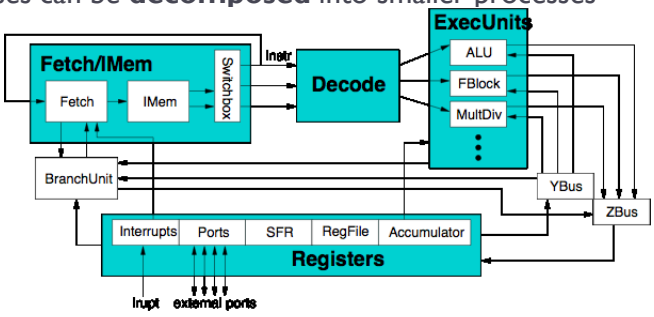
CE653 – Asynchronous Circuit Design

Instructor: C. Sotiriou

http://inf-server.inf.uth.gr/courses/CE653/

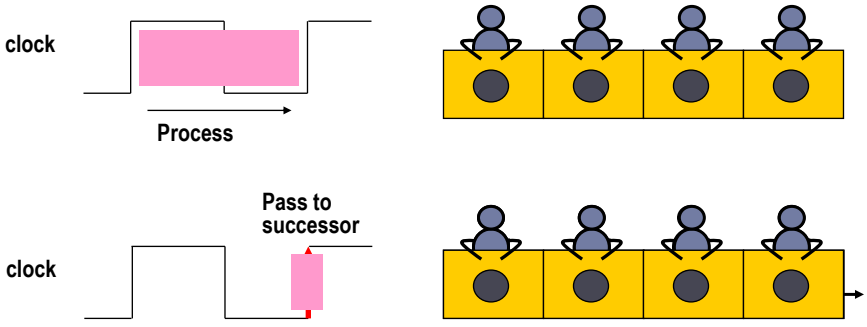
Hardware Abstraction

- System:
  - Collection of "**Processes**" linked by **Channels**
  - Channels pass messages with **guaranteed delivery**
  - Processes synchronize
  - Processes can be **decomposed** into smaller processes



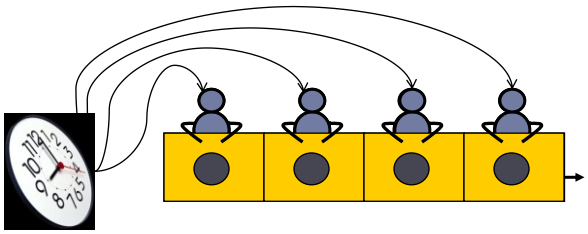
Synchronous Version

- In case of edge triggered stages
  - During the cycle: Process
  - At the edge of the clock: Pass to successor



Synchronous Version

- Central synchronizer
  - `SYNC(clk)



```
module  ff(clk, left, right)
input           clk;
input           left;
output reg      right;

always
begin
  right = left;
end
endmodule
```

Synchronous FF Stage

- Abstract synchronization
  - `SYNC(clk)

```
module ff(clk, left, right)
input     clk;
input              left;
output reg         right;

always @(posedge clk) begin
 right = left;
end
endmodule
```

```
module  ff(clk, left, right)
input           clk;
input           left;
output reg      right;

always
begin
  `SYNC(clk);
 right = left;
end
endmodule
```


Asynchronous Version

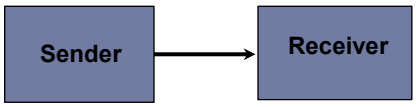
- Distributed Synchronization
- Sender
  - Provide data
  - Synchronize
- Receiver
  - Synchronize
  - Sample data

```
module Sender(right)
output reg      right;
output reg      data;

always
begin
 right = data;
[redacted]
end
endmodule
```

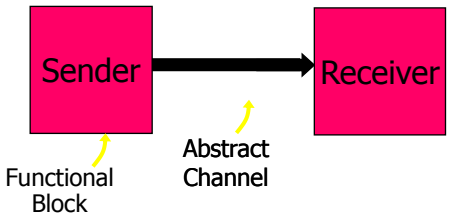
```
module Receiver(left)
input           left;
output reg      data;

always
begin
[redacted]
  data= left;
end
endmodule
```

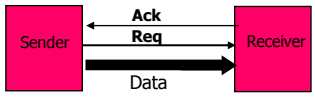


Asynchronous Channels

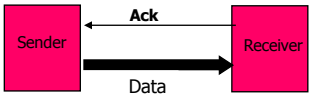
- Channel: A bundle of wires and a protocol for communicating data/control called a token
  - Data/control encoding: Dual-rail or single-rail
  - Communication protocol: Specific form of handshaking over request and acknowledgement wires



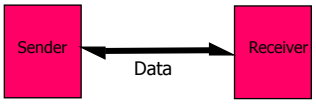
Asynchronous Channels



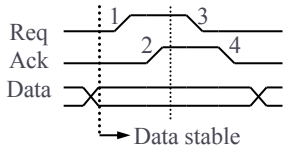
Bundled-Data Channel

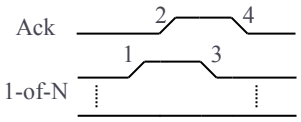


Dual-Rail (1-of-N) Channel

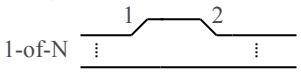


Single-Track Channel



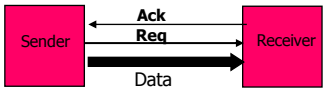


1 of the N wires is risen
(N-1 remains zero)

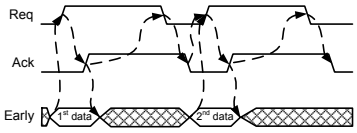


The s The receiver drives
1 of th 1 of the N wires low

Early, Late, Broad

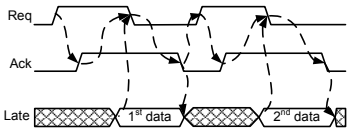


Bundled-Data Channel



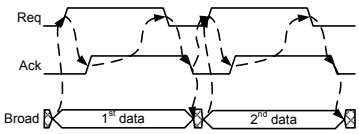
(a)

Narrow/Early:
- Data stable after Req+
- Data stable until Ack+



(b)

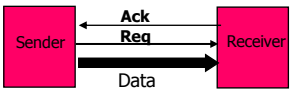
Late:
- Data stable after Req-
- Data stable until Ack-



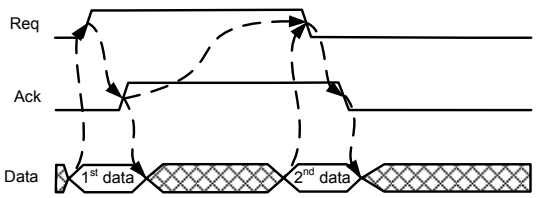
(c)

Broad:
- Data stable after Req+
- Data stable until Ack-

2-Phase Bundled-Data

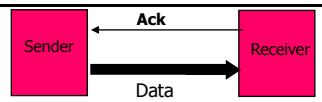


Bundled-Data Channel

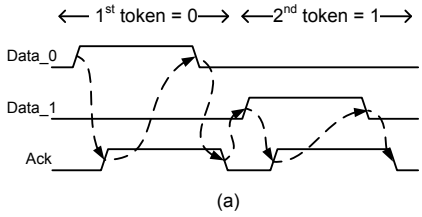


- Two-phase Bundled-Data Protocol
  - Both rising and falling transitions on *Req*
    - *Means new data is available*
  - Both rising and falling transitions on *Ack*
    - *Means data has been acknowledged*
  - Sometimes called *transition signalling*
    - It is the transition that is meaningful, not the values

1-of-N Protocols

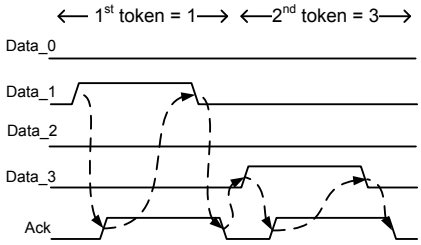


Dual-Rail (1-of-N) Channel



(a)

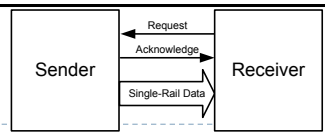
Dual-Rail
- 4 phase
- 2 wires per bit

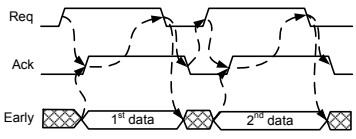


(b)

1-of-4
- 4 phase
- 4 wires per 2 bits

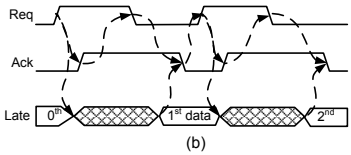
Pull Channels





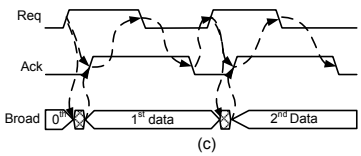
(a)

Early:
- Data stable after Ack+
- Data stable until Req-



(b)

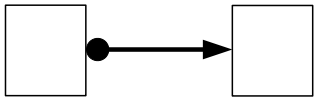
Late:
- Data stable after Ack-
- Data stable until Req+



(c)

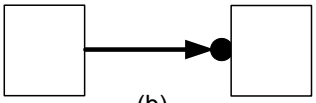
Broad:
- Data stable after Ack+
- Data stable until Req-

Abstract Channel Diagrams



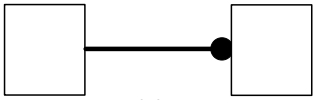
(a)

Push channel



(b)

Pull channel



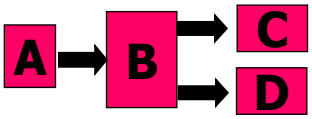
(c)

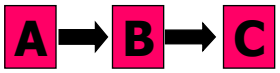
Nonput/Synchronization channel
- No data – Control only
- Active on right side

Handshaking details omitted

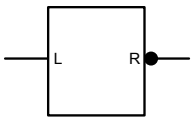
Sequencing and Concurrency

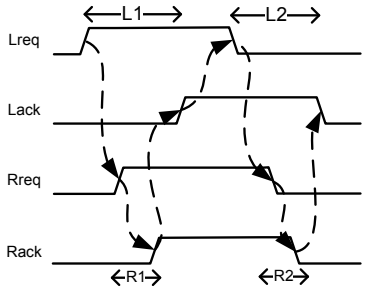
- Enclosed Handshaking
  - B completes handshake w/ C before starting handshake w/ D
    - Operation associated with C occurs before operation associated with D
  - B can enclose both handshakes in handshake w/ A
    - Completion of handshake w/ A is ack that C and D's task are done
- Pipelining Handshake
  - B overlaps handshake w/ C and handshake w/ A
    - Creates pipeline behavior
      - Tokens on both channels
    - Increases throughput





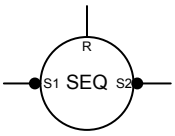
Enclosed Handshaking

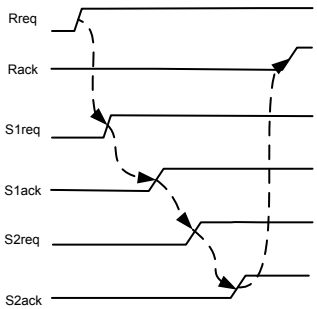




- Internal handshake on R represents the completion of some operation
- The enclosed handshake represents a "function call"
  - Initiated by the request on on L
  - Terminated by the acknowledgement on L

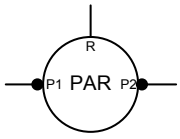
Sequencer

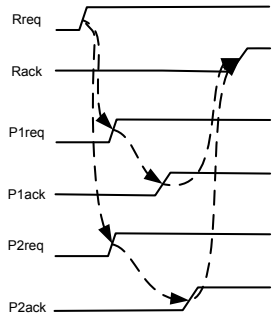




- Handshakes on S1 first then S2
  - Used to sequence operations associated with S1 and S2
- Both handshakes enclosed in handshake on R
  - Initiated by request on R
  - Terminated by acknowledgement of R

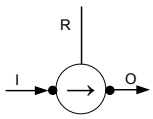
Parallel

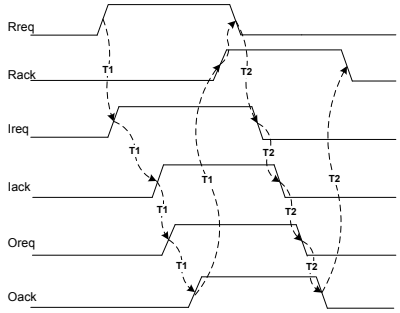




- Handshakes on P1 in parallel with P2
  - Used to execute operations associated with P1 and P2 in parallel
- Both handshakes enclosed in handshake on R
  - Initiated by request on R
  - Terminated by acknowledgement of R
    - *Both P1 and P2 must complete before R is acknowledged*

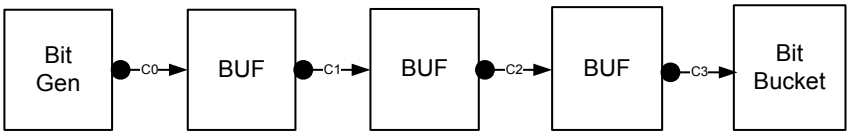
Transferer





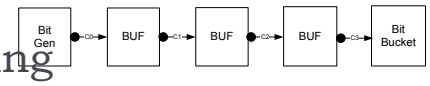
- Purpose
  - Pulls data from its input channel and pushes it onto its output channel
  - All enclosed in handshake on request channel R
- Operation
  - Waits for a request on its passive nonput port
  - Then initiates a handshake on its pull input port
  - The handshake on the pull input channel is relayed to the push output channel
  - Finally, it completes the handshaking on the passive nonput channel

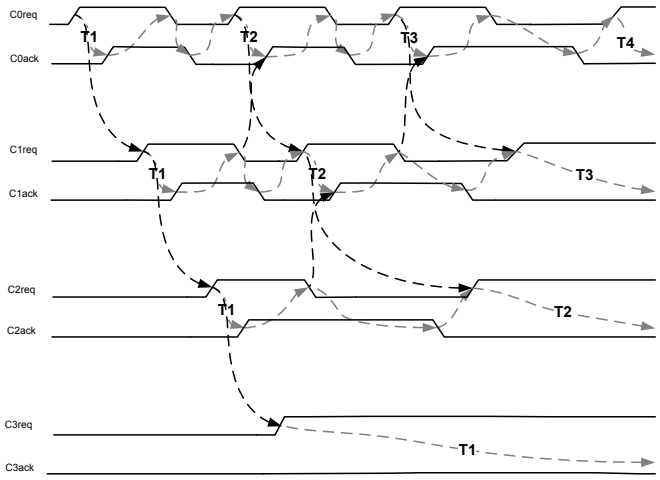
Pipelined Handshaking



- Pipeline handshaking enables multiple tokens to exist in pipeline
  - Each token represents intermediate result of different problem instance
  - Increases throughput of system
  - No tokens lost despite relative speed of stages – has *implicit* flow control
- Two types
  - *Full buffers* can support distinct tokens on inputs/output channels
  - *Half buffers* cannot support distinct tokens on inputs/outputs
    - N-stage pipeline of half-buffers can support a maximum of N/2 tokens

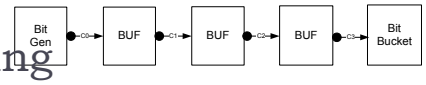
Full-Buffer Handshaking





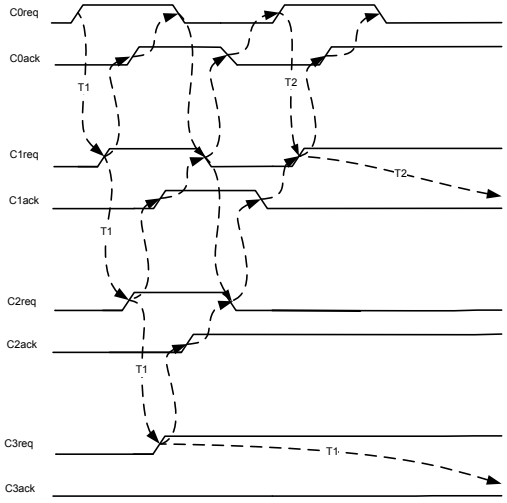
Handshaking assuming very slow Bit Bucket

Half-Buffer Handshaking

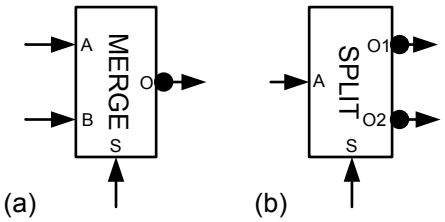


- Handshaking constraint that leads to a half-buffer
  - Output channel must be acknowledged (e.g., c2ack+)
    - indicating that the output token (e.g., on channel c2) has been consumed (and thus is in the subsequent channel (e.g., in channel c3))
  - Before the reset phases of the input channel is complete (e.g., before c1ack-) which is before
    - A new token on the input channel (e.g., c1) can be generated.

Handshaking assuming very slow Bit Bucket

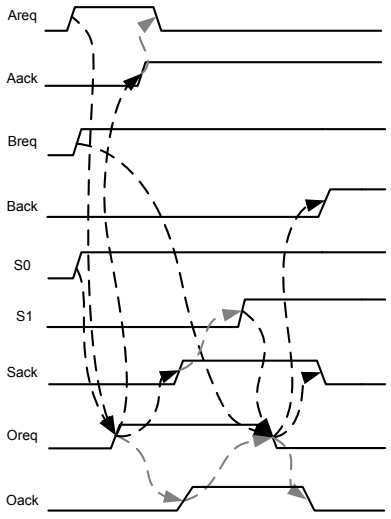


Conditional and Non-Linear Pipeines



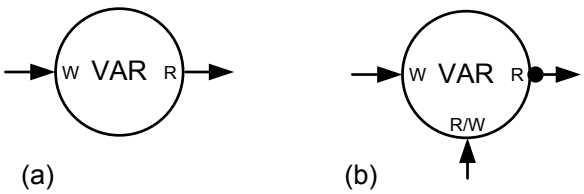
- MERGE
  - Wait for token on S.
  - Depending on value, wait for token on either A or B and send onto O
- SPLIT
  - Wait for token on S and A.
  - Dependent upon value of S, send copy of token on A to O1 or O2

Timing Diagram of Merge



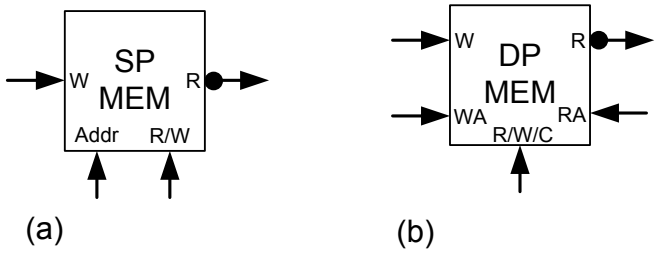
- Assumptions (in this example)
  - full-buffer two-phase handshaking
  - dual-rail select signal
- Functionality
  - Token on A consumed first
    - After token on S = 0
    - I.e., S0 changes
  - Token on B stalled until consumed second
    - After token on S = 1
    - I.e., Once S1 changes
  - Result: two tokens on O
    - First = Oreq+
    - Second = Oreq-

Variables



- Purpose
  - Store state of some program variable
  - W is a write port and R is a read port
- Type (a)
  - R and W are assumed to be mutually exclusive
- Type (b)
  - Waits for token on R/W 1-bit port
  - Dependent on value…
    - *Waits for request on W and store data value received or*
    - *Generates a token on R with the previously stored data value*

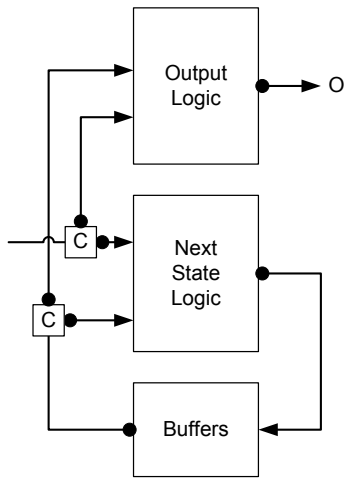
Multi-Bit Variables



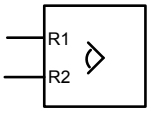
- Functionality
  - Store multi-bit state of some program variable
  - W is a write port and R is a read port
  - Addr is the address port
  - R/W controls read/write
  - C is simultaneous read and write to different addresses

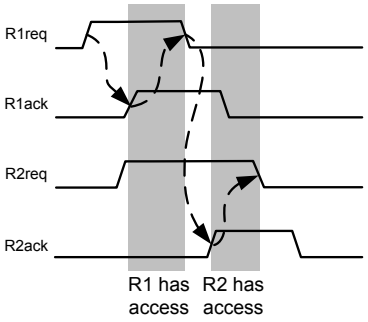
Channel-Based FSM

- Purpose
  - Implement FSM based on channels
- State
  - Stored in token in Buffers
  - Consumed every "cycle"
  - New state generated by Next State Logic
- Output
  - New output token generated in response to input token and current state
  - Copy cells needed to route input tokens and state to both NSL and OL.
- Buffers
  - One of the buffers must be token buffer, reflecting initial state of FSM



Basic 2-way Arbiter

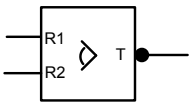


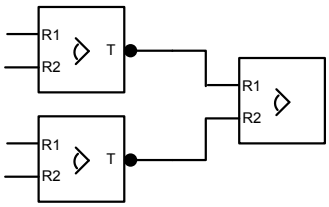


- Purpose
  - Used to control access to shared resource
- Approach
  - Acknowledge handshake on request port that arrives first, granting access
  - Requires four-phase protocol
    - winner maintains mutually-exclusive access of resource until it resets request
- Caveat
  - Make take an exponential amount of time to determine who came first when requests arrive very close together
- Sometimes called *slackless arbiter*

Slackless Tree Arbiters

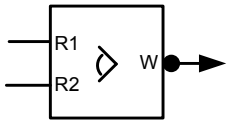
- Slackless Tree Arbiter cell
  - Used to build multi-way arbiters
- Approach
  - Add T channel to normal arbiter
  - Delay ack of request channels until T channel acknowledged
    - Can send request on T channel as soon as any request arrives





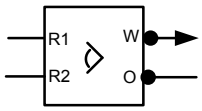
4-way Slackless Tree Arbiter

2-way (Pipelined) Arbiter



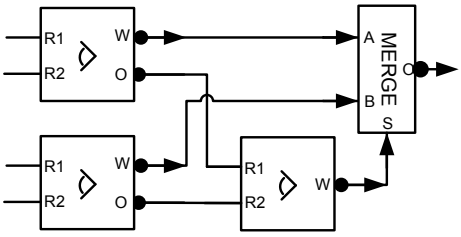
- Purpose
  - Used to control access to shared resource in a pipelined design
- Approach
  - Acknowledge of request not used to signify winner
  - Instead, additional W channel used to identify who won
  - Handshake on W simultaneously with acknowledging winning request
- Note
  - Still may take exponential time
  - In principle, this can use a two or four-phase protocol

Pipelined Tree Arbiter Cells



- Tree Arbiter cell
  - Used to build multi-way pipelined arbiters
- Approach
  - Add synchronization channel O to 2-way (pipelined) arbiter
  - Send request on O channel as soon as any request arrives
- Question
  - How can use this cell as the basis of a 4-way pipelined arbiter?

Pipelined Tree Arbiter – Naïve Solution

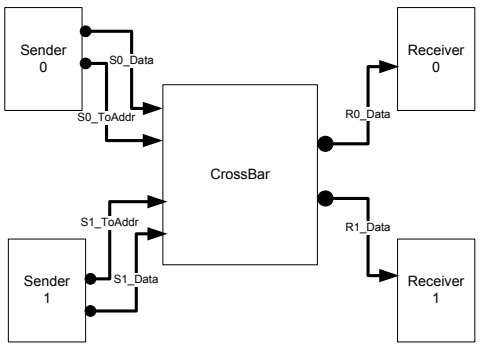


N.B. Assume this Merge operates on 1-bit data channels

- The Problem Scenario
  - All 4 requests arrive at same time
  - Output generates 1-bit output
  - Which of the 4 requests does this 1-bit output identify?
    - Need notion of addresses to distinguish between 4 requests

Design Example I: 2x2 Crossbar

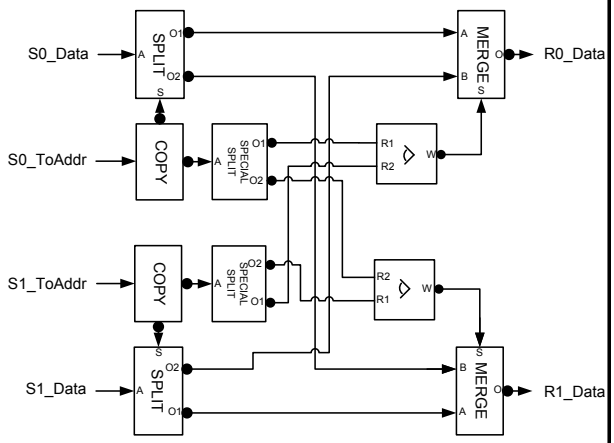
- Features
  - Provides any sender to any receiver communication
  - Concurrent communication enabled
    - S0 -> R0 && S1 -> R1
      OR
    - S0 ->R1 && S1 -> R0
  - No packets lost
  - Implicit flow control



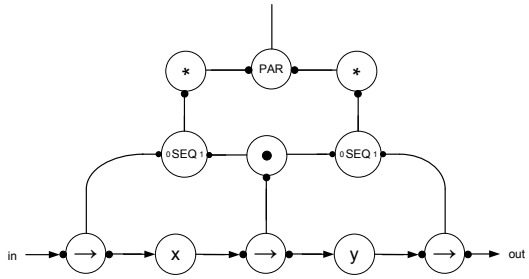
2x2 Crossbar Implementation

Design notes

- Addresses are 1-bit wide
- Special Splits
  - Splits 1-bit input into one of two synchronization tokens
- Not slack elastic
  - Adding pipeline buffers can cause design to deadlock
  - Occurs when control path has more slack than data channel

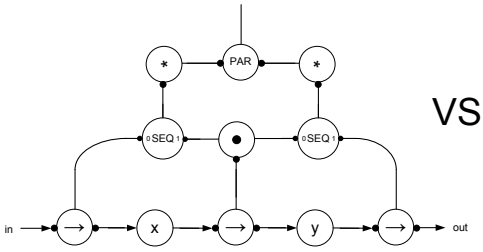


Design Example II: Control-driven 2-place FIFO

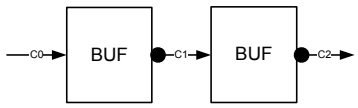


- Design notes
  - Each half of the pipeline is controlled by a repeater (denoted '*")
  - Repeatedly handshake its active nonput channel with a sequencer
  - Sequencer (denoted 'SEQ') is responsible for
    - first transferring the input to the corresponding variable
    - and then onto the next stage.
  - The join element, denoted by a '•', is responsible for synchronizing the transfer between the two variables.

Control-driven FIFO vs BUF-based pipeline



VS



- Control-driven 2-place FIFO
  - Transfer of data is control-driven and transfer elements have active (pull) inputs.
  - Data stored in designated variable elements "x" and "y"
- BUF-based linear pipeline
  - Data stored in channels
  - Can store two tokens (on C1 and C2) assuming BUF is a full-buffer
  - Data-driven consisting of pipeline buffers that have passive (push) inputs.