Chapter 9

Algorithms and Architectures for Digital Processors

Algorithm

- An algorithm is a sequence of processing steps that create and/or transform data objects in memory
- A step-by-step problem-solving procedure, especially an established, recursive computational procedure for solving a problem in a finite number of step
- Ex: In view of Mathematics， In view of computer system，In view of circuit system ...

The notion of “algorithm”

- **Algorithm**
  - Description of a procedure which is
    - ✓ finite (i.e., consists of a finite sequence of characters)
    - ✓ complete (i.e., describes all computation steps)
    - ✓ unique (i.e., there are no ambiguities)
    - ✓ effective (i.e., each step has a defined effect and can be executed in finite time)
- **Desired properties of algorithms**
  - Correctness
    - ✓ For each input, the algorithm calculates the requested value
  - Termination
    - ✓ For each input, the algorithm performs only a finite number of steps
  - Efficiency
    - ✓ Runtime: The algorithm runs as fast as possible
    - ✓ Storage space: The algorithm requires as little storage space as possible

A general-purpose machine

- A general-purpose machine, or processor, can be programmed to execute a variety of algorithms
- But its architecture might not yield the highest performance for a particular application
- Might be underused by some applications
- Might not have a good balance between the processor's speed and its input-output (I/O) throughput over the domain of application

Compared to ASICs

- Compared to dedicated application-specific integrated circuits (ASICs)
- **A general-performance processor** might consume **more power, require more area, and have a higher cost** (depending on volume of sales)
- **ASIC chips sacrifice flexibility for performance, because their architecture is fixed**

The two primary tasks of High-level design

1. It **constructs an algorithm** that realizes a behavioral specification
2. It **maps the algorithm into an architecture** that *implements the behavior in hardware*

Notes:

- alternative algorithms may exhibit the same behavior
- Multiple architectures may implement a given algorithm

Focus on

(1) **developing an algorithm processor** (i.e., a fixed architecture that implements a given algorithm)

(2) **exploring architectural tradeoffs** (both for a network of FUs and for fine-grained implementations of the FUs themselves)

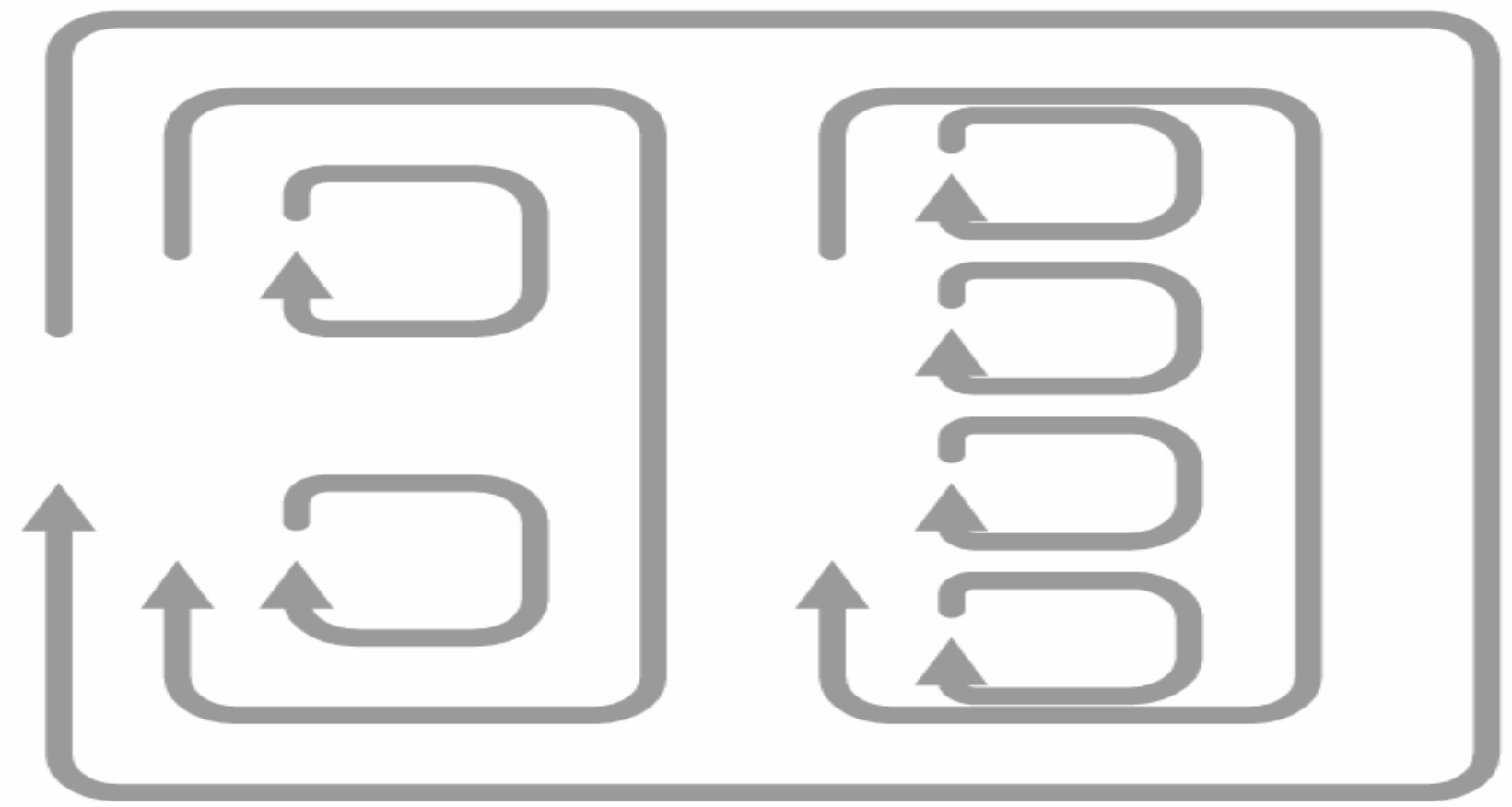
(3) **developing Verilog descriptions** of the architectures, and

(4) **synthesizing the architectures**

9.1 Algorithms, Nested-Loop Programs, and Data Flow Graphs

- Algorithmic processors are composed of FU, each executing in an environment of coordinated dataflow
- **A sequential algorithm can be described by a Nested-Loop Program (NLP)**
- NLP consists of a set of nested for loops, and a loop body written in a programming language such as C, in HDL

A sequential algorithm consists of a set of nested for loops



DFG: Data Flow Graph

- The sequential ordering of operations and the dependencies of data in an NLP for an algorithm can be represented by a data flow graph (DFG)
- Language parsers **extract a DFG from an NLP or from an HDL-based behavioral description**
- **DFG is a key tool** in developing an architecture for an algorithmic processor and for its alternative equivalent architectures

Data flow graph

- **DFG: data flow graph.**
- Does not represent control.
- Models basic block: code with one entry, exit.
- Describes the minimal ordering requirements on operations.

Data flow graph

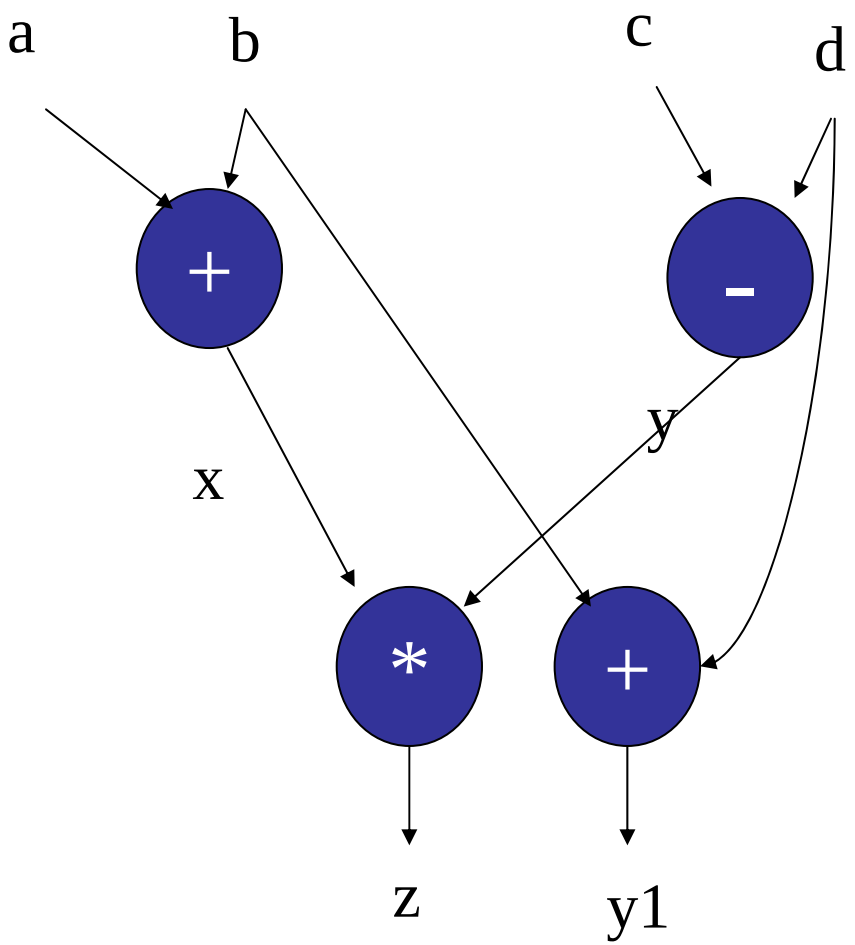
x = a + b;

y = c - d;

z = x * y;

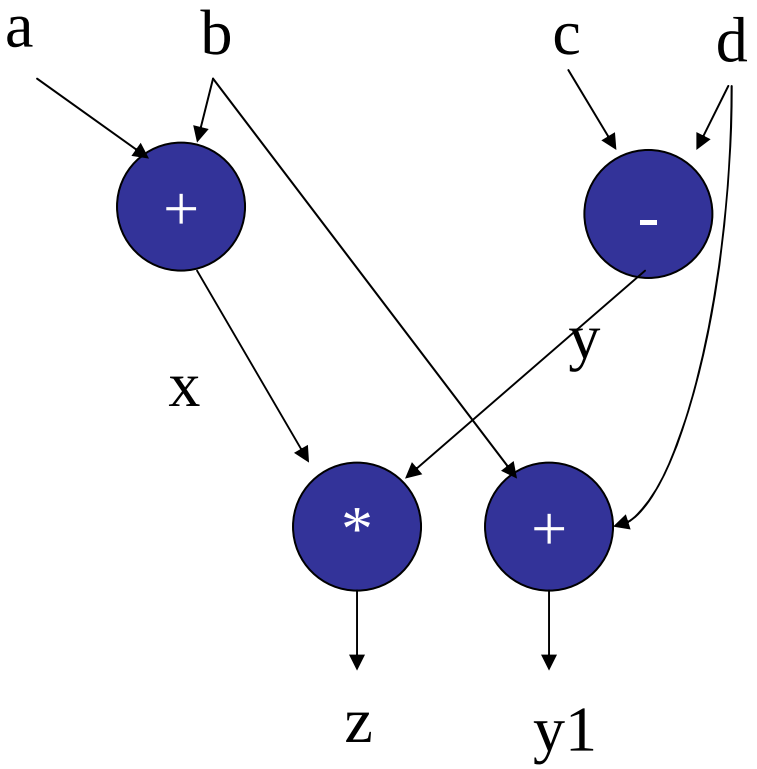
y1 = b + d;

single assignment form



DFG

DFGs and partial orders

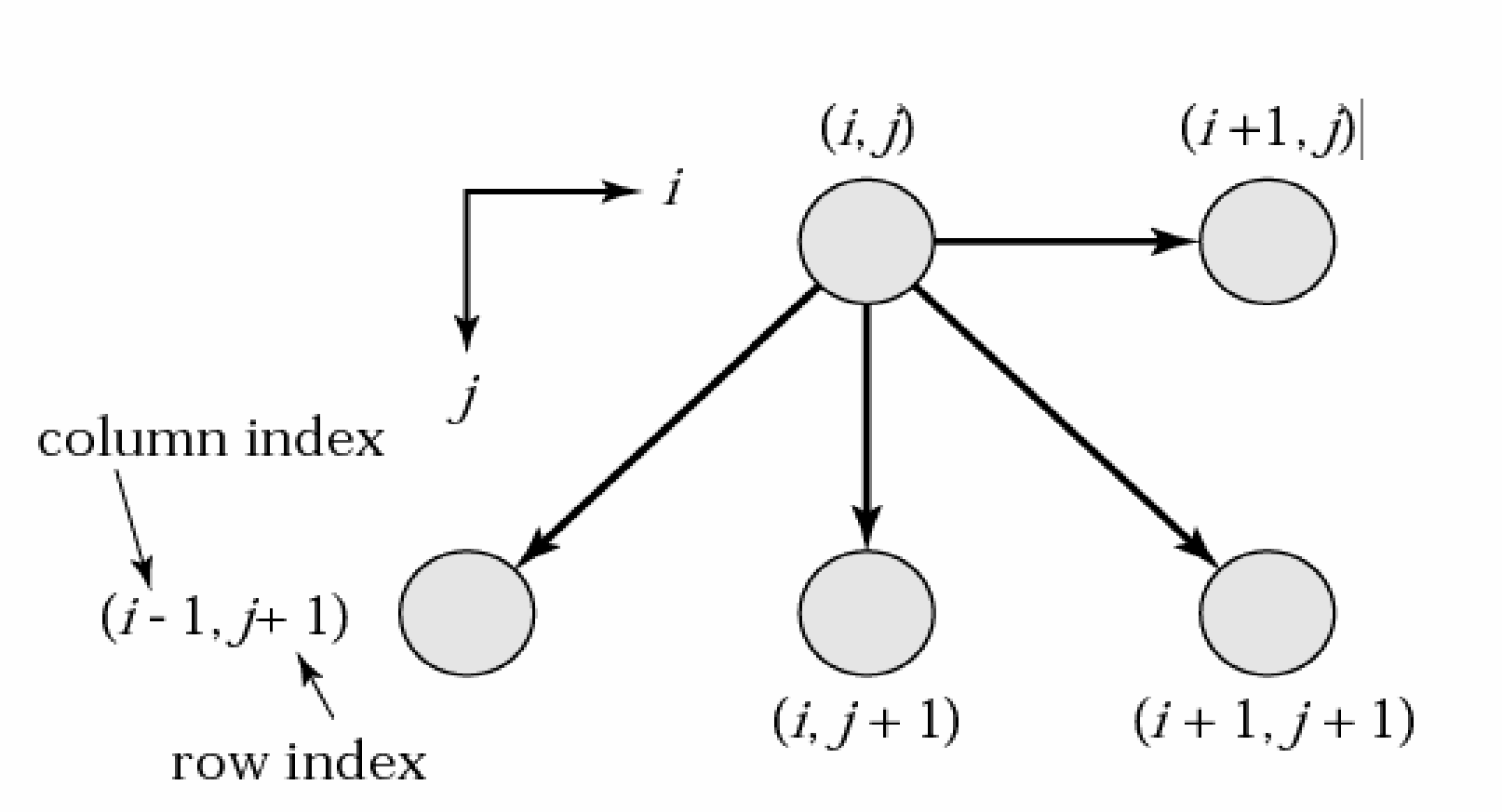


Partial order:

- a+b, c-d; b+d x*y

Can do pairs of operations in any order.

DFG



A DFG is a directed acyclic graph

- **A DFG is a directed acyclic graph, G(V, E),** where: V is the set of nodes of the graph, and E is the set of edges of the graph
- **Each node represents a functional unit (FU),** which operates on its inputs (data) to produce its outputs
- **Each directed edge e_{ij} originates at a node v_i and terminates at node v_j**
- **Given an edge $e_{i\,j}$, the data produced by node v_i is consumed by node v_j**

A data dependency exists between a pair of nodes

- **A data dependency exists between a pair of nodes (e_i,e_j)corresponding to edge eij,**
  - If the functional unit vj uses the results of functional unit vi
  - if the operation of vj cannot begin until the operation of vi has finished
- The DFG reveals the producers of data, the order in which the data will be generated, and the consumers of data
- It also exposes parallelism in the dataflow, opportunities for concurrent computation and has implications for the lifetime of variables

The DFG's synthesis

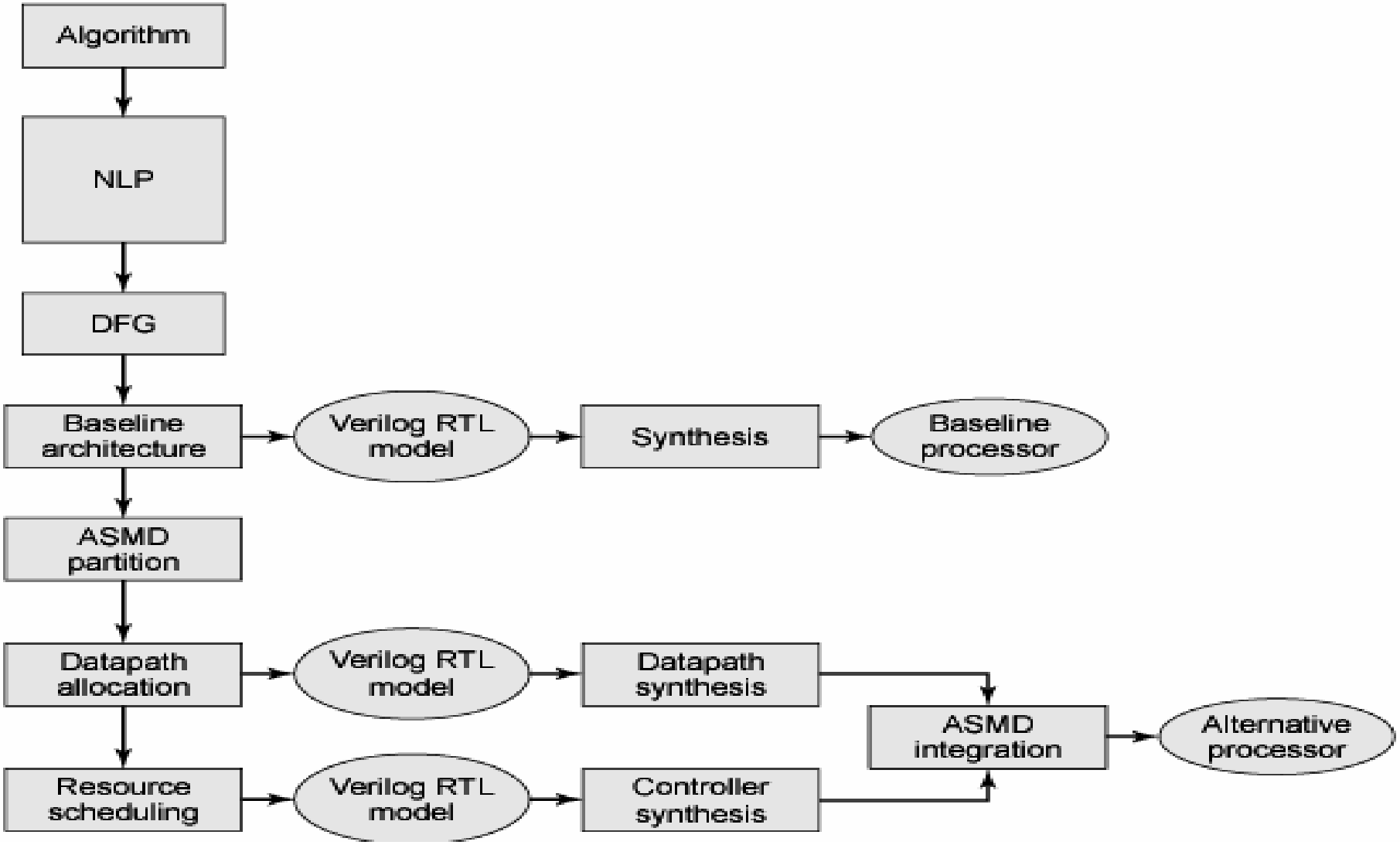
- Beginning with the DFG, the high-level synthesis task of datapath allocation consists of **transforming the DFG of an algorithm into an architecture of processors, datapaths, and registers** from which a synthesizable register transfer level (RTL) model

- A baseline architecture that implements a given DFG can always be formed as a set of FUs connected in a structure that is isomorphic to the DFG

Resource scheduling

- The high-level task of resource scheduling **assigns resources and a time slot to each node of a DFG.**
- Given the parallelism of a DFG, there are many schedules that could map nodes into time slots, creating several alternatives for realizing the corresponding algorithm in hardware
- Scheduling must be conflict-free (i.e., a resource cannot be allocated to multiple FUs in the same time slot)

Figure 9-2 Design flow for algorithm-based synthesis of a sequential machine



Three general approaches

- Three general approaches are used to reorganize the baseline architecture obtained from the DFG:
  - **Re-composition**
  - **pipelining**
  - **Replication**

Recomposition

- Recomposition segments the FU into a sequence of functions that execute one after the other to implement the algorithm
- The sequence of execution may be further **distributed over space (hardware units) and time**
- In Space , the DFG are mapped isomorphically to the FUs;
- In Time, a single FU executes over as many clock cycles as required to complete the operations represented by the DFG
- This approach saves hardware **by replicating the activity of a single FU over multiple time steps**, rather than using multiple processors that execute concurrently in a single step

Pipelining

- Pipelining inserts registers into a datapath to shorten computational paths and thereby increase the throughput of a system
- Incurring a penalty in latency and the number of registers

Replication

- In contrast to recomposition
- Replication uses multiple, identical, concurrently executing processors to improve performance
- But **at the expense of hardware**

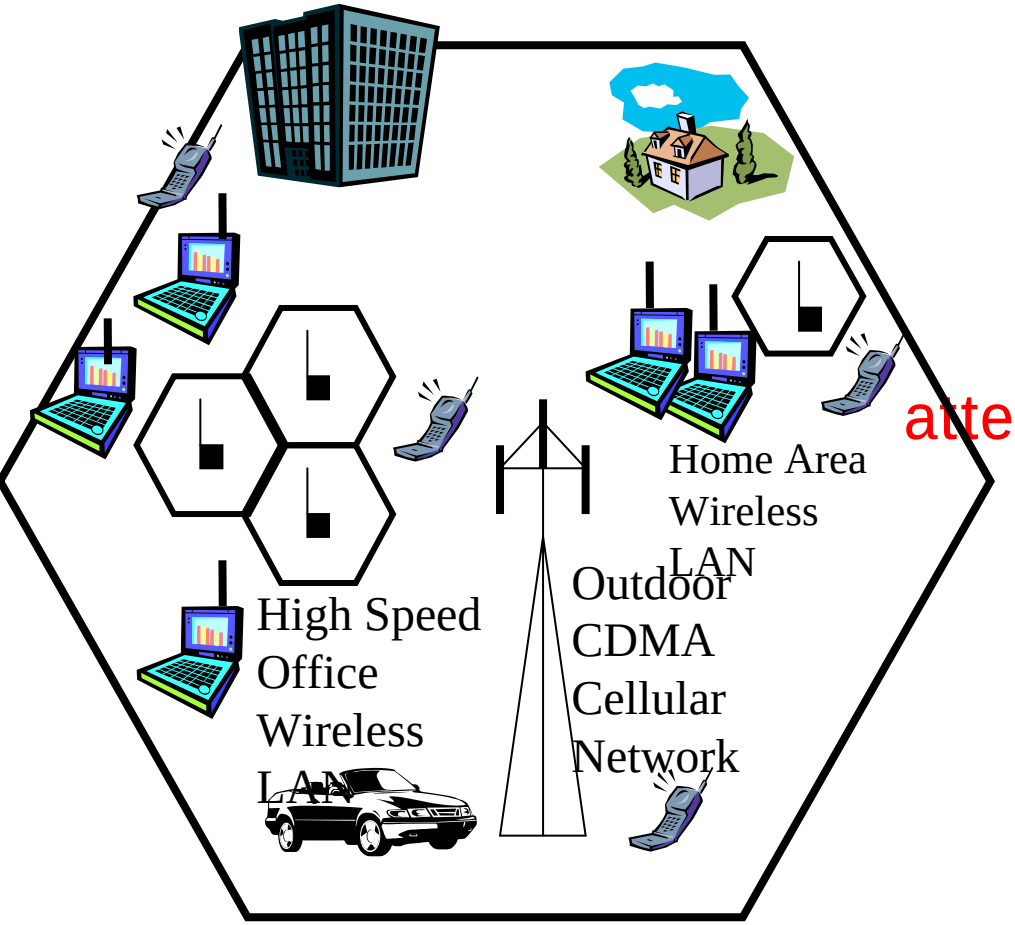
9.3 Digital Filters and Signal Processors

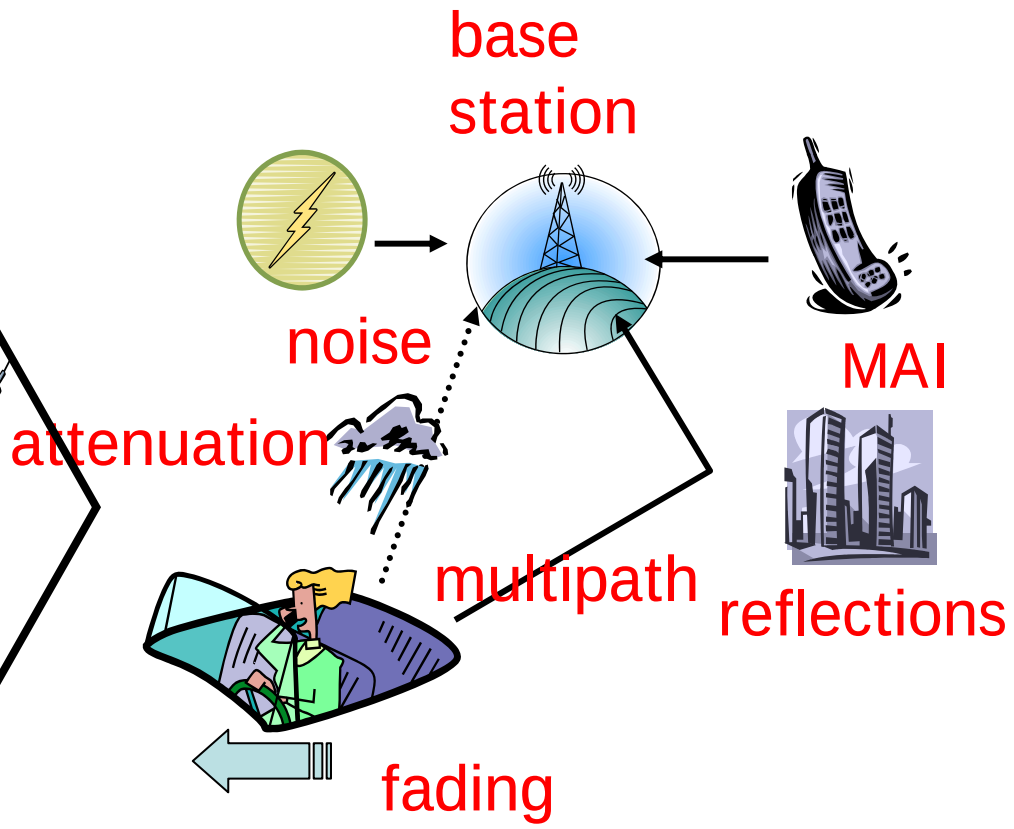
- Digital signal processors (DSPs) are prominent
- In cellular phones, personal digital assistants, still-image cameras, video cameras, and video recorders
- They provide superior performance, at lower cost and lower power, as compared with analog circuits

What Is DSP?

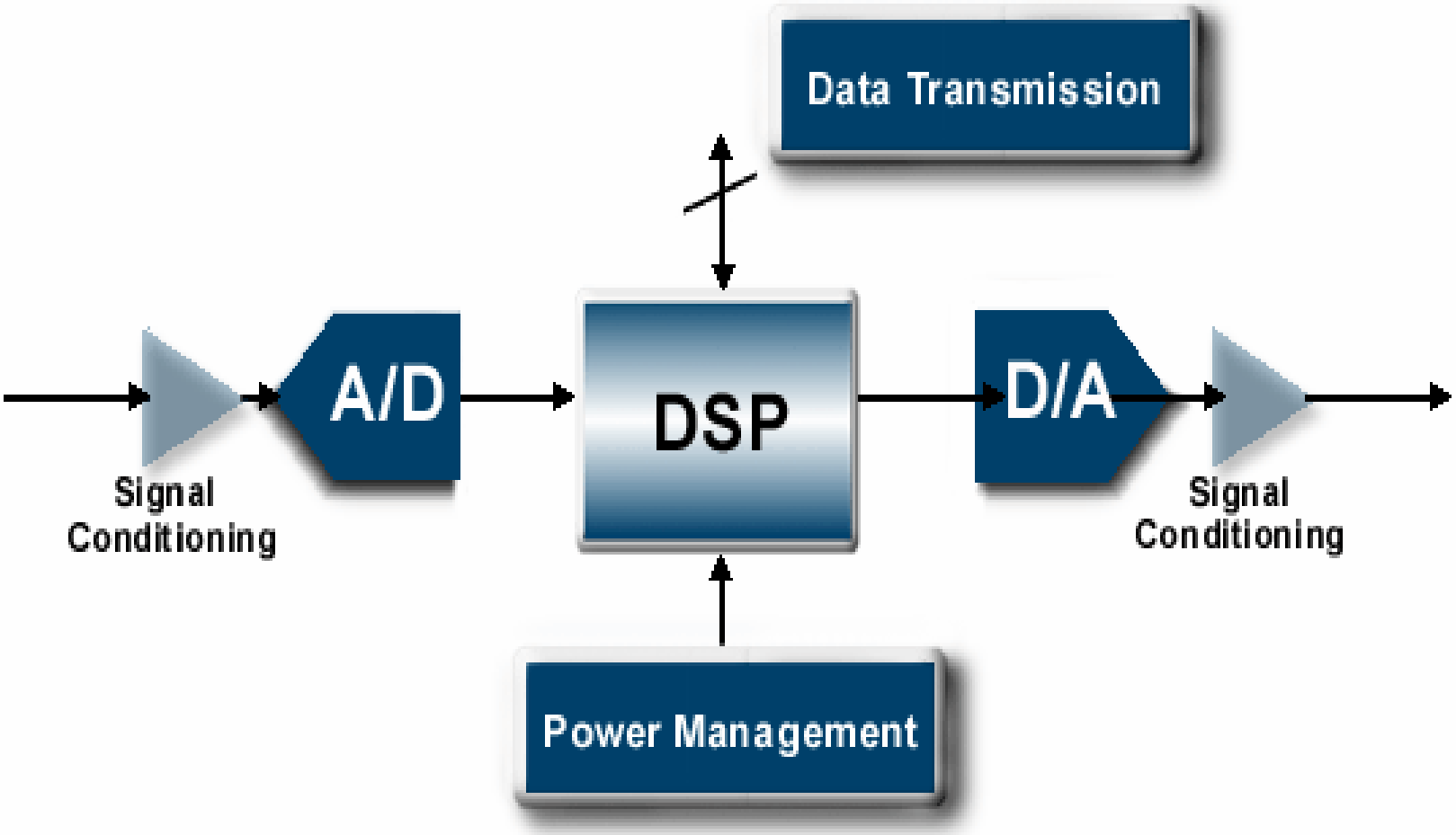
- **Digital Signal Processing**
  - Techniques for Clarifying, Analyzing & Converting Information Signals by Digital Means
- **Digital Signal Processor** With Specialized Instructions & Hardware for DSP Applications
- Optimized For High-Performance, Repetitive and Numerically Intensive Tasks
- Hardware
  - Fixed & Floating Point Multipliers
  - Co-Processors
  - Special Memory Structures
    - Multiple Access Memories For Faster Computations

Wireless Challenges:Higher data rates ,lower power signals,....

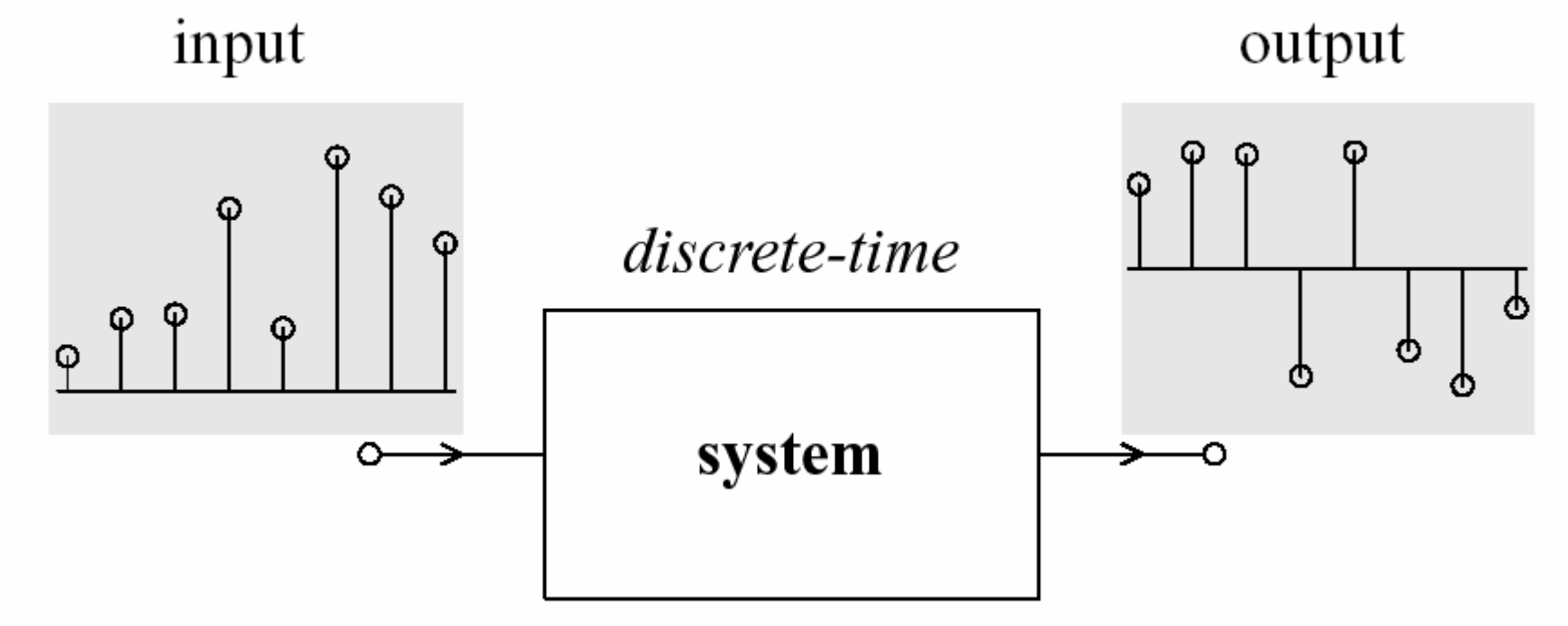




DSP Solution

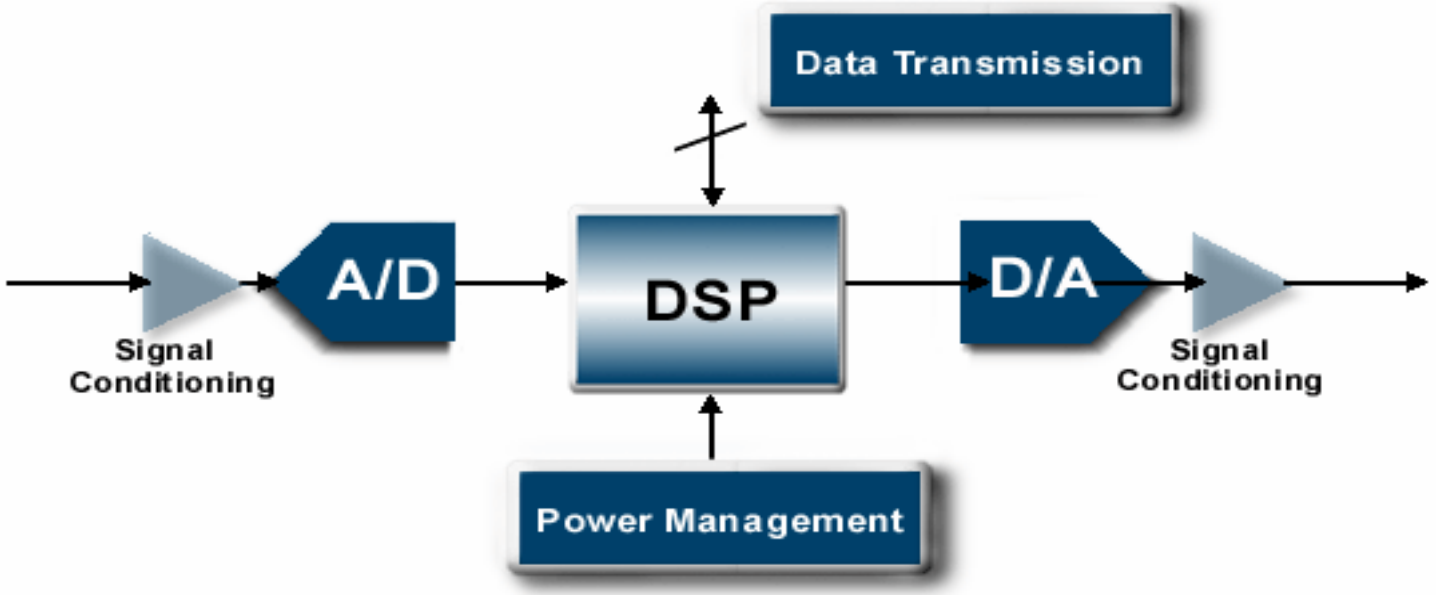


Discrete-time system



Discrete-time system has discrete-time input and output signals

The basic components of a DSP system (add,multiply,delay)



$$y[n] = \sum_{k=-\infty}^{\infty} h[k]\, x[n-k]$$

DSPs operation

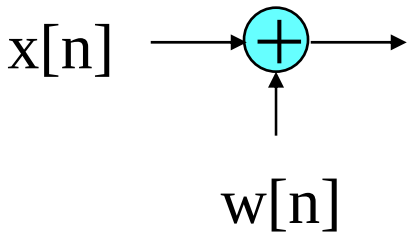
- **DSPs operate synchronously on fixed-word-length samples of data** that arrive at regular intervals of time
- The instruction sets of a DSP typically includes the **two most fundamental arithmetic operations:**
- Multiplication and Addition (Accumulation), commonly referred to as

Multiply and ACcumulate (MAC)

- MAC functional units must be implemented efficiently and must give high performance

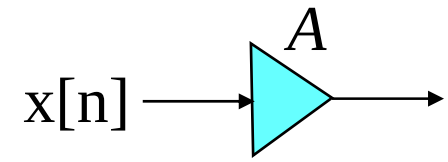
Basic Operations

- **Addition operation:**

  –**Adder** 

y[n]=x[n]+w[n]

- **Multiplication operation**

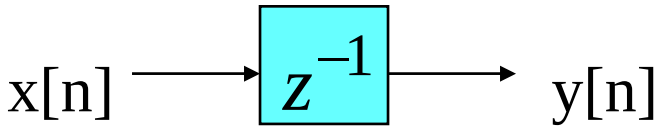
  –**Multiplier** 

y[n]=A.x[n]

Basic Operations

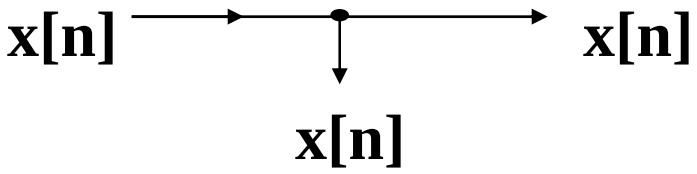
- **Time-shifting operation, where N is an integer**

  –Unit delay



y[n]=x[n-1]

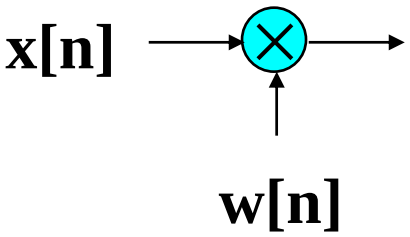
- **Branching operation: Used to provide multiple copies of a sequence**



Another Basic Operation

- **Product (modulation) operation:**

Modulator

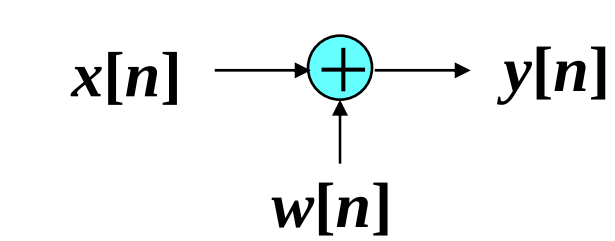


$$y[n]=x[n].w[n]$$

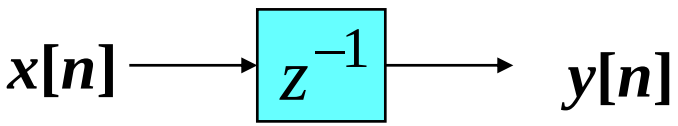
- **An application is in forming a finite-length sequence from an infinite-length sequence by multiplying the latter with a finite-length sequence called an window sequence**
- **The process is called windowing**

Basic Building Blocks

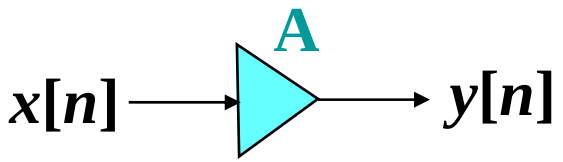
- The computational algorithm of **an LTI digital filter** can be conveniently represented in block diagram form using the basic building blocks shown below



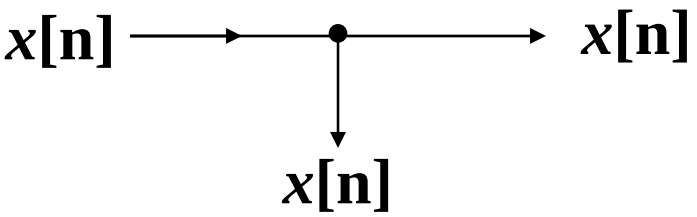
Adder



Unit delay



Multiplier



Pick-off node

The general algorithms of DSP's

- To consider the use of Verilog to model functional units that :
- **Encode of signals**
- **Transmition of signals**
- **Transformation of signals**
- **Filtering of signals**
- **Correlation of signals**
- **Convolution of signals**

The implementation of DSPs

Three ways:

① **Hardware**

② **Software**

③ **Combination of both**

- execute a DSP algorithm on a general-purpose processor
- execute a DSP algorithm on a special purpose, hardwired, high performance, customized processor

Choosing an FPGA: Practical onsiderations Harry

Joseph Fourier

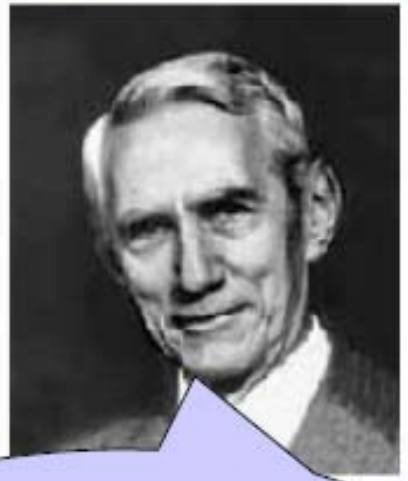
What is a microprocessor?

Leonhard Euler

But what is the FPGA?

Harry Nyquist

Claude Shannon

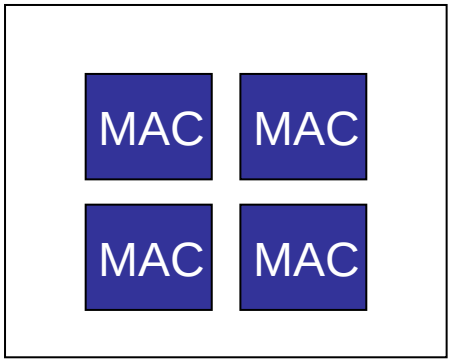


That's easy! You'll be a software guru in no time

FPGA-based implementation requires a unique set of skills not often found in the DSP community

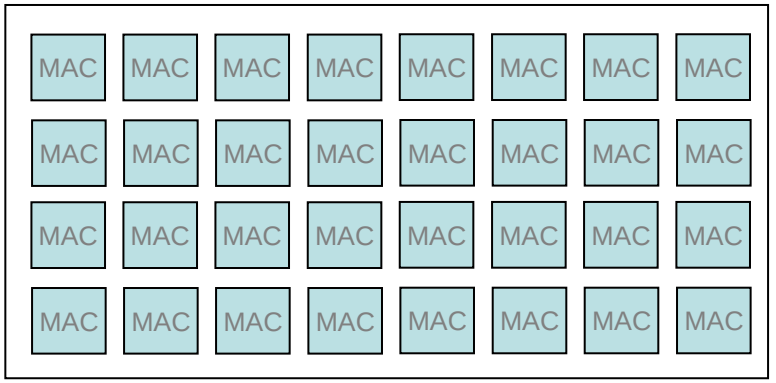
DSP Processors vs. FPGAs

High Speed DSP Processor



- 1-8 Multipliers
  - Needs looping for more than 8 multiplications
- Needs multiple clock cycles because of serial computation
  - 200 Tap FIR Filter would need 25+ clock cycles per sample with an 8 MAC unit processor

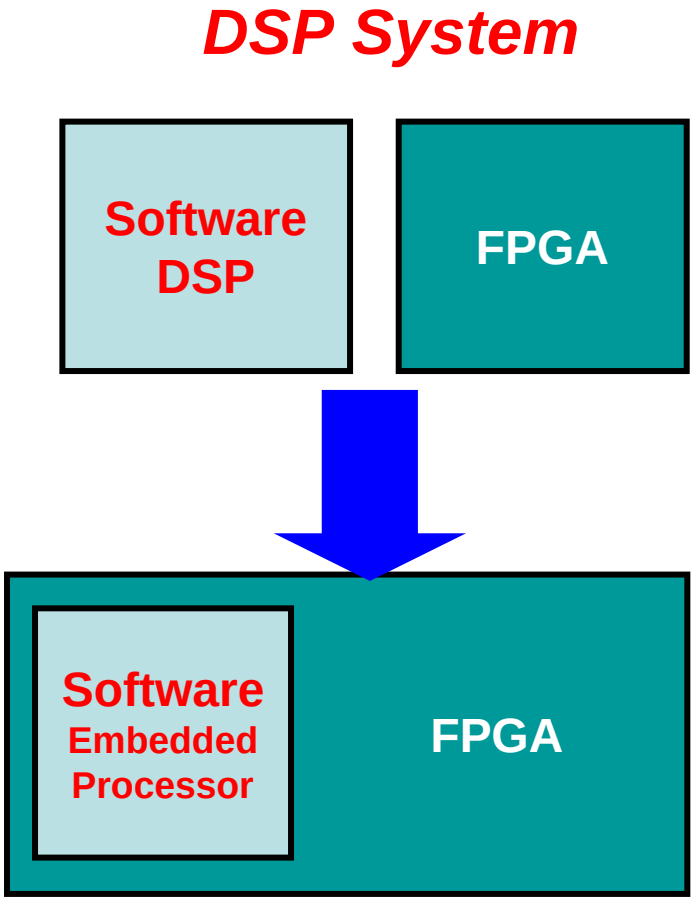
High Level of Parallel Processing in FPGA



- Can implement hundreds of MAC functions in an FPGA
- Parallel implementation allows for faster throughput
  - 200 Tap FIR Filter would need 1 clock cycle per sample

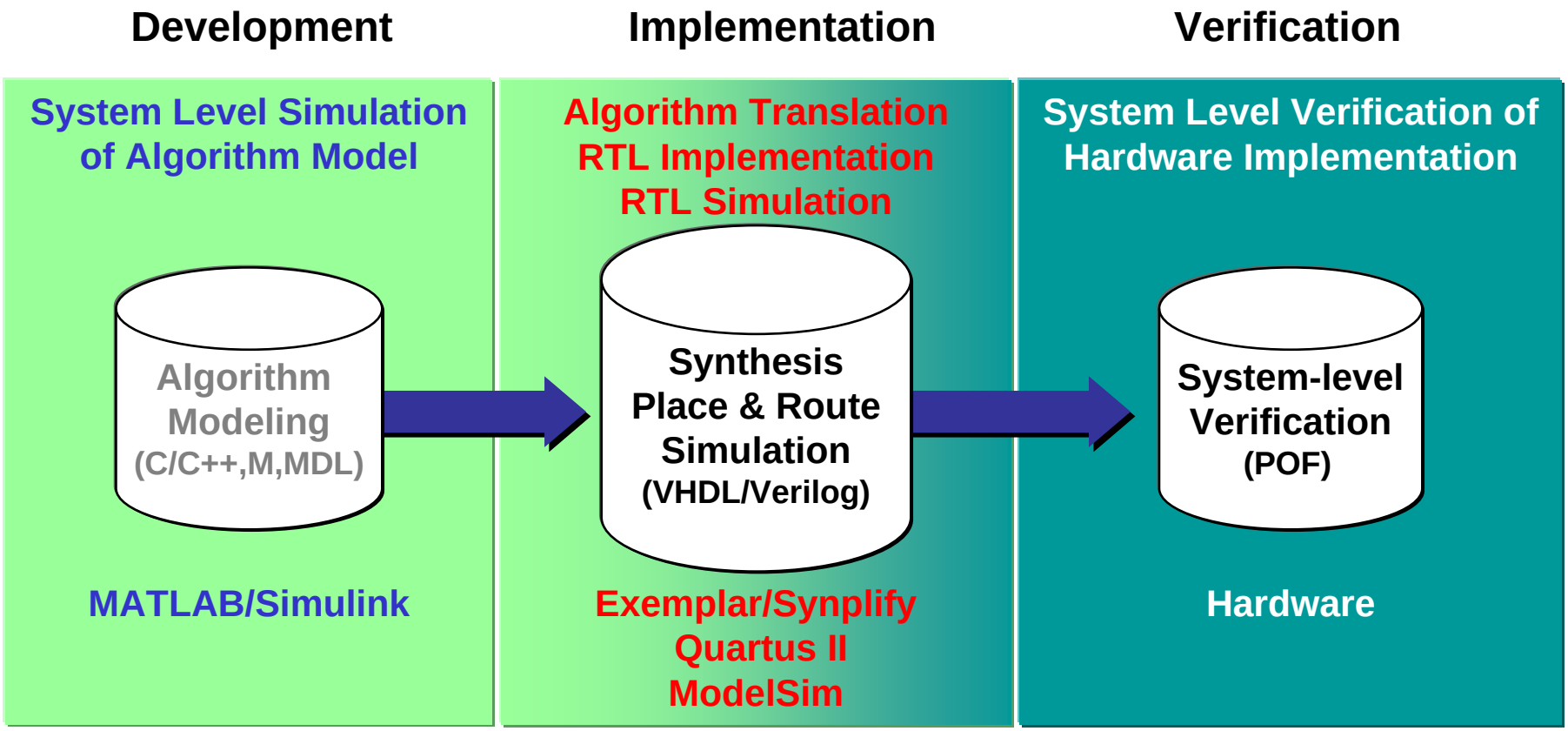
Why Use FPGAs in DSP Applications?

- 10x More DSP Throughput Than DSP Processors
  - Parallel vs. Serial Architecture
- Cost-Effective for Multi-Channel Applications
- Flexible Hardware Implementation
- Single-Chip Solution
  - System (Hardware/Software) Integration Benefits



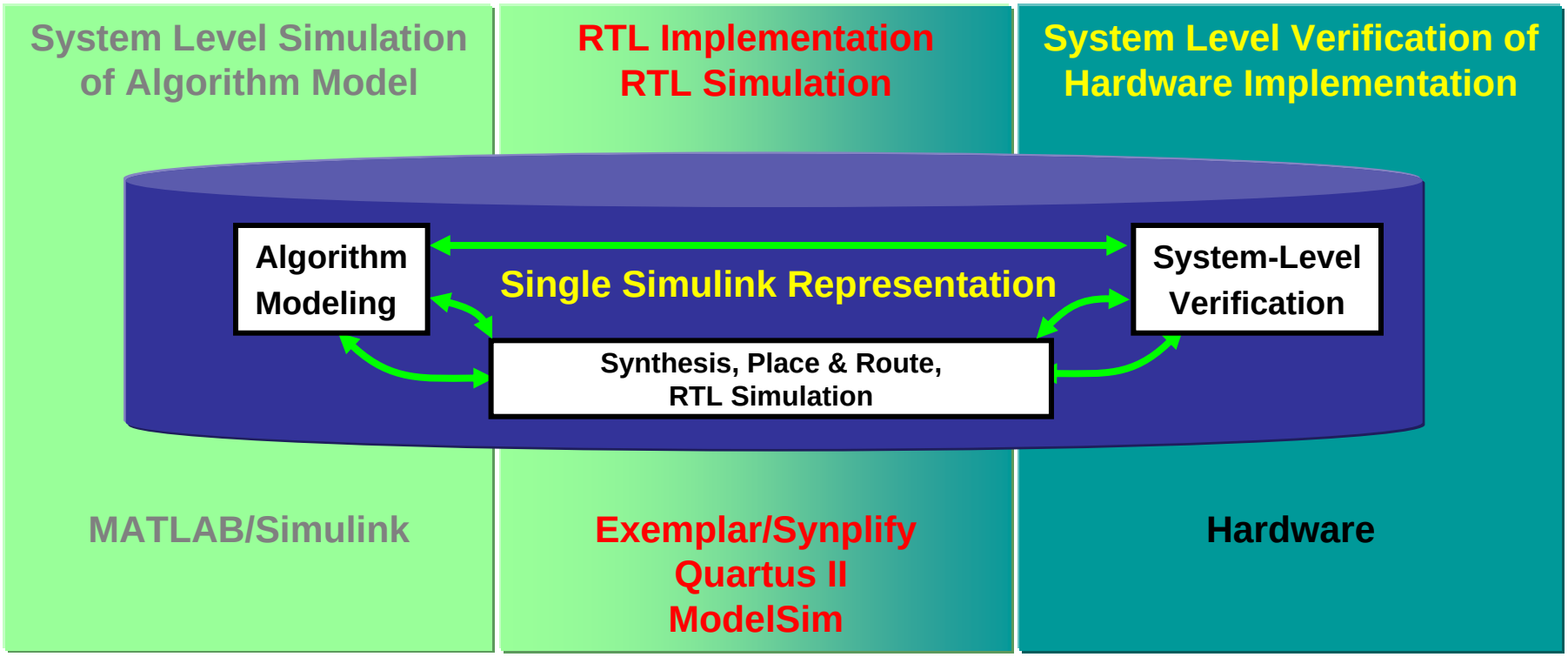
DSP Design Flow in FPGA

- System Algorithm Design & FPGA Design Separate



Integration Using DSP Builder

- System Algorithm Design & FPGA Design Integrated



The general algorithms of DSP's

- To consider the use of Verilog to model functional units that :
- **Filtering of signals**
- **Encode of signals**
- **Transmition of signals**
- **Transformation of signals**
- **Correlation of signals**
- **Convolution of signals**

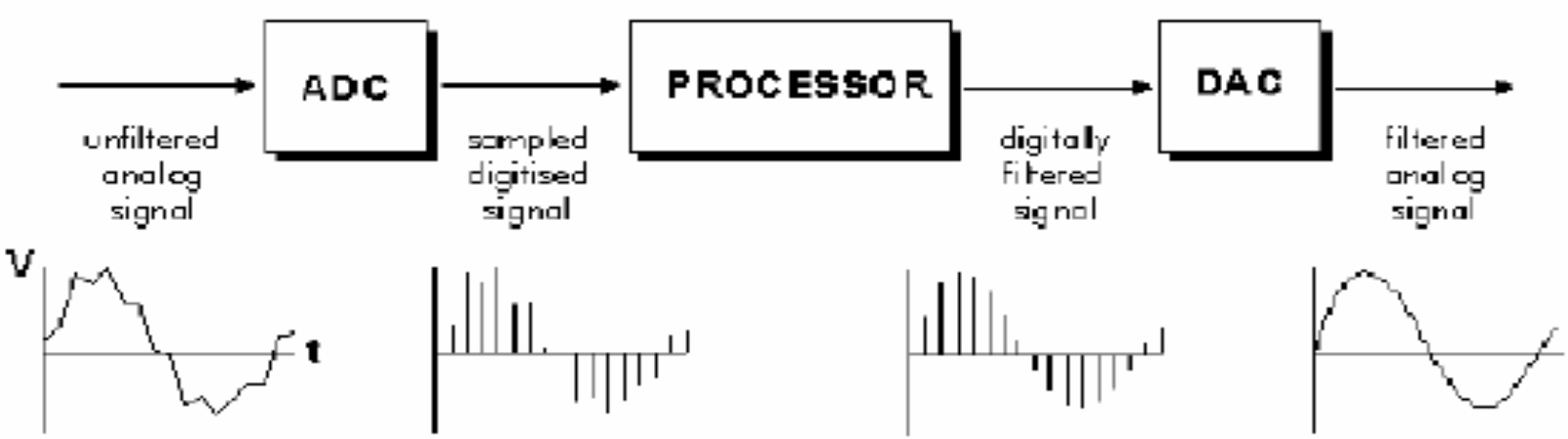
Digital filters

- A digital filter is essentially **"a system designed to extract information about a prescribed quantity of interest from noisy data".**

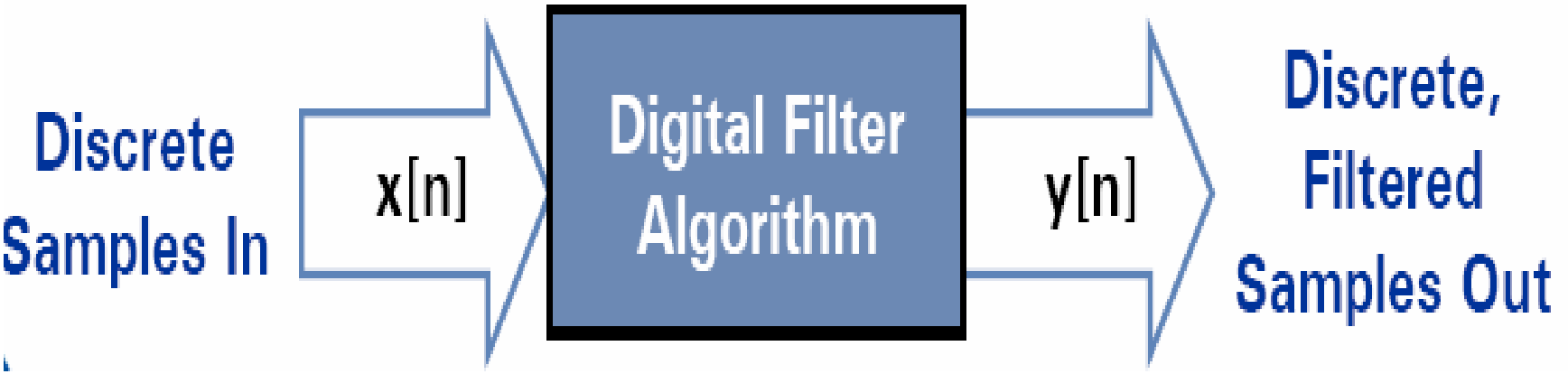
- Characterization of digital filters:
  - Impulse response
  - Transfer function
  - Frequency response

Digital filters

- To remove noise and other unwanted signal components and to shape the spectrum characteristics of the resulting signal
- To operate on a finite-precision digital representation of a signal
- Design must consider finite word length effects that result from the representation of the signal samples, the weighting coefficients of the filter, and the arithmetic operations performed by the filter

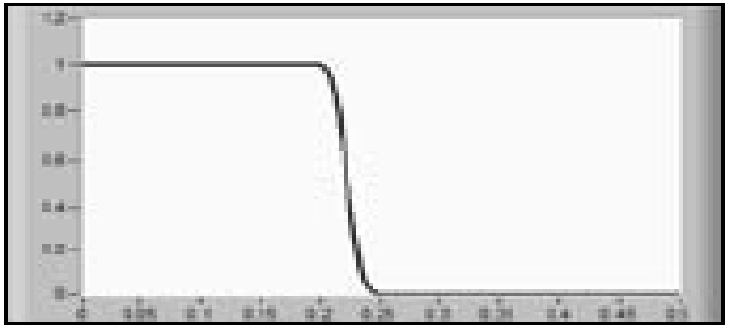


What is the Digital Filter ?



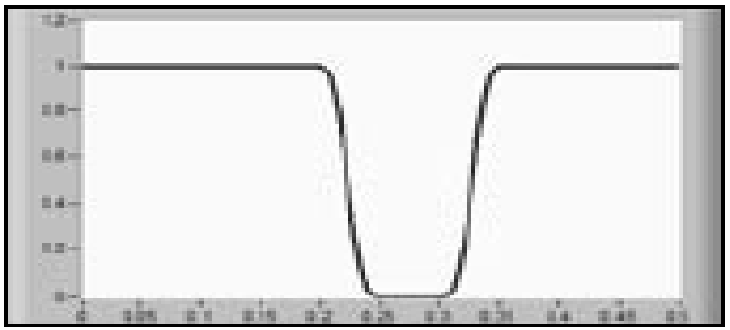
- **As a function of some combination of**
- **– Current input samples**
- **– Past input samples**
- **– Past output samples**

Filters alter the frequency spectrum of an input signal

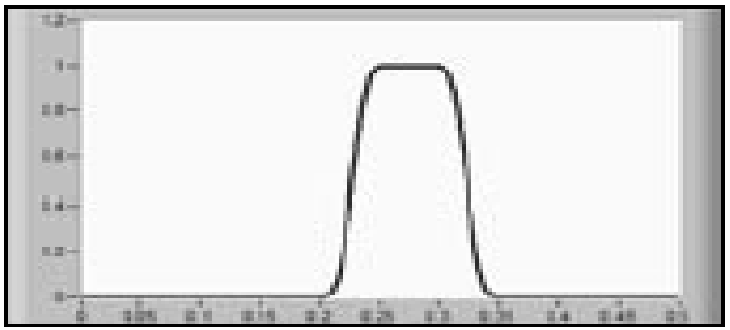


Lowpass

Highpass



Bandstop



Bandpass

The constraint

- A DSP unit is **constrained by the physical technology** in which it is implemented, which fundamentally **limits the speed** and also **determines the physical area**
- A DSP may be constrained **by the number of channels of data, the rate at which data are exchanged** with the machine's environment, and **the size of the input and output words**

DSP's distributed spatially & temporally

- The operations of a DSP unit can be distributed spatially or temporally, **depending on whether the unit executes its operations in a single cycle of the clock or over multiple cycles**
- **In space**, **the hardware resources must complete the operation within the period of the clock**
- **In time, the machine operates on part of the data word in each clock cycle**, so that the capacity and performance of the individual operational units can be relaxed

Distributing operations over the temporal axis

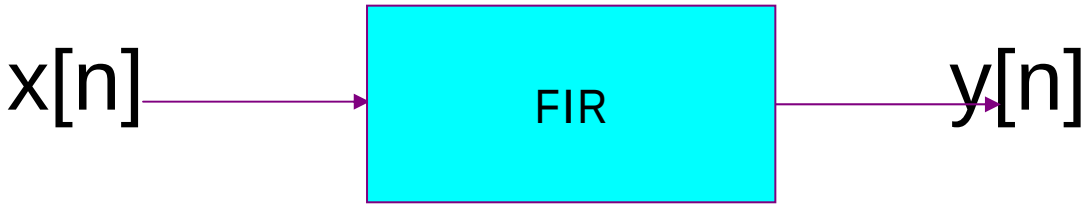
- Distributing operations over the temporal axis allows the machine to operate with higher throughput, but at the expense of a latency between the arrival of data and the availability of the results
- **Latency can be tolerated in many applications, such as in digital communications.**

Digital filter design

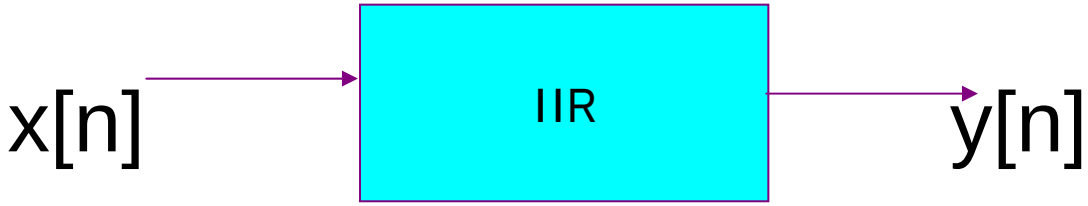
- ***Digital filter design*** is a process in which we **construct a digital hardware or a program (software) that meets the given specification**
- The design starts with the ***specification***, and it consists of four basic steps:
  - **approximation**
  - **realization**
  - **study of imperfections**
  - **implementation**

The kinds of digital filter

- FIR: finite-duration impulse response



- IIR:infinite-duration impulse response



FIR and IIR Filters

- **Digital filter's two important characteristics:**

  Causality and Linear phase

- ***FIR (Finite Impulsive Response)*:**

  it is a **non-recursive** filter. Output depends **only on present and previous inputs**; its impulse response has **a finite impulse response for n > M > 0** ,Integer number M is called the length of impulse response

- ***IIR(Infinite Impulsive Response)*:**

  it is a recursive filter. **Output depends on one or more previous output values**

FIR, IIR Digital Filter

- Let {h[n}: impulse response
  {x(n)}: input, {y(n)}: output
- Finite impulse response (FIR) filter:

$$y(n)=\sum_{j=0}^{J-1}h(j)x(n-j)$$

- Computation is the same as convolution.
- Impulse input:

$$\delta(n)=\begin{cases}1 & n=0,\\ 0 & n\neq 0.\end{cases}$$

  if x(n)=δ(n), y(n)=h(n) is the impulse response that has finite extent.

- Infinite impulse response (IIR) filter

$$y(n)=\sum_{i=1}^{P}a(i)y(n-i)+\sum_{k=0}^{Q}b(k)x(n-k)$$

- The length of {y(n)} may be infinite!
- Recursive formula will impact on computation methods
- Stability concerns:
  - The magnitude of y(n) may become infinity even all x(n) are finite!
  - coefficient values,
  - quantization error

The distinctions between the two types of filters

- The distinction between the two types of filters is that FIR filters cannot accumulate **round-off error**;
- IIR filters can accumulate round-off error as the output is successively passed through the filter

Advantages of FIR filters

- FIR filters are simple to design;
- they are guaranteed to be bounded input-bounded output (BIBO) stable.
- FIR filter can be guaranteed to have linear phase.
- FIR filters also have a low sensitivity to filter coefficient quantization errors.

$$y(n) = \sum_{j=0}^{J-1} h(j)\,x(n-j)$$

Advantages of IIR filters

- IIR filters are useful for high-speed designs because they typically require a lower number of multiplies compared to FIR filters.
- IIR filters can be designed to have a frequency response that is a discrete version of the frequency response of an analog filter.

$$y(n) = \sum_{i=1}^{P} a(i)y(n-i) + \sum_{k=0}^{Q} b(k)x(n-k)$$

FIR & IIR Filter Design

- FIR filters
  - Linear-phase FIR filters
  - FIR design by optimization

    Weighted least-squares design, Minimax design
  - FIR design in practice

    `Windows', Equiripple design, Software (Matlab,…)
- IIR filters
  - Poles and Zeros
  - IIR design by optimization

    Weighted least-squares design, Minimax design
  - IIR design in practice

    Analog IIR design : Butterworth/Chebyshev/elliptic

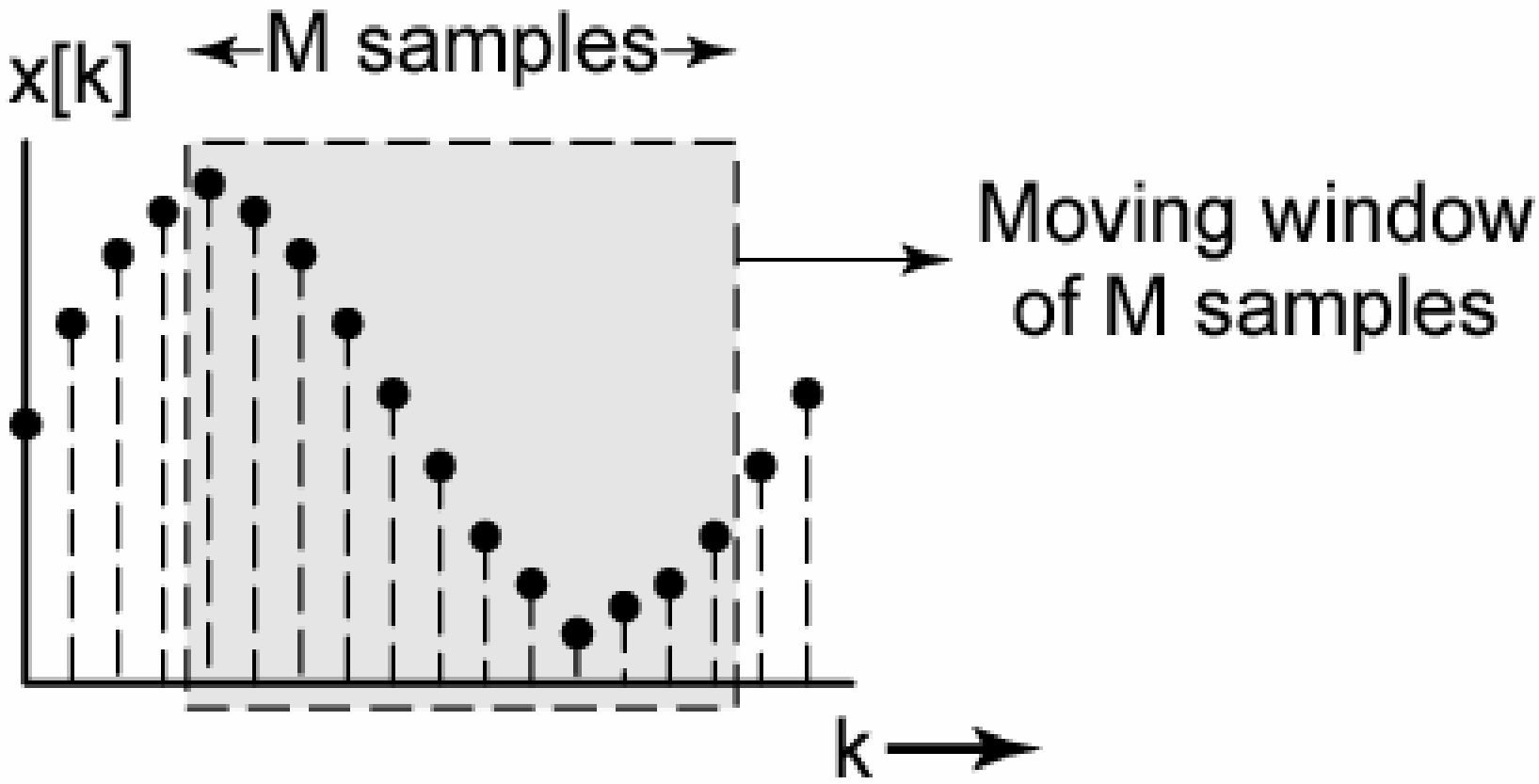
    Analog->digital : impulse invariant, bilinear transform,…

    Software (Matlab)

9.3.1 Finite-Duration Impulse Response (FIR) Filter

- A FIR digital filter forms its **output as a weighted sum of present and past samples of its input**
- FIR filters are called **moving average filters** because their output at any time index depends on a window containing only the most recent M samples of the input
- Because its response depends on only a finite record of inputs, a FIR filter will have a finite-length, nonzero response to a discrete-time impulse

Fig 9-21 a window containing only the most recent M samples of the input



Basic FIR Digital Filter Structures

- A causal FIR filter of order N is characterized by a transfer function H(z) given by

$$H(z) = \sum_{n=0}^{N} h[n] z^{-n}$$

which is a polynomial in z^{-1}

- In the time-domain , A FIR digital filter forms its output as a weighted sum of present and past samples of its input
- The input-output relation of the above FIR filter is given by

$$y[n] = \sum_{k=0}^{N} h[k] x[n-k]$$

FIR Filters

FIR filter = finite impulse response filter

$$H(z) = \frac{B(z)}{z^N} = b_0 + b_1 z^{-1} + \ldots + b_N z^{-N}$$

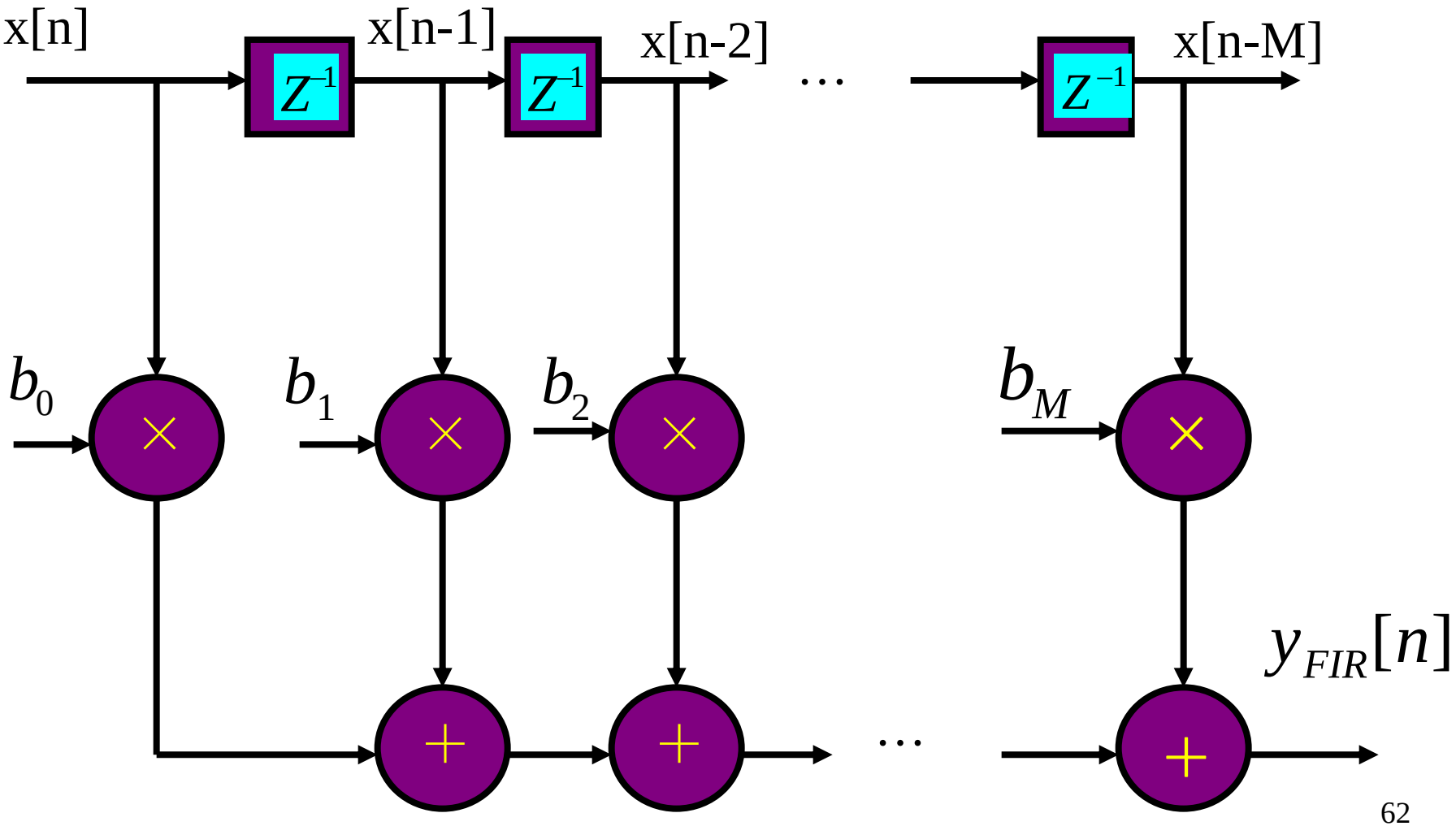
- **Also known as `moving average filters' (MA)**
- **N poles at the origin z=0 (hence guaranteed stability)**
- **N zeros (zeros of B(z)), `all zero' filters**
- **corresponds to difference equation**

$$y[k] = b_0 .u[k] + b_1 .u[k-1] + \ldots + b_N .u[k-N]$$

- **impulse response**

$$h[0] = b_0, h[1] = b_1, \ldots, h[N] = b_N, h[N+1] = 0, \ldots$$

Fig 9.22 Functional block diagram for an Mth-order FIR digital filter



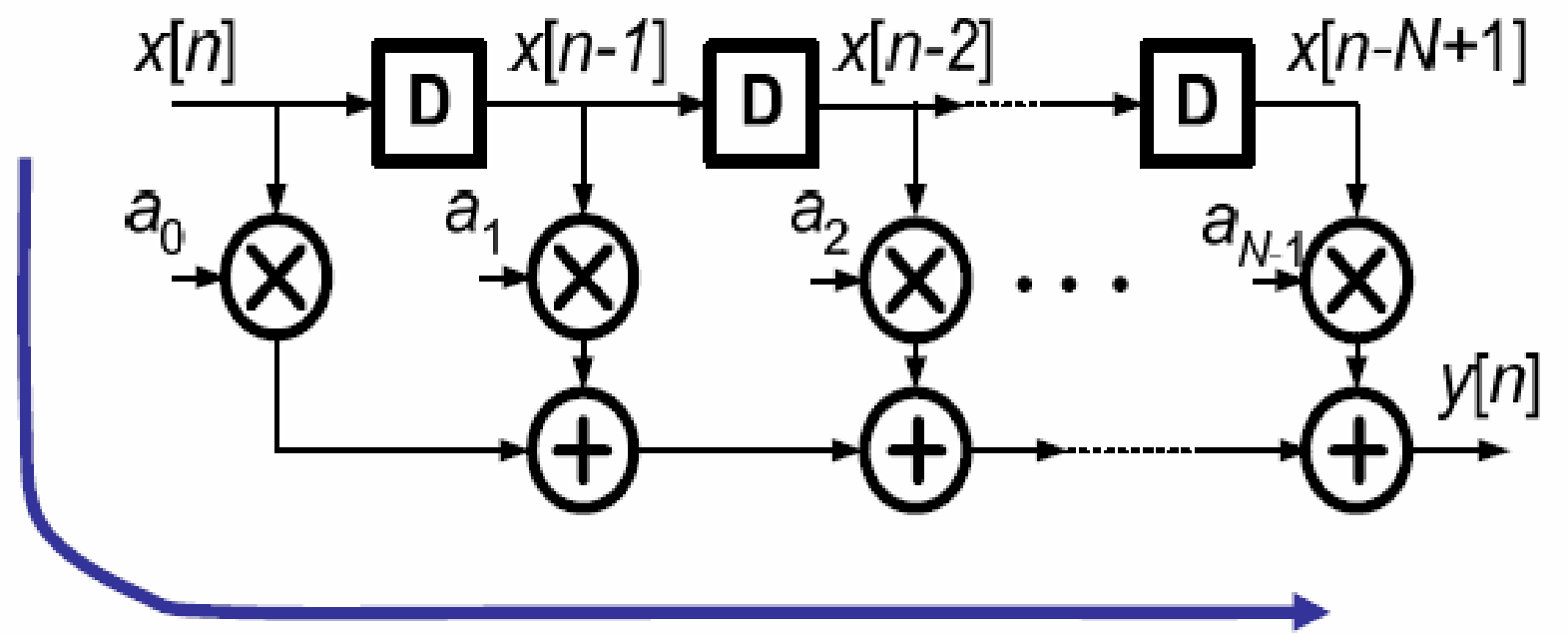
An Mth order FIR

- **An Mth order FIR will have M+1 taps**
- **The adders and multipliers of the filter must be fast enough to form y[n] before the next clock, and at each stage they must be sized to accommodate the width of their datapaths**
- In applications in which numerical accuracy is a driving consideration, **lattice architectures** may reduce the effects of finite word length, but at the expense of increased computational cost

In most applications

- The goal of the implementation is to do the filtering as fast as possible to achieve the highest sampling frequency
- **The longest signal path through the combinational logic includes M stages of addition and one stage of multiplication**

The longest signal path : Critical Path



$$T = T_{mult} + (N\text{-}1)T_{add}$$

The architecture of a FIR

- **The architecture consists of a shift register, multipliers, and adders implementing an Mth-order FIR**
- **The datapaths must be wide enough** to accommodate the output of the multipliers and adders
- The samples are encoded as finite-length words, then shifted in parallel through a series of M registers
- A cascaded chain of MACs form the machine

Linear Phase FIR Filters

Type-1	Type-2	Type-3	Type-4
N=2L=even	N=2L+1=odd	N=2L=even	N=2L+1=odd
symmetric	symmetric	anti-symmetric	anti-symmetric
h[k]=h[N-k]	h[k]=h[N-k]	h[k]=-h[N-k]	h[k]=-h[N-k]
$e^{-j\omega N/2}.\sum_{k=0}^{L} d_k.\cos(\omega.k)$	$e^{-j\omega N/2}\cos(\frac{\omega}{2}).\sum_{k=0}^{L} d_k.\cos(\omega.k)$	$j.e^{-j\omega N/2}\sin(\omega).\sum_{k=0}^{L-1} d_k.\cos(\omega.k)$	$j.e^{-j\omega N/2}\sin(\frac{\omega}{2}).\sum_{k=0}^{L} d_k.\cos(\omega.k)$
LP/HP/BP	zero at $\omega=\pi$ LP/BP	zero at $\omega=0,\pi$ BP	zero at $\omega=0$ HP

PS: `modulating' Type-2 with 1,-1,1,-1,.. gives Type-4 (LP->HP)
PS: `modulating' Type-4 with 1,-1,1,-1,.. gives Type-2 (HP->LP)
PS: `modulating' Type-1 with 1,-1,1,-1,.. gives Type-1 (LP<->HP)
PS: `modulating' Type-3 with 1,-1,1,-1,.. gives Type-3 (BP<->BP)

PS: IIR filters can NEVER have linear-phase property ! (proof see literature)

Fig 9-23 MAC-based architecture for a type-I,Mth-order FIR filer

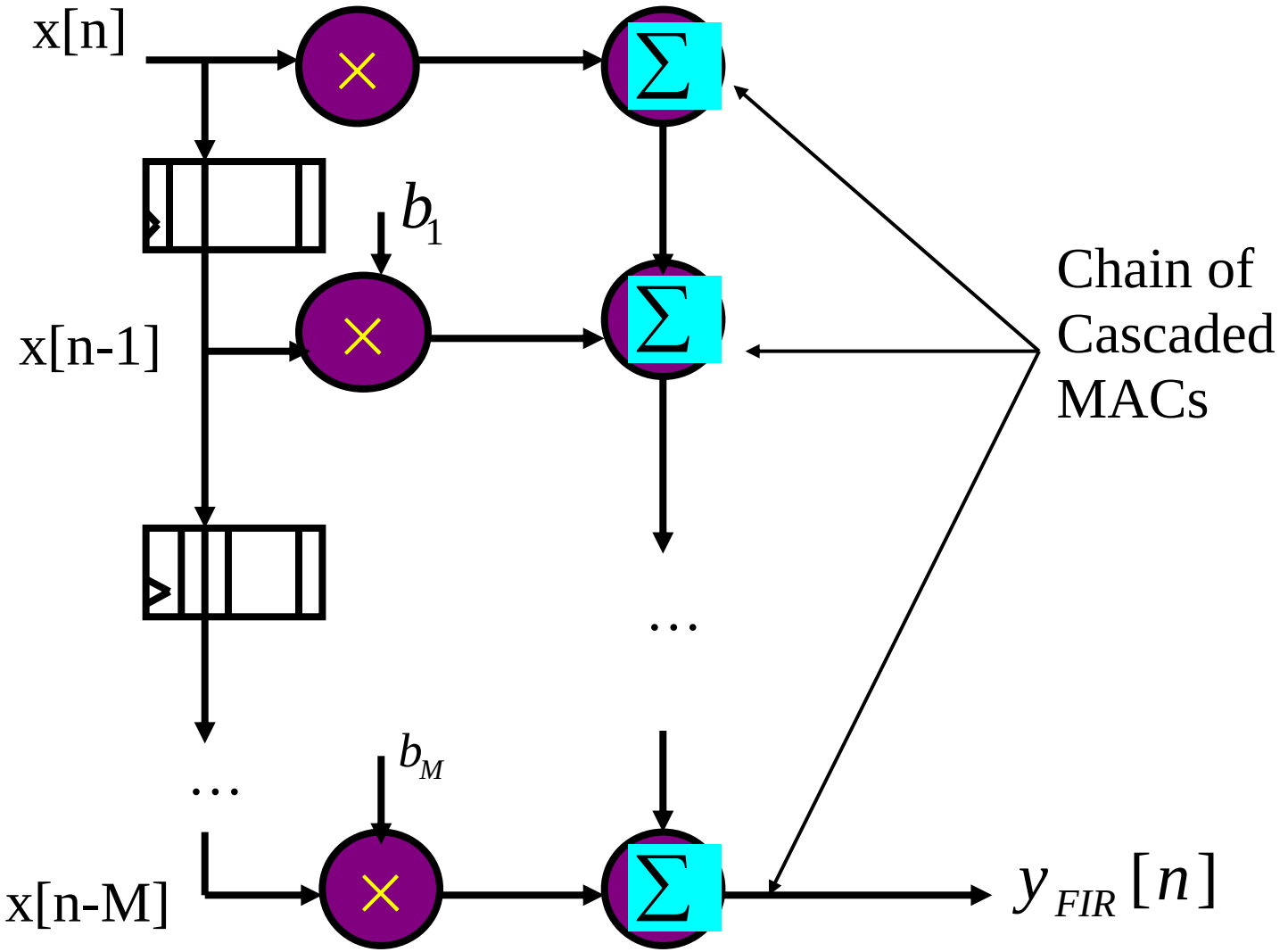
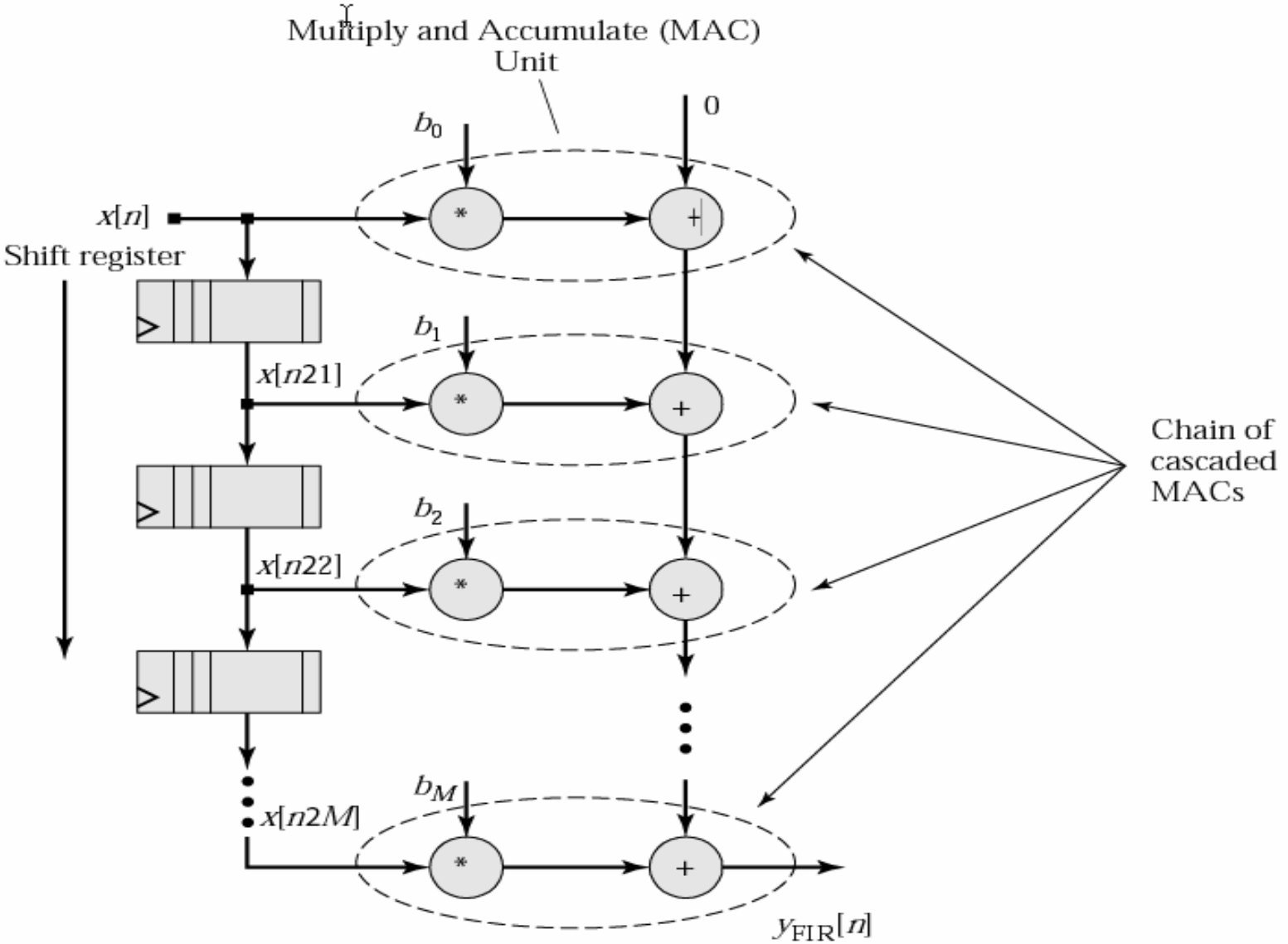


Fig 9-23 MAC-based architecture for a type-I,Mth-order FIR filer

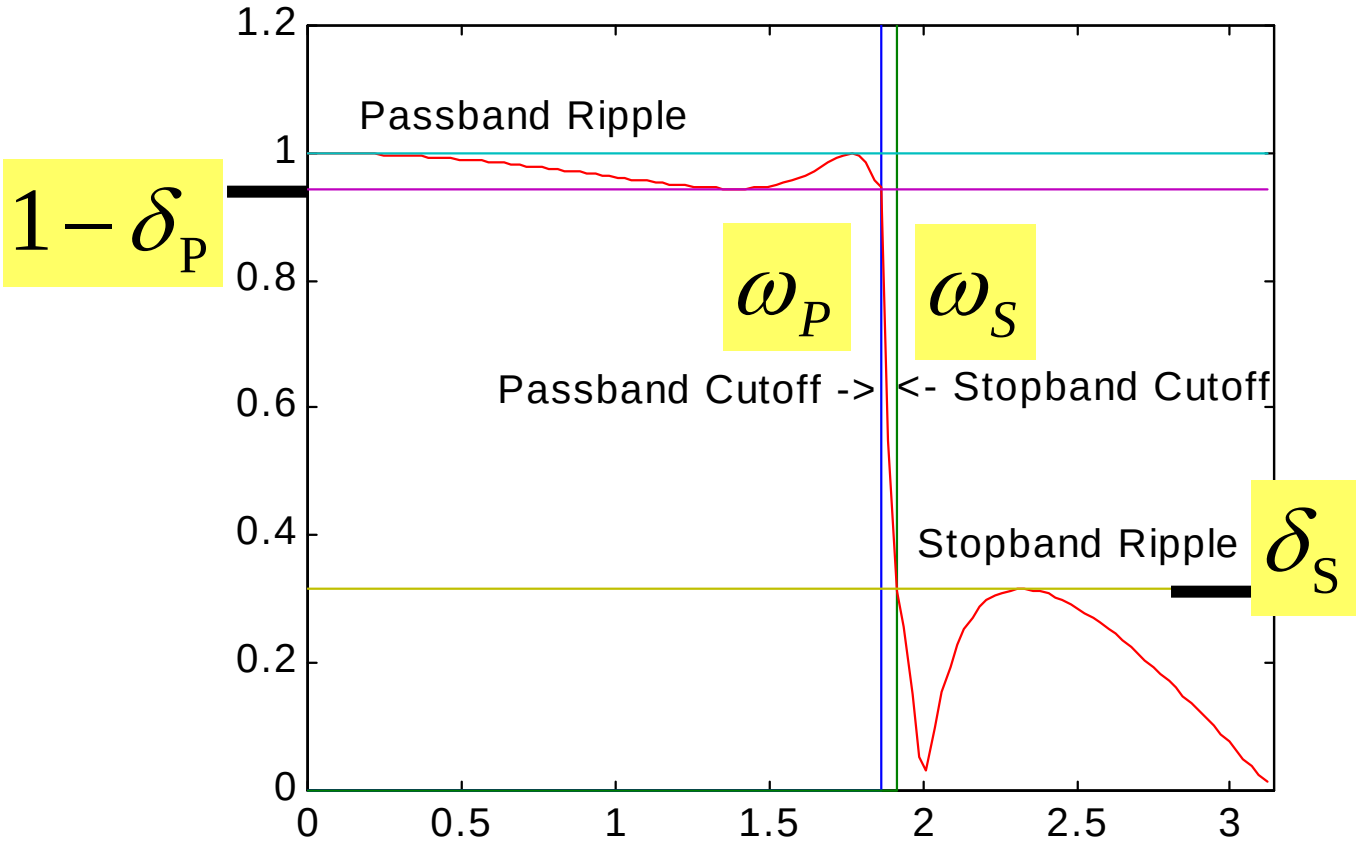


9.3.2 Digital Filter Design Process

- A design begins with development of performance **specifications for**
  - ✓**cutoff frequency,**
  - ✓**transition band limits,**
  - ✓**in-band ripple,**
  - ✓**minimum stop band attenuation**, etc
- A process for designing an ASIC- or FPGA-based digital filter has the main steps shown in Following Figure

Filter Specification

Ex: Low-pass



Filter Design/Realization

- **Step-1 : define filter specs**

  (pass-band, stop-band, optimization criterion,…)

- **Step-2 : derive optimal transfer function**

  FIR or IIR design

- **Step-3 : filter realization (block scheme/flow graph)**

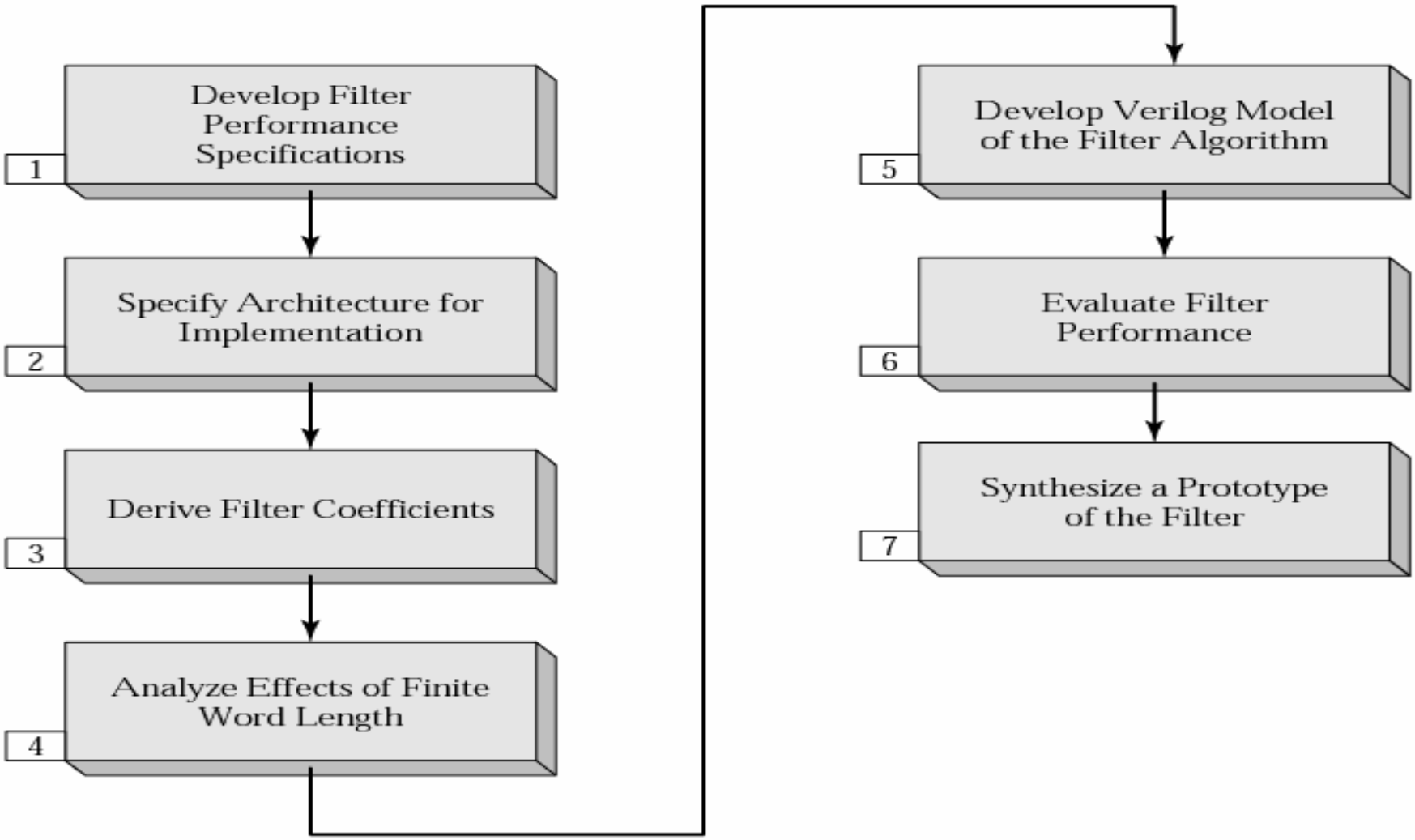
  direct form realizations, lattice realizations,…

- **Step-4 : filter implementation (software/hardware)**

  finite word-length issues, …

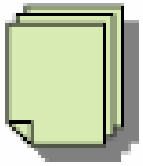
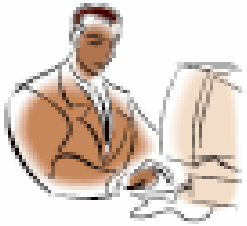
  question: implemented filter = designed filter ?

Fig 9.24 Digital Filter Design Process



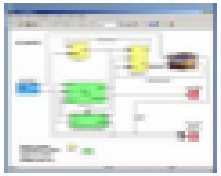
Design Process

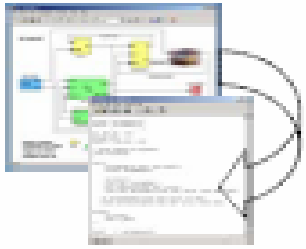
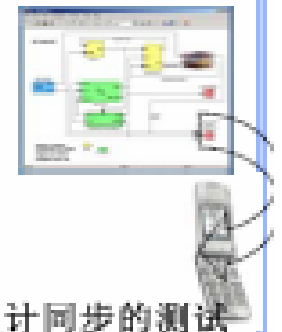
需求与规范 | 设计 | 实现 | 测试与验证

详细开发的模型

连续不断的测试



可执行的规范
准确、无二义性错误

仿真
产生真正的原型、系统的"what-if"分析

自动代码生成
减少编写代码的错误

与设计同步的测试
较早发现错误

The distinctions between Verilog & C

- The design flow is not ideal, because the algorithm's description in C must be translated into Verilog, creating the possibility for errors to occur
- In C, the variables can be represented as floating-point numbers, but in Verilog the parameters and other data values are expressed in a fixed-point, finite-word-length format

The quantization errors of FIR and IIR filters

- For a given architecture, tools such as MATLAB can be used **to determine the filter coefficients** that implement a filter that satisfies the specifications of the design
- Digital filters operate on finite-word-length representations of physical (analog) values
- **The finite word length of the data limits the resolution and the dynamic range that can be represented by the filter, leading to quantization errors**

The quantization and truncation errors of FIR,IIR

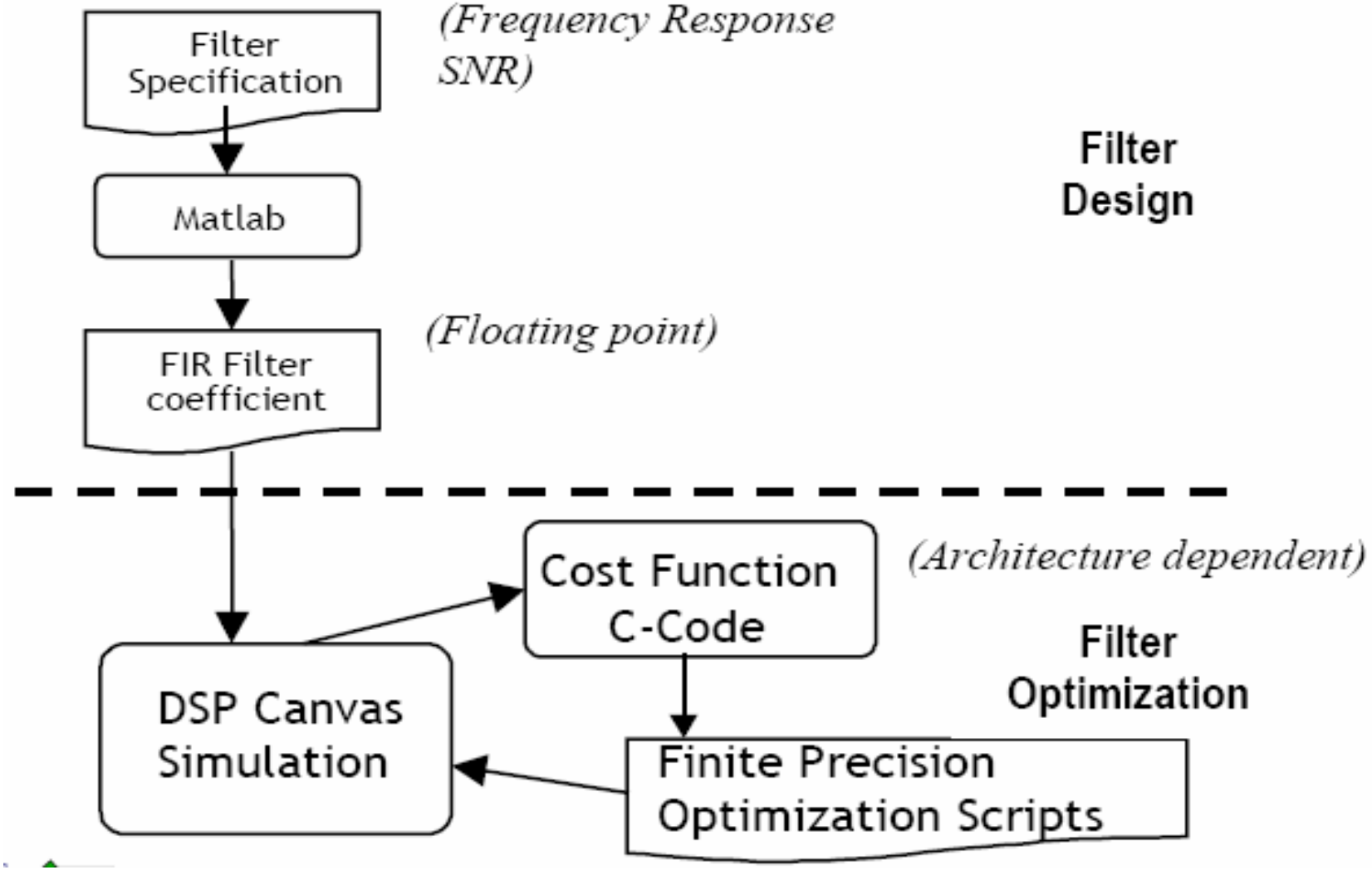
- The representations of **the numerical coefficients of the filter have a finite word length,** which contributes to **additional quantization and truncation error**
- When data are represented by integers there is an error caused by truncation of the fractional part produced by an arithmetic operation
- **The arithmetic operations** that are performed by the filter **can lead to overflow and underflow errors**, which must be detected by the machine

Why Fixed-Point?

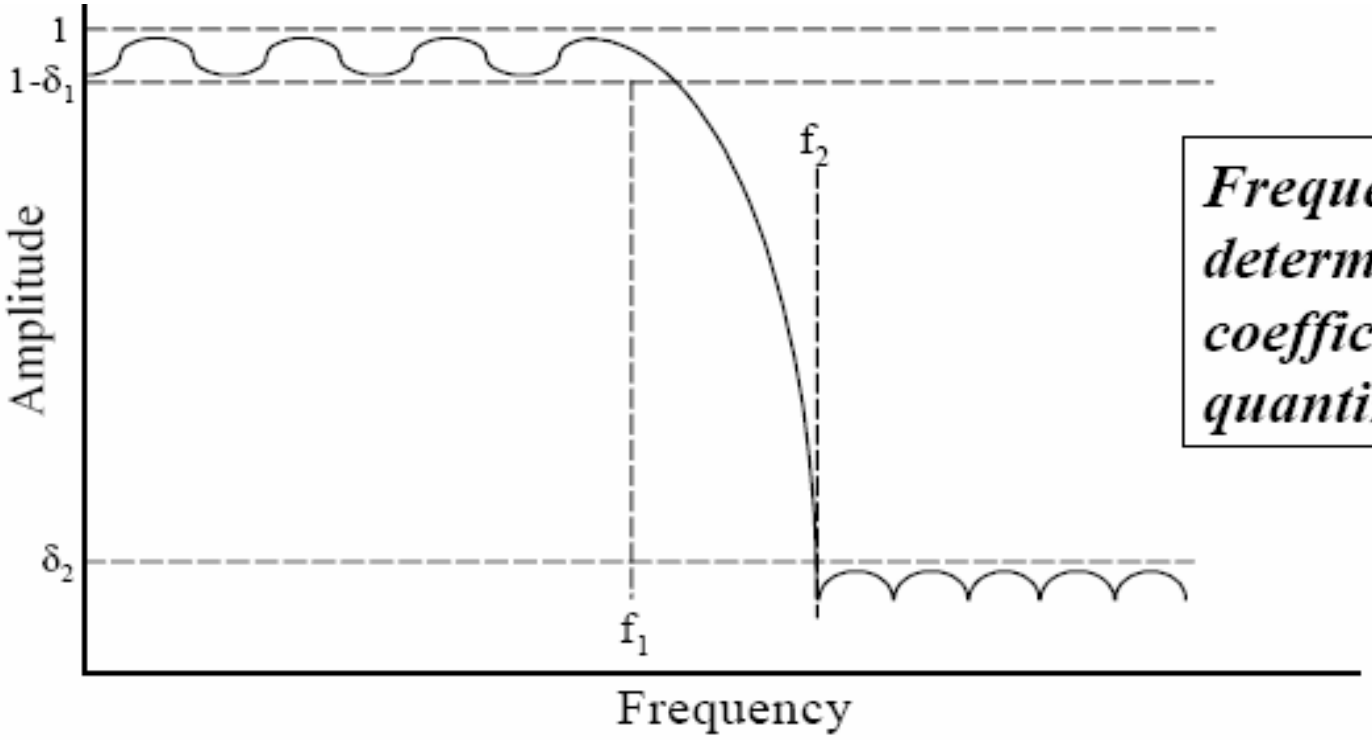
- Floating-point math:
  - Provides excellent dynamic range and accuracy
  - Does not require a designer to worry about overflow, or rounding
  - Is better than fixed-point in every respect but one: it is too expensive
- Executing floating-point math on the DSP uP, FPGA or ASIC, it is difficult to achieve real-time performance.
  - Requires more DSP/µp clock cycles
  - Requires more logic on an FPGA/ASIC

- ***Conversion to the fixed-point isn't straightforward***

Filter Design Flow



FIR System Specification



Frequency response determined by coefficient quantization

$$\mathrm{SNR} = 10\log\frac{E\left(y_{ref}^2\right)}{E\left(\left(y_{ref} - y_{finite}\right)^2\right)}$$

SNR determined by signal precision
i.e. quantization noise

根据过渡带宽选择FIR滤波器窗函数类型和长度M的公式

名称	近似过渡带宽	精确过渡带宽	最小阻带衰减
矩形	4π/M	1.8π/M	21dB
巴特利特	8π/M	6.1π/M	25dB
汉宁	8π/M	6.2π/M	44dB
哈明	8π/M	6.6π/M	51dB
布莱克曼	12π/M	11/M	74dB
取Kaiser窗时用MATLAB中的kaiserord函数来得到长度M			

4 Tap FIR

Input Wordsize (bits)	Resource Usage (LUTs)	No. 18x18 multipliers	Frequency (MSamples/s)	Computations (MMACS/s)
8	109	4	92.2	368.8
12	168	4	73.8	295.5
18	259	4	53.3	213.2

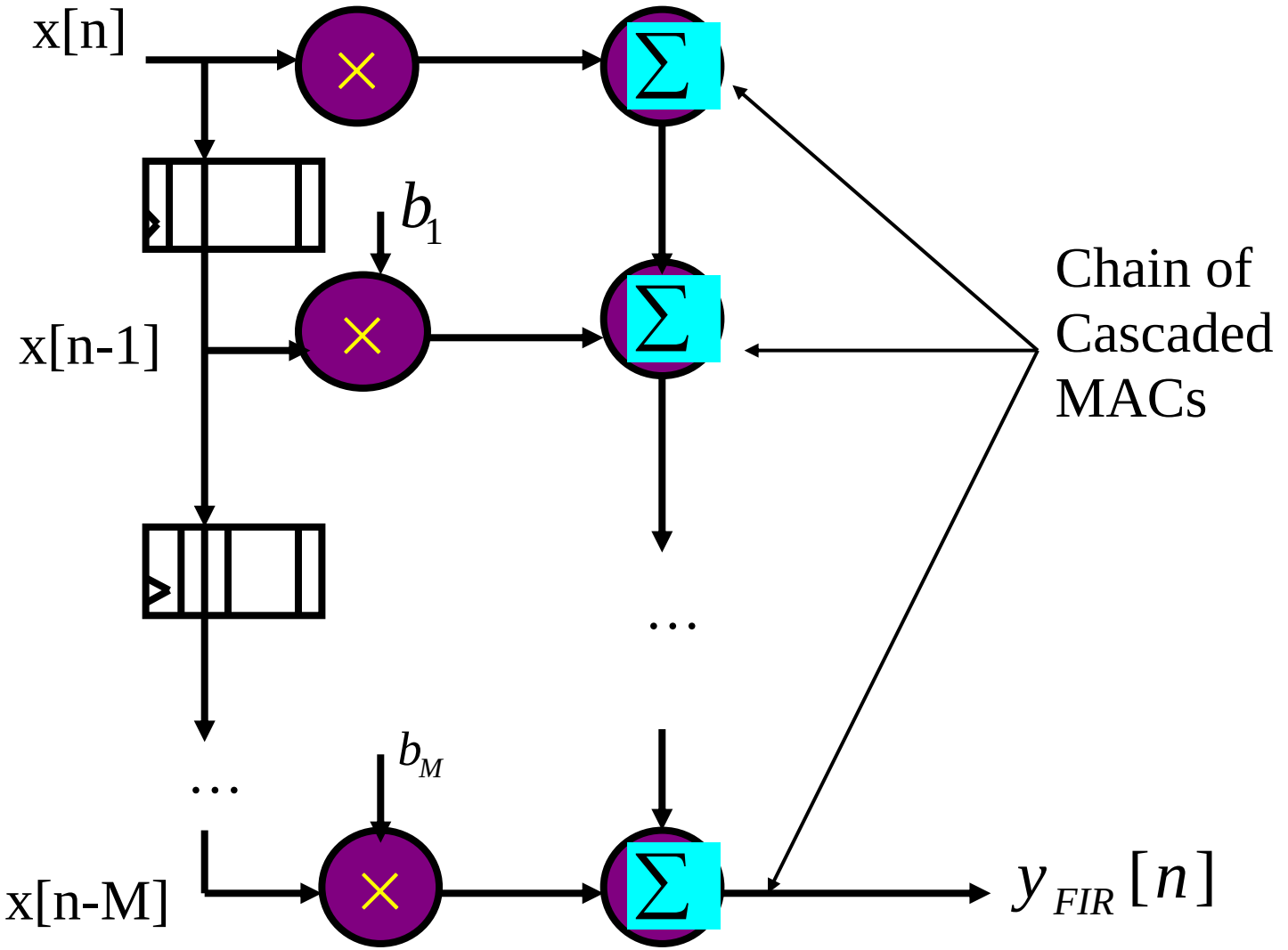
20 Tap FIR

Input Wordsize (bits)	Resource Usage (LUTs)	No. 18x18 multipliers	Frequency (MSamples/s)	Computation (GMAC/s)
8	680	20	94.7	1.894
12	1063	20	73.8	1.476
18	1632	20	54.1	1.082

Example 9.2

- **An eighth-order Gaussian, lowpass FIR filter** is modeled by FIR_Gaussian_Lowpass
- The design is fully synchronous, with active-high synchronous reset
- The filter's tap coefficients are implemented as unsigned 8-bit words (for unsigned integer math), chosen to be even-symmetric to guarantee that the phase characteristic will be linear

MAC-based architecture for a type-I, Mth-order FIR filer

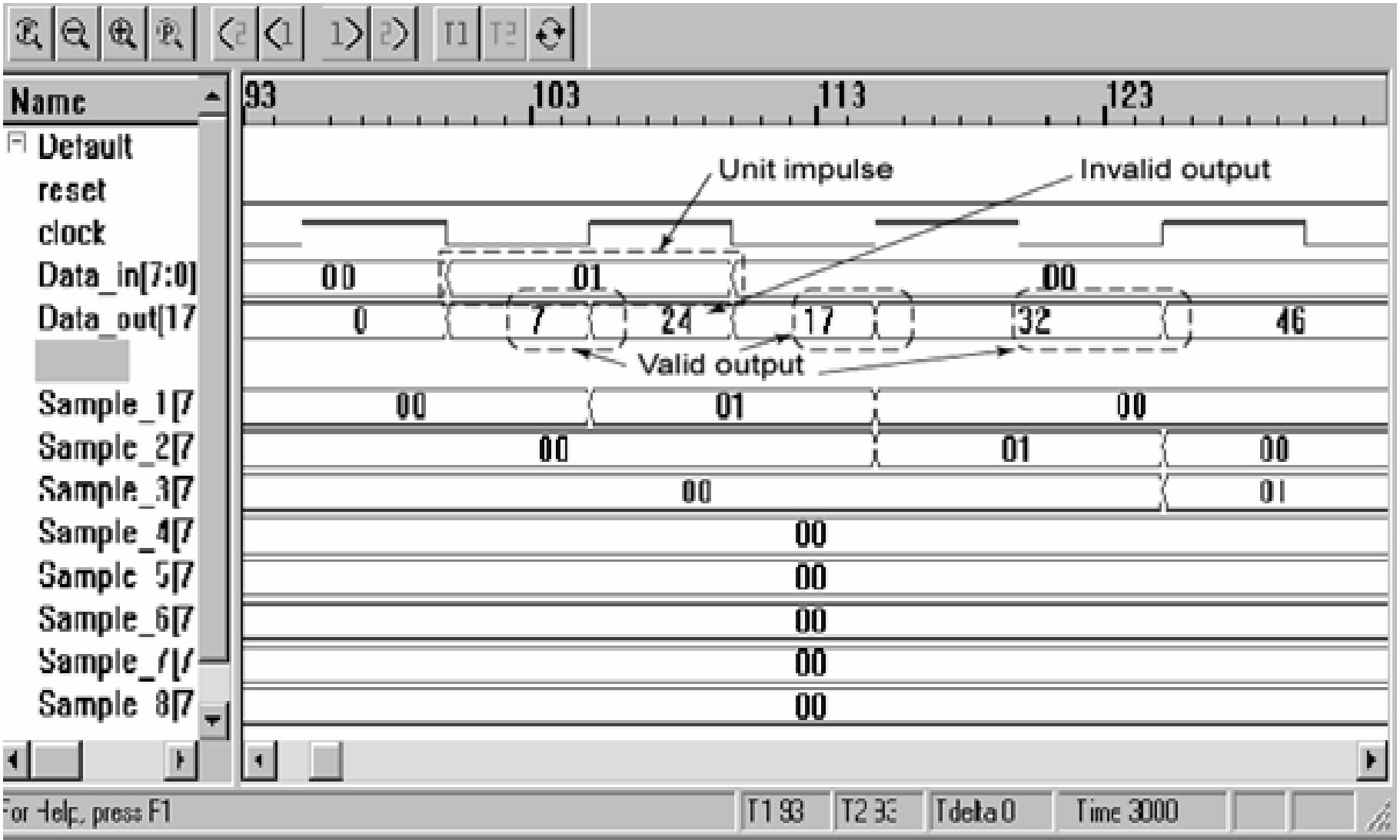


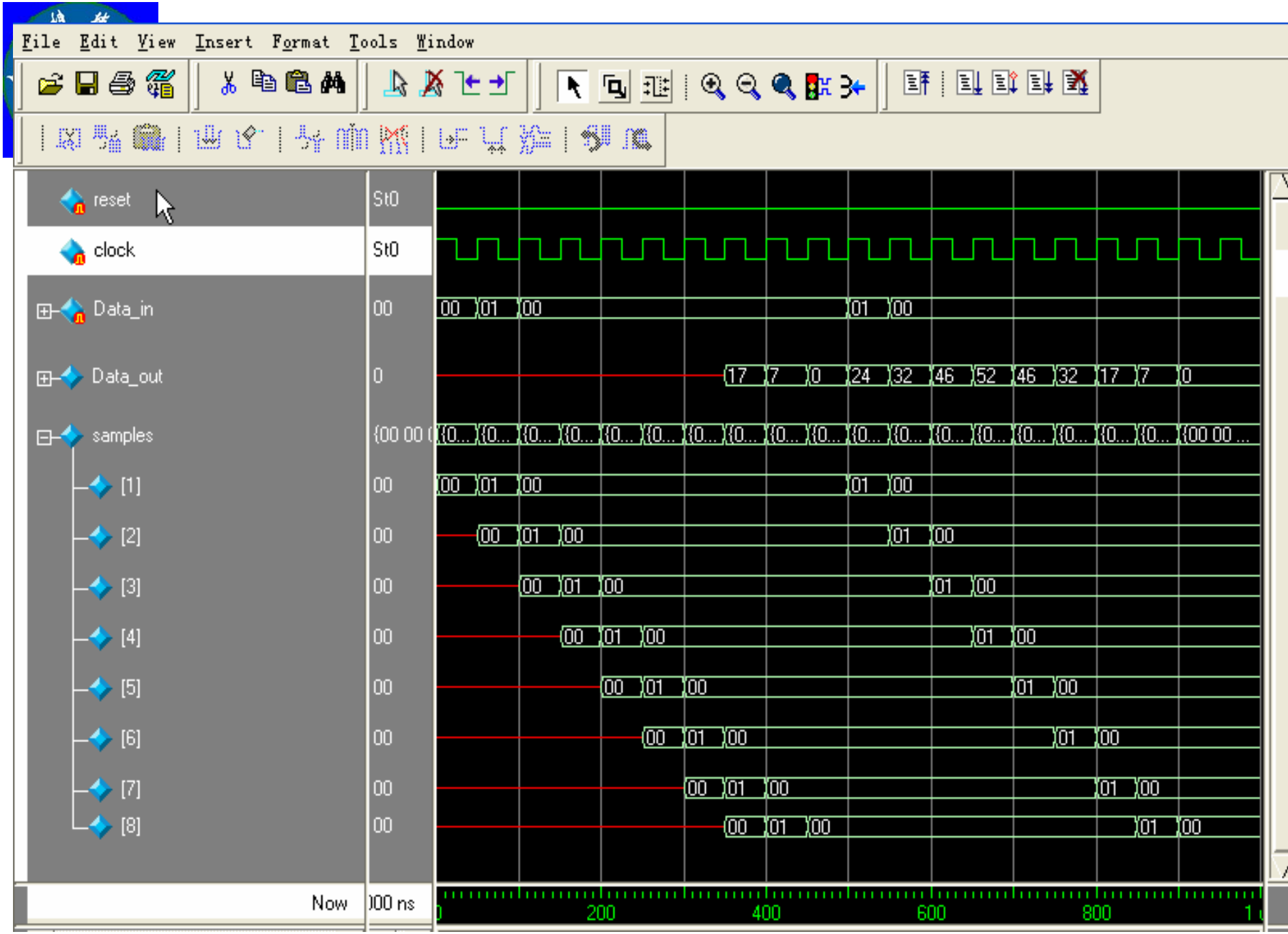
Ex 9.2 8-order Gaussian lowpass FIR filter

```
Module  FIR_Gaussian_Lowpass (Data_out,Data_in,clock,reset);
parameter order=8;
parameter word_size_in=8;
parameter word_size_out=2*word_size_in+2;
parameter b0=8'd7;
parameter b1=8'd17;
parameter b2=8'd32;
parameter b3=8'd46;
parameter b4=8'd52;
parameter b5=8'd46;
parameter b6=8'd32;
parameter b7=8'd17;
parameter b8=8'd7;
```

```
output [word_size_out-1:0] Data_out;
input [word_size_in-1:0]  Data_in;
input clock,reset;
reg [word_size_in-1:0] Samples[1:order];
integer k;
assign Data_out =  b0*Data_in
                          +b1*Samples[1]
                           +b2*Samples[2]
                           +b3*Samples[3]
                           +b4*Samples[4]
                           +b5*Samples[5]
                           +b6*Samples[6]
                           +b7*Samples[7]
                           +b8*Samples[8];
always@(posedge clock)
   if(reset==1) begin for(k=1;k<=order; k=k+1)Samples[k]<=0;end
    else begin
      Samples[1]<=Data_in;
      for(k=2;k<=order;k=k+1) Samples[k] <=Samples[k-1];
    end
endmodule
```

Fig 9-25(a) The impulse response of a FIR filter



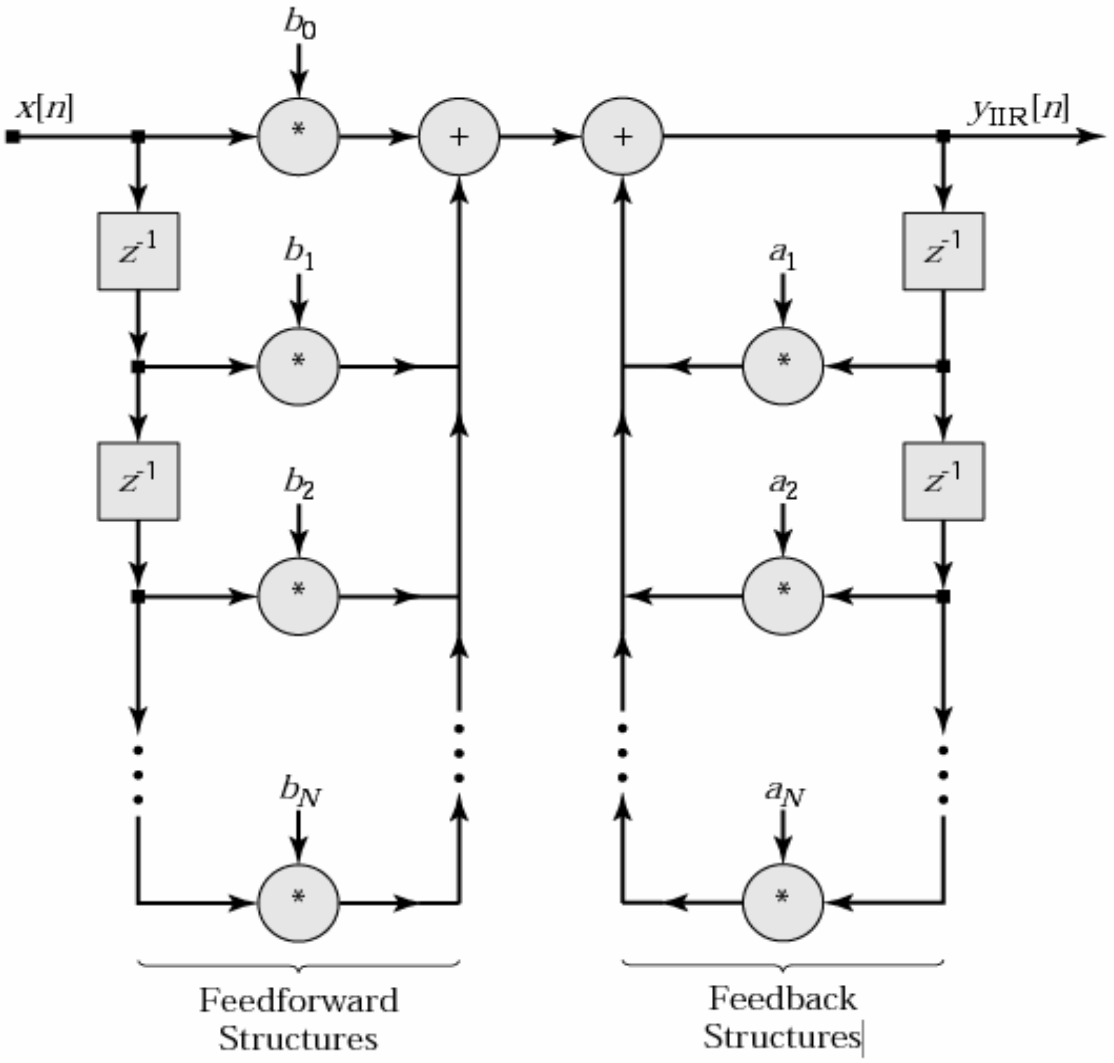

File Edit View Insert Format Tools Window
reset
clock
Data_in
Data_out
samples
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
St0
St0
00
0
(00 00 (
Now
)00 ns
200
400
600
800

9.3.3 Infinite-Duration Impulse Response (IIR) Filter

- IIR filter are the most general class of linear digital filters.
- Their output in a given time step depend on their inputs and on previously computed outputs.
- IIR filters are recursive

$$y_{IIR}[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$

Fig 9-26 The structure of Type-1 IIR



The IIR's response to an impulse may have infinite duration

- The filter is recursive because the difference equation has feedback
- Consequently, the filter's response to an impulse may have infinite duration **(i.e., it does not become 0 in a finite time).**
- An IIR filter is modeled in the z domain by its

  z-domain system function, or transfer function, which is a ratio of polynomials formed

The tap coefficients

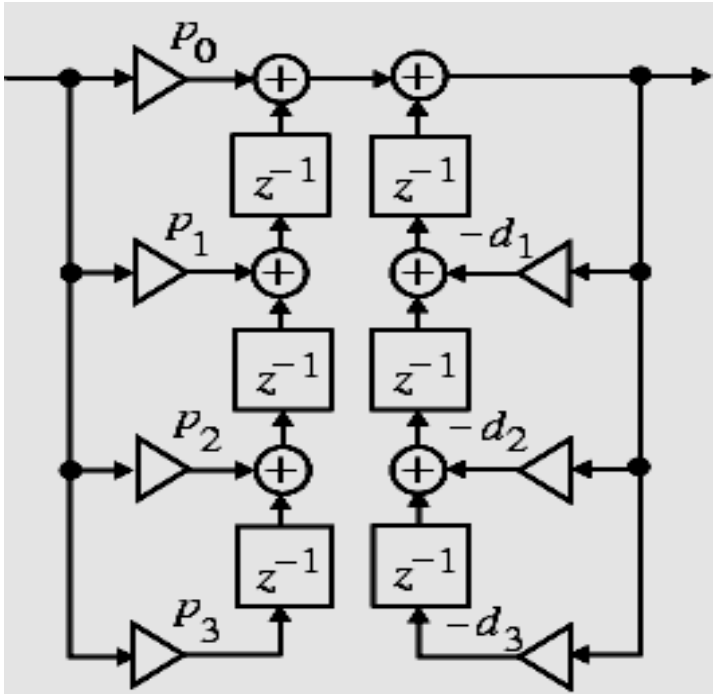
$$y_{IIR}[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$

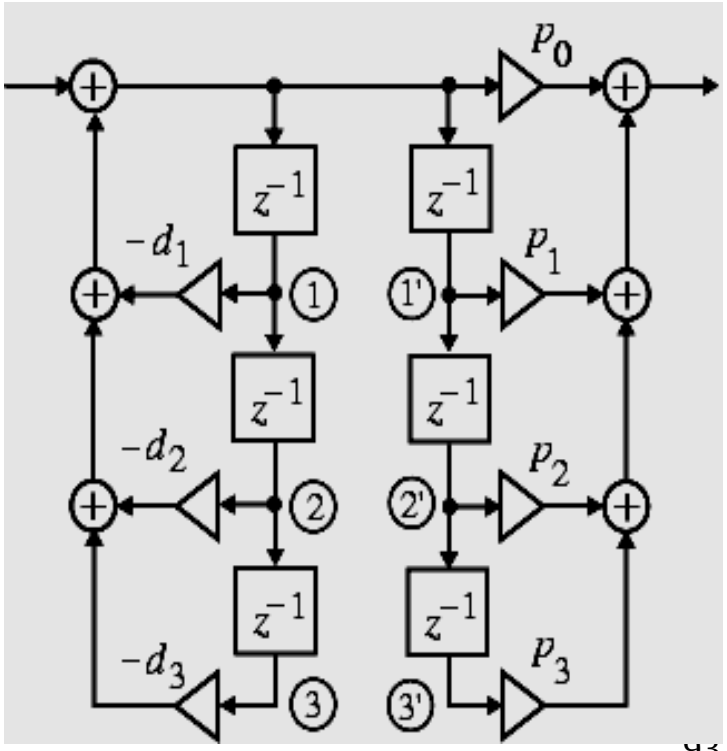
- The tap coefficients of the IIR filter form the sets of the filter's tap coefficients,

  {aj} and {bk} ,commonly referred to as the feedback and feedforward coefficients, respectively
- The parameter N is the order of the filter; it specifies the number of prior samples of the output that must be saved to form the current output; it also **determines the latency of the output**
- The value of the parameter M specifies how many prior samples of the input will be used to form the output.

Direct Form IIR Digital Filter Structures

- **Various other noncanonic direct form structures can be derived by simple block diagram manipulations as shown below**

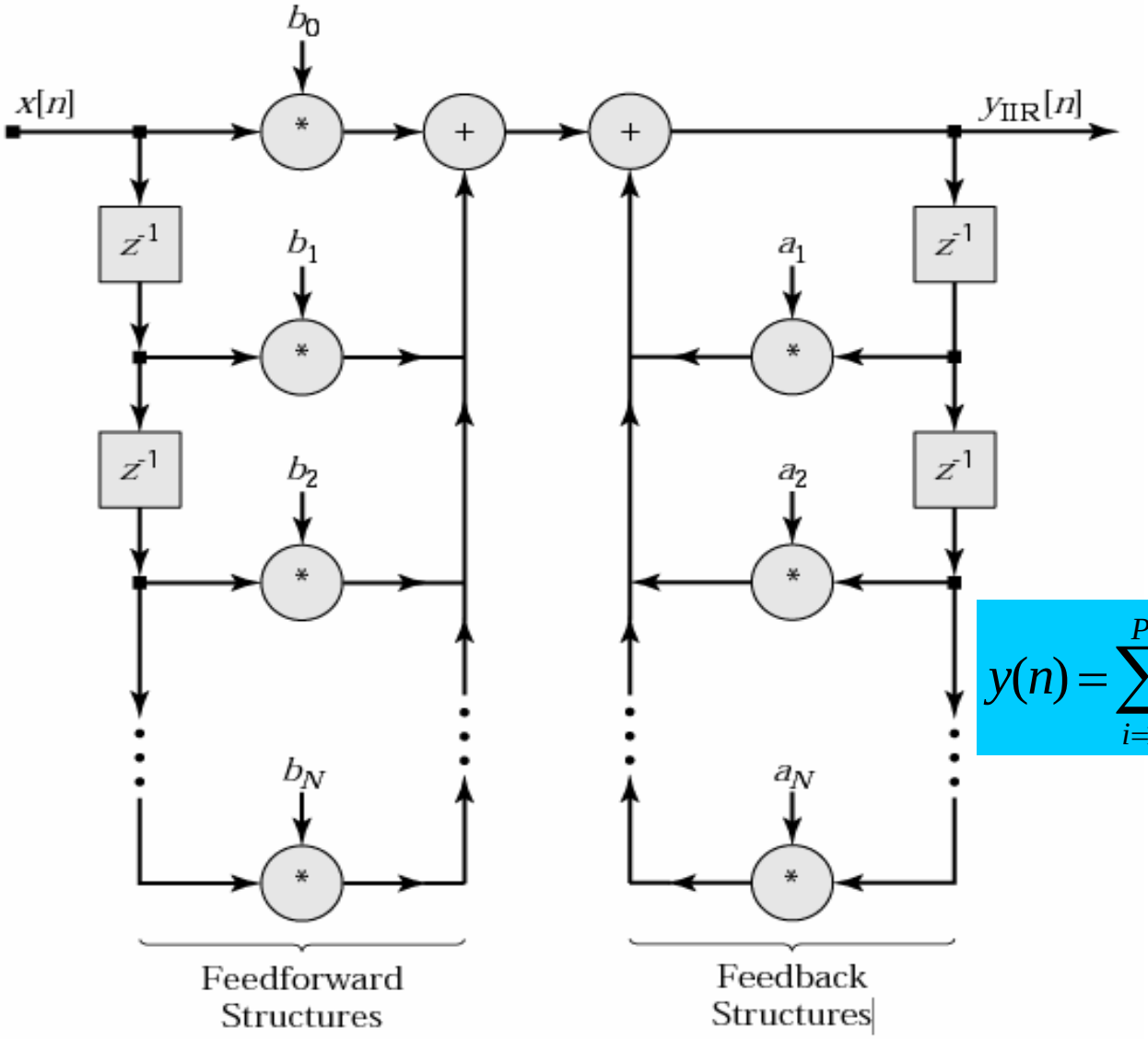




Example 9.3

- The Verilog model IIR_Filter_8 can be used to implement eighth-order IIR filters, depending on the selection of tap coefficients

The structure of Type-1 IIR



$$y(n)=\sum_{i=1}^{P}a(i)y(n-i)+\sum_{k=0}^{Q}b(k)x(n-k)$$

```
module IIR_Filter_8 (Data_out, Data_in, clock, reset);
    // Eighth-order, Generic IIR Filter
    parameter       order = 8;
    parameter       word_size_in = 8;
    parameter       word_size_out = 2*word_size_in + 2;
    parameter       b0 = 8'd7;
        // Feedforward filter coefficients
    parameter       b1 = 0;
    parameter       b2 = 0;
    parameter       b3 = 0;
    parameter       b4 = 0;
    parameter       b5 = 0;
    parameter       b6 = 0;
    parameter       b7 = 0;
    parameter       b8 = 0;
```

```
// Feedback filter coefficients
  parameter a1 = 8'd46;
  parameter        a2 = 8'd32;
  parameter        a3 = 8'd17;
  parameter        a4 = 8'd0;
  parameter        a5 = 8'd17;
  parameter        a6 = 8'd32;
  parameter        a7 = 8'd46;
  parameter        a8 = 8'd52;
  output     [word_size_out -1: 0]          Data_out;
  input      [word_size_in-1: 0]   Data_in;
  input      clock, reset;
  reg        [word_size_in-1: 0]   Samples_in [1: order];
  reg        [word_size_in-1: 0]   Samples_out [1: order];
  wire       [word_size_out -1: 0]          Data_feedforward;
  wire       [word_size_out -1: 0]          Data_feedback;
  integer    k;
```

```
assign Data_feedforward =        b0 * Data_in
                               + b1 * Samples_in[1]
                               + b2 * Samples_in[2]
                               + b3 * Samples_in[3]
                               + b4 * Samples_in[4]
                               + b5 * Samples_in[5]
                               + b6 * Samples_in[6]
                               + b7 * Samples_in[7]
                               + b8 * Samples_in[8];
assign Data_feedback =   a1 * Samples_out [1]
                               + a2 * Samples_out [2]
                               + a3 * Samples_out [3]
                               + a4 * Samples_out [4]
                               + a5 * Samples_out [5]
                               + a6 * Samples_out [6]
                               + a7 * Samples_out [7]
                               + a8 * Samples_out [8];
```

```
ssign Data_out = Data_feedforward +Data_feedback;
  always @ (posedge clock)
    if (reset == 1)
       for (k = 1; k <= order; k = k+1) begin
        Samples_in [k] <= 0;
        Samples_out [k] <= 0;
      end
    else begin
        Samples_in [1] <= Data_in;
        Samples_out [1] <= Data_out;
      for (k = 2; k <= order; k = k+1) begin
        Samples_in [k] <= Samples_in [k-1];
        Samples_out [k] <= Samples_out [k-1];
      end
    end
endmodule
```

9.4 Building Blocks for Signal Processors

- In this section we will consider models of basic operations of
- **Integration**
- **differentiation**
- **decimation**
- **interpolation**
  - which are common to many digital processors.

9.4.1 Integrators (Accumulators)

- Digital integrators are used in a popular type of analog-to-digital converter, called a **sigma-delta modulator**
- Digital integrators **accumulate a running sum of sample values**
- Two implementations are common: parallel and sequential

Example 9.4

- The model *Integrator_Par* below describes an integrator for a parallel datapath
- At each clock cycle the machine adds *data_in* to the content of the register *data_out*
- The signal *hold* pauses the accumulation of samples until it is de-asserted.

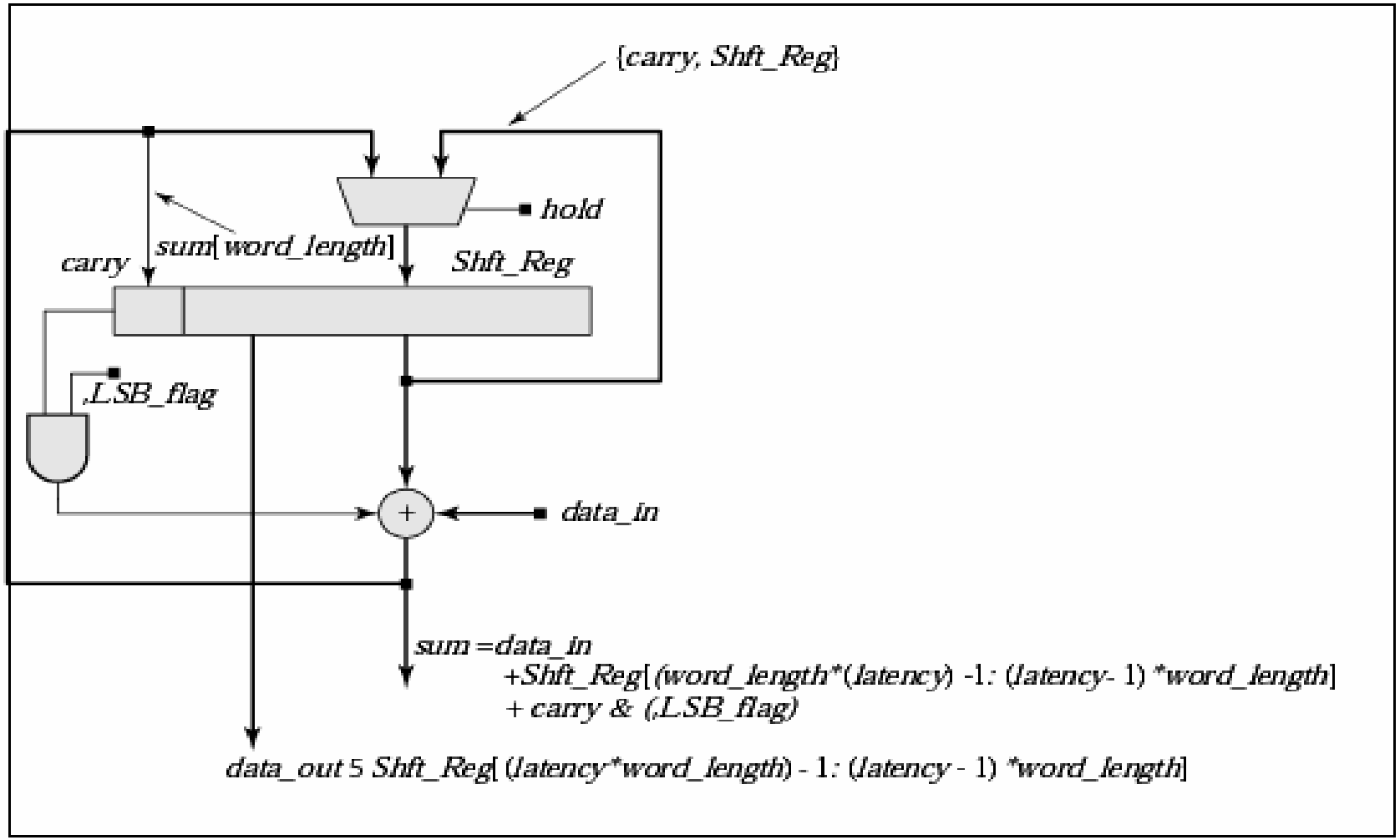
Exe 9.4 The model below describes an integrator for a parallel datapath

```
module
   integrator_Par(data_out,data_in,hold,clock,reset);
parameter word_length=8;
output      [word_length-1:0]     data_out;
input       [word_length-1:0]     data_in;
input                             hold,clock,reset;
reg                               data_out;

 always@(posedge clock) begin
  if (reset) data_out<=0;
   else if (hold) data_out<=data_out;
          else data_out<=data_out+data_in;
 end
endmodule
```

Ex 9.5 A byte-sequential integrator

- In this example, the unit is to accumulate 32-bit words, but receives data sequentially, in 8-bit bytes
- The signal hold pauses the accumulation of samples until it is de-asserted
- This architecture performs byte-wide addition, with the current data sample being added to the leftmost byte of the shift register Shft_Reg, to form sum
- It is common for a processor **to receive data via a narrower datapath than the datapath within the processor**

Figure 9-28 architecture of a byte – sequential integrator



Example 9.5

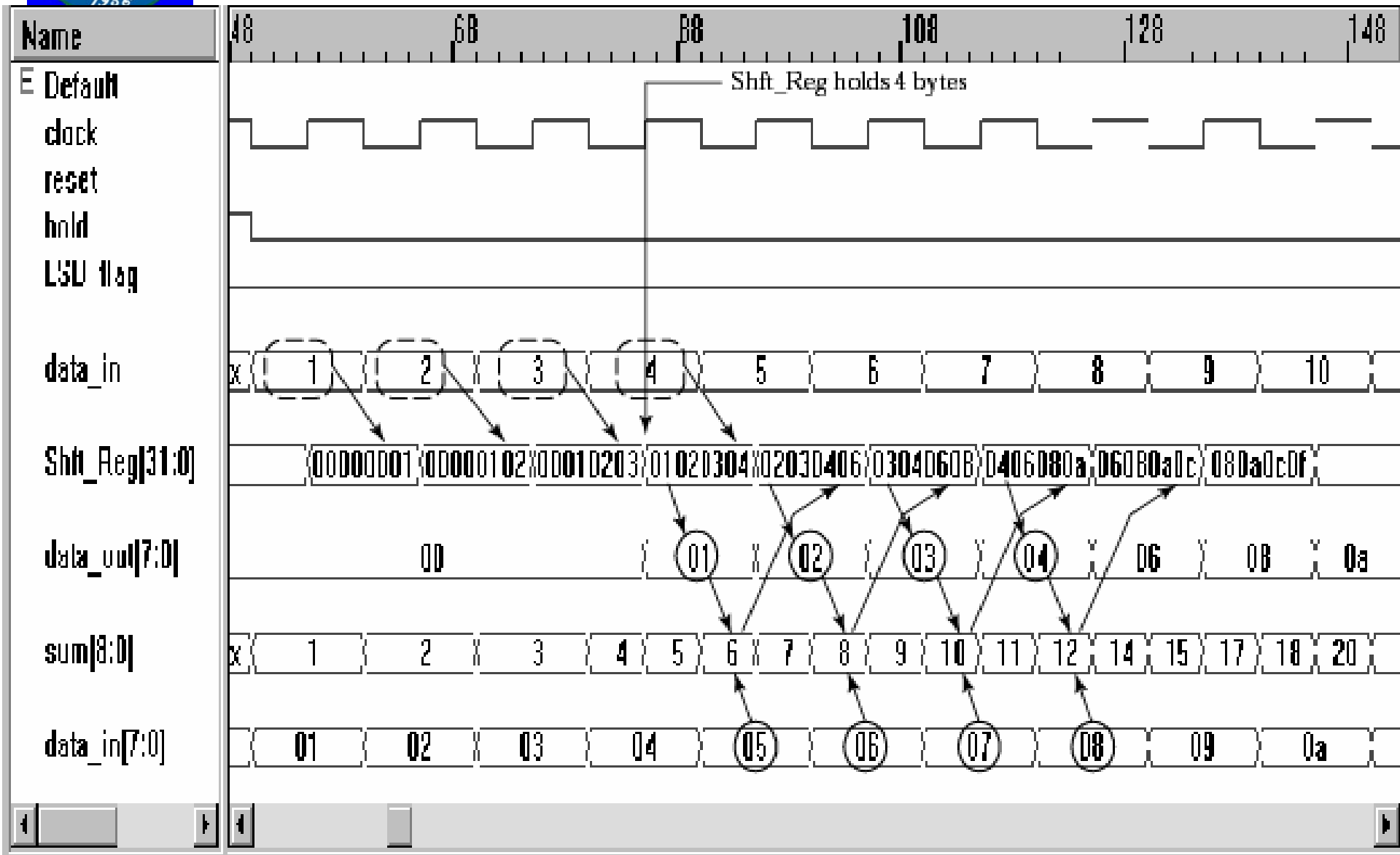
```
module Integrator_Seq (data_out, data_in, hold, LSB_flag,
    clock, reset);
    parameter word_length = 8;
    parameter latency = 4;
    output      [word_length -1: 0]   data_out;
    input       [word_length -1: 0]   data_in;
    input       hold,   LSB_flag, clock, reset;
   reg [(word_length * latency) -1: 0]  Shft_Reg; //32 bits
    reg         carry;
   wire [word_length: 0]                sum; // 9 bits
```

```
always @ (posedge clock) begin
      if (reset) begin Shft_Reg <= 0; carry <= 0; end
        else if (hold) begin
          Shft_Reg <= Shft_Reg;
          carry <= carry;
        end
       else begin
 // shifted a byte toward its MSByte,sum load into the left byte
        Shft_Reg <= {Shft_Reg[word_length*(latency -1) -1: 0],
                                                sum[word_length-1: 0]};
        end
 end
 assign sum = data_in + Shft_Reg [ (latency * word_length) -1:
      (latency -1)*word_length ] + (carry & (~LSB_flag));
 // add data_in  to the leftmst byte of Shft_Reg, to form sum
 assign data_out = Shft_Reg[(latency * word_length) -1:
      (latency -1)*word_length];
 // output  the leftmst byte of Shft_Reg
endmodule
```

Fig 9-29

t	Shft_Reg			
	Byte_1	Byte_2	Byte_3	Byte_4
	Byte_1 + Byte_5	Byte_2 +Byte_6	Byte_3 + Byte_7	Byte_4 + Byte_8
	Byte_1 + Byte_5 + Byte_9	Byte_2 + Byte_6 + Byte_10	Byte_3 +Byte_7 +Byte_11	Byte_4 +Byte_8 + Byte_12
	Byte_1 + Byte_5 + Byte_9 +ByteE_13	Byte_2 + Byte_6+ Byte_10+ Byte_14	Byte_3 +Byte_7 1 Byte_11 + Byte_15	Byte_4 1 Byte_8 + Byte_12 + Byte_16

Fig 9-30 The simulation of the byte-sequential integrator



9.4.2 Differentiators

- A differentiator provides a measure of the sample-to-sample change in a signal
- The backward difference is implemented with a buffer and a subtractor

Differentiator

```
module differentiator(data_out,data_in,hold,clock,reset)
parameter  [word_size-1:0]    data_out;
parameter  [word_size-1:0]    data_in;
input                         hold;
input                         clock,reset;
reg          [word_size-1:0]    buffer;
wire         [word_size-1:0]   data_out=data_in - buffer;
always  @  ( posedge clock) begin
if (reset) buffer<=0;
  else if (hold)  buffer<=buffer;
    else  buffer<=data_in;
end
endmodule
```

9.4.3 Decimation and Interpolation Filters

- Decimation and interpolation filters are used **to achieve sample rate conversion** in digital signal processors
- Decimation filters decrease the sample rate
- Interpolation filters increase the sample rate

Decimation and Interpolation Filters

- Interpolation filters **enable a signal to be oversampled**, thereby **reducing the effects of aliasing**
- If a signal is not sampled properly, it cannot be recovered with fidelity
- Decimation is used **to reduce the bandwidth of a signal that has been oversampled**
- Decimation achieves sample rate reduction

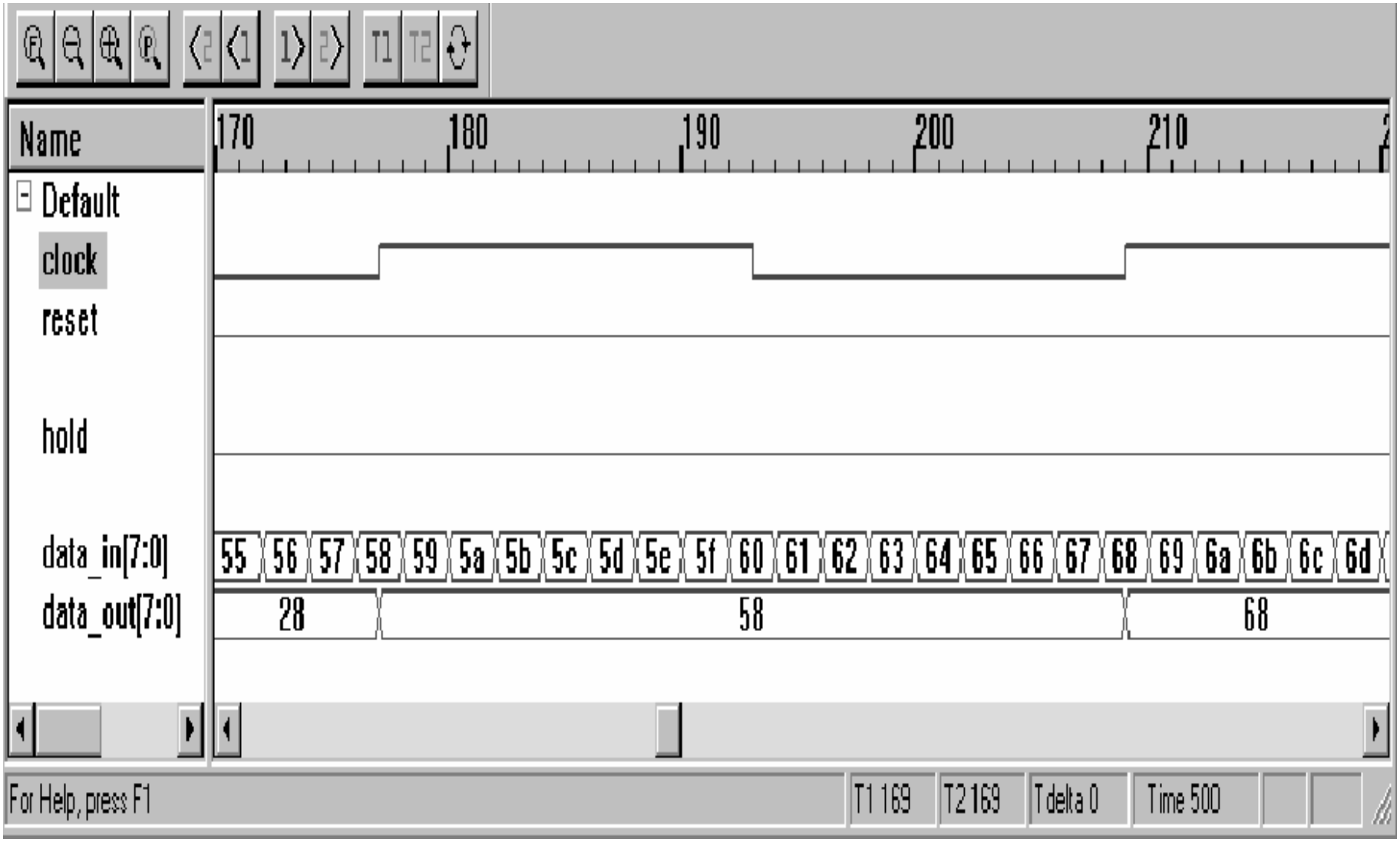
Example 9.6

- The Verilog model decimator_1 describes the behavior of a **parallel-in-parallel-out decimator**, which samples its input at a rate determined by clock unless hold is asserted
- Note that samples of data_in in Figure 9-31 are dropped because clock is running at a rate that is slower than the rate at which data_in had been sampled

Example 9.6

```
module
    decimator_1(data_out,data_in,hold,clock,reset);
parameter word_length=8;
output [word_length-1:0]        data_out;
input    [word_length-1:0]       data_in;
input              hold;
input              clock;
reg                data_out;
always@(posedge clock)
begin
if(reset) data_out<=0;
  else if(hold) data_out<=data_out;
    else data_out<=data_in;
end
endmodule
```

Fig 9-31


Name
170
180
190
200
210
Default
clock
reset
hold
data_in[7:0]
55 56 57 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d
data_out[7:0]
28
58
68
For Help, press F1
T1 169
T2 169
Tdelta 0
Time 500

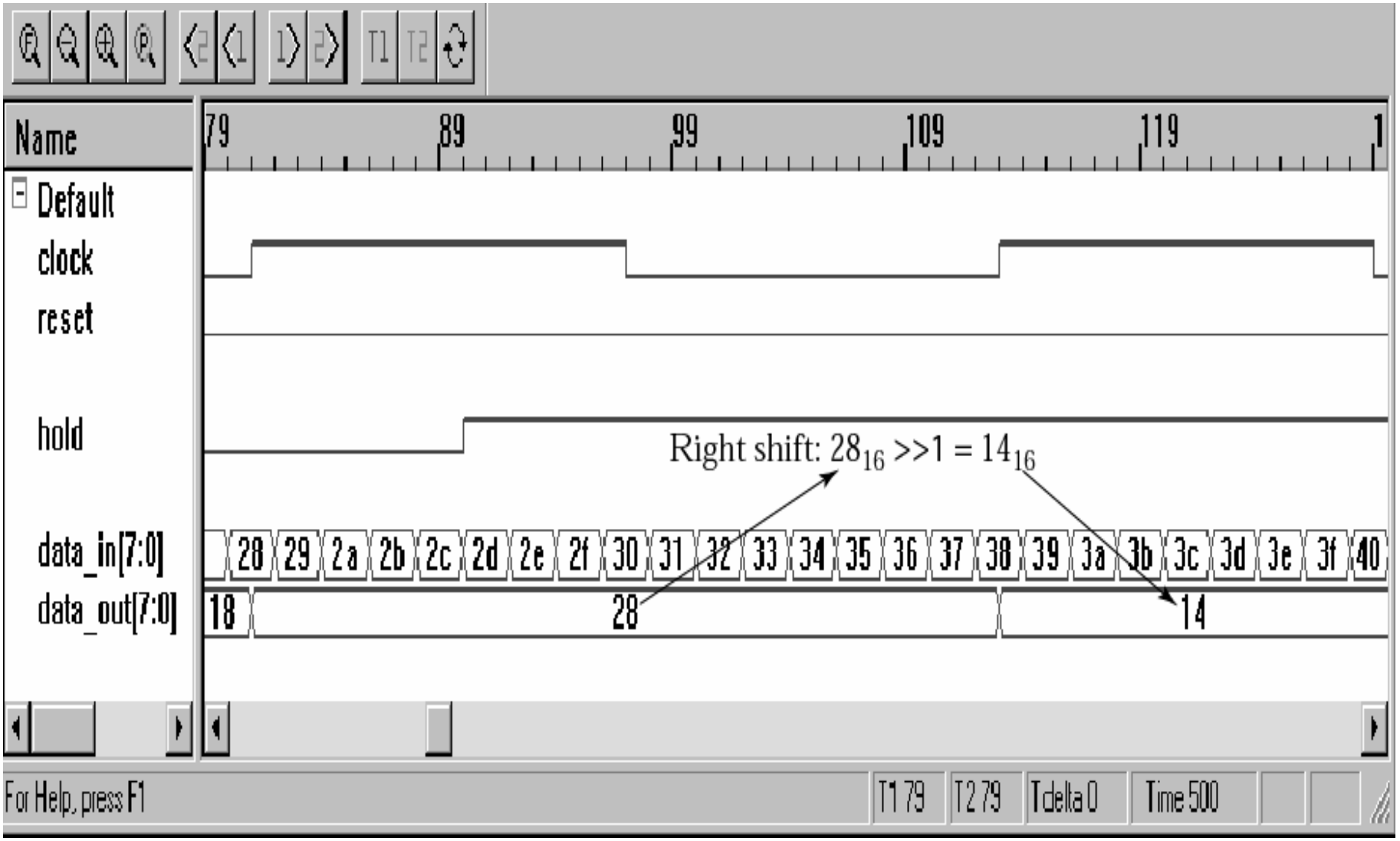
Example 9.7

- The Verilog model decimator_2 **samples a parallel input and produces a parallel output**, but includes an option to form a serial output by shifting the output word through the LSB while hold is asserted
- This action is apparent in the waveforms shown in Figure 9-32

Example 9.7

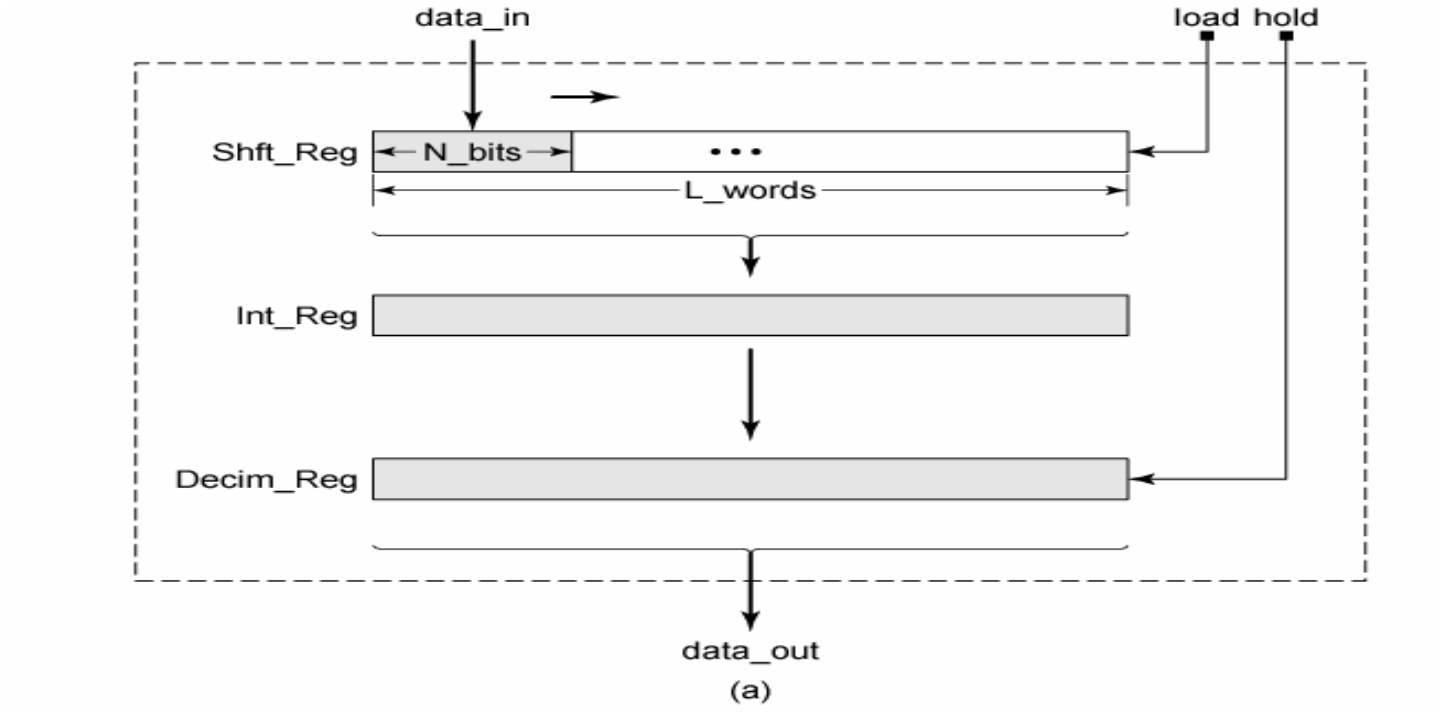
```
module
      decimator_2(data_out,data_in,hold,clock,reset);
parameter     word_length=8;
output    [word_length-1:0]   data_out;
input     [word_length-1:0]   data_in;
input                         hold;
input                         clock;
input                         reset;
reg                           data_out;
always@(posedge clock)
   if (reset)  data_out<=0;
     else if (hold)  data_out<=data_out>>1;
         else data_out<=data_in;
endmodule
```

Fig 9-32

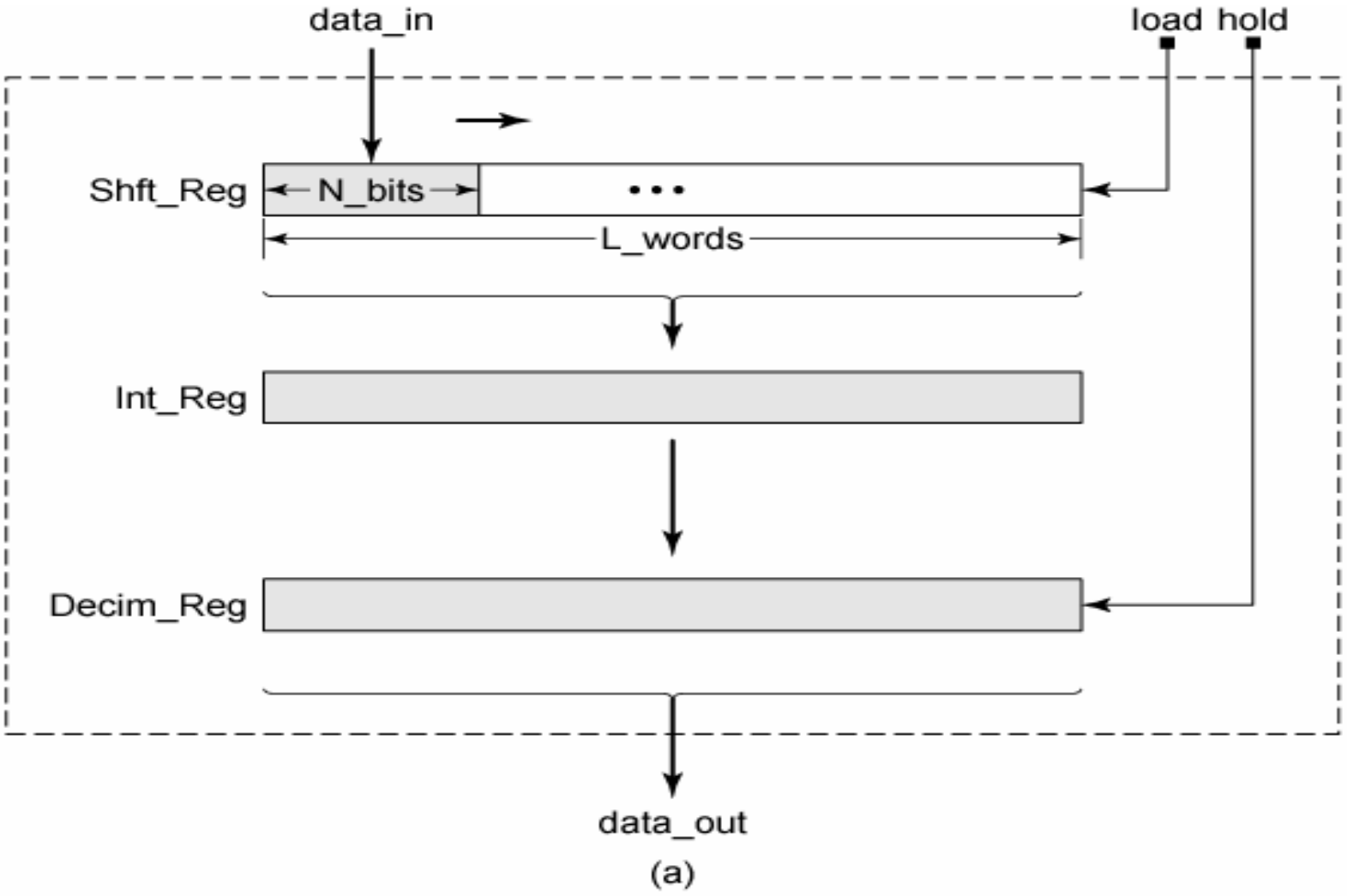

Name
79
89
99
109
119
Default
clock
reset
hold
Right shift: 28_{16} >>1 = 14_{16}
data_in[7:0]
28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f 40
data_out[7:0]
18
28
14
For Help, press F1
T1 79
T2 79
Tdelta 0
Time 500

Ex 9.8 The decimator with a sequential integrator

- The decimator's architecture consists of three registers, *Shft_Reg, Int_Reg, and Decim_Reg*
- All three are sized **to hold multiple bytes (samples),** as determined by a parameter *latency*



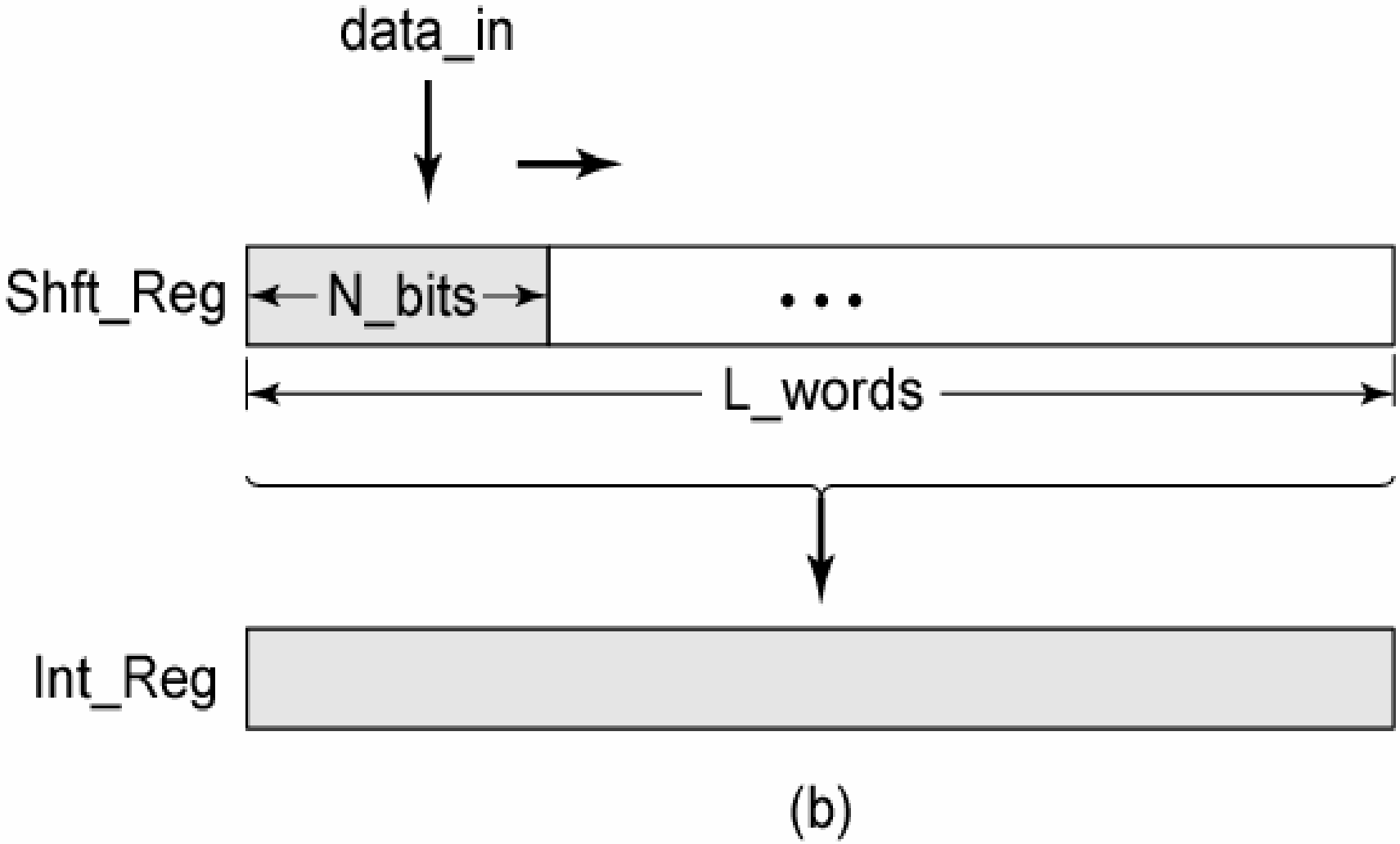
Ex 9-8(a)
Overall architecture



When Shft_Reg is full

- The two register transfers occur concurrently:
- (1) the contents of Shft_Reg are loaded into an intermediate holding register, Int_Reg, and
- (2) the LSByte of a new word is loaded into the MSByte of Shft_Reg and transfers load Shft_Reg until it is full

Ex 9-8(b) Concurrent registers transfer while loading



Ex 9-8(c) shifting contents & loading

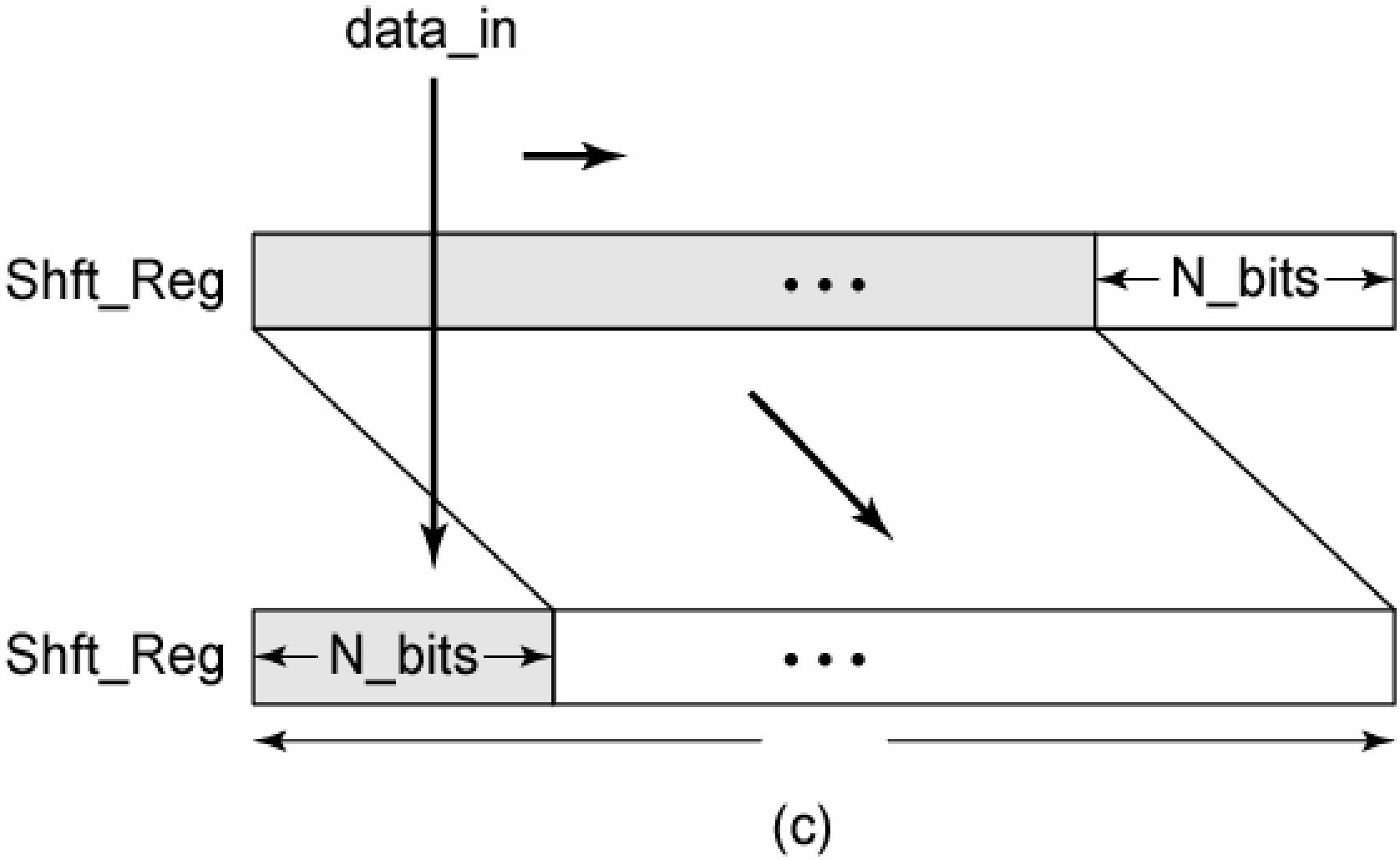
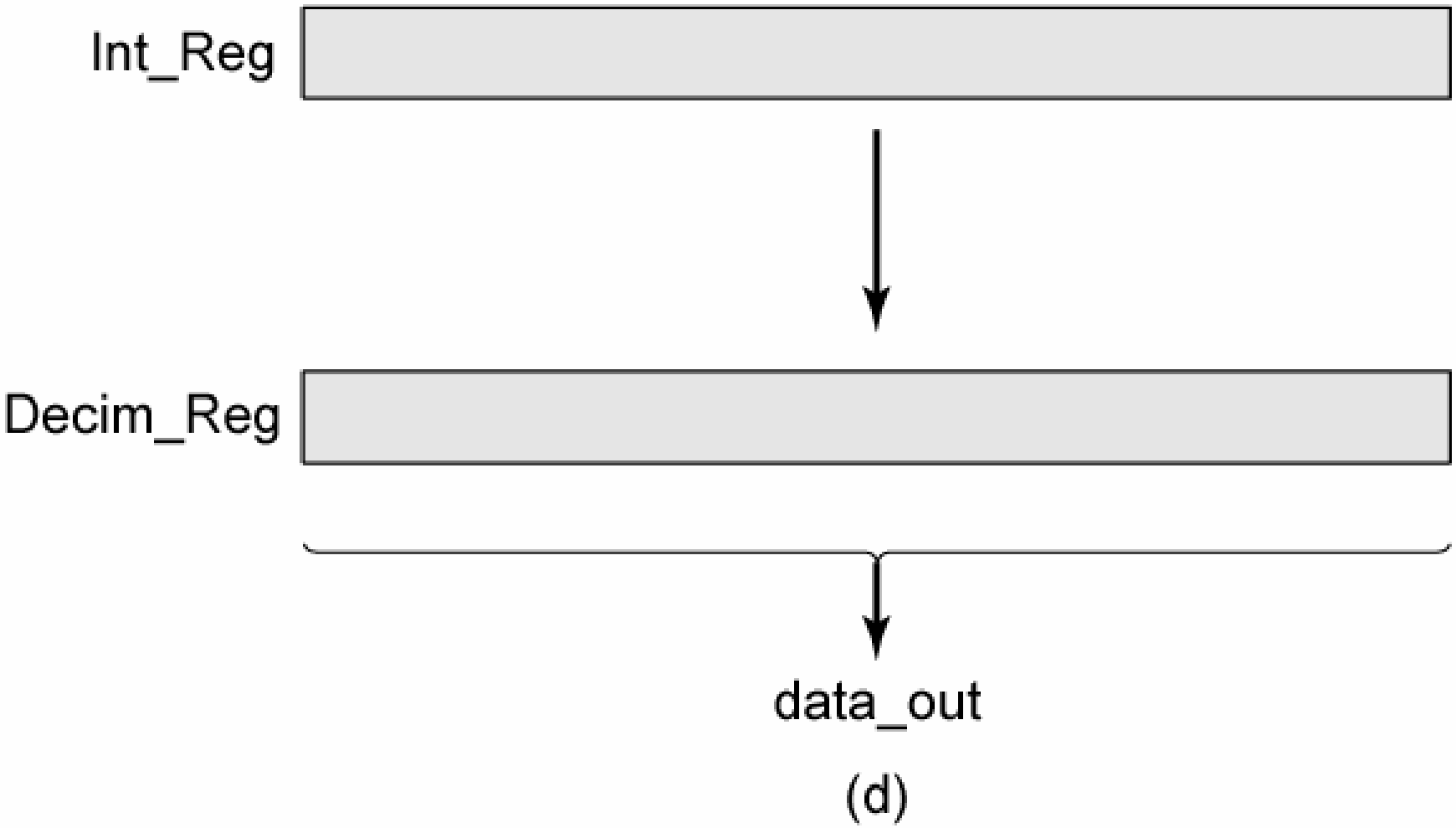


Fig 9-8(d) loading contents from Int_Reg to Decin_Reg

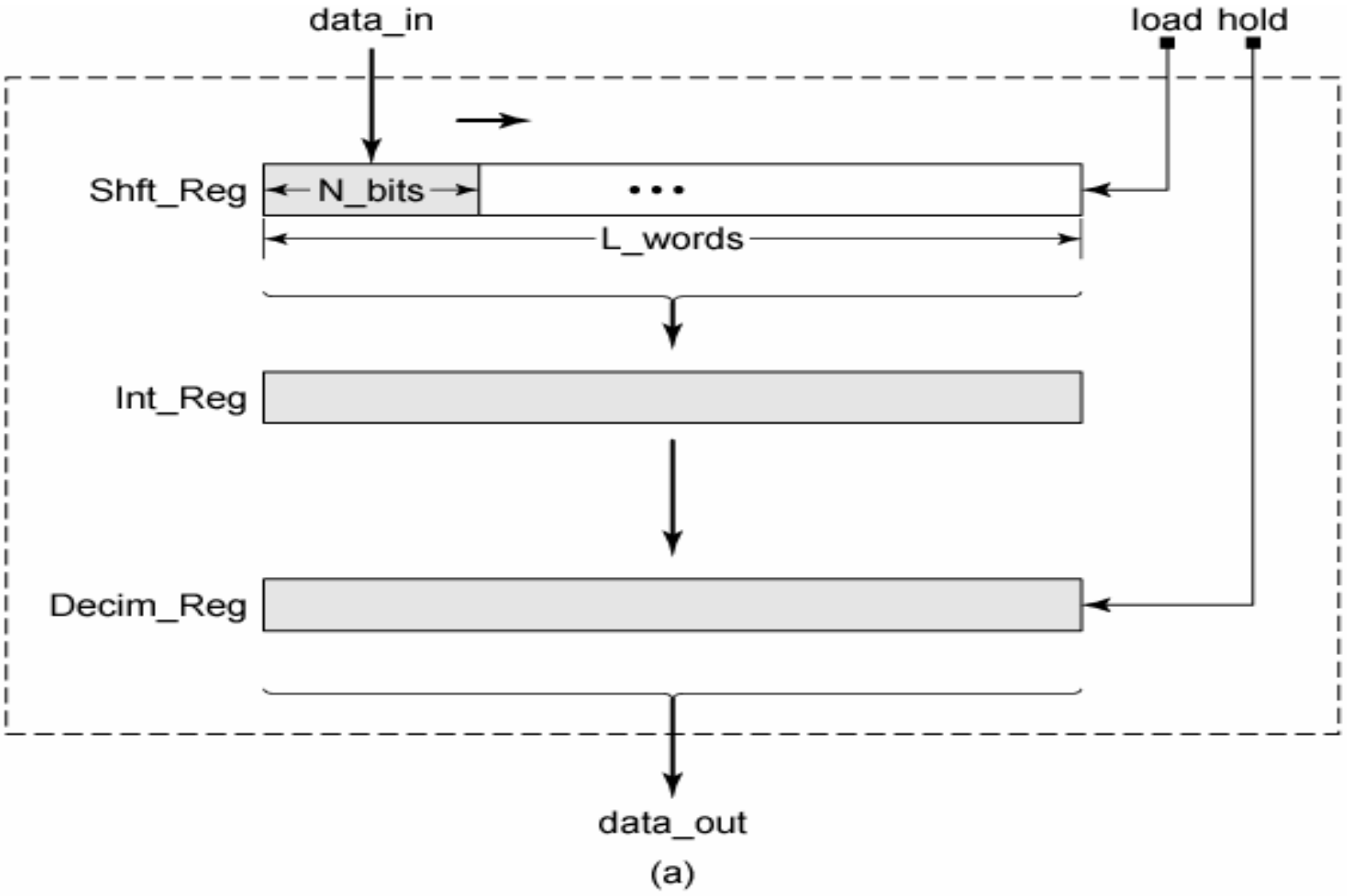


Ex 9.8

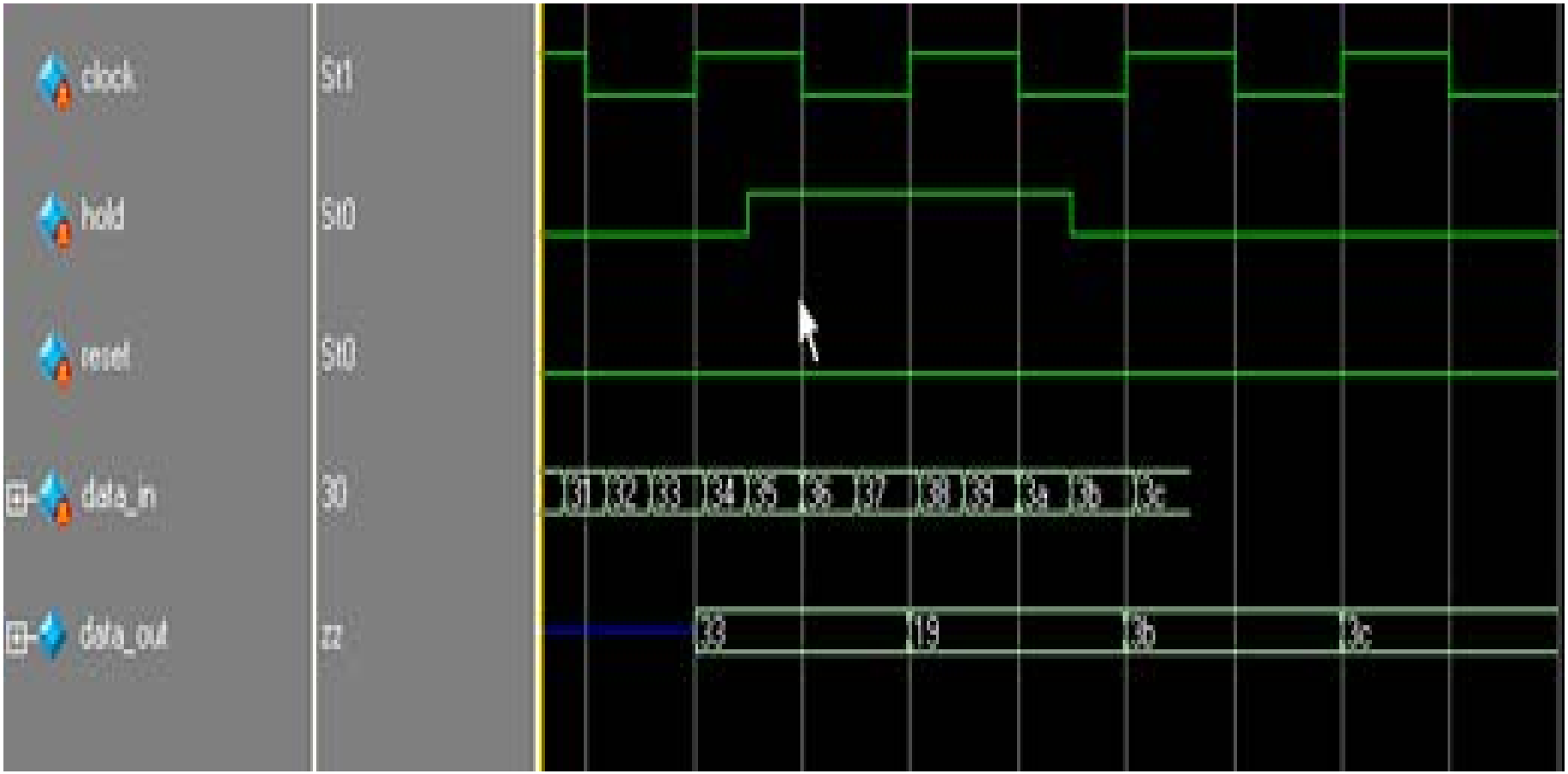
```
module decimator_3 (data_out, data_in,
                    hold, load, clock, reset);
parameter     word_length = 8;
parameter     latency = 4;
output [(word_length*latency) -1: 0] data_out; //32bits
input   [word_length-1: 0] data_in; // 8 bits
input   hold;
input   load;
input   clock;
input   reset;
reg     [(word_length*latency) -1: 0] Shft_Reg;
        // Shift reg
reg     [(word_length*latency) -1: 0] Int_Reg;
          // Intermediate reg
reg     [(word_length*latency) -1: 0] Decim_Reg;
        // Decimation reg
```

```
always @ (posedge clock)
  if (reset) begin
    Shft_Reg <= 0;
     Int_Reg <= 0;
   end
   else if (load) begin
     Shft_Reg[(word_length * latency) -1:
           (word_length*(latency- 1))]  <=  data_in; // BIT(31-24)=data_in
      Int_Reg <= Shft_Reg;
    end
     else begin // right shift 8 bits
       Shft_Reg <=
          {data_in, Shft_Reg[(word_length*latency) -1: word_length]};
          Int_Reg <= Int_Reg;
     end
 always @ (posedge clock)
          if (reset) Decim_Reg <= 0;
          else if (hold) Decim_Reg <= Decim_Reg;
          else Decim_Reg <= Int_Reg;
 assign data_out = Decim_Reg;
endmodule
```

Ex 9-8(a)
Overall architecture



Simulation



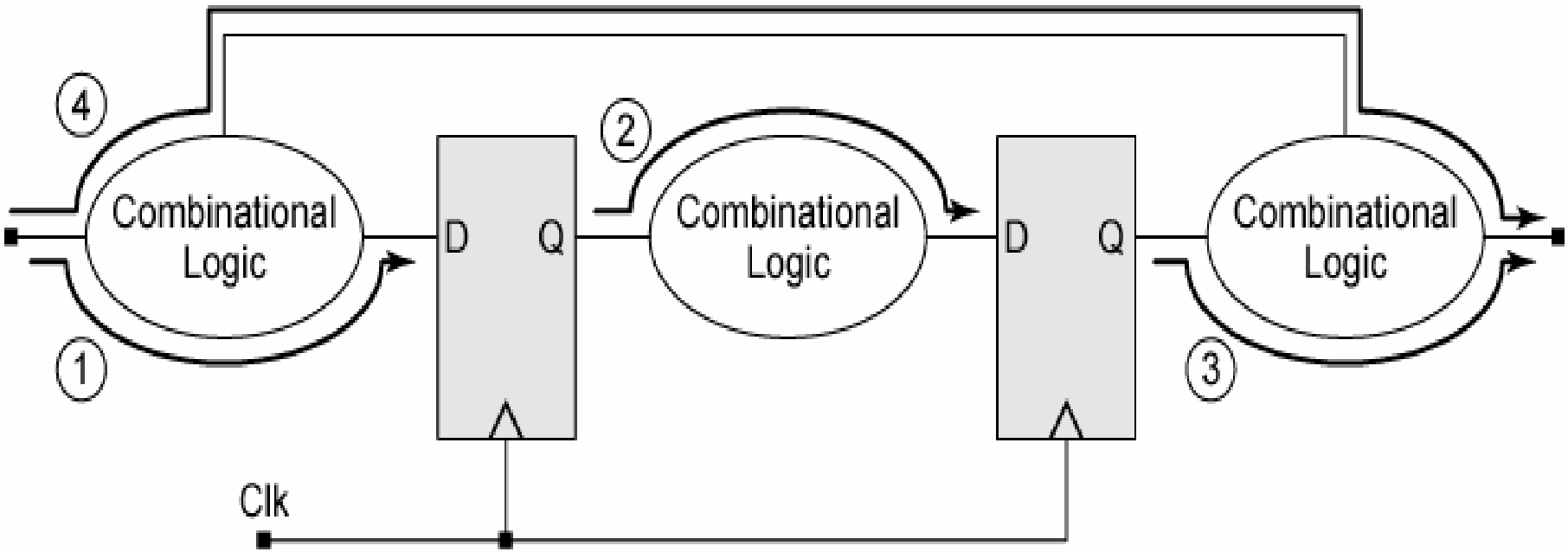
9.5 Pipelined Architectures

- **The shortest cycle** time of the clock of a synchronous sequential machine is a measure of its performance, and it is bounded by the propagation delay through the combinational logic of the machine
- **The throughput of a synchronous machine** is the rate at which data is supplied to and produced by the machine

Throughput ‘s limits

- Throughput is ultimately **limited by the path with the largest propagation delay** between
- (1) a primary input and a register
- (2) a path between a pair of registers
- (3) a path from a register to a primary output
  (4) a path from a primary input to a primary output
- **In each case, combinational logic limits the performance of the machine**

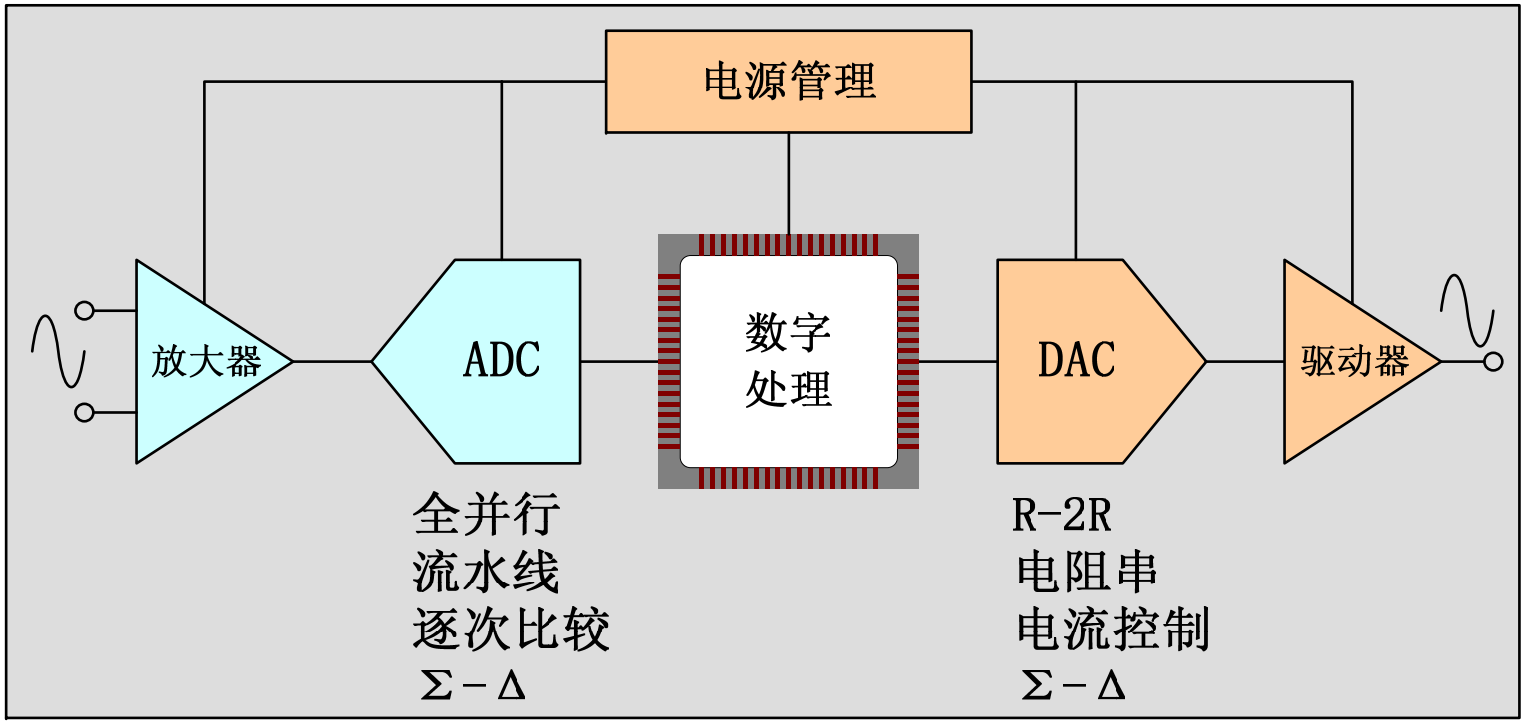
A circuit may have four types of timing paths



1. Input port to register data terminal
2. Register to register
3. Register to output port
4. Input port to output port

DSP Solution：

The basic components of a DSP system (add,multiply,delay)



$$y[n] = \sum_{k=-\infty}^{\infty} h[k]\,x[n-k]$$

Approach to gaining performance

- Three general approaches
  - **Recomposition**
  - **Replication**
  - **pipelining**

Recomposition

- Recomposition segments the FU into a sequence of functions that execute one after the other to implement the algorithm.
- The sequence of execution may be further **distributed over space (hardware units) and time**
- **In Space** , the DFG are mapped isomorphically to the FUs;
- **In Time,** a single FU executes over as many clock cycles as required to complete the operations represented by the DFG.

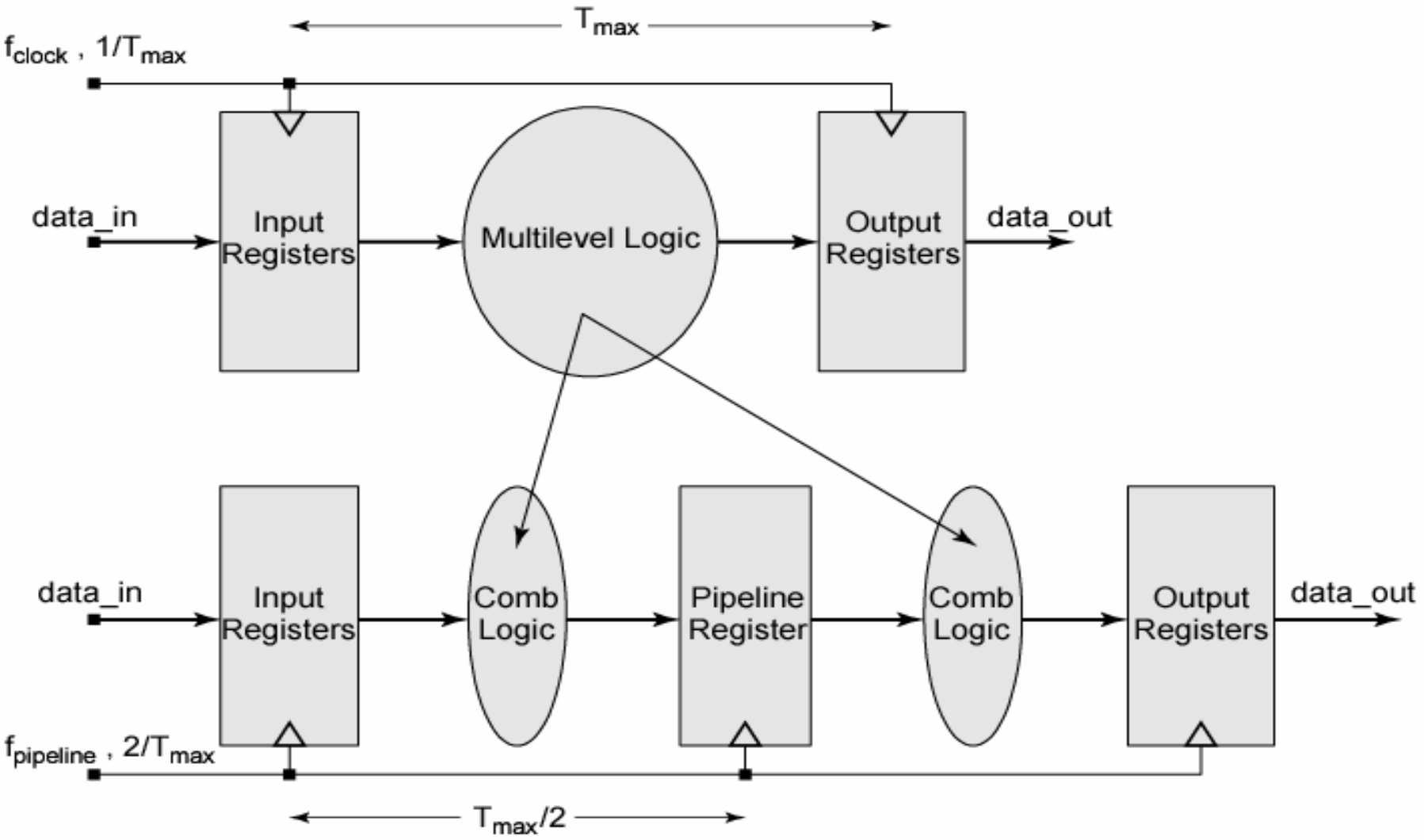
Replication

- In contrast to recomposition
- Replication uses multiple, identical, concurrently executing processors to improve performance
- But **at the expense of hardware**

Pipelining

- Pipelining becomes increasingly important in high-speed, wide-word data transmission and processing
- Pipeline registers can be inserted into the combinational logic datapaths **at strategic locations** to partition the logic into groups with shorter paths

Fig 9-34 Data Pipeline



The caution:

- **The partition of a datapath must maintain coherency of the data's datapath** traced from any primary input to any primary output **must pass through the same number of pipeline registers**
- Pipelining:
  - reduces the number of levels in the blocks of combinational logic,
  - shortens the datapaths between storage elements, and
  - increases the throughput of the circuit, by allowing the clock to run faster

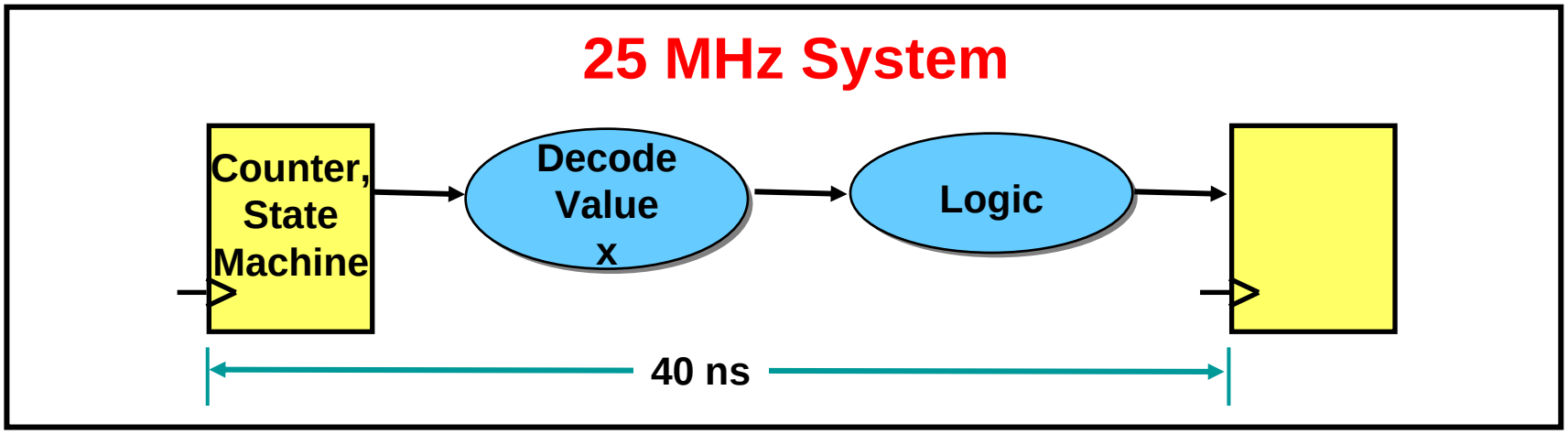
Pipelining's cost

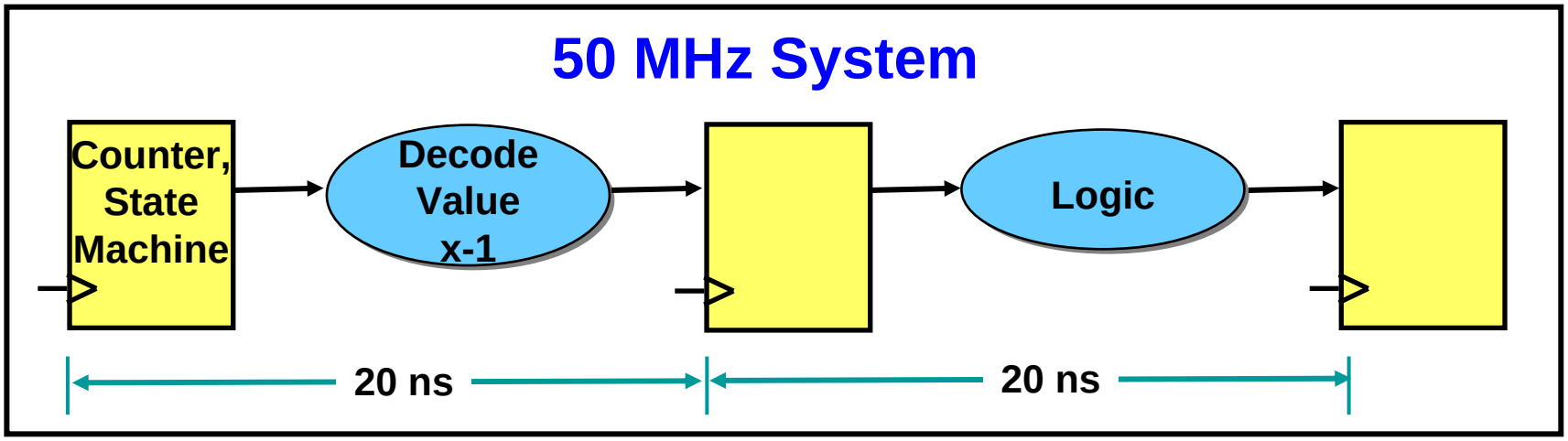
- The pipeline registers **introduce additional area** in the physical layout of an ASIC, and **require additional routing of clock resources**
- This could be an issue for an ASIC
- But high-end FPGAs are register-rich, and they readily support pipelined architectures

Partitioning a circuit to create a pipeline

- The delay of the slowest combinational logic stage determines the performance of the pipelined circuit and the speed at which the circuit's common clock can run
- Pipelining shortens the clock cycle and increases the throughput, but introduces input-output latency

Adding Single Pipeline Stage

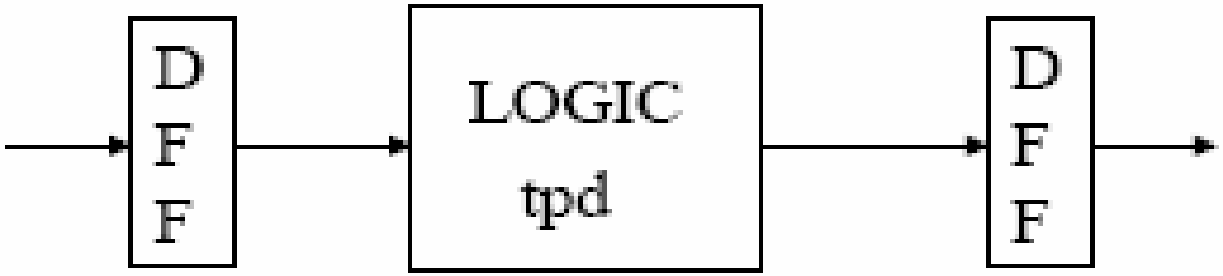




The latency accumulation

- The latency accumulates through the pipeline
- Latency effectively introduces a time shift between a circuit's input transitions and its output transitions
- **Pipelining trades spatial (hardware) complexity for temporal complexity (performance) by computing smaller functions in less time**

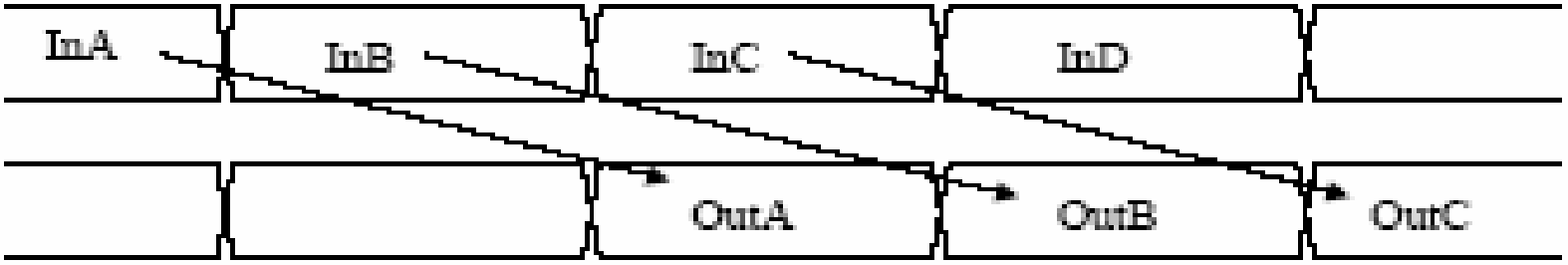
Registered Datapath



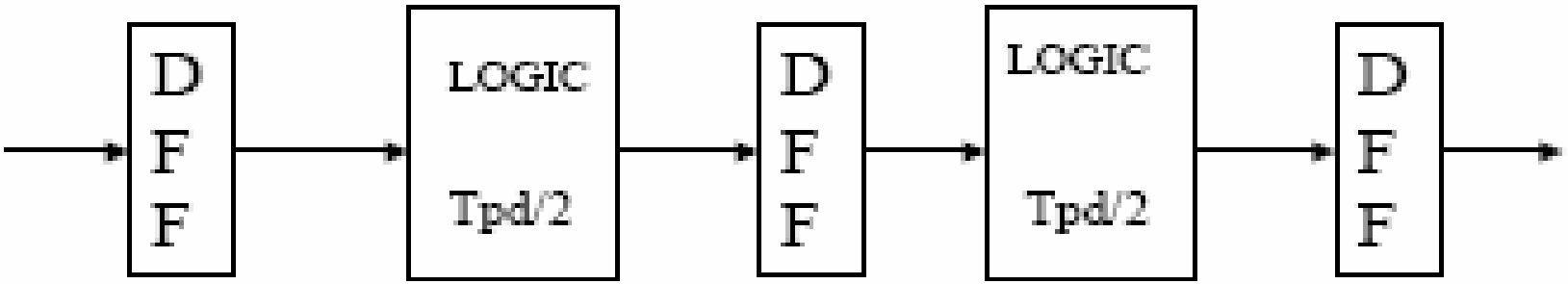
DFFs are rising edge triggered

Clk Freq = 1/ (Tclk2q + Tpd + Tsu)

Latency = 1 clk

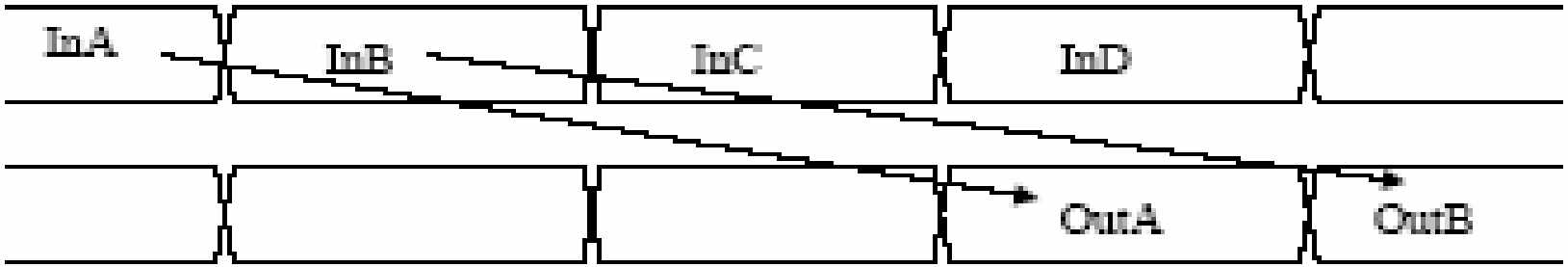


Add a pipeline stage



Clk Freq = 1/ (Tclk2q + Tpd/2 + Tsu)

Latency = 2 clks



Three major benefits

1. Dedicated hardware performs the same single task in every clock cycle
2. The logic to perform a single , delicated task can be streamlined and optimized as a unit.
3. The datapaths between adjacent stages of the pipeline are short and direct

The design of a circuit with pipelined datapaths must address the following issues:

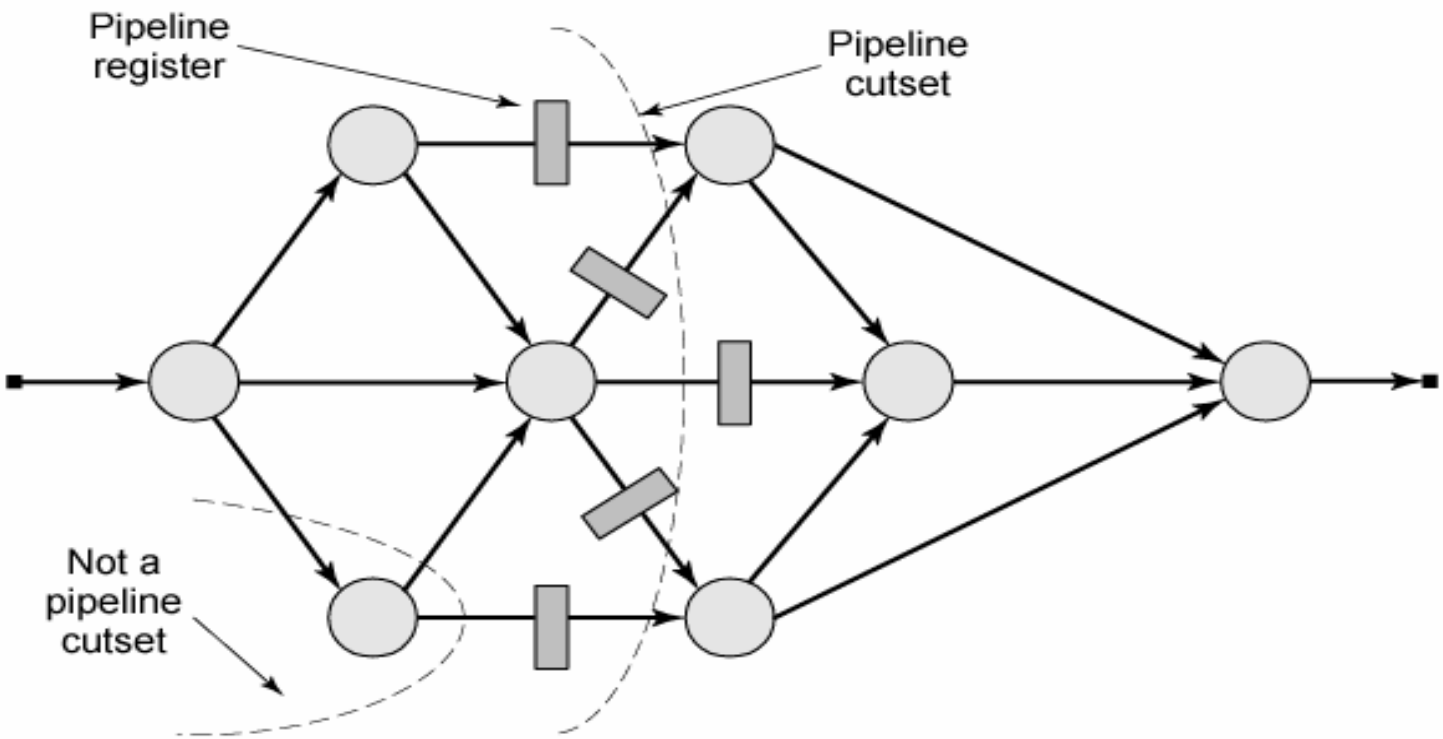
- (1) When should pipelining be considered?
- (2) Where should the pipeline registers be inserted?
- (3) How much latency will be introduced by the pipeline?

The feedforward cutsets of the DFG

- **A cutset of a connected graph is a set of branches that isolates a node of the graph if removed from the graph**
- A pipeline cutset or feedforward cutset is **a minimum set of edges** that, if removed from the graph, partitions it into two connected subgraphs such that there is no path between an input node and an output node
- **Cutsets are used to determine alternative placements of pipeline registers**

Cutset

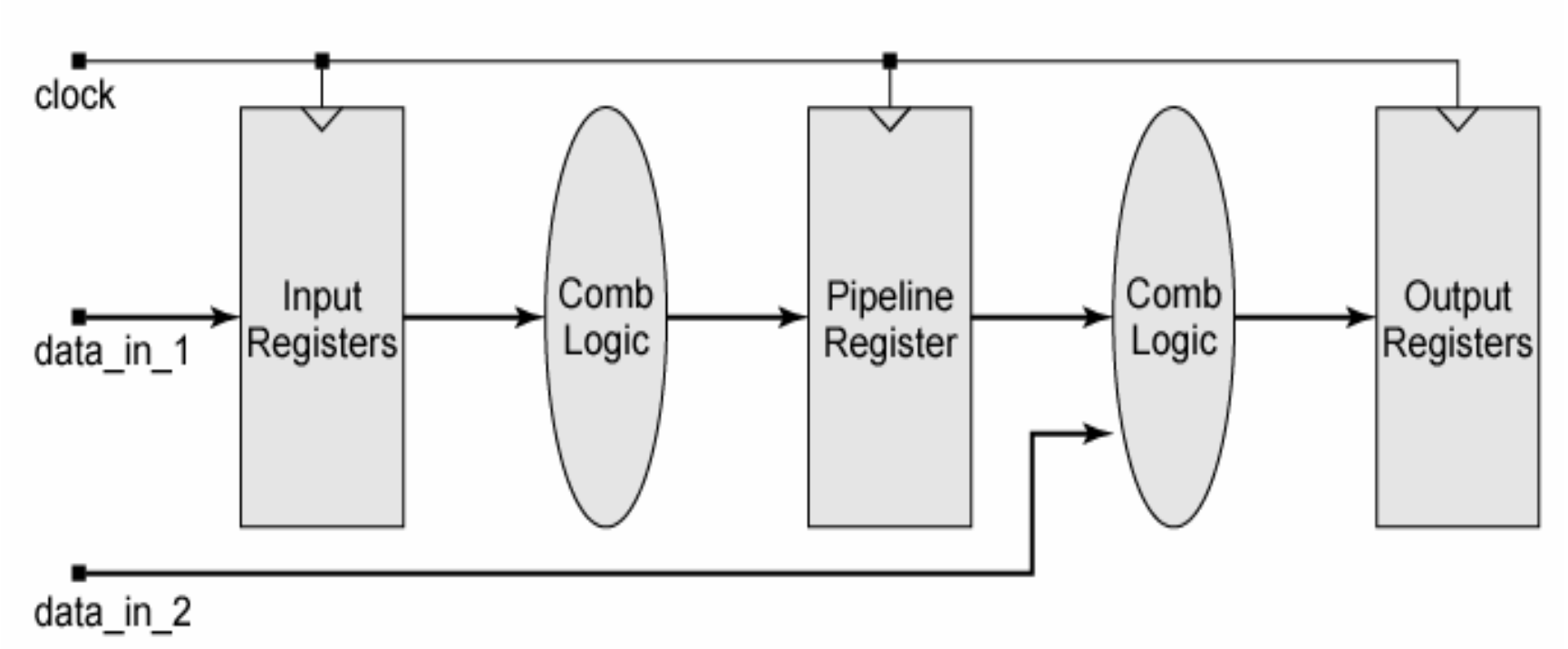
Pipeline registers can be inserted into the combinational logic datapaths **at strategic locations** to partition the logic into groups with shorter paths



Problem:

Find a design error in the circuit in following Figure.

Redesign the circuit to implement pipelining correctly.



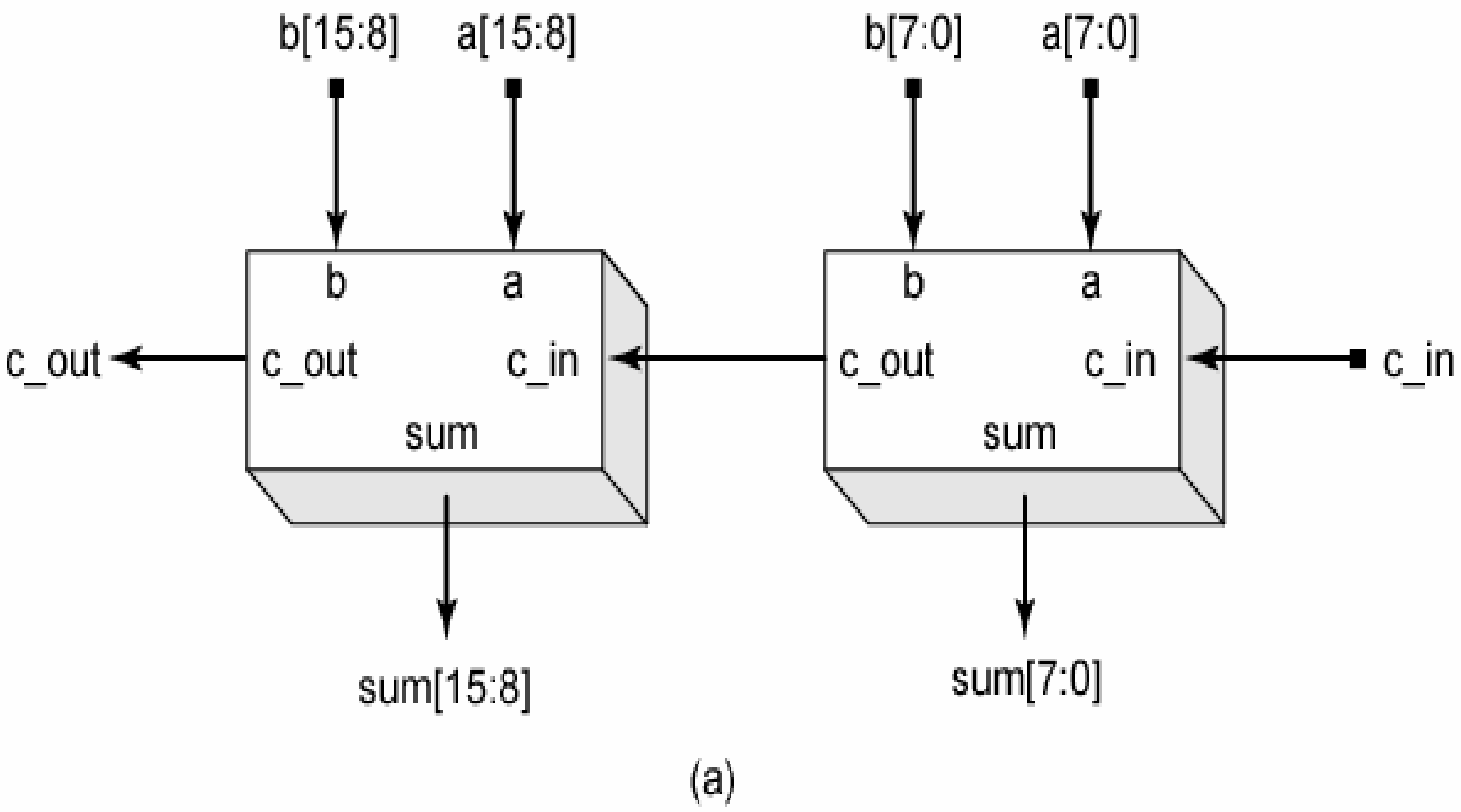
9.5.1 Design Example: Pipelined Adder

- Digital systems that operate on arrays of data typically contain a large number of adders in an array structure.
- The processing speed in these applications is usually critical, and may warrant pipelining.

The 16-bit adder

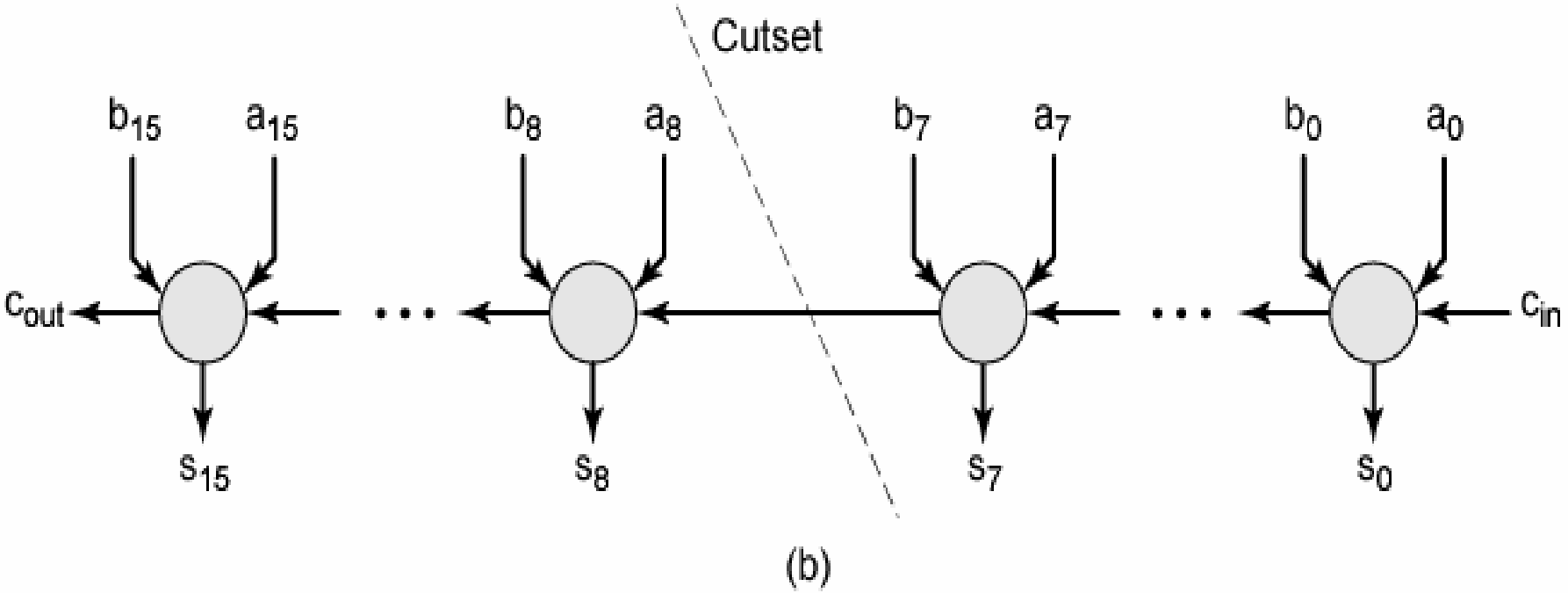
- The 16-bit adder is formed by chaining two 8-bit adders in a serial connection
- If each 8-bit adder has a throughput delay of 100 ns the worst-case delay of the configuration will be 200 ns
- In a synchronous environment, this structure is organized to have all operations occur in the same clock cycle
- An alternative structure can be pipelined to operate at a higher throughput by distributing the processing over multiple cycles of the clock

Serial connection of two 8-bit adder to form a 16-bit adder

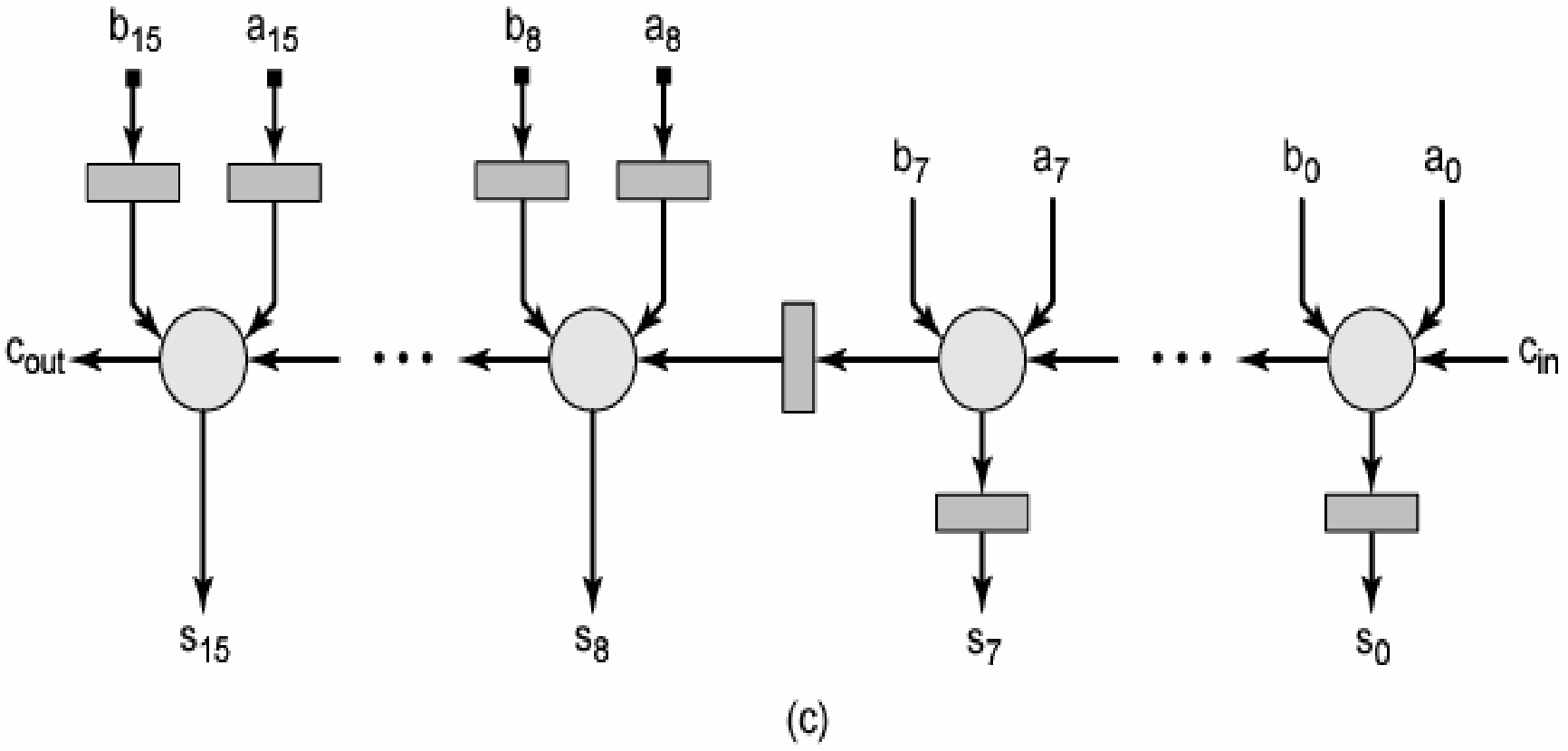


(a)

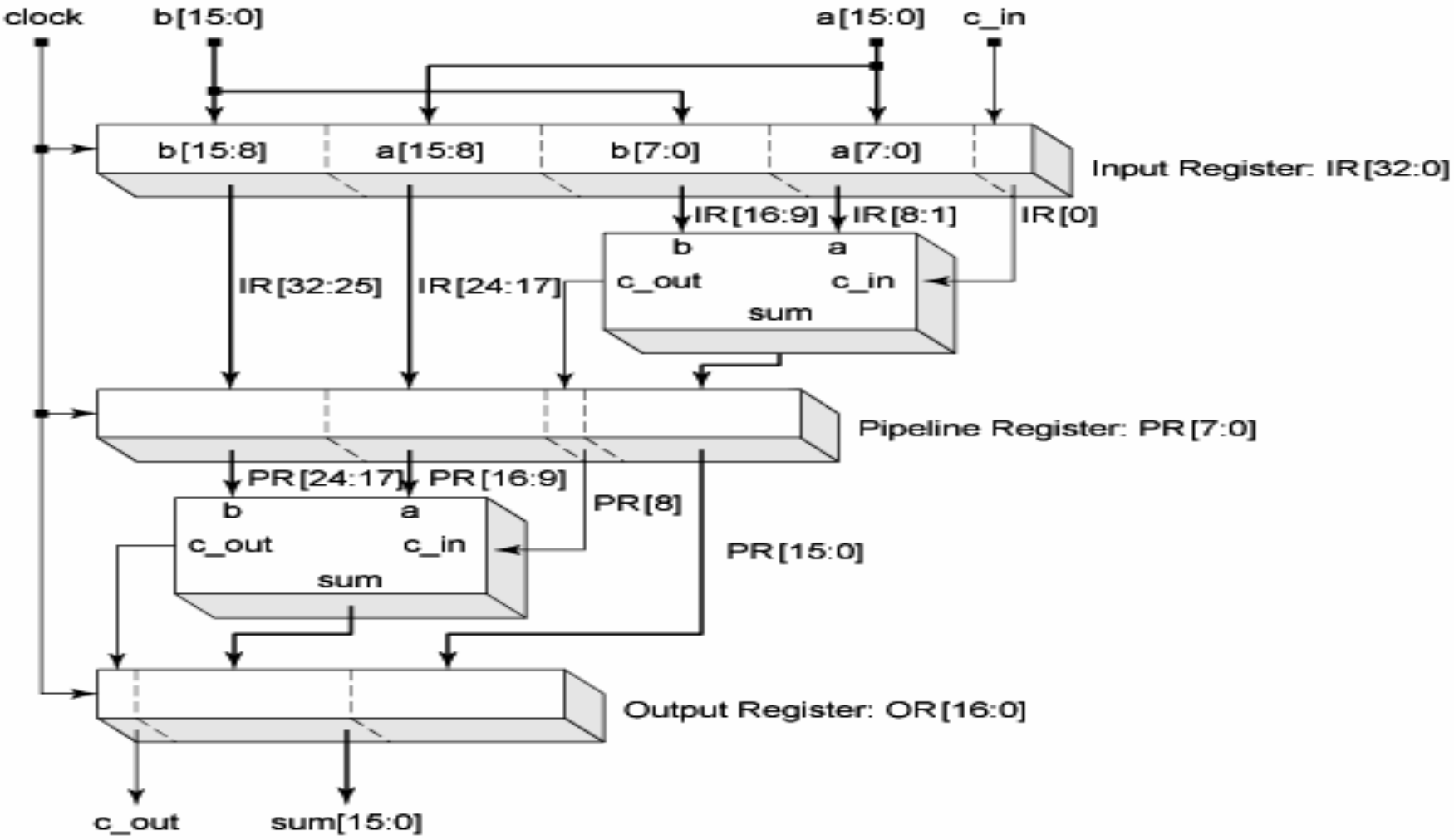
DFGs before pipelining



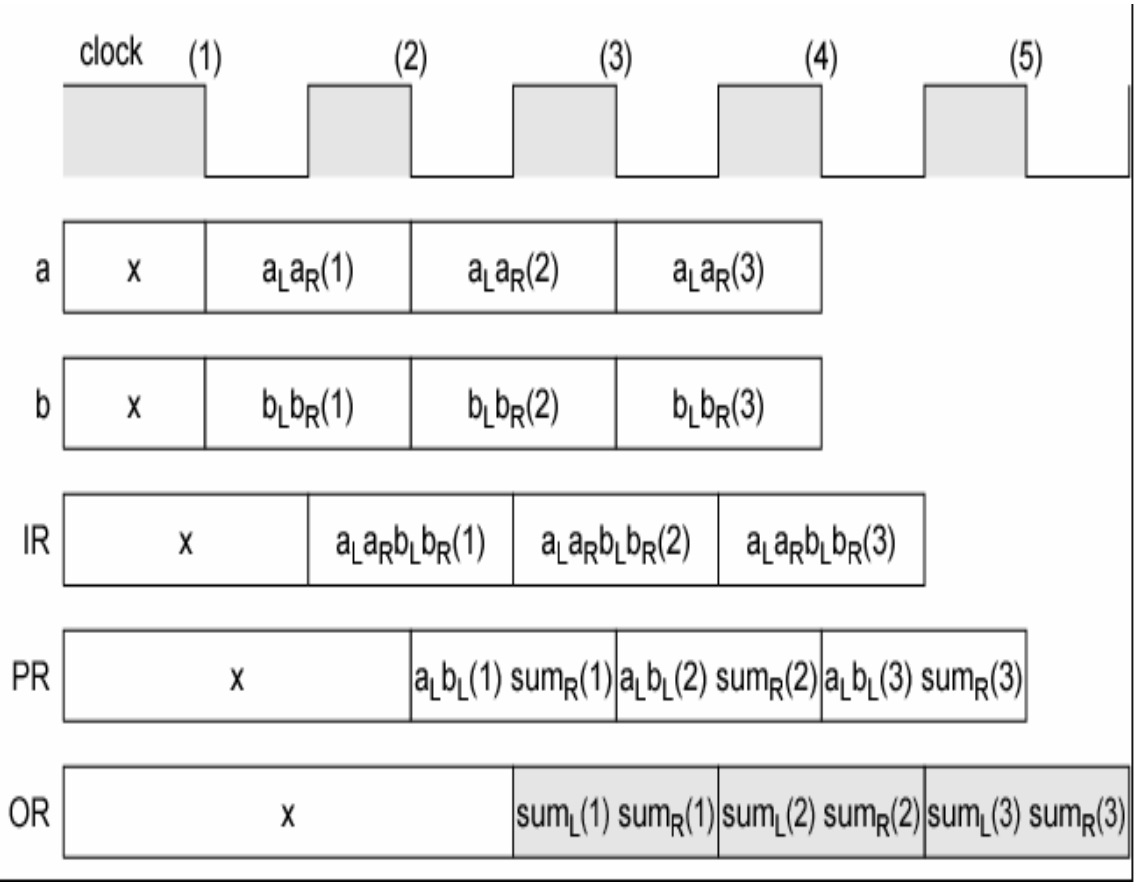
After pipelining for balanced stage delays

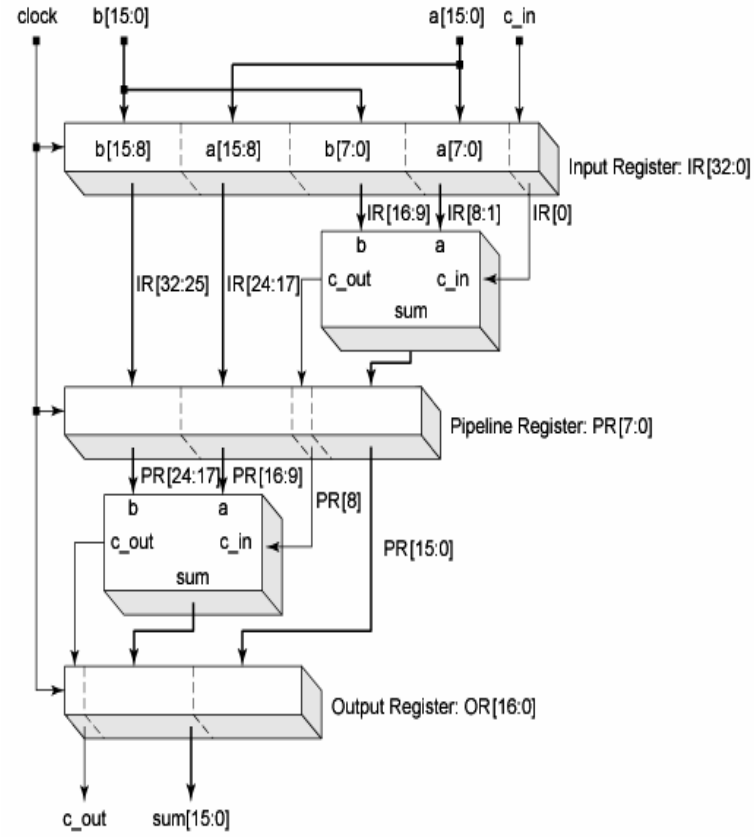


Pipelined 16-bit adder



Data movement through a pipelined 16-bit adder

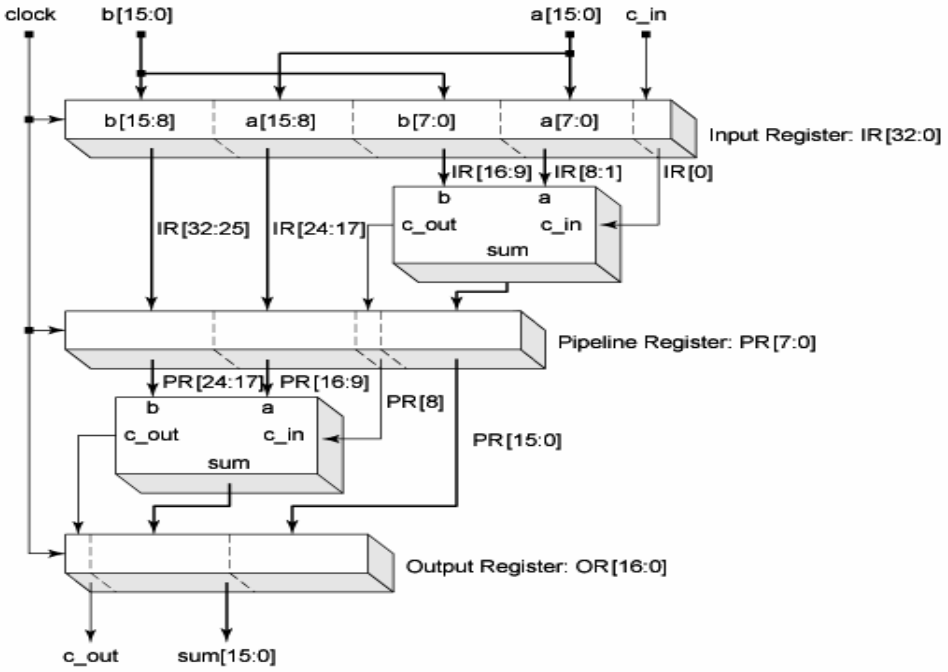




Pipelined Adder

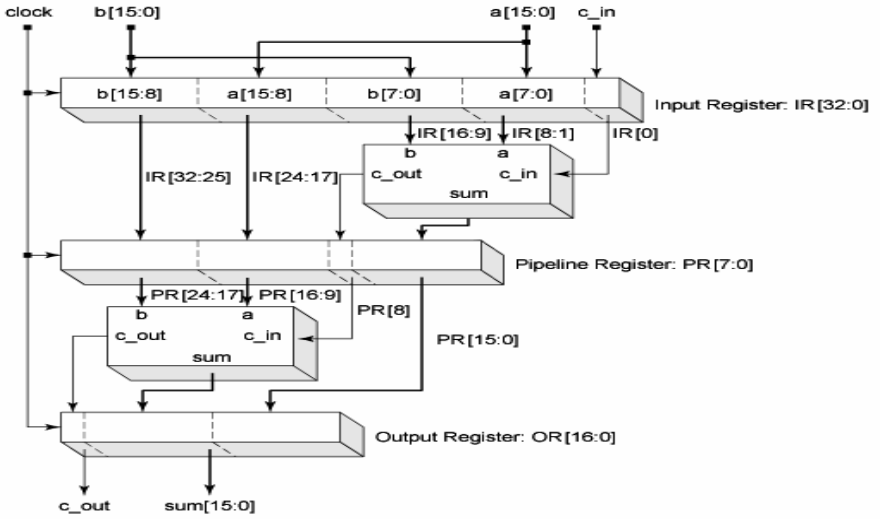
- IR: data input register
- OR: data output register
- PR: pipelined register
- **After using PR,the longest path is through an 8-bit adder instead a 16-bit one**
- So the frequency will be twice of the original one.

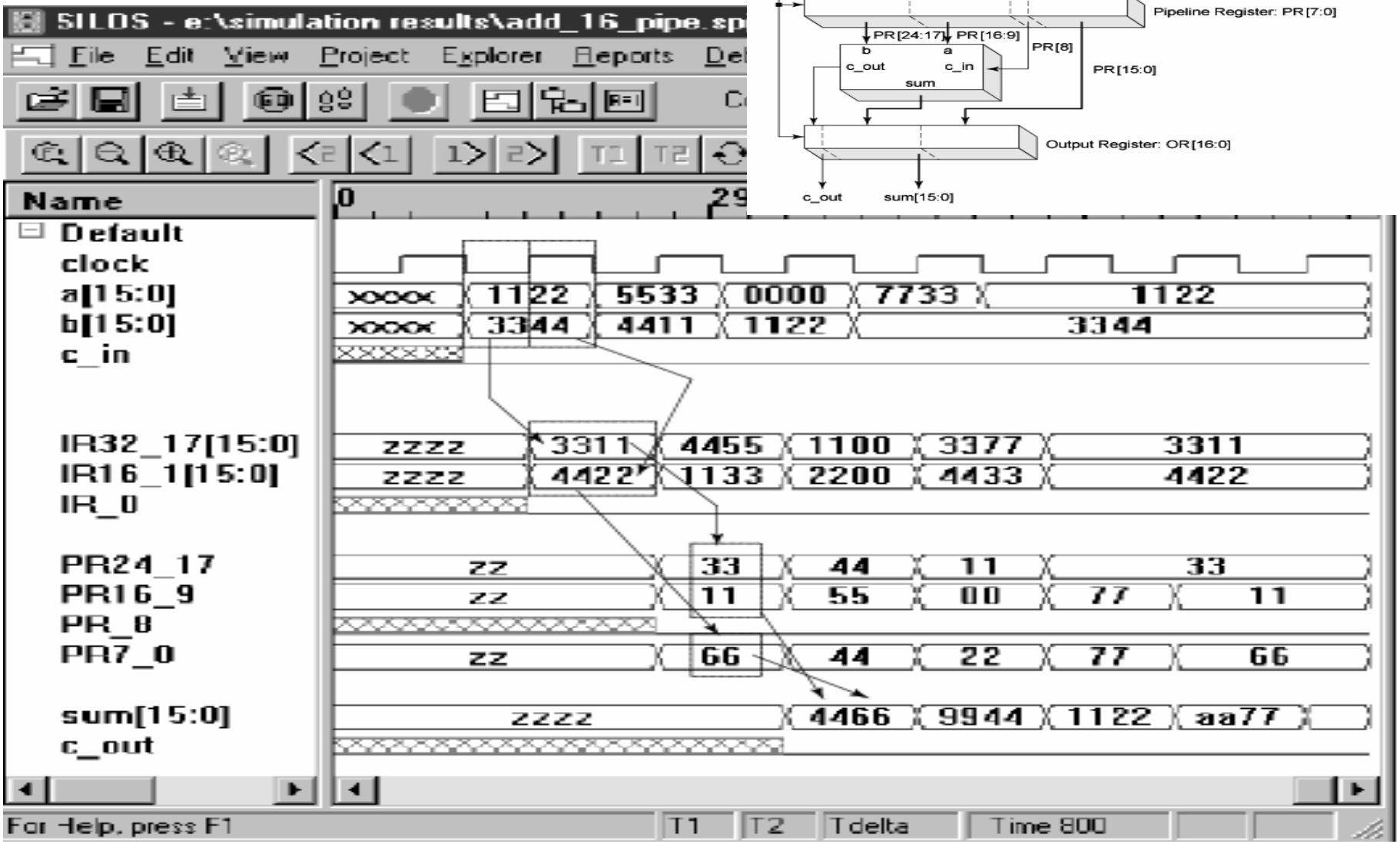
```
module add_16_pipe  (c_out,sum,a,b,c_in,clock,IR32_17,
                                IR16_1,PR24_17,PR16_9,PR7_0);
  parameter size=16;
  parameter half=size/2; //8
  parameter double=2*size;//32
  parameter triple=3*half;//24
  parameter size1=half-1;//7
  parameter size2=size-1;//15
  parameter size3=half+1;//9
  parameter R1=1;
  parameter L1=half;
  parameter R2=size3;
  parameter L2=size;
  parameter R3=size+1;
  parameter L3=size+half;
  parameter R4=double-half+1;
  parameter L4=double;

  input [size2:0] a,b;
  input   c_in,clock;
  output [size2:0] sum;
  output  c_out;
  output [size2:0] IR32_17;
  output [size2:0] IR16_1;
  output [half-1:0] PR24_17;
  output [half-1:0]  PR16_9;
  output [half-1:0]  PR7_0;
```




```
reg [double:0] IR;
reg [triple:0] PR;
reg [size:0]  OR;
assign {c_out,sum}=OR;
assign  IR32_17=IR[32:17];
assign  IR16_1=IR[16:1];
assign PR24_17=PR[24:17]
assign PR16_9=PR[16:9];
assign PR7_0=PR[7:0];
always@(posedge clock)begin
    IR[0]<=c_in;      //load input register
    IR[L1:R1]<=a[size1:0];   // IR[8:1]
    IR[L2:R2]<=b[size1:0];  // IR[17:9]
    IR[L3:R3]<=a[size2:half];  // IR[25:18]
    IR[L4:R4]<=b[size2:half];   // IR[32:26] //load pipeling register
    PR[L3:R3]<=IR[L4:R4];  //PR[24:17]
    PR[L2:R2]<=IR[L3:R3];  // PR[16:9]
    PR[half:0]<=IR[L2:R2]+IR[L1:R1]+IR[0]; //PR[8:0] // 7-0 bits add
    OR<={{1'b0,PR[L3:R3]}+{1'b0,PR[L2:R2]}+PR[half],PR[size1:0]};
  end
endmodule
```

Simulation result

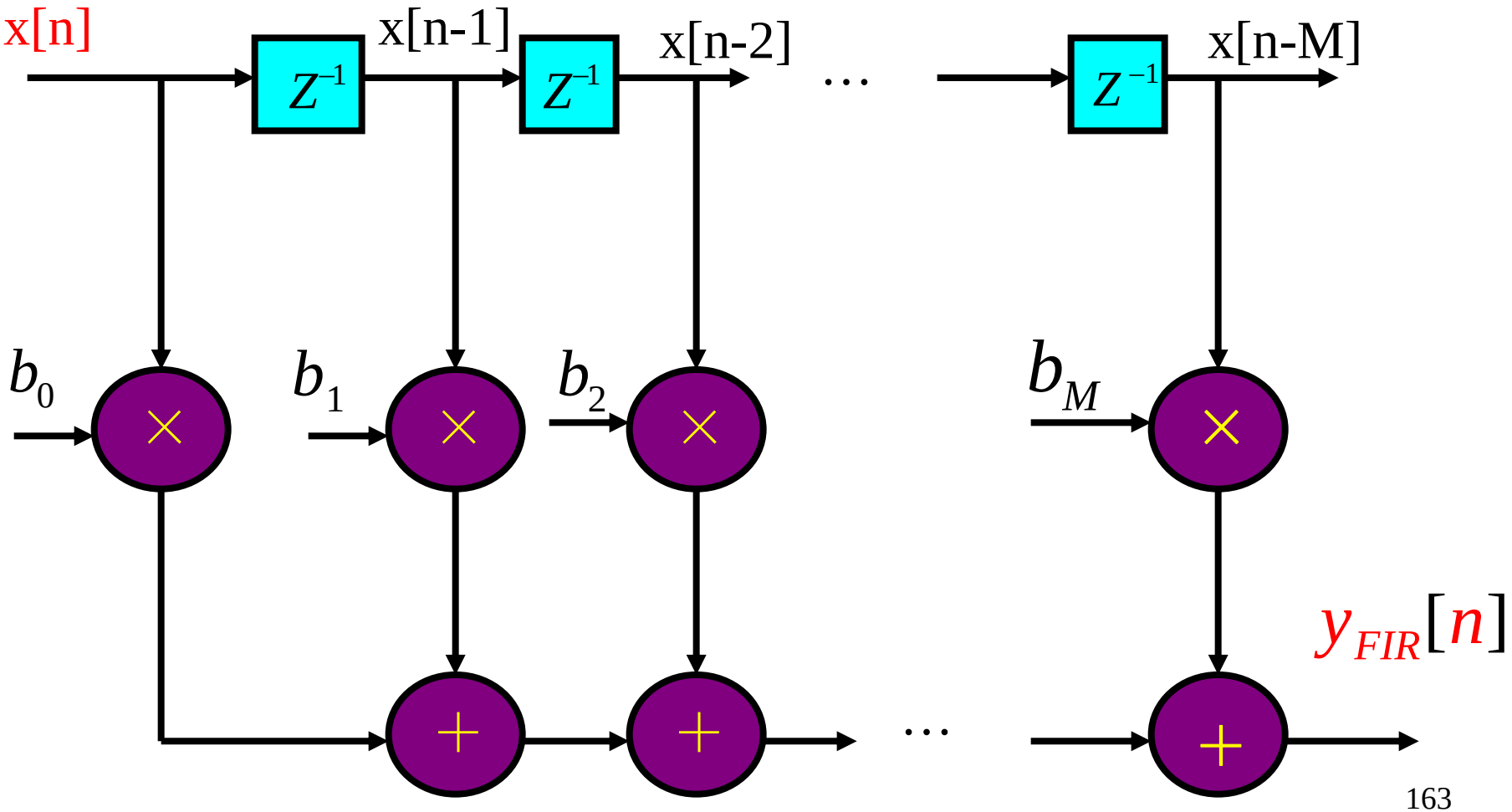




9.5.2 Design Example: Pipelined FIR Filter

- MACs dominate the performance of digital signal processors
- In many applications long chains of MACs must be pipelined to increase the throughput of the unit
- For example, the architecture of the FIR filter that was presented in Figure 9-22 consists of a shift register and an array of cascaded MAC units

Fig 9.22 Functional block diagram for an Mth-order FIR digital filter



To increase the throughput

- The longest path through the circuit is proportional to the length of the chain of MACs between the input and the output
- **The methods of increasing the throughput**

① **To require that high-speed multipliers and/or adders be used to implement MACs**

② **To pipeline the datapaths to increase the throughput of the filter**

The FIR filter can be pipelined in a variety of ways

① In Figure 9-41(a) the cutsets place the pipeline registers at the output of the multipliers

② The cutsets of the alternative structure in Figure 9-41(b) have pipeline registers at the inputs of the adders

- Note: The structure in Figure 9-41(b) **must be modified to achieve coherency of the datapaths**

Fig 9-41(a) Pipeline at the outputs of the multipliers

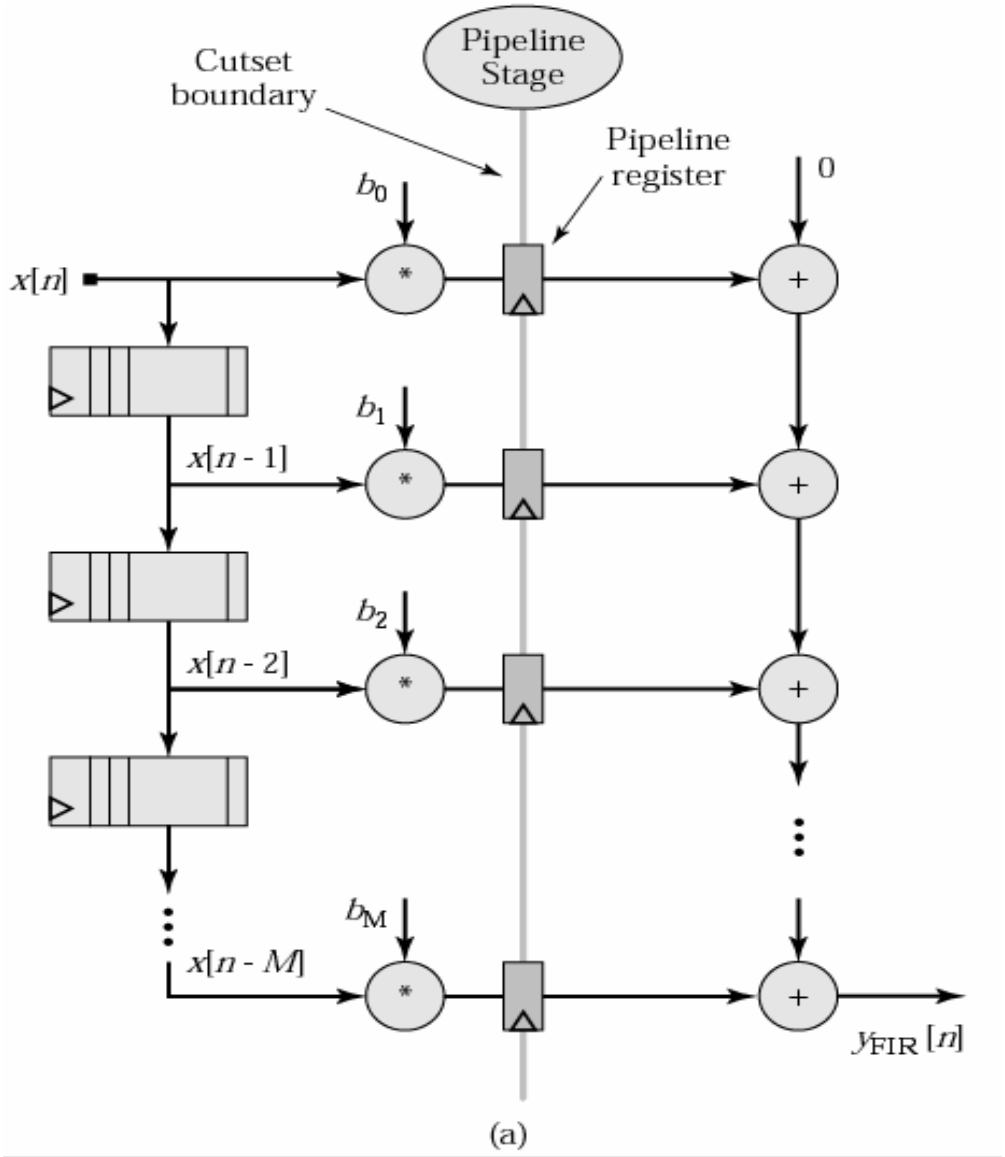
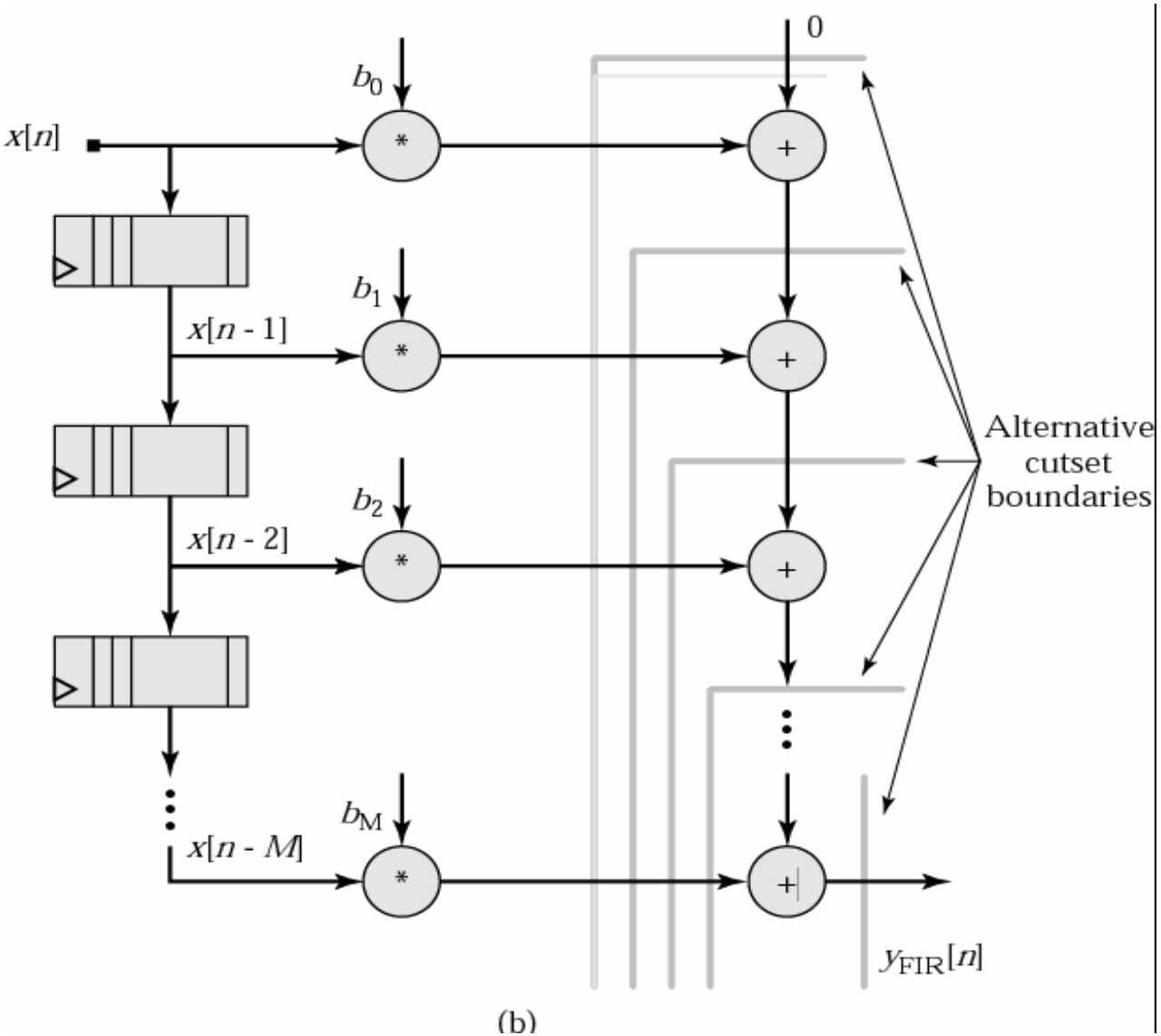


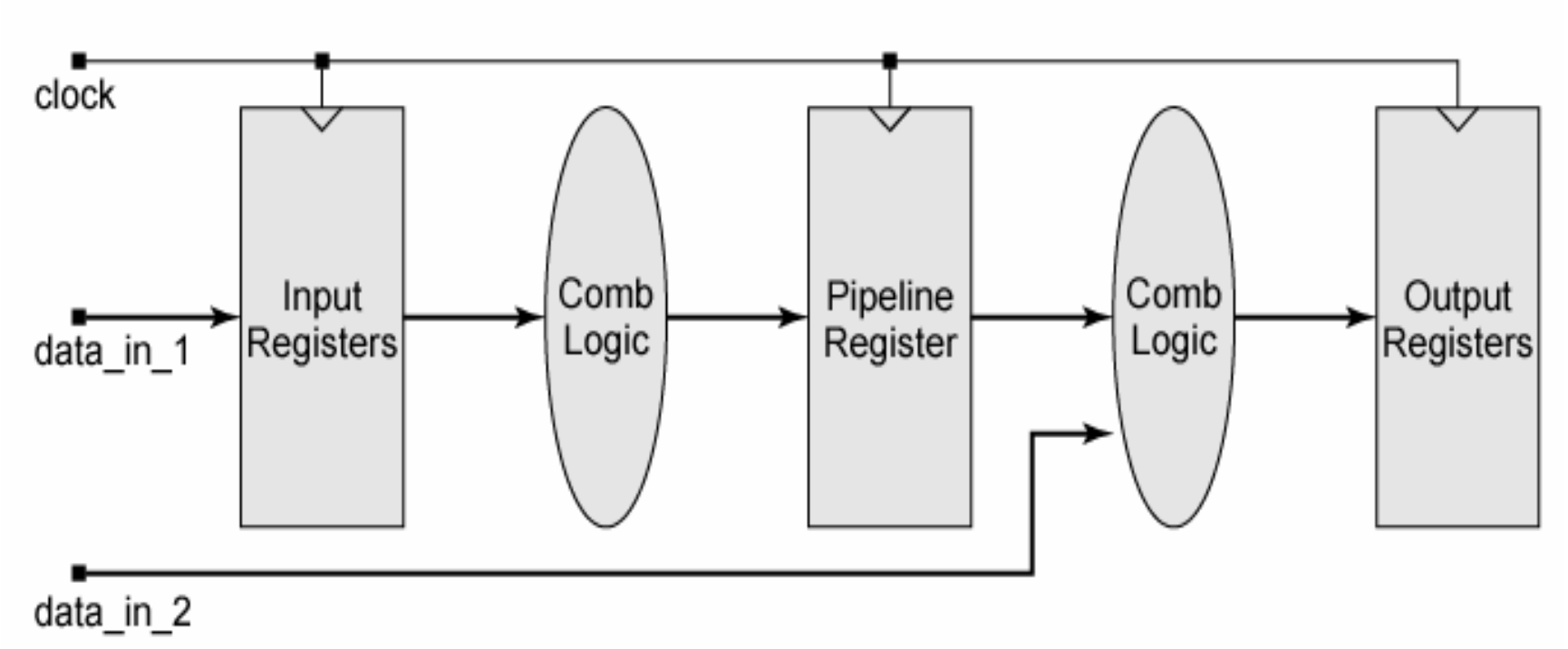
Fig 9-41(b) Pipeline at the inputs of the adders



Problem:

Find a design error in the circuit in following Figure.

Redesign the circuit to implement pipelining correctly.



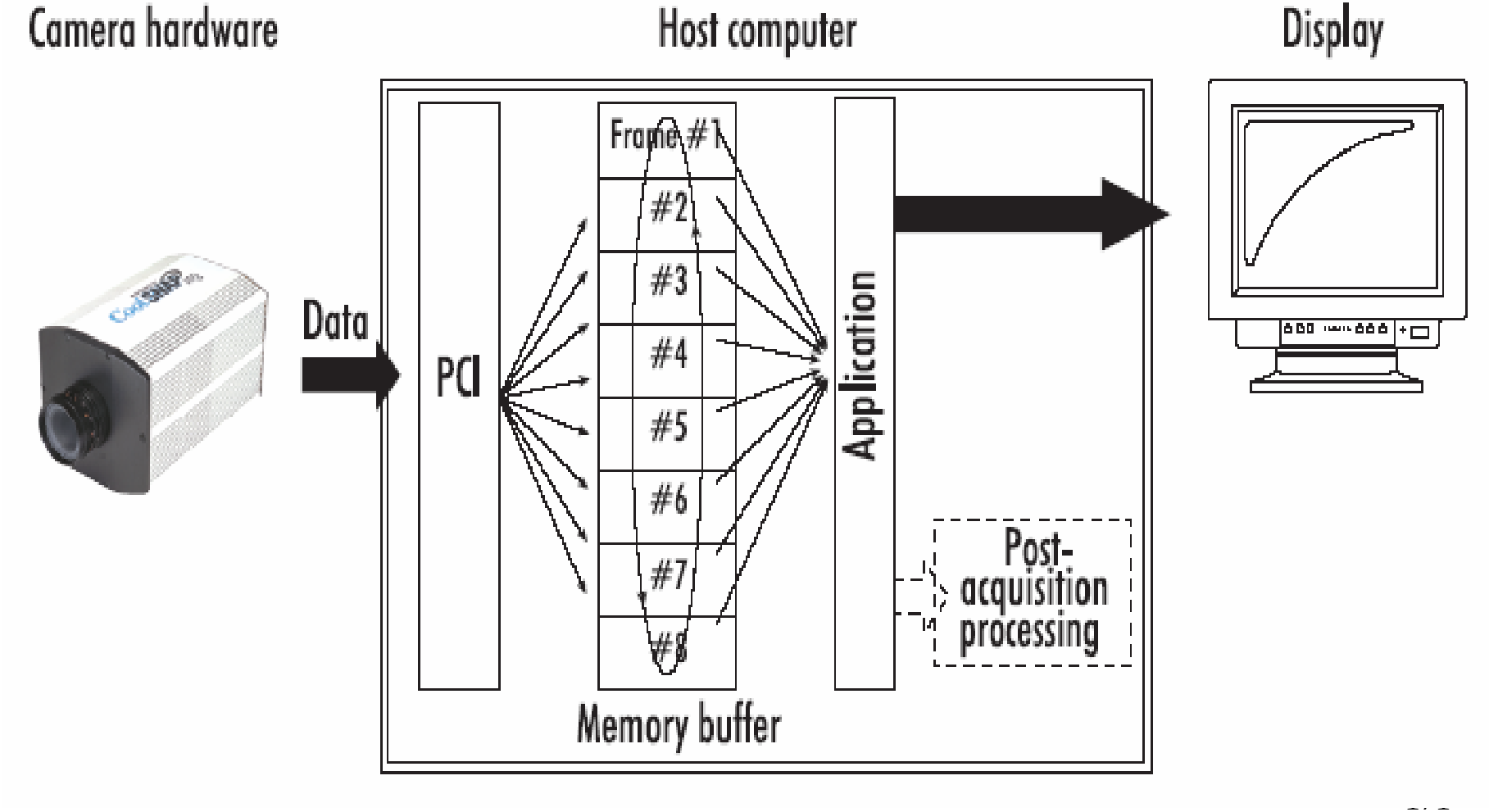
The caution

The data's datapath traced from any primary input to any primary output must pass through the same number of pipeline registers

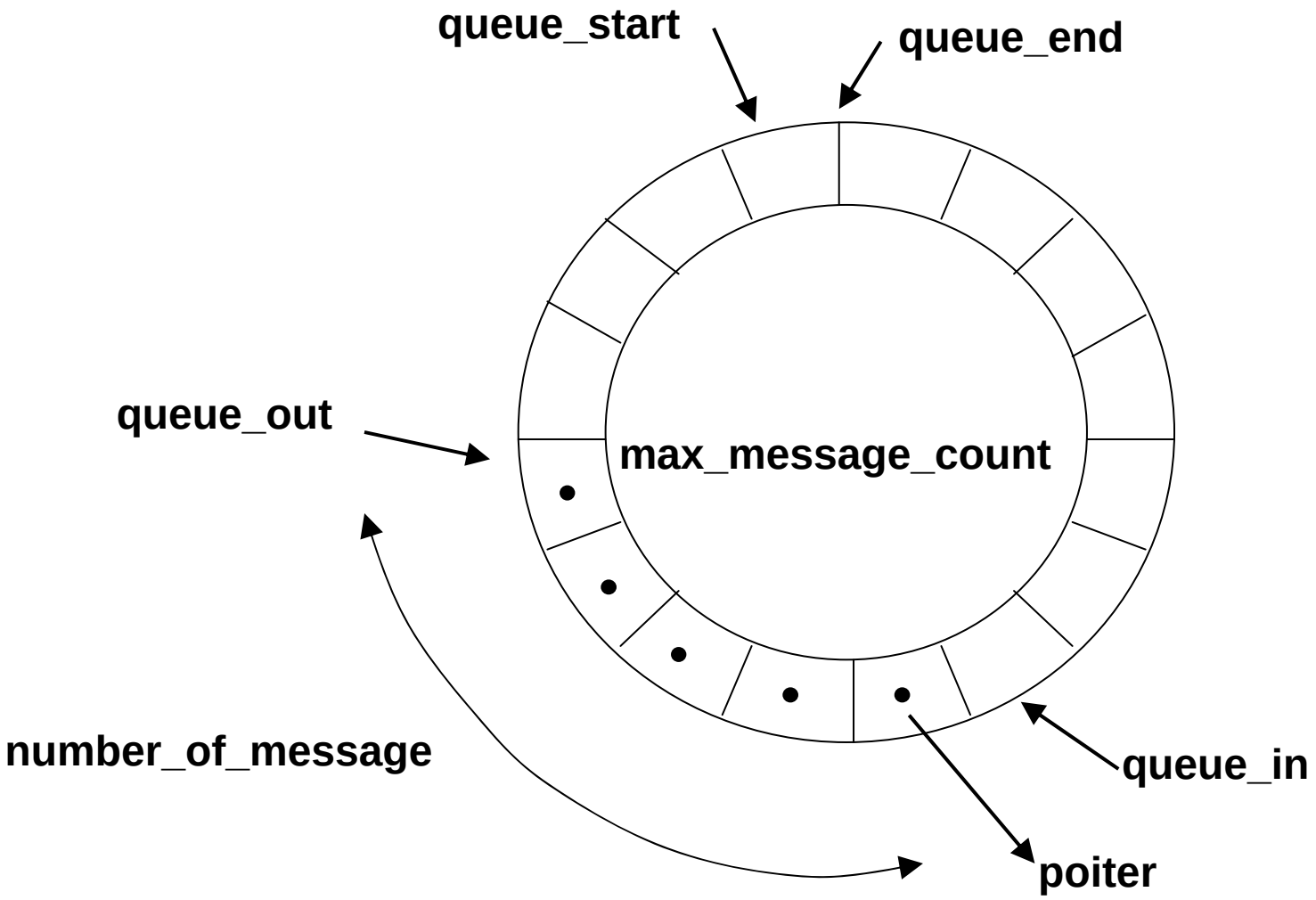
9.6 Circular buffers

- The algorithms of many digital filters and other signal processors repeatedly shift and store samples of data that are taken over a moving window in the time-sequence domain
- For example, a filter might form its output from a weighted sum of samples of the present and most recent N-1 samples of the input

The concept of circular buffer



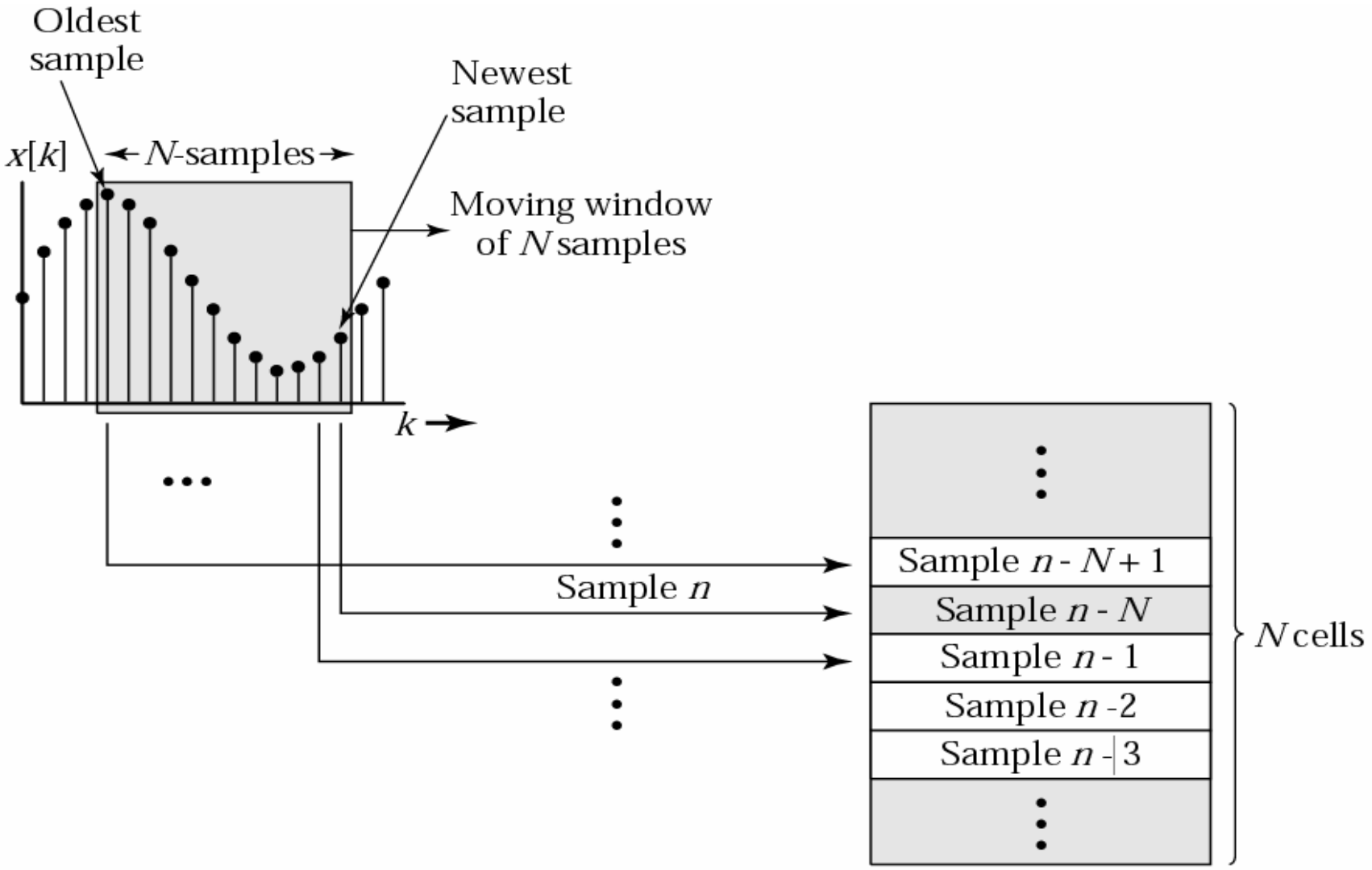
Data Structure



Circular buffers

- **Circular buffers use an address mechanism that moves pointers to register cells, instead of moving the actual data**
- To realize **circular buffers hardware** efficiency and speed, are used to create the effect of moving a N entire window of samples through memory, without actually moving all of the data

Fig 9.42 An circular buffer storing data from an N-sample moving window



Circular buffers

- When the nth sample is received, the data held in the entire array of registers is not shifted
- The cell addressed by the pointer receives the nth sample, overwriting the previous contents at the location previously occupied by the n-Nth sample
- **The net effect is that the storage of data can occur in significantly fewer memory cycles**

Example 9.9

- Two Verilog models of an N-sample moving window memory are described below
- The first version, Circular_Buffer_1, uses an array of parallel-load shift registers to shift the entire contents of the buffer at each time step

```
module Circular_Buffer_1
        (cell_3, cell_2, cell_1, cell_0, Data_in, clock, reset);
    parameter    buff_size = 4;
    parameter    word_size = 8;
    output       [word_size -1: 0]  cell_3, cell_2, cell_1, cell_0;
    input        [word_size -1: 0] Data_in;
    input        clock, reset;
    reg          [word_size-1: 0]  Buff_Array [buff_size-1: 0];
                                          //4 * 8 bits memory units
    wire  cell_3 = Buff_Array[3],
          cell_2 = Buff_Array[2];
     wire cell_1 = Buff_Array[1],
          cell_0 = Buff_Array[0];
    integer      k;
  always @ (posedge clock) begin
   if (reset == 1)
     for (k = 0; k <= buff_size -1; k = k+1)
       Buff_Array[k] <= 0;
   else for (k = 1; k <= buff_size -1; k = k+1) begin
       Buff_Array[k] <= Buff_Array[k-1];
       Buff_Array[0] <= Data_in;
     end
   end
endmodule
```

The second version

- Circular_Buffer_2, includes write_ptr, which points to the next cell to be read
- The contents of the register do not move
- The pointer is incremented at each time step, so the data below the pointer is aged

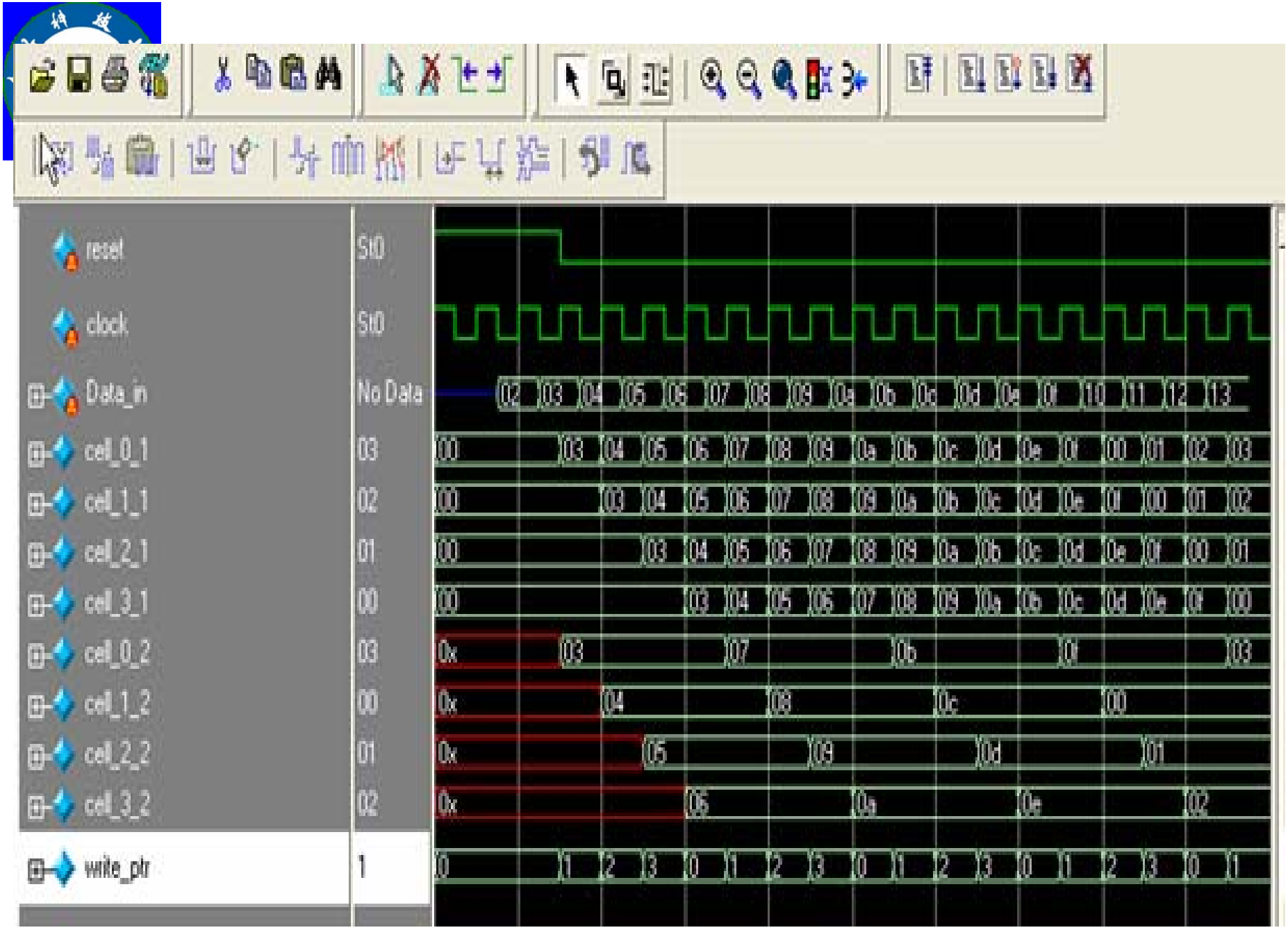
```
module Circular_Buffer_2 (cell_3, cell_2, cell_1, cell_0,
                                        Data_in, clock, reset);
 parameter buff_size = 4;
 parameter word_size = 8;
 output  [word_size -1: 0]  cell_3, cell_2, cell_1, cell_0;
 input       [word_size -1: 0] Data_in;
 input       clock, reset;
 reg   [word_size -1: 0]  Buff_Array [buff_size -1: 0];
                    //4 *  8 bits memory units
 wire       cell_3 = Buff_Array[3], cell_2 = Buff_Array[2];
 wire       cell_1 = Buff_Array[1], cell_0 = Buff_Array[0];
 integer     k;
 parameter write_ptr_width = 2;
       // Width of write pointer
 parameter          max_write_ptr = 3;
 reg  [write_ptr_width -1 : 0]      write_ptr;
       // Pointer writing for
```

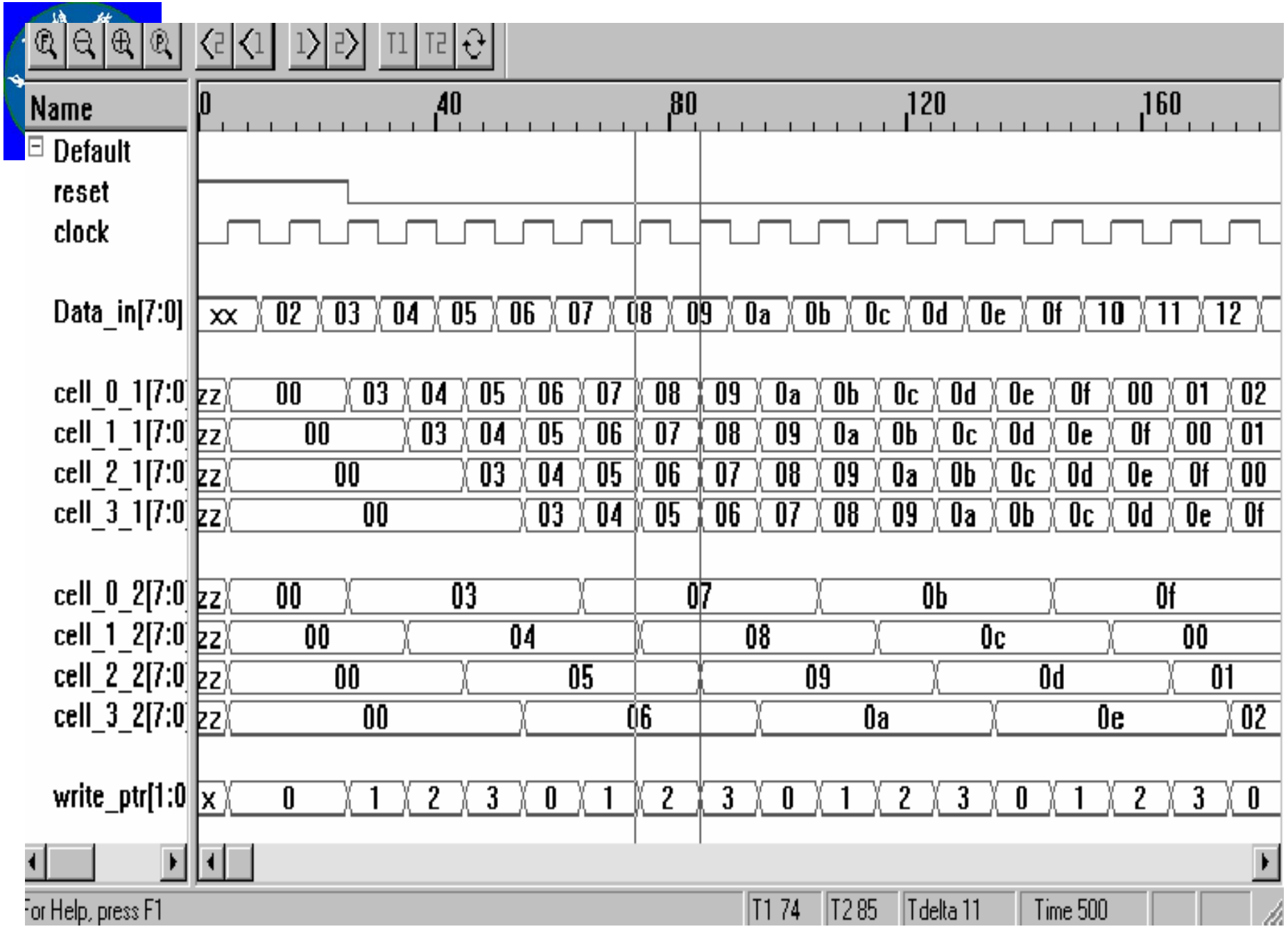
```
always @ (posedge clock) begin
    if (reset == 1 ) begin
      write_ptr <= 0;
      for (k = 0; k <= buff_size -1; k = k+1)
            Buff_Array[k] <= 0;
      end
      else begin
         Buff_Array[write_ptr] <= Data_in;
         if (write_ptr < max_write_ptr)  // < 3 ,loop
           write_ptr <= write_ptr + 1;
         else write_ptr <= 0;
       end
  end
endmodule
```

Note

- The contents of Circular_Buffer_1 change at every cycle,

  while only the one cell addressed by write_ptr in Circular_Buffer_2 changes as new data arrive


reset
clock
Data_in
cell_0_1
cell_1_1
cell_2_1
cell_3_1
cell_0_2
cell_1_2
cell_2_2
cell_3_2
wrike_ptr
StD
No Data
1


Name
0
40
80
120
160
Default
reset
clock
Data_in[7:0]
xx 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12
cell_0_1[7:0
zz 00 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 00 01 02
cell_1_1[7:0
zz 00 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 00 01
cell_2_1[7:0
zz 00 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 00
cell_3_1[7:0
zz 00 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
cell_0_2[7:0
zz 00 03 07 0b 0f
cell_1_2[7:0
zz 00 04 08 0c 00
cell_2_2[7:0
zz 00 05 09 0d 01
cell_3_2[7:0
zz 00 06 0a 0e 02
write_ptr[1:0
x 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0
For Help, press F1
T1 74
T2 85
Tdelta 11
Time 500

9.7 FIFOs and Synchronization across Clock Domains

- FIFO can implement high-performance parallel interfaces between independent clock domains
- **A FIFO consists of a block of *memory* and a *controller* that manages the traffic of data to and from the FIFO**
- FIFO' architecture provides **access to only one register cell at a time**
- It has two address points ,one for *writing*, another for *reading*

Fig 9-44 FIFO Buffer: input-output port

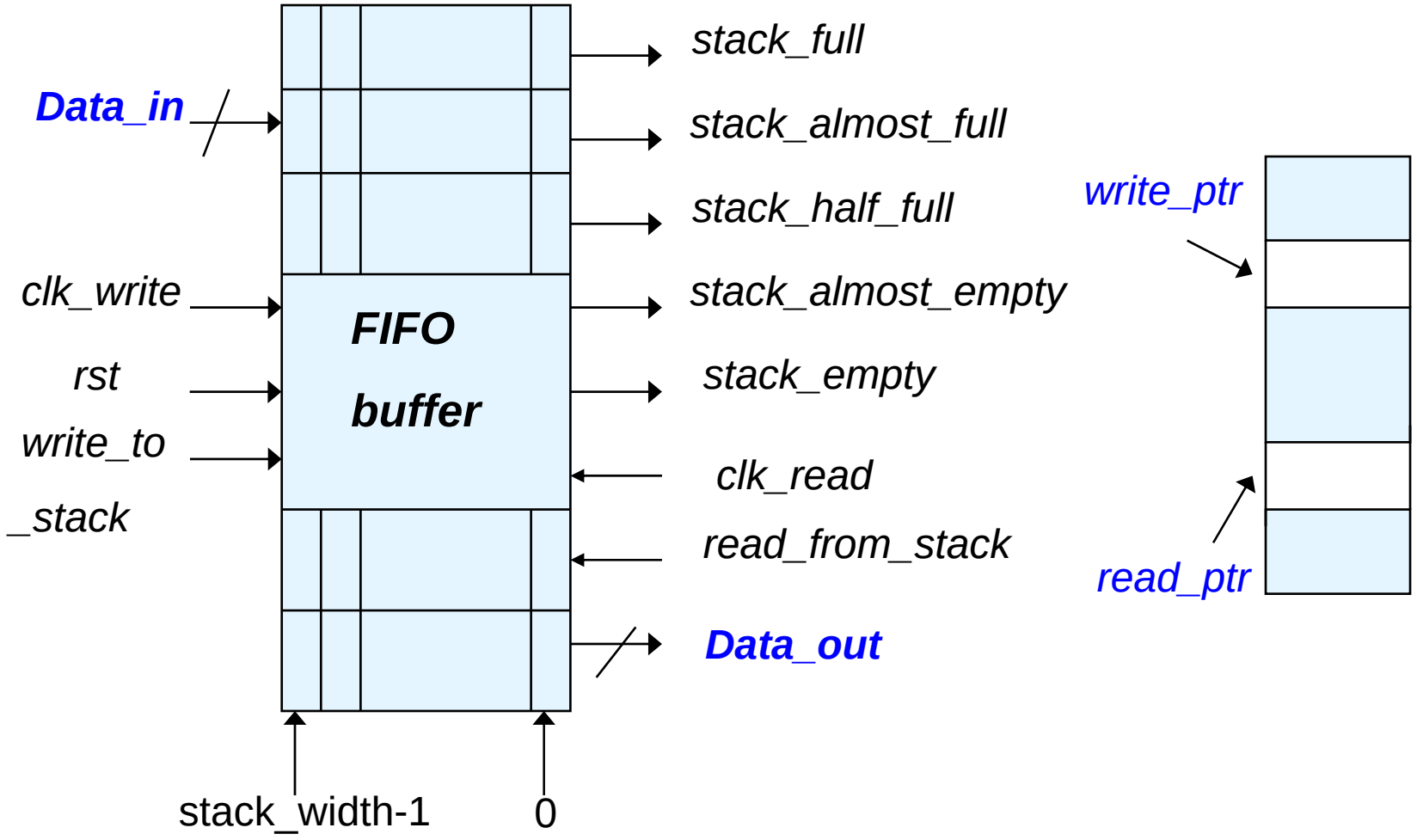
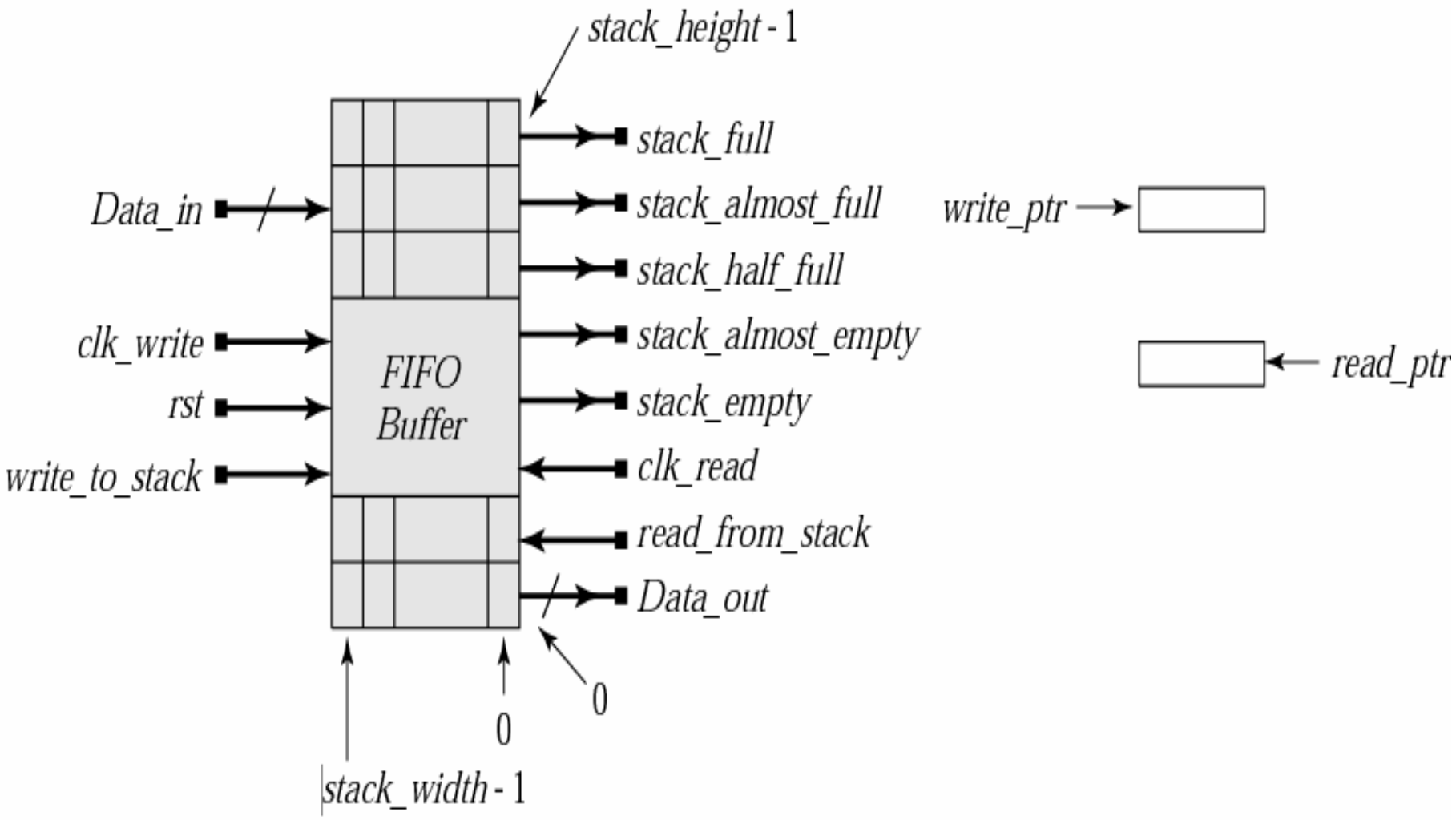


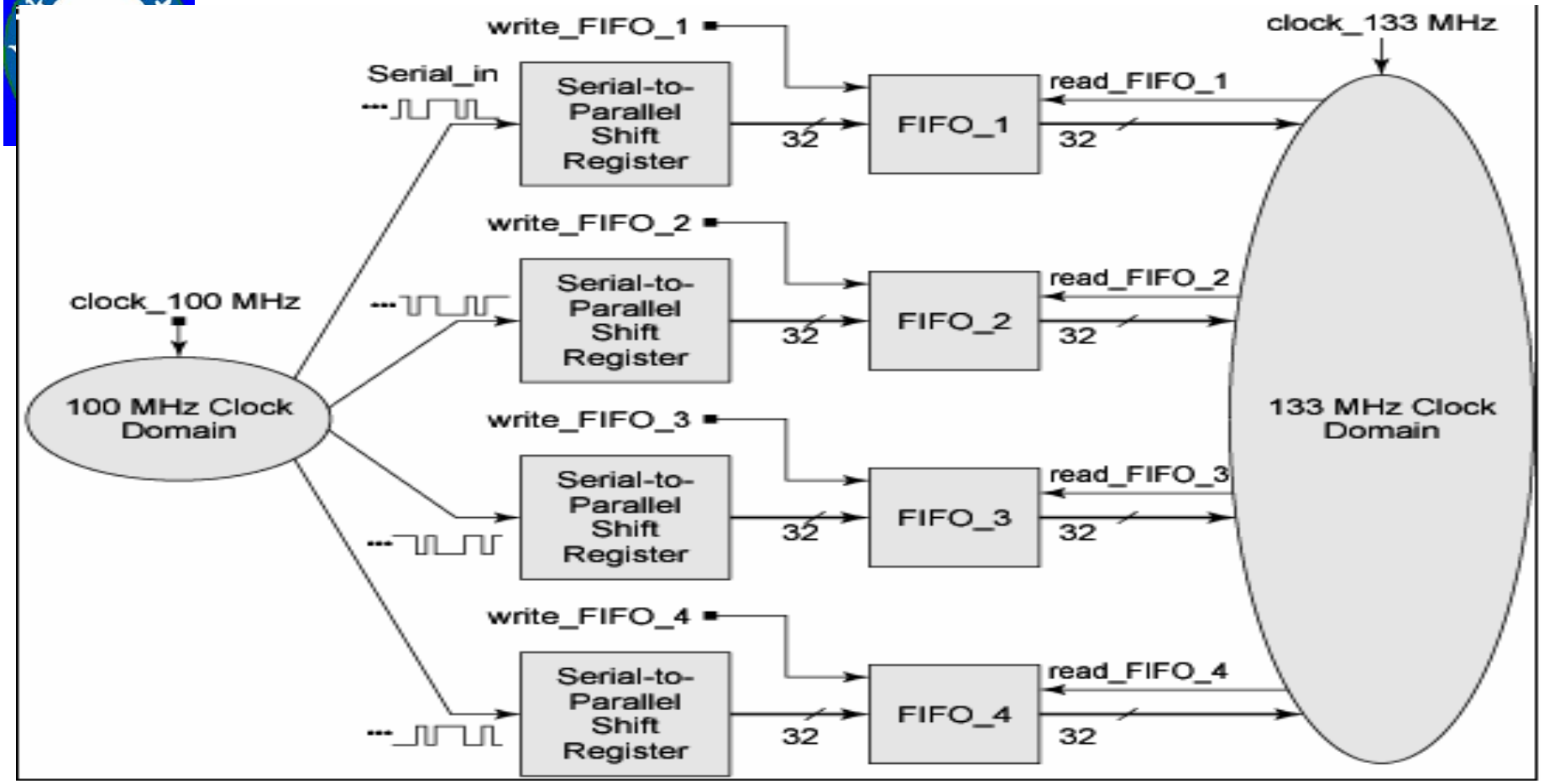
Fig 9.44 FIFO Buffer : I/O Ports

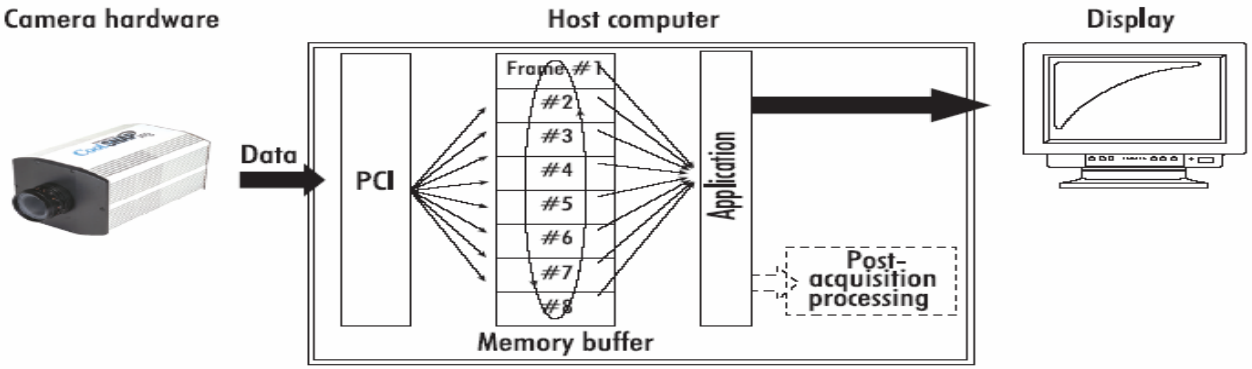


Example 9.10

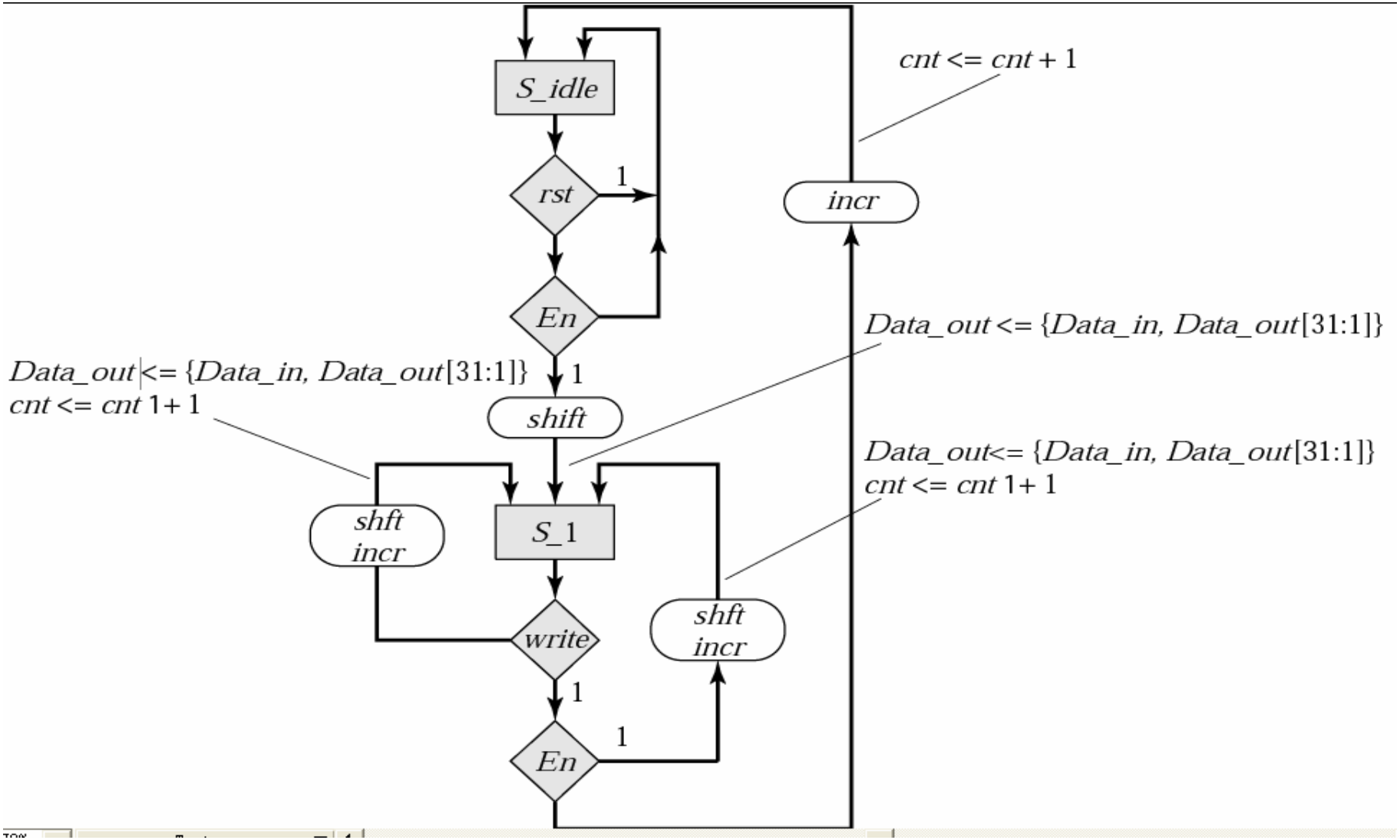
- The multi-channel circuit in Figure 9-45 has four channels of serial bit streams originating in a 100 MHz clock domain, with each passing though a serial-to-parallel converter that forms a 32-bit word for transfer to a dual-port FIFO
- Data are directed to each serial-to-parallel converter at a rate of 100 MHz;
- A processor operating with a clock of 133 MHz is to read data from the FIFOs and multiplex the four channels of data onto a common datapath

A FIFO –Buffered clock domain interface





ASMD Chart for a 32bit serial to parallel converter: Data_out,incr



A 32 bits serial to parallel converter

```
module Ser_Par_Conv_32
            (Data_out, write, Data_in, En, clk, rst);
    output [31: 0]    Data_out;
    output            write;
    input             Data_in;
    input             En, clk, rst;
    parameter         S_idle = 0;
    parameter         S_1 = 1;
    reg               state, next_state;
    reg               [4: 0]   cnt;
    reg               Data_out;
    reg               shift, incr;
```

```
always @ (posedge clk or posedge rst) //reset or next state
   if (rst) begin state <= S_idle; cnt <= 0; end
    else state <= next_state;
 always @ (state or En or write) // find  next state
  begin
   shift = 0;  incr = 0;
   next_state = state;
   case (state)
   S_idle:if (En) begin next_state  = S_1; shift = 1; end
   S_1: if (!write) begin shift = 1; incr = 1; end
            else if (En) begin shift = 1; incr = 1;   end
              else begin next_state = S_idle; incr = 1; end
   endcase
  end
 always @ (posedge clk or posedge rst) //count
   if (rst)   begin cnt <= 0;  end
      else if (incr)   cnt <= cnt +1;
 always @ (posedge clk or posedge rst) //shift & output
        if (rst)   Data_out <= 0;
         else if (shift)   Data_out <= {Data_in, Data_out [31:1]};
  assign write = (cnt == 31);// shifting  32 bits ,write enable
endmodule
```

Fig 9.47 Simulation Result

(The format of the arriving data has the LSB of a word arriving first)

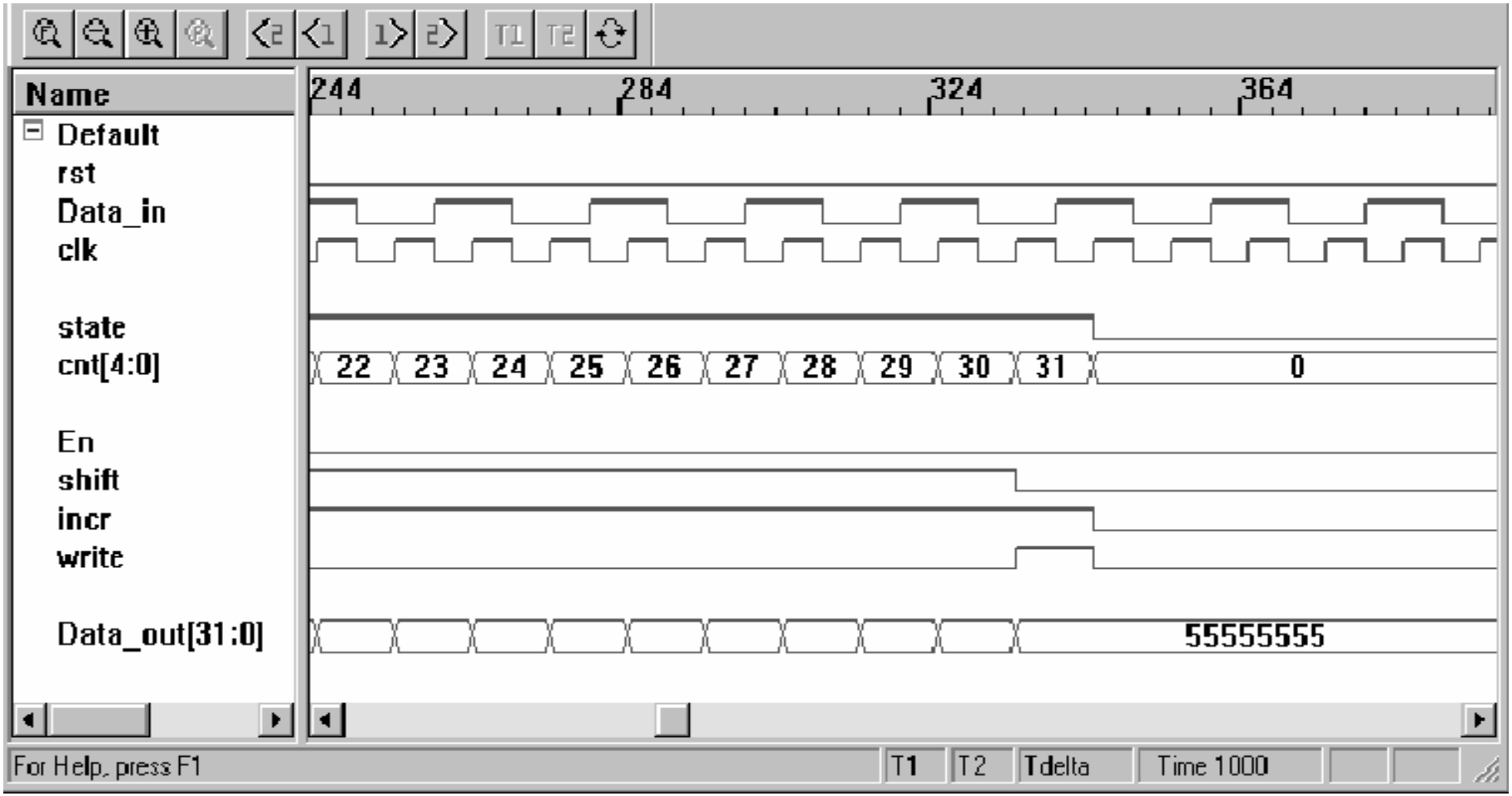


Fig 9.45 A FIFO –Buffered clock domain interface

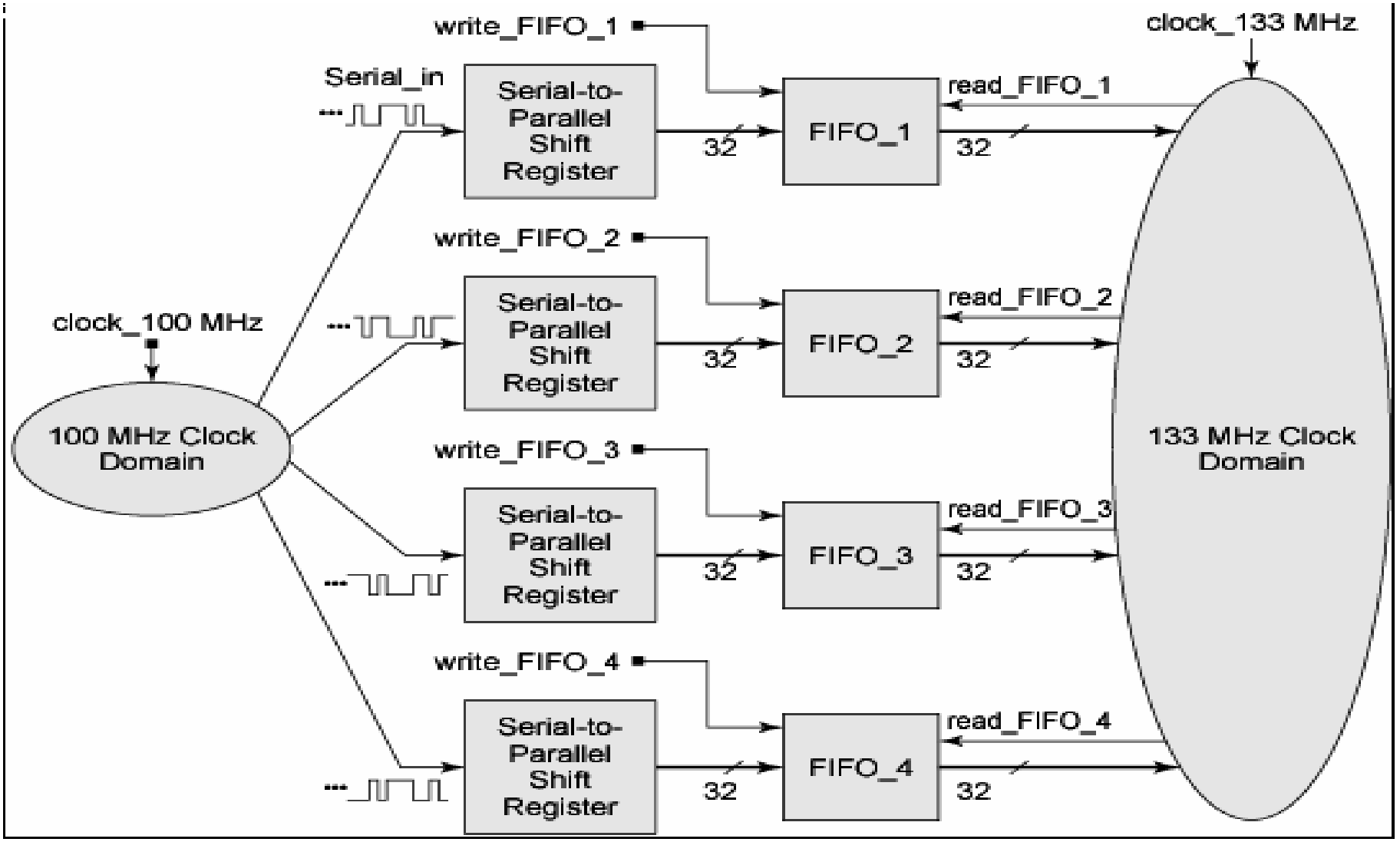
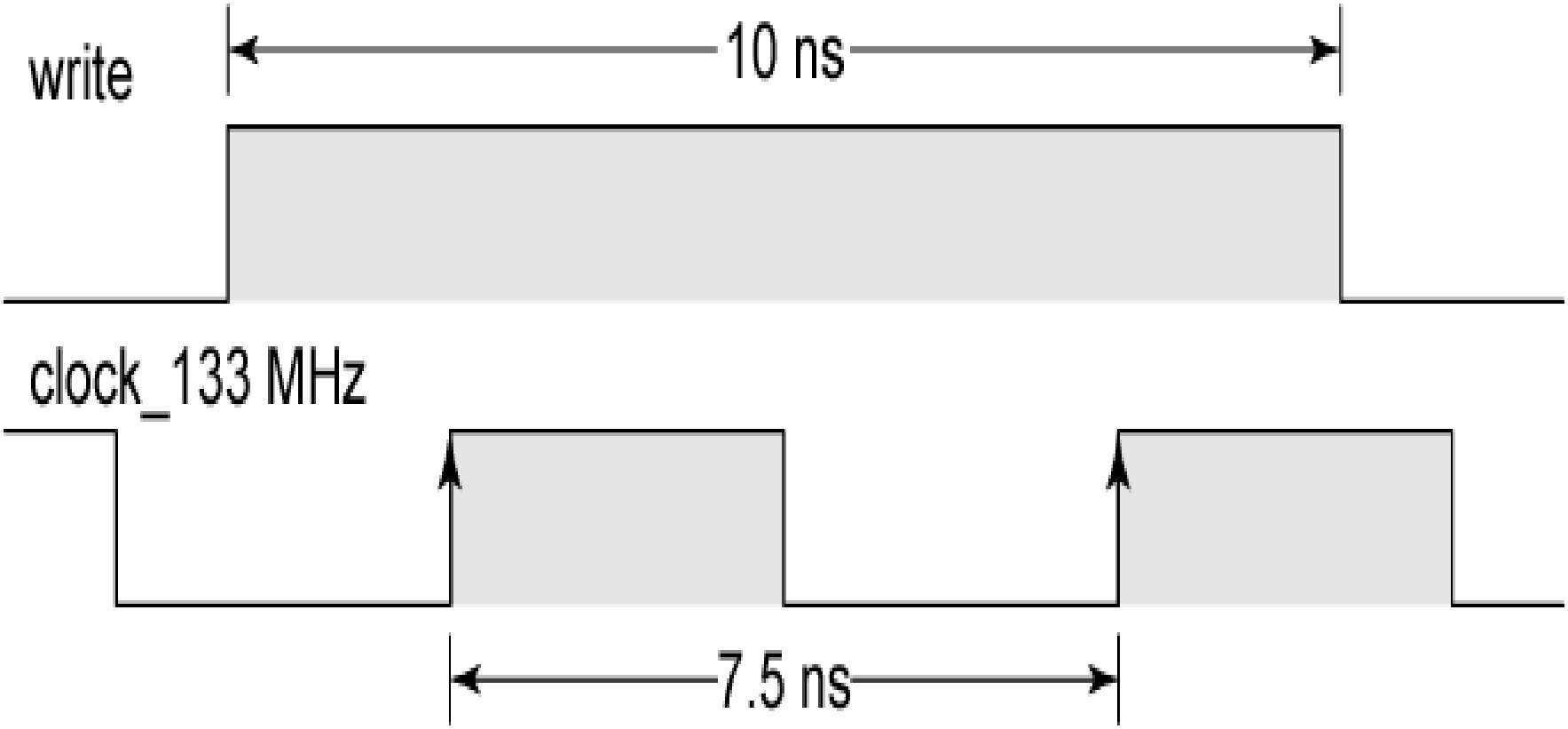


Fig 9.48 The duration of the asynchronous write pulse covers two edges of clock_133MHZ



synchronizer circuits for use *when the width of the asynchronous input pulse is greater than the period of the clock*

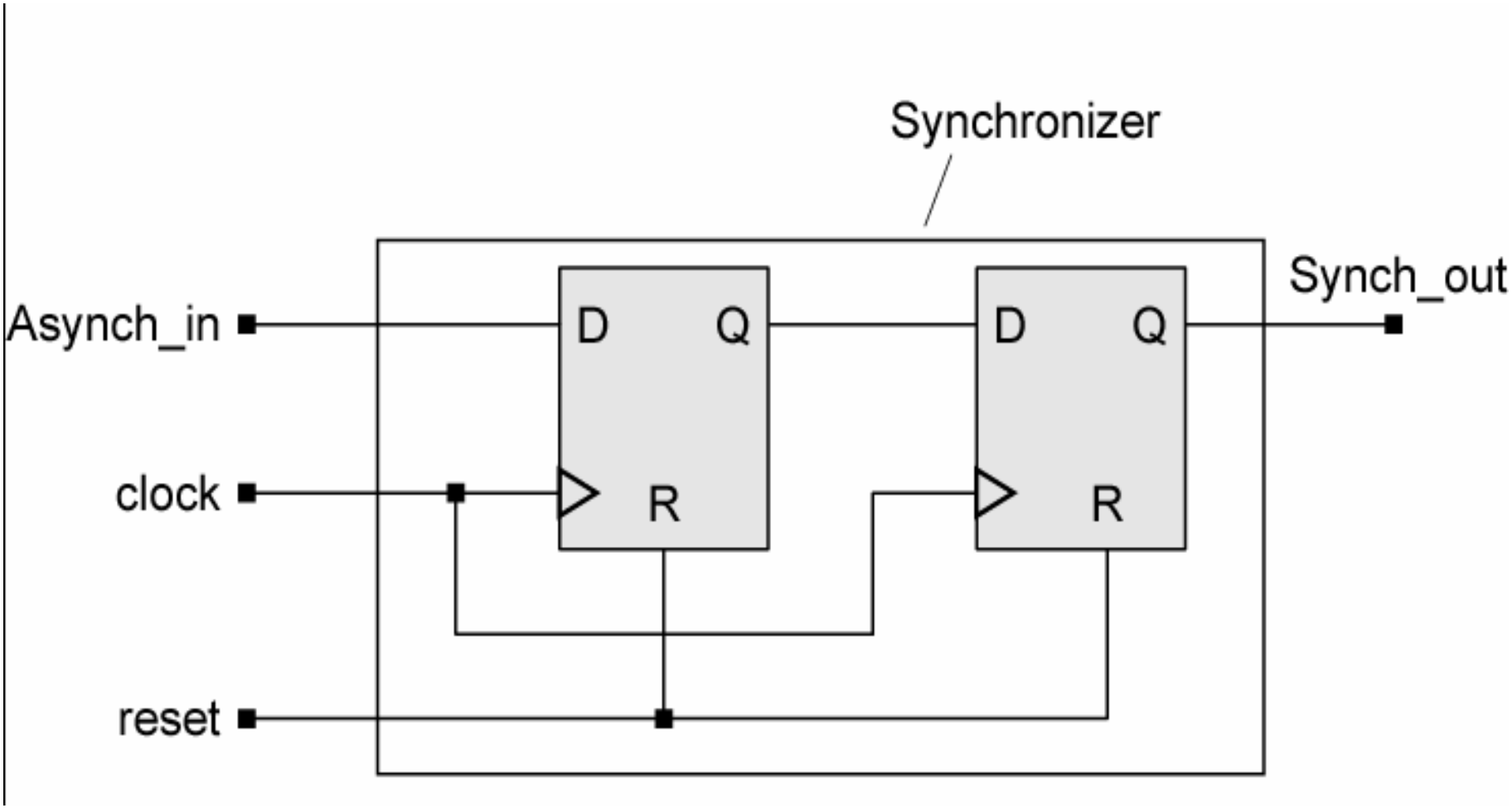
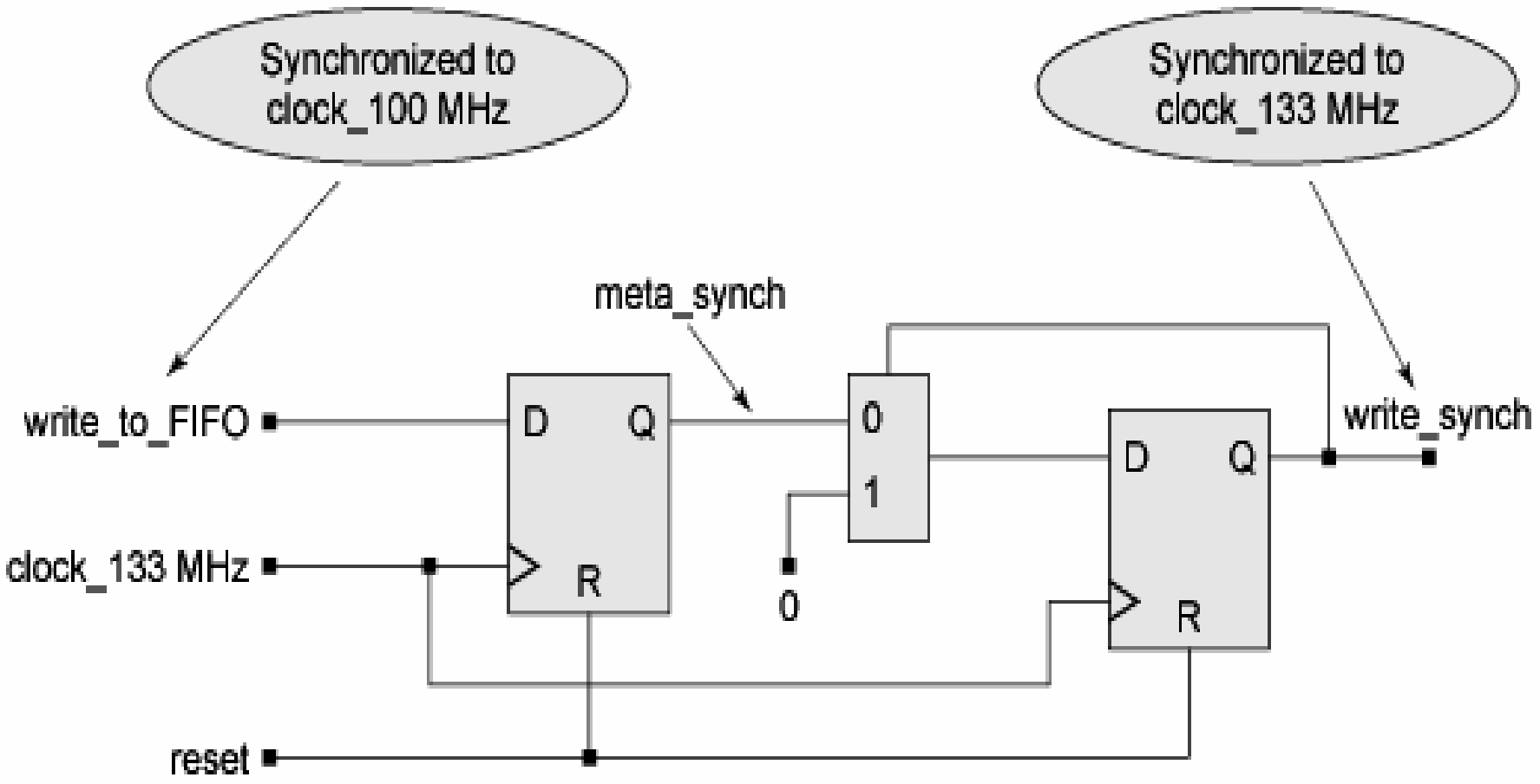


Fig 9.49 Circuit to synchronize the write signal across the clock domain boundary and allow only one write pulse



```
module write_synchronizer (write_synch,
                           write_to_FIFO, clock, reset);
    output     write_synch;
    input      write_to_FIFO;
    input      clock, reset;
    reg        meta_synch, write_synch;
    always @ (negedge clock)
      if (reset == 1)
      begin
        meta_synch <= 0;
        write_synch <= 0;
      end
      else
       begin
        meta_synch <= write_to_FIFO;
        write_synch <= write_synch ? 0: meta_synch;
      end
  endmodule
```

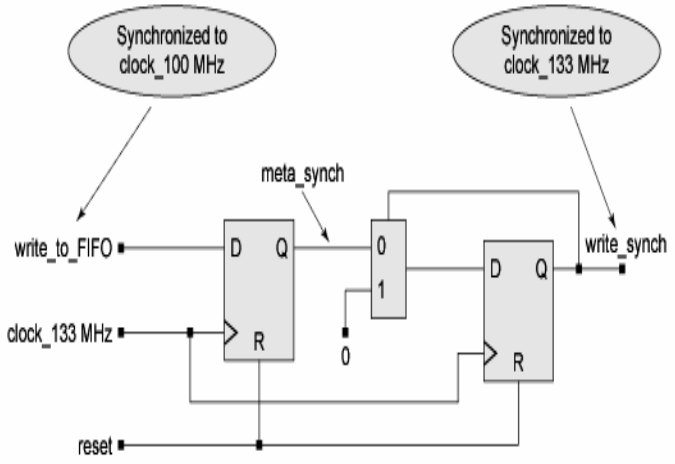
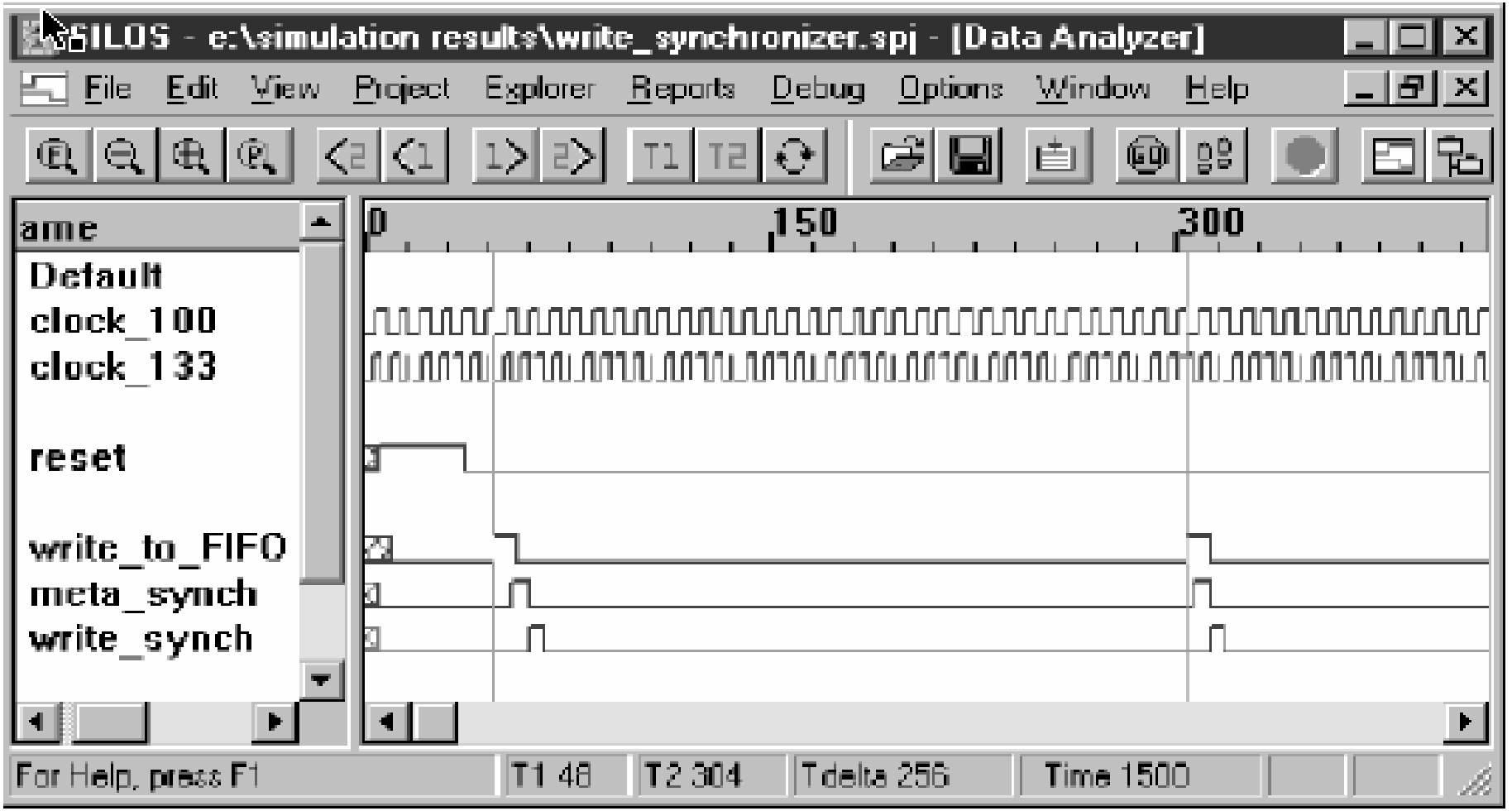


Fig 9.50 Waveform illustrating synchronization of the FIFO write signal across the clock domain



The latency of the synchronizer introduces a design trap

- When write_synch finally asserts, **the contents of the shift register will have already shifted by two more clocks,** destroying coherency between write_synch and the contents of the register
- **To restore coherency**, we can lengthen the shift register datapath by adding buffer registers synchronized to the falling edge of clock_133MHz
- The circuit in Figure 9-51 includes the two-stage synchronizer and the buffer registers

Fig 9.45 A FIFO –Buffered clock domain interface

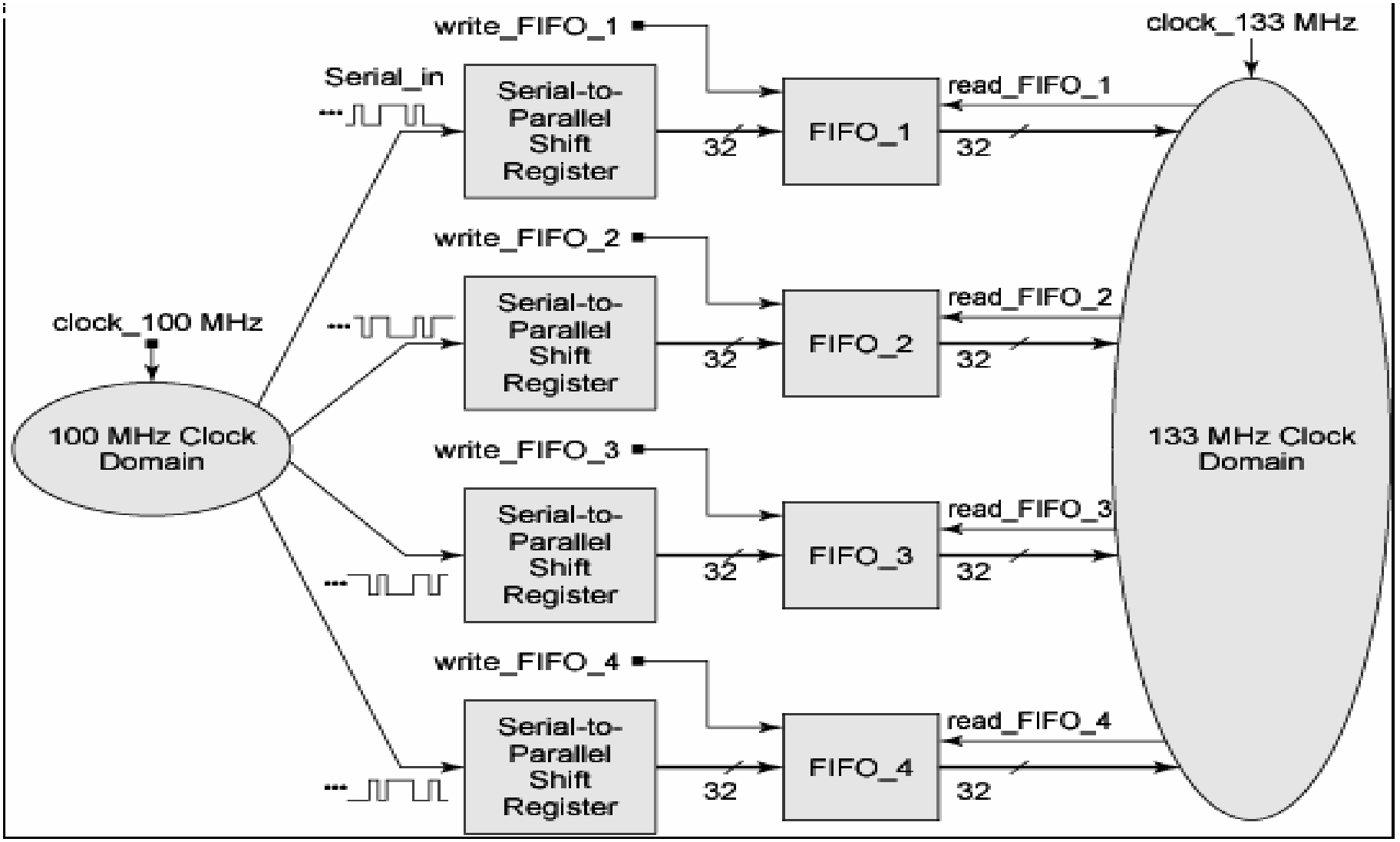
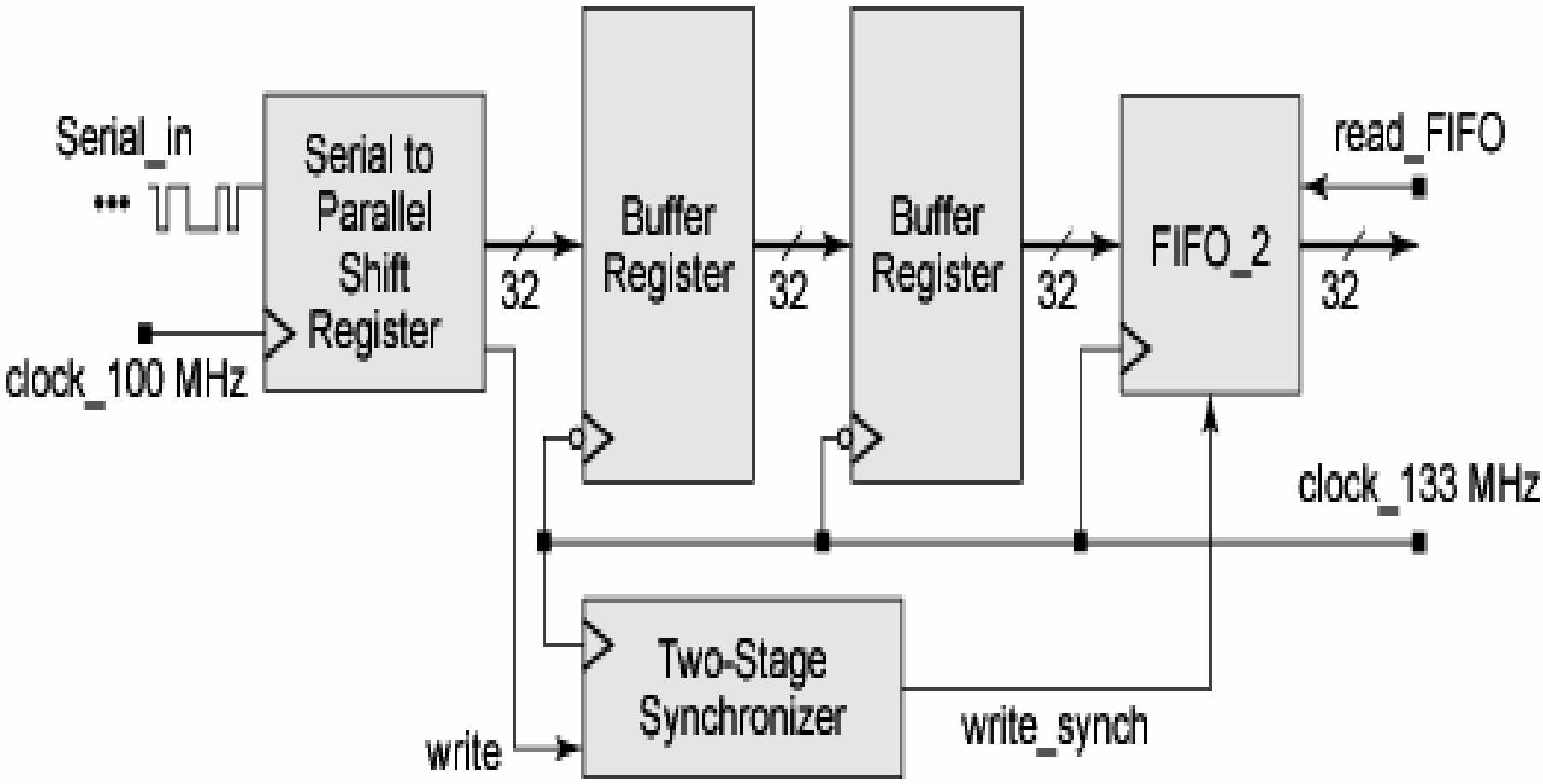


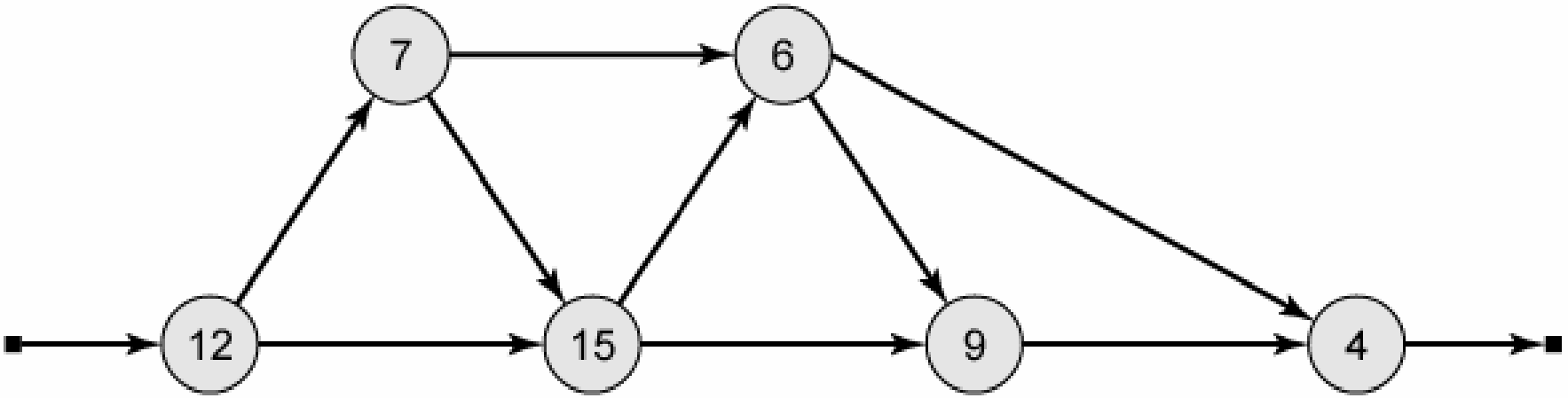
Fig 9.51 The circuit includes the two-stage synchronizer and the buffer registers.



Problem

9-19 The nodes of the DFG shown in Figure P9-19 have been annotated with propagation delays.

Find the optimal placement of pipeline registers in the circuit

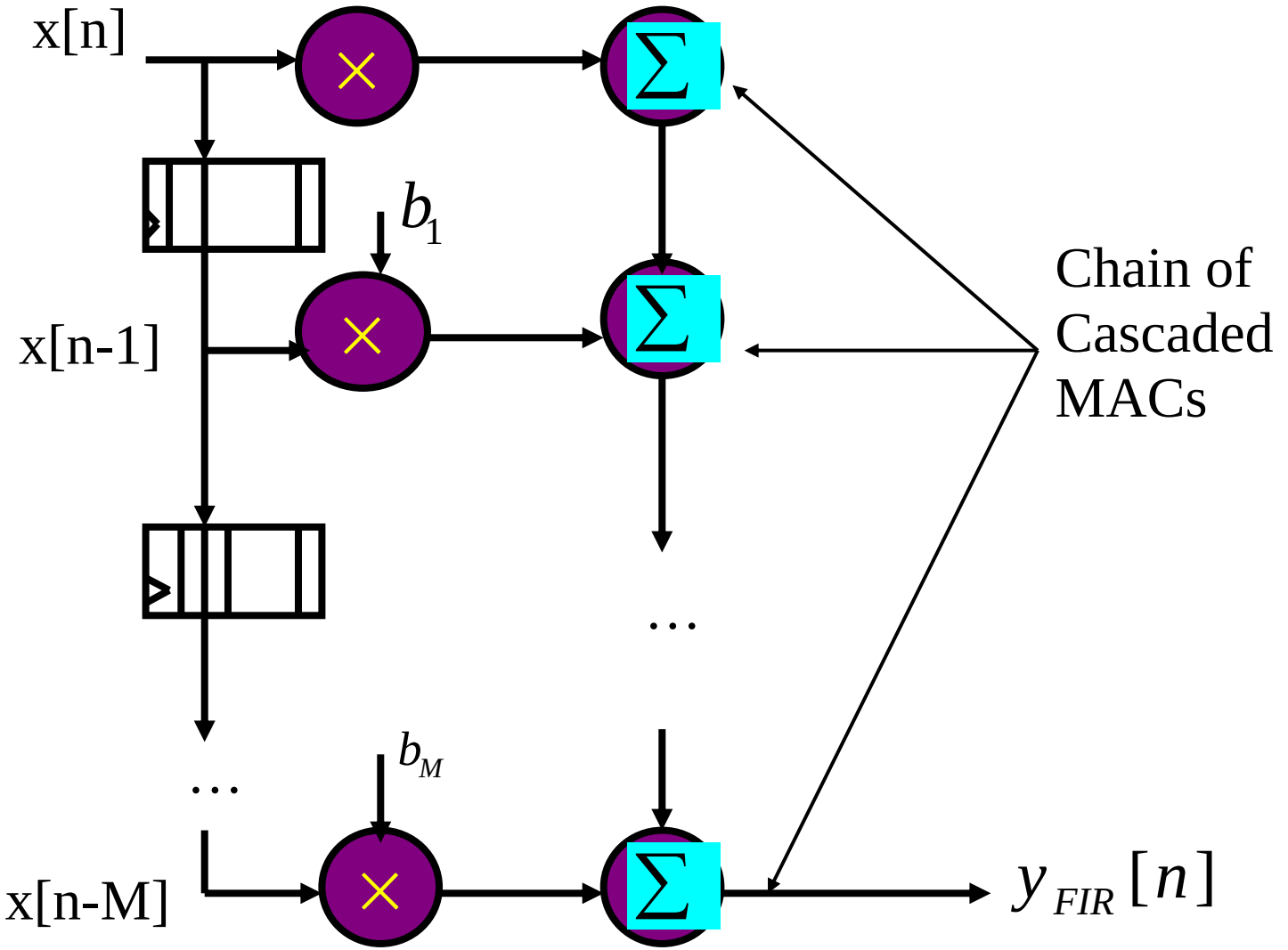


P-15

Implement and compare two different architectures for an eight-tap FIR with a 16-bit datapath.

- The first is to use the architecture shown in Figure 9-23, which has a shift-register structure that stores and shifts the samples of the input sequence.
- The second is to implement the FIR with a circular buffer controlled by a state machine.

Fig 9-23 MAC-based architecture for a type-I,Mth-order FIR filer



P-18. Find a design error in the circuit in following Figure. Redesign the circuit to implement pipelining correctly