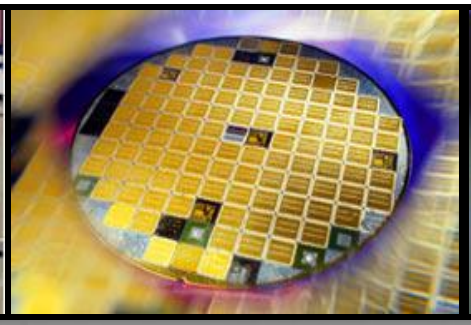
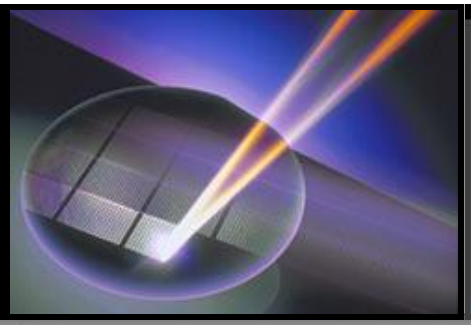
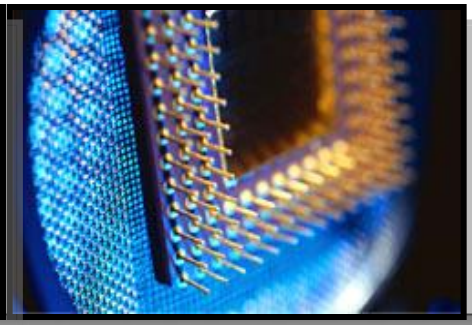
FPGA复杂逻辑设计

FPGA设计的两条思路

- 数据通路
  - 关注算法到结构的映射
- 控制通路
  - 有限状态机的设计

硬件结构划分

系统的算法模型通常具有两大特征：

(1) 含有若干子运算，这些子运算实现对欲处理数据或信息的传输、存储或加工处理。

(2) 具有相应的控制序列，控制子运算按一定的规律有序地进行。

根据算法模型的两大特征，RTL级可将系统硬件分为：

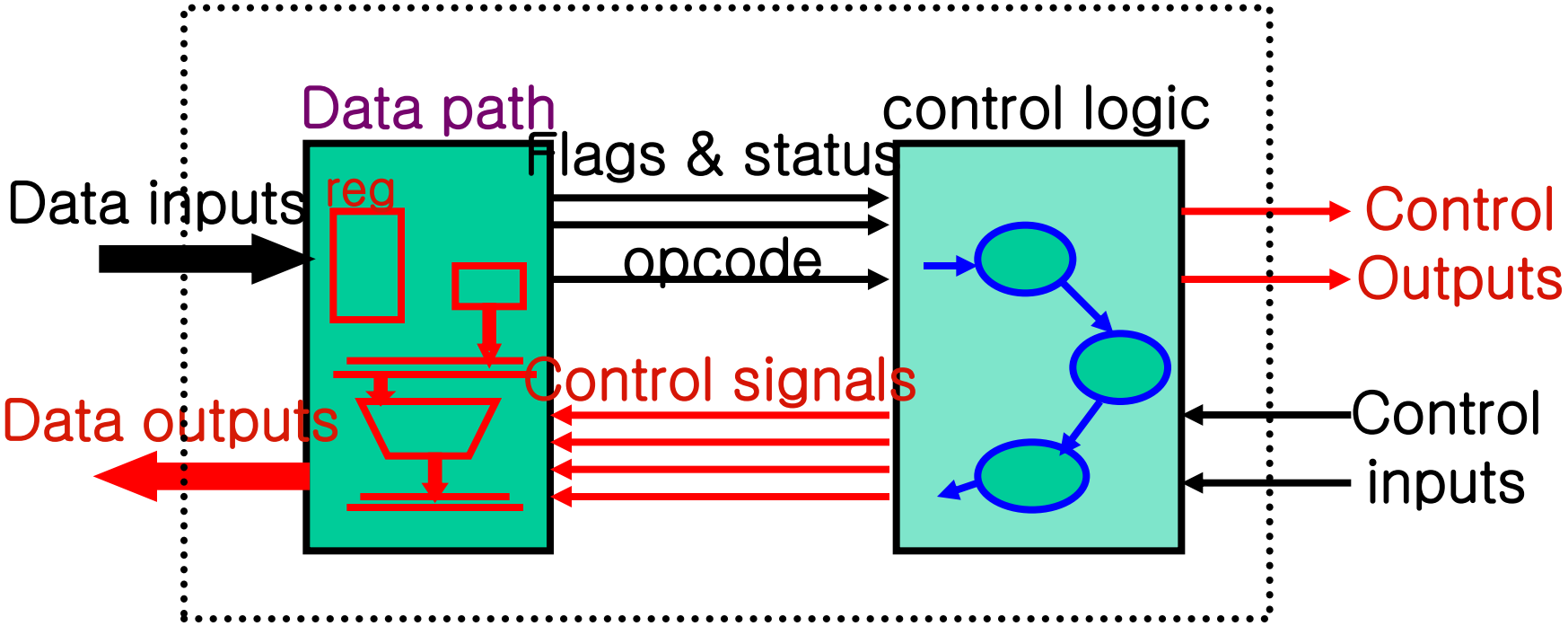
(1) 数据处理单元（Data path），实现算法规定的子运算，即数据的传输、存储或加工处理。

(2) 控制单元（Control logic），实现对数据处理单元运算的次序控制，即产生控制序列。

复杂数字系统的RTL设计思想

- 数字系统（RTL级）设计
  - Data Path
    - 运算单元(组合电路)
    - 存储单元(寄存器)
  - Control Logic
    - 有限状态机(组合部分和时序部分)
    - 微码ROM描述

RTL设计


Data path
control logic
flags & status
Data inputs
reg
Control
Outputs
opcode
Control signals
Data outputs
Control
inputs

数据路径(Data Path)

- 数据路径通常包括算术单元，如ALU、加法器、乘法器、寄存器、移位寄存器及连接它们的总线。它的输入输出来自外部，控制信号也来自并反馈给外部的控制单元。
- 数据路径需要有控制信号来控制时序以及功能。
- 数据路径的总线应设置控制信号，由该信号来选择数据的源以及数据传送的路径。数据部分还应有输出信号，将数据的标志和状态通知控制部分。

控制逻辑(Control Logic)

- 控制逻辑：又称控制单元，用来产生控制信号序列，以决定何时进行何种数据运算。控制单元要从数据单元及外界得到条件信号，以决定继续进行那些数据运算。数据单元要产生输出信号、数据运算状态等有用信息。
- 控制部分通常由状态机构成，通过判断当前的数据的状态来决定应该对这个数据什么操作，应该把这个数据送到数据路径的什么部分。
- 控制逻辑可由算法状态机（ASM：Algorithm State Machine）图来描述。

RTL设计

复杂系统的RTL设计关键在于数据路径和控制单元的划分，可遵循以下步骤：

（1）由算法流程图明确数据路径的操作，并为每个操作分配一个控制信号。

（2）确定控制信号时序，在明确各控制信号的基础上，对他们进行排序，列出控制信号排序表，从而归纳并确定控制信号的时序，作为对控制单元设计的技术要求，使系统正确执行算法流程。

（3）由算法流程图导出ASM图，结合数据路径构成ASMD图，同时明确数据路径到控制逻辑的反馈信号。

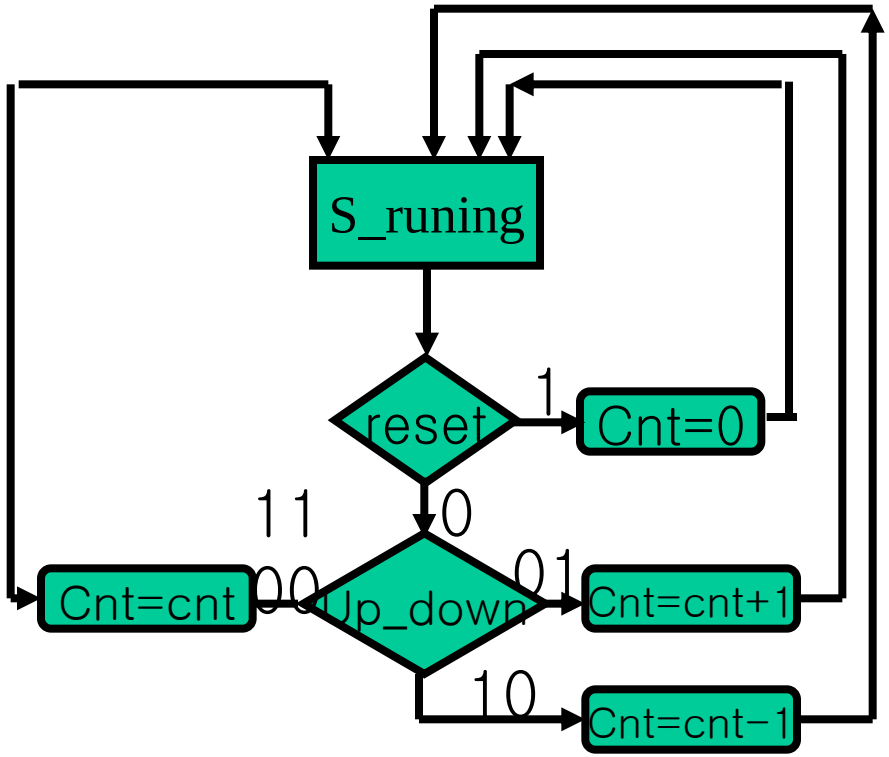
（4）用HDL语言分别设计数据路径和控制逻辑，构成整个数字系统。

算法状态机数据路径图（ASMD）

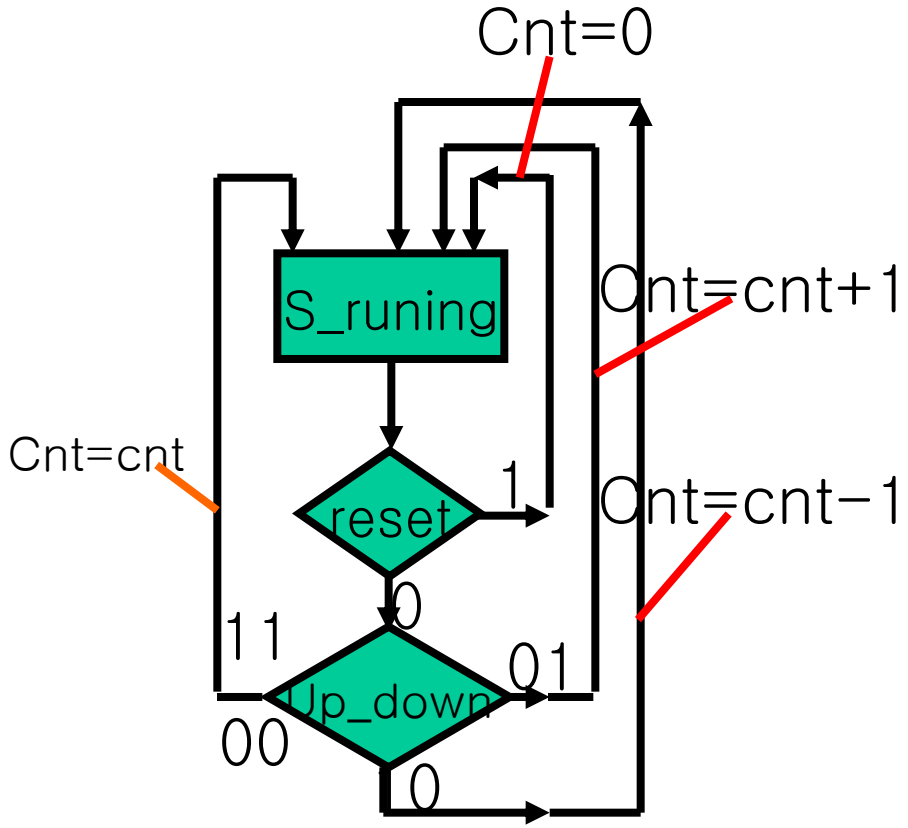
数据路径通常用数据流图描述，而控制逻辑则常用ASM图描述，对于以控制为主的数字系统，直接用ASM图描述即可。对于数据为主的系统，修改ASM图，将其链接到控制单元所控制的数据路径上，当ASM图状态沿着通道发生转移时，通过标注每个路径来指出那些在相关数据路径中所发生的并发寄存器操作。

以这种方式连接到数据路径的ASM图，称为算法状态机数据路径（ASMD）图。这是一种表述硬件设计的通用模型。

例.一可逆的3位二进制计数器，在一个二位二进制输入字的控制下选择递增、递减或保持计数，且有同步复位功能。



ASM图

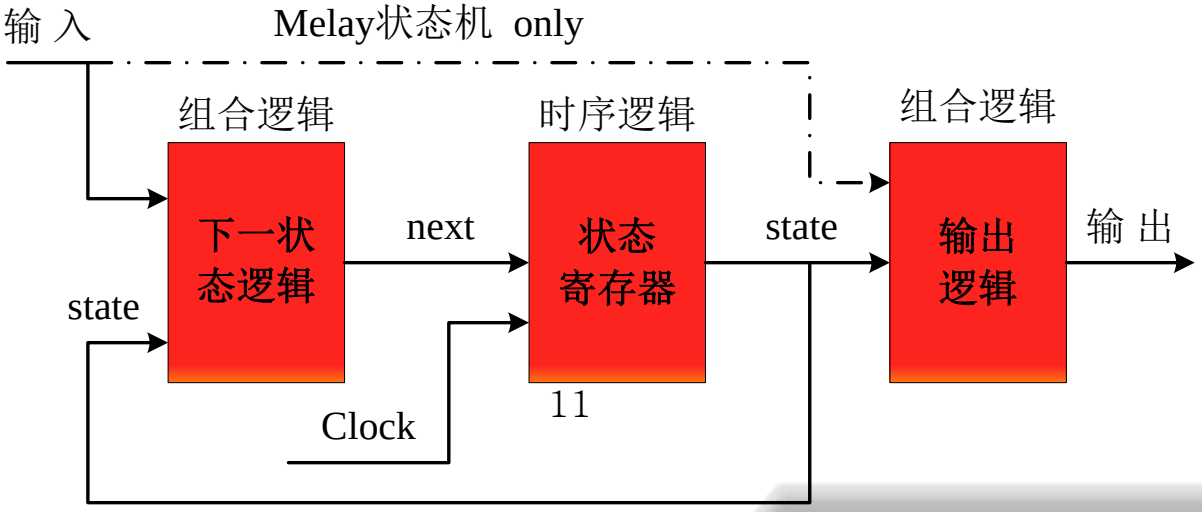


ASMD图

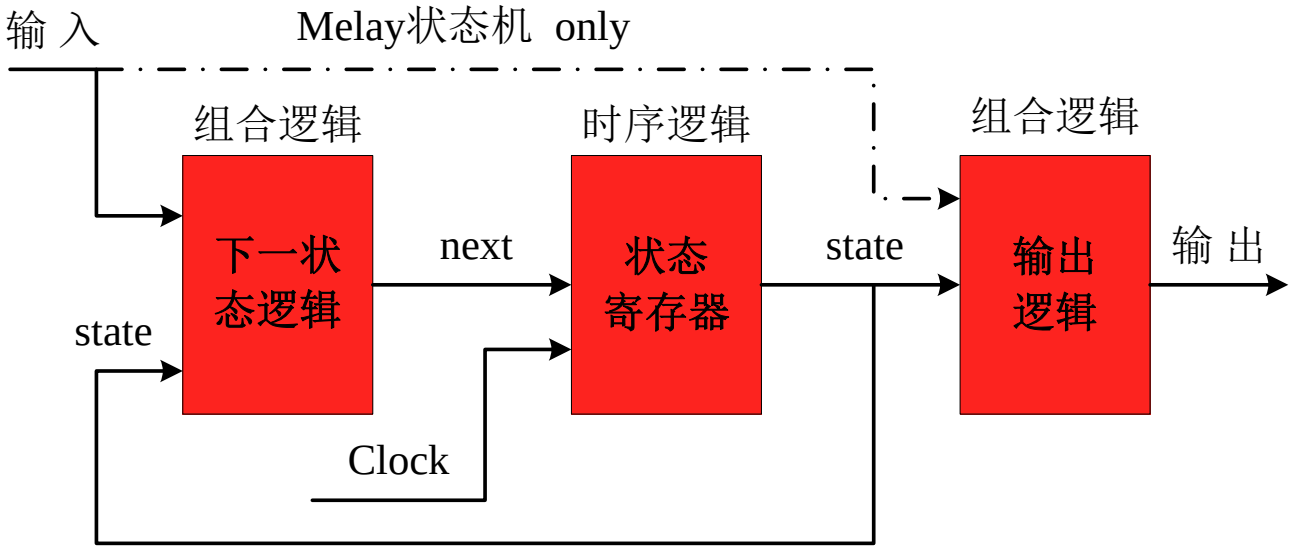
ASMD图有利于在将数据路径设计和控制逻辑设计分离开，并在两个单元之间保持清晰联系。

同步状态机的概念

- 同步状态机由下一状态逻辑，状态寄存器和输出逻辑三个部分组成。
- 驱动方程决定了状态机的下一个状态，驱动方程是输入信号和当前状态的组合函数；
- 状态寄存器由一组触发器组成，用来记忆当前状态；
- 状态机的输出是由输出函数得到，它也是当前状态和输入信号的函数；



同步状态机的分类



Moore状态机：输出仅与当前状态有关

Melay状态机：输出与当前状态和输入信号都有关

状态机设计

- RTL级状态机性能评判
  - 稳定性好
  - FSM速度快，满足设计频率要求
  - 面积小
  - FSM设计清晰易懂、易维护

状态机设计

- 状态机编码
  - 为了使状态机能够在多个状态中切换，必须对每个状态进行有效的编码
  - 常用的编码方式有

    binary code 二进制编码

    one hot code 独热码

    gray code 格雷码

状态机设计

- 状态机设计通常有三段式的和两段式的方法，一段式由于可读性太差，不推荐使用
- 两段式
  - 第一个always块用于状态的更迭（时序）
  - 第二个always块用于产生新的状态以及当前状态对应的信号输出（组合）

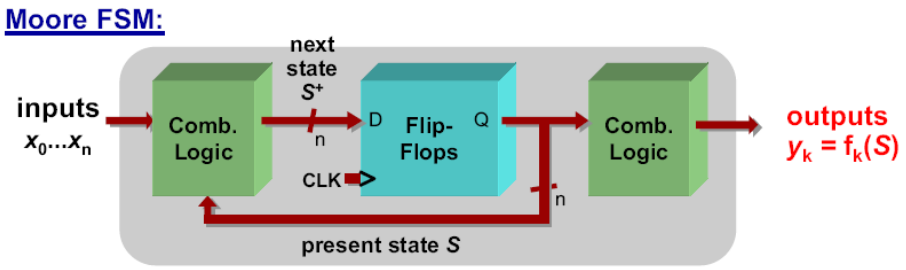
状态机设计

- 三段式
  - 第一个always块用于状态的更迭（时序）
  - 第二个always块用于产生新的状态（组合）
  - 第三个always块用于同步产生相应的输出信号（时序）

Moore状态机的Verilog实现

```
`timescale 1ns/100ps
module state4 (clock, reset, out);
    input reset, clock;
    output [1: 0] out;
    reg [1: 0] out;
    parameter //状态变量枚举
        stateA = 4 ‘b0000, stateB = 4’ b0001…
    reg [3: 0] state, nextstate;
   //定义时序逻辑
    always @( posedge clock)
        if (reset)  //同步复位
            state <= stateA;
        else
```

```
always @( state) // 定义下一状态的组合逻辑
    case (state)
        stateA: begin
            nextstate = stateB;
            out = 2 ‘b00; // 输出决定于当前状态
            end
        ……
        stateD: begin
            nextstate = stateA;
            out = 2'b00;
            end
    endcase
endmodule
```



总结：2个并行模块

1) always block:
   下一状态的组合逻辑

2) always block:
   更新状态的时序逻辑

Mealy状态机的Verilog实现

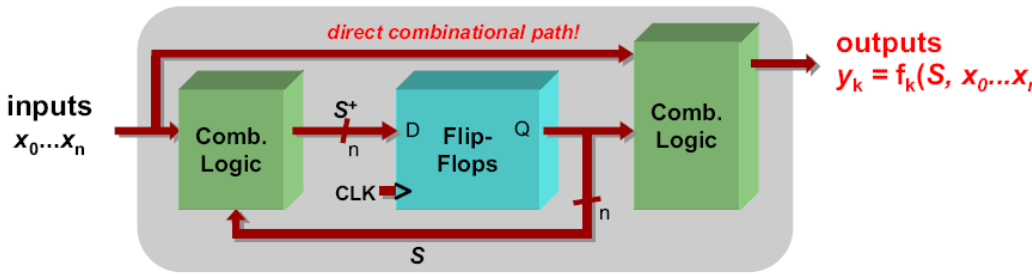
```
module FSM_name (Clock, Resetn, input_signal, output_signal);
    input Clock, Resetn, input_signal;
    output output_signal;
    reg [3:0] state_present, STATE_NEXT;
    parameter [3:0] STATE1 = 4'b0000, STATE2 = 4'b0001…. ;

    // Define the next state combinational circuit and outputs
    always @(input_signal or state_present)
      case (state_present)
          STATE1: if (input_signal) define output and next st
    ate;
                  else define output and next state;
          STATE2: if (input_signal) define output and next st
    ate;
                  else define output and next state;
           ……
          default: define output and next state;
    endcase

    // Define the sequential block
    always @(posedge Clock)
      if (Resetn == 0) state_present <= STATE1;
      else state_present <= STATE_NEXT;
endmodule
```

总结：2个并行模块

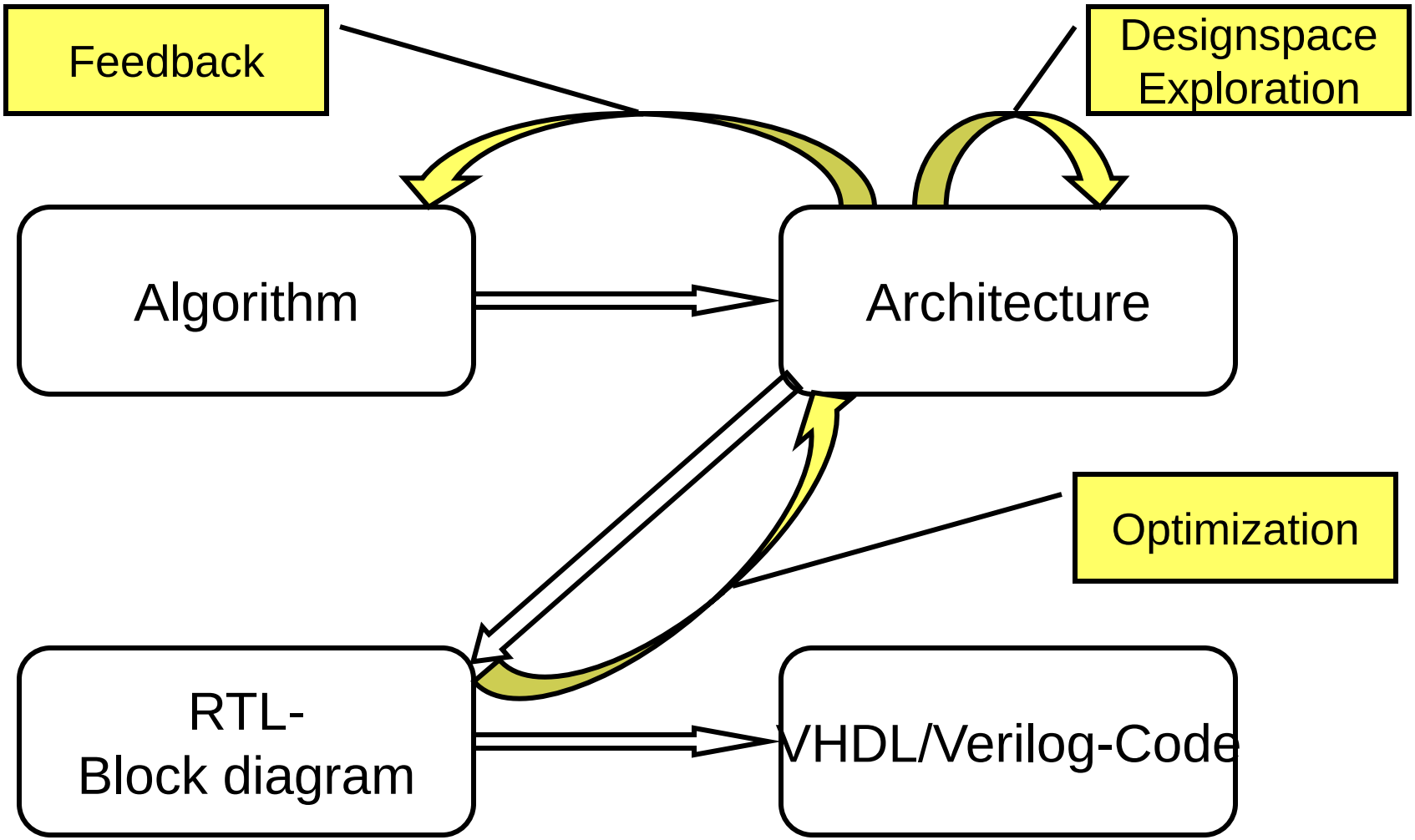
1) always block:
   下一状态的组合逻辑和输出

2) always block:
   更新状态的时序逻辑

Mealy FSM:



Where to start?



算法⇔结构

- 算法的好坏决定了基本结构与复杂度
- 结构向算法提供反馈，以发现主要问题，改善甚至更改算法
- 行为级综合与高层次综合提供了算法到结构的转换方法。
- RTL设计 => 算法设计

算法到结构的映射

- 算法：解决问题的方法和步骤。
- 计算理论证明每一个有效算法都可以由一个图灵机，通常即一个FSMD（Finite State Machine and Data Path）实现。
- 常规算法到结构的映射，就是算法到FSMD的映射，也就是控制部分和数据部分的设计。
- 核心设计就是：
  - 时间设计：Scheduling
  - 空间设计：Binding

设计约束

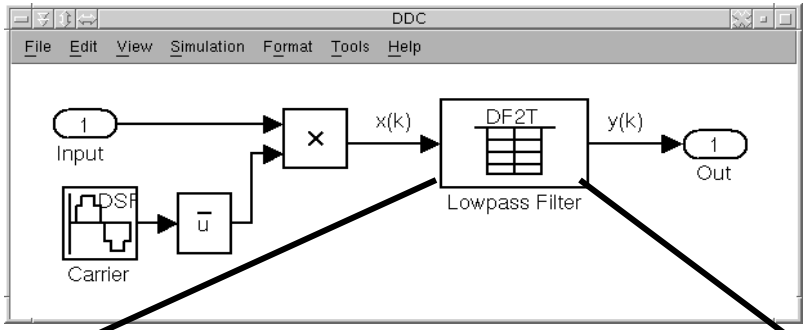
- 资源
  - 处理资源
  - 存储资源
  - 接口资源
- 时序
  - Cycle Time
  - Latency
  - Throughput
- 功耗
  - 静态功率
  - 动态功率
  - 峰值功率
- …

性能估计

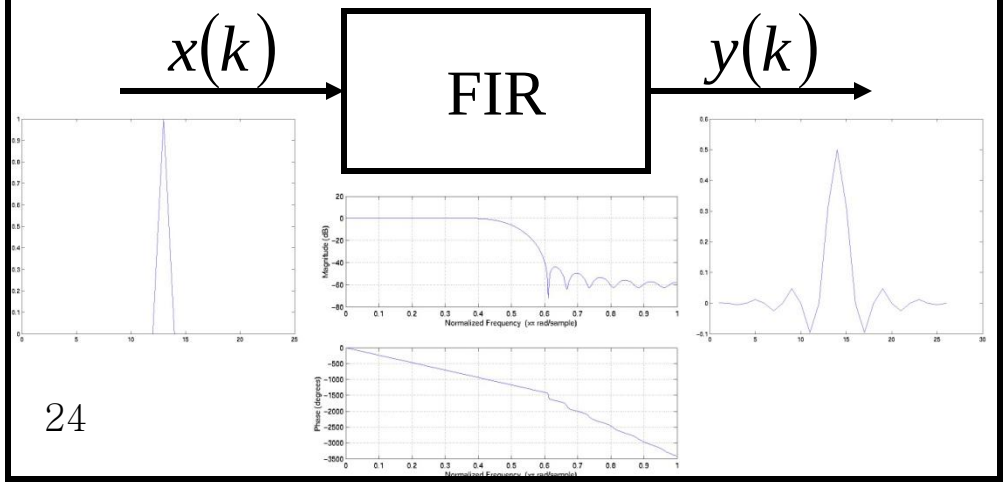
- 资源
  - 处理运算逻辑
  - 寄存器
  - 导引逻辑(Steering Logic)：multiplexers, buses
  - 布线
  - 控制逻辑
- 时序
  - 逻辑级数复杂度
  - 节拍个数
  - 流水线

Algorithm

- High-Level System Diagram
  - Context of the design
    - Inputs and Outputs
    - Throughput/rates
    - Algorithmic requirements
- Algorithm Description
  - Mathematical Description
  - Performance Criteria
    - Accuracy
    - Optimization constraints
  - Implementation constraints
    - Area
    - Speed

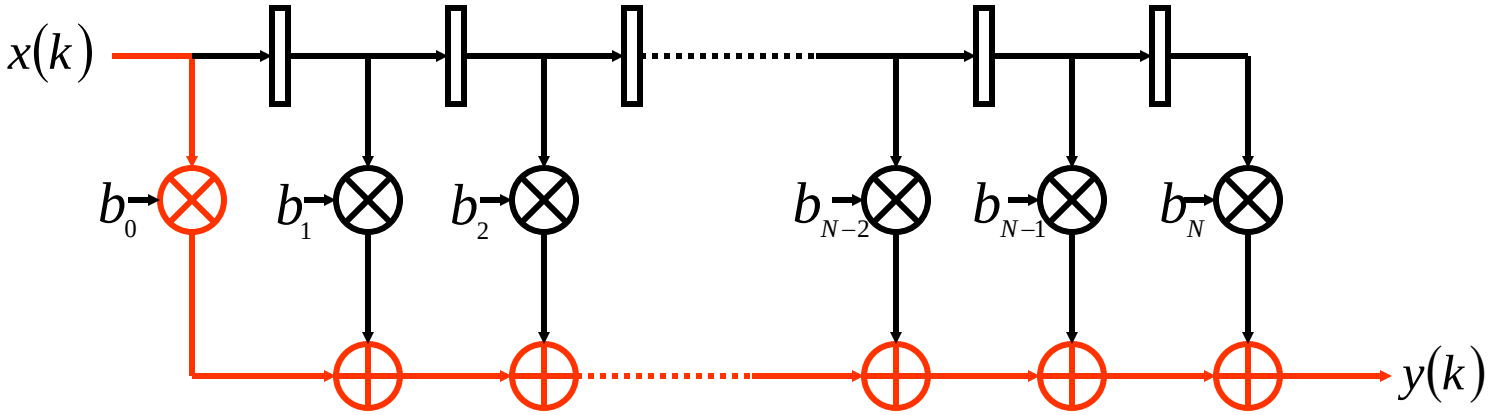


$$y(k) = \sum_{i=0}^{N} b_i x(k-i)$$



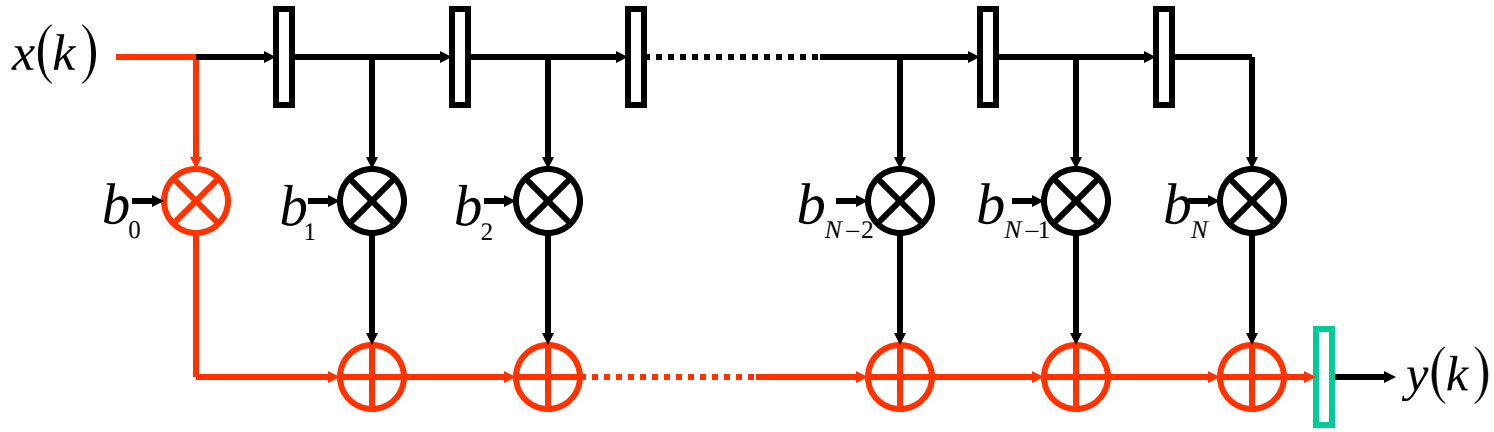
Architecture (1)

- 直接形式：



Architecture (2)

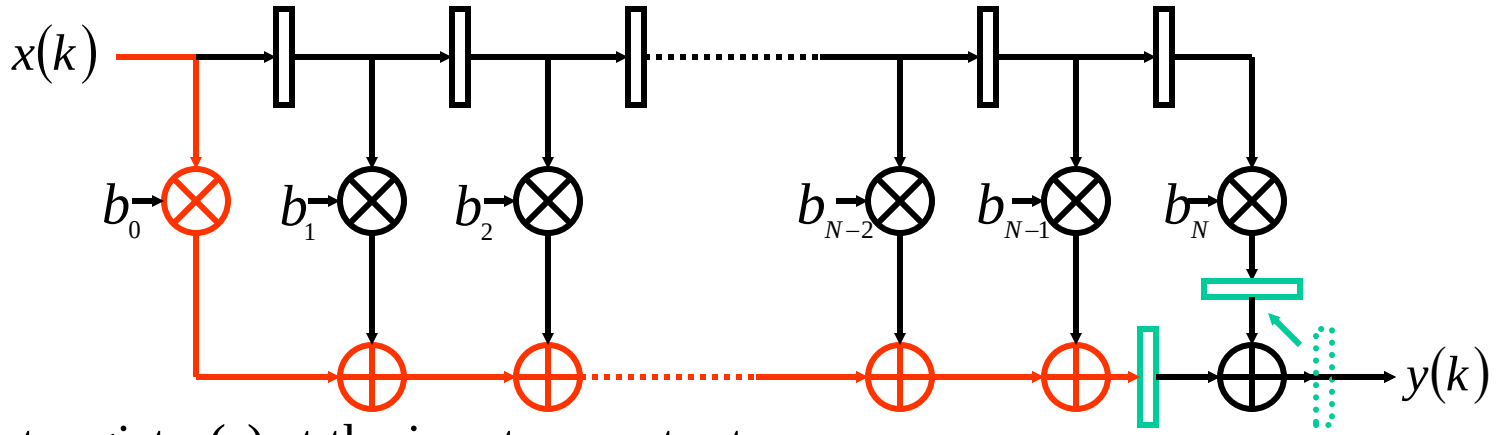
- Pipelining/Retiming:
  - Improve timing



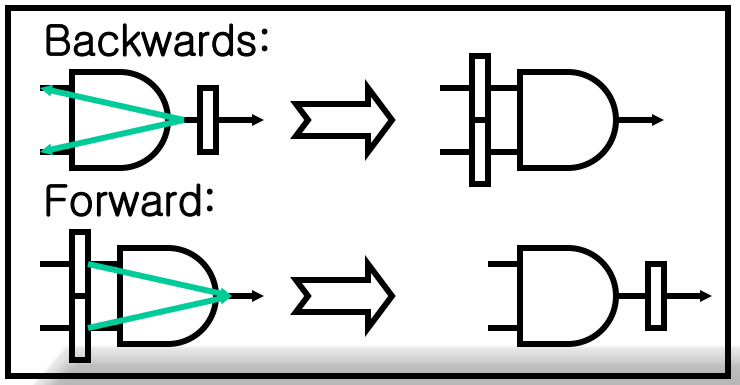
  - Insert register(s) at the inputs or outputs
    - Increases Latency

Architecture (2)

- Pipelining/Retiming:
  - Improve timing

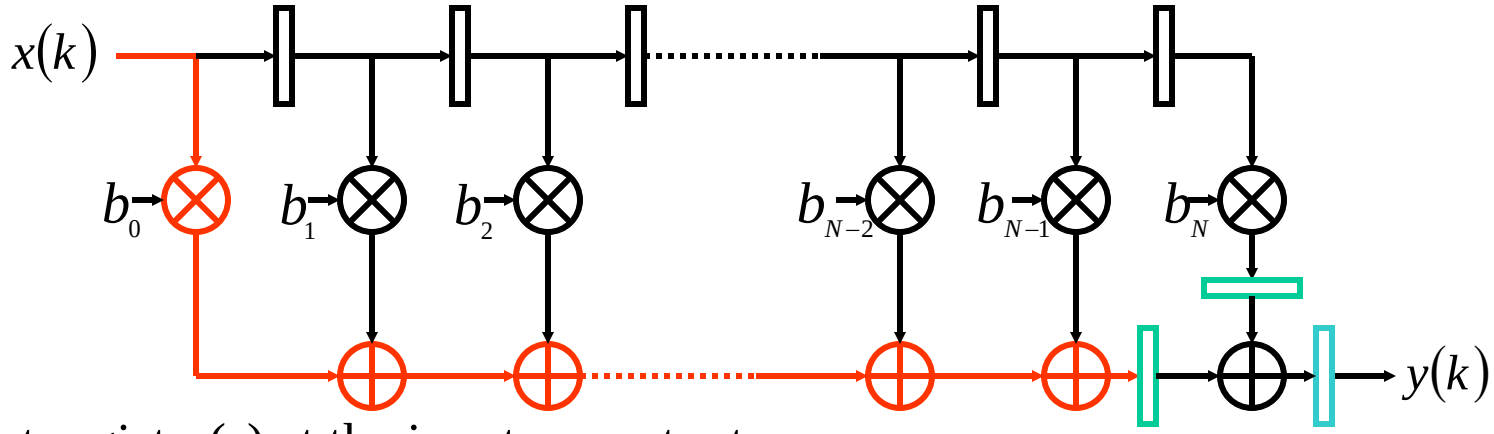


- Insert register(s) at the inputs or outputs
  - Increases Latency
- Perform Retiming:
  - Move registers through the logic without changing functionality

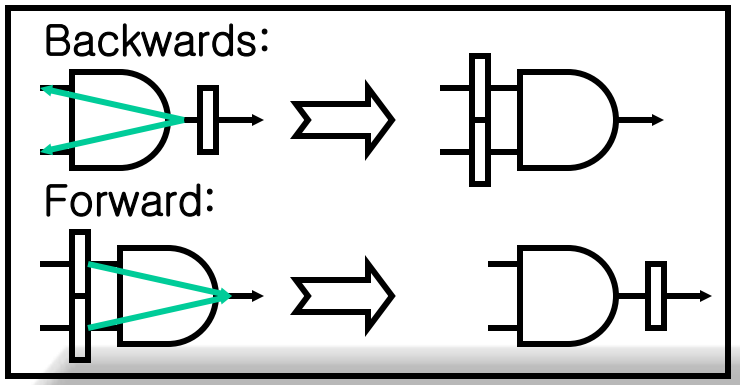


Architecture (2)

- Pipelining/Retiming:
  - Improve timing

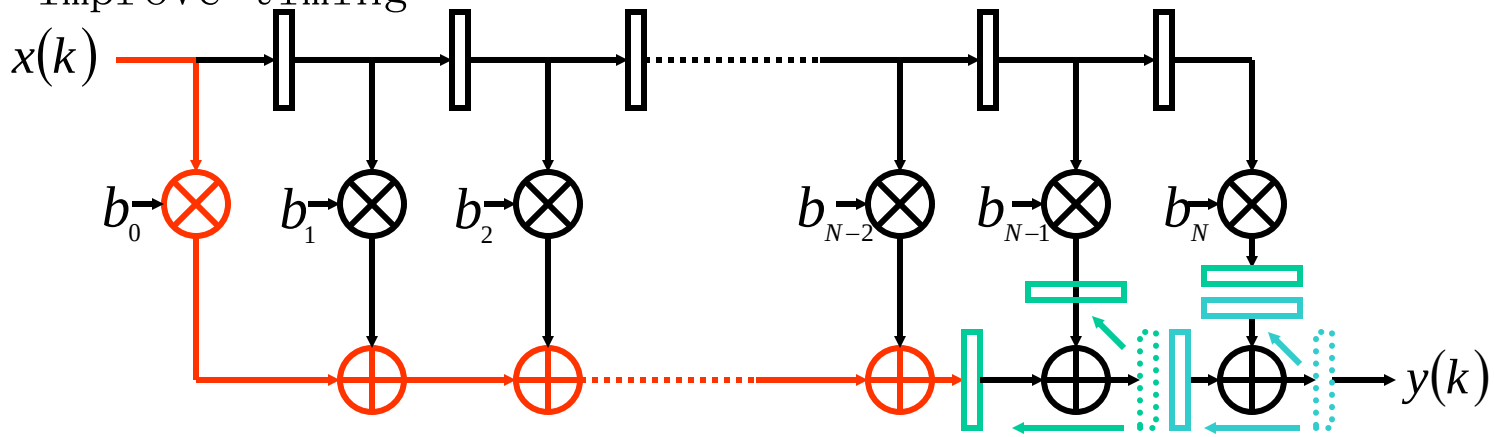


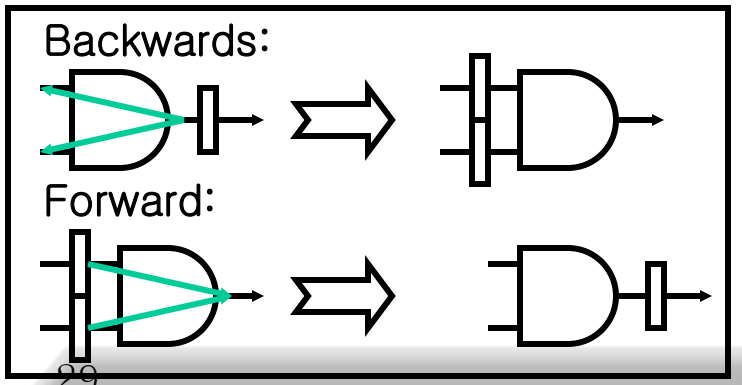
- Insert register(s) at the inputs or outputs
  - Increases Latency
- Perform Retiming:
  - Move registers through the logic without changing functionality



Architecture (2)

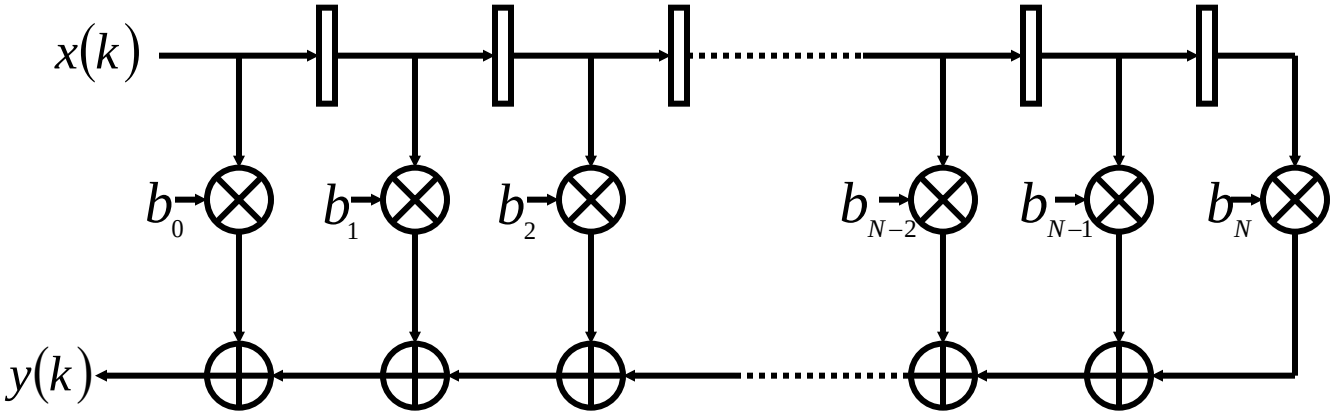
- Pipelining/Retiming:
  - Improve timing





Architecture (3)

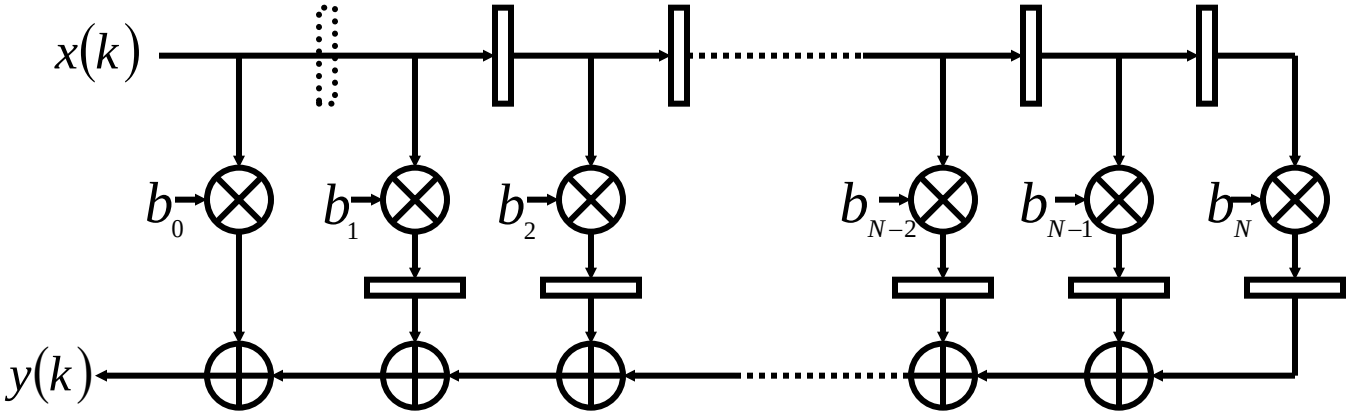
- Retiming and simple transformation:
  - Optimization



  - Reverse the adder chain

Architecture (3)

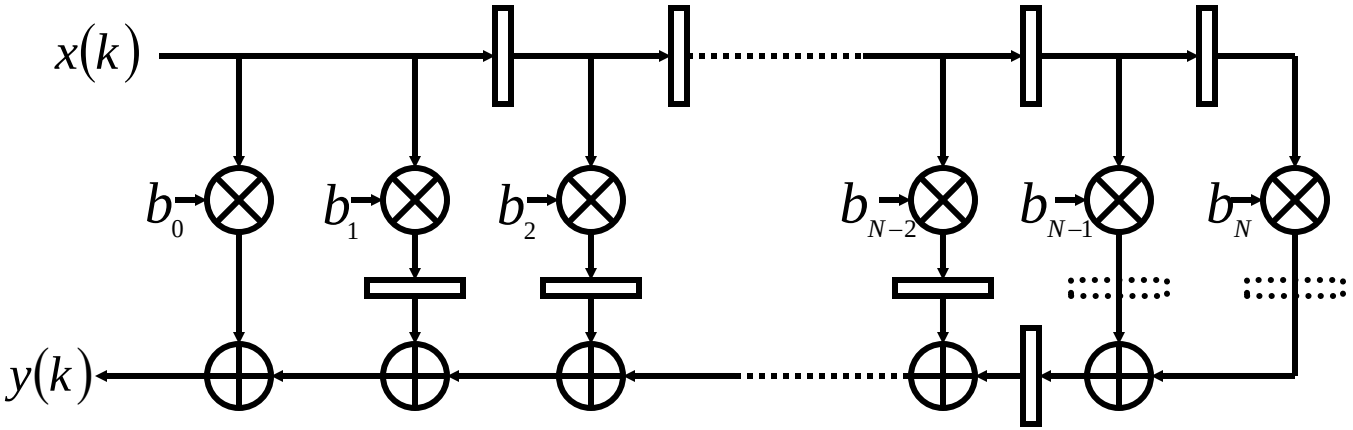
- Retiming and simple transformation:
  - Optimization



- Reverse the adder chain
- Perform Retiming

Architecture (3)

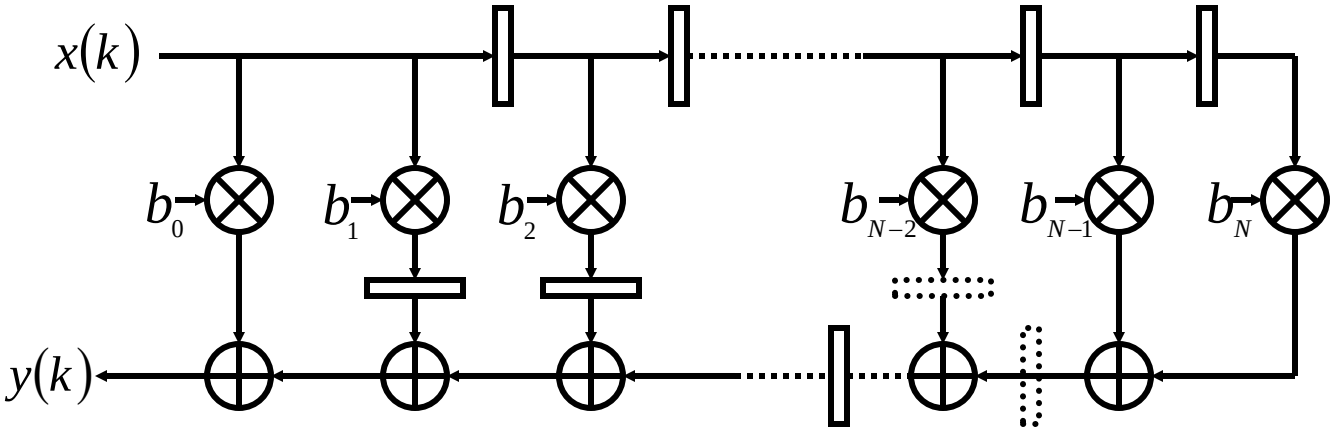
- Retiming and simple transformation:
  - Optimization



- Reverse the adder chain
- Perform Retiming

Architecture (3)

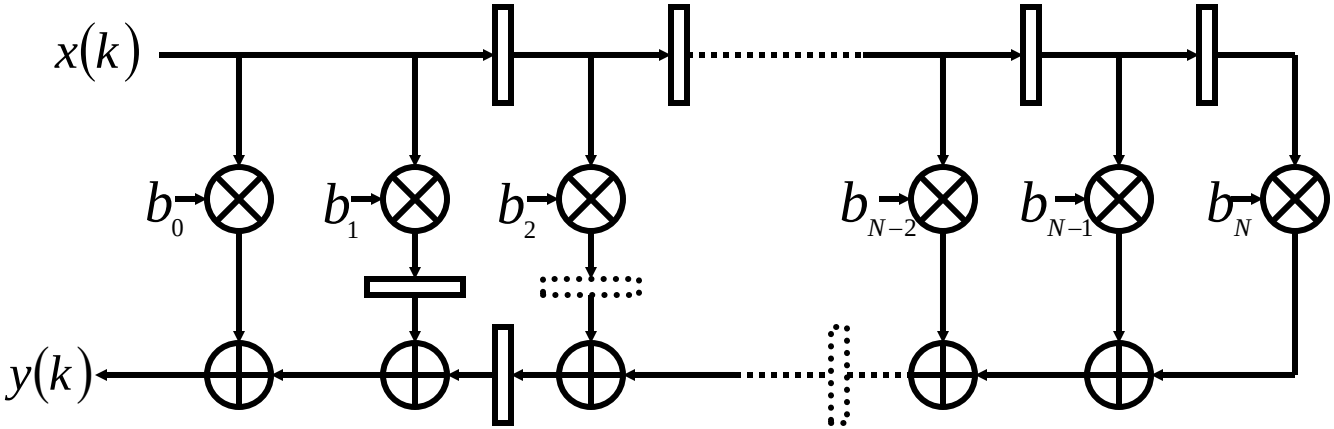
- Retiming and simple transformation:
  - Optimization



- Reverse the adder chain
- Perform Retiming

Architecture (3)

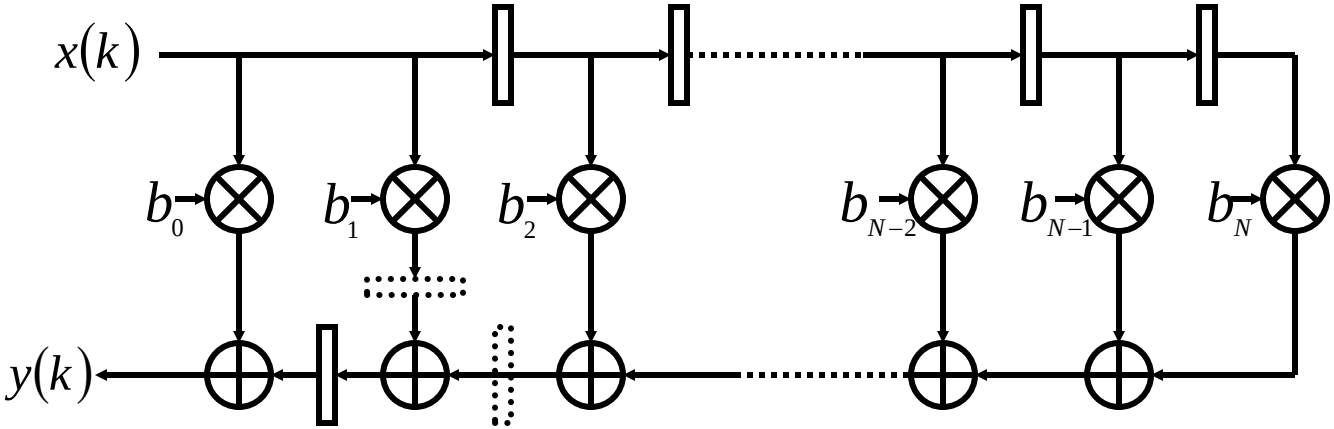
- Retiming and simple transformation:
  - Optimization



- Reverse the adder chain
- Perform Retiming

Architecture (3)

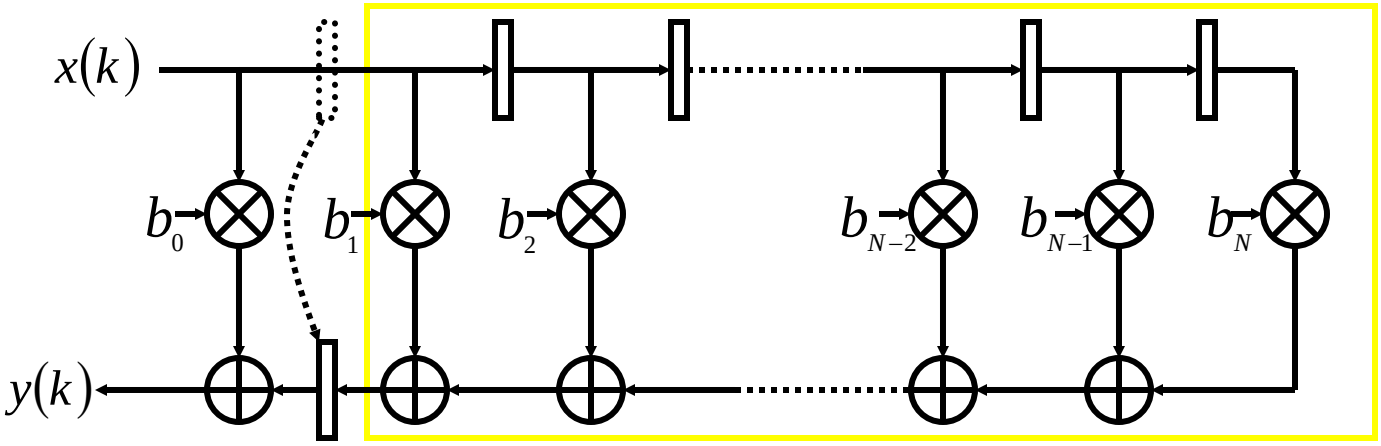
- Retiming and simple transformation:
  - Optimization



- Reverse the adder chain
- Perform Retiming

Architecture (3)

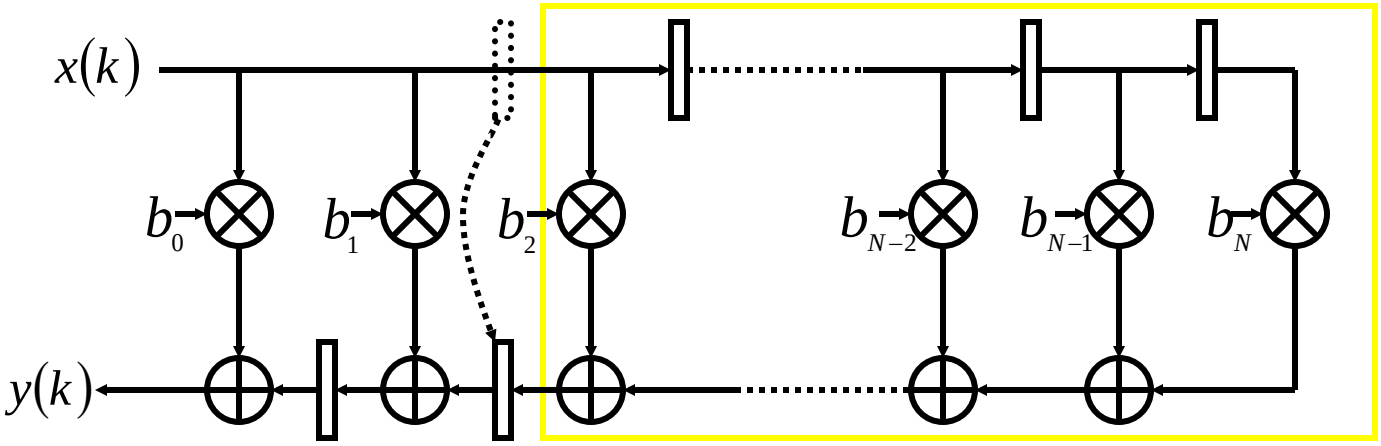
- Retiming and simple transformation:
  - Optimization



- Reverse the adder chain
- Perform Retiming

Architecture (3)

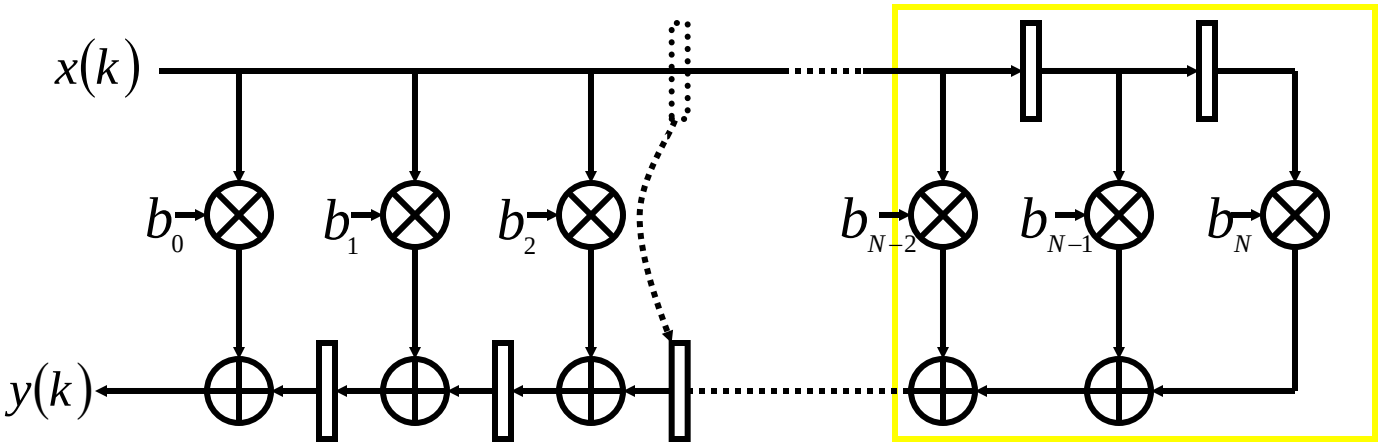
- Retiming and simple transformation:
  - Optimization



- Reverse the adder chain
- Perform Retiming

Architecture (3)

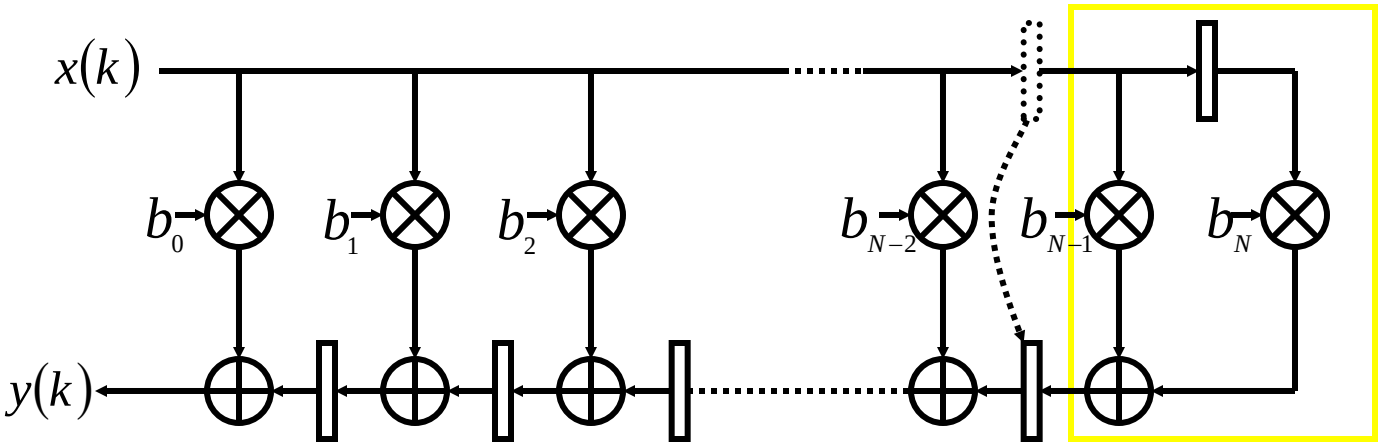
- Retiming and simple transformation:
  - Optimization



  - Reverse the adder chain
  - Perform Retiming

Architecture (3)

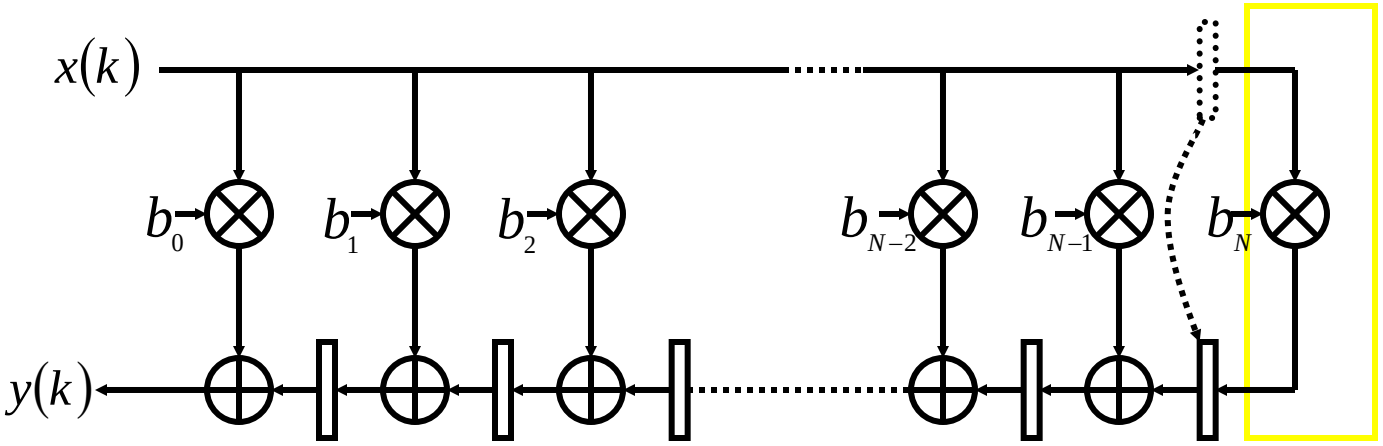
- Retiming and simple transformation:
  - Optimization



- Reverse the adder chain
- Perform Retiming

Architecture (3)

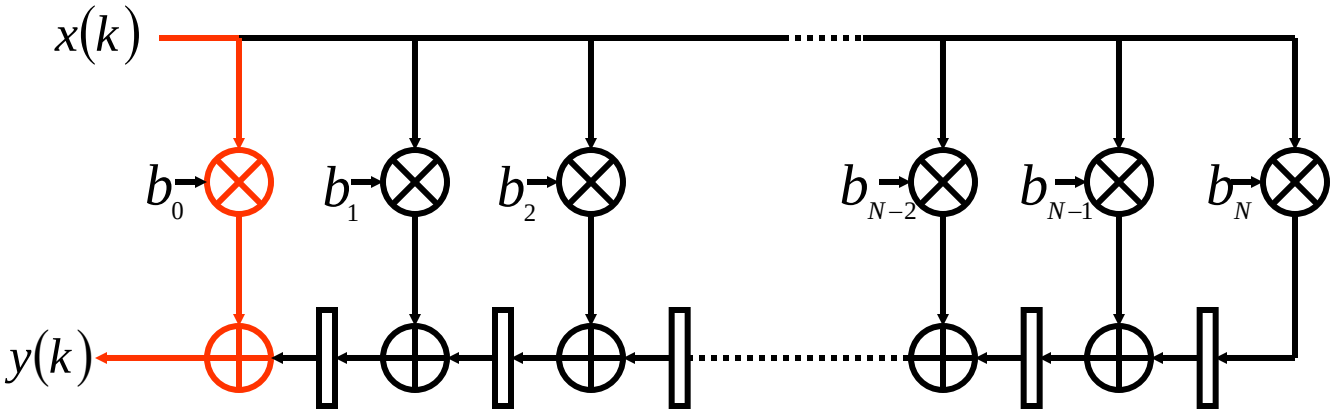
- Retiming and simple transformation:
  - Optimization



- Reverse the adder chain
- Perform Retiming

Architecture (3)

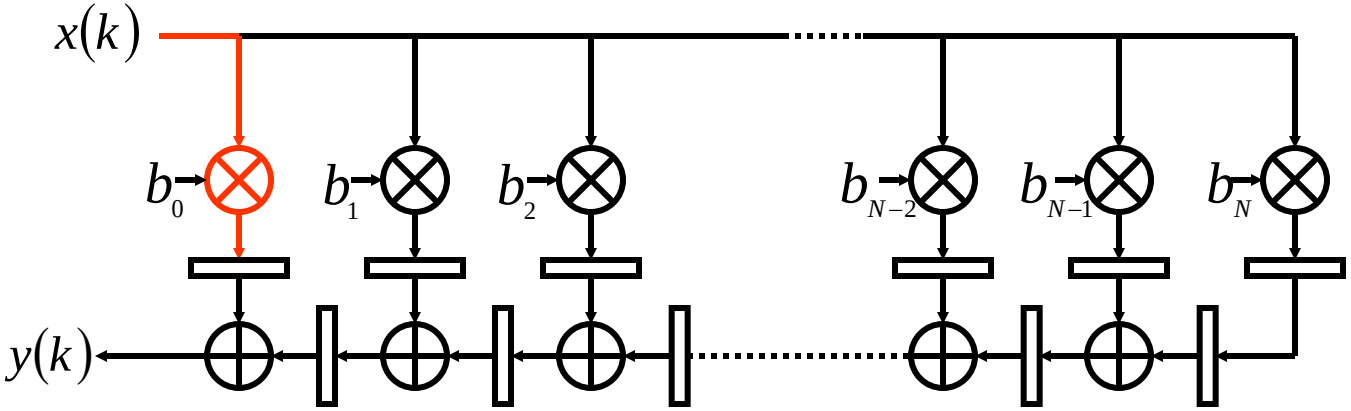
- Retiming and simple transformation:
  - Optimization



- Reverse the adder chain
- Perform Retiming

Architecture (4)

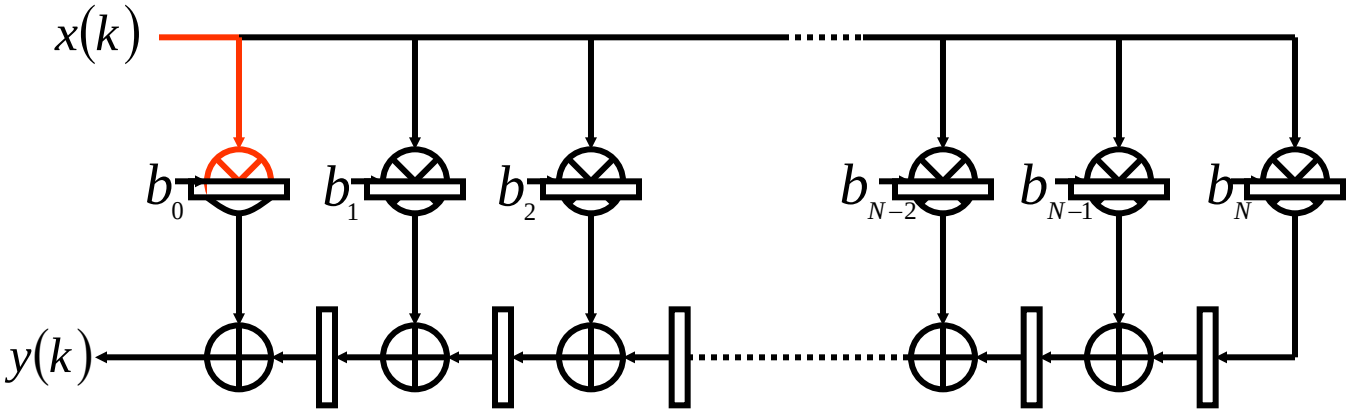
- More pipelining:
  - Add one pipelining stage to the retimed circuit



  - The longest path is given by the multiplier
    - Unbalanced: The delay from input to the first p ipeline stage is much longer than the delay from the first to th e second stage

Architecture (5)

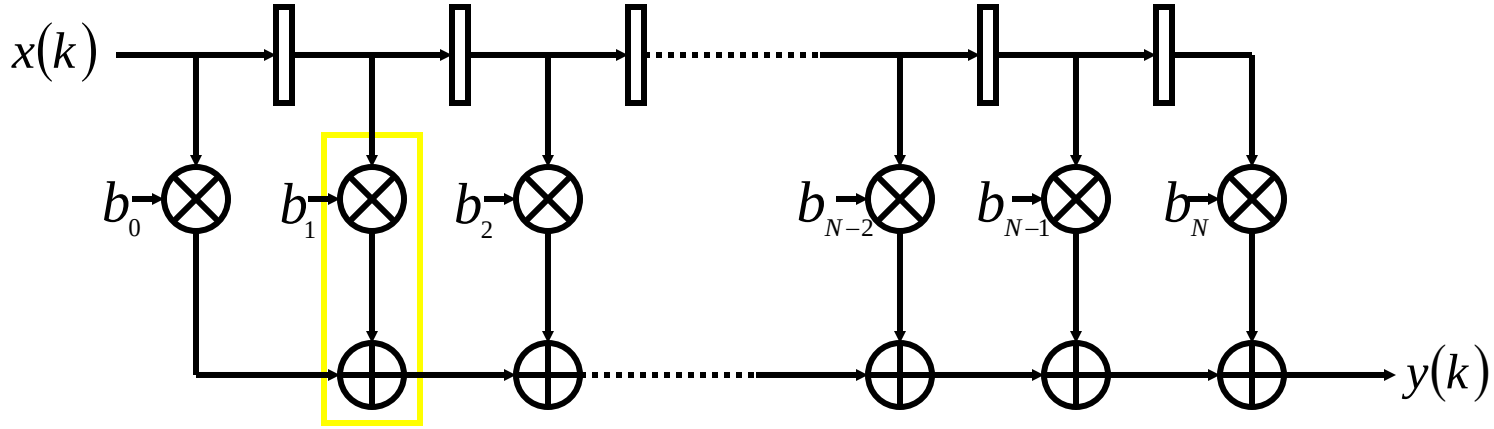
- More pipelining:
  - Add one pipelining stage to the retimed circuit



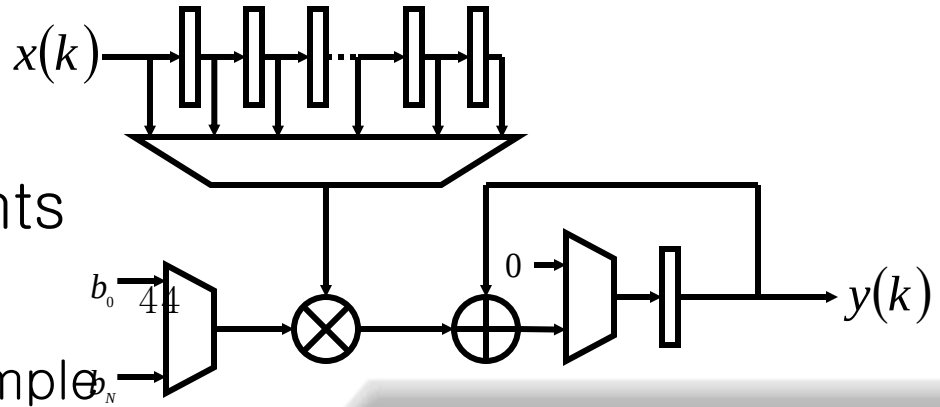
  - Move the pipeline registers into the multiplier:
    - Paths between pipeline stages are balanced
    - Improved timing
  - Tclock = (Tadd + Tmult)/2 + Treg

Architecture (6)

- Iterative Decomposition:
  - Reuse Hardware



- Identify regularity and reusable hardware components
- Add control
  - multiplexers
  - storage elements
  - Control
- Increases Cycles/Sample

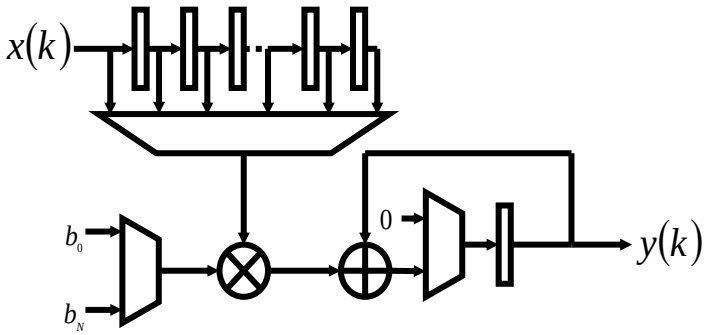


RTL-Design

- Choose an architecture under the following constraints:
  - It meets ALL timing specifications/constraints:
    - Throughput
    - Latency
  - It consumes the smallest possible area
  - It requires the least possible amount of power

Iterative Decomposition

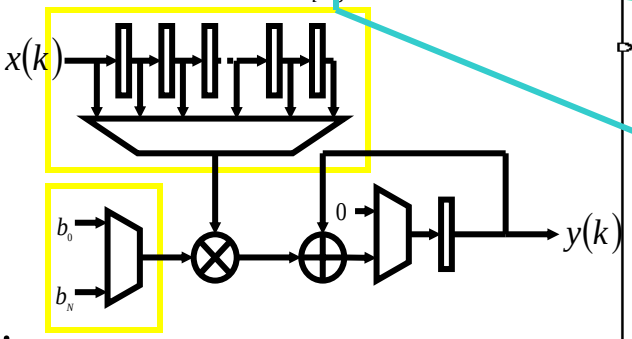
- Decide which additional functions are needed and how they can be implemented efficiently:
  - Storage of samples *x(k)* => MEMORY
  - Storage of coefficients b_i => LUT
  - Address generators for MEMORY and LUT => COUNTERS
  - Control => FSM



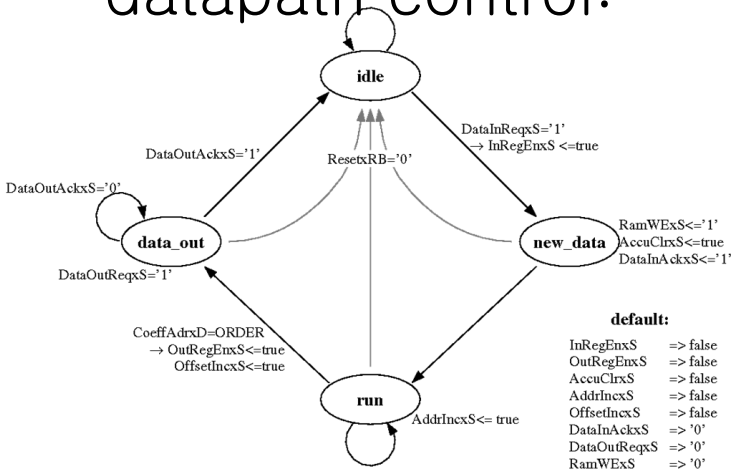
RTL-Design

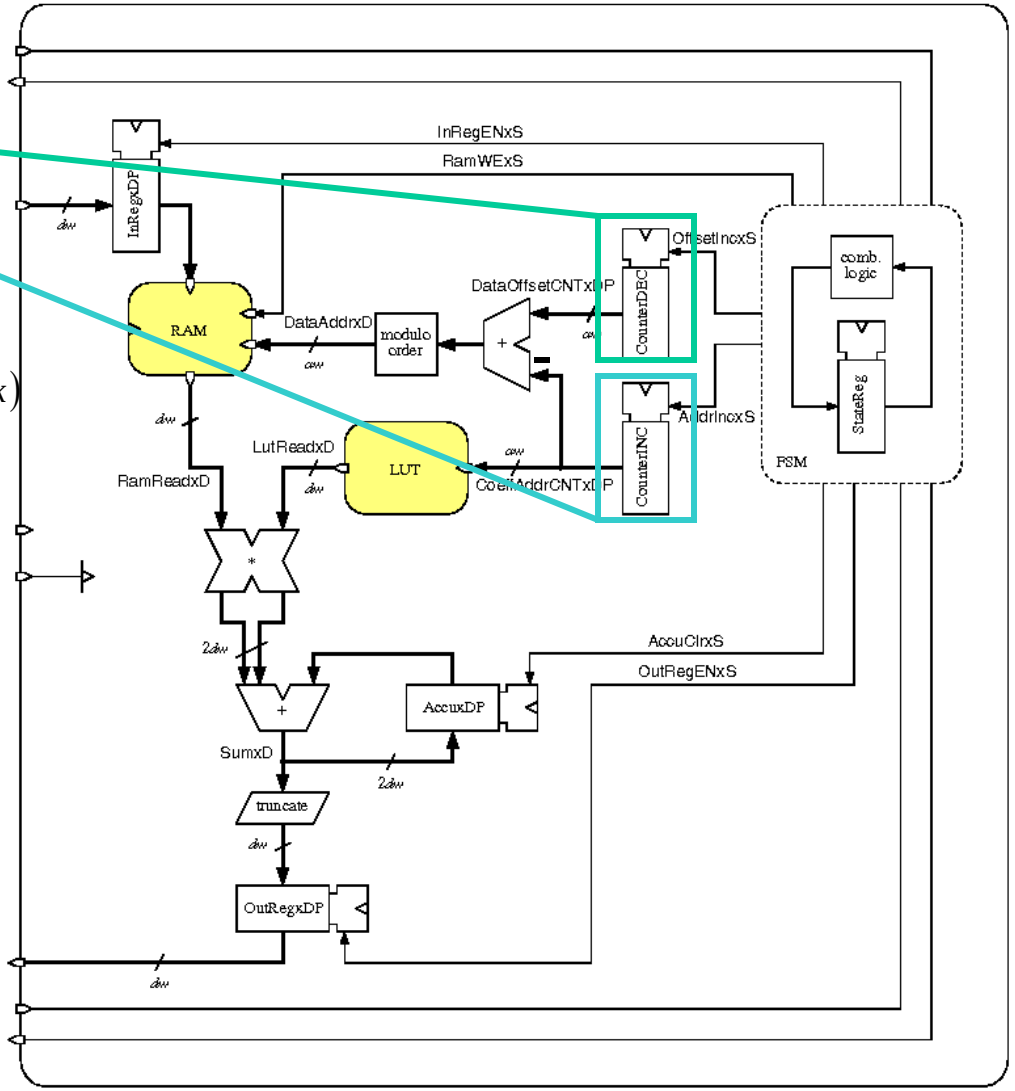
- RTL Block-diagram:
  - Datapath

$$y(k) = \sum_{i=0}^{N} b_i x(k-i)$$



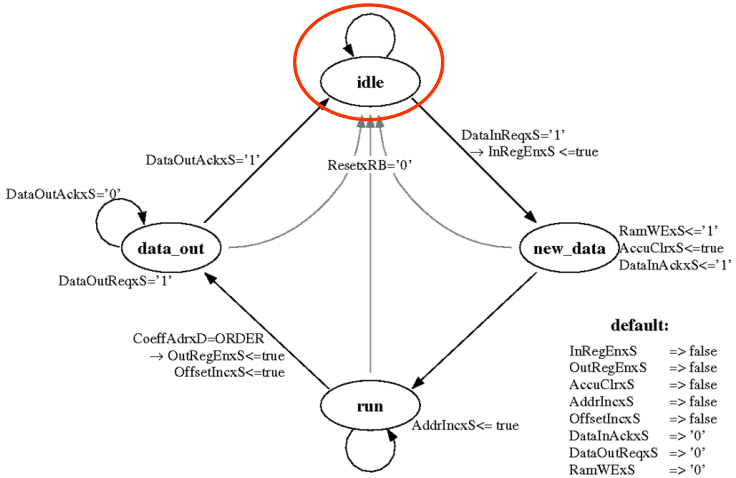
- FSM:
  - Interface protocols datapath control:

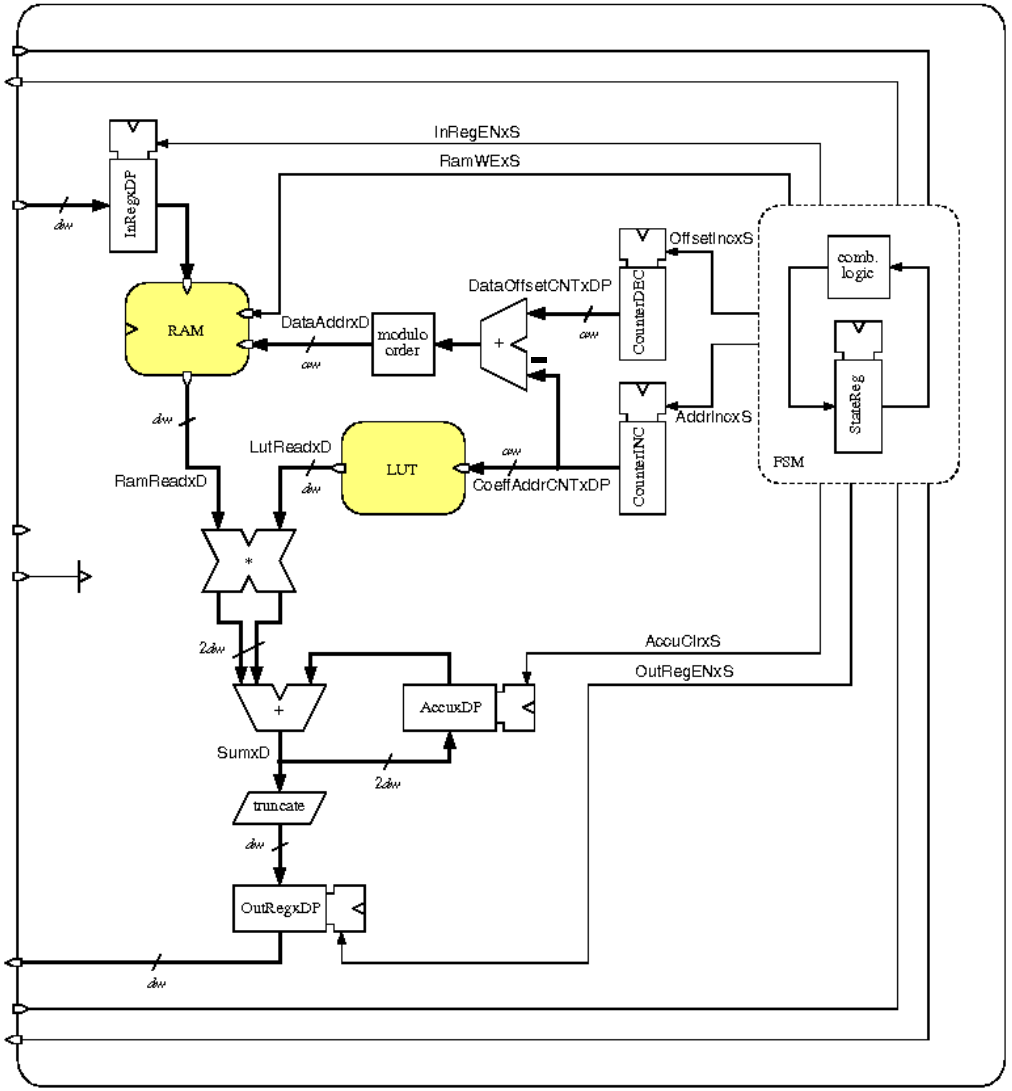




RTL-Design

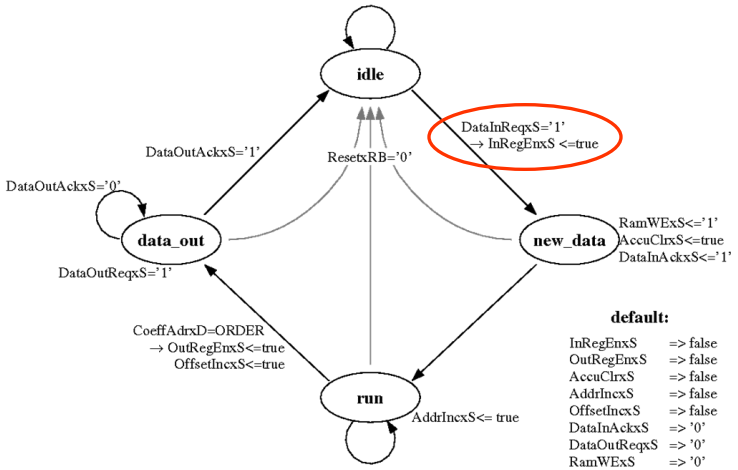
- How it works: $y(k) = \sum_{i=0}^{N} b_i x(k-i)$
  - IDLE
    - Wait for new sample

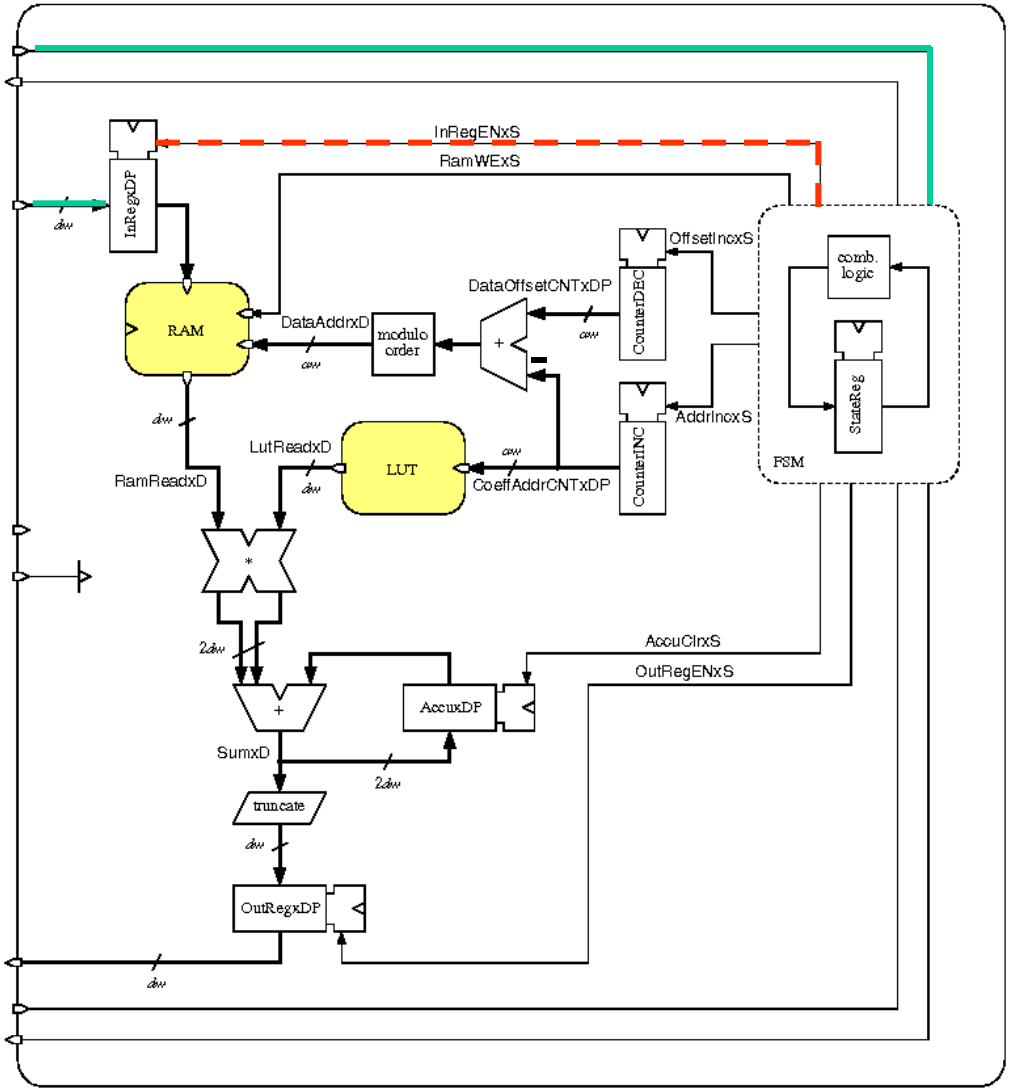




RTL-Design

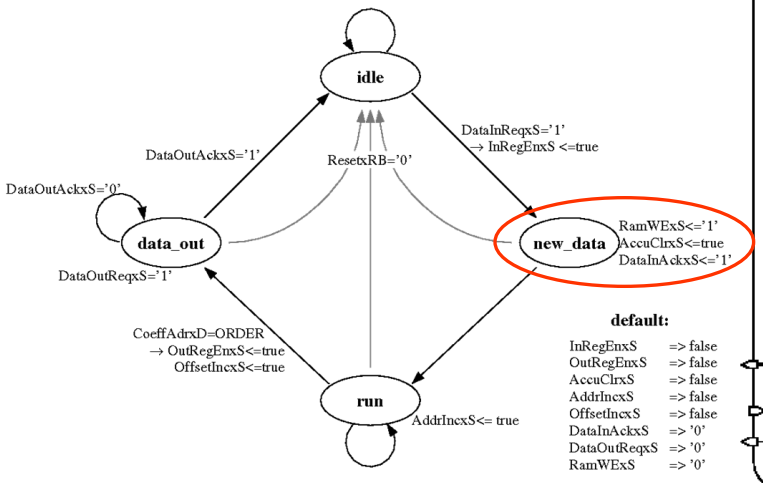
- How it works $y(k) = \sum_{i=0}^{N} b_i x(k-i)$
  - IDLE
    - Wait for new sample
    - Store to input register

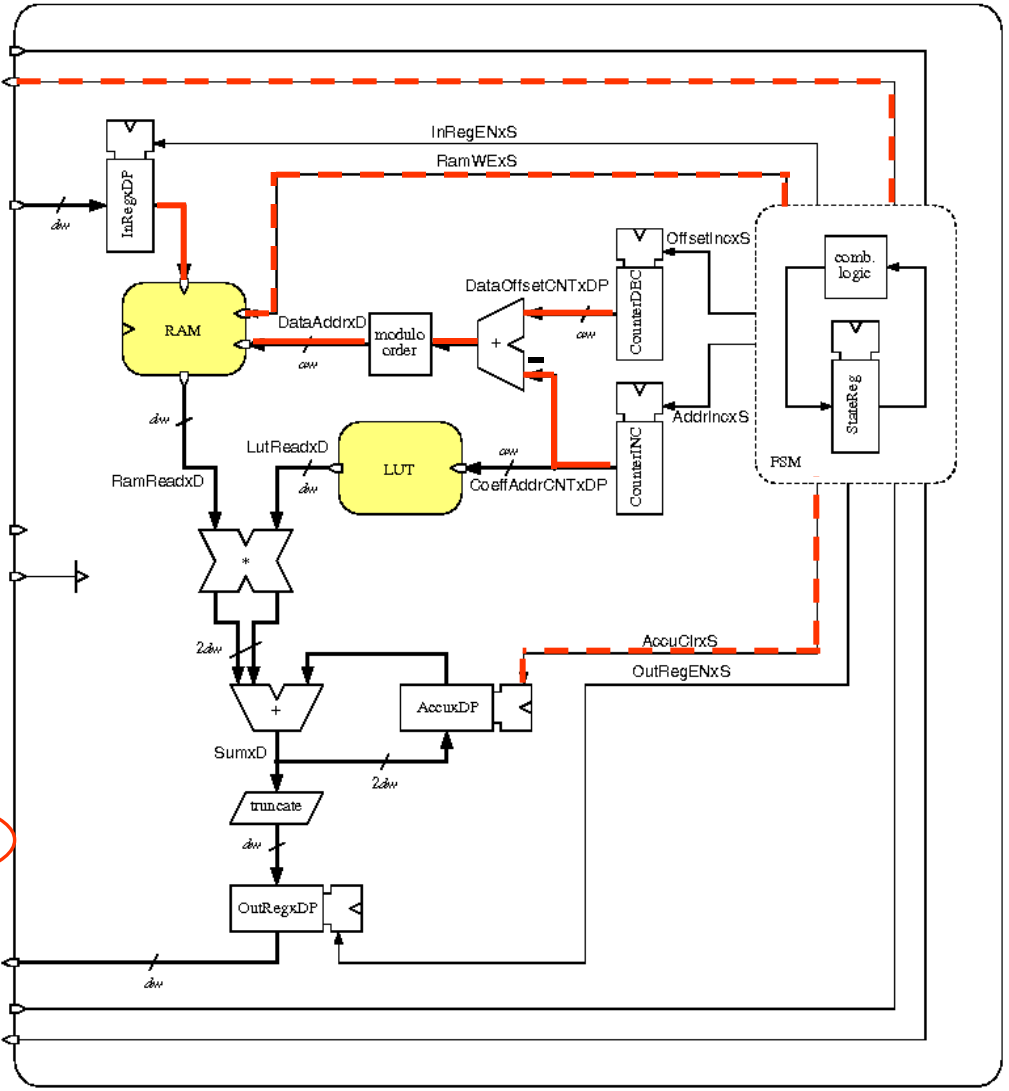




RTL-Design

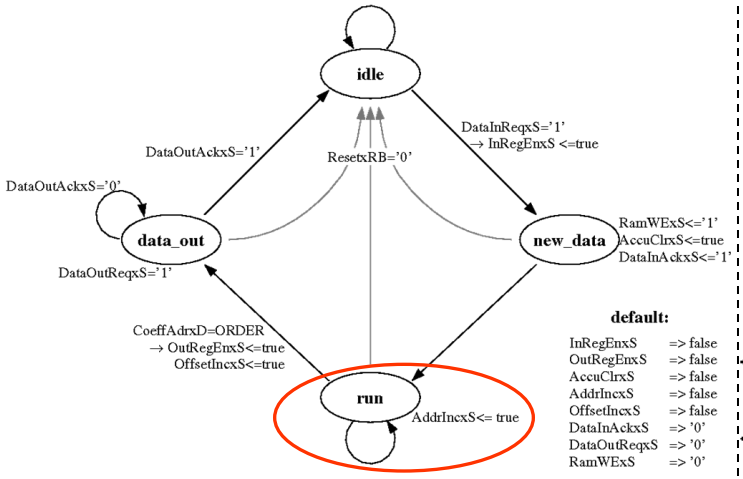
- How it works $y(k) = \sum_{i=0}^{N} b_i x(k-i)$
  - IDLE
    - Wait for new sample
    - Store to input register
  - NEW DATA:
    - Store new sample to memory

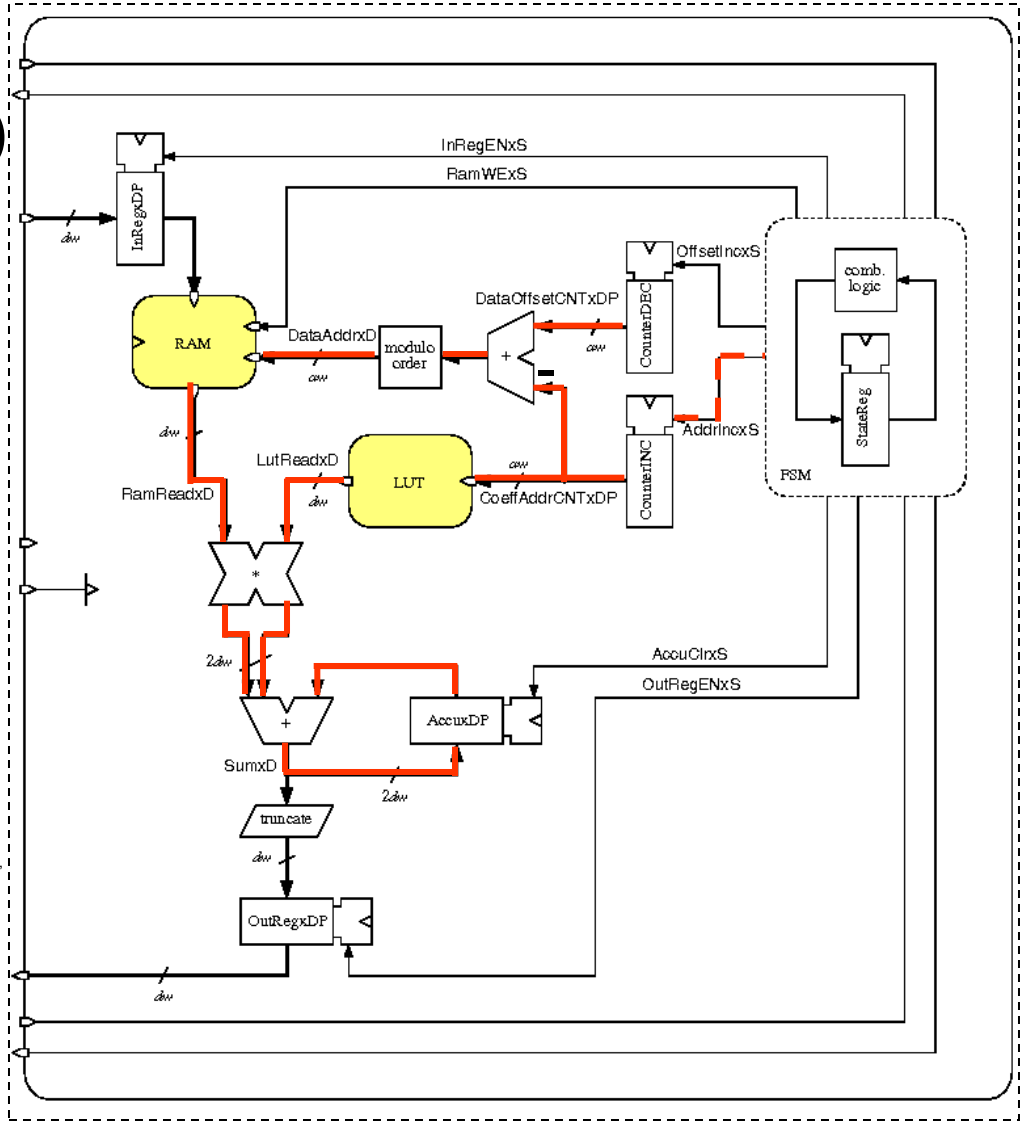




RTL-Design

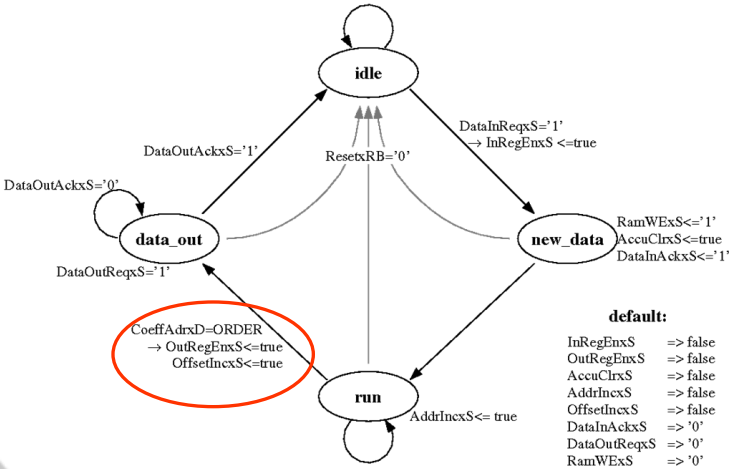
- How it works $y(k)=\sum_{i=0}^{N} b_i x(k-i)$
  - IDLE
    - Wait for new sample
    - Store to input register
  - NEW DATA:
    - Store new sample to memory
  - RUN:
    - $y(k)=\sum_{i=0}^{N} b_i x(k-i)$

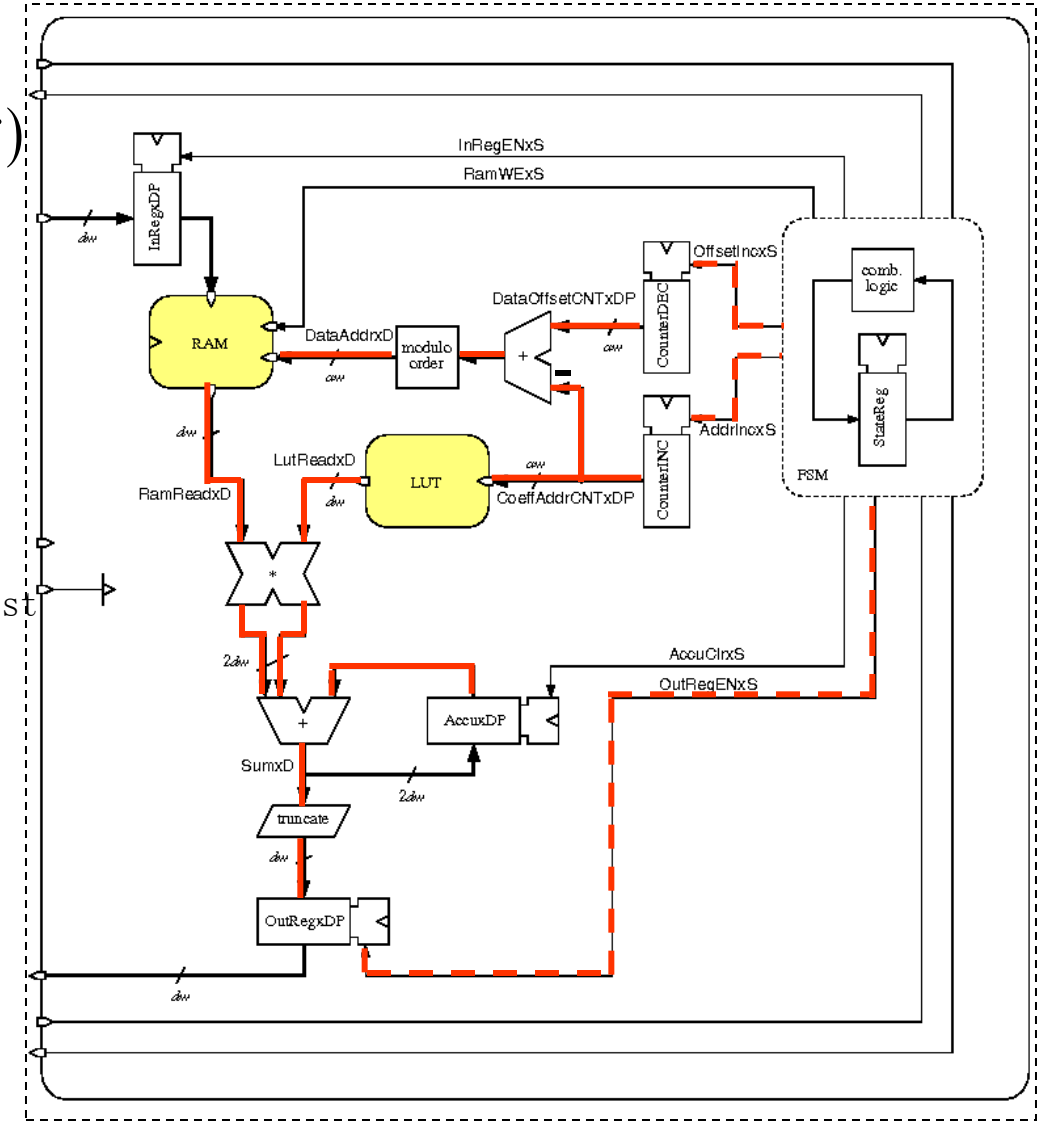




RTL-Design

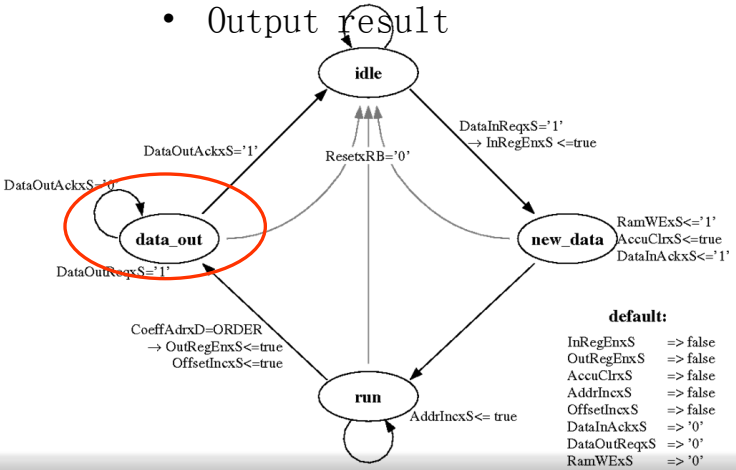
- How it works $y(k)=\sum_{i=0}^{N} b_i x(k-i)$
  - IDLE
    - Wait for new sample
    - Store to input register
  - NEW DATA:
    - Store new sample to memory
  - RUN:
    - $y(k)=\sum_{i=0}^{N} b_i x(k-i)$
    - Store result to output register

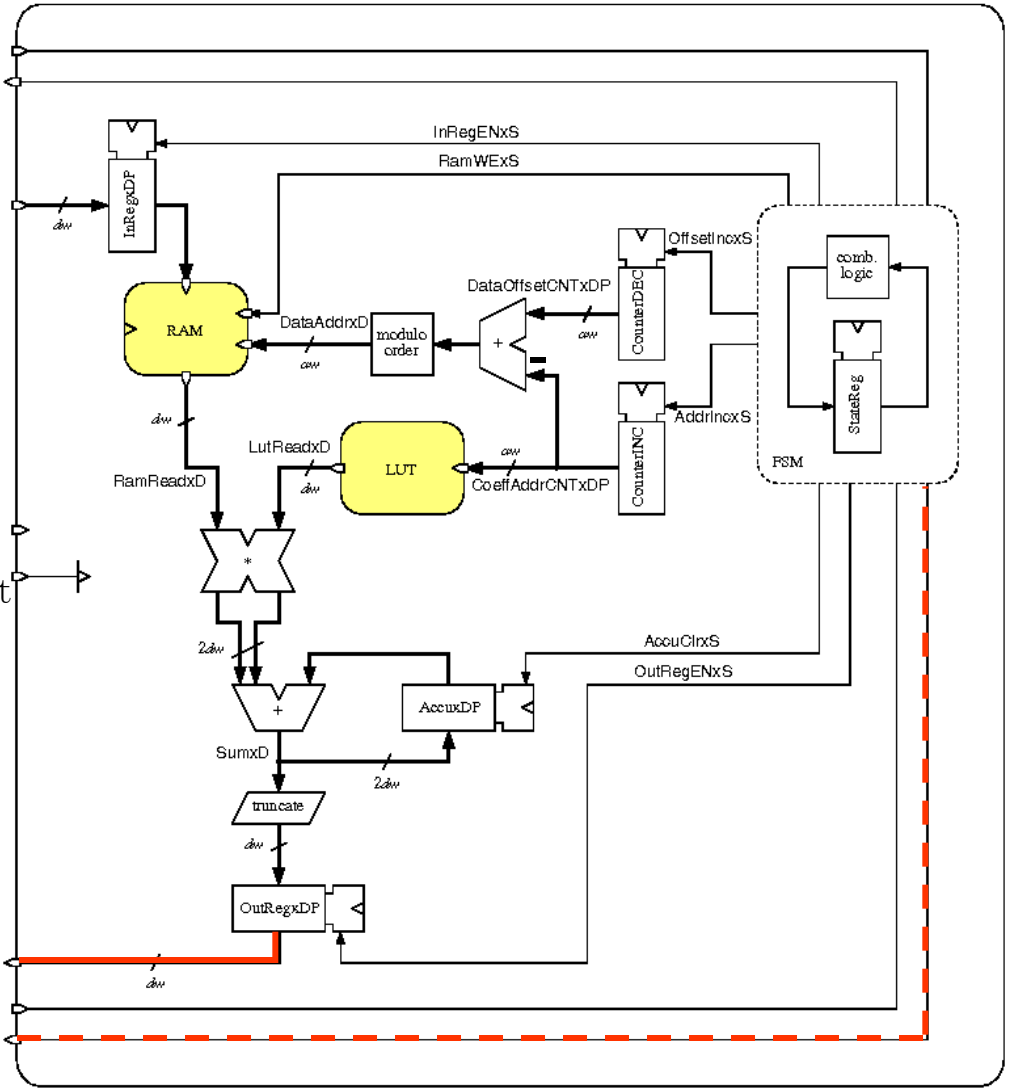




RTL-Design

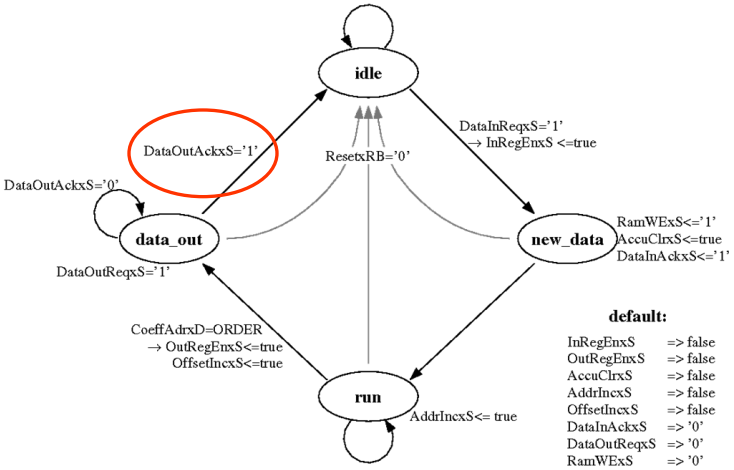
- How it works $y(k)=\sum_{i=0}^{N} b_i x(k-i)$
  - IDLE
    - Wait for new sample
    - Store to input register
  - NEW DATA:
    - Store new sample to memory
  - RUN:
    - $y(k)=\sum_{i=0}^{N} b_i x(k-i)$
    - Store result to output register
  - DATA OUT:
    - Output result

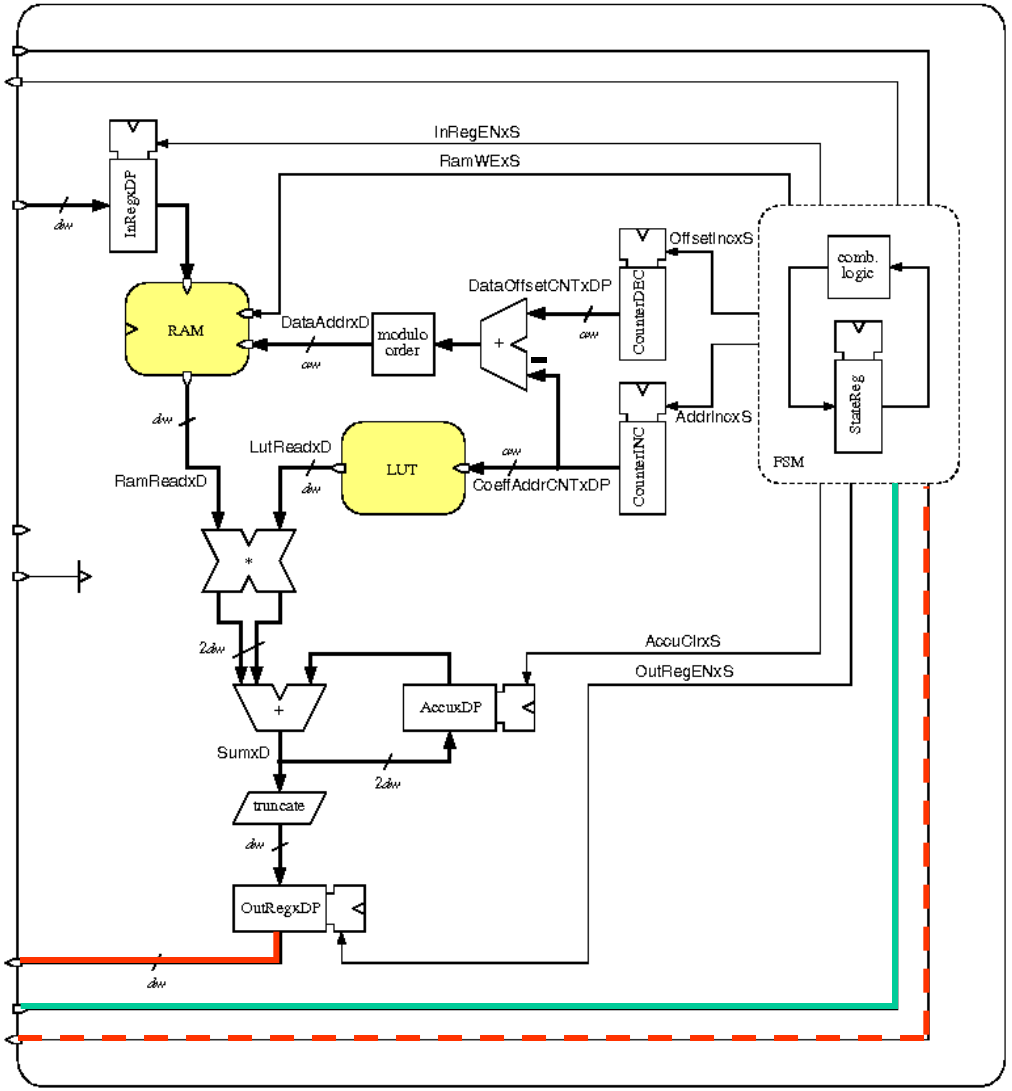




RTL-Design

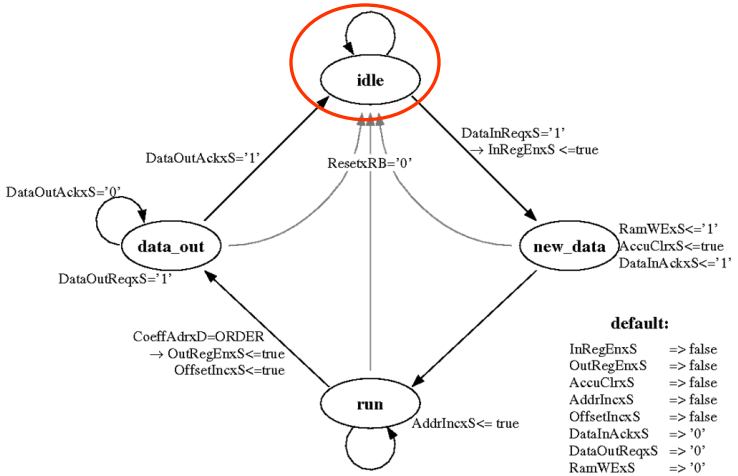
- How it works $y(k) = \sum_{i=0}^{N} b_i x(k-i)$
  - IDLE
    - Wait for new sample
    - Store to input register
  - NEW DATA:
    - Store new sample to memory
  - RUN:
    - $y(k) = \sum_{i=0}^{N} b_i x(k-i)$
    - Store result to output register
  - DATA OUT:
    - Output result / Wait for ACK

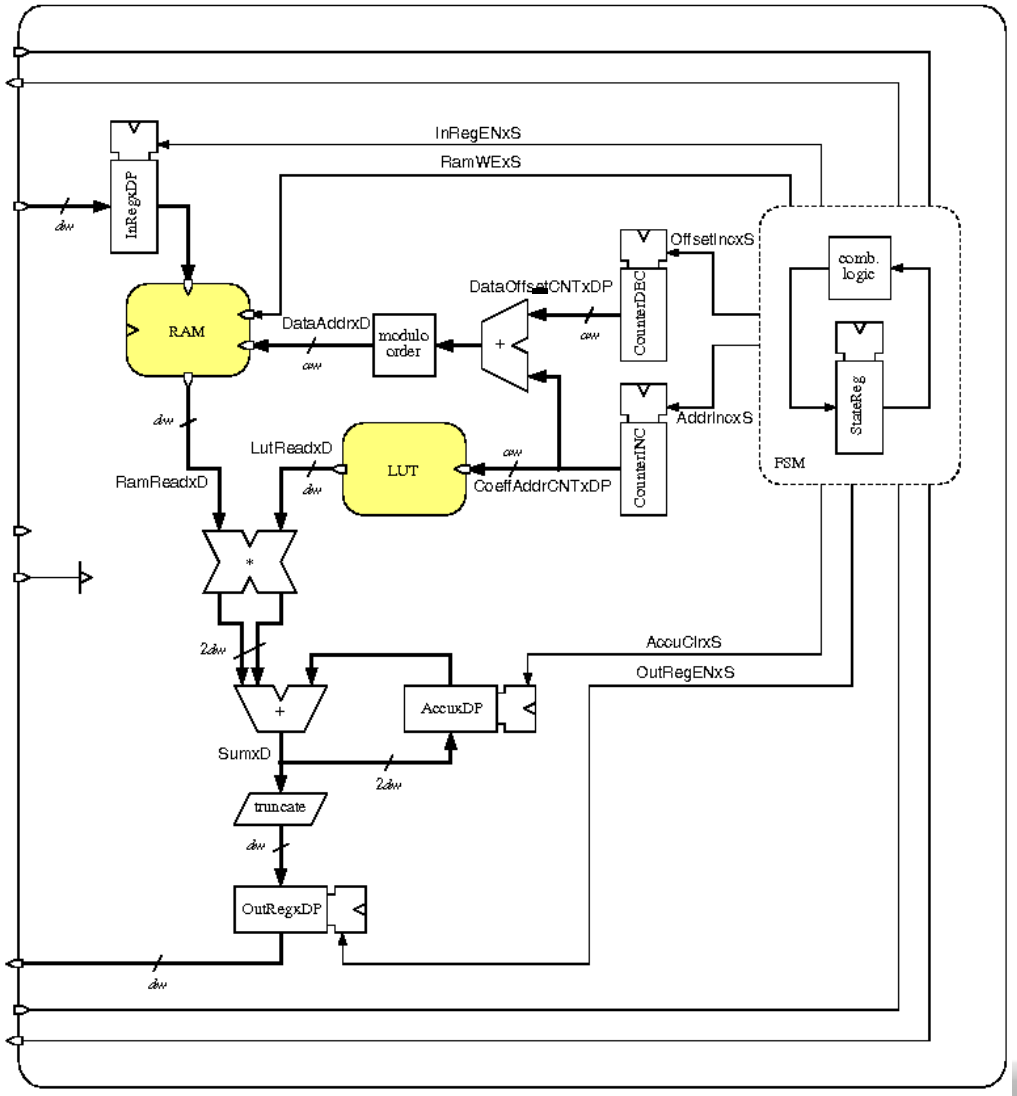




RTL-Design

- How it works: $y(k) = \sum_{i=0}^{N} b_i x(k-i)$
  - IDLE
    - Wait for new sample
    - Store to input register
  - NEW DATA:
    - Store new sample to memory
  - RUN:
    - $y(k) = \sum_{i=0}^{N} b_i x(k-i)$
    - Store result to output register
  - DATA OUT:
    - Output result / Wait for ACK
  - IDLE: ...





Translation into RTL

- Some basic rules:
  - Sequential processes (FlipFlops)
    - Only CLOCK and RESET in the sensitivity list
    - Logic signals are NEVER used as clock signals
  - Combinatorial processes
    - Multiple assignments to the same signal are ONLY possible within the same process => ONLY the last assignment is valid
    - Something must be assigned to each signal in any case OR There MUST be an ELSE for every IF statement

- More rules that help to avoid problems and surprises:
  - Use separate signals for the PRESENT state and the NEXT state of every FlipFlop in your design.
  - Use variables ONLY to store intermediate results or even avoid them whenever possible in an RTL design.

其他方法

- FIR filter implementation
  - Traditional Method
    - MAC (Multiply Accumulate) implementation
  - Other Methods
    - DA (Distributed Arithmetic) implementation
    - Add and Shift method