

ACK - An Asynchronous Circuit Design Framework

Prabhakar Kudva
IBM T.J. Watson Research Center

Ganesh Gopalakrishnan, Hans Jacobson
University of Utah
Computer Science Department

Motivation

- Async Group at U of Utah has been working on a
 - High level synthesis framework
- That will allow
 - Easy experimentation of system and controller level performance issues
- Further advances in the framework will be studied
 - In the context of large driving examples
- This framework is the subject of this talk

Overview

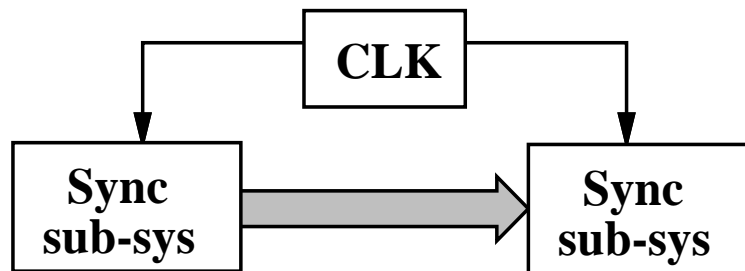
- Synchronous vs. Asynchronous
- Different asynchronous approaches
- ACK: An Asynchronous Design Framework
 - Design Specification
 - High Level Synthesis
 - Low Level Synthesis
- Conclusions and Future Work

Conceptual Differences

Synchronous - Time domain

Lock step execution

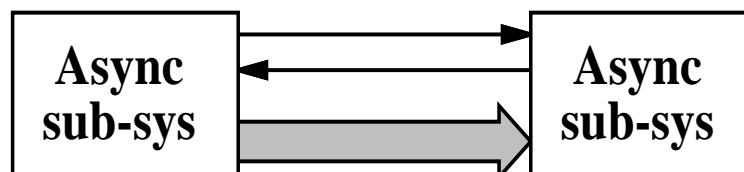
Requires both sequencing and timing for implementation



Asynchronous - Event domain

Execute on individual request

- Requires only sequencing for implementation
- Timing used for performance optimization



Synchronous Difficulties

- **Skew related problems**
 - System level and on-chip
- **Worst case performance**
 - Slowest sub circuit
 - Temperature and voltage variations
- **Power**
 - Idle power dissipation
 - High peak-power

Asynchronous Difficulties

- **Sensitive to glitches**
 - Susceptible to noise
 - Require hazard-free logic
- **Handshake and completion detection latency**
- **Different design methodologies**
 - Lack of available tools and joint focus on research

General Asynchronous Advantages

Potential advantages:

- **Performance**

- Average completion time due to

- Data input

- 29% improvement in 64-bit Brent-Kung adder

- Operating conditions

- 38% improvement at 22C vs. 100C in 32-bit diffeq

- **Power**

- Automatic powerdown
- Voltage scaling

Demonstrated advantage:

- **Modularity and composability**

- Simplify system organization
- Increased system lifetime
- Easy performance upgrading

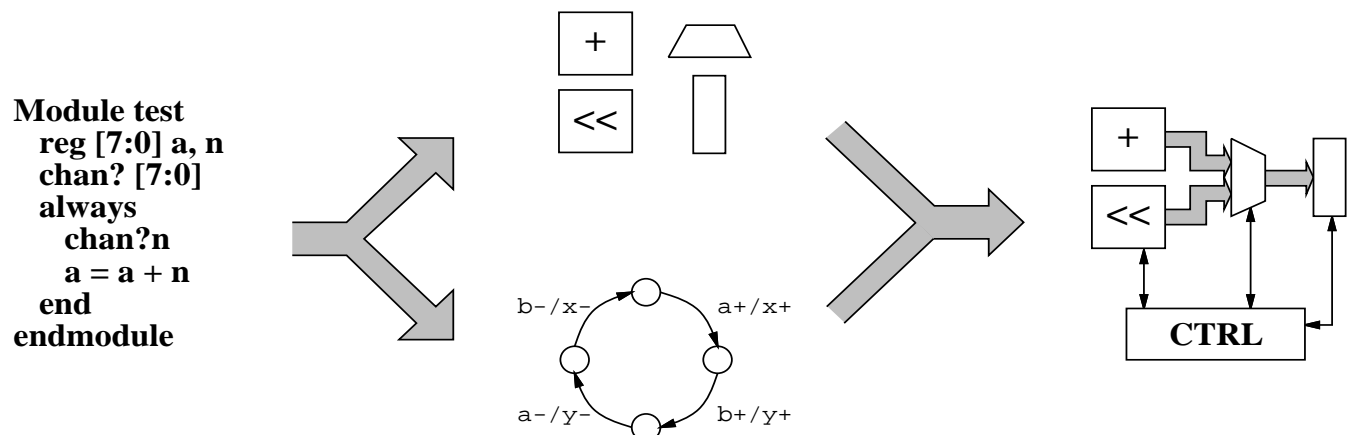
Asynchronous Design Styles

Previous work:

- **High level synthesis by translation**
 - Macromodules
 - Program transformation
- **Low level synthesis by graphs**
 - Signal Transition Graphs

Our approach:

- **Integrate automatic HL and SM synthesis**
 - High level synthesis targeting customized datapath and control
 - Low level state machine synthesis
 - Similar to synchronous methods

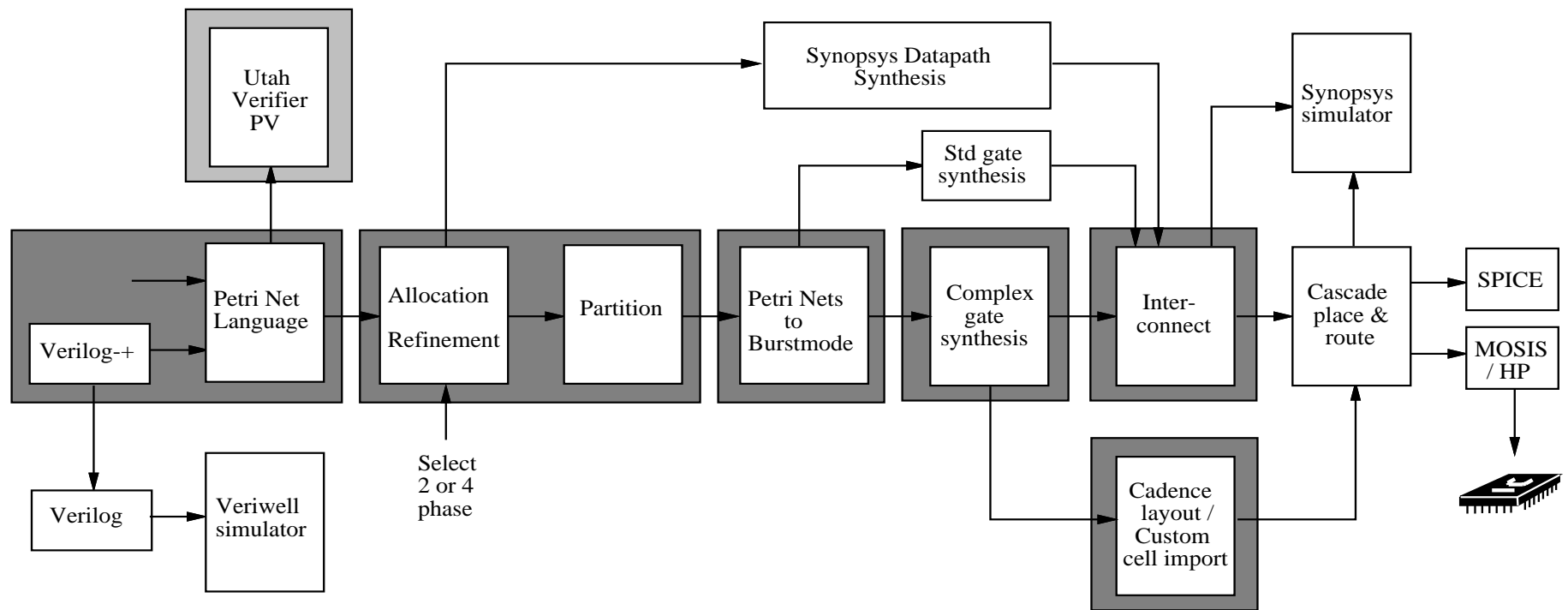


ACK

An Asynchronous Design Framework

- **High level synthesis**
 - Capture system structure rather than individual control structure
 - Allow system validation using verification and simulation
 - Generates both control and datapath
 - Support both two and four phase implementations
- **Synthesis flow similar to synchronous**
 - Leverage off of synchronous research
 - Allow use of standard tools
- **Target asynchronous FSMs** for efficient controllers
- **Complex gate realizations** for high performance

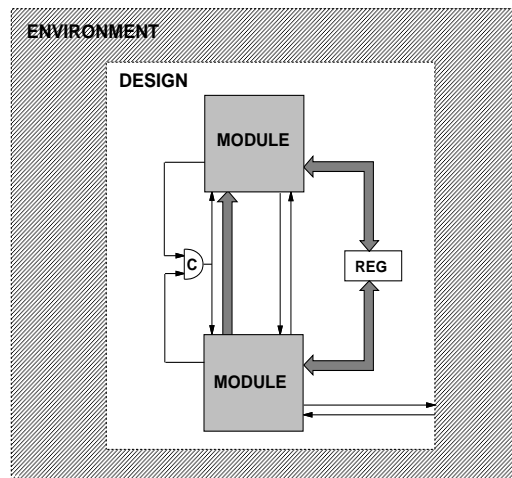
ACK - System Flow



Design Structure

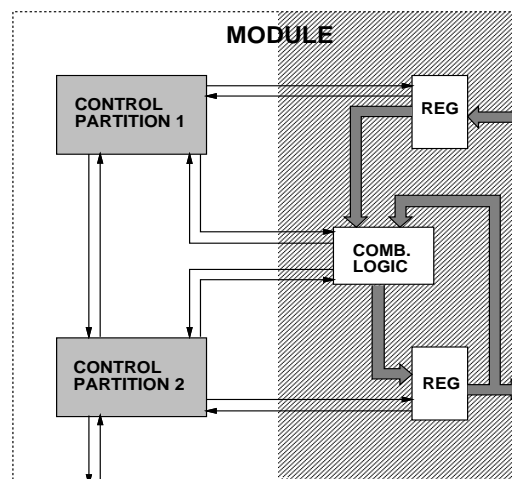
- **System:**

- Concurrent modules sharing registers, channels, event wires



- **Module:**

- sequential machine with limited fork-join concurrency
- local datapath resources and partitioned control structure



Behavioral Specification and Translation

```

Module Factorial;

reg [7:0] a, n;
channel? [7:0] nchan;
channel! [7:0] reschan;
event START;

always
begin
  @START;
  forever
  begin
    nchan?n;
    a = 1;
    while (n != 0)
    begin
      a = a * n;
      n = n - 1;
    end
    reschan!a;
  end
end
endmodule

```

Initial Verilog+ specification

```

Module Factorial

Event START?? : bit;
Variable a, n : array [7:0] of bit;
Channel nchan?, reschan! : array [7:0] of bit;

Behavior

<always> <= START?? -> <forever>;

<forever> <= nchan?n
  -> a = 1
  -> <while>;

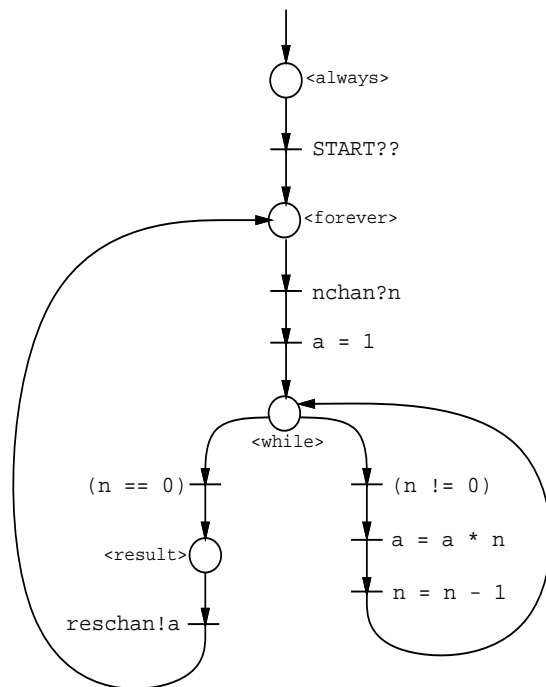
<while> <= (n == 0)
  -> <result>
  |(n != 0)
  -> a = a * n
  -> n = n - 1
  -> <while>;

<result> <= reschan!a
  -> <forever>

End

```

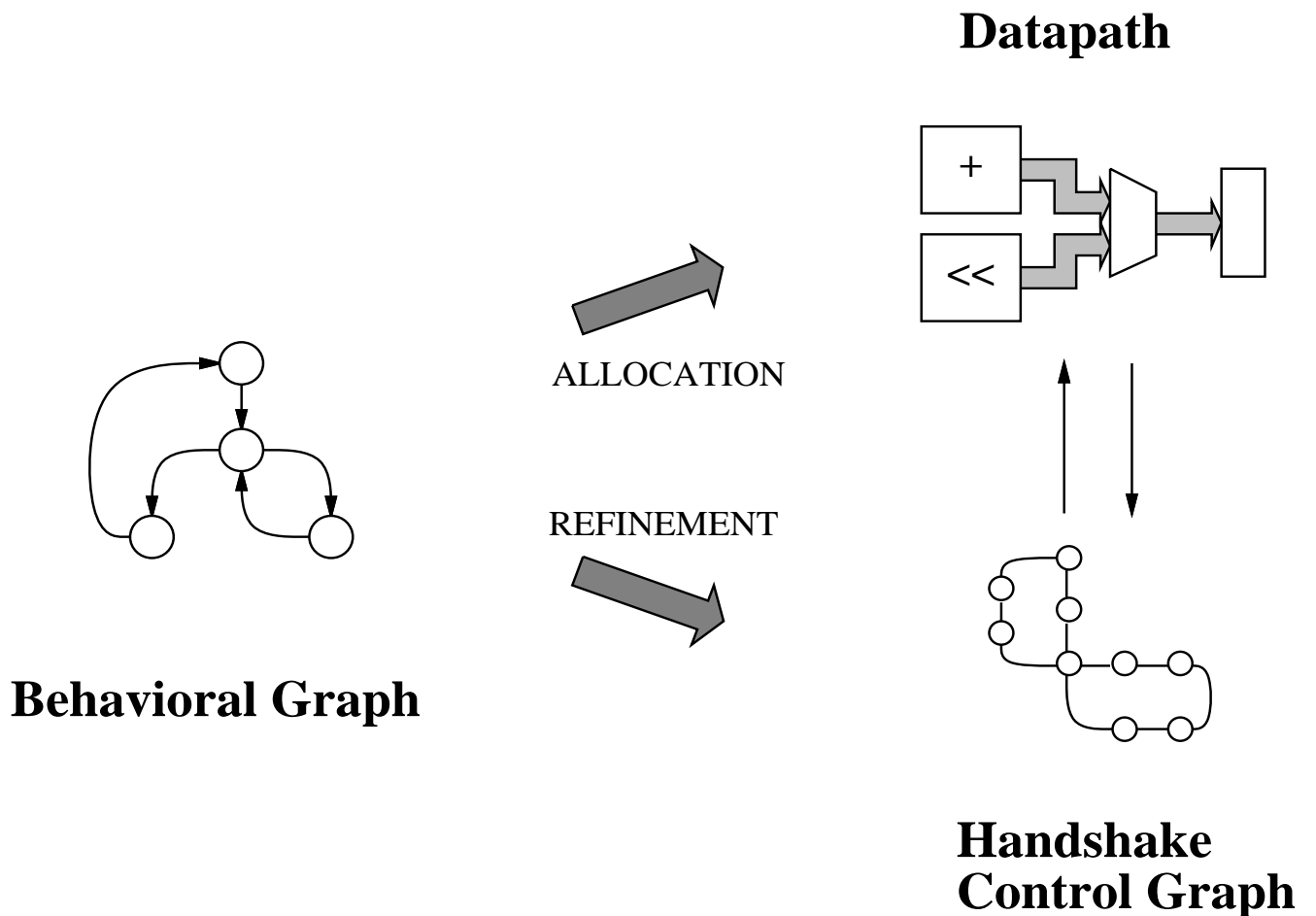
Translated HOP specification



Petri Net graph representation

Allocation and Refinement

- Allocate datapath resources
- Generate control graph acting on datapath



Control Graph Synthesis

- **Graph partitioning**

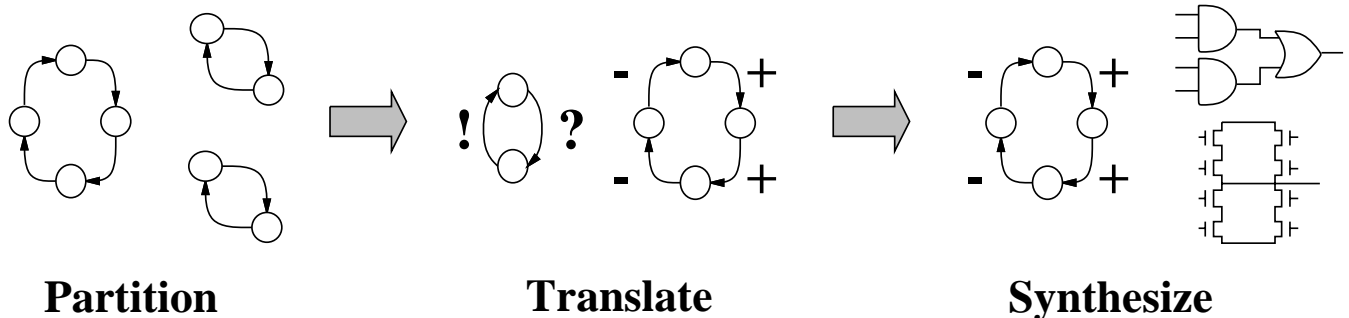
- If too large to synthesize
- To enhance area/performance

- **Graph translation**

- From abstract event based graph to level signal based FSM

- **FSM synthesis**

- To std. gate or complex gate realizations



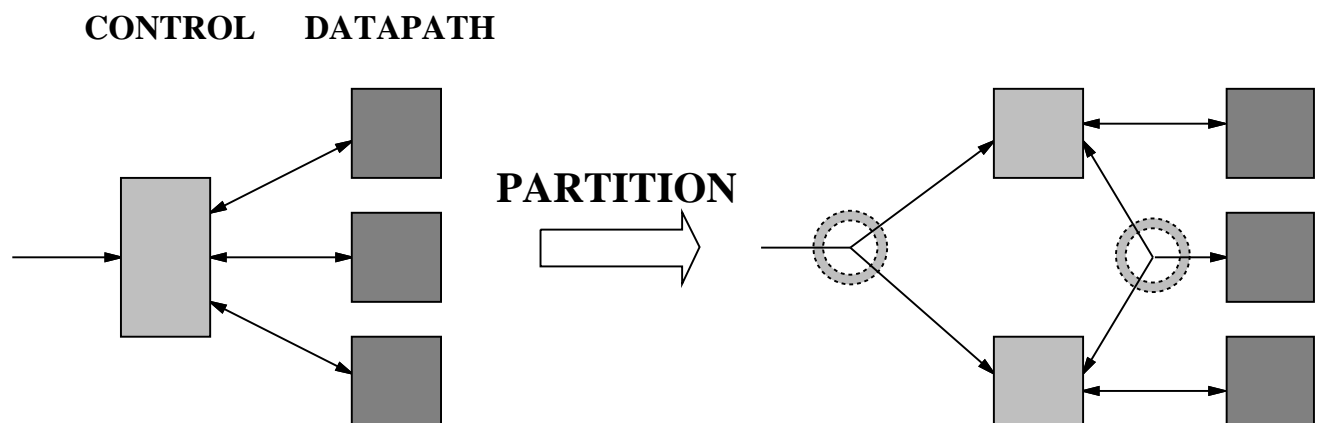
Control Graph Partitioning

- **Why partition?**

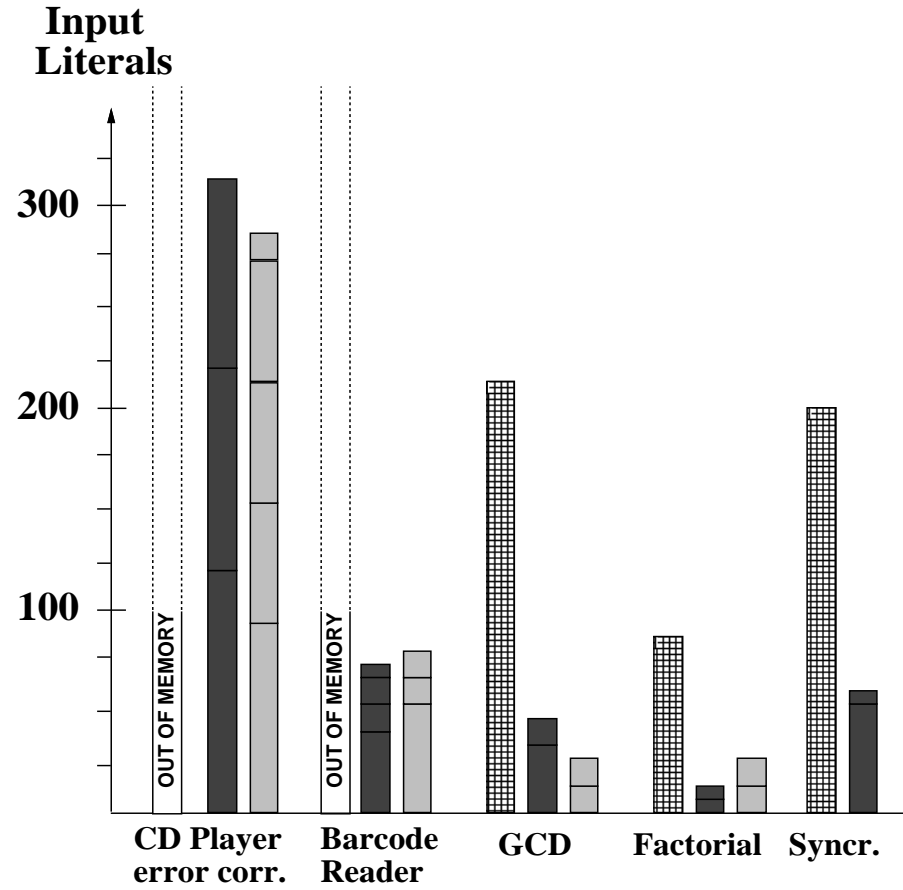
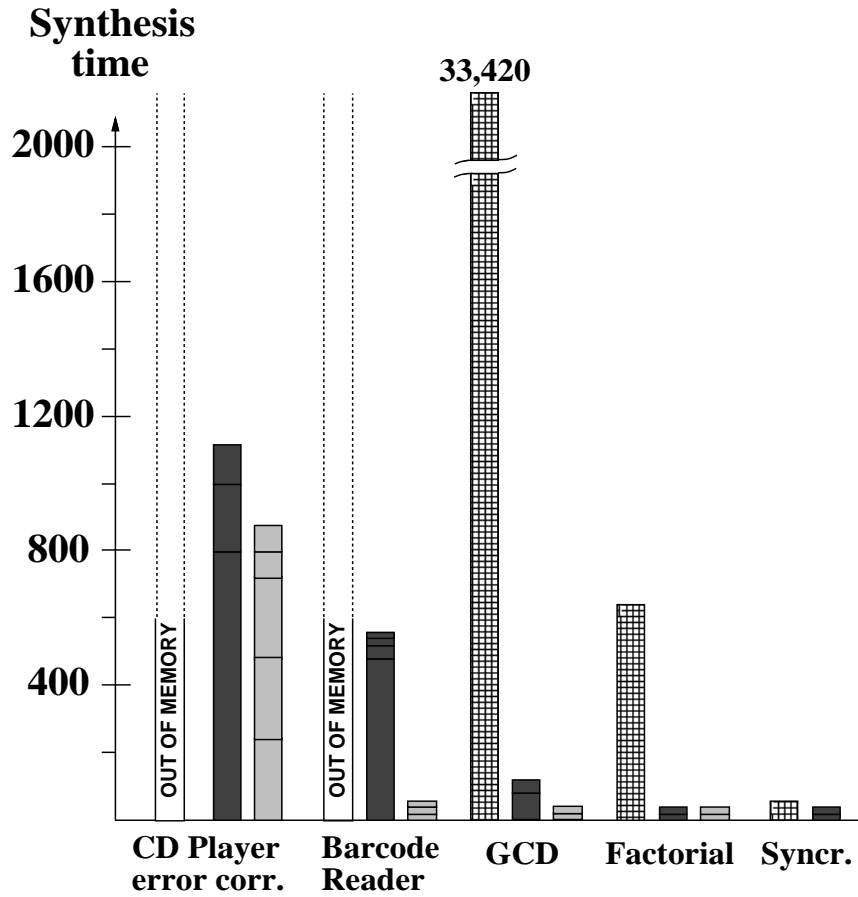
- Controller often large, FSM synthesis complex
- Exploit spatial and temporal locality
 - Monolithic controller
 - Large propagation delay
 - Long handshake wires

- **Why assisted partitioning?**

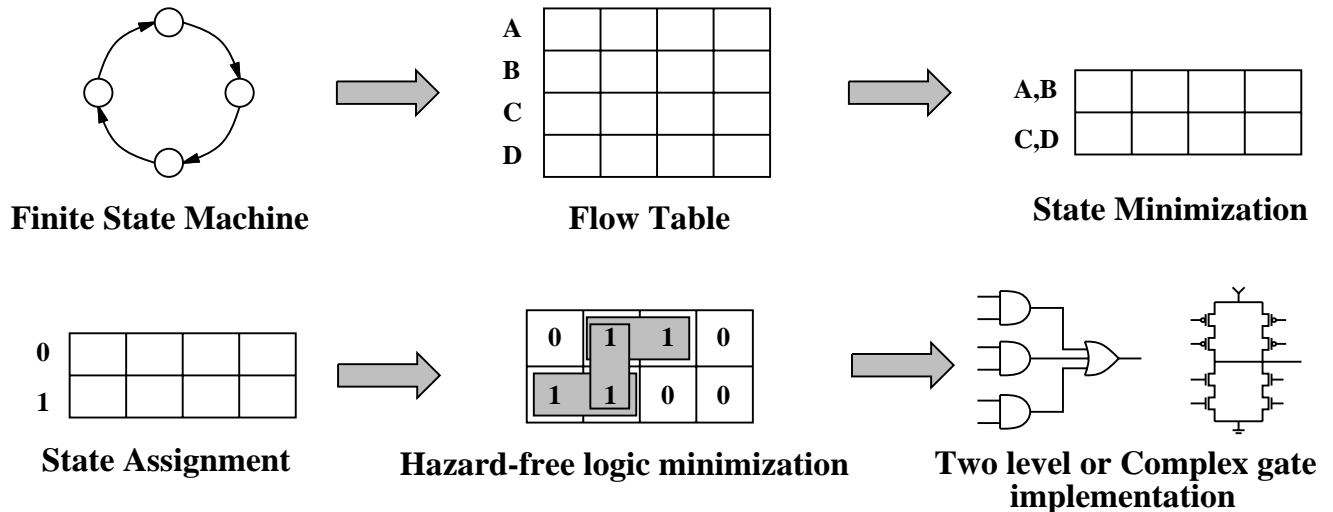
- Partition procedure must handle
 - Control flow between partitions
 - Complex signal sharing arrangements
- Large decision space
 - # shared signals
 - # shared datapath elements
 - Subgraph iteration frequency



PARTITIONING RESULTS



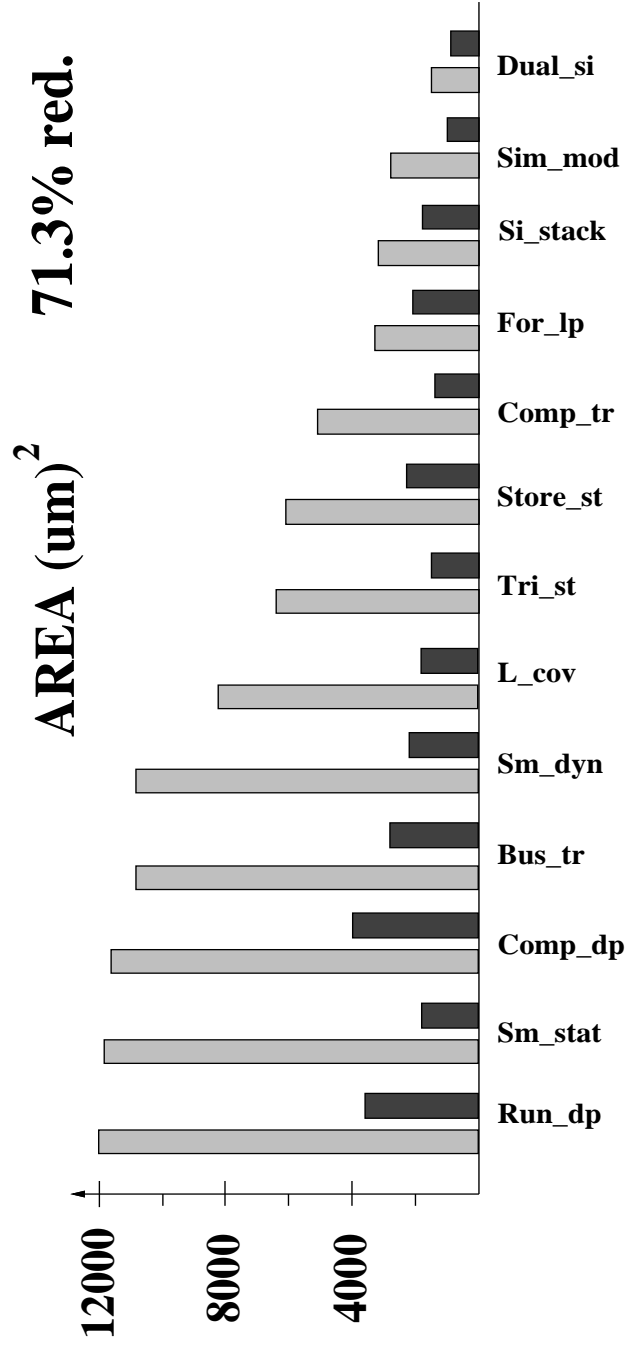
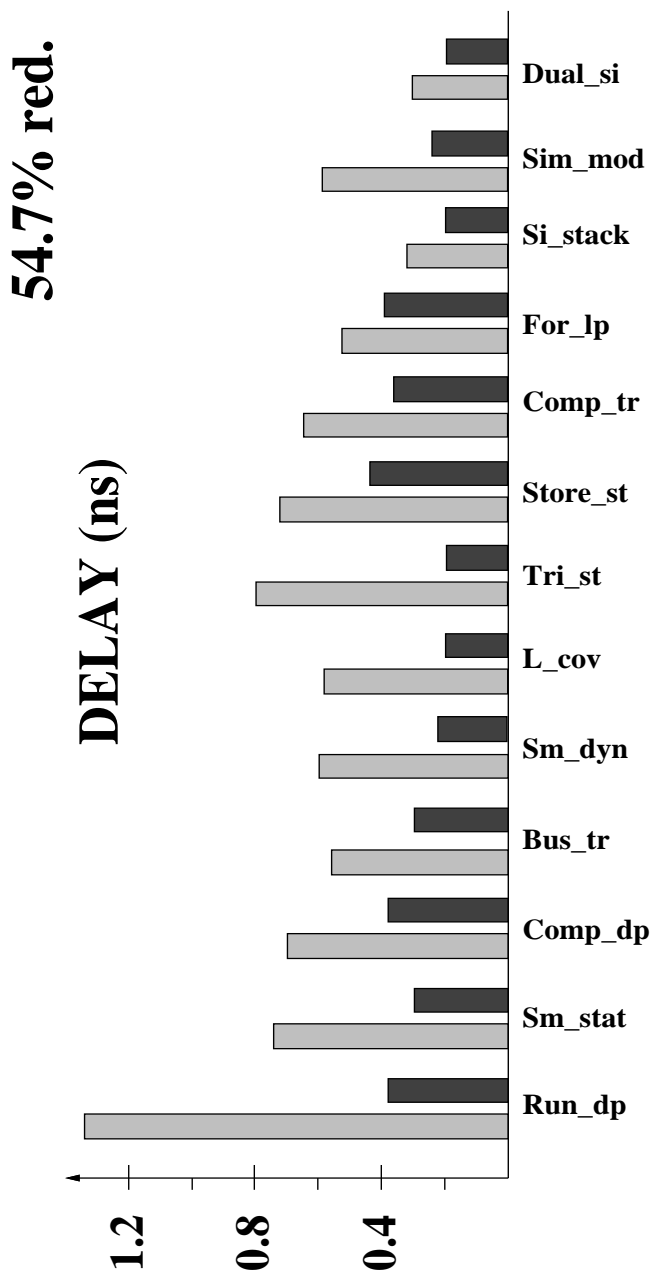
State Machine Synthesis



Gate/Transistor Level Realizations

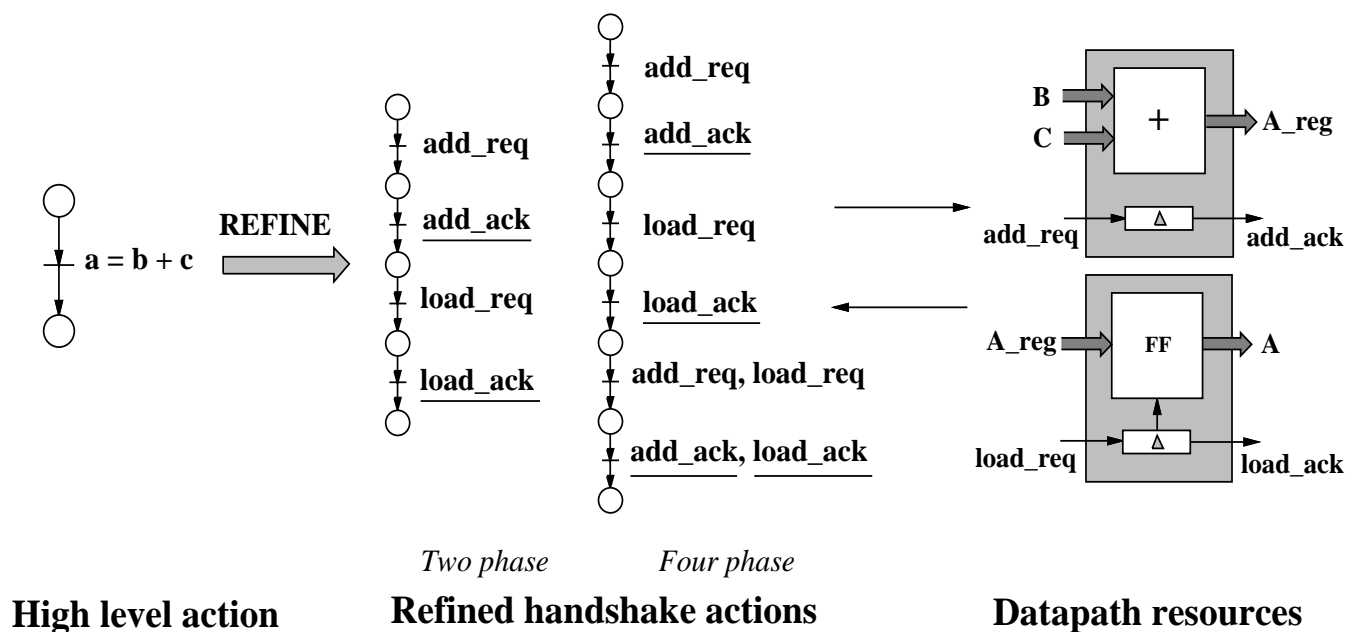
- **Two level Sum Of Products (AND/OR)**
 - **Standard complex gates**
 - **Customized complex gates**
 - Not dependent on restricted libraries
 - Advantage as feature sizes decrease
 - Transistor sizing
 - **SOP/SOP form of complex gate**
 - Advantages in static hazard behavior
 - Multilevel approach deals with dynamic hazards
- Reduce constraints in hazard-free synthesis

STD GATE vs COMPLEX GATE



Allocation and Refinement - revisited

- **Allocate datapath resources** with respect to chosen handshake protocol
 - Self-timed elements
 - Currently use matching delay but other realizations possible
 - Still avg. temp delay
- **Generate control graph** acting on allocated resources
 - Bundled data communication



Datapath Synthesis

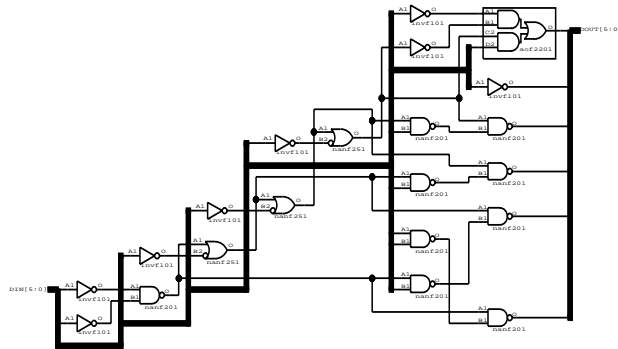
- Generate and synthesize VHDL for arithmetic functions and muxes

```

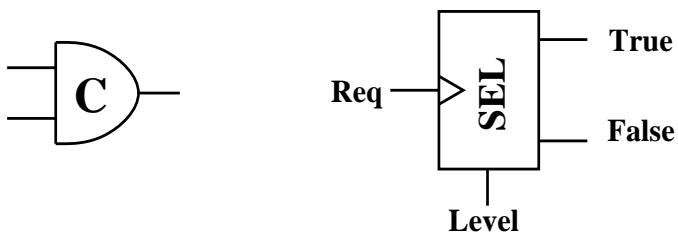
entity DEC is port(
  signal din : in vlbit_ld(5 downto 0);
  signal dout : out vlbit_ld(5 downto 0)
);
end DEC;

architecture DECbeh of DEC is
begin
process(din)
begin
  dout <= din -1;
end process;
end DECbeh;

```



- Std components for sync.channels (C-elements) and data dependent choices (Select-elements)

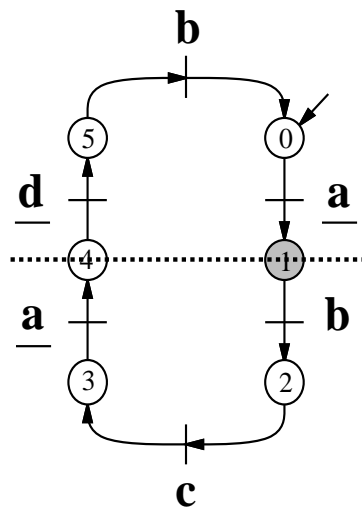


Control Graph Partitioning - revisited

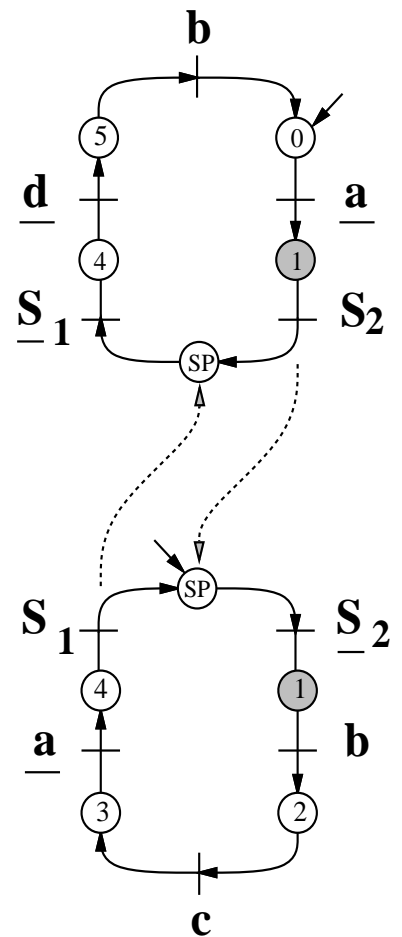
- **Partitioning procedure must:**

- Handle control flow between partitions

Solution: Split sequential controller and introduce handshaking to solve control flow



PARTITION



a and b shared!

- Handle signal sharing between partitions of an incompletely specified machine

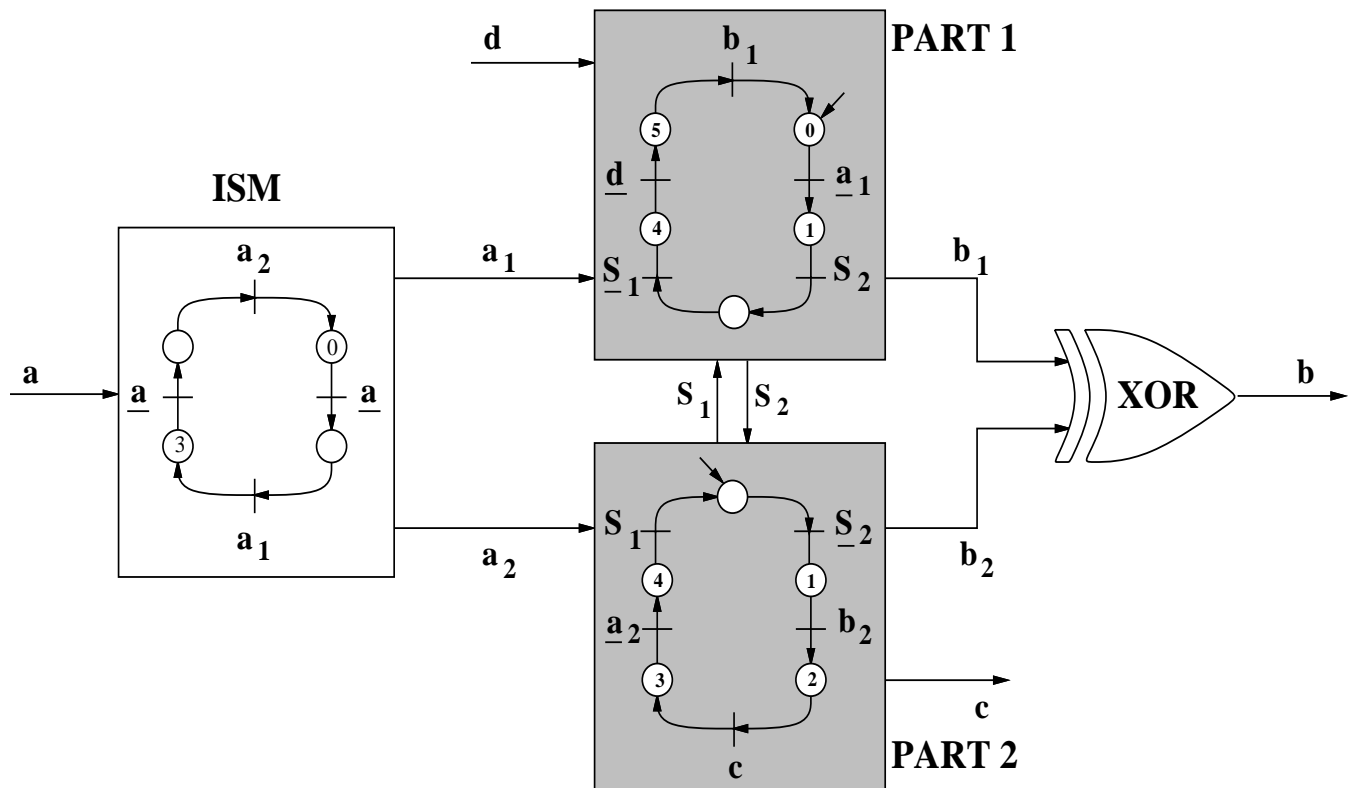
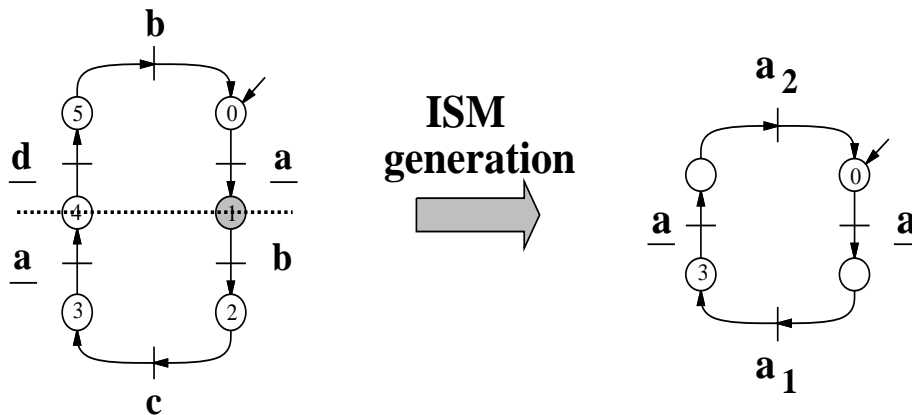
Partitioning - General Approach

- Handle I/O signal sharings between partitions

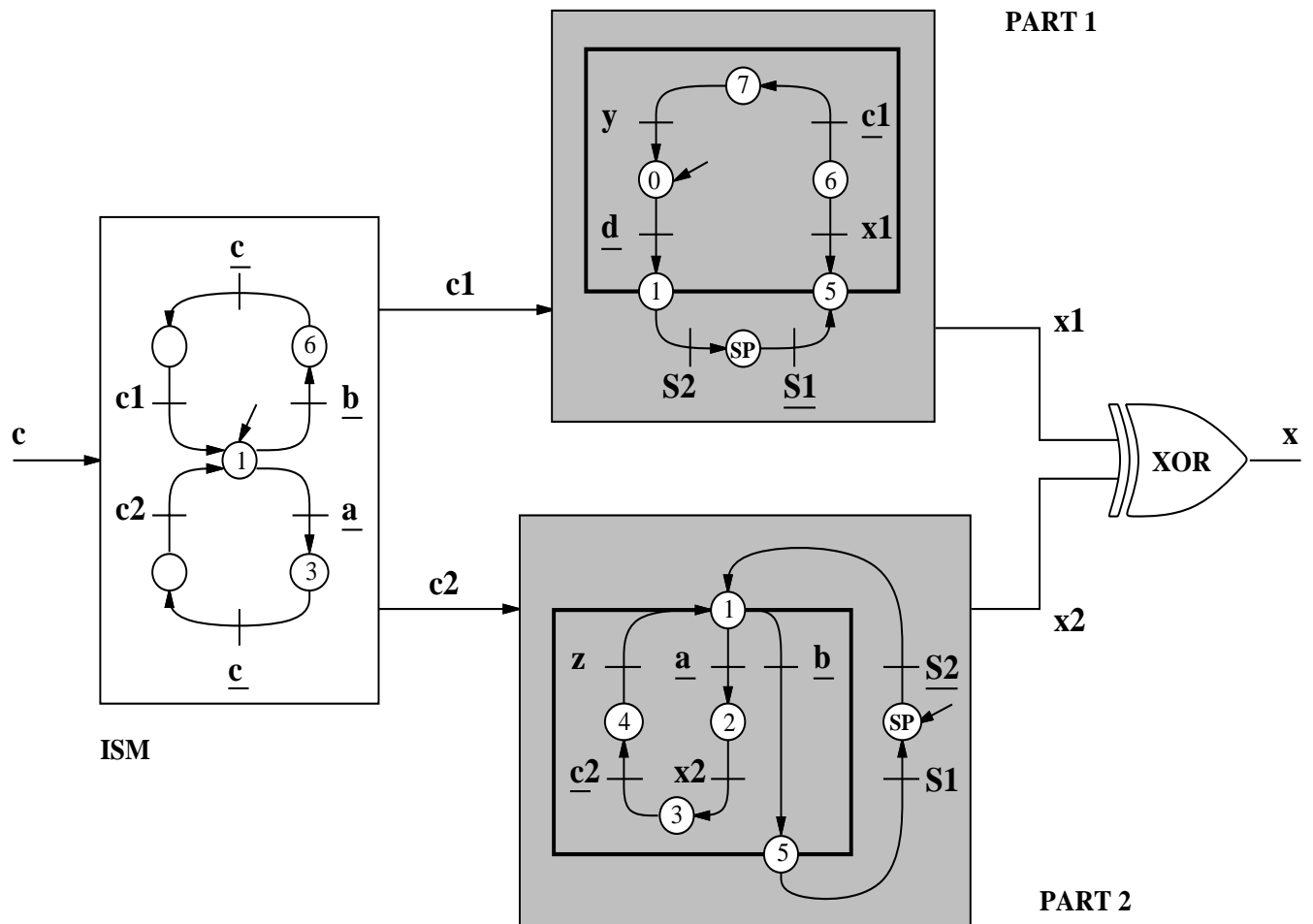
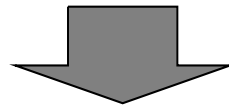
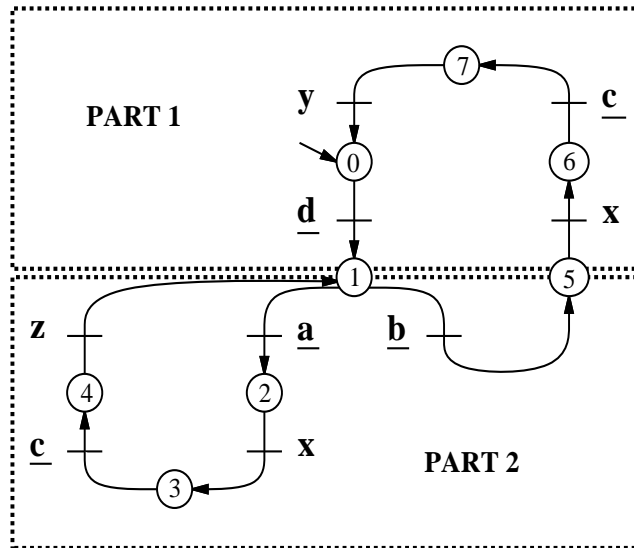
Solution:

Outputs: Output state machine to merge output signals (XOR/OR)

Inputs: Input state machine to distribute input signals to right controller at right time

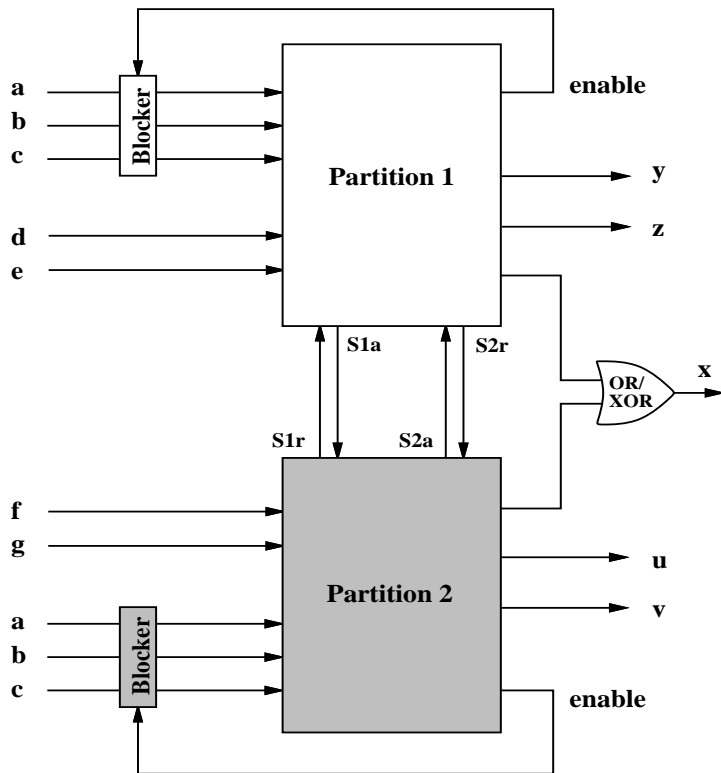


Partitioning - Example



Partitioning - Time Critical Signals

- Use propagation gates - "blockers"
 - Depends only local control flow signals
- Constant delay for input sharing
 - Single AND gate when 4 phase
 - Single MUX when 2 phase

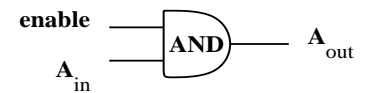


(a) Partition 1 is active. The propagation gates of partition 2 are disabled.

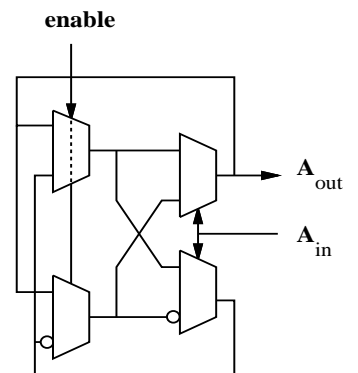
= passive/disabled
 = active/enabled

Propagation gates

"Blockers"



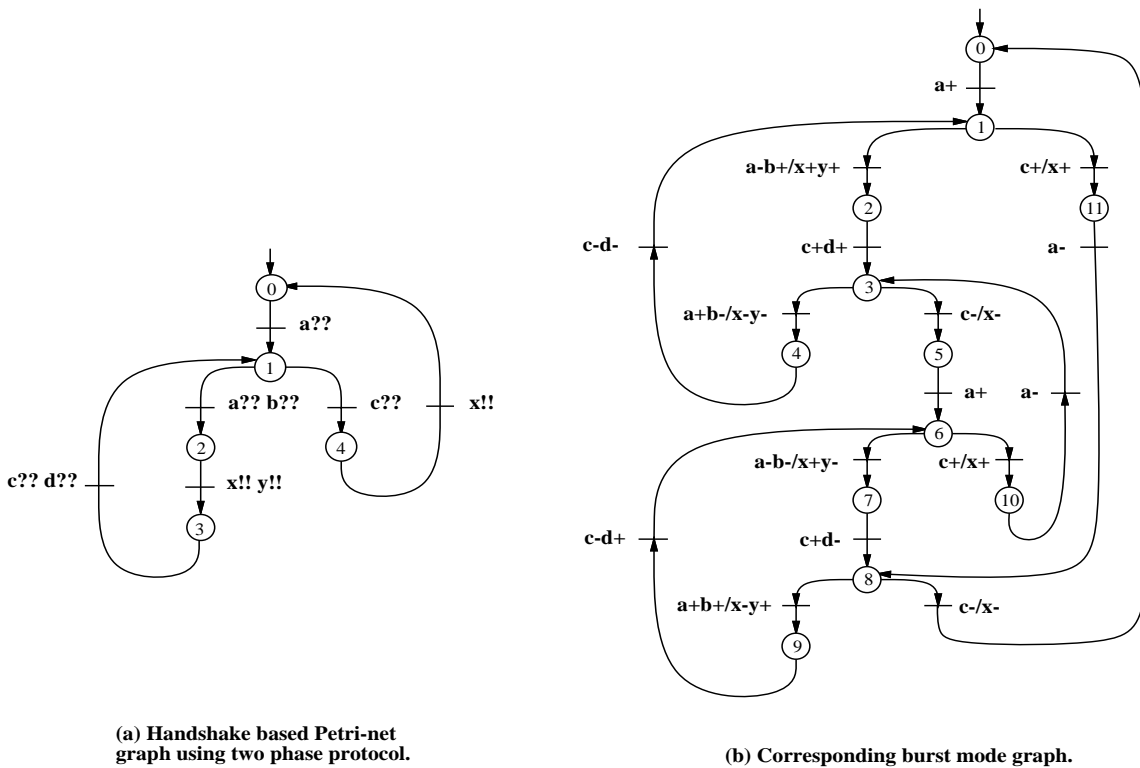
4 phase



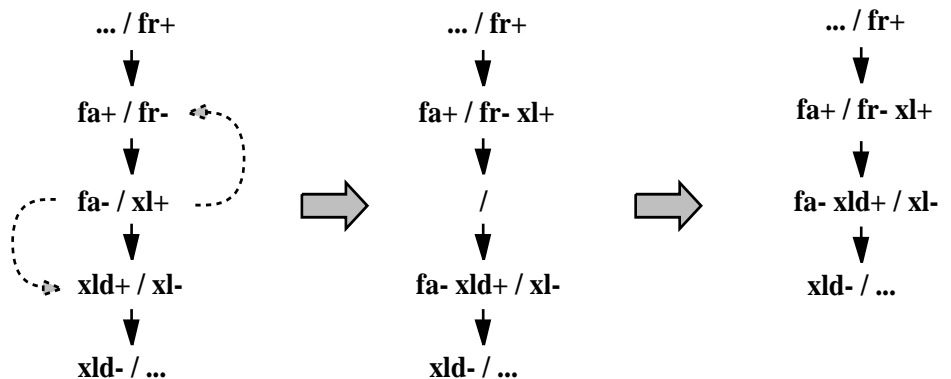
2 phase

Burstmode FSM Generation - revisited

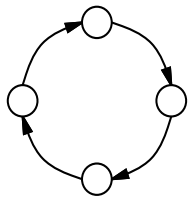
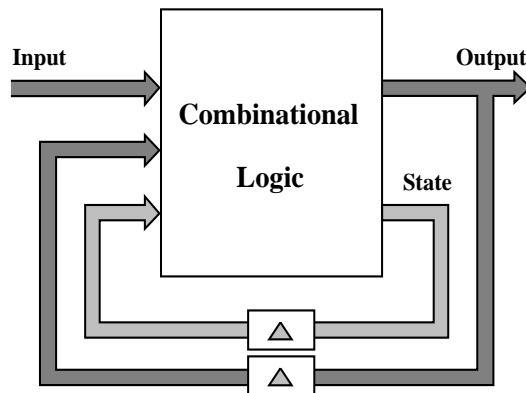
- Translate event based Petri-net to Burstmode machine with polarized transitions
- Event based Petri-nets also used as compact low level specification format



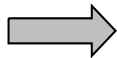
- 4 phase protocol introduce overhead - reshuffle



Burstmode FSM Synthesis - revisited

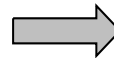


Burst Mode Machine



A				
B				
C				
D				

Flow Table

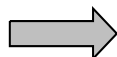


A,B			
C,D			

State Minimization

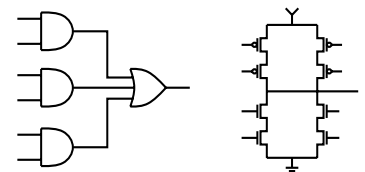
0				
1				

State Assignment



0	1	1	0
1	1	0	0

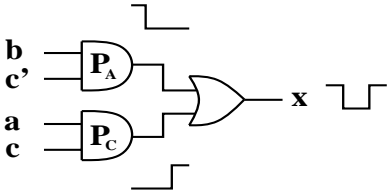
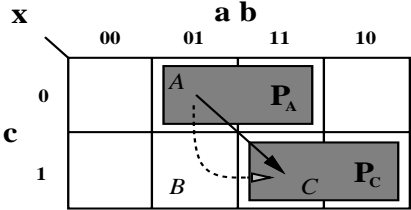
Hazard-free logic minimization



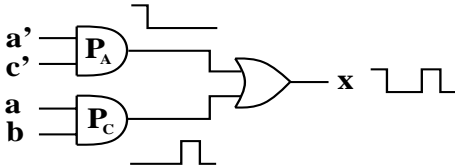
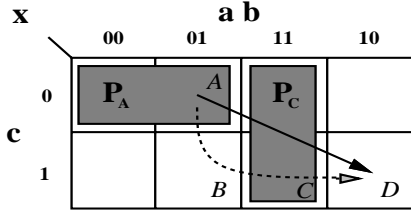
Two level or Complex gate implementation

Background - Hazards

- Function hazards - property of specification

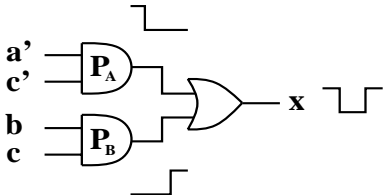
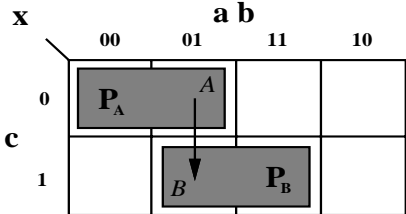


(a) Static 1 -> 0 -> 1 function hazard

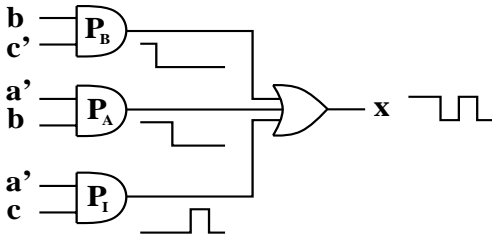
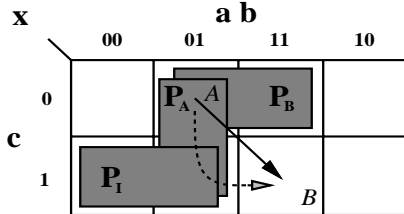


(a) Dynamic 1 -> 0 -> 1 -> 0 function hazard

- Logic hazards - property of gate realization



(a) Static 1 -> 0 -> 1 logic hazard

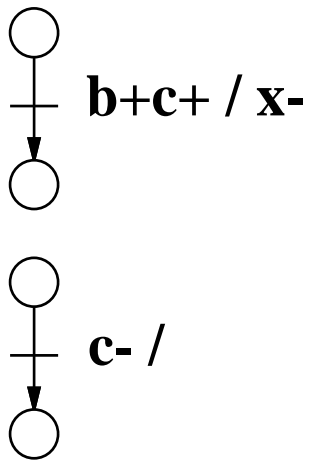


(a) Dynamic 1 -> 0 -> 1 -> 0 logic hazard

Background - Burstmode transition

Static transition:

Req-cube: ab'

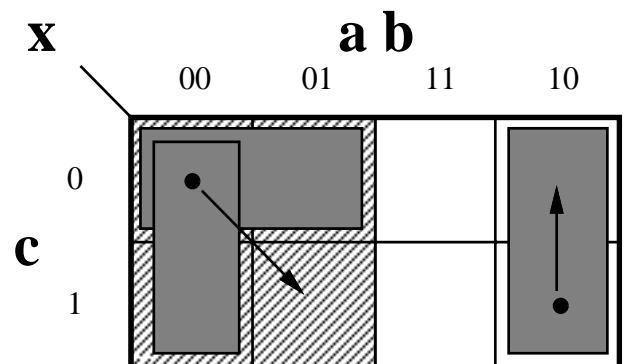


Burst mode spec

Dynamic transition:

Req-cubes: $a'c'$
 $a'b'$

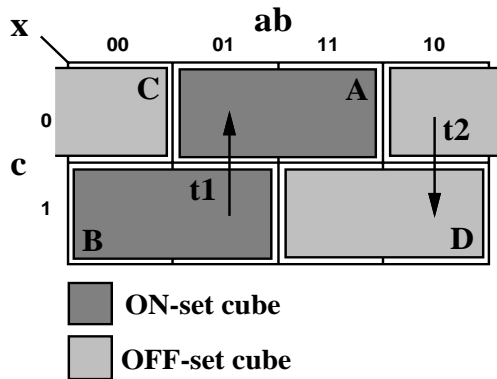
Priv-cube: a'



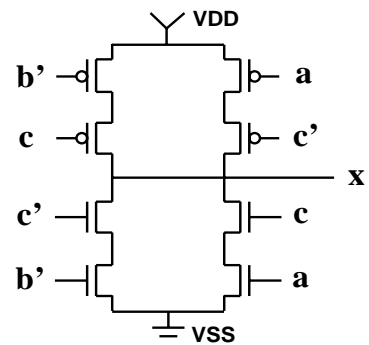
Karnaugh map

Static Hazard Properties

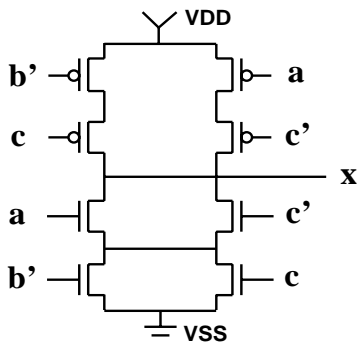
- SOP/SOP complex gate is static hazard free at the output



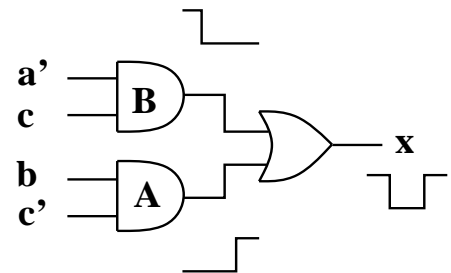
Karnaugh map



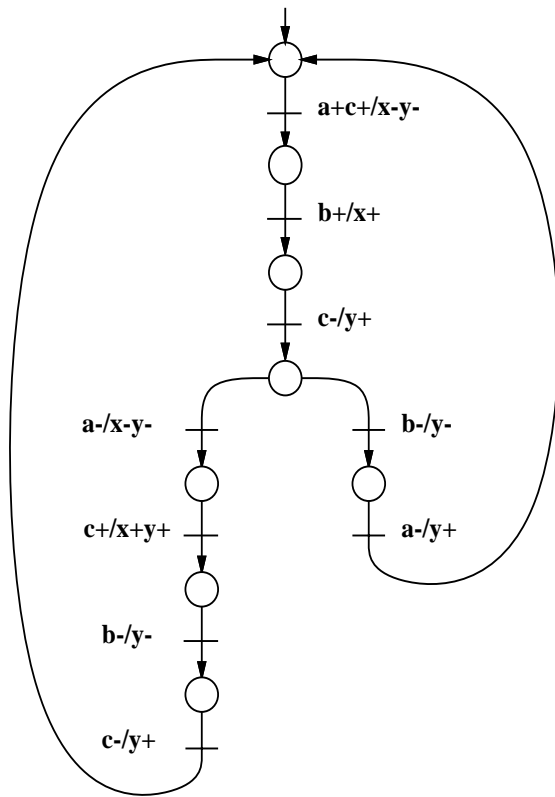
Hazard free SOP/SOP complex gate



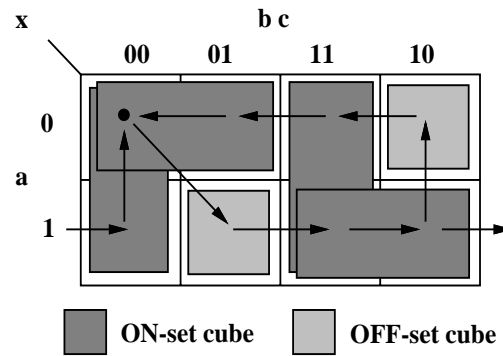
Hazardous SOP/POS complex gate



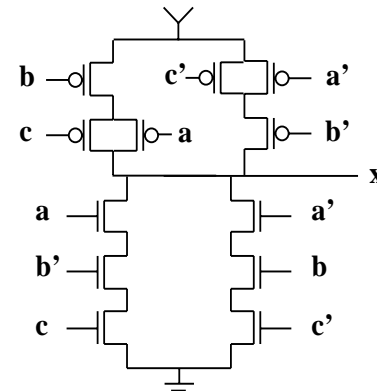
Hazardous simple gate implementation



(a) Burst mode specification

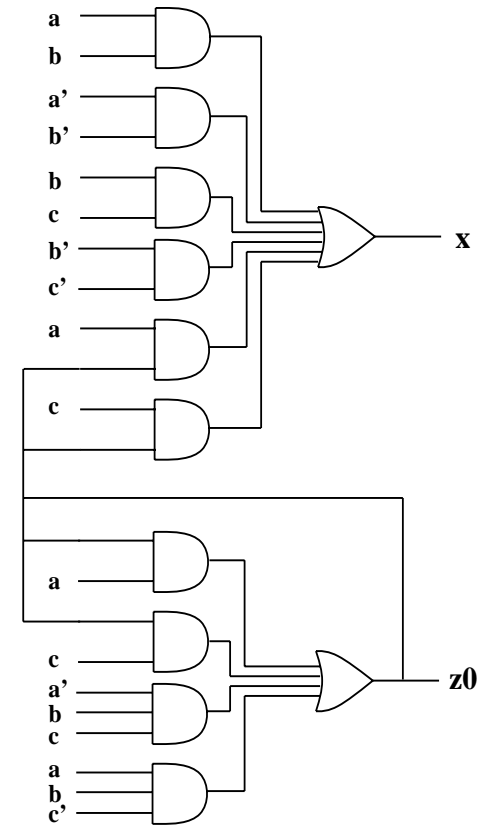


(b) Karnaugh map for output x .



Transistors = 12

(c) Custom single complex gate.

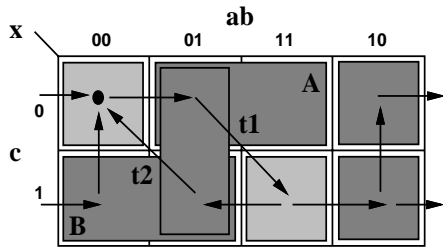
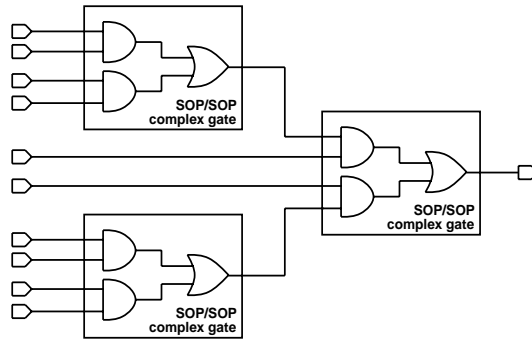


Transistors = 64

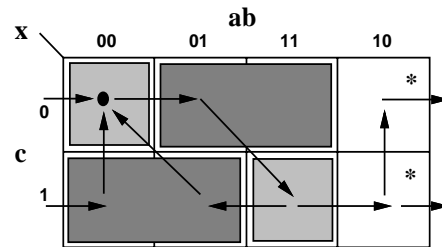
(d) Standard AND/OR gate implementation.

Dynamic Hazards

- Multilevel SOP/SOP complex gate have solution to dynamic hazards

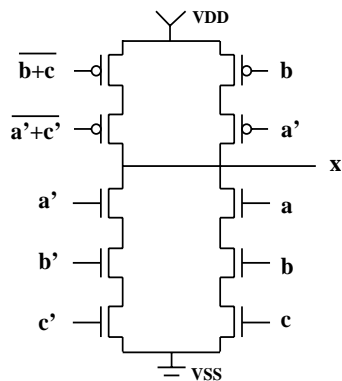


(a) Karnaugh map for output x



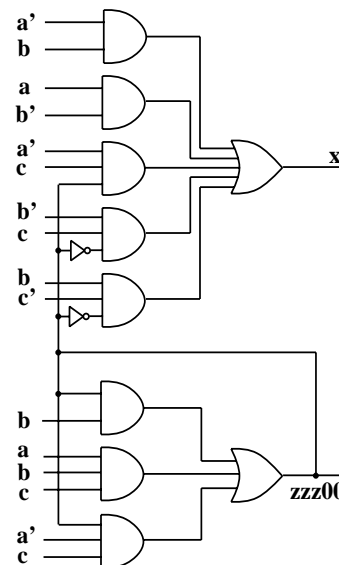
(b) Reduced problem to derive POS

ON-set req-cube
 OFF-set req-cube



of transistors: **18**

(c) Complex gate implementation



of transistors: **58**

(d) Circuit for x derived by 3D

Conclusions I - Contributions

What we have:

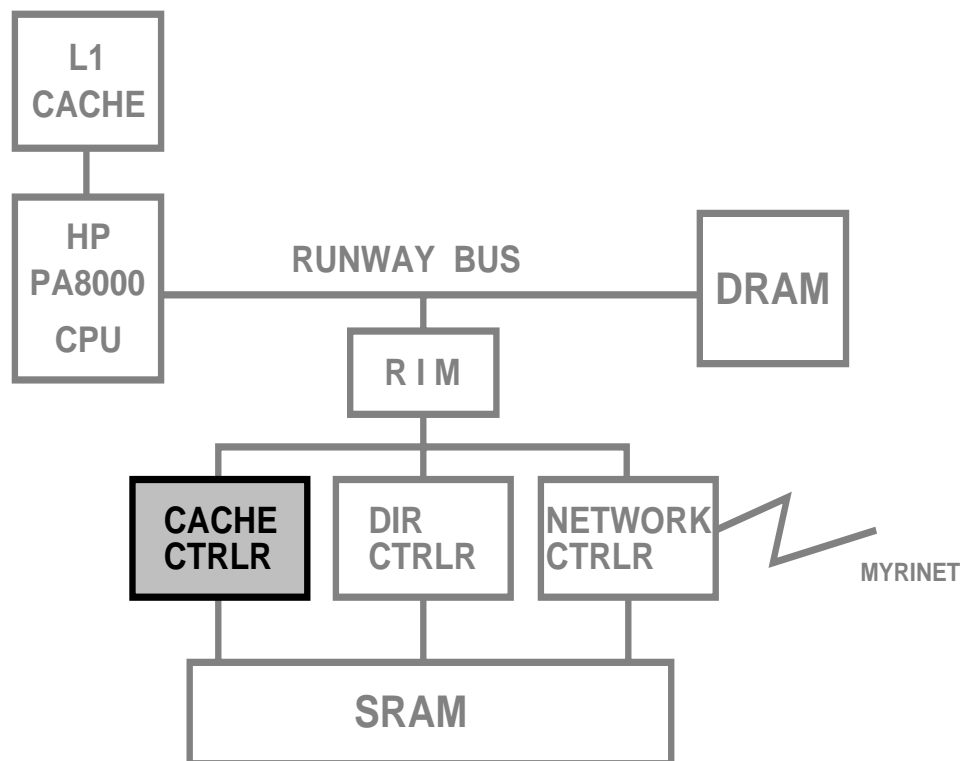
ACK is a framework that offers:

- **Behavioral description in standard language**
Allow system level description
- **High level synthesis targeting state machines**
Efficient individual controllers
- **Two and four phase implementation**
Allow easy comparison of efficiency
- **Partitioning of incompletely specified machines**
Important for synth, performance and area
- **Hazardfree logic synthesis to complex gates**
Reduce individual controller latency and area

Future work - Driving Design Example

We have a nice base for experimentation

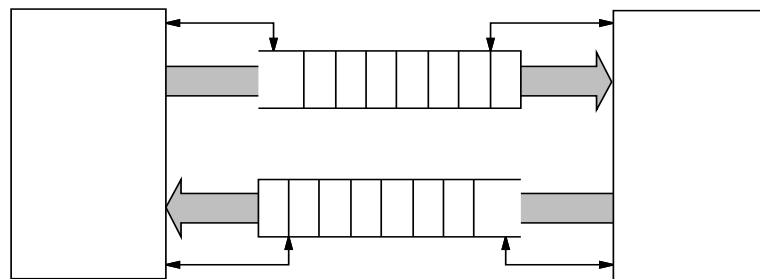
- **Real-life designs** of Avalanche MPC Widget to drive further tool development
- First design target is cache controller
- Great opportunity to look into sync async interfacing



Future work - Architectural Issues

System and Architecture level performance issues

- What can be done at Protocol level to improve module interaction and concurrency?
 - Concurrency - buffered channels

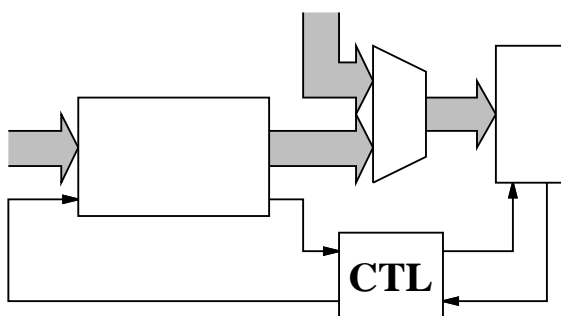


Buffered Channels

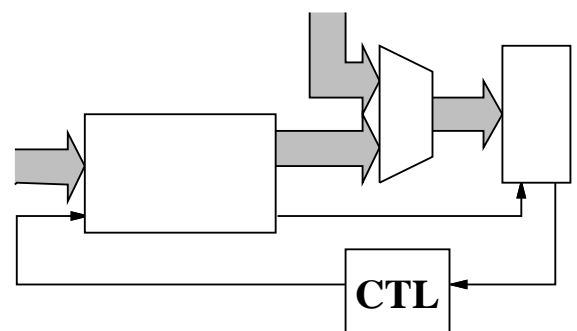
- What can be done at Architecture level to reduce control overhead?

Hide overhead in concurrent actions

Handshake *flow through*



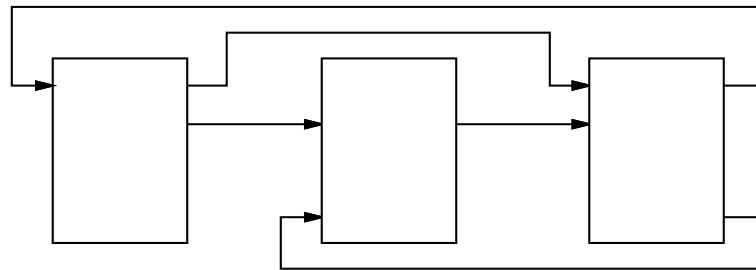
Hide control overhead



Request flow through

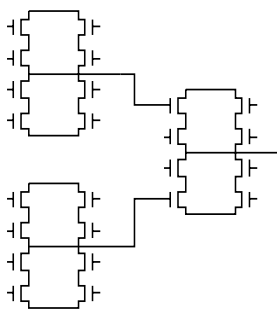
Future work - Controller Issues

- **Timing-analysis of interacting controllers**
 - Complex when multiple stages

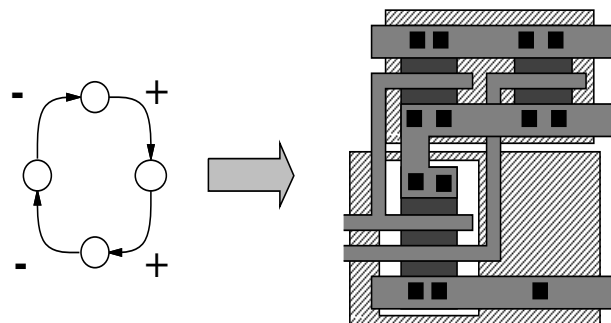


Interacting Controllers

- **Further work on complex gates**
 - Transistor sizing
 - Automate generation and layout



Transistor sizing



**Automate complex gate
synthesis and layout**