codesign environment

Schedule

- 1. Introduction
- 2. System Modeling Language: System C *
- 3. HW/SW Cosimulation *
- 4. C-based Design *
- 5. Data-flow Model and SW Synthesis
- 6. HW and Interface Synthesis
  (Midterm)
- 7. Models of Computation
- 8. Model based Design of Embedded SW
- 9. Design Space Exploration
  (Final Exam)
  (Term Project)

Reference

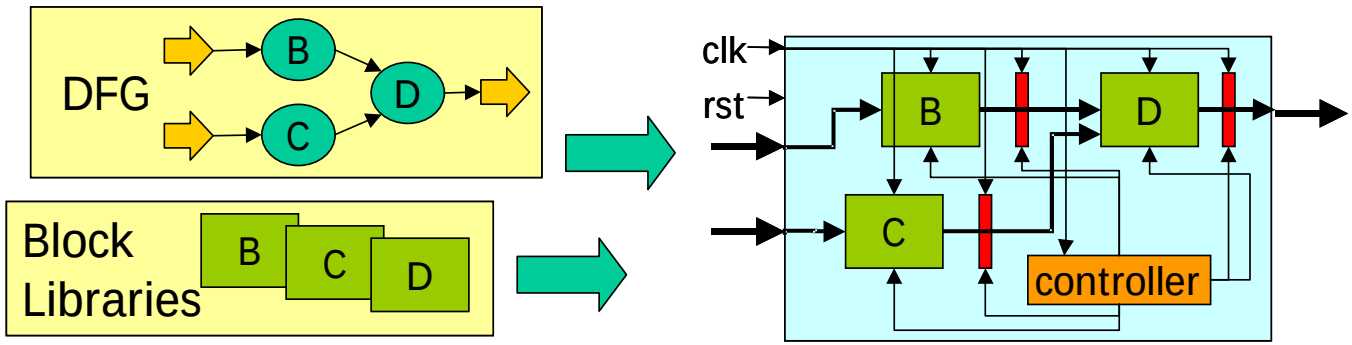
- **PeaCE Approach**
  - Hyunuk Jung, Kangnyoung Lee, and Soonhoi Ha, “Efficient Hardware Controller Synthesis for Synchronous Data Flow in System Level Design,” IEEE Transactions on Very Large Scale Integration(VLSI) Systems Vol. 10 pp 423-428 August 2002
  - Hyunuk Jung, Hoeseok Yang, and Soonhoi Ha, "Optimized RTL Code Generation from Coarse-Grain Dataflow Specification for Fast HW/SW Cosynthesis", Journal of VLSI Signal Processing (online published) 10, June. 2007

Contents

- **Introduction**
  - Higher-level design from dataflow model
  - PeaCE design flow & cosynthesis
- **Previous works**
- **Block Definition**
- **Controller Synthesis** → IEEE Transaction on VLSI Systems Vol.10, August 2002
- **Schedule-Based Design** → J. VLSI Signal Processing, 2007
- **FRDF Specification for more efficient HW implementation**
- **Conclusions**

Hardware Synthesis Problem

- Automatic Hardware Synthesis from Coarse-Grained Dataflow Specification in System Level Design
  - A node represents a coarse grain computation block such as FIR filter or DCT.
  - A node has complex properties such as data sample rates, I/O timings, data types, and its internal states.
  - A central controller should be generated automatically in order to control these complex coarse-grain HW library blocks and registers.

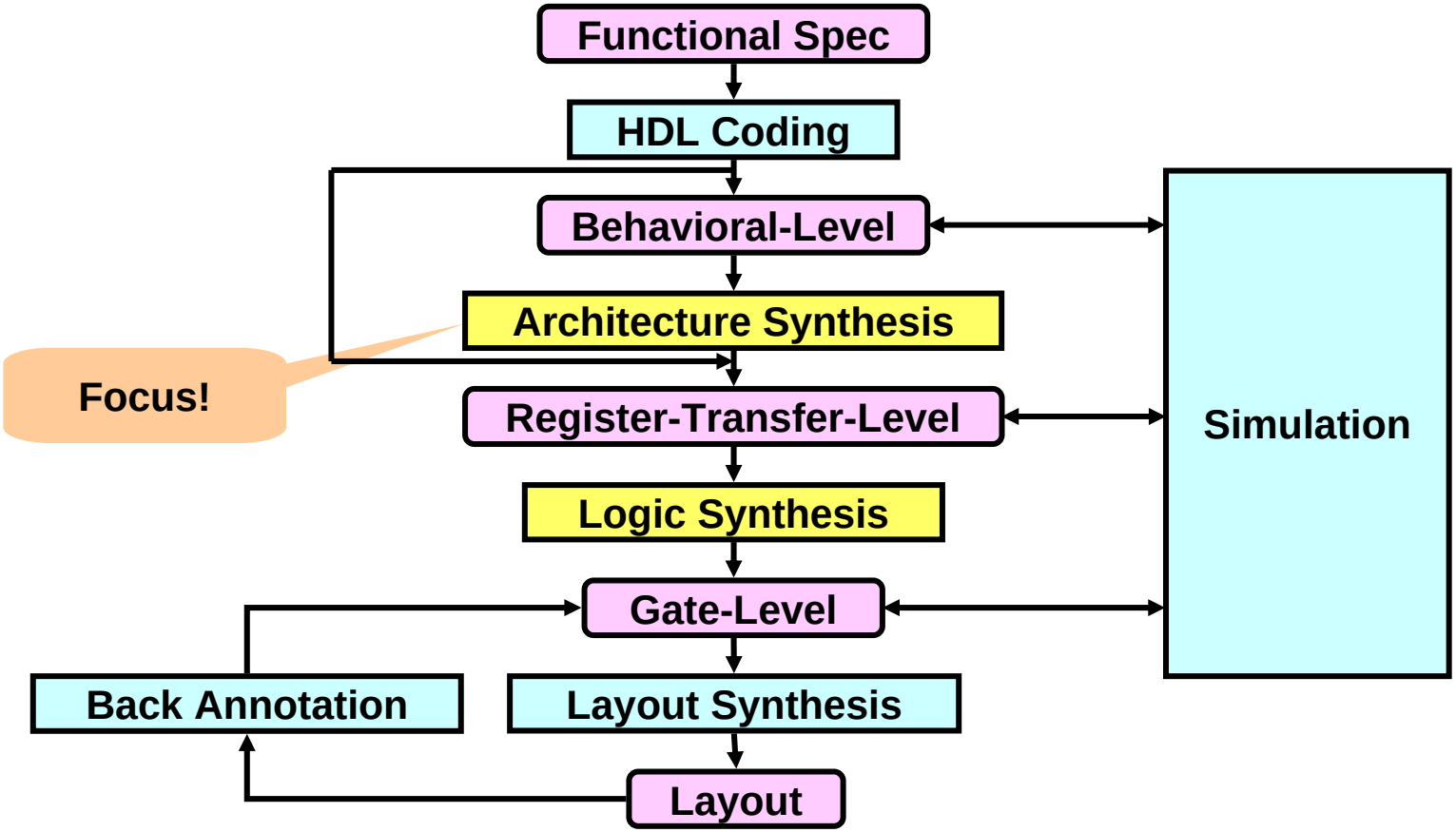


Design Size & Abstraction Level

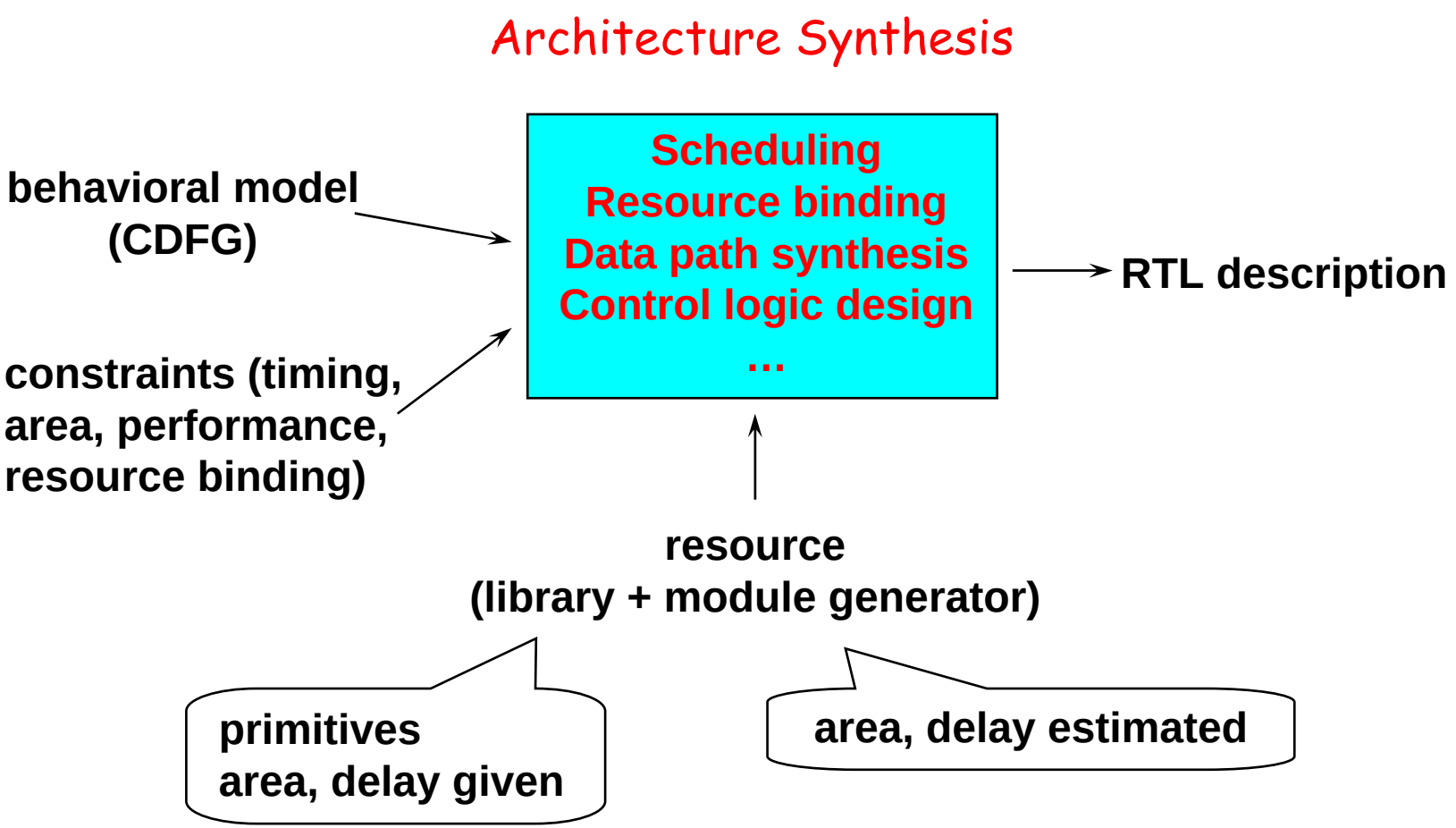
- 1970s, several hundred transistors
  - Transistor and gate level design
- 1980s
  - Register Transfer Level (RTL) design
  - Hardware Description Language (HDL)
- 1990s
  - High-level synthesis (or behavioral synthesis)
  - Behavioral HDL, imperative programming languages (C,C++)
- **Today, several million gates**
  - Exponential increase in transistor density
  - HW/SW Co-design (System Level Design)
  - We need higher-level tools for system design and hardware description

Conventional High-level Synthesis

- **Hardware synthesis**
  - Conventional architecture and logic synthesis techniques are used



Architecture Synthesis Problem



New Trend: C-based design

- **From C/C++ to Hardware**
  - Mentor Graphics(www.mentor.com): “Catapult C Cynthesis”
  - Forte design systems (www.forteds.com): “Cynthesizer”
  - Synfora(www.synfora.com): “PICO Express”
  - Y Explorations Inc.(www.yxi.com): “eXCite”
  - Celoxica(www.celoxica.com): “Agility Compiler”
    - *RTL level compiler*

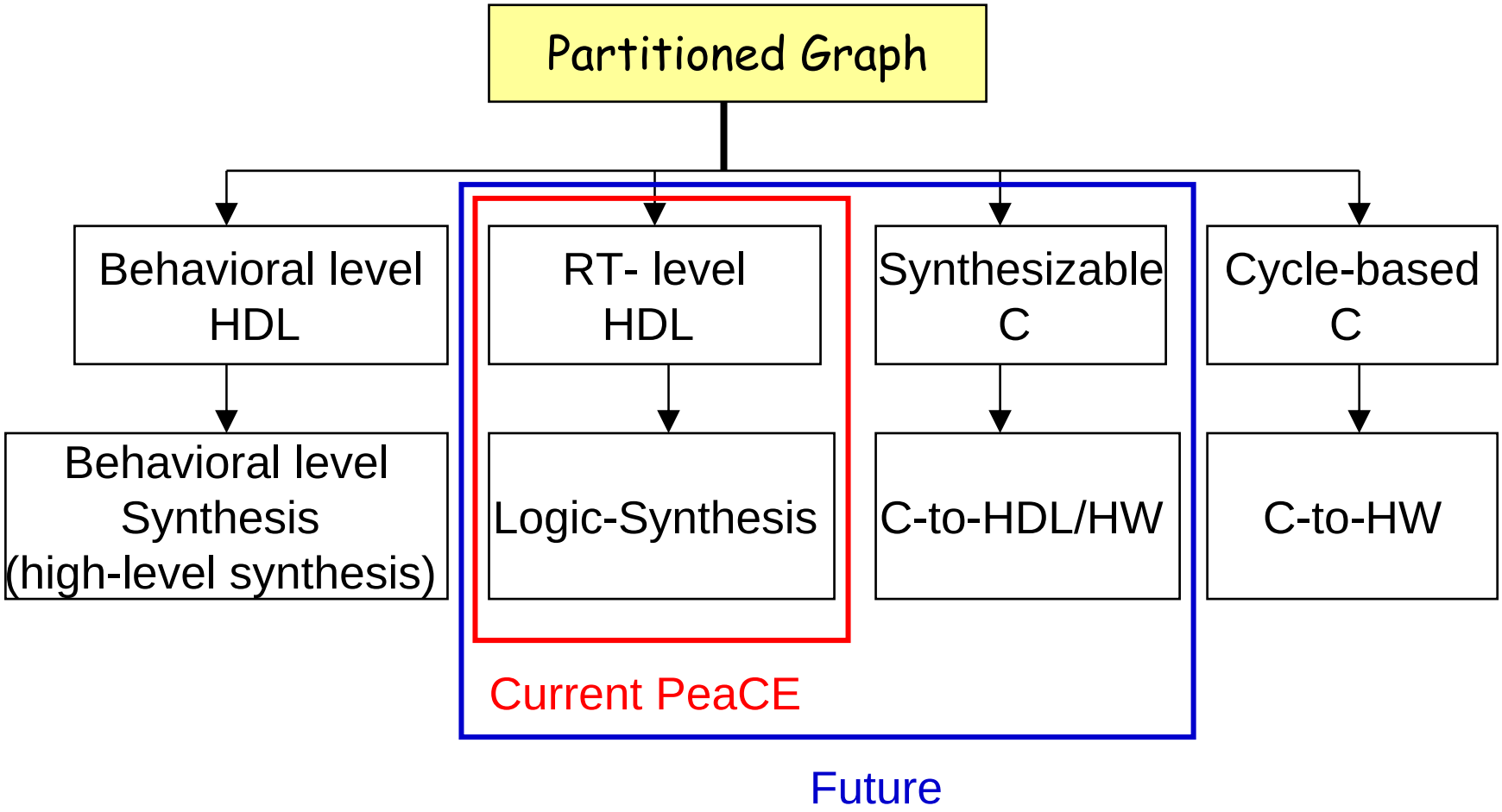
System-level HW Synthesis

- **Higher Level Hardware Synthesis**
  - Increasing need for a design methodology of higher abstraction level
  - Growing complexity, fast design turn-around time
  - Easy to modify and maintain

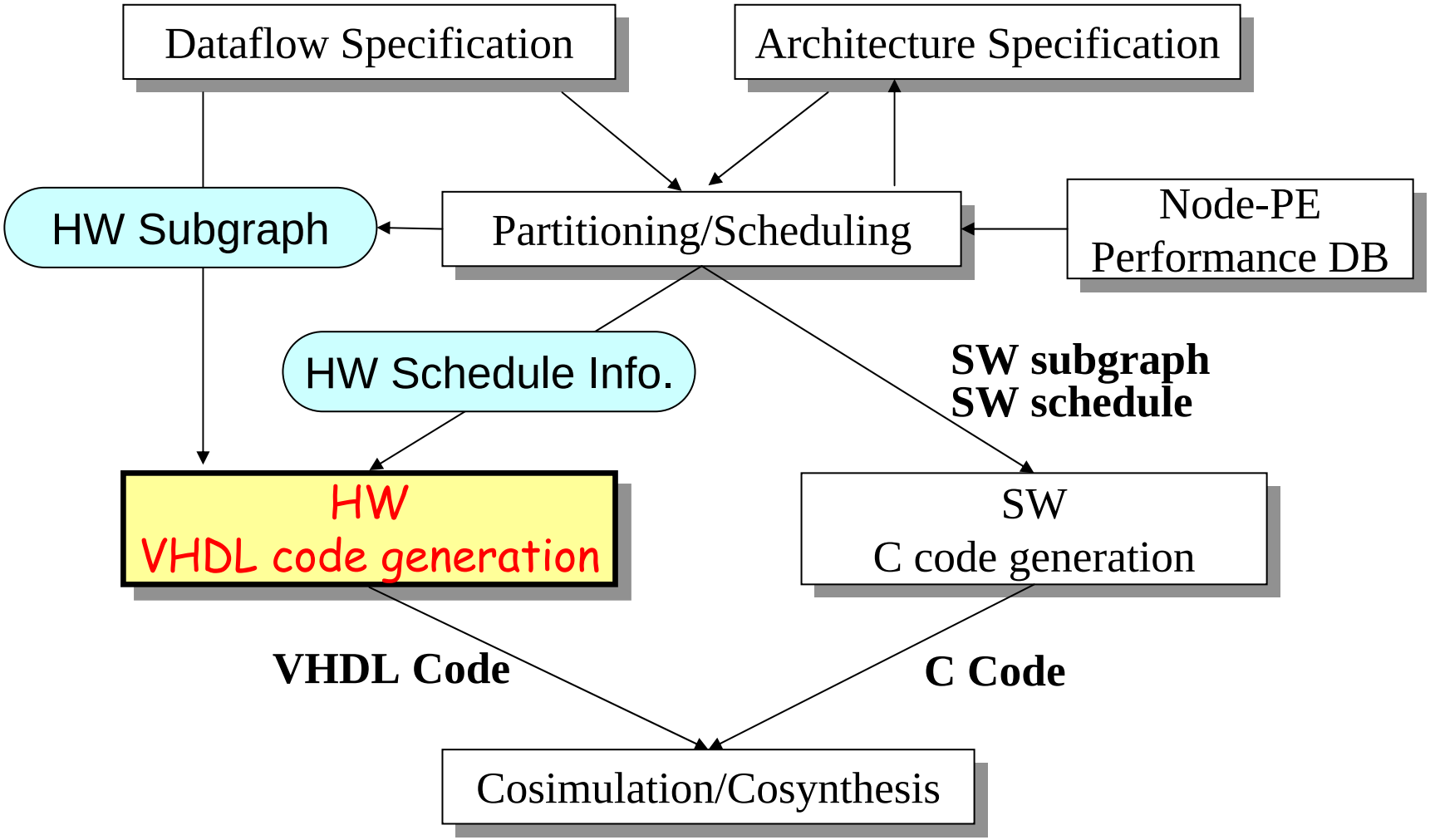
- **Automatic code generation from data flow graph**
  - SDF semantics should be preserved - “refinement”
  - The kernel code of a block is already optimized in the library.
  - Determine the schedule and resource allocation.
  - Controller is generated according to the scheduled sequence and the resource mapping

- **Fundamental Question**
  - Can we generate the HDL code with the synthesizable area and the similar performance as manually optimized code?

Hardware Synthesis Strategies

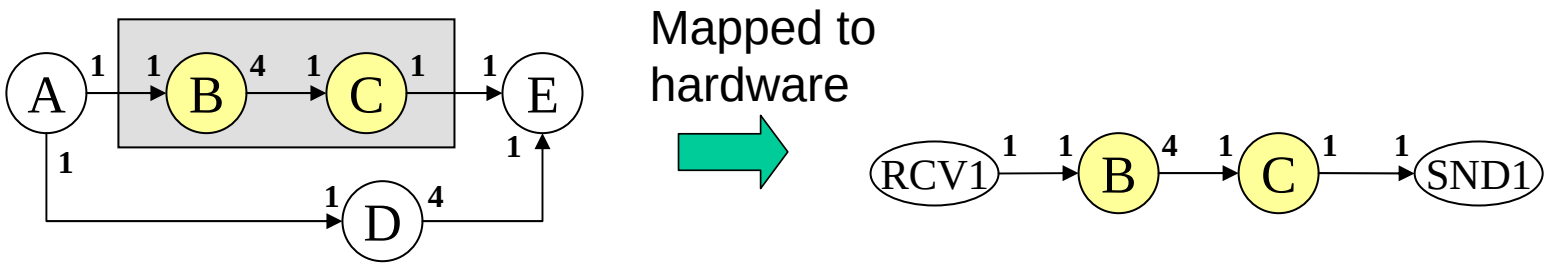


System Design Flow in PeaCE



HW/SW Interface

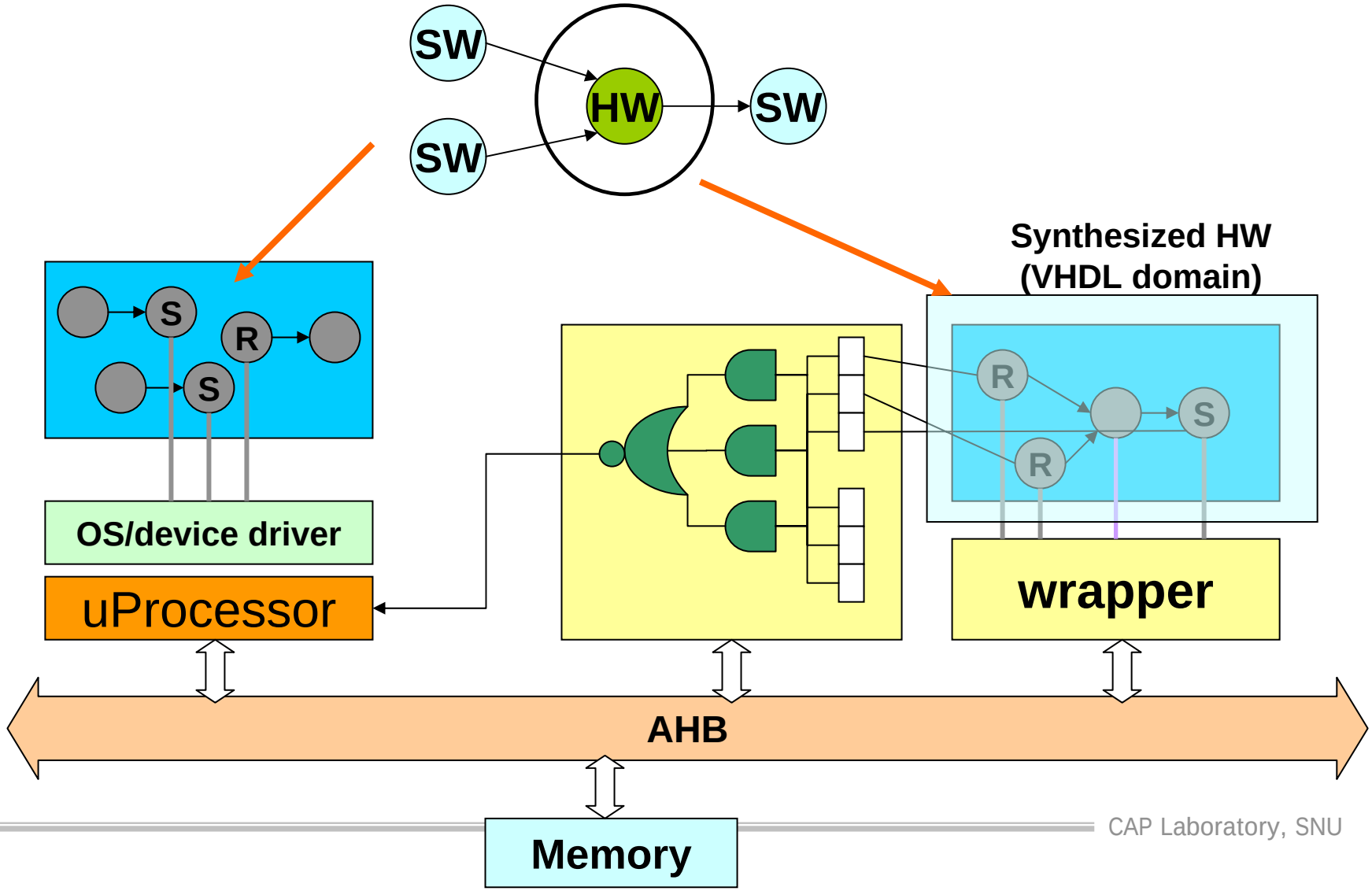
- **Hardware Synthesis for HW/SW Cosynthesis**
  - After HW/SW partitioning of an initial dataflow specification, a partitioned subgraph mapped to hardware is automatically generated.
  - A partitioned subgraph has interfacing blocks such as SND(send) and RCV(receive) blocks for communication.
    - *These interfacing blocks have internal buffers and shared memory access logics.*



Initial Dataflow Specification

Hardware Subgraph

HW/SW Cosynthesis



Contents

- **Introduction**
- **Previous works**
  - GRAPE, Meyr's work, Ptolemy, and PeaCE
- **Block Definition**
- **Controller Synthesis**
- **Schedule-Based Design**
- **FRDF Specification for more efficient HW implementation**
- **Conclusions & Future Directions**

GRAPE Approach

- **GRAPE (Graphical Rapid Prototyping Environment) is a HW/SW codesign environment for the functional emulation of DSP systems.**

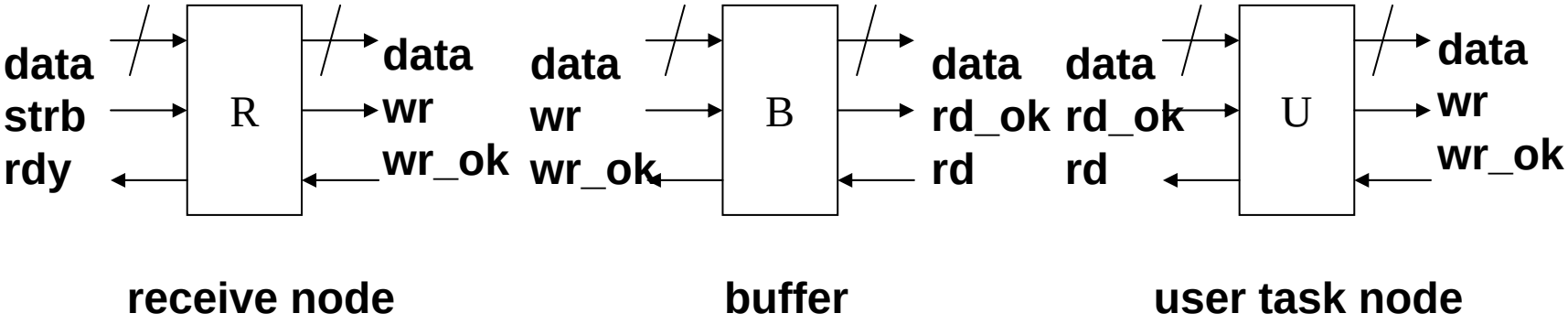
- **Using cyclo-static dataflow specification**
  - Sample rates are changed periodically

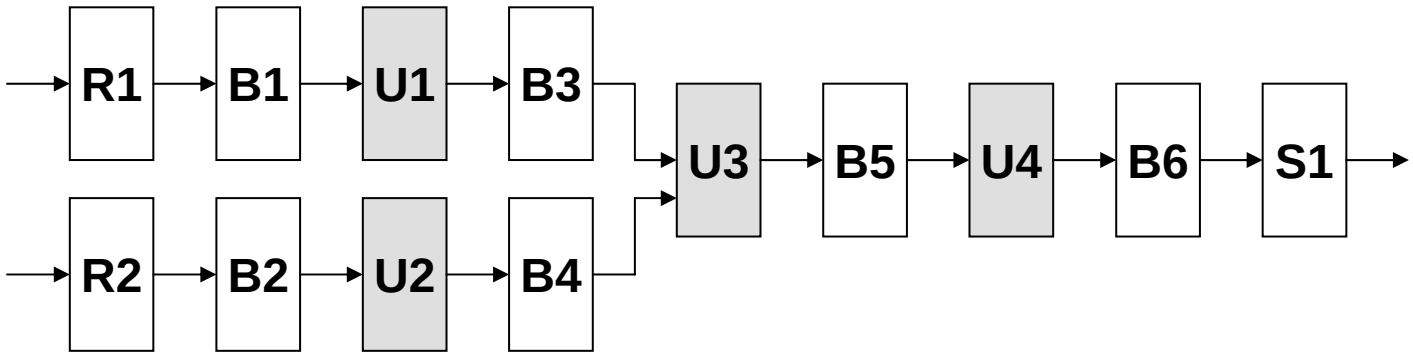
- **Distributed controls**
  - Using hand-shaking protocol between blocks
  - FIFO buffers
  - No central controller

- **Generated hardware implementation has one-to-one correspondence to dataflow specification**
  - Simple architecture generation

GRAPE Standard Interface

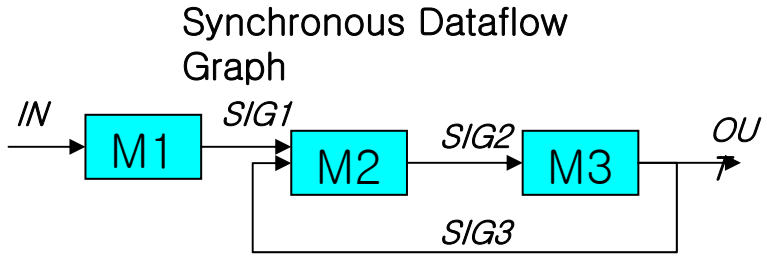
- Asynchronous communication between every blocks using hand-shaking protocol



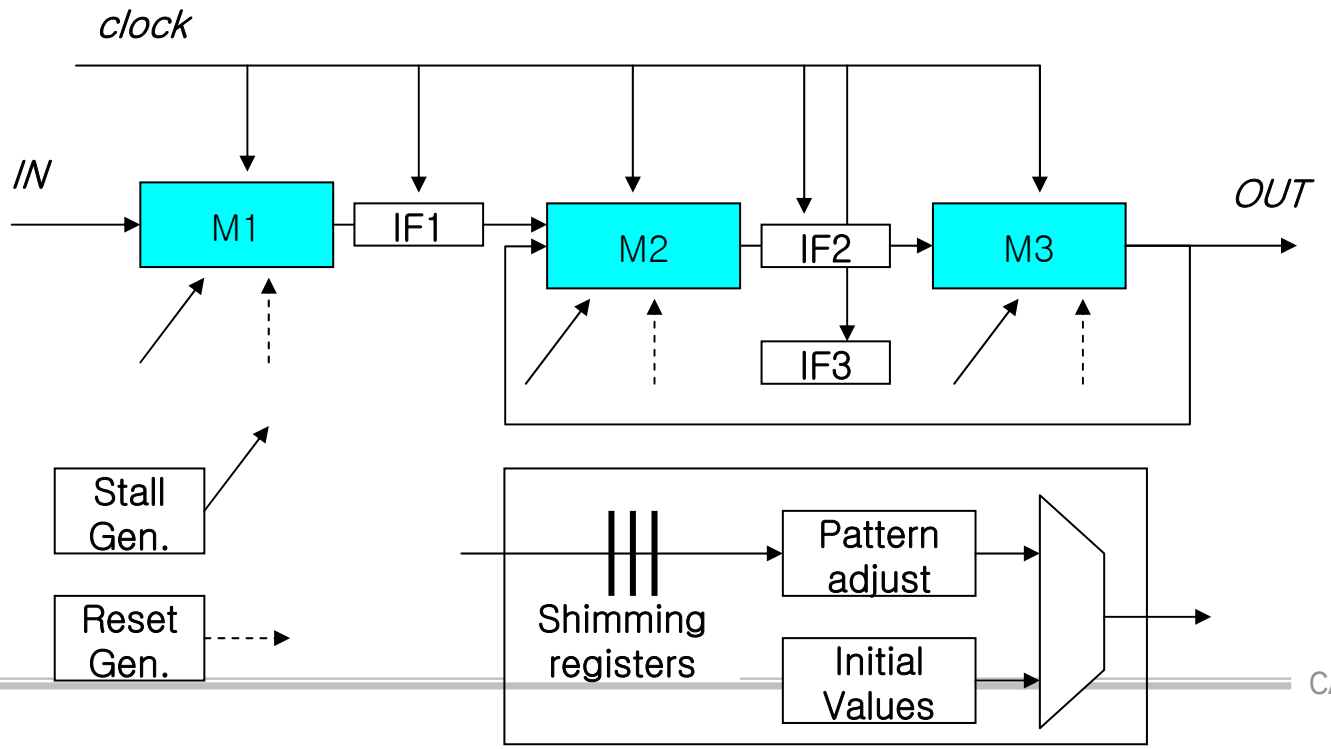


Meyr's Approach

- **Using Synchronous Dataflow**
- **Serialized I/O of multi-rate port**
- **Fully static I/O timing analysis**

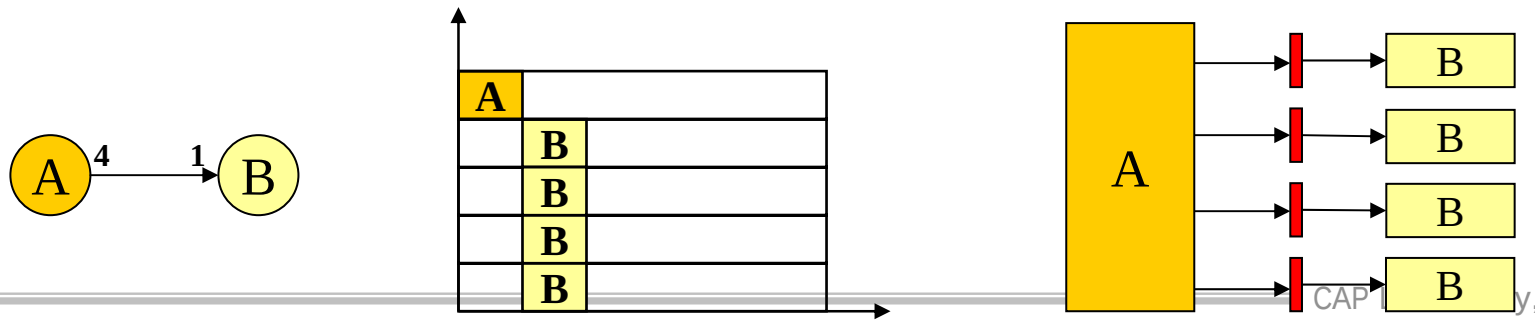


RTL Target Architecture



Ptolemy Approach: VHDL domain

- **Sequential VHDL code generation**
  - For simulation
  - Entire application is described in a single process using only variables.
- **Structural VHDL code generation**
  - For synthesis
  - Individual firings(or invocations) of a node are instantiated in separate hardware resources
    - *Fully parallel HW architecture for multi-rate specification*



Limitations of Previous Approaches

- **GRAPE**
  - Distributed control using handshaking protocol
  - Synthesis problem is simple
  - Only for rapid prototyping
- **Meyr's**
  - Supports only static I/O timings
  - Resource sharing is not considered
- **Ptolemy**
  - Impractically large area overhead in case of multi-rate specification

Comparison among Approaches

Approaches	Ptolemy	Meyr's	GRAPE	PeaCE
Implementation of multi-rate spec.	Parallel implementation	Sequential implementation	Sequential implementation	Parallel/Sequential/Hybrid implementation
Resource allocation	Multiple-resource allocation	Single-resource allocation	Single-resource allocation	Multiple/Single/Shared –resource allocation
Inter-block Communication	Synchronous	Synchronous	Asynchronous (FIFO)	Synchronous
Block control	Centralized control	Centralized control	Distributed control	Centralized Control
Block exec. time	Fixed	Fixed	Variable	Variable

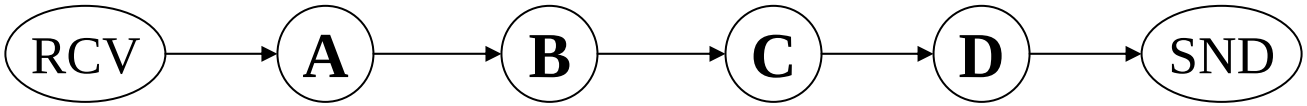
Contents

- **Introduction**
- **Previous works and our contributions**
- **Block Definition**
  - Block I/O Model & Block Types
- **Controller Synthesis**
- **Schedule-Based Design**
- **FRDF Specification for more efficient HW implementation**
- **Conclusions & Future Directions**

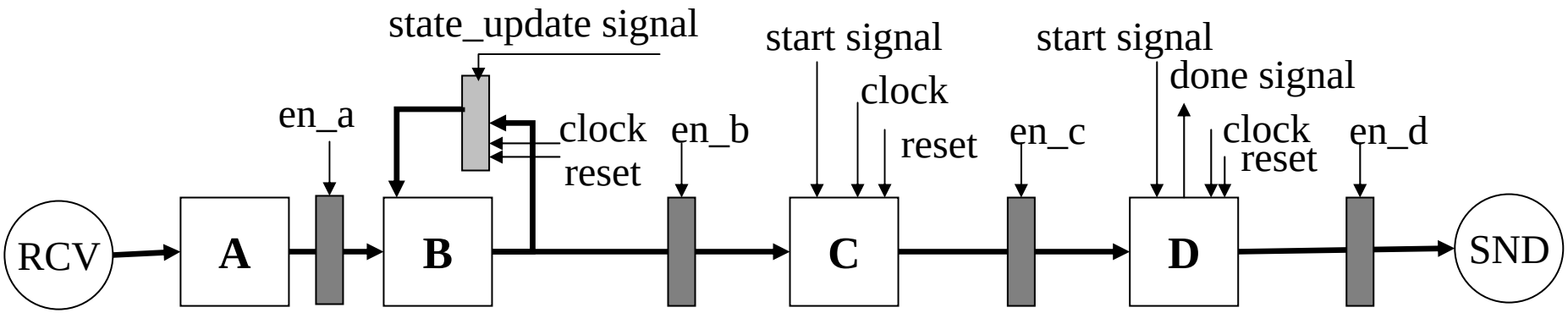
Block Libraries

- **Types of block implementations**
  - A : Combinational logic
  - B : Single-cycle sequential logic
  - C : Multi-cycle sequential logic with fixed execution time
  - D : Multi-cycle sequential logic with variable execution time.
- **Timing model of HW block**
  - Execution time of block
  - I/O of multi-rate block

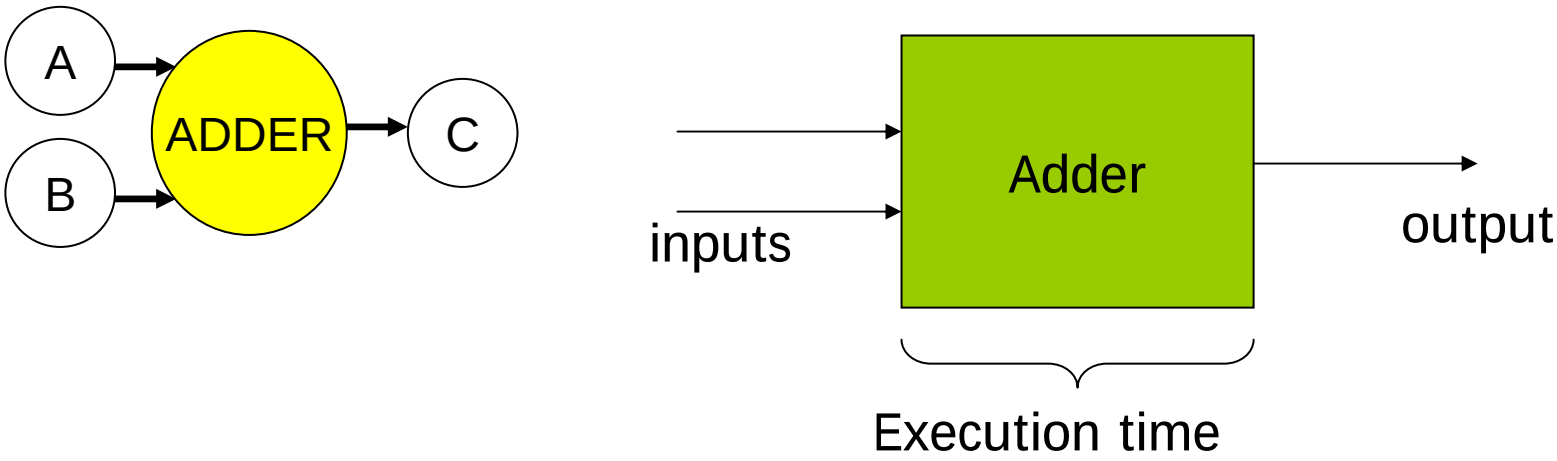
Block Types & Control Signals



Type A : combinational logic
Type B : single-cycle sequential logic
Type C : multi-cycle sequential logic with fixed execution time
Type D : multi-cycle sequential logic with variable execution time

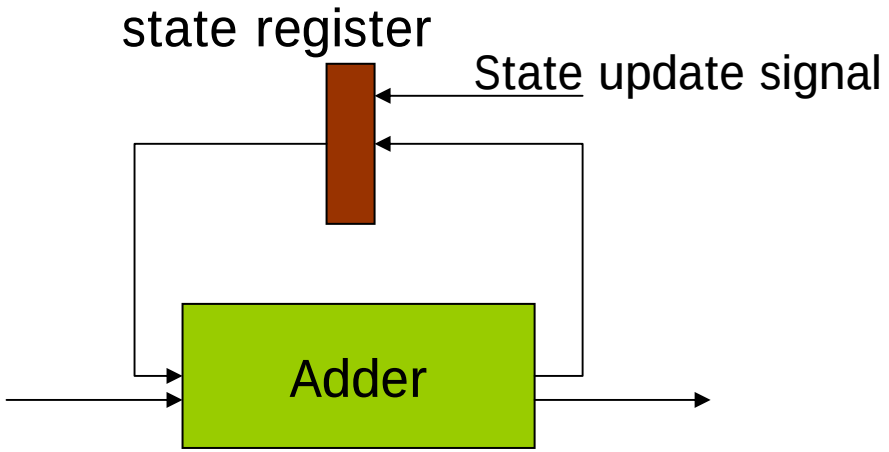


Type A : Combinational logic



Execution Time = propagation delay / clock period (cycles)

Type B : Single-cycle sequential logic



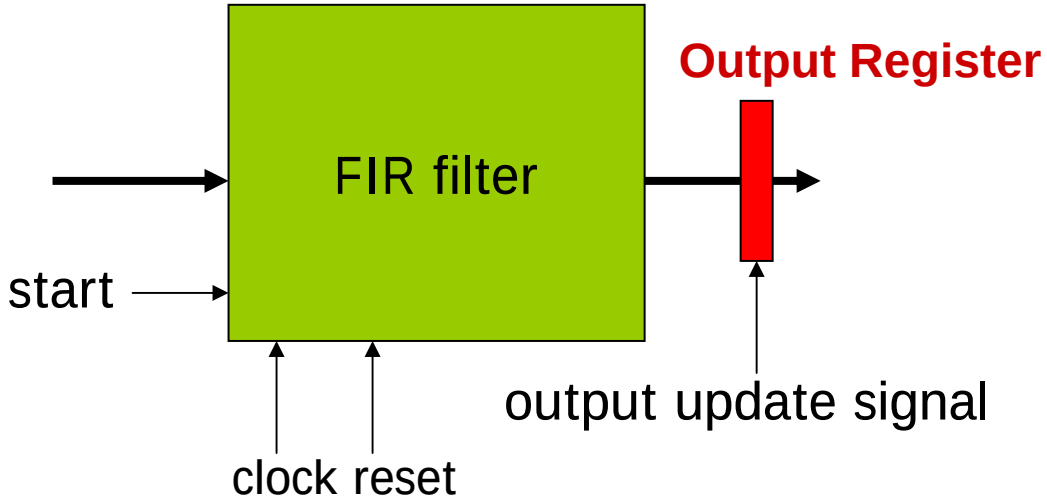
Mealy type state machine

VHDL Star with States

This logic is separated into combinational logic and state and implemented as Mealy machine.

Execution Time = propagation delay / clock period (cycles)

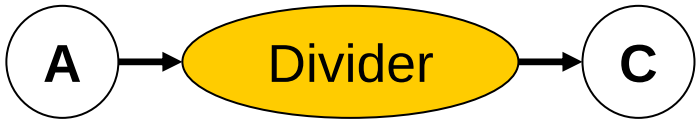
Type C : Multi-cycle sequential logic with fixed execution time

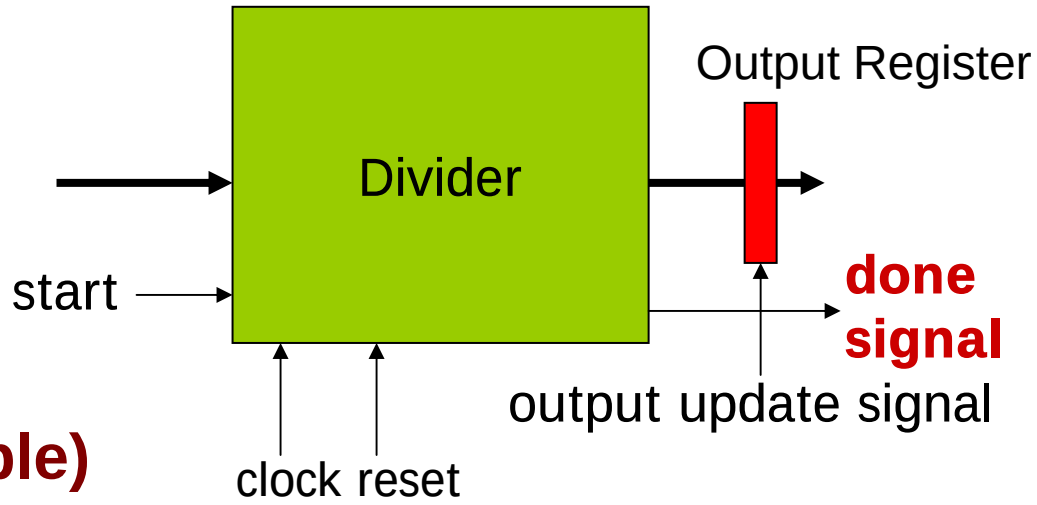


Multi-cycle logic (fixed)

The number of cycles is fixed.
Clock and reset signal are needed.
Controller should provide start and output update signal.
Execution time = specified number of cycles

Type D : Multi-cycle sequential logic with variable execution time

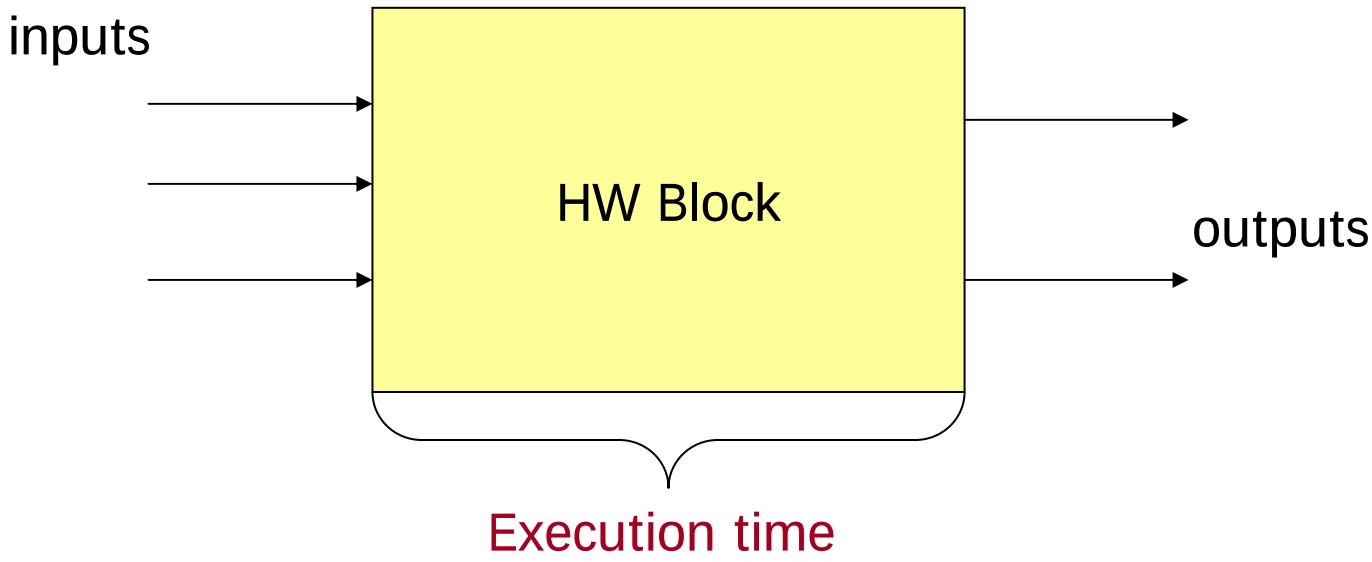




Multi-cycle logic (variable)

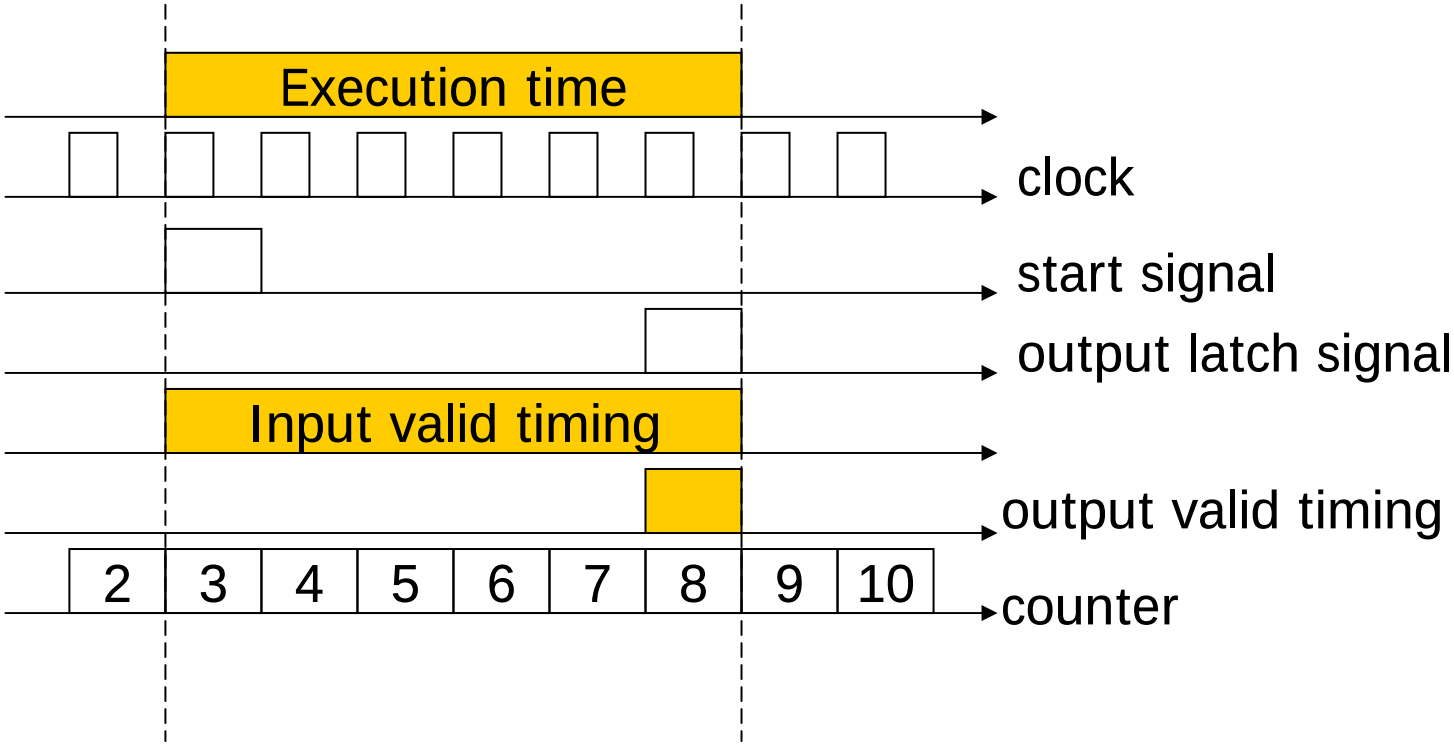
The number of cycles varies at run-time.
Clock and reset signal are needed.
Controller should provide start and output update signal.
Done signal should be generated by a library block and be used to decide its finish time by the controller.
Execution time = specified number of cycles

Timing Model of HW Library Block



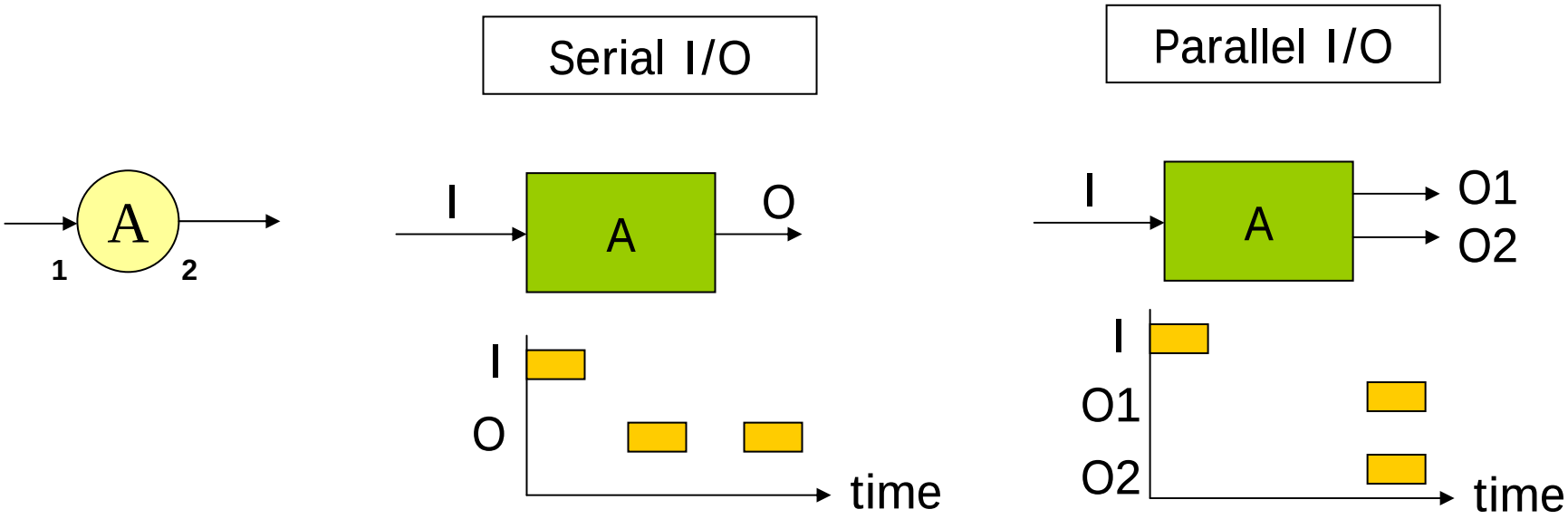
- **Strict Execution**
  - A block can start its execution after all its inputs are valid and finish its execution after all its outputs are valid.

Timing Model of HW Library Block

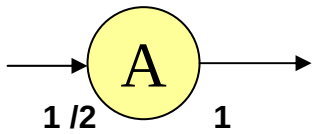


- **Start time = 3, End time = 8**
- **Execution time = End time – Start time + 1 = 6(cycles)**

Timing Model of Multi-rate Block



- **Only Parallel I/O of multi-rate block is supported.**
- **FRDF implementation can make it possible to serialize the I/O operation**



Contents

- **Introduction**
- **Previous works and our contributions**
- **Block Definition**
- **Controller Synthesis**
  - Interface Problem
  - Cascaded counter controller
  - Looping control & buffer management
- **Schedule-Based Design**
- **FRDF Specification for more efficient HW implementation**
- **Conclusions & Future Directions**

Controller Synthesis

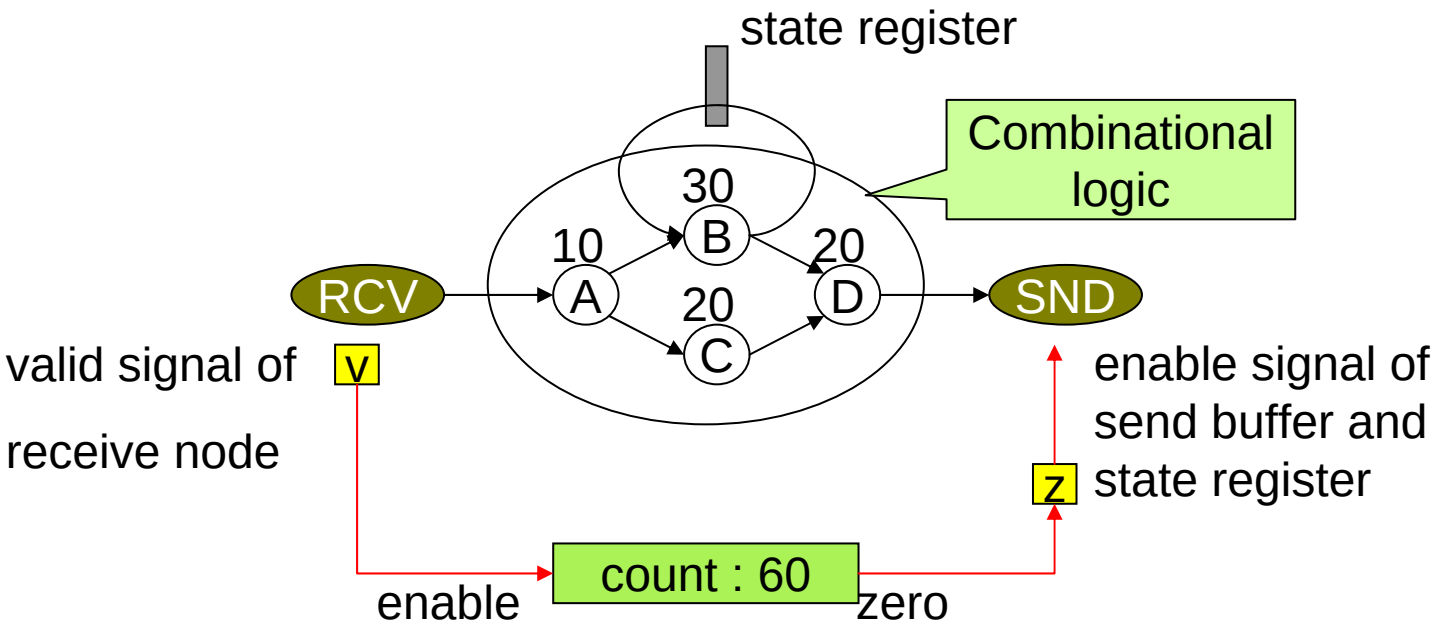
- **Issue 1: Solving non-deterministic timing of I/O**
  - Communication with the outside of hardware module
    - *HW/SW Interface*
  - Communication between blocks inside of hardware module
    - *Blocks with variable execution time*
- **Issue2 : Supporting various schedule**
  - Looped scheduling
  - Resource sharing
  - Buffer management

Communication between Modules

- **The types of communication schemes**
  - Synchronous communication
    - *Communication timing is predetermined.*
    - *Drawback : Tasks should be scheduled assuming the worst case execution time.*
  - Batch communication
    - *It is possible to emulate synchronous communication with buffers in asynchronous interface.*
    - *Drawback : It cannot be applied to DFG with global feedback.*
  - **Asynchronous communication**
    - *Communication timing is varied at run-time.*
  - There exist many cases in which asynchronous communication scheme is an efficient or a unique solution.
  - The asynchronous communication with the outside is not considered in the previous approaches except GRAPE

Basic Idea

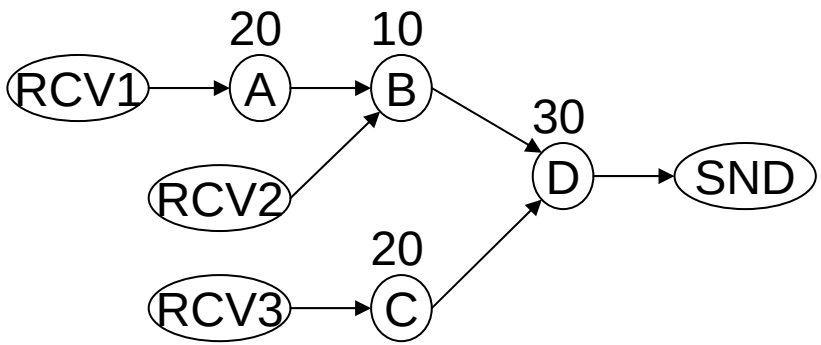
- **Counter-based solution**
  - simple and intuitive



Multiple RCV nodes

- **Main goal**
  - Obtaining the earliest time for the readiness of output regardless of the order in which the inputs arrive
- **Valid timing equation of send node**

$$VT = \max_i (RT_i + D_i)$$



$D_1 = 60$

$D_2 = 40$

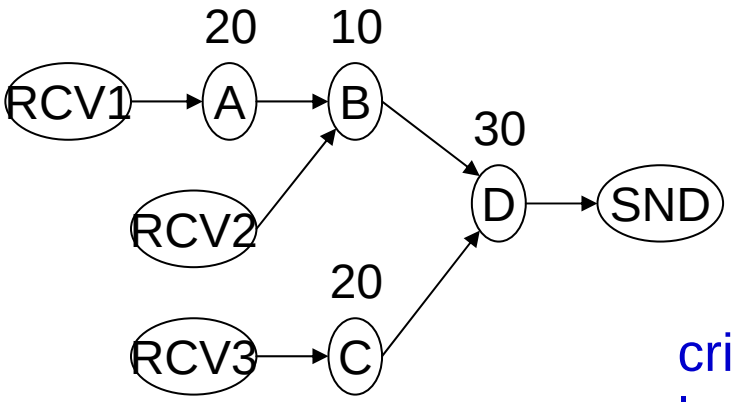
$D_3 = 50$

VT : valid timing

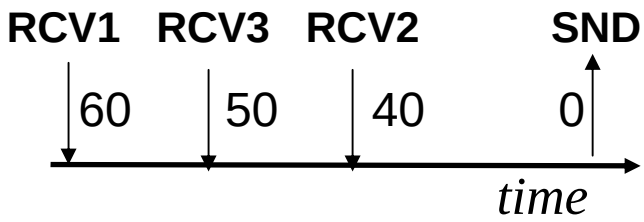
RT_i : receive timing of *i*-th receive node

D_i : critical path length from the *i*-th receive node

Cascaded Counter: Idea

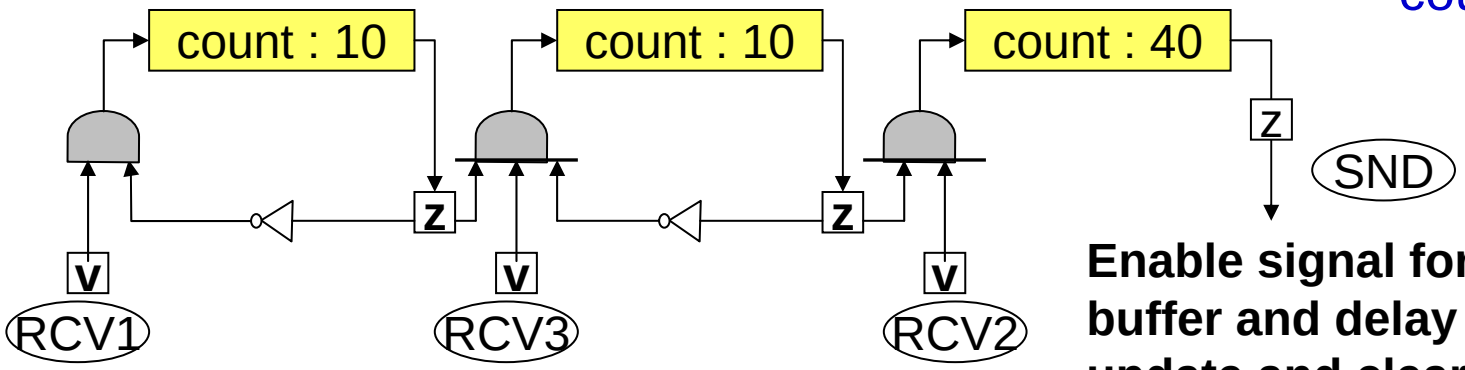


D1 = 60, D2 = 40, D3 = 50



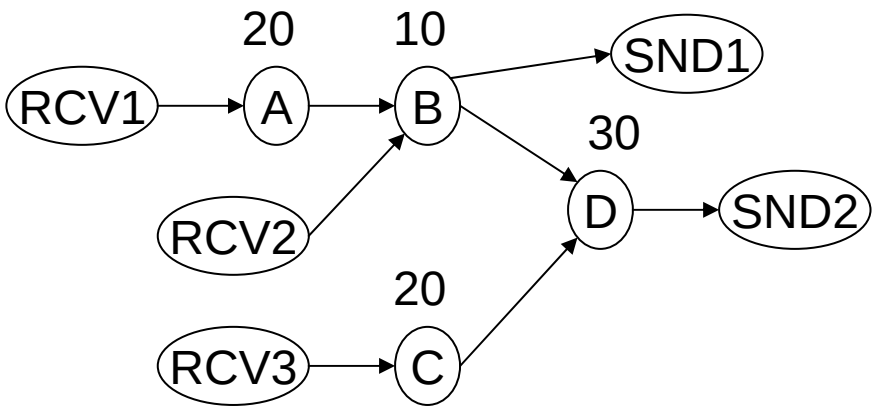
critical path
length computation

cascaded
counter



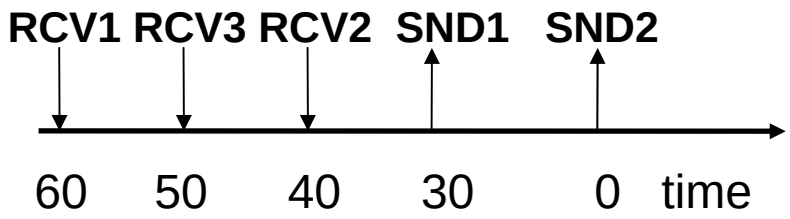
Enable signal for send buffer and delay register update and clear signal for valid and zero register

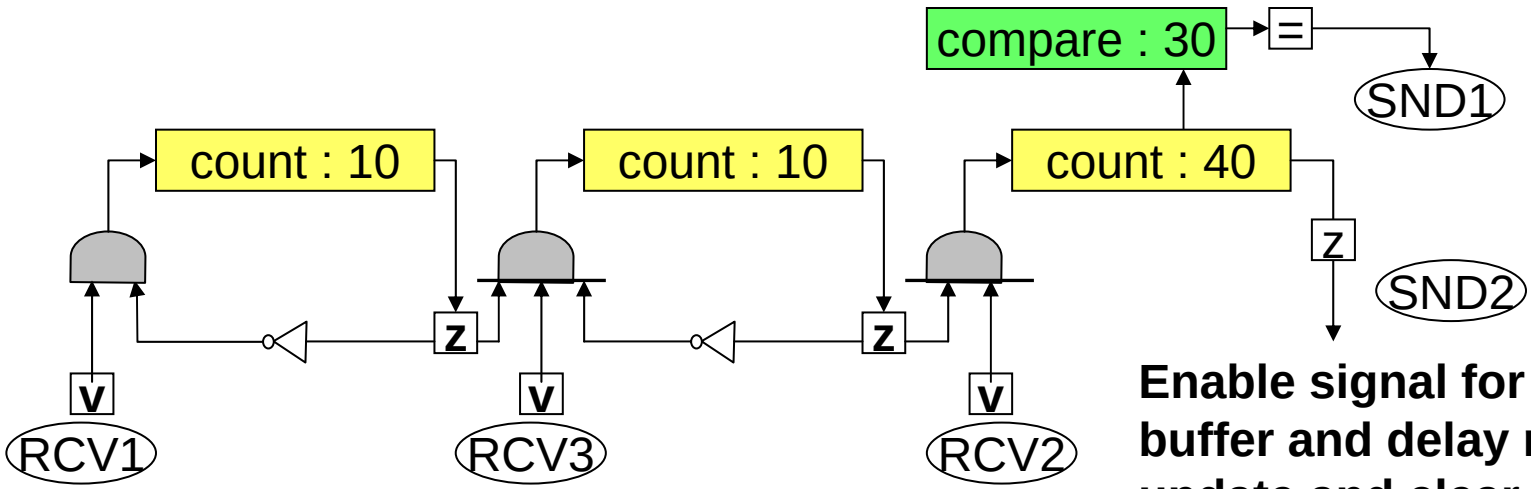
With Multiple Send Nodes



D1,1 = 30 D1,2 = 60 D2,1 = 10

D2,2 = 40 D3,2 = 50



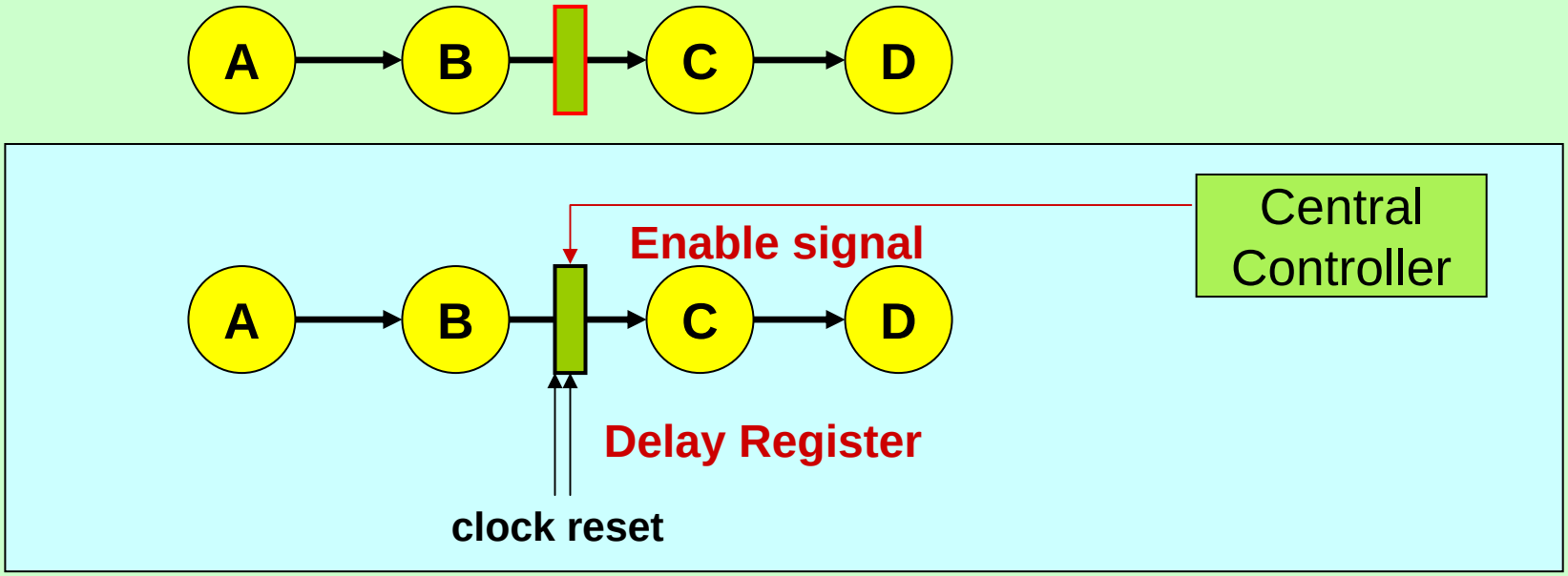


Enable signal for send buffer and delay register update and clear signal for valid and zero register

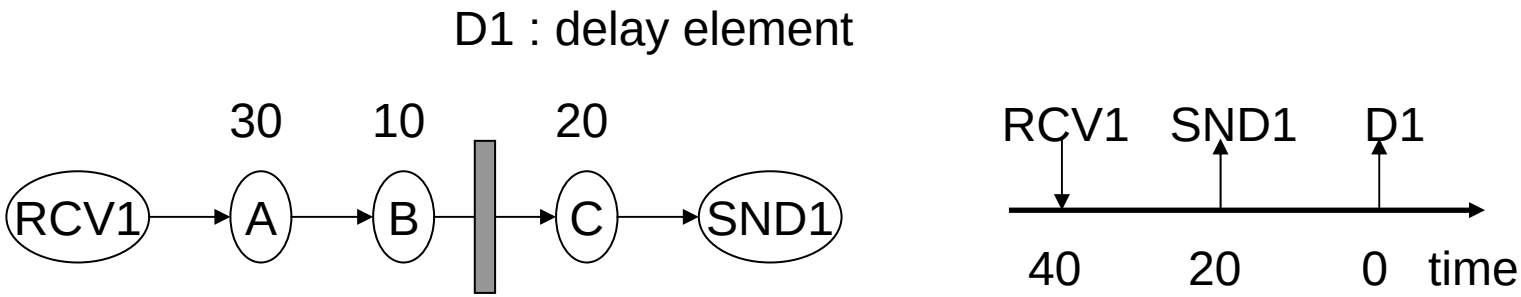
With Delay Registers

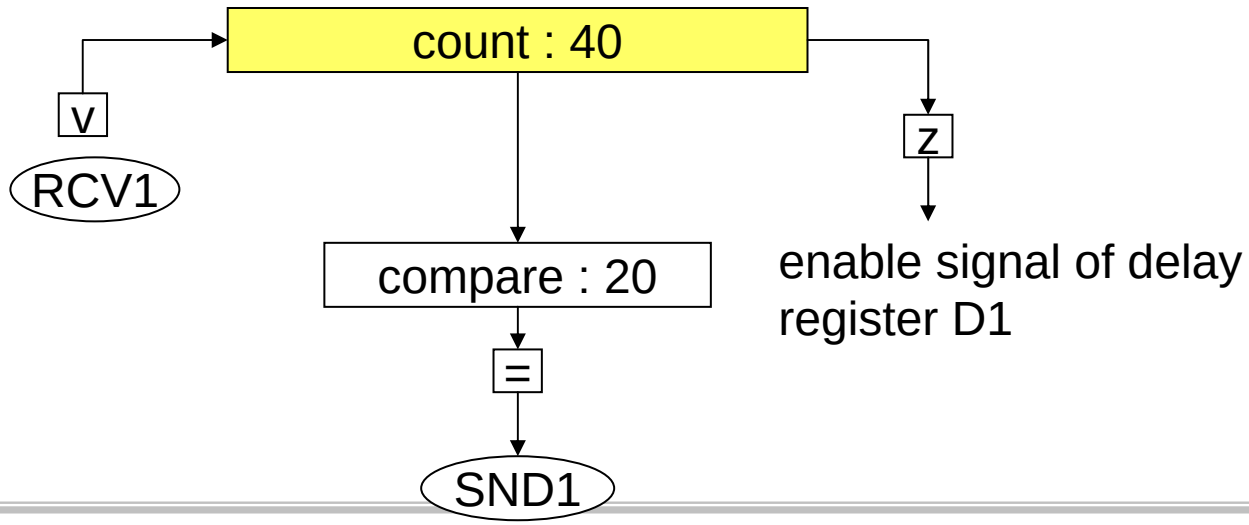
Delay elements

Delay elements may exist and they correspond to data registers in hardware implementation.



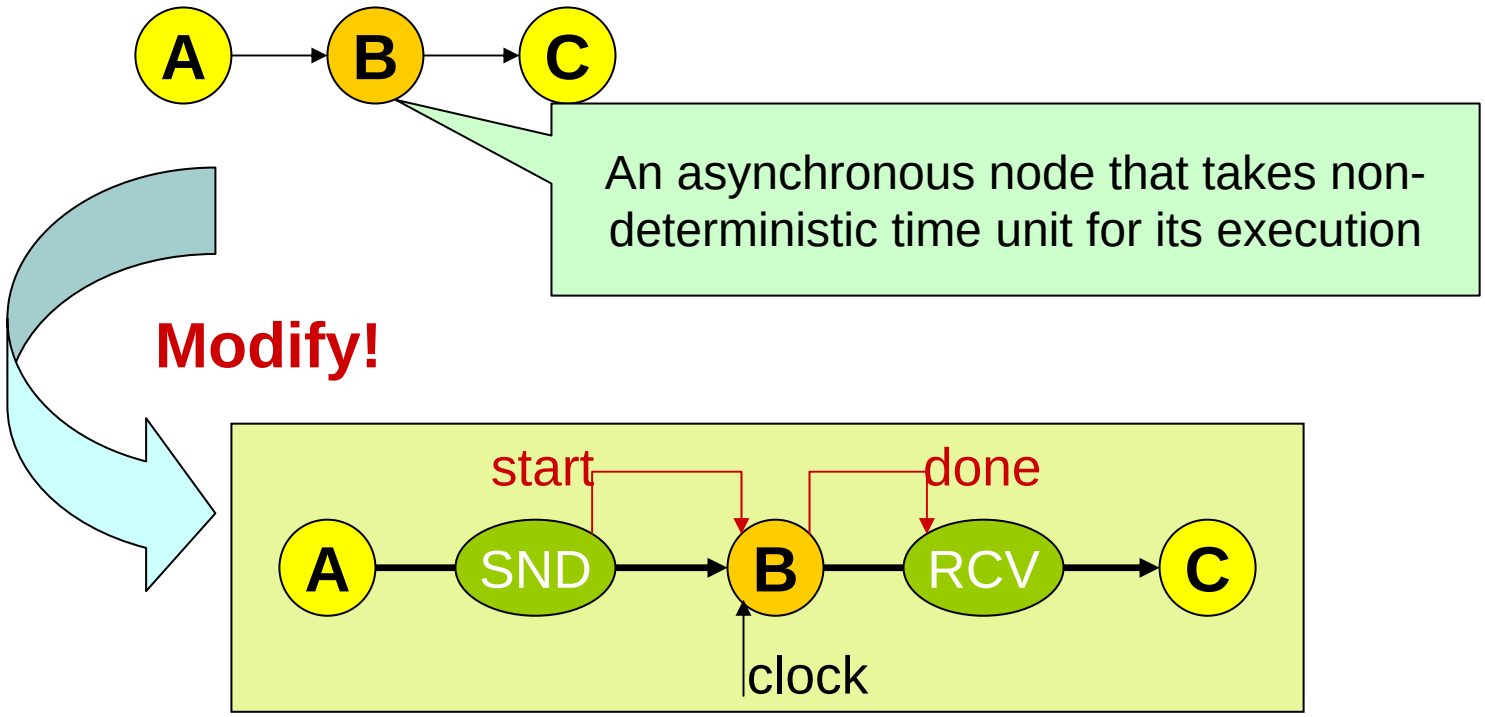
With Delay Registers





Nodes with Variable Execution Time

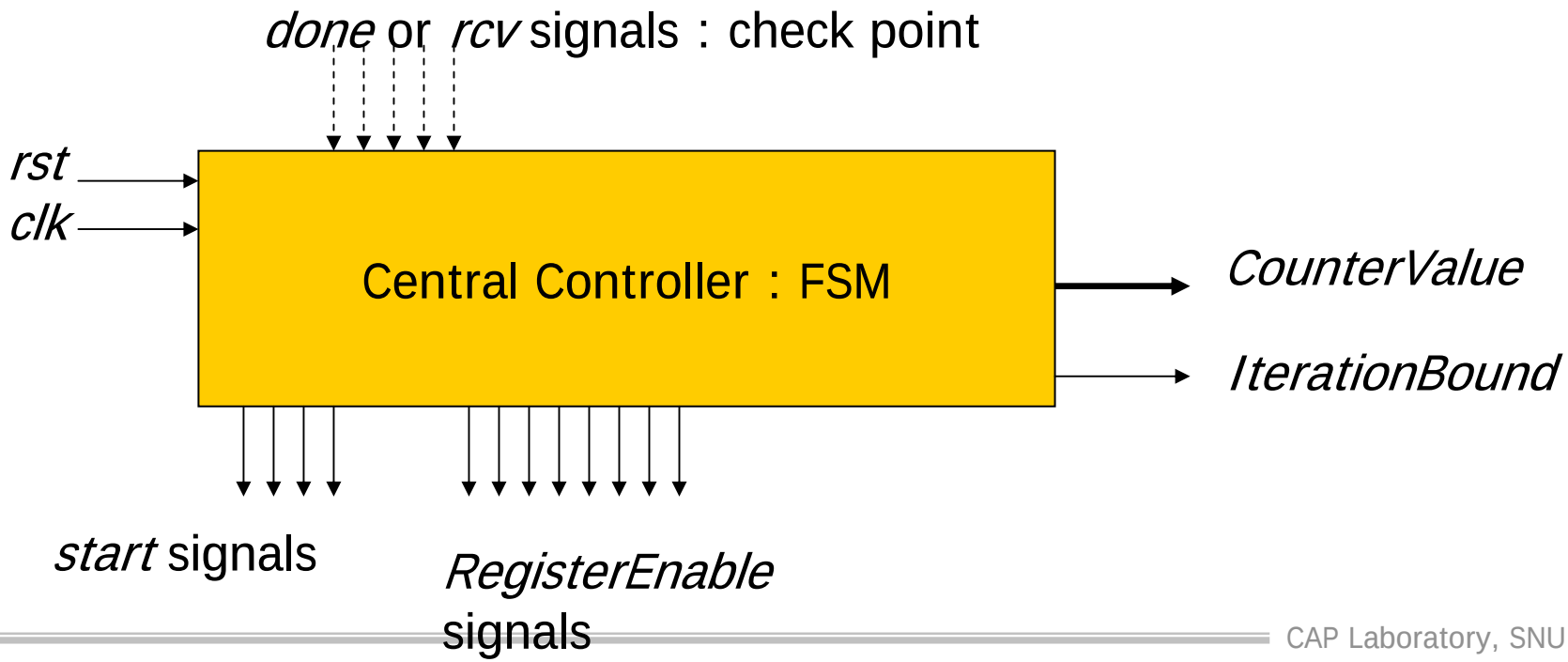
- **The cascaded counter controller provides a clean solution for this node.**



Equivalent FSM Controller Implementation

Currently, we implement FSM controller equivalent to cascaded counter in VHDL domain of PeaCE.

In this implementation, we use only one increasing counter with multiple *check* logics.



Equivalent FSM Controller Implementation

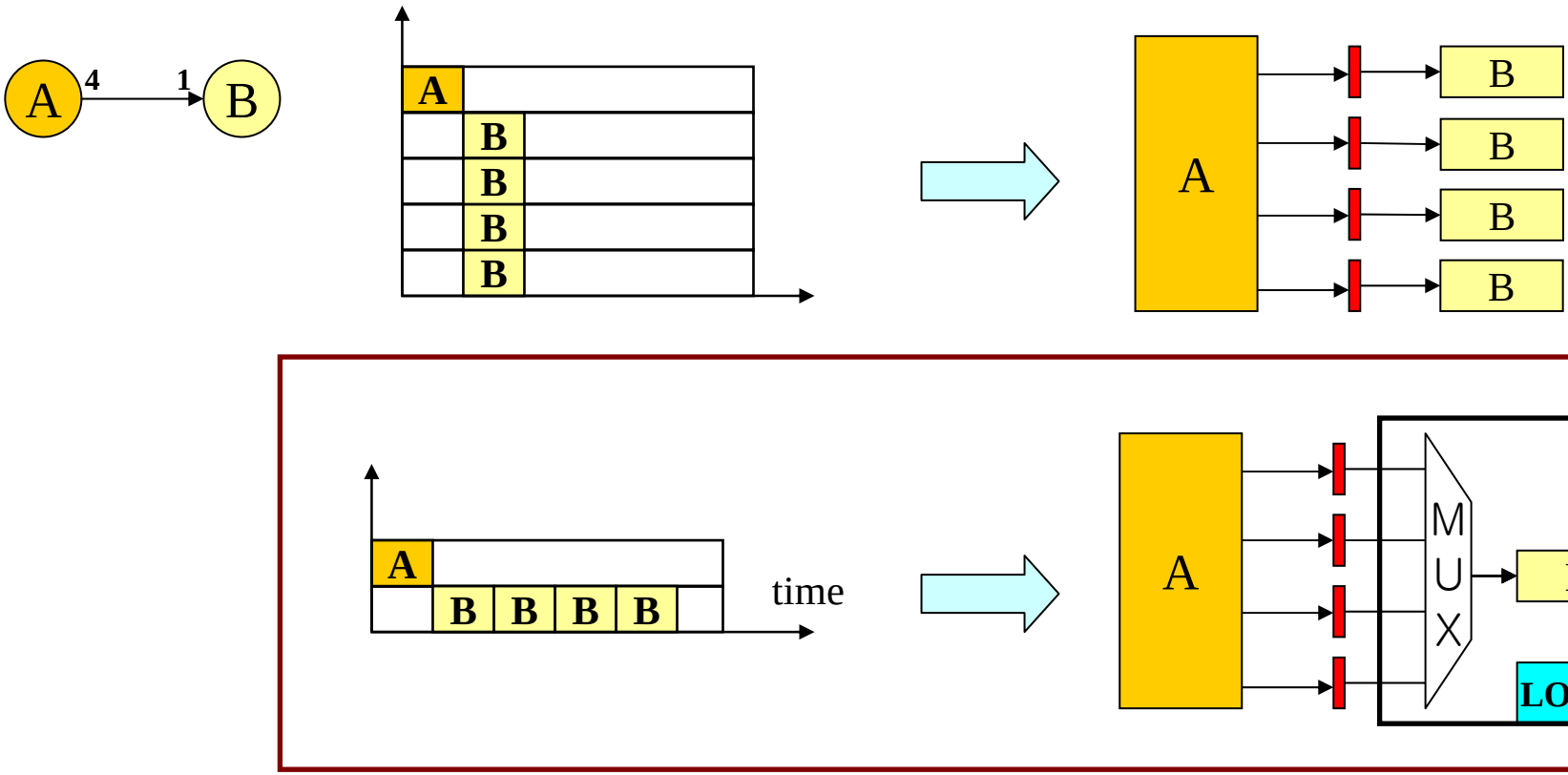
Example Code

```
If rst = '1' then
        Counter <= 0;
elsif rising_edge(clk) then
        if Counter = CheckValue0 and CheckSig(0) = '0' then
                Counter <= Counter;   -- hold value
        elsif Counter = CheckValue1 and CheckSig(1) = '0' then
                Counter <= Counter;   -- hold value
        elsif Counter = LastValue then
                Counter <= 0;         -- initialize
        else
                Counter <= Counter+1;   -- counting..
        end if;
end if;
```

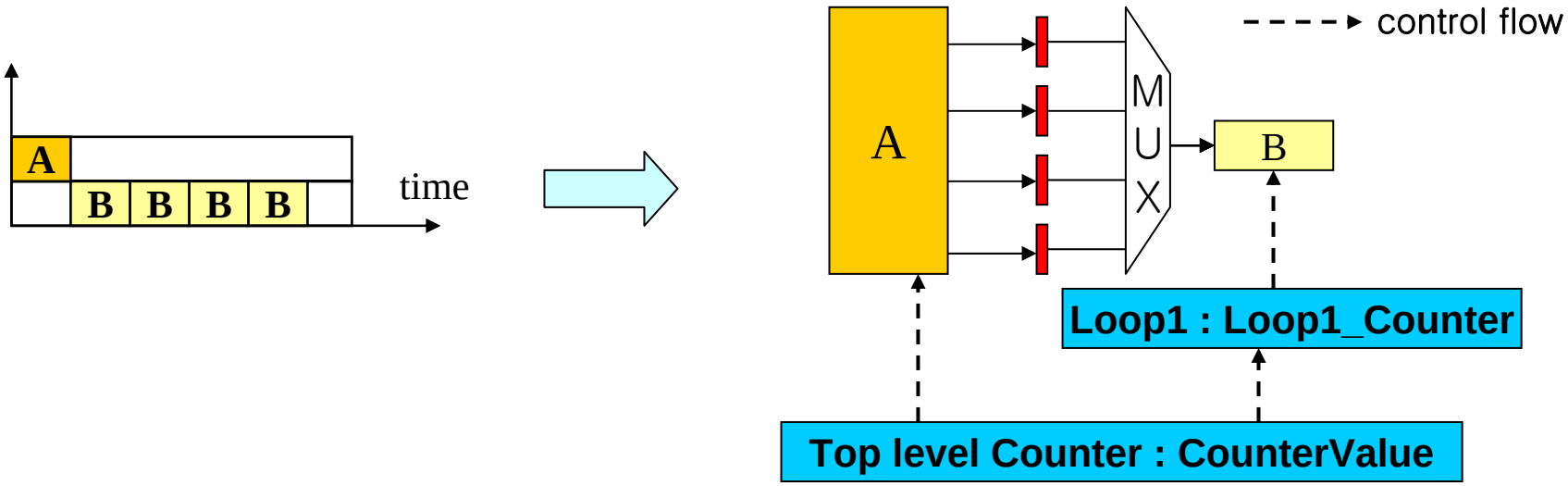
```
If Counter = LastValue then
        IterationBound <= '1';
Else IterationBound <= '0';
```

Looping Control

- PeaCE supports controller generation for looped schedule
- Looped schedule can be structured hierarchically.



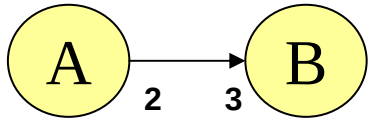
Looping Control



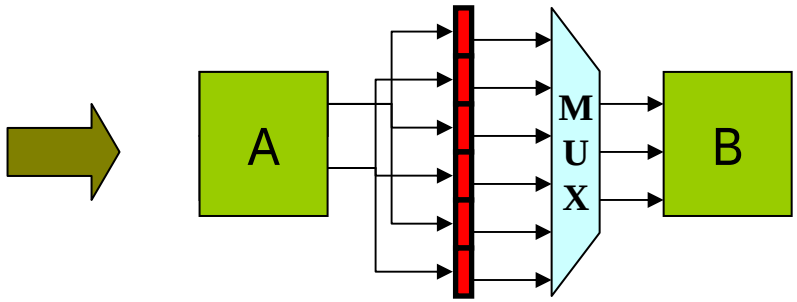

```
A_start <=
    '1' when CounterValue = 0 else  '0';

B_start <=
    '1' when Loop1_Counter = 0 and Loop1_busy = '1' else  '0';

Loop1_start <=
    '1' when CounterValue = 20 else '0';
```

Buffer Management

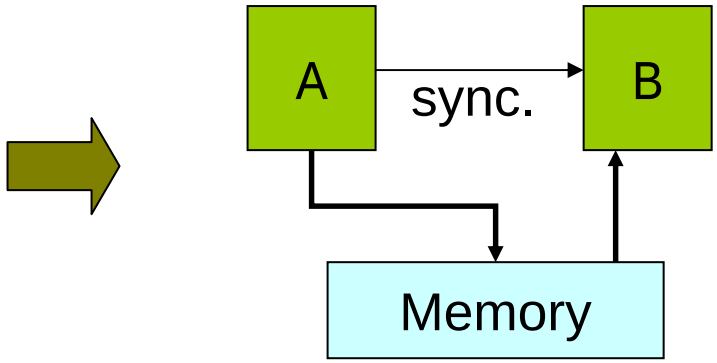
- **Multi-rate buffering**
- **Data types**
  - Int, Macroblock(16x16), Frame(176x144)



- **Register : small data type**
  - I/O timing control
  - Buffer allocation

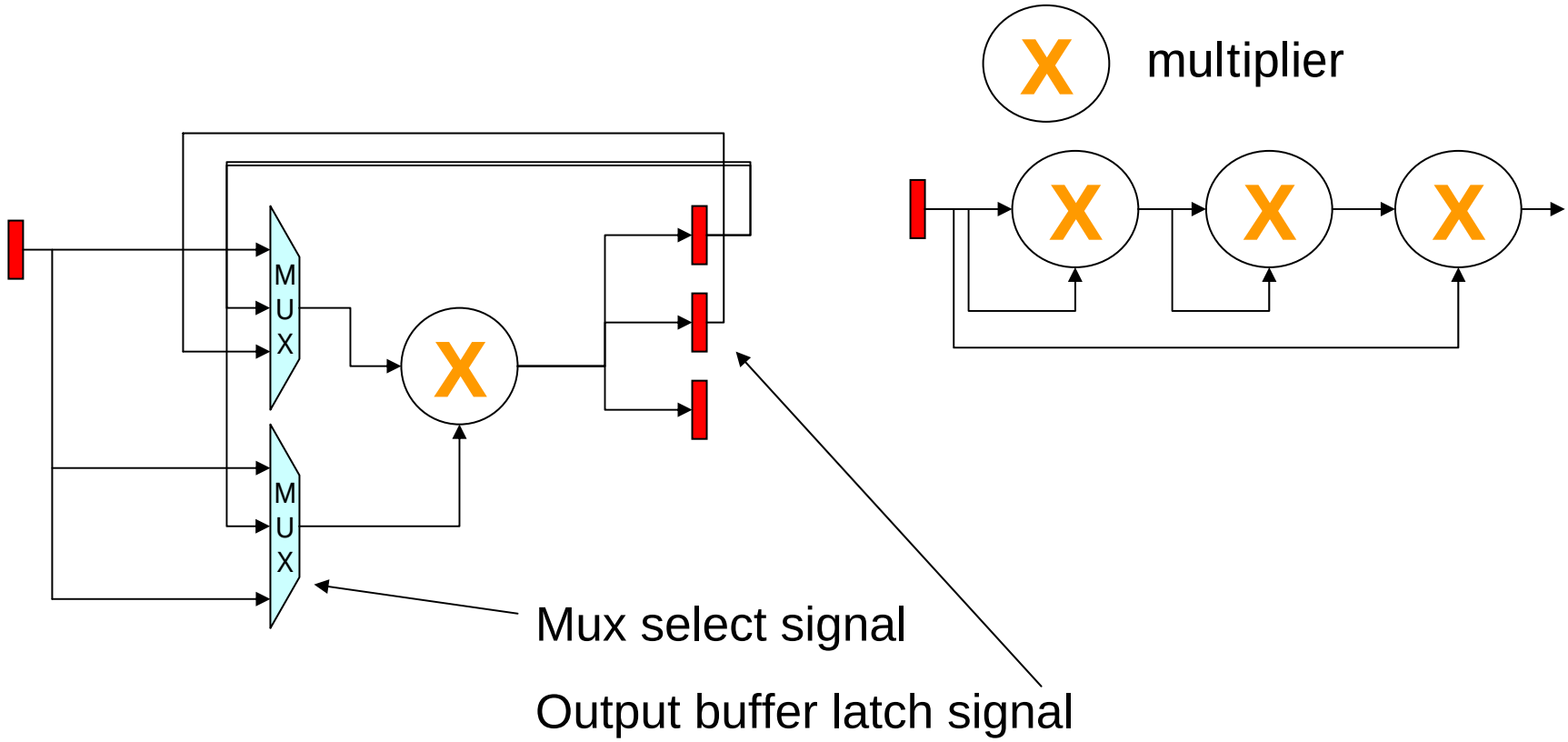


- **Memory : large data type**
  - Memory access logic
  - Synchronization
  - Memory allocation



Resource Management

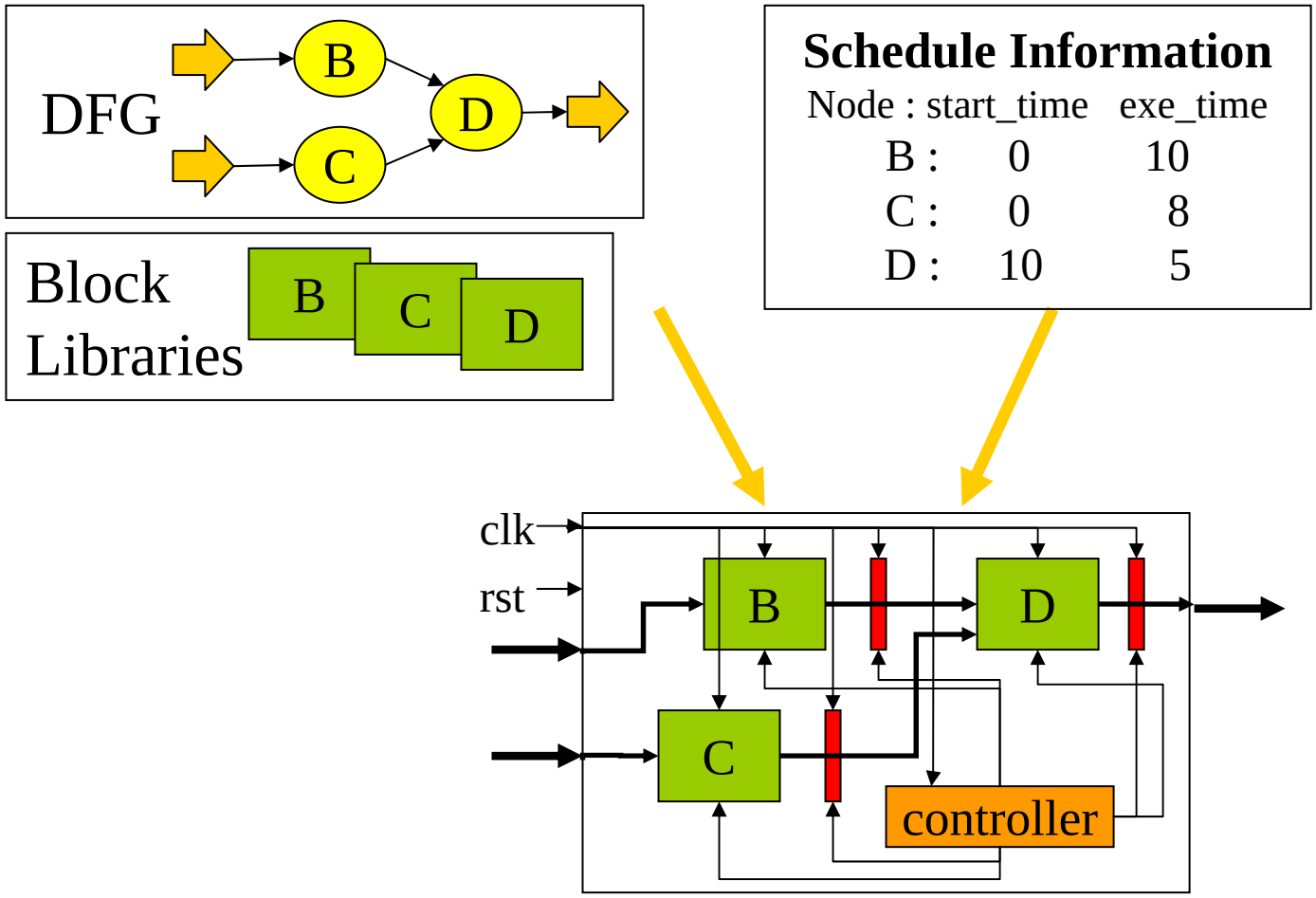
- **Resource sharing or Multiple instantiation**
- **Input multiplexing and output buffer access**



Contents

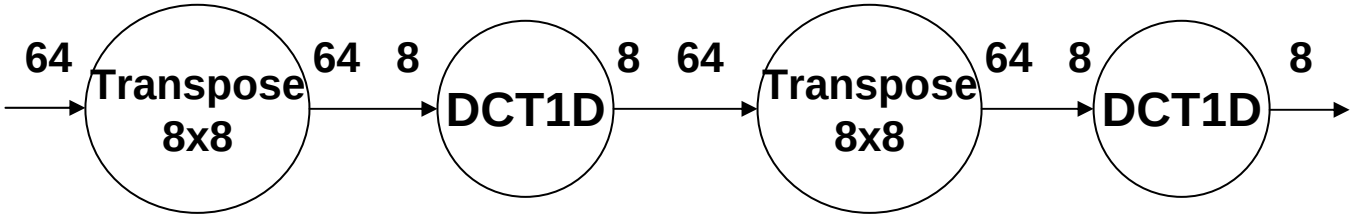
- **Introduction**
- **Previous works and our contributions**
- **Block Definition**
- **Controller Synthesis**
- **Schedule-Based Design**
  - Motivation
  - Schedule information & controller generation
  - Experiments
- **FRDF Specification for more efficient HW implementation**
- **Conclusions & Future Directions**

Schedule-based HW Synthesis

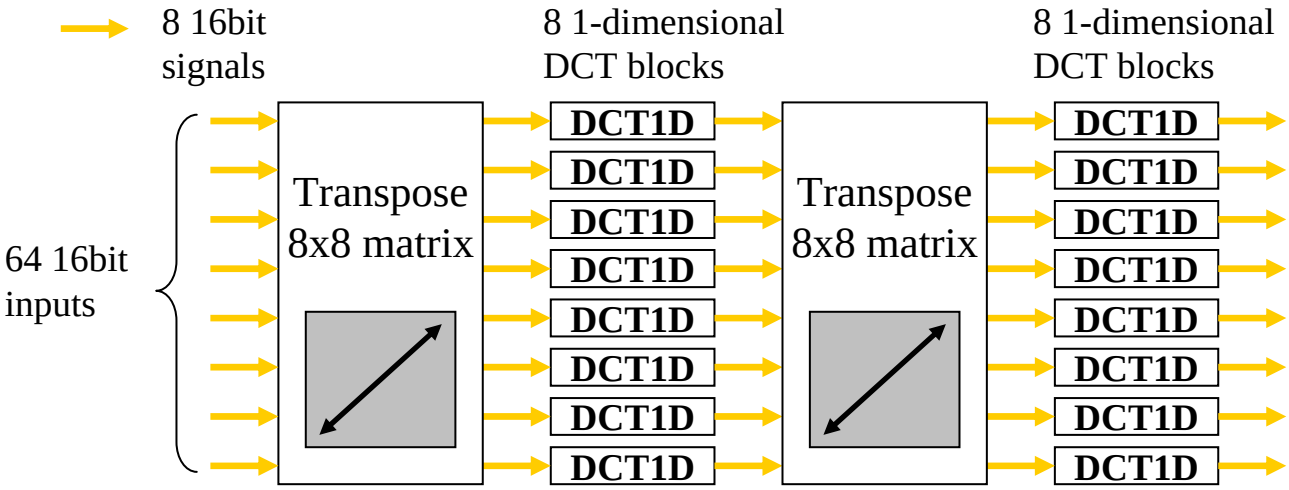


Previous Works

- **2-dimensional DCT algorithm**

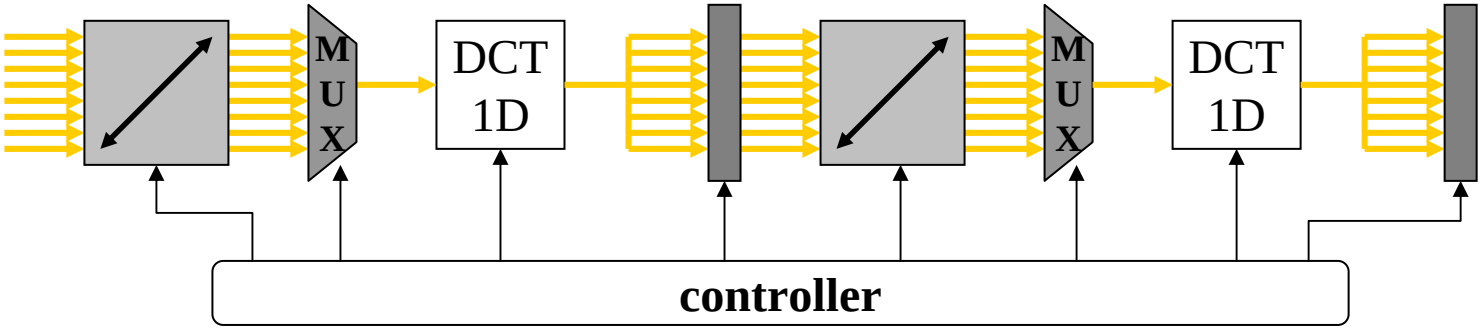


- **Generated Hardware Architecture from Ptolemy**

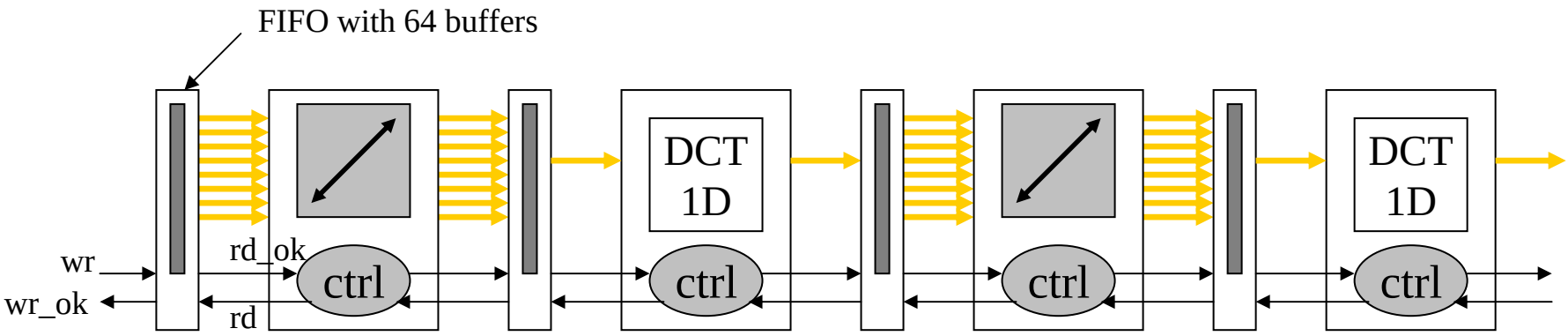


Previous Works

- **Generated Hardware Architecture from Meyr's works**

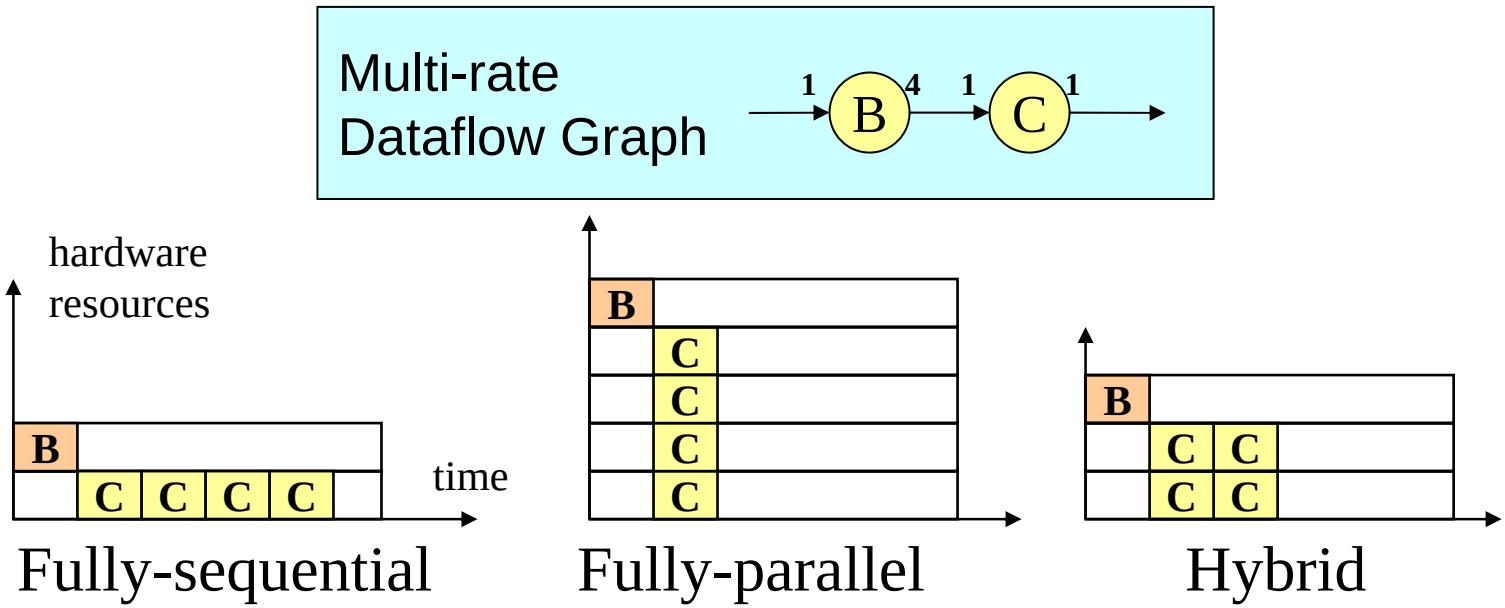


- **Generated Hardware Architecture from GRAPE**

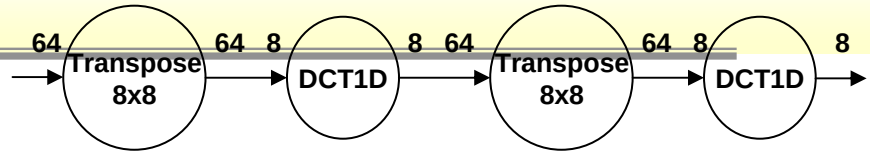


Motivation

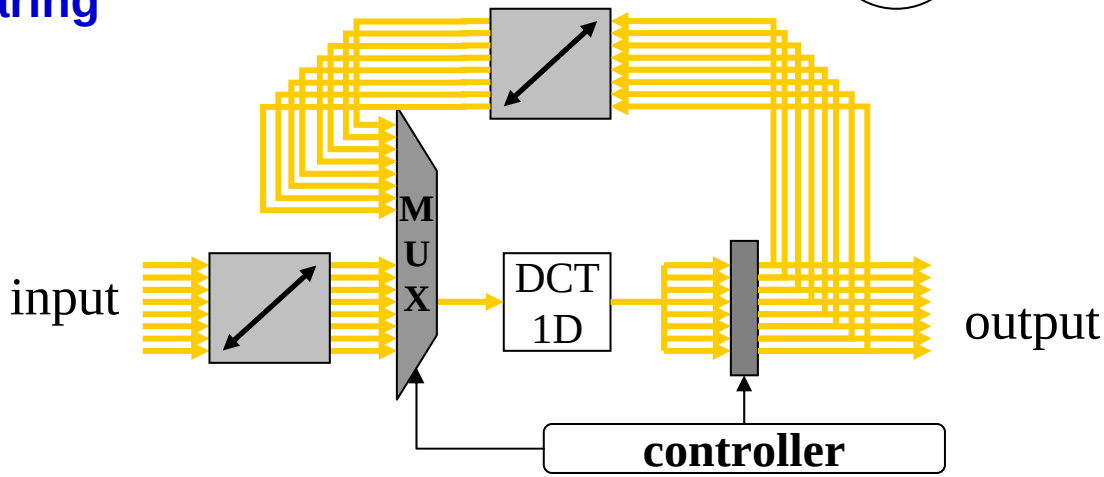
- **In the previous works, a single execution schedule is assumed for HW implementation.**
- But, proposed approach allows the designer to provide the execution schedule: a multi-rate dataflow graph can be implemented into many hardware architectures.



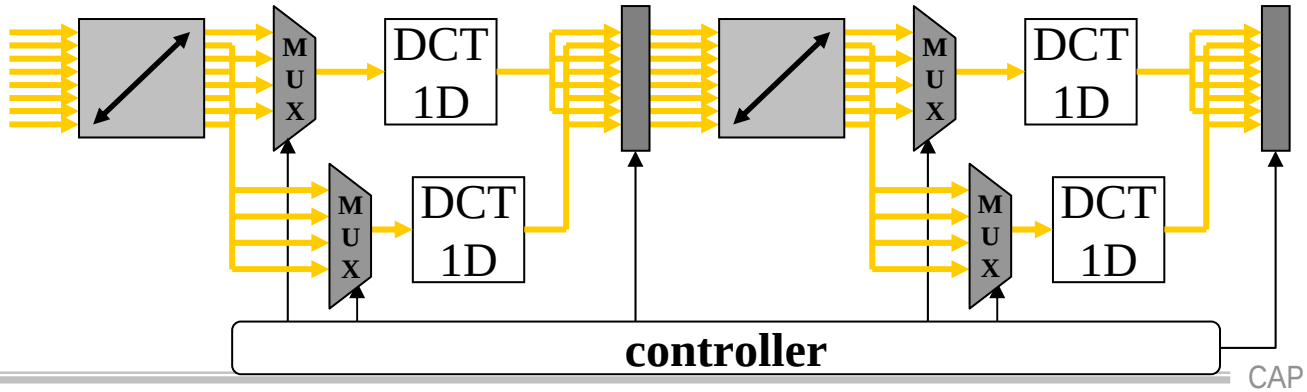
Motivation



- **Sharing**



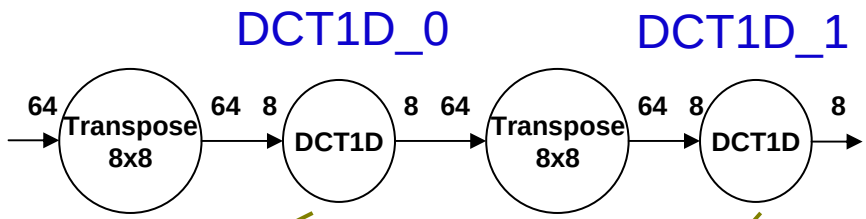
- **multiple-instantiation**

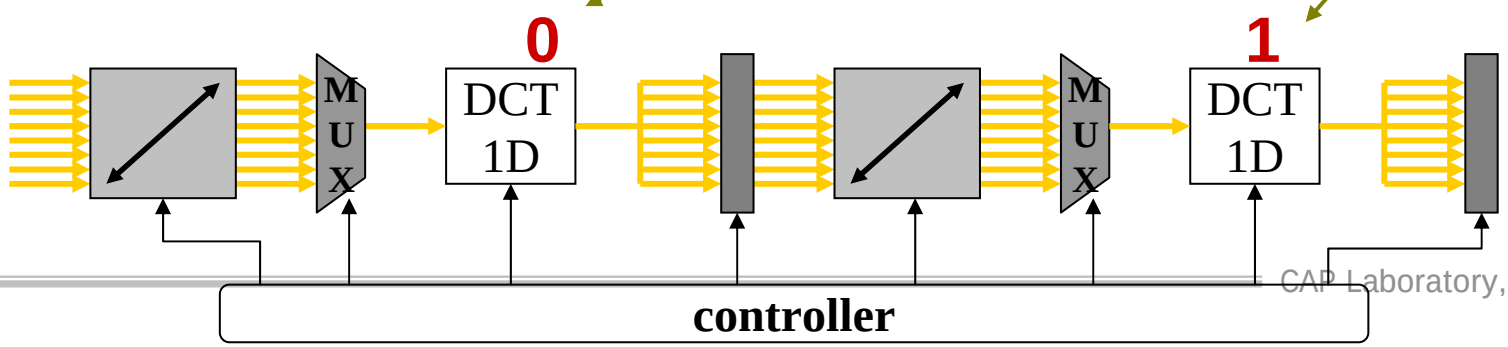


Schedule Information 1

1-to-1 mapping of
graph node ←→ HW resource

```
# resource allocation table
Transpose    2
DCT1D        2
# resource mapping & schedule information
# (instance name, resource number, start, duration)
# loop ( loop count, start, loop period)
Transpose_0     0     0     1
Loop   8   1   2  {
        DCT1D_0     0     0     2
}
Transpose_1     1    17     1
Loop   8    18    2 {
        DCT1D_1     1     0     2
 }
```

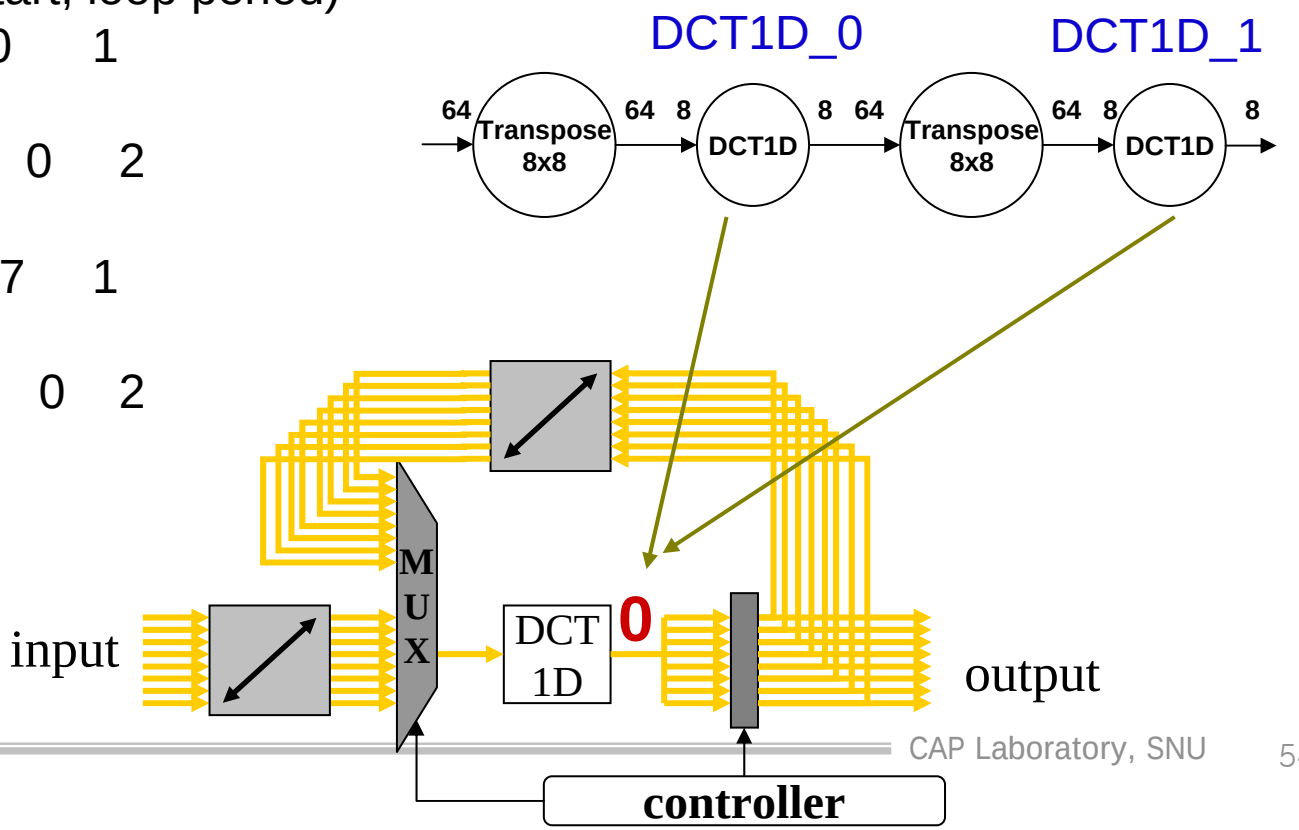




Schedule Information 2 : Sharing

N-to-1 mapping of
graph node ←→ HW resource

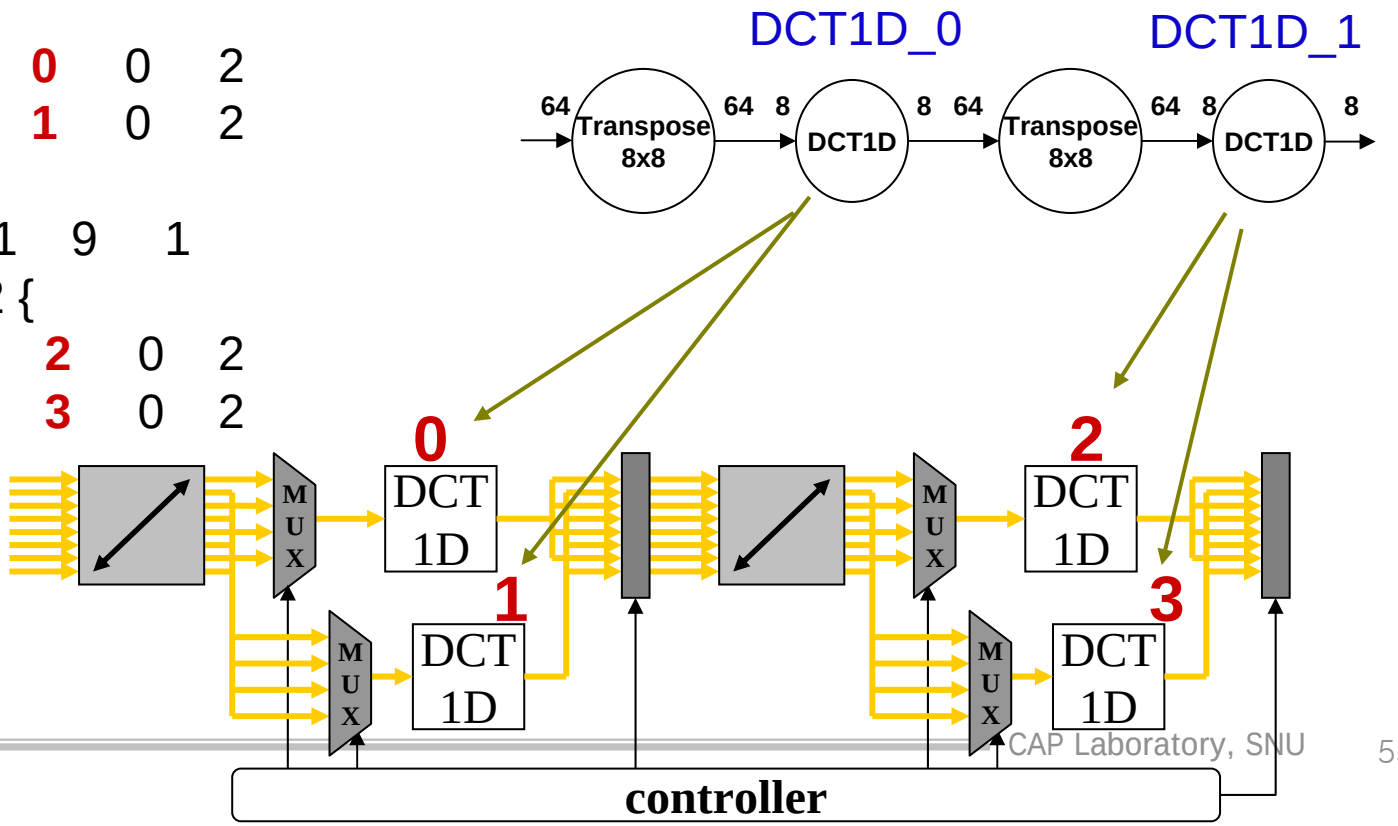
```
# resource allocation table
Transpose    2
DCT1D        1
# resource mapping & schedule information
# (instance name, resource number, start, duration)
# loop ( loop count, start, loop period)
Transpose_0     0     0     1
Loop   8   1   2  {
        DCT1D_0     0     0     2
}
Transpose_1     1    17     1
Loop   8    18    2 {
        DCT1D_1     0     0     2
 }
```



Schedule Information 3 : Multiple Instantiation

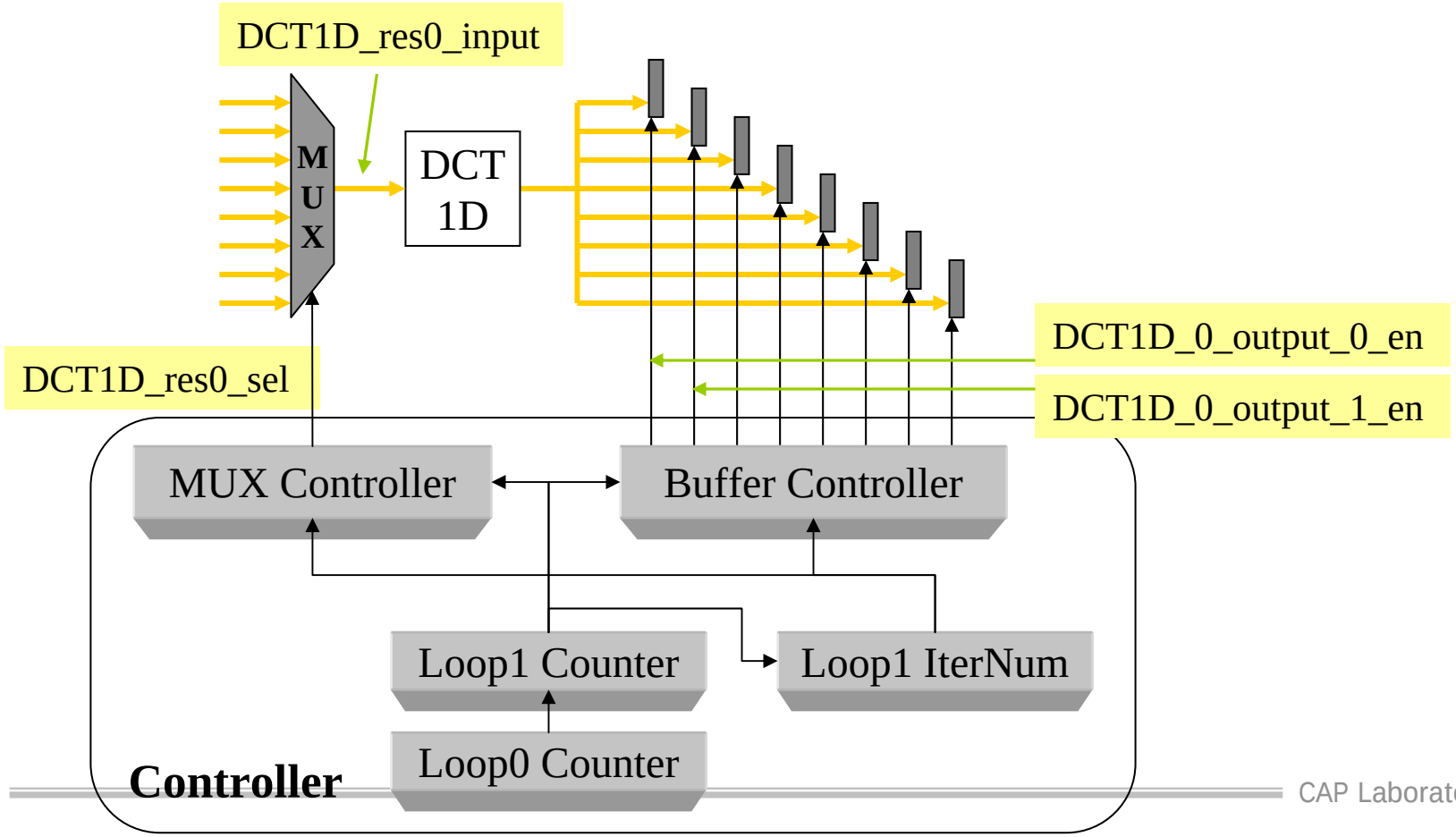
1-to-N mapping of
graph node ←→ HW resource

```
# resource allocation table
Transpose    2
DCT1D        4
# resource mapping & schedule information
Transpose_0     0     0      1
Loop   4   1   2  {
        DCT1D_0     0     0     2
        DCT1D_0     1     0     2
}
Transpose_1     1    9     1
Loop   4    10    2 {
        DCT1D_1     2     0     2
        DCT1D_1     3     0     2
}
```

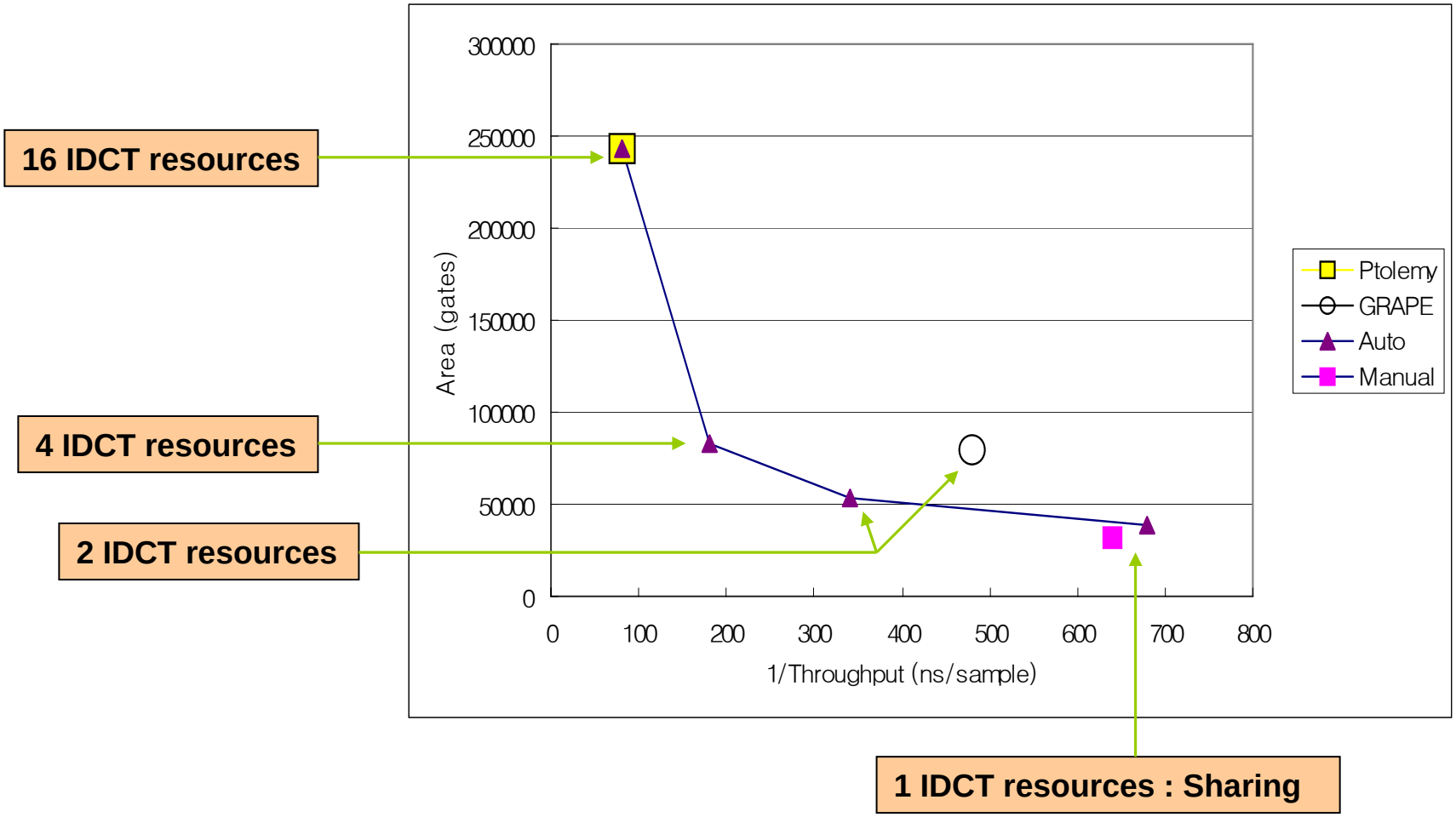


HW Controller Generation

- **Counter-based Controller**
  - Buffer control, Mux control, start and done signal of block



Experiment 1 : 2-dimensional DCT Algorithm



codesign environment

Contents

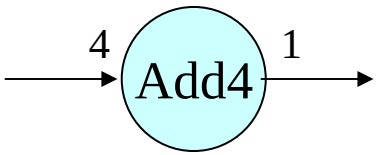
- **Introduction**
- **Previous works and our contributions**
- **Block Definition**
- **Controller Synthesis**
- **Schedule-Based Design**
- **FRDF Specification for more efficient HW implementation**
  - FRDF model
  - Examples & Experiments
- **Conclusions & Future Directions**

Fractional Rate Dataflow Specification

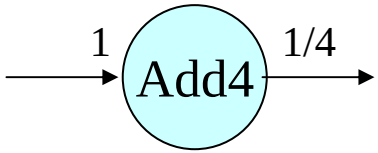
- **The gap between automatic and manual design still exists.**
  - We cannot optimize the automatic design further because of dataflow semantic.
  - Dataflow semantic has more strict rules for firing.
    - *This requires **more buffers** to satisfy firing condition.*
  - Real design has more freedom of implementation for efficient design
- **It is necessary to reduce the buffer requirements for practical efficient design**
  - We choose FRDF in which fractional number of data samples can be produced or consumed.
  - FRDF makes the automatic design a little closer to the manual design.

Fractional Rate Dataflow Specification

- **Every block with multi-rate specification has its equivalent block with FRDF specification.**
  - Functionally equivalent
  - Internal algorithm and its schedule can be different.



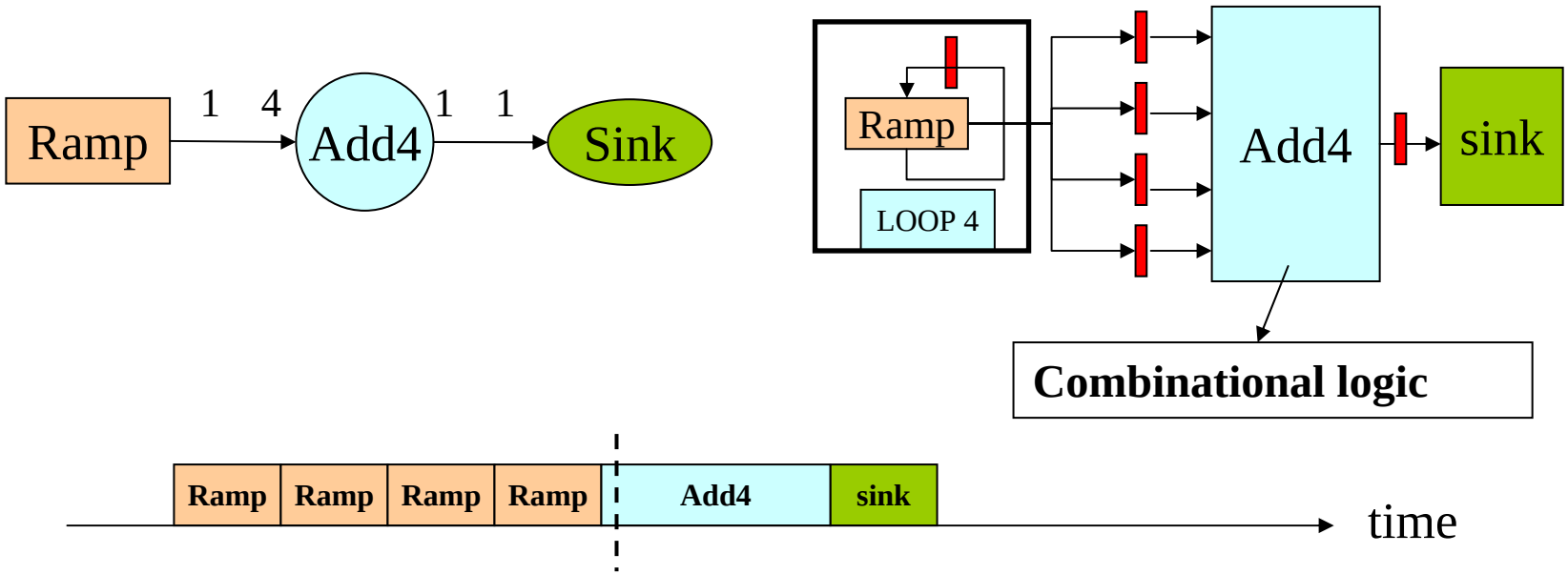
- In one invocation (or firing),
  - Comsumes 4 input data samples
  - Produces 1 output data sample
- Requires 4 input buffers



- Consumes 1 input data sample in one invocation
- Produces only 1 output data sample during 4 invocations
- Requires only 1 input buffers
- Requires 4 invocations to perform entire function

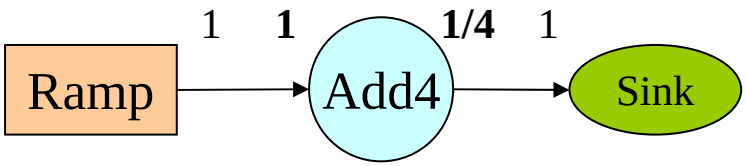
Fractional Rate Dataflow(FRDF)

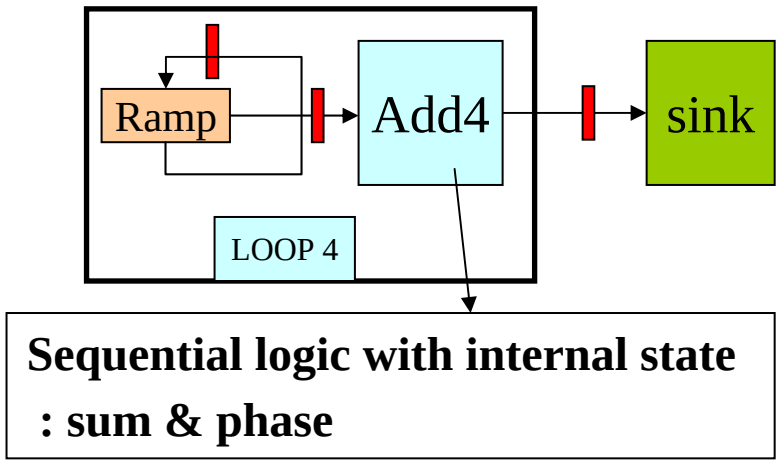
- **Non-FRDF implementation**
  - *"Add4"* block is invoked after its all inputs are valid.
  - Parallel I/O

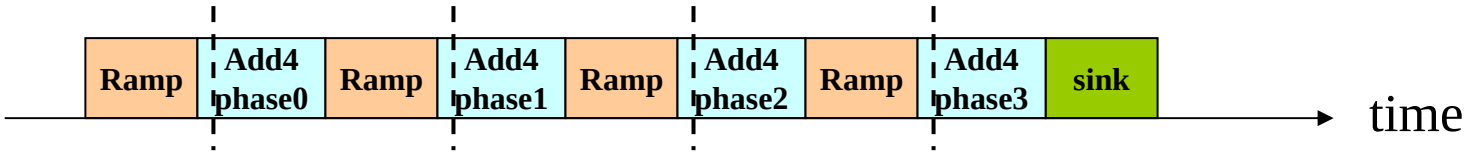


Fractional Rate Dataflow(FRDF)

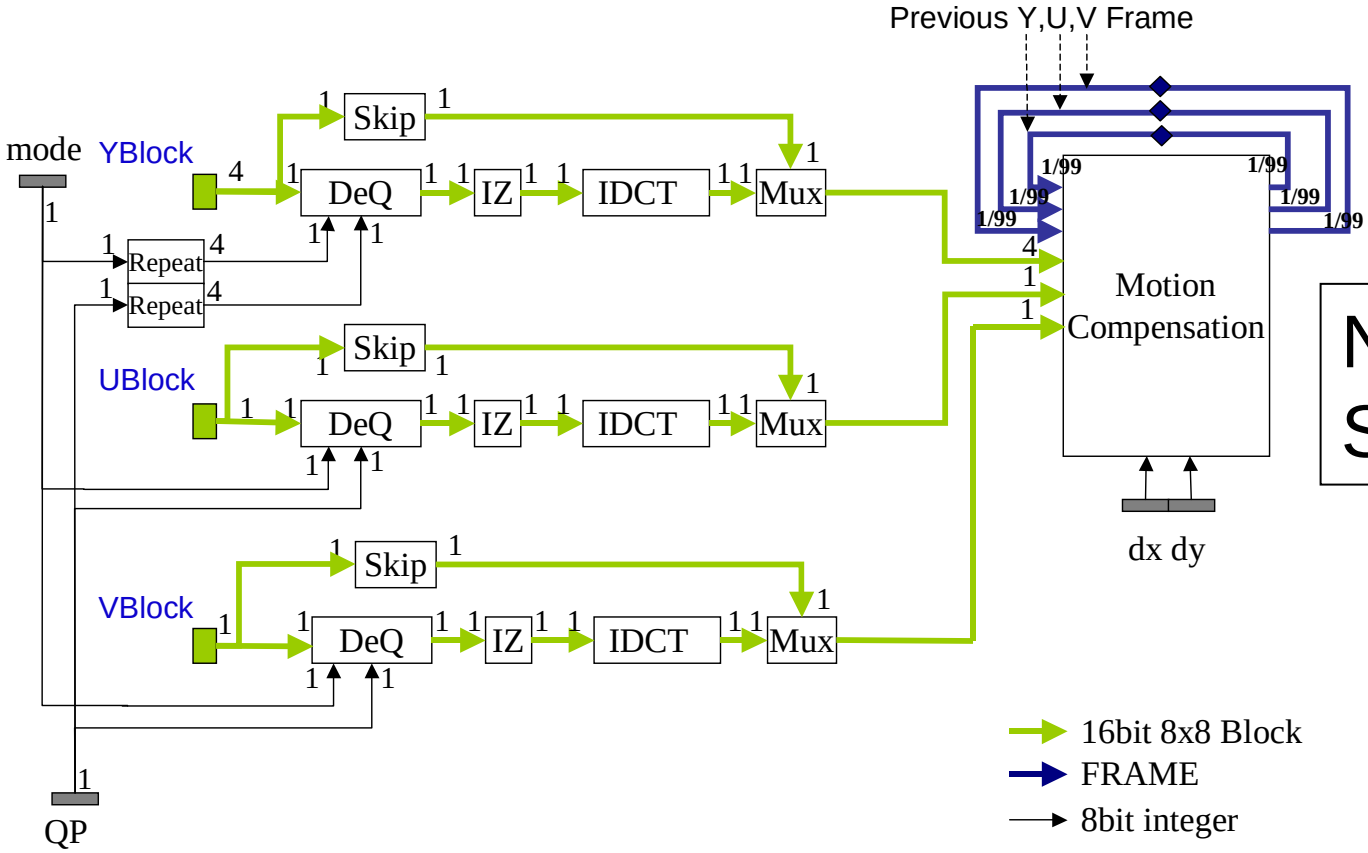
- **FRDF implementation**
  - The execution of block “Add4” is divided into 4 phases.
  - Serial I/O at each phase







Experiment 2 : Parts of H.263 Decoder

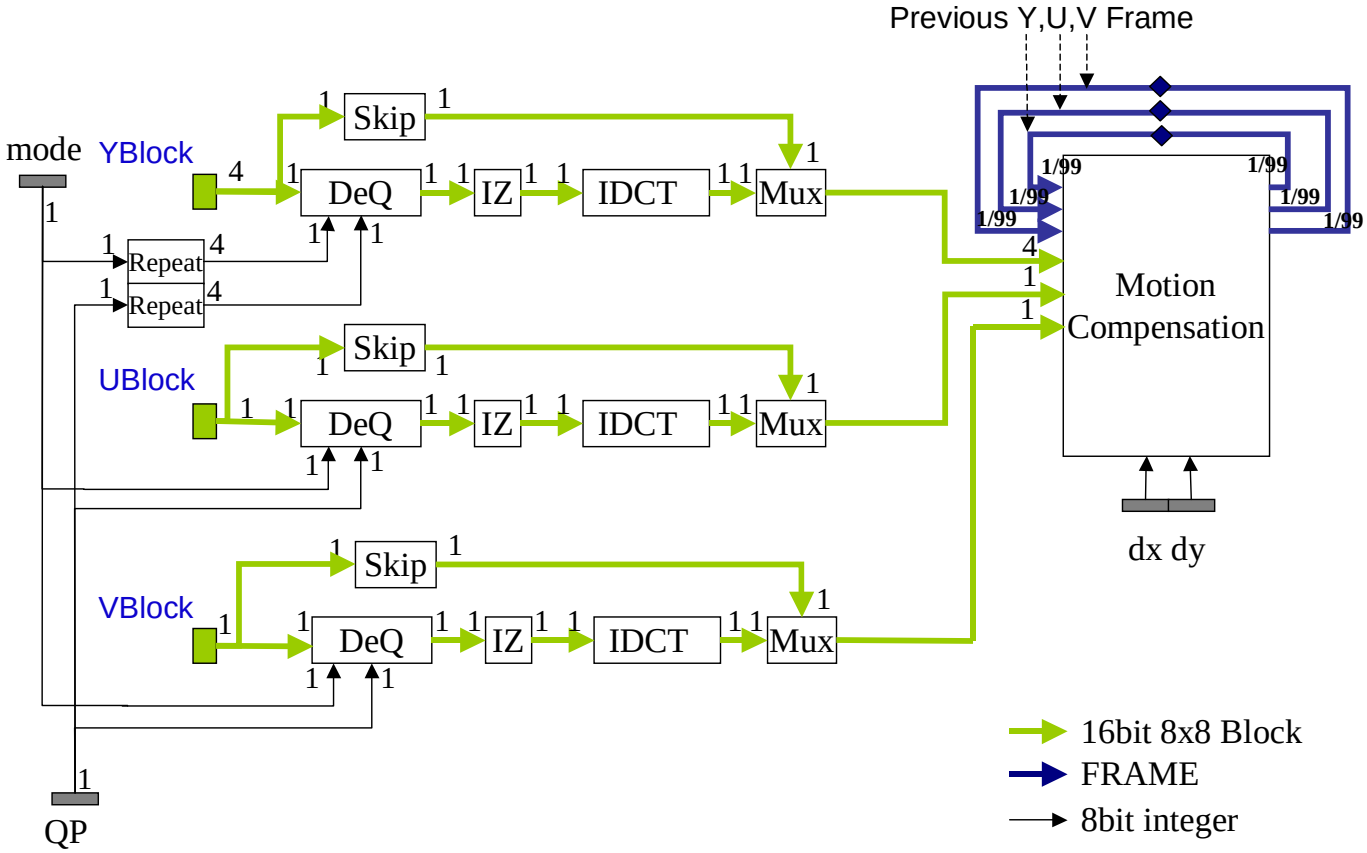


No Resource Sharing

Core: 282383, Buffer: 172032, Glue logic: 52575

Total Area: 506,987 gates

Experiment 2 : Parts of H.263 Decoder

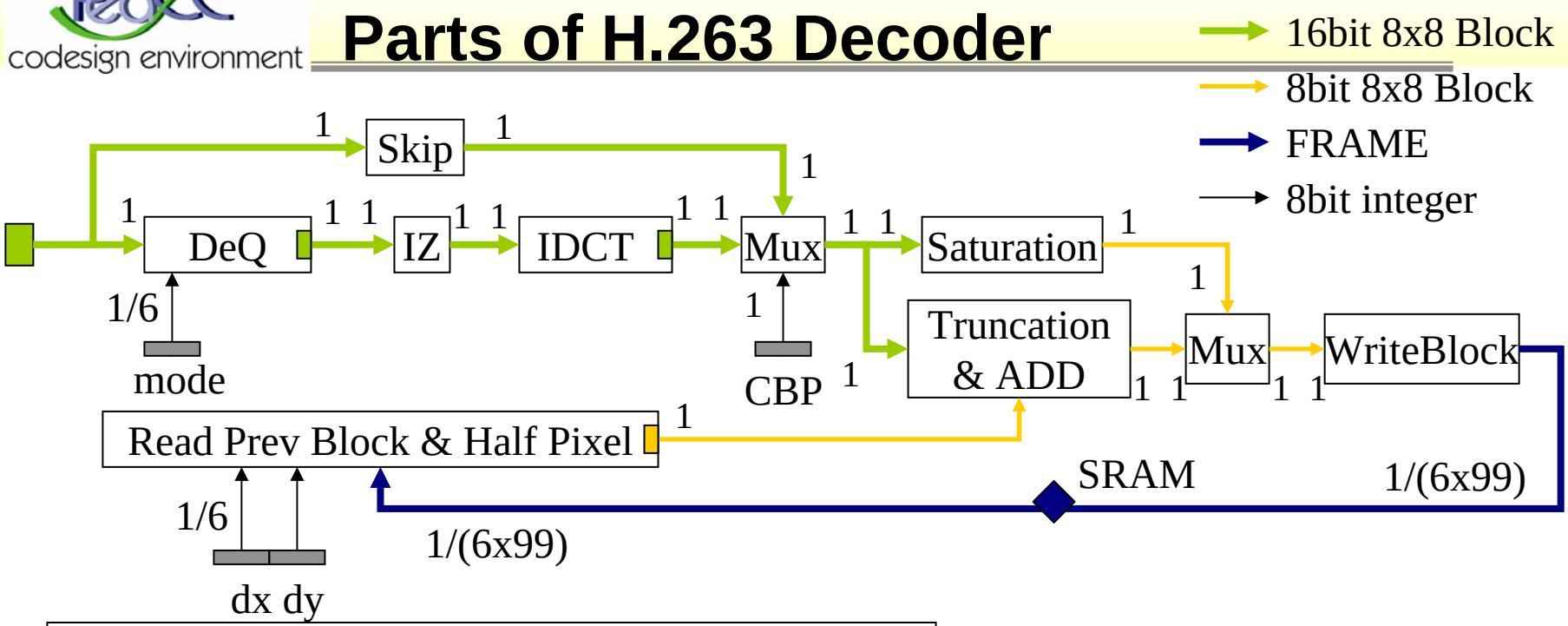


Maximum Resource Sharing

Core: 161164, Buffer: 172032, Glue logic: 66304

Total Area: 399,500 gates

Experiment 2 : Parts of H.263 Decoder

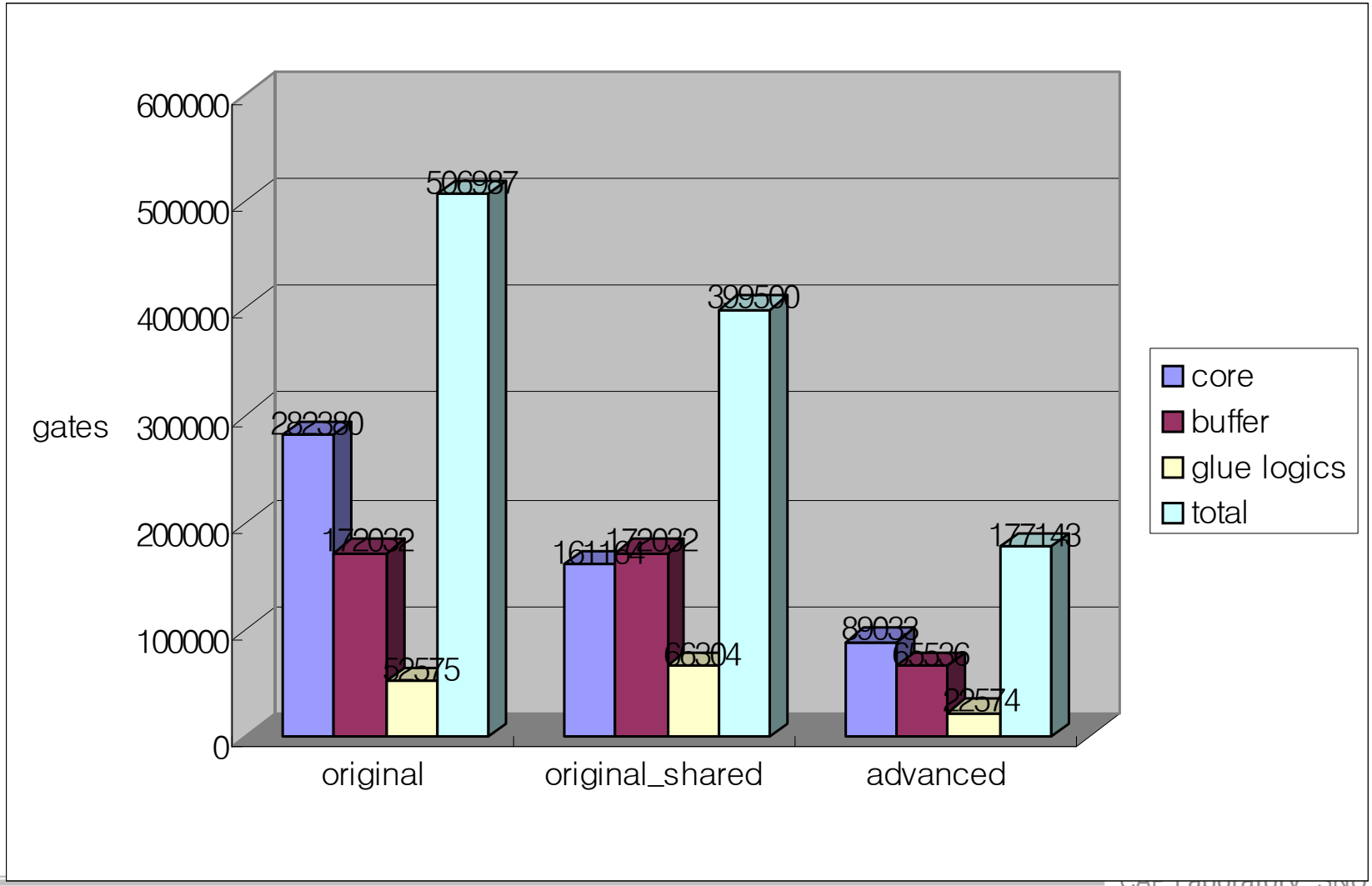


- **More fractional rate specification**
  - Separate Y, U, and V data paths are merged
  - MC block is divided into several small blocks for FRDF

Core: 89033, Buffer: 65536, Glue logic: 22574

Total Area: 177,143 gates

Experiment 2 : Parts of H.263 Decoder



codesign environment

Contents

- Introduction
- Previous works and our contributions
- Block Definition
- Controller Synthesis
- Schedule-Based Design
- FRDF Specification for more efficient HW implementation
- Conclusions & Future Directions

Conclusions

- **Synthesize efficient hardware from dataflow specification.**
  - We use SDF and its extension to FRDF.
- **The main goal of our research**
  - Overcoming the limitations of previous approaches
    - *Solving non-deterministic timing of I/O*
    - *Schedule-based design: resource sharing, looping control*
  - Efficient hardware synthesis applicable to practical HW design
    - *Supporting FRDF specification*
- **All of these techniques are implemented in VHDL domain of PeaCE codesign environment and verified by some examples; DCT and H.263 decoder**

Future work

- **Extension of expressiveness**
  - Piggybacked dataflow
  - Dynamic construct : for (data-dependent iteration), case (if-then-else, conditional execution)
- **Support of legacy HW platforms**
  - Support legacy HW IP ← SW code generation
  - Support various types of BUS & memory interface
    - *Local SRAM, dual-port memory, shared memory*
  - Current : Shared memory through AMBA interface
- **Optimization issues**
  - Buffer elimination
  - Buffer sharing
  - FRDF