The Design of Low-Latency Interfaces for Mixed-Timing Systems

Tiberiu Chelcea and Steven M. Nowick

Department of Computer Science
Columbia University

Keynote Invited Talk: “GALS” Session
IEEE Workshop on Complexity-Effective Design (ISCA-02)
May 26, 2002
Trends and Challenges

Trends in Chip Design: next decade

- "Semiconductor Industry Association (SIA) Roadmap" (97-8)

Unprecedented Challenges:

- complexity and scale (= size of systems)
- clock speeds
- power management
- reusability & scalability
- “time-to-market”

Design becoming unmanageable using a centralized single clock (synchronous) approach....
1. **Clock Rate:**

- **1980:** several MegaHertz
- **2001:** ~750 MegaHertz - 1+ GigaHertz
- **2004:** several GigaHertz

**Design Challenge:**

- "*clock skew*": clock must be near-simultaneous across entire chip
2. Chip Size and Density:

Total #Transistors per Chip: \textit{60-80\% increase/year}

- \textit{\sim1970}: 4 thousand (Intel 4004)
- \textit{today}: 10-100+ million
- \textit{2004 and beyond}: 100 million-1 billion

Design Challenges:

- system complexity, design time, clock distribution
- \textit{clock will not reach across chip in 1 cycle}
3. Power Consumption

- Low power: ever-increasing demand
  - consumer electronics: battery-powered
  - high-end processors: avoid expensive fans, packaging

Design Challenge:

- clock inherently consumes power continuously
- “power-down” techniques: only partly effective
Trends and Challenges (cont.)

4. Time-to-Market, Design Re-Use, Scalability

Increasing pressure for faster "time-to-market". Need:

- **reusable components:** "plug-and-play" design
- **scalable design:** easy system upgrades

Design Challenge: mismatch w/ central fixed-rate clock
5. Future Trends: “Mixed Timing” Domains

Chips themselves becoming *distributed systems*....

* contain many sub-regions, *operating at different speeds:*

Design Challenge: breakdown of single centralized clock control
Example: System-on-a-Chip (SoC) Design

- Building entire large-scale system on a single chip

Benefit: Higher-level of integration
  - Improved performance, cost, area

Challenges:
  - Mixed-timing: moving to multiple timing domains
  - Performance degradation: synchronization overhead
  - Complexity, scale, integration
  - Designing & incorporating of asynchronous subsystems
Goal #1: interface mixed-timing domains with low latency
Goal #2: synthesis + optimization of asynchronous systems
Two key issues not yet completely addressed:

1. **Communication** between mixed-timing domains:
   * Goals: performance and scalability

2. **Synthesis** of large-scale asynchronous systems:
   * Goals: develop powerful optimizing CAD tools, facilitating “design-space exploration”
Asynchronous Design: Motivation

Need for large-scale asynchronous systems:

- **Future chips**: likely a mix of async and sync domains

Asynchronous Systems: offer a number of advantages

GALS: “globally-asynchronous, locally-synchronous”

- **Hybrid style**: introduced by Chapiro [84]
  - synchronous “processing elements” (“satellites”)
  - asynchronous communication

- **Recent interest**: “Communication-Based Design”
  - UC Berkeley/Stanford: W. Dally, K. Keutzer, A. Sangiovanni
  - **orthogonalization of concerns**: function vs. communication
Asynchronous Design: Potential Advantages

- **Modularity:**
  - Interface easily with sync domains & environment

- **Reusability and scalability:**
  - Handle wide range of interface speeds ⇒ reuse
  - Scalability: easily add new subsystems

- **Average-case performance:**
  - Intel RAPPID instruction-length decoder: 3-4x faster than sync design
  - Differential equation solver: 1.5x faster than sync design

- **Lower power consumption:**
  - Avoids clock distribution power
  - Provides automatic “clock gating” ... at arbitrary granularity
  - Digital hearing aid chip: 4-5.5x less power

- **Low electromagnetic interference (EMI):** no regular clock spikes
  - Philips, commercial 80c51 microcontrollers: in cell phones, pagers

**Industrial interest:** Intel, Sun, IBM, Philips, Theseus, Fulcrum
Related Work #1: Interfacing in Single Clock Domain

Handling Timing Discrepancies…:

Clock Skew:

- **STARI Chip** [M. Greenstreet, ICCD-95]
  
  Use *async buffer* to smooth out discrepancies between *sender* and *receiver*

- **Skew-Tolerant Domino** [M. Horowitz]

- **Clock-Skew Scheduling** [E. Friedman]

- **Long interconnect delays** [Carloni99]: limited to single clock

Long Interconnect Delays:

- **“Relay Stations”** [Carloni, Sangiovanni-Vincentelli, DAC-00]
  
  Break up overlong wires by *pipelining communication*
Two common approaches...:

- **Modify Receiver’s Clock:**
  - “stretchable” and “pausable” clocks
  - Chapiro84, Yun96, Bormann97, Sjogren/Myers97, Moore02
  - **drawbacks:**
    - Penalties in restarting clock
    - Does not support design reuse

- **Use Synchronization Components:**
  - data/control synchronization
  - Seitz80, Seizovic94, Intel97, Sarmenta95, Kol98
  - **drawbacks:** overheads in throughput, latency, area
A complete family of mixed-timing FIFO’s

Characteristics:

- Low-latency
- Modular and scalable:
  - Define interfaces for each combination of:
    - Synchronous or Asynchronous domains
    - Combine interfaces to design new async-sync FIFO’s
- High throughput:
  - In steady state: no synchronization overhead, no failure probability
  - Enqueue/Dequeue data items: one/cycle
- Low area overheads

Also, solve issue of long interconnect delays between domains
**Contribution: Mixed-Timing Interfaces**

**Publications**

**Latest Solution:**


*IEEE/ACM Design Automation Conference (DAC, June 2001)*


**Initial Solution:**

*IEEE Computer Society Workshop on VLSI (WVLSI, April 2000)*

T. Chelcea and S.M. Nowick, “A Low-Latency FIFO for Mixed-Clock Systems”

**See also:**

A. Iyer and D. Marculescu, *ISCA-02*. 
I. Mixed-Timing Interface Circuits
   - Sync/Sync
   - Async/Async
   - Async/Sync

II. Handling Long Interconnect Delays

Experimental Results

Conclusions
Part I

Mixed-Timing Interface Circuits
Problem: potential data synchronization errors
Mixed-Timing Interfaces: Overview

Problem: potential data synchronization errors
Solution: insert mixed-timing FIFO’s ⇒ safe data transfer
Mixed-Clock FIFO: Block Level

- synchronous put interface:
  - req_put
  - data_put
  - CLK_put

- Mixed-Clock FIFO:
  - full
  - req_get
  - valid_get
  - empty
  - data_get
  - CLK_get

- synchronous get interface:
Mixed-Clock FIFO: Block Level

- Initiates put operations
- Initiates get operations

Bus for data items

synchronous put interface

synchronous get interface

Mixed-Clock FIFO

- full
- req_put
- data_put
- CLK_put
- req_get
- valid_get
- empty
- data_get
- CLK_get

Controls put operations

Controls get operations
Mixed-Clock FIFO: Block Level

Indicates when FIFO full

Indicates data items validity (always 1 in this design)

Synchronous put interface

Synchronous get interface

Indicates when FIFO empty
Mixed-Clock FIFO: Architecture

Array of identical cells

Token Ring Architecture
Mixed-Clock FIFO: Architecture

Put Interface

Common Data/Control Buses for put interface

Put Controller

Full Detector

cell
cell
cell
cell
cell

data_put
CLK_put
req_put

CLK_get
data_get
req_get
valid_get
empty

Empty Detector
Mixed-Clock FIFO: Architecture

Put Token Ring: used to enqueue data items
Cell with put token = tail of queue

Cell with put token = tail of queue
**Mixed-Clock FIFO: Architecture**

**Put Controller:**
- enables & disables put operations
- stalls put interface when FIFO full

**Full Detector:**
- detects when FIFO full

The diagram illustrates the architecture of a mixed-clock FIFO, showing the interactions between the put controller, full detector, and various cells within the FIFO. Key signals include `req_put`, `data_put`, `CLK_put`, `CLK_get`, `data_get`, `req_get`, `valid_get`, and `empty`. The put controller manages put operations, while the full detector indicates when the FIFO is full. The diagram also shows the flow of data and control signals through the FIFO cells.
Mixed-Clock FIFO: Architecture

Get Token Ring

Cell with get token = head of queue

Get Token: used to dequeue data items

Get Interface
Mixed-Clock FIFO: Architecture

Get Controller:
- enables & disables get operations
- stalls get interface when FIFO empty

Empty Detector:
detects when FIFO empty
Mixed-Clock FIFO: Cell Implementation

The diagram illustrates the mixed-clock FIFO cell implementation. It shows the interaction between different components such as the En, SR, and REG blocks, along with signals like CLK_put, en_put, req_put, data_put, ptok_out, ptok_in, f_i, e_i, CLK_get, en_get, valid, and data_get. The diagram highlights the flow of data and signals within the FIFO cell.
Mixed-Clock FIFO: Cell Implementation
Mixed-Clock FIFO: Cell Implementation

**PUT INTERFACE**
- CLK_put
- en_put
- valid
- data_put
- ptok_out
- en
- f_i
- e_i
- SR
- REG

**GET INTERFACE**
- CLK_get
- en_get
- valid
- data_get
- gtok_out
- en
- gtok_in

Enables put operation
Data Bus: item in

Enables get operation
Data Bus: item out
Mixed-Clock FIFO: Cell Implementation

Status Bits:

Cell FULL → f_i
Cell EMPTY → e_i

CLK_put

en_put
req_put
data_put

ptok_out

En

SR

REG

CLK_get

en_get
valid
data_get

ptok_in
gtok_out

gtok_in
Mixed-Clock FIFO: Cell Implementation

Token Passing:
Mixed-Clock FIFO Cell: Put Operation

Simulation #1: Put Operation

Cell Has Put Token:

- `CLK_put`
- `en_put`
- `req_put`
- `data_put`
- `ptok_in = 1`

- `CLK_get`
- `en_get`
- `valid`
- `data_get`

- `ptok_out`
- `f_i`
- `e_i`
- `gtok_out`
- `gtok_in`
Mixed-Clock FIFO Cell: Put Operation

Put Request Arrives:

- en_put
- valid
- data_put

CLK_put

ptok_out → En

f_i e_i → SR

CLK_get

en_get valid data_get

ptok_in

gtok_out → En

gtok_in
Mixed-Clock FIFO Cell: Put Operation

```
\text{En} \quad \text{ptok}_{\text{out}} \quad \text{CLK}_{\text{put}} \quad \text{en}_{\text{put}} \quad \text{valid} \quad \text{data}_{\text{put}}
```

```
\text{f}_{\text{I}} = 1 \quad \text{e}_{\text{i}} \quad \text{SR} \quad \text{REG} \quad \text{ptok}_{\text{in}}
```

```
\text{En} \quad \text{clk}_{\text{get}} \quad \text{en}_{\text{get}} \quad \text{valid} \quad \text{data}_{\text{get}}
```

"FULL CELL" Asserted

Data Latch Enabled:

```
gtok_{\text{out}} \quad \text{gtok}_{\text{in}}
```
Mixed-Clock FIFO Cell: Put Operation

- **ptok_out = 1**
- **ptok_in = 0**

**NEXT CLK:**
- Token Passed
- Data Latched

**CLK_put**
- \( en_{put} \)
- \( req_{put} \)
- \( data_{put} \)

**CLK_get**
- \( en_{get} \)
- \( valid \)
- \( data_{get} \)

**Next CLK:**
- **Token Passed**
- **Data Latched**
Mixed-Clock FIFO Cell: Get Operation

Simulation #2: Get Operation
Mixed-Clock FIFO Cell: Get Operation

- **CLK_put**: Clock signal for put operation.
- **en_put**: Enable signal for put operation.
- **req_put**: Request signal for put operation.
- **data_put**: Data signal for put operation.
- **ptok_in**: Put token input.
- **ptok_out**: Put token output.
- **f_i**, **e_i**: Input signals for SR.
- **SR**: SR (Set-Reset) flip-flop.
- **CLK_get**: Clock signal for get operation.
- **en_get**: Enable signal for get operation.
- **valid**: Valid signal.
- **data_get**: Data signal for get operation.
- **gtok_out**: Get token output.
- **gtok_in**: Get token input.

**Cell Has Get Token**

Note: The diagram illustrates the flow of signals and states in a mixed-clock FIFO cell, focusing on the get operation. The labels such as `En`, `req_put`, `data_put`, `ptok_in`, `f_i`, `e_i`, `SR`, `CLK_get`, `en_get`, `valid`, `data_get`, and `gtok_in` represent the key components and signals involved in the get operation of a mixed-clock FIFO cell.
Mixed-Clock FIFO Cell: Get Operation

Get Request Arrives

ptok_out

En

CLK_put
en_put
req_put
data_put
ptok_in

f_i
e_i

SR

CLK_get
en_get
valid
data_get

gtok_out

gtok_in = 1

Get Request Arrives
Mixed-Clock FIFO Cell: Get Operation

- **EN**
  - $f_I = 0$
  - $e_I = 1$

- **SR**
  - "EMPTY CELL" Asserted

- **REG**
  - $gtok_{in} = 1$
  - $gtok_{out}$

- **Tri-State Buffers**
  - Enabled

- **Connections**
  - $ptok_{out}$
  - $ptok_{in}$
  - $CLK_{put}$
  - $en_{put}$
  - $req_{put}$
  - $data_{put}$
  - $CLK_{get}$
  - $en_{get}$
  - $valid$
  - $data_{get}$
Mixed-Clock FIFO Cell: Get Operation

- **f_I = 0**
- **e_I = 1**

Data Broadcast on Get Bus
Mixed-Clock FIFO Cell: Get Operation

- **f_I = 0**
- **e_I = 1**
- **ptok_out = 1**
- **gtok_out = 0**
- **clk_get**
- **en_get**
- **valid**
- **data_get**

**NEXT CLK:**

Token Passed
**Synchronization Issues: Overview**

**Challenge:** highly concurrent behavior
- Global FIFO state controlled by two different clocks

**Problem #1:** Metastability
- Each FIFO interface needs clean state signals

**Solution #1:** Synchronize “full” & “empty” signals
- “full” with CLK_put
- “empty” with CLK_get

Add 2 synchronizing latches each
**Problem #2:** FIFO now may underflow/overflow!

- synchronizing latches add extra latency

**Solution #2:** Change Full/Empty definitions

New FULL: 0 or 1 empty cells left
New EMPTY: 0 or 1 full cells left

Observable full/empty *safely approximate* FIFO’s state
Mixed-Clock FIFO: Full/Empty Detectors

Problem #2: FIFO now may underflow/overflow!
- synchronizing latches add extra latency

Solution #2: Change Full/Empty definitions

New FULL: 0 or 1 empty cells left
New EMPTY: 0 or 1 full cells left

New Full Detector

Observable full/empty safely approximate FIFO's state
Problem #2: FIFO now may underflow/overflow!

* synchronizing latches add extra latency

Solution #2: Change Full/Empty definitions

New FULL: 0 or 1 empty cells left
New EMPTY: 0 or 1 full cells left

New Full Detector

Observable full/empty safely approximate FIFO’s state
Mixed-Clock FIFO: Full/Empty Detectors

**Problem #2:** FIFO now may underflow/overflow!
- Synchronizing latches add extra latency

**Solution #2:** Change Full/Empty definitions
- New FULL: 0 or 1 empty cells left
- New EMPTY: 0 or 1 full cells left

New Full Detector

Observable full/empty *safely approximate* FIFO’s state
**Deadlock Avoidance**

**Problem #3:** potential for **deadlock**

**Scenario:** only 1 data item in FIFO

- FIFO **still** considered “empty” (new definition)
- Get interface: *cannot dequeue item!*

**Solution #3:** **bi-modal empty detector**

- "**New empty**” detector (0 or 1 data items)
- "**True empty**” detector (0 data items)

Combine **two** results into single global **“empty”**
Mixed-Clock FIFO: Deadlock Avoidance

CLK_get

f_0
f_1
f_2
f_3
CLK_get

f_0
f_1
f_2
f_3
CLK_get

CLK_get

ne
empty
en_get
oe
req_get
Mixed-Clock FIFO: Deadlock Avoidance

Detects “new empty” (0 or 1 empty cells)

Detects “true empty” (0 empty cells)

Combine into global “empty”
Mixed-Clock FIFO: Deadlock Avoidance

Bi-modal empty detection: select either ne or oe

Reconfigure whenever active get interface
Mixed-Clock FIFO: Deadlock Avoidance

Bi-modal empty detection:

When reconfigured, use “ne”:
FIFO active ⇒ avoids underflow

Reconfigure whenever active get interface
**Mixed-Clock FIFO: Deadlock Avoidance**

**Bi-modal empty detection:**

When NOT reconfigured, use “oe”:

FIFO quiescent $\Rightarrow$ avoids deadlock
Related Work: Intel Mixed-Clock Synchronizer

Intel Patent [1997]: J. Jex, C. Dike, K. Self (5,598,113)
- Similar FIFO structure
- Similar notion of “almost full”/”almost empty”

Differences/Limitations: N-stage FIFO

# synchronizers required:
- INTEL: N+1
- US: 3

Interface types:
- INTEL: only sync-sync
- US: introduce a complete family (sync+async combinations)
Async-Async FIFO: Architecture

cell | cell | cell | cell | cell | cell

put_ack -> put_req -> put_data

data_get -> get_req -> get_ack
Async-Async FIFO: Architecture

Asynchronous Put Part

put_ack → put_req → put_data

cell → cell → cell → cell → cell → cell

data_get → get_req → get_ack
Async-Async FIFO: Architecture

Asynchronous Get Part
Async-Async FIFO: Architecture

**Put Interface:** 4-phase bundled data channel

- put_ack
- put_req
- put_data

**Get Interface:** 4-phase bundled data channel

- data_get
- get_req
- get_ack
Async-Async FIFO: Architecture.

No Detectors or External Controllers

- put_ack
- put_req
- put_data
- data_get
- get_req
- get_ack

Cell connections diagram.
When FIFO full, acknowledgment withheld until safe to perform the put operation
Async-Async FIFO Cell

Asynchronous Put Part

Data Validity Controller

Asynchronous Get Part

we \rightarrow put\_req \rightarrow put\_data \rightarrow put\_ack

GC \rightarrow REG \rightarrow OPT \rightarrow OGT

gc \leftarrow get\_data \leftarrow get\_req \leftarrow get\_ack

we1 \rightarrow reusable

PC \rightarrow reusable

DV \rightarrow reusable

Data Validity Controller

Async-Async FIFO Cell
**Reusability: Async-Sync FIFO Architecture**

**Asynchronous Put Interface:** exactly as in **Async-Async FIFO**

- put_ack
- put_req
- put_data

**Synchronous Get Interface:** exactly as in **Mixed-Clock FIFO**
**Reusability: Async-Sync FIFO Cell**

Data Validity Controller

Asynchronous Put Part
- put_req
- put_data
- put_ack

Synchronous Get Part
- CLK_get
- get_data

**Controller**
- e_i
- DV
- En

**REG**
- OPT

**Reused**
- we
- gtok_in
- gtok_out
- CLK_get
- En
- put_ack
- put_req
- put_data

**New**
- f_i
- we_i

(from mixed-clock FIFO)

(from async-async FIFO)
Part II

Handling Long Interconnect Delays
Issues in Handling Long Interconnect

Relay Stations: Background [Carloni, Sangiovanni-Vincentelli ’99]

system 1 sends “data items” to system 2
Issues in Handling Long Interconnect

Relay Stations Background [Carloni’99]

Delay = > 1 cycle
Issues in Handling Long Interconnect

Relay Stations Background [Carloni’99]

system 1 now sends “data packets” to system 2
Issues in Handling Long Interconnect

Relay Stations Background [Carloni’99]

Delay = 1 cycle

Data Packet = data item + validity bit
Issues in Handling Long Interconnect

Relay Stations Background [Carloni’99]

Steady State: pass data on every cycle (either valid or invalid)
Issues in Handling Long Interconnect

Relay Stations Background [Carloni’99]

“stop” control = stopIn + stopOut
- apply counter-pressure
- result: stall communication

Problem: Works only for single-clock systems!
**Relay Station**

- **Steady state:** always pass data
- **Data items:** both valid & invalid
- **Stopping mechanism:** stopIn & stopOut

**Mixed-Clock FIFO**

- **Steady state:** only pass data when requested
- **Data items:** only valid data
- **Stopping mechanism:** none (only full/empty)
Mixed-Clock Relay Stations (MCRS)

Mixed-Clock Relay Station: derived from Mixed-Clock FIFO

Change ONLY Put and Get Controllers

packetIn
valid_put
data_put
CLK1

Mixed-Clock Relay Station
stopOut
valid_get
data_get
CLK2

packetOut
Part III

Experimental Results
Each new Mixed-Timing FIFO designed:

- using both academic and industry tools
  - **MINIMALIST**: Burst-Mode controllers [Nowick et al. ‘99]
  - **PETRIFY**: Petri-Net controllers [Cortadella et al. ‘97]

Pre-layout simulations in 0.6µm HP CMOS technology

Experiments:

- various FIFO capacities (4/16 cells)
- 8-bit data items
Preliminary Results: Latency

Experimental setup: 8-bit data items + various FIFO capacities (4, 16)

Latency = time from enqueuing to dequeueing data into an empty FIFO

<table>
<thead>
<tr>
<th>Version</th>
<th>4-place</th>
<th></th>
<th>16-place</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>Mixed-Clock FIFO</td>
<td>5.43</td>
<td>6.34</td>
<td>6.14</td>
<td>7.17</td>
</tr>
<tr>
<td>Async-Async FIFO</td>
<td>1.73</td>
<td></td>
<td>2.29</td>
<td></td>
</tr>
<tr>
<td>Async-Sync FIFO</td>
<td>5.53</td>
<td>6.45</td>
<td>6.47</td>
<td>7.51</td>
</tr>
<tr>
<td>Sync-Async FIFO</td>
<td>1.95</td>
<td></td>
<td>2.44</td>
<td></td>
</tr>
<tr>
<td>Mixed-Clock RS</td>
<td>5.48</td>
<td>6.41</td>
<td>6.23</td>
<td>7.28</td>
</tr>
<tr>
<td>Async-Sync RS</td>
<td>5.61</td>
<td>6.35</td>
<td>6.57</td>
<td>7.62</td>
</tr>
<tr>
<td>Sync-Async RS</td>
<td>1.86</td>
<td></td>
<td>2.43</td>
<td></td>
</tr>
</tbody>
</table>

Sync receiver ⇒ latency not uniquely defined: Min/Max
**Preliminary Results: Latency**

Experimental setup: 8-bit data items + various FIFO capacities (4, 16)

Latency = time from enqueuing to dequeueing data into an empty FIFO

<table>
<thead>
<tr>
<th>Version</th>
<th>4-place</th>
<th>16-place</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>Mixed-Clock FIFO</td>
<td>5.43</td>
<td>6.34</td>
</tr>
<tr>
<td>Async-Async FIFO</td>
<td>1.73</td>
<td></td>
</tr>
<tr>
<td>Async-Sync FIFO</td>
<td>5.53</td>
<td>6.45</td>
</tr>
<tr>
<td>Sync-Async FIFO</td>
<td>1.95</td>
<td></td>
</tr>
<tr>
<td>Mixed-Clock RS</td>
<td>5.48</td>
<td>6.41</td>
</tr>
<tr>
<td>Async-Sync RS</td>
<td>5.61</td>
<td>6.35</td>
</tr>
<tr>
<td>Sync-Async RS</td>
<td>1.86</td>
<td></td>
</tr>
</tbody>
</table>

Async receiver ⇒ lower, unique latency, no synchronization
### Preliminary Results: Maximum Operating Rate

Synchronous interfaces: MegaHertz  
Asynchronous interfaces: MegaOps/sec

<table>
<thead>
<tr>
<th>Design</th>
<th>4-place</th>
<th></th>
<th>16-place</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Put</td>
<td>Get</td>
<td>Put</td>
<td>Get</td>
</tr>
<tr>
<td>Mixed-Clock FIFO</td>
<td>565</td>
<td>549</td>
<td>505</td>
<td>484</td>
</tr>
<tr>
<td>Async-Async FIFO</td>
<td>423</td>
<td>454</td>
<td>359</td>
<td>357</td>
</tr>
<tr>
<td>Async-Sync FIFO</td>
<td>421</td>
<td>549</td>
<td>357</td>
<td>484</td>
</tr>
<tr>
<td>Sync-Async FIFO</td>
<td>565</td>
<td>454</td>
<td>505</td>
<td>360</td>
</tr>
<tr>
<td>Mixed-Clock RS</td>
<td>580</td>
<td>539</td>
<td>509</td>
<td>475</td>
</tr>
<tr>
<td>Async-Sync RS</td>
<td>421</td>
<td>539</td>
<td>357</td>
<td>475</td>
</tr>
<tr>
<td>Sync-Async RS</td>
<td>580</td>
<td>454</td>
<td>509</td>
<td>360</td>
</tr>
</tbody>
</table>

Put vs. Get rates:  
- sync put faster than sync get  
- async put slower than async get

Async vs. Sync rates:  
- async slower than sync
Conclusions

Introduced complete family of mixed-timing FIFO’s:

- sync-sync, async-asyn, async-sync, sync-asyn
- create FIFO’s from *reusable parts*
- extend to handle *issue of long interconnect delays*

Characteristics:

- **Low-latency**
- **Modular and scalable:** distributed token-ring architecture
- **High throughput:**
  - steady state: *no synchronization overhead, no failure probability*
  - enqueue/dequeue data items: one/cycle
- **Low area overheads:** simple design

Extensions:

- Deeper synchronizers (more latches) => arbitrary robustness
- powering down of inactive cells
On every clock cycle:
- packet latched in MR
- packet passed to next RS by end of clock cycle
- **steady state:** pass valid/invalid data items

Station stopped from right:
- stop_in = 1
- extra packet latched in AR
- stop_out = 1

Station started (stop_in = 0):
- Output packet in MR first
- Output packet in AR second
- Resume normal operation (stop_out = 0)