Asynchronous FPGA

(Based on Achronix FPGA)

Ophir Erez

037661584
048879 Seminar in VLSI Architecture
Agenda

- Introduction
- Synchronous FPGA (Virtex)
- Asynchronous design
- Achronix FPGA - introduction
- Basic asynchronous FPGA design
- Optimized asynchronous FPGA design
- Synthesis
- Comparisons
Introduction

- Asynchronous design methodologies seek to address:
  - Design complexity
  - Energy consumption
  - Timing issues

- Most experimental high-performance asynchronous designs have been designed with labor-intensive custom layout

- The idea: asynchronous dataflow FPGAs as an alternative method for prototyping these asynchronous systems
Xilinx FPGA

(Virtex 2.5V – synchronous FPGA)
Virtex™ 2.5 V
Field Programmable Gate Arrays
Virtex™ 2.5 V
Field Programmable Gate Arrays

Figure 4: 2-Slice Virtex CLB
Figure 5: Detailed View of Virtex Slice
Figure 7: Virtex Local Routing
Asynchronous vs. Synchronous

An asynchronous dataflow FPGA architecture distinguishes itself from that of a clocked FPGA architecture on the following criteria:

- **Ease of pipelining**
  - Asynchronous pipelines enable the design of high-throughput logic cores that are easily composable and reusable, where asynchronous pipeline handshakes enforce correctness instead of circuit delays or pipeline depths as in clocked pipelines.

- **Event-driven energy consumption**
  - Asynchronous logic implements perfect “clock gating” by automatically turning off unused circuits since the parts of an asynchronous circuit that do not contribute to the computation being performed have no switching activity.

- **Robustness**
  - Asynchronous circuits are automatically adaptive to delay variations resulting from temperature fluctuations, supply voltage changes, and the imperfect physical manufacturing of a chip, which are increasingly difficult to control in deep submicron technologies.

- **Tool compatibility**
  - Able to use existing place and route CAD tools developed for clocked FPGAs.
Asynchronous Pipelines

- Since there is no clock in an asynchronous design, processes use handshake protocols to send and receive tokens on channels.

- Most of the channels in the FPGA use three wires, two data wires and one enable wire, to implement a four-phase handshake protocol.
Asynchronous Pipelines

- The data wires encode bits using a dual-rail code such that setting “wire-0” transmits a “logic-0” and setting “wire-1” transmits a “logic-1”.

- The four-phase protocol:
  - The sender sets one of the data wires
  - The receiver latches the data and lowers the enable wire
  - The sender lowers all data wires
  - The receiver raises the enable wire when it is ready to accept new data.
Asynchronous Pipelines

- Data wires start and end the handshake with their values in the lowered position ➔ this handshake is an example of an asynchronous return-to-zero protocol.

- The cycle time of a pipeline stage is the time required to complete one four-phase handshake.

- The throughput, or the inverse of the cycle time, is the rate at which tokens travel through the pipeline.
Slack elastic & retiming

- Changing local pipeline depths in a clocked circuit often requires global retiming of the entire system.
- Slack elastic
  - Increasing the pipeline depth (= slack) will not change the logical correctness of the original system.
- Allows to locally add pipelining anywhere in the system without having to adjust or re-synthesize the global pipeline structure of the system.
- The pipelines in the asynchronous FPGAs were designed so that they are slack elastic.
  - pipeline depth is chosen for performance and not because of correctness.
- In a pipelined interconnect, the channel routes can go through an arbitrary number of interconnect pipeline stages without affecting the correctness of the logic.
  - Banks of retiming registers (a significant overhead in pipelined clock FPGAs) aren’t necessary.
Dataflow Computations

- Token traveling through an asynchronous pipeline explicitly indicates the flow of data.
- Channel handshakes ensure that pipeline stages consume and produce tokens in sequential order so that new data items cannot overwrite old data items.
- In this dataflow model, data items have one producer and one consumer.
  - Data items needed by more than one consumer are duplicated by copy processes that produce a new token for every concurrent consumer.

- Clocked logic uses a global clock to separate data items in a pipeline, which allows data items to fan out to multiple receivers because they are all synchronized to the clock.
- The default behavior for a clocked pipeline is to overwrite data items on the next clock cycle, regardless of whether they were actually used for a computation in the previous cycle.
Dataflow Computations

\[ y_n = y_{n-1} + c(a + b) \]
Dataflow Computations

- Logical pipeline stages
  - Dataflow nodes in an asynchronous dataflow graph

- Physical pipeline stages
  - Dataflow node can be built from an arbitrary number of stages

- Designer only needs to understand how to program for this token-based dataflow computation model and is not required to know the underlying asynchronous pipelining details.
Asynchronous Dataflow Nodes

- **Copy:**
  - Dataflow node duplicates tokens to $N$ receivers.
  - It receives a token on its input channel and copies the token to all of its output channels.
Asynchronous Dataflow Nodes

- **Function:**
  - Dataflow node computes arbitrary functions of $N$ variables.
  - It waits until tokens have been received on all its input channels and then generates a token on its output channel.
Asynchronous Dataflow Nodes

- Merge:
  - Dataflow node performs a token merge and allows tokens to be conditionally read on channels.
  - It receives a control token on channel C.
  - If the control token has a zero value, it reads a data token from channel A, otherwise it reads a data token from channel B.
  - Finally, the data token is sent on channel Z.
  - A merge node is similar to a clocked multiplexer except that a token on the unused conditional input channel will not be consumed and need not be present for the merge node to process tokens on the active input data channel.
Asynchronous Dataflow Nodes

- Multi way merge blocks can be constructed from a combination of two-way merge nodes, two-way split nodes, and copy nodes.
Asynchronous Dataflow Nodes

- Split:
  - Dataflow node performs a split and allows tokens to be conditionally sent on channels.
  - It receives a control token on channel C and a data token on channel A.
  - If the control token has a zero value, it sends the data token on channel Y, otherwise it sends the data token on channel Z.
  - A split node is similar to a clocked demultiplexer except that no token is generated on the unused conditional output channel in its asynchronous implementation.
Asynchronous Dataflow Nodes

- Multiway split blocks can be built using a combination of two-way split nodes and copy nodes
Pipelined Asynchronous Circuits

- Delay model dictates the assumptions made about delays in the gates and wires during the design process.
- The less restrictive the delay assumptions, the more robust is the design to delay variations caused by a variety of factors:
  - Manufacturing process variations
  - Unpredictable wire lengths
  - Crosstalk noise
- This robustness, however, often comes at a cost:
  - Larger area
  - Lower performance
  - Higher power
Pipelined Asynchronous Circuits

- Delay-insensitive (DI) design
- The most robust of all asynchronous circuit delay models
- Makes no assumptions on the delay of wires or gates, i.e. they can have zero to infinity delay and can be time varying
- To build delay-insensitive circuits the smallest building block must be larger than a single-output gate
Pipelined Asynchronous Circuits

- Quasi Delay-insensitive (QDI) design
- All gates and wires can have arbitrary delays, except for a set of designated wire forks labeled as isochronic.
- Isochronic forks, have the additional constraint that the delay to the different ends of the fork must be the same.
- The purpose of the isochronic-fork assumption is to ensure that the ordering of transitions at the various inputs of a gate, necessary to preserve hazard freedom, can be guaranteed.
Pipelined Asynchronous Circuits

- If the fork F is isochronic, it is guaranteed that the rising transition at B arrives before the falling transition at A. This means that there is no glitch at C. If the fork were not isochronic then a glitch at C could occur and the circuit would be hazardous.

- In other words, the isochronic-fork assumption can be relaxed to mean that the delay from one end of the fork to its terminal gate G must be less than the delay of any fanout path through any other end of the fork that also terminates at G.
Pipelined Asynchronous Circuits

- The difference in the times at which a signal arrives at the ends of an isochronic fork is assumed to be less than the minimum gate delay.
- If these isochronic forks are guaranteed to be physically localized to a small region, this assumption can be easily met and the circuits can be practically as robust as DI circuits.

- Another interesting feature of a QDI design is that, primary inputs to the design should be unordered.
- Even if the specification indicates they are ordered, because of the unbounded wire delays to the gates that they drive, the ordering is not guaranteed at these gates.
Pipelines

Pipeline:

- Linear sequence of buffers where the output of one buffer connects to the input of the next buffer.
- Tokens are sent into the input end of the pipeline and flow through each buffer to the output end.
- The tokens remain in FIFO order.
- For synchronous pipelines the tokens usually advance through one stage on each clock cycle.
- For asynchronous pipelines there is no global clock to synchronize the movement. Instead each token moves forward down the pipeline when there is an empty cell in front of it. Otherwise it stalls.
Pipelines

- **Buffer capacity (slack)**
  - The maximum number of tokens that can packed into the buffer without stalling the input end of the pipeline.

- **Throughput**
  - The number of tokens per second which pass a given stage in the pipeline

- **Forward latency**
  - The time it takes a given token to travel the length of the pipeline
Buffer Reshuffling

- CSP
  - Communicating Sequential Processes
- A single rail buffer has the CSP specification *[L; R]*
- The buffer will have the handshaking:
  
  \[*[[L]; L^a\uparrow; [\neg L]; L^a\downarrow; [\neg R^a]; R\uparrow; [R^a]; R\downarrow]*

- The environment will perform:
  
  \[*[[\neg L^a]; L\uparrow; [L^a]; L\downarrow]\text{ and } *[[R]; R^a\uparrow; [\neg R]; R^a\downarrow]*

Buffer Reshuffling

- The wait for \([L]\) = the arrival of an input token
- The transition \(R^\uparrow\) is the beginning of the output token.
- An array of these buffers preserves the desired FIFO order and properties of a pipeline
Buffer Reshuffling

- Direct implementation of this CSP will require a state variable to distinguish the first half from the second half and has too much sequencing per cycle.
- It is better to reshuffle the waits and events to reduce the amount of sequencing and the number of state variables.
- We wish to maximize the throughput and minimize the latency of a pipeline.
Buffer Reshuffling
requirements for a valid reshuffling

First:
- The HSE maintains the handshaking protocols on L and R
- That is the projection on the L channel is:

  \[ * \begin{bmatrix} \langle \neg L \rangle & L^a \uparrow \end{bmatrix} \begin{bmatrix} \neg L \rangle & \uparrow \end{bmatrix} \]

- And the projection on the R channel is:

  \[ * \begin{bmatrix} \langle \neg R^a \rangle & R \uparrow \end{bmatrix} \begin{bmatrix} \neg R^a \rangle & \uparrow \end{bmatrix} \]
Buffer Reshuffling
requirements for a valid reshuffling

Second:
- #completed L↑ - #completed R↑ ≥ 0
- (i.e. the slack of the buffer ≥ 0)
- This conserves the number of tokens in the pipeline

Third: (constant response time requirement)
- The buffer should introduce some nonzero slack
- The Lᵃ↑ must not wait for the corresponding [Rᵃ]
- If it’ll wait, the reshuffling will have zero slack
Buffer Reshuffling

- These three requirements are sufficient to guarantee a correct implementation

- Expanding the L and R channels to encode data:
  - can move the $R^\uparrow$ past the corresponding $L^a\uparrow$
  - that data would need to be saved in internal state variables proportional to the number of bits on R or L
Buffer Reshuffling

\[ MSFB \equiv \ast [[ \neg R^a \land L ]; \ R \uparrow; ( [ R^a ]; \ R \downarrow), ( L^a \uparrow; [ \neg L ]; \ L^a \downarrow) ] \]

\[ PCFB \equiv \ast [[ \neg R^a \land L ]; \ R \uparrow; L^a \uparrow; ( [ R^a ]; \ R \downarrow), ( [ \neg L ]; \ L^a \downarrow) ] \]

\[ PCHB \equiv \ast [[ \neg R^a \land L ]; \ R \uparrow; L^a \uparrow; [ R^a ]; \ R \downarrow; [ \neg L ]; \ L^a \downarrow] \]

\[ WCHB \equiv \ast [[ \neg R^a \land L ]; \ R \uparrow; L^a \uparrow; [ R^a \land \neg L ]; \ R \downarrow; L^a \downarrow] \]

\[ B1 \equiv \ast [[ \neg R^a \land L ]; \ R \uparrow; L^a \uparrow; [ R^a \land \neg L ]; \ L^a \downarrow; R \downarrow] \]

\[ B2 \equiv \ast [[ \neg R^a \land L ]; \ R \uparrow; L^a \uparrow; [ \neg L ]; \ L^a \downarrow; [ R^a ]; \ R \downarrow] \]

\[ B3 \equiv \ast [[ \neg R^a \land L ]; \ R \uparrow; L^a \uparrow; [ \neg L ]; ( [ R^a ]; \ R \downarrow), \ L^a \downarrow] \]

\[ B4 \equiv \ast [[ \neg R^a \land L ]; \ R \uparrow; L^a \uparrow; [ R^a ]; \ R \downarrow, ( [ \neg L ]; \ L^a \downarrow)] \]

\[ B5 \equiv \ast [[ \neg R^a \land L ]; \ R \uparrow; L^a \uparrow; [ R^a \land \neg L ]; \ R \downarrow, \ L^a \downarrow] \]
Buffer Reshuffling

- WC = weak condition logic
- PC = pre-charge logic
- HB = half buffer (slack \( \frac{1}{2} \)) - The L and R channels cannot simultaneously hold two distinct data tokens
- FB = full buffer (slack 1)

**Weak condition:**

- The validity and neutrality of the output data R implies the validity and neutrality of the input data L
- It means that the L does not need to be checked anywhere else besides in R

**In the half buffer reshufflings only every other stage can have token on its output channel since a token on that channel blocks the previous stage from producing an output token**
Buffer Reshuffling

\[
P_{CFB} \equiv \ast \left[ [\neg R^a \land L]; \; R\uparrow; \; L^a\uparrow; \; en\downarrow; \; ([R^a]; \; R\downarrow), \; ([\neg L]; \; L^a\downarrow); \; en\uparrow \right]
\]

\[
P_{CHB} \equiv \ast \left[ [\neg R^a \land L]; \; R\uparrow; \; L^a\uparrow; \; [R^a]; \; R\downarrow; \; [\neg L]; \; L^a\downarrow \right]
\]

\[
W_{CHB} \equiv \ast \left[ [\neg R^a \land L]; \; R\uparrow; \; L^a\uparrow; \; [R^a \land \neg L]; \; R\downarrow; \; L^a\downarrow \right]
\]
Buffer Reshuffling

- Expanding for more than single rail:

  - The L and L^a will represent all the input data and acknowledges
  - The R and R^a will represent all the output data and acknowledges
  - [L] indicates a wait for the validity of all inputs and
  - [¬L] wait for the neutrality of all inputs
  - [¬R^a] indicates a wait for all the output acknowledges to be false
  - [R^a] indicates a wait for all the output acknowledges to be true
  - L^a↑ indicates raising all the input acknowledges in parallel
  - L^a↓ indicates lowering them
  - R↑ means that all the outputs are set to their valid states in parallel
  - R↓ means that all the outputs are set to their neutral states
  - When R↑ occurs it is meant that particular rails of the outputs are raised depending on which rails of L are true. This expands R↑ into a set of exclusive selection statements executing in parallel
Buffer Reshuffling - WCHB

- The CSP is \( [L?x; R!x] \)

\[
\begin{align*}
WCHB\_BUF & \equiv \\
* \left( [\neg R^a \land L^0 \rightarrow R^0 \uparrow; \neg R^a \land L^1 \rightarrow R^1 \uparrow]; L^a \uparrow; \\
\neg L^0 \land \neg L^1 \rightarrow R^0 \downarrow, R^1 \downarrow]; L^a \downarrow \right)
\end{align*}
\]

\[
\begin{align*}
R^e \land L^0 & \rightarrow \overline{R^0} \downarrow \\
R^e \land L^1 & \rightarrow \overline{R^1} \downarrow \\
\neg R^0 & \rightarrow R^0 \uparrow \\
\neg R^1 & \rightarrow R^1 \uparrow \\
\neg R^0 \lor \neg R^1 & \rightarrow \overline{L^e} \uparrow \\
\overline{L^e} & \rightarrow L^e \downarrow \\
\neg R^e \land \neg L^0 & \rightarrow \overline{R^0} \uparrow \\
\neg R^e \land \neg L^1 & \rightarrow \overline{R^1} \uparrow \\
\overline{R^0} & \rightarrow R^0 \downarrow \\
\overline{R^1} & \rightarrow R^1 \downarrow \\
\overline{R^0} \land \overline{R^1} & \rightarrow \overline{L^e} \downarrow \\
\neg \overline{L^e} & \rightarrow \overline{L^e} \uparrow
\end{align*}
\]

Figure 1: WCHB\_BUF
Buffer Reshuffling - WCHB
Buffer Reshuffling - WCHB
WCHB

Bit Generator

Bit Bucket
WCHB
C-element restriction

Implementing complex function with the WCHB leads to a poor performance
Precharged half buffer - PCHB

\[ PCHB_{BUF} \equiv \]

\[ \ast \left[ \left[ \neg R^a \land L^0 \rightarrow R^0 \uparrow \right] ; \left[ \neg R^a \land L^1 \rightarrow R^1 \uparrow \right] ; \right] ; \left[ R^a \rightarrow R^0 \downarrow , R^1 \downarrow \right] ; \left[ \neg L^0 \land \neg L^1 \rightarrow L^a \downarrow \right] \]
Precharged half buffer - PCHB
The evaluation of the domino logic depends on four precursors:
- the input data is valid
- the LCD indicates that the previous data has gone neutral
- the RCD indicates that the previous output data has gone neutral
- the acknowledge signal has gone high

The pre-charge of the domino block depends on three precursors:
- the LCD indicates that the input is valid
- the RCD indicates that the output is valid
- the acknowledge signal has gone low and indicates that the output data has been consumed by the next stage

The dependency between the data-input falling and the data-output falling has been removed and, instead, the input falling is a precursor for the output’s re-evaluating
Achronix FPGA - introduction
Achronix FPGA

Frame contains: I/Os, SerDes, clocks, PLLs, etc.
Achronix FPGA

- Each RLB contains:
  - 8 x 4-input LUTs,
  - storage elements
  - 128 bits of RAM.

- In addition to RLBs and programmable routing, the picoPIPE fabric also contains Block RAMs and dedicated multipliers.

- The RLBs and routing are implemented using picoPIPE technology, enabling much higher throughput than non-picoPIPE-based FPGAs.

- The device operates @1.5GHz
Achronix FPGA

**Figure 2: picoPIPE Building Blocks**

**Figure 3: PicoPIPE Pipeline Stages**
Achronix FPGA

- CEs are capable of being initialized into one of two states:
  - state holding
  - not state holding.
- When initialized as state-holding, a CE behaves as a register. When initialized as not state-holding, it behaves like a repeater.
- The main difference between a series of un-initialized CEs and a wire, is that each pipeline stage is still capable of containing a Data Token, even if it doesn’t start with one initially.
Achronix FPGA

- FEs have functionality equivalent to combinatorial logic.
- The only difference relates to how ingress and egress data is handled.
- The local handshaking within a picoPIPE network means the FEs must also handshake data in and out.
- This handshaking ensures only valid, settled data is propagated.
Achronix FPGA

- BEs are only used at the boundary where the picoPIPE fabric meets the FPGA Frame.
- These elements are responsible for converting Data Tokens in the Frame into Data Tokens in the picoPIPE fabric (ingress).
- They are also used for converting Data Tokens in the fabric back into Data Tokens in the Frame (egress).
- The conventional I/O Frame ensures that every Data Token enters the picoPIPE core at a clock edge, and every Data Token leaving the picoPIPE core is clocked out at a clock edge.
Benefits of picoPIPE technology in terms of physical Layout

Figure 5: Conventional Implementation vs. picoPIPE Implementation
John Teifel and Rajit Manohar

“Programmable Asynchronous Pipeline Arrays (PAPA)”

(2003)

Base Architecture
The PAPA Architecture

- Logic Cells surrounded by Channel Routers
- Logic cells communicate through 1-bit wide, dual-rail encoded channels that have programmable connections configured by the channel routers
This router is implemented as a switch matrix and is un-pipelined.

Copies result tokens from channels Y and Z to one or more of the physical output ports (Nout; Eout; Sout; Wout) or sinks the result tokens before they reach any output port.

Two arbitrary functions of three variables. Receives tokens on channels (A; B; C) and sends function results on output channels (Y; Z).

Initializes with a token on its output. Upon system reset a token is sent on channel Y. Unit is used for state initialization.

Receives a control token on channel C. If it equals “logic-0" it reads a data token from channel A, otherwise it reads a data token from channel B. The data token is sent on channel Z.

Receives a control token on channel C and a data token on channel A. If the control token equals “logic-0" it sends the data token on channel Y, otherwise it sends the data token on channel Z.
Channel Router

- A PAPA channel router is an un-pipelined switch matrix.
- It statically routes channels between logic cells.
- Route all channels on point-to-point pathways and all routes are three wires wide (necessary to support the dual-rail channel protocol).
- Each channel router has 12 channel ports (6 input and 6 output).
- Four of the ports are reserved for connecting channels to adjacent logic cells and the remaining ports are used to route channels to other channel routers.
- To keep the configuration overhead manageable, the router does not allow “backward” routes (i.e., changing a channel’s route direction by 180 degrees).
Asynchronous circuits

- The asynchronous circuits that were built are quasi-delay-insensitive (QDI).
- The PAPA architecture is the first to adapt these circuits for programmable asynchronous logic

Features of the circuits:

- High throughput
  - Minimum pipeline cycle times of 10-16 FO4 delays (competitive with clocked domino logic).
- Low forward latency
  - Delay of a token through a pipeline stage is 2 FO4 delays (superior to clocked domino logic).
- Data-dependent pipeline throughput
  - Operating frequency depends on arrival rate of input tokens (varies from idle to full throughput).
- Energy efficient
  - Power savings from no extra output latch, no clock tree, and no dynamic power dissipation when the pipeline stage is idle.
Asynchronous circuits

- WCHB is most useful for token buffering and token copying
- PCHB is optimized for performing logic computations
  - (similar to dual-rail clocked domino circuits).
- Since the weak-condition and pre-charge pipeline stages both use the dual-rail handshake protocol, they can be freely mixed together in the same pipeline.
- WCHB stages are used in the
  - Token unit,
  - Output-Copy,
  - Copy processes of the Function unit.
- PCHB stages are used in the
  - Split unit,
  - Merge unit,
  - Evaluation part of the Function unit
Asynchronous circuits
Physical Design

- TSMC's 0.25μm CMOS process
  - (FO4 delay ≈120ps)

- An array cell includes:
  - one logic cell
  - two channel routers
  - 144 x 204 μm² in area
    - which is 50-100% larger than a conventional clocked FPGA cell
Physical Design

- To minimize cell area and simplify programming, configuration bits are programmed using JTAG clocked circuitry.
- The area breakdown for the architecture components is:
  - function unit (14.4%),
  - merge unit (2.5%),
  - split unit (2.9%),
  - token unit (2.6%),
  - output copies (12.5%),
  - configuration bits (37.7%),
  - channel/input routers (18.2%),
  - miscellaneous (9.1%).
Physical Design

- Simulation of the layout in SPICE found the maximum inter-cell operating frequency for PAPA logic to be 395MHz.
- Internally the logical units can operate much faster, but are slowed by the channel routers.
- To observe this the logical units were configured to internally source “logic-1" tokens on their inputs and the Output-Copy stages were configured to sink all result tokens (bypassing all routers). The results are:
  - Function unit (498MHz, 26pJ/cycle),
  - Merge unit (543MHz, 11pJ/cycle),
  - Split unit (484MHz, 12pJ/cycle),
  - Token unit (887MHz, 7pJ/cycle).
- These measurements compare favorably to the pipelined clock FPGA that operates at 250MHz and consumes 15pJ/cycle of energy per logic cell.
John Teifel and Rajit Manohar

“Highly Pipelined Asynchronous FPGAs”

Optimized Architecture

(2004)
Island style architecture

(a) SB SB SB SB
    |     |     |
    LB   LB   LB   LB
(b) SB SB SB SB
    |     |     |
    SB   SB   SB   SB

Logic Block

Nin  Nout

Connection Box

Switch Box
dual-rail channels
Logic Cell

Input Pipelining and Routing  Pipelined Computation Block  Output Pipelining and Routing
Logic Cell – Input stage

- Single WCHB pipeline stages.
- Buffer input tokens from the connection box switches.
- Upon system reset, the input buffers can optionally initialize the internal logic block channels (N,E,S,W) with tokens.
- Three constant token sources can also be routed to any of the function or conditional unit inputs.
Carry chains can be routed using the normal interconnect or using low latency carry channels that run vertically south-to-north between adjacent vertical logic blocks.
Logic Cell – Asynchronous LUT circuit
Logic Cell – State Unit

- The state unit is a small pipeline built from two WCHB stages that feeds the function unit output back as an input to the function unit, forming a fast token-ring pipeline.
- Upon system reset, a token is generated by the state unit to initialize this token-ring.
- This state feedback mechanism is better than in basic design, where all feedback token rings needed to be routed through the global interconnect.
Logic Cell – Conditional unit

(a) Merge

(b) Split

Precharge Evaluation Circuit

Handshake Control Circuit

Configuration Memory Cell
Logical Cell – Output Copy

Performs both token copying and token routing
Pipelined Interconnect

[Diagram of pipelined interconnect with labels and connections]

Logic Block

Connection Box

Switch Box

dual-rail channels

N E W S

switch point
Pipelined Interconnect

- Channel connecting two logic blocks can be routed through an arbitrary number of pipelined switch boxes without changing the correctness of the resulting logic system (slack elasticity)
- System performance can still decrease if a channel is routed through a large number of switch boxes
- Need to determine sensitivity of channel route lengths on pipelined logic performance
Performance of linear pipelines

![Diagram showing switch box mismatch and linear pipelines LB1 and LB2 with frequency f [MHz] vs switch box mismatch]
Performance of token-ring pipelines
John Teifel and Rajit Manohar

“Static Tokens: Using Dataflow to Automate Concurrent Pipeline Synthesis”

(2004)

Synthesizing Logic into FPGA
Logic Synthesis

- The compiler generates concurrent processes using only seven types of simple pipeline templates.
- Variable definitions = producers of data tokens
- Uses of Variables = consumers of data tokens
Logic synthesis

- Sequential CHP Specification
  - lexer/parser
- Abstract Syntax Tree
  - AST transformations
- Canonical Sequential CHP
  - sequential compiler analysis
- Sequential Control Flow Graph
  - concurrent dataflow decomposition
- Concurrent Dataflow Graph
  - technology mapping
- Asynchronous Pipeline Stages
Summary of CHP Notation

- **Assignment:** \((a := b)\)
  - “assign the value of \(b\) to \(a\)”
  - \(a\uparrow\) for \(a := \text{true}\)
  - \(a\downarrow\) for \(a := \text{false}\)

- **Selection:** \([G_1 \rightarrow S_1 [] ... []G_n \rightarrow S_n]\)
  - \(G_i\)’s are boolean expressions (guards)
  - \(S_i\)’s are program parts
  - \([G]\) is short-hand for \([G \rightarrow \text{skip}]\)
  - If the guards are not mutually exclusive use: \(|\) instead of \([]\)
Summary of CHP Notation

- Repetition: *[G1 → S1 [] ... [] Gn → Sn]*
  - Choosing one of the true guards and executing the corresponding statement
  - Repeating this until all guards evaluate to false
  - *[S] is short-hand for *[true → S]*

- Send: X!e
  - send the value of e over channel X

- Receive: Y?v
  - Receive a value over channel Y and store it in variable v.
Summary of CHP Notation

- Sequential Composition: $S;T$
- Parallel Composition: $S \parallel T$ or $S;T$
- Simultaneous Composition: $S \cdot T$
  - both $S$ and $T$ are communication actions and they complete simultaneously.
Canonical CHP Form

- Guard expressions are of the form “v = i”
  - v is an integer variable
  - i is a constant

- Unique channel actions
  - *[Z?x ; Z!x] is allowed, but not *[A?x ; Z!x ; Z!x]*

```
[ G_0 \rightarrow \ldots ]
[ G_1 \rightarrow \ldots ]
[ \ldots ]
[ G_{n-1} \rightarrow \ldots ]
[ \ldots ]
[ g := \text{selection}(\ldots); ]
[ g = 0 \rightarrow \ldots ]
[ g = 1 \rightarrow \ldots ]
[ \ldots ]
[ g = n - 1 \rightarrow \ldots ]
[ \ldots ]
```
Concurrent Dataflow Decomposition

- Copy $\equiv \*[A?a; Z_0!a; \ldots; Z_{n-1}!a]$
- Function $\equiv \*[A_0?a_0; \ldots; A_{n-1}?a_{n-1}; Z!f(a_0; \ldots; a_{n-1})]$
- Split $\equiv \*[C?c; A?a$
  $[c = 0 \rightarrow Z_0!a[\ldots[[c = n-1 \rightarrow Z_{n-1}!a]]$
- Merge $\equiv \*[C?c$
  $[c = 0 \rightarrow A_0?a[\ldots[[c = n-1 \rightarrow A_{n-1}?a];$
  $Z!a]$
- Source $\equiv \*[Z!"constant"]$
- Sink $\equiv \*[A?a]$
- Initializer $\equiv a := \"constant\"; Z_0!a; \ldots; Z_{n-1}!a;$
  $\*[A?a; Z_0!a; \ldots; Z_{n-1}!a]$
Steps of synthesis

- Begin with a high-level sequential specification of the logic
- Apply semantics-preserving program transformations to partition the original specification into high-level concurrent function blocks.
- The function blocks are further decomposed into sets of highly concurrent processes that are guaranteed to be functionally equivalent to the original sequential specification.
- Map the results to the FPGA’s logical cells
while(1){
    U?u;
    V?v;
    A?a;
    if(u&v){
        B?x;
        C?c;
        F!f(a,x,c)
    }else{
        D?d;
        E?e;
        x=g(d,e);
    }
    X!x;
}
Implementing scaling accumulators
## Base Vs. Optimized Asynchronous Dataflow FPGA

<table>
<thead>
<tr>
<th>Feature</th>
<th>Base Design</th>
<th>Optimized Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Block</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clustered</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Inputs/outputs per logic block</td>
<td>4/4</td>
<td>4/4</td>
</tr>
<tr>
<td>Maximum number of independent outputs</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Function unit</td>
<td>3-LUT</td>
<td>4-LUT</td>
</tr>
<tr>
<td>Merge and split units</td>
<td>independent</td>
<td>unified</td>
</tr>
<tr>
<td>Carry logic</td>
<td>blocking</td>
<td>early-out</td>
</tr>
<tr>
<td>Input pipeline stages</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Low-latency copies</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Token initializers</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Token sources</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Token sinks</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Interconnect</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Routing channels</td>
<td>unidirectional</td>
<td>bidirectional</td>
</tr>
<tr>
<td>Horizontal/Vertical routing tracks</td>
<td>4/4</td>
<td>4/4</td>
</tr>
<tr>
<td>Inter-logic-block routing</td>
<td>channel switches</td>
<td>connection boxes/switch boxes</td>
</tr>
<tr>
<td>Dedicated carry-chain routing</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Pipelined routing</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Retiming registers</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Compatible with clocked tools</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Physical Implementation of Dataflow FPGA Architectures

<table>
<thead>
<tr>
<th></th>
<th>Base Design</th>
<th>Optimized Design</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Area</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arrayable tile</td>
<td>2.0 Mλ² / 29k µm²</td>
<td>2.6 Mλ² / 37k µm²</td>
</tr>
<tr>
<td>Logic block computation units</td>
<td>33%</td>
<td>40%</td>
</tr>
<tr>
<td>Logic block input stages</td>
<td>11%</td>
<td>16%</td>
</tr>
<tr>
<td>Logic block output stages</td>
<td>17%</td>
<td>10%</td>
</tr>
<tr>
<td>Inter-logic-block routing</td>
<td>39%</td>
<td>34%</td>
</tr>
<tr>
<td><strong>Programming</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration bits per tile</td>
<td>96</td>
<td>201</td>
</tr>
<tr>
<td>Logic block bits</td>
<td>46%</td>
<td>52%</td>
</tr>
<tr>
<td>Interconnect bits</td>
<td>54%</td>
<td>48%</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peak inter-logic-block speed</td>
<td>400 MHz</td>
<td>400 MHz</td>
</tr>
<tr>
<td>Ripple-carry adder</td>
<td>292 MHz</td>
<td>395 MHz</td>
</tr>
<tr>
<td>Booth multiplier core [2]</td>
<td>222 MHz</td>
<td>395 MHz</td>
</tr>
</tbody>
</table>
Asynchronous Dataflow FPGA Vs. Clocked FPGAs

<table>
<thead>
<tr>
<th>Design</th>
<th>Tile area</th>
<th>Routing tracks</th>
<th>Process feature size</th>
<th>Peak throughput</th>
<th>Energy per logic block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Virtex™ [27]</td>
<td>1.25 Mλ²</td>
<td>30</td>
<td>0.22 μm</td>
<td>200 MHz</td>
<td>–</td>
</tr>
<tr>
<td>SFRA [25]</td>
<td>4.9 Mλ²</td>
<td>30</td>
<td>0.18 μm</td>
<td>300 MHz</td>
<td>–</td>
</tr>
<tr>
<td>HSRA [24]</td>
<td>6.3 Mλ²</td>
<td>-</td>
<td>0.40 μm</td>
<td>250 MHz</td>
<td>15 pJ/cycle</td>
</tr>
<tr>
<td><strong>Dataflow FPGA</strong></td>
<td><strong>2.6 Mλ²</strong></td>
<td><strong>4</strong></td>
<td><strong>0.25 μm</strong></td>
<td><strong>400 MHz</strong></td>
<td><strong>18 pJ/cycle</strong></td>
</tr>
<tr>
<td>Dataflow FPGA†</td>
<td>5.0 Mλ²</td>
<td>15</td>
<td>0.25 μm</td>
<td>400 MHz</td>
<td>18 pJ/cycle</td>
</tr>
<tr>
<td>Dataflow FPGA†</td>
<td>6.1 Mλ²</td>
<td>20</td>
<td>0.25 μm</td>
<td>400 MHz</td>
<td>18 pJ/cycle</td>
</tr>
<tr>
<td>Dataflow FPGA†</td>
<td>8.3 Mλ²</td>
<td>30</td>
<td>0.25 μm</td>
<td>400 MHz</td>
<td>18 pJ/cycle</td>
</tr>
</tbody>
</table>
Circuit Performance

Uniform switch distribution

Long channel route

(a) 

(b)
Circuit Performance

Uniform switch distribution

Long channel route

(a)  
(b)
Questions?
Programmable C-element

Configuration Memory Cell

Programmable Pull-down

Standard C-element