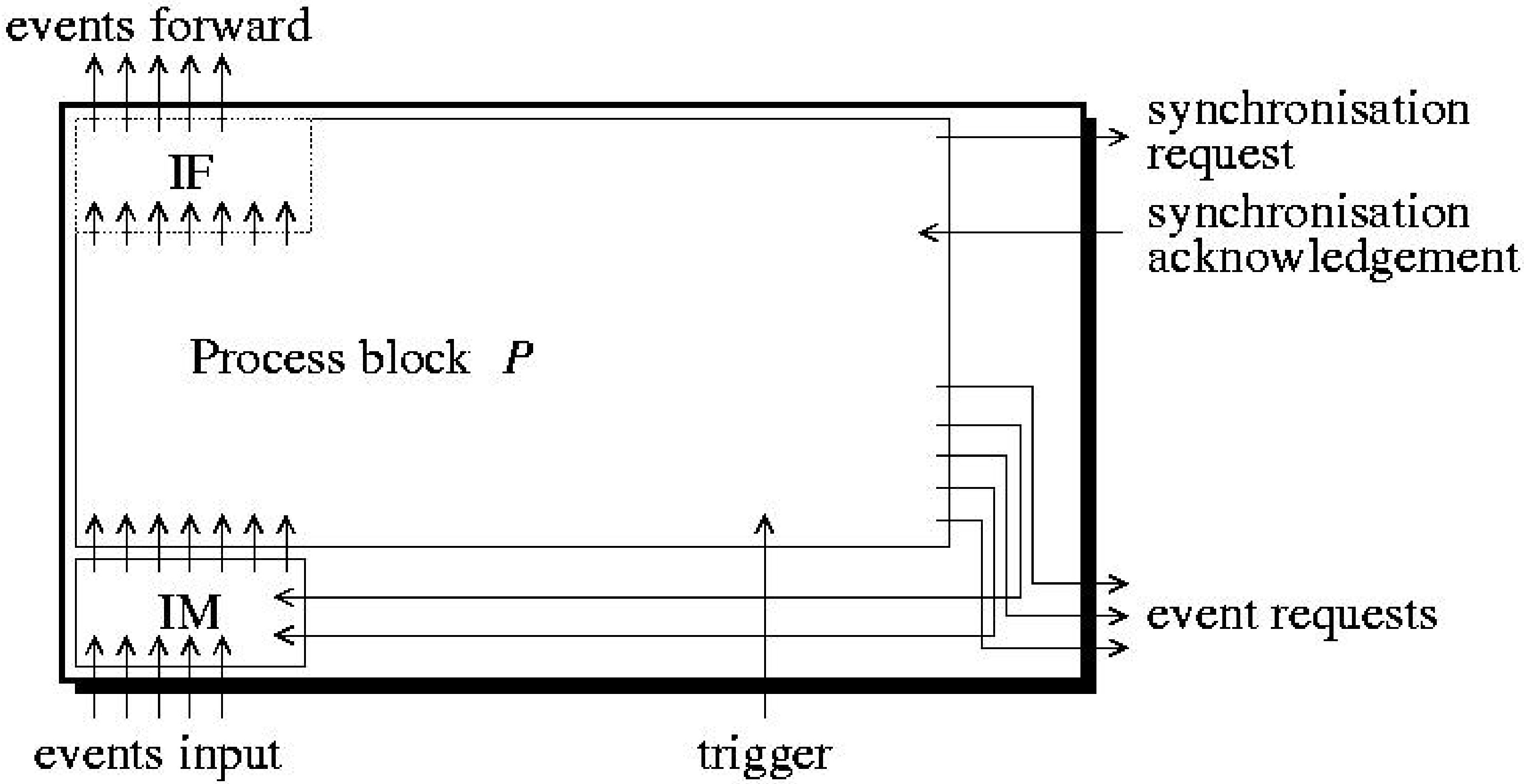


FSM process

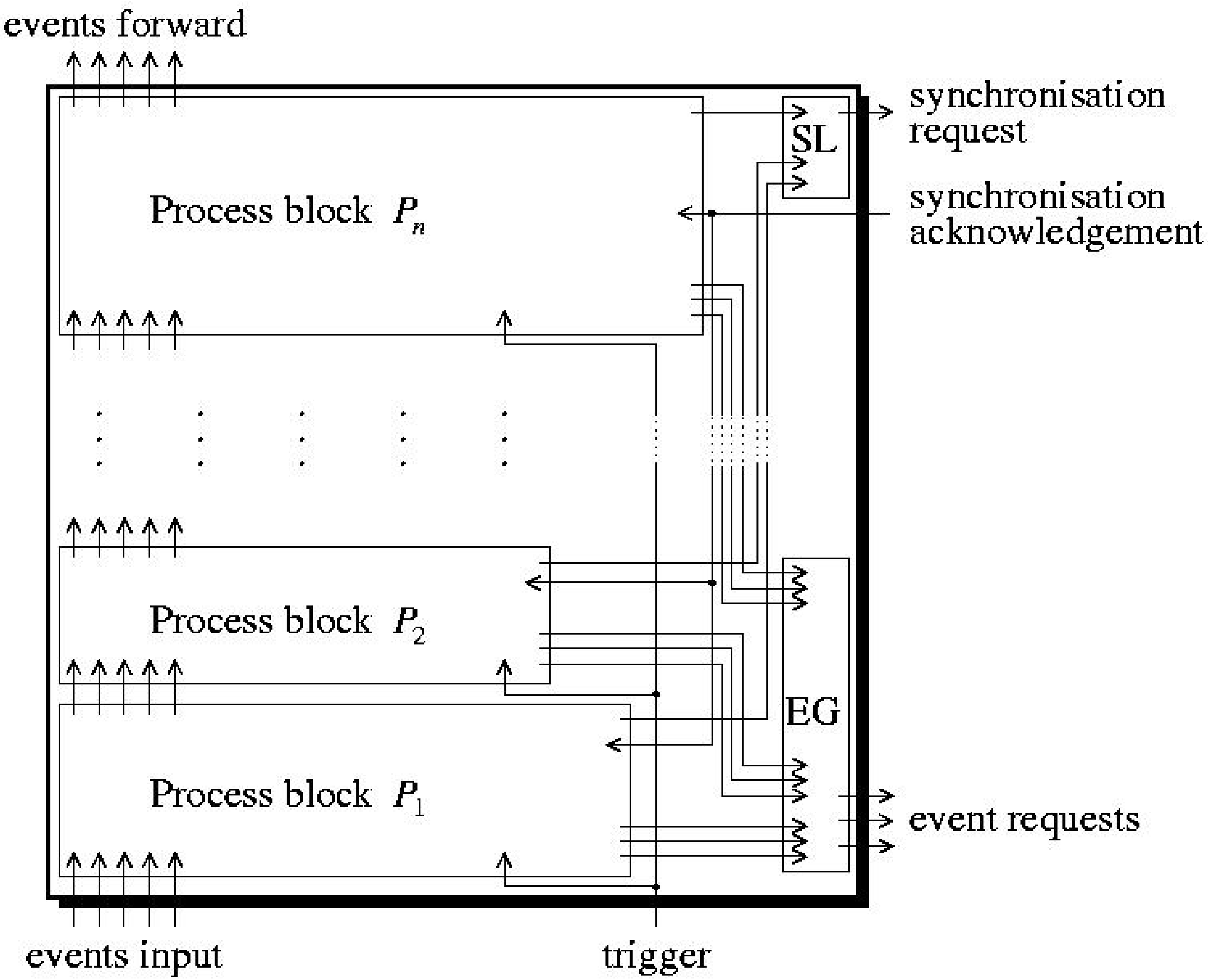
- As before, only allows *choice* and *guarding*, but process can be switched on/off and provides for *abstracted (internal)* events



Implementing event abstraction



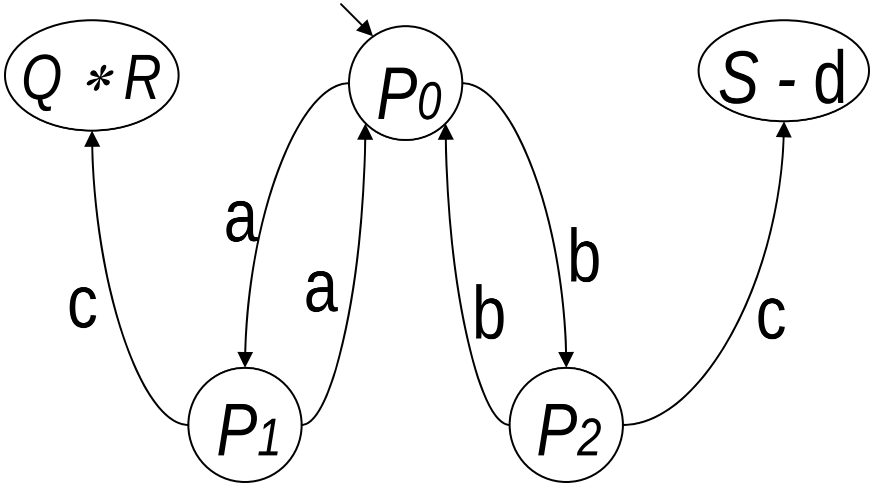
Hierarchical composition



Hybrid process

- FSM-like behaviour
- State could be *composition* or *abstraction*
- For example, consider process P_0 defined as

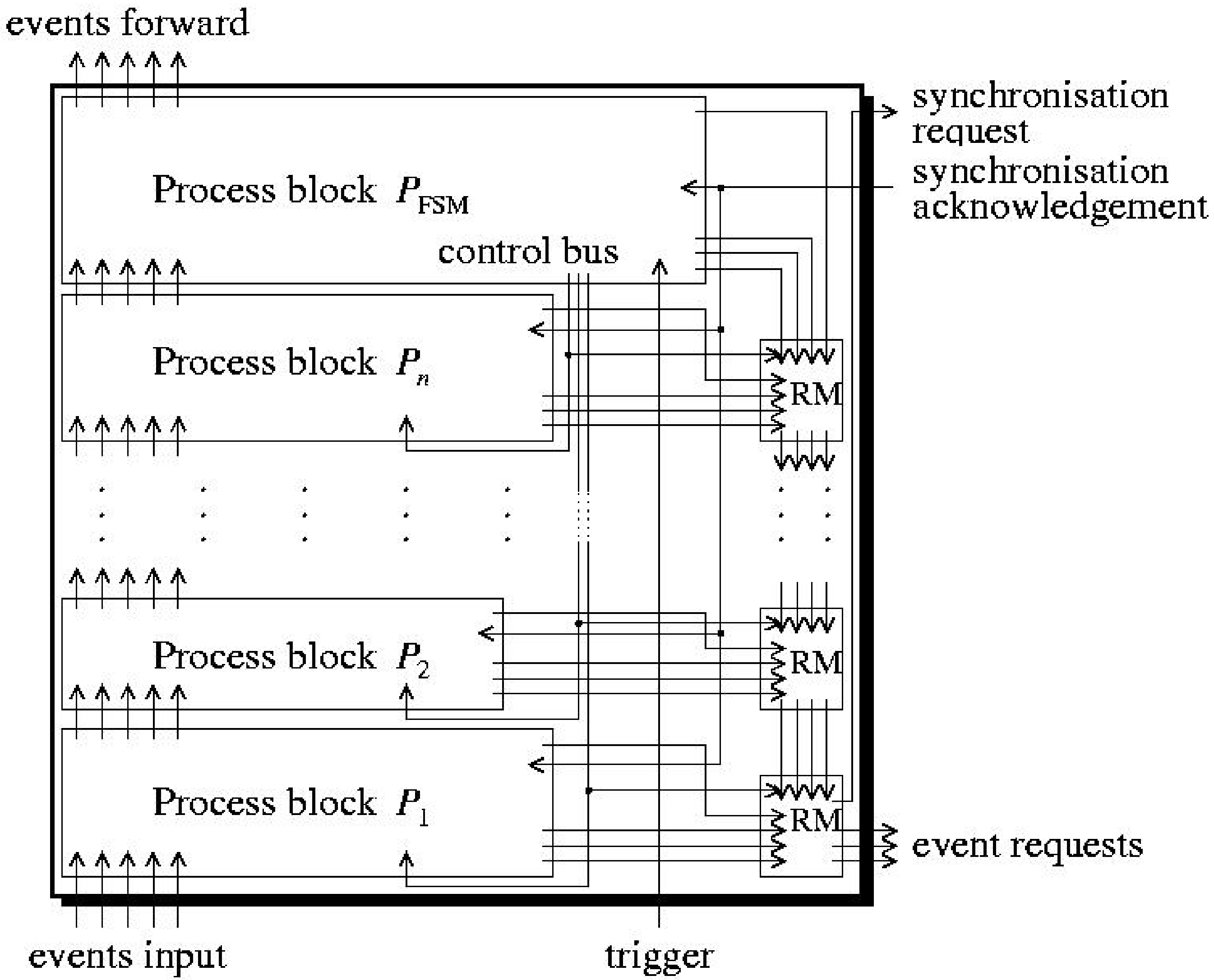
$$P_0 \leftarrow aP_1 + bP_2$$
$$P_1 \leftarrow aP_0 + c(Q * R)$$
$$P_2 \leftarrow bP_0 + c(S - d)$$



Implementing hybrid processes

- Determine the FSM part, called P_{FSM}
- Determine the other composition and abstraction processes, $P_1,\ldots, P_n$
- Currently, lay them all out statically

Hybrid process block layout



Status

- Tested these ideas by implementing (on the Wildcard) a Turing Machine using Circal as the specification language
  - Involved use of hierarchy, composition, and abstraction
  - Statically preallocated tape of given length, but activated tape squares as head moved over them

Future work

- Support generalized processes in the interpreter
  - Replace the interpreter front end
  - Define dynamic layouts for generalized processes
- Develop concepts for dynamic allocation of hybrid processes

Conclusion

In summary

- We’ve made progress towards describing and implementing static and automatically virtualized process logic from a high-level
  - The ability to describe & implement traditional datapath elements needs to be incorporated
- The real work in formally describing and implementing dynamic circuits is still to come…

Modelling reconfigurable devices

- How can dynamic process creation/destruction be described in Circal?
  - Need additional semantics
  - Is Circal, indeed process algebra best?
  - What hardware realisation supports the semantics?
- Effort also going into deriving a low-level model of reconfiguration using Circal
  - So far, we are focusing on the “reconfigurator”
  - Intend this model to act as target of compiler/interpreter
  - May need additional high-level syntax to steer compiler