Chapter 7

数字系统设计（同步）

目录

1 同步数字系统结构

2 总线结构

3 简单处理器

4 乘法器

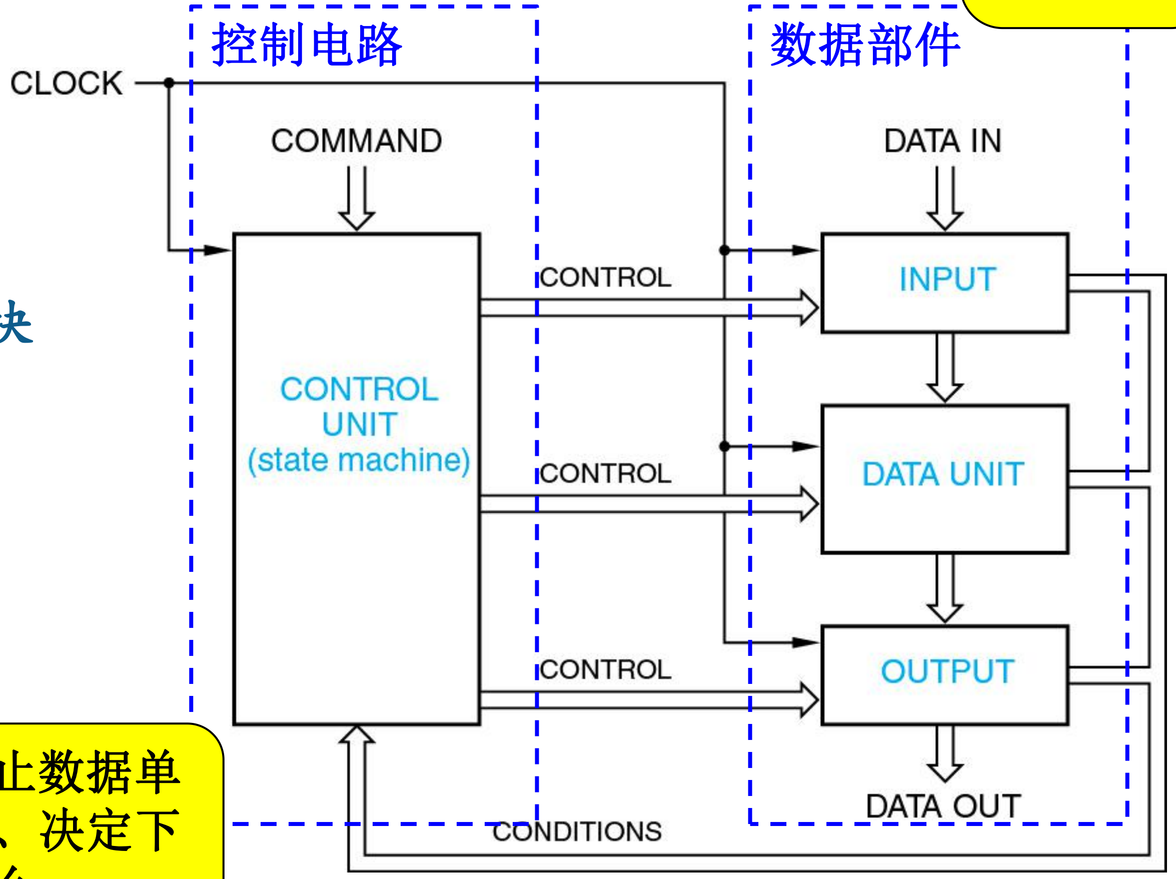
5 设计思想

数字系统的概念

- 数字系统按照功能分为数据处理电路和控制电路。

存储、传输、组合、处理“数据”

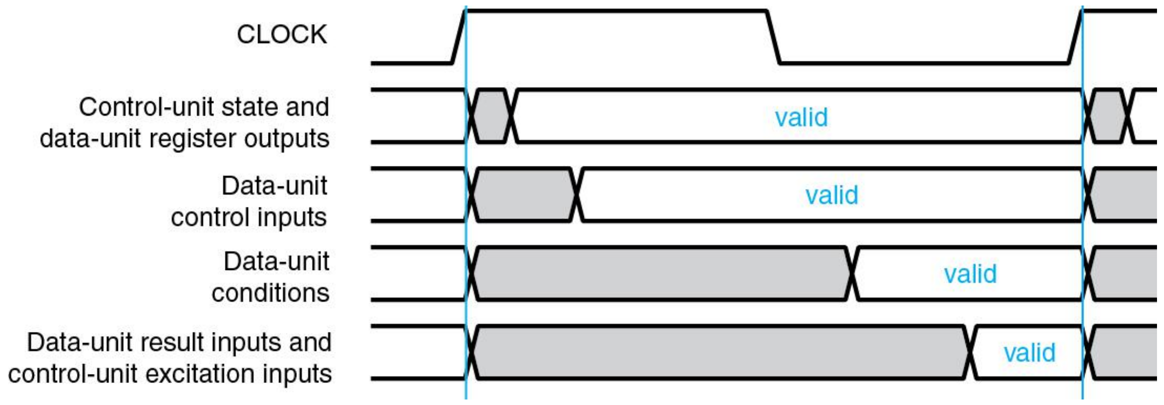
- 数据部件
  - 组合型功能
  - 寄存器
  - 特定时序功能模块
  - 存储器操作
- 控制部件
  - 状态机

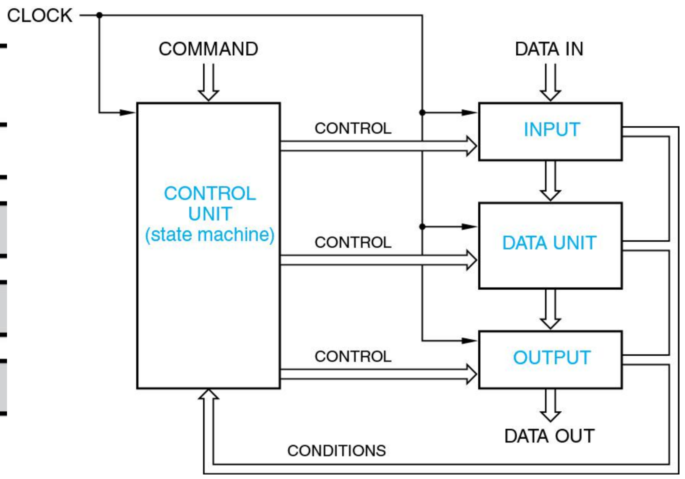


启动、停止数据单元的动作、决定下一步做什么

同步系统的时序

- 同步系统的关键特点：所有电路单元使用同一个时钟信号


CLOCK
Control-unit state and data-unit register outputs
valid
Data-unit control inputs
valid
Data-unit conditions
valid
Data-unit result inputs and control-unit excitation inputs
valid



- 时序特点：
  - 时钟沿之后，控制单元的状态和数据单元的寄存器输入有效
  - 经过一个组合逻辑延迟，FSM的Moore输出有效，给数据单元用于控制
  - 时钟周期后期，数据单元输出有效，送给控制单元
  - 时钟周期末，状态机的次态逻辑结果有效；数据单元运算结果被载入到数据单元寄存器

同步系统的时序

- 状态机的输出信号可以是Moore型、Mealy型、寄存器后Mealy型
- Moore型：上图中时序
- Mealy型：依据数据单元当前状态和命令条件等输入来选择，灵活，但数据通路延迟变长；一定不能有反馈
- 寄存器后Mealy型：比Moore型输出的延迟更小，提高整个系统的工作频率（缩短时钟周期）

目录

1 同步数字系统结构

2 总线结构

3 简单处理器

4 乘法器

5 设计思想

基于三态缓冲器的总线

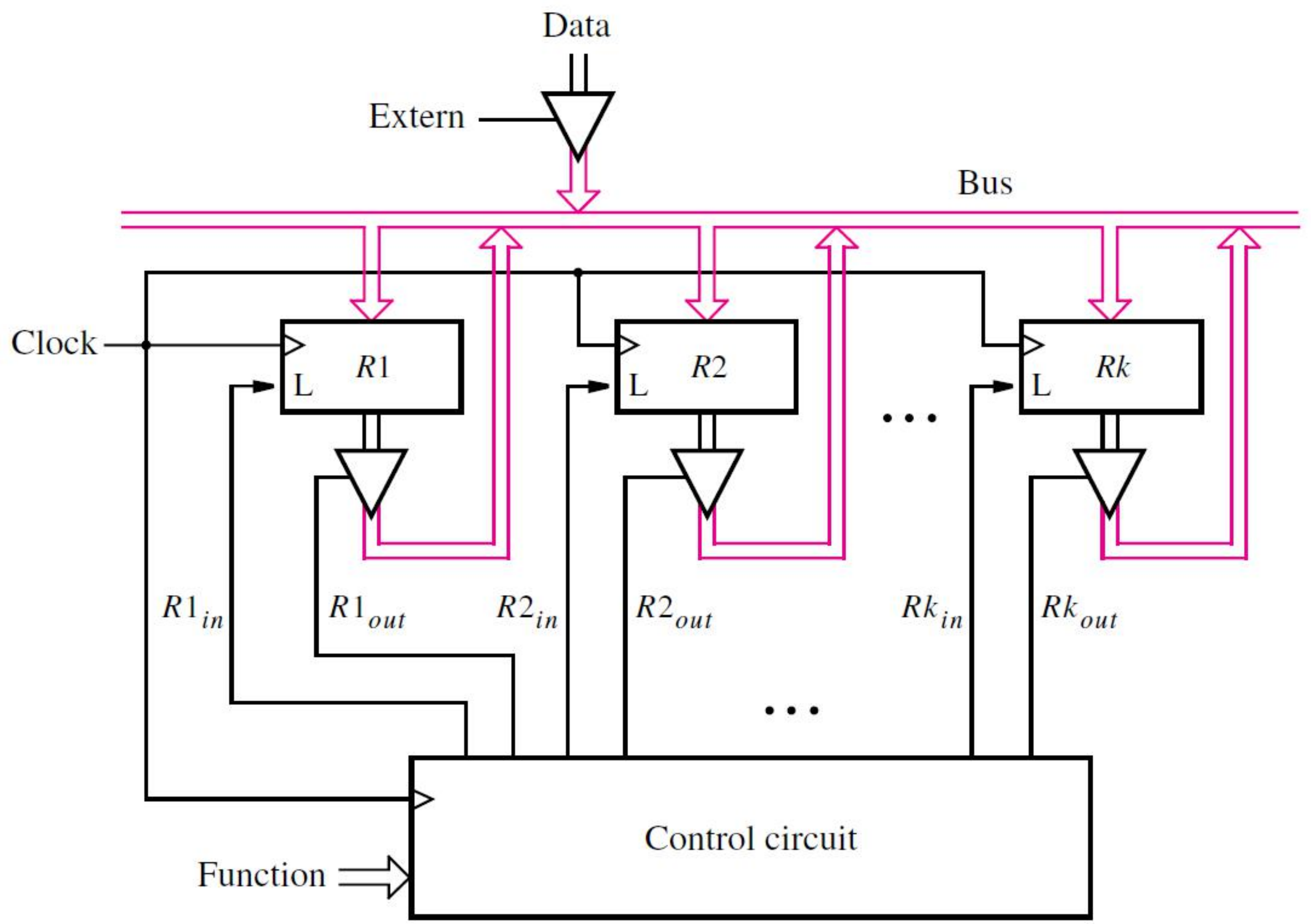
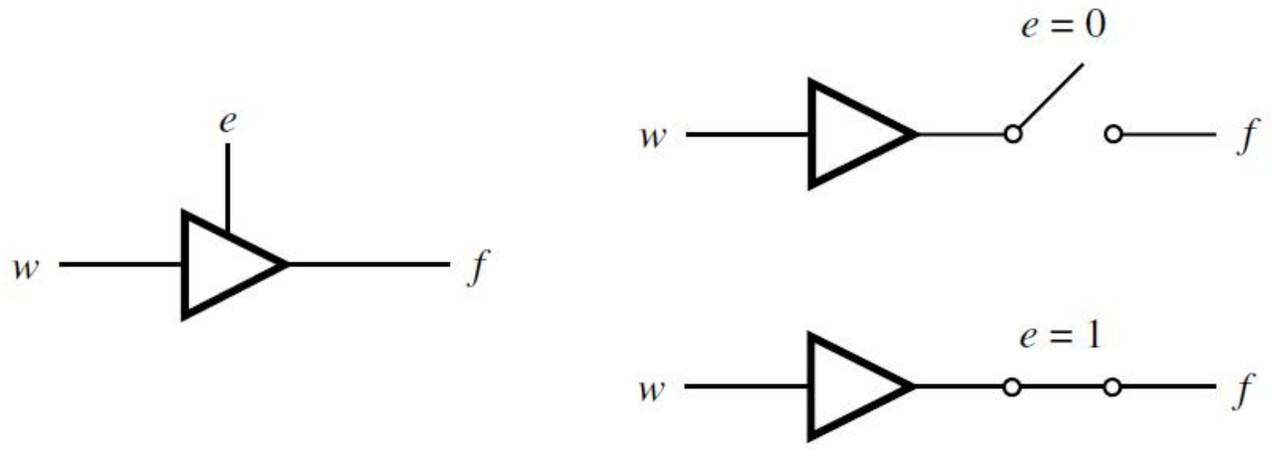


Figure 7.2. A digital system with *k* registers.

三态缓冲器



(a) Symbol

(b) Equivalent circuit

e	w	f
0	0	Z
0	1	Z
1	0	0
1	1	1

(c) Truth table

Figure 7.1. Tri-state driver.

基于三态缓冲器的总线

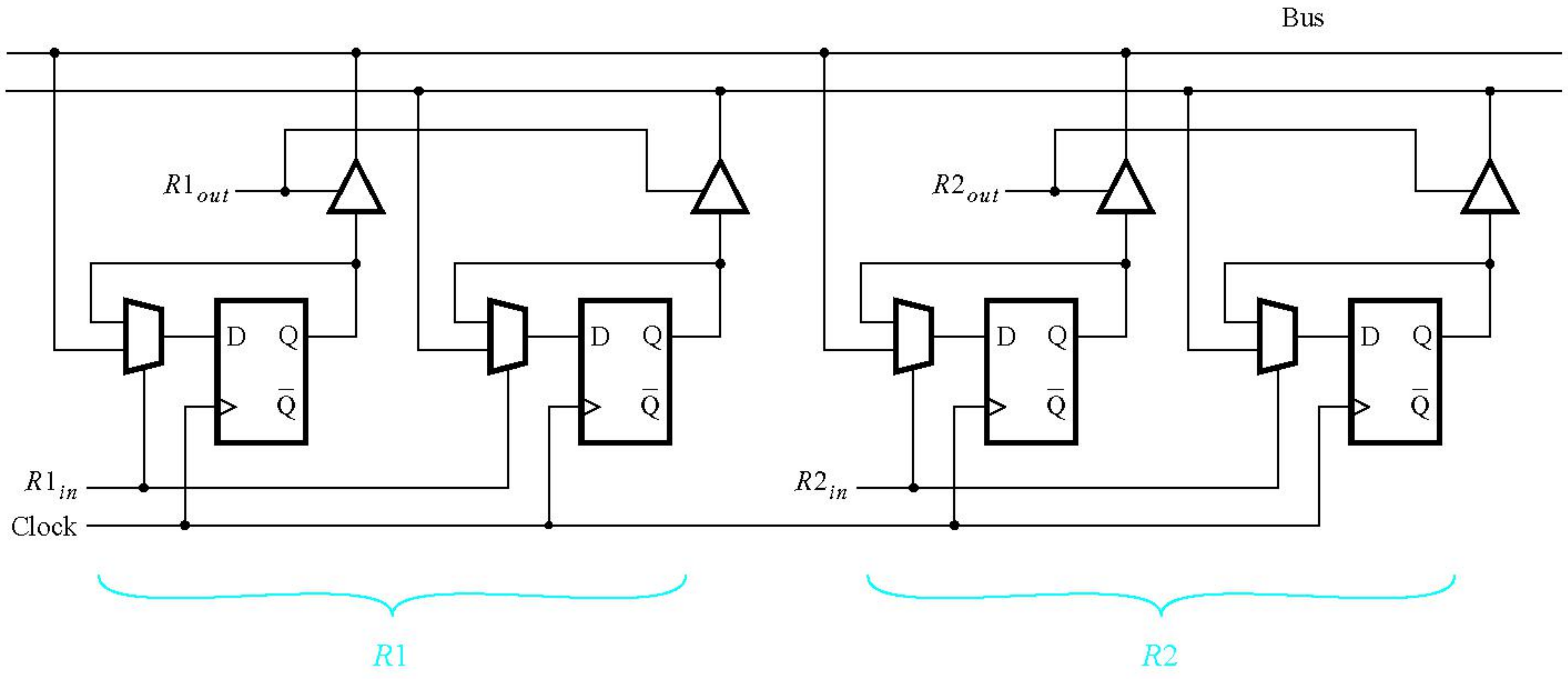


Figure 7.3. Details for connecting registers to a bus.

基于多路选择器的总线

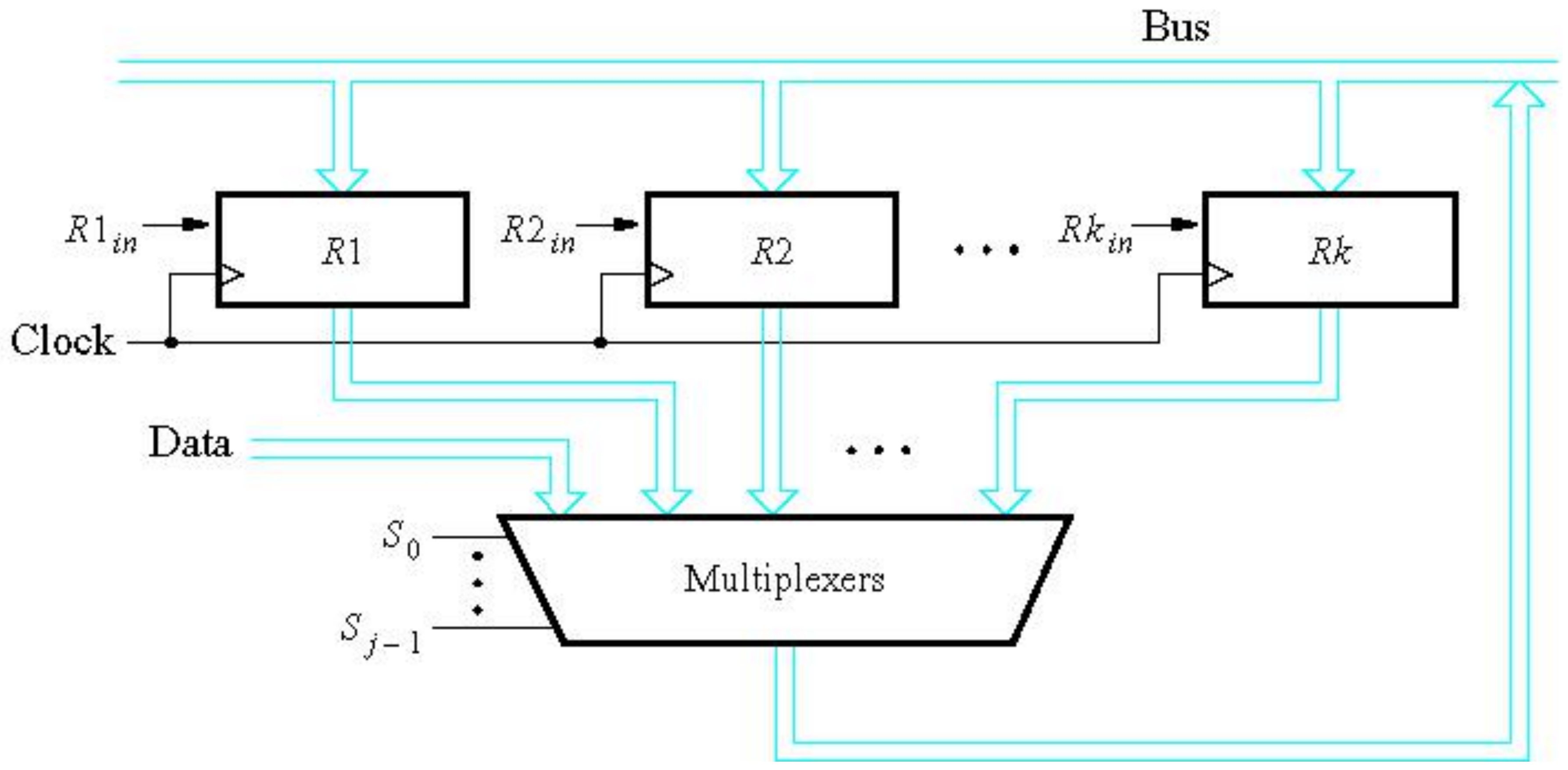
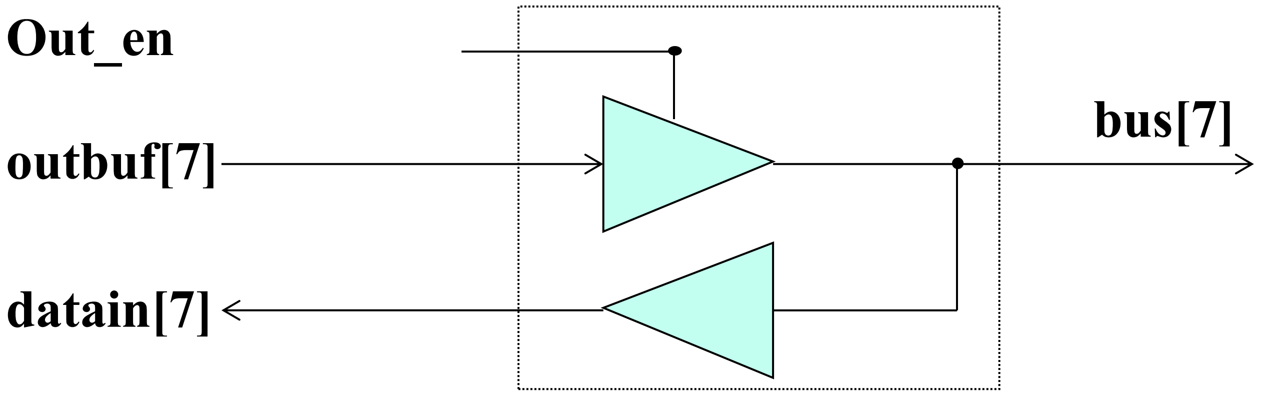


Figure 7.4. Using multiplexers to implement a bus.

三态总线的Verilog描述



```
`define   ON   1 ‘b 1
`define  OFF  1 ‘b 0
 wire  Out_en;
 wire [7:0]  outbuf, datain;
 inout [7:0] bus;
 assign  bus =  (Out_en == `ON) ? outbuf : 8 ‘hzz;
 assign  datain = bus;
```

三态总线的Verilog描述

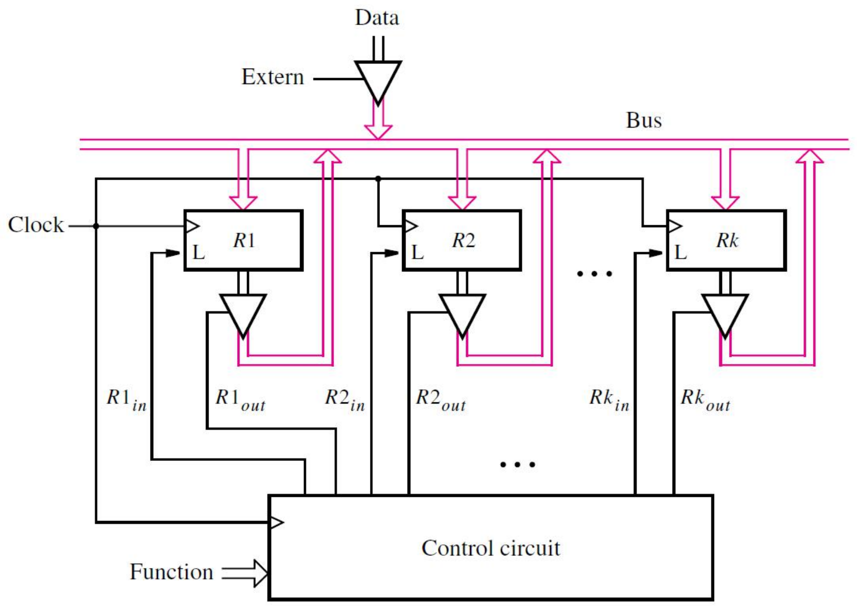
```
module  regn (R, L, Clock, Q);
    parameter n = 8;
    input  [n – 1:0] R;
    input  L, Clock;
    output  reg  [n – 1:0] Q;

    always @(posedge Clock)
        if (L)
            Q < = R;

endmodule
```

```
module trin (Y, E, F);
        parameter n = 8;
        input [n-1:0] Y;
        input E;
        output wire [n-1:0] F;

        assign F = E ? Y : n'bz;

endmodule
```

```
module  swap (Resetn, Clock, w, Data, Extern, RinExt1, RinExt2, RinExt3, BusWires, Done);
   parameter  n = 8;
   input  Resetn, Clock, w, Extern, RinExt1, RinExt2, RinExt3;
   input  [n–1:0] Data;
   output tri  [n–1:0] BusWires;
   output  Done;
   wire  [n–1:0] R1, R2, R3;
   wire  R1in, R1out, R2in, R2out, R3in, R3out;
   reg  [2:1] y, Y;
   parameter  [2:1] A = 2'b00,  B = 2'b01,  C = 2'b10,  D = 2'b11;

   // Define the next state combinational circuit for FSM
   always @(w, y)
      case (y)
         A:  if (w)    Y = B;
             else      Y = A;
         B:            Y = C;
         C:            Y = D;
         D:            Y = A;
      endcase

   // Define the sequential block for FSM
   always @(negedge Resetn, posedge Clock)
      if  (Resetn == 0)   y <= A;
      else   y <= Y;

   // Define outputs of FSM
   assign  R2out = (y == B);
   assign  R3in = (y == B);
   assign  R1out = (y == C);
   assign  R2in = (y == C);
   assign  R3out = (y == D);
   assign  R1in = (y == D);
   assign  Done = (y == D);

   // Instantiate registers
   regn reg_1 (BusWires, RinExt1 | R1in, Clock, R1);
   regn reg_2 (BusWires, RinExt2 | R2in, Clock, R2);
   regn reg_3 (BusWires, RinExt3 | R3in, Clock, R3);
   // Instantiate tri-state drivers
   trin tri_ext (Data, Extern, BusWires);
   trin tri_1 (R1, R1out, BusWires);
   trin tri_2 (R2, R2out, BusWires);
   trin tri_3 (R3, R3out, BusWires);
endmodule
```

控制电路

数据通路

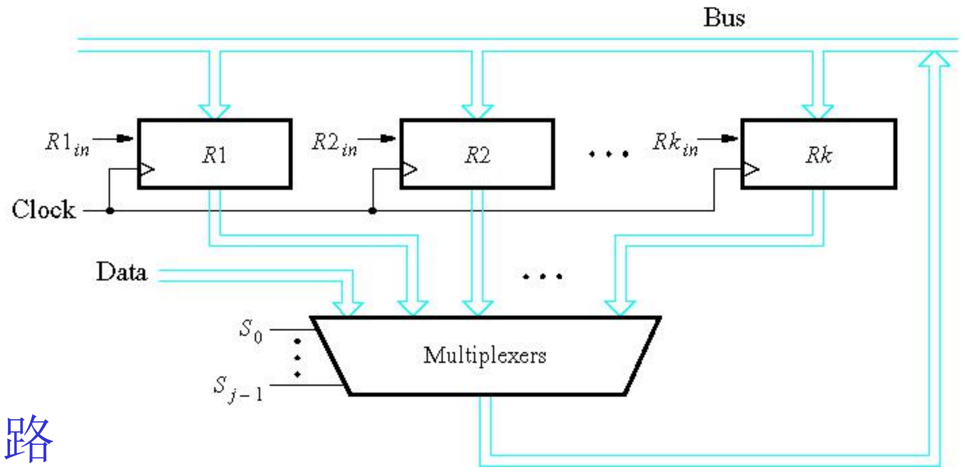


```
module swapmux (Resetn, Clock, w, Data, RinExt1, RinExt2, RinExt3, BusWires, Done);
   parameter n = 8;
   input Resetn, Clock, w, RinExt1, RinExt2, RinExt3;
   input [n–1:0] Data;
   output reg [n–1:0] BusWires;
   output Done;
   wire [n–1:0] R1, R2, R3;
   wire R1in, R2in, R3in;
   reg [2:1] y, Y;
   parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;

   // Define the next state combinational circuit for FSM
   always @(w, y)
      case (y)
         A: if (w)    Y = B;
            else      Y = A;
         B:           Y = C;
         C:           Y = D;
         D:           Y = A;
      endcase

   // Define the sequential block for FSM
   always @(negedge Resetn, posedge Clock)
      if (Resetn == 0)  y <= A;
      else  y <= Y;

   // Define control signals
   assign R3in = (y == B);
   assign R2in = (y == C);
   assign R1in = (y == D);
   assign Done = (y == D);
```

控制电路

```
   // Instantiate registers
   regn reg_1 (BusWires, RinExt1 | R1in, Clock, R1);
   regn reg_2 (BusWires, RinExt2 | R2in, Clock, R2);
   regn reg_3 (BusWires, RinExt3 | R3in, Clock, R3);

   // Define the multiplexers
   always @(y, Data, R1, R2, R3)
      if (y == A) BusWires = Data;
      else if (y == B) BusWires = R2;
      else if (y == C) BusWires = R1;
      else BusWires = R3;
endmodule
```

数据通路



目录

1 同步数字系统结构

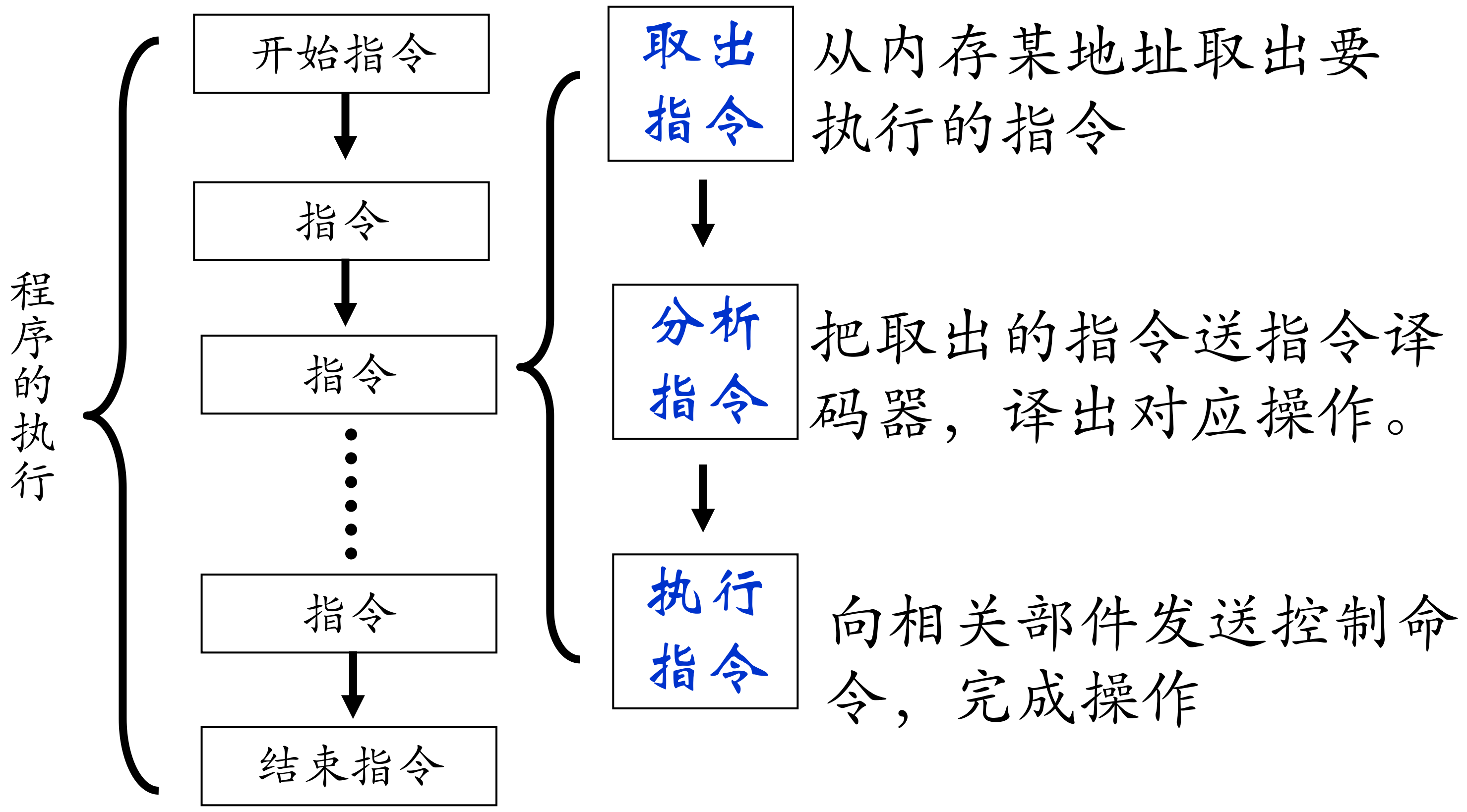
2 总线结构

3 简单处理器

4 乘法器

5 设计思想

指令的执行过程



从内存某地址取出要执行的指令

把取出的指令送指令译码器，译出对应操作。

向相关部件发送控制命令，完成操作

处理器指令集

	Operation	Function Performed
单周期	Load *Rx, Data*	*Rx* ← *Data*
	Move *Rx, Ry*	*Rx* ← *[Ry]*
多周期	Add *Rx, Ry*	*Rx* ← *[Rx]+[Ry]*
	Sub *Rx, Ry*	*Rx* ← *[Rx]-[Ry]*

Table 7.1. Operations performed in the processor.

控制逻辑

生成与时钟精确配合的开关时序是计算逻辑的核心。

	T1	T2	T3
(Load):I_0	*Extern*, $R_{in}=X$, *Done*		
(Move):I_1	$R_{in}=X$, $R_{out}=Y$, *Done*		
(Add):I_2	$R_{out}=X$, A_{in}	$R_{out}=Y$，G_{in}，*AddSub=0*	G_{out}，$R_{in}=X$, *Done*
(Sub):I_3	$R_{out}=X$, A_{in}	$R_{out}=Y$, G_{in}，*AddSub=1*	G_{out}，$R_{in}=X$, *Done*

$$Extern = I_0T_1$$

$$AddSub = I_3$$

$$A_{in} = (I_2 + I_3)\ T_1$$

$$Done = (I_0 + I_1)\ T_1 + (I_2 + I_3)\ T_3$$

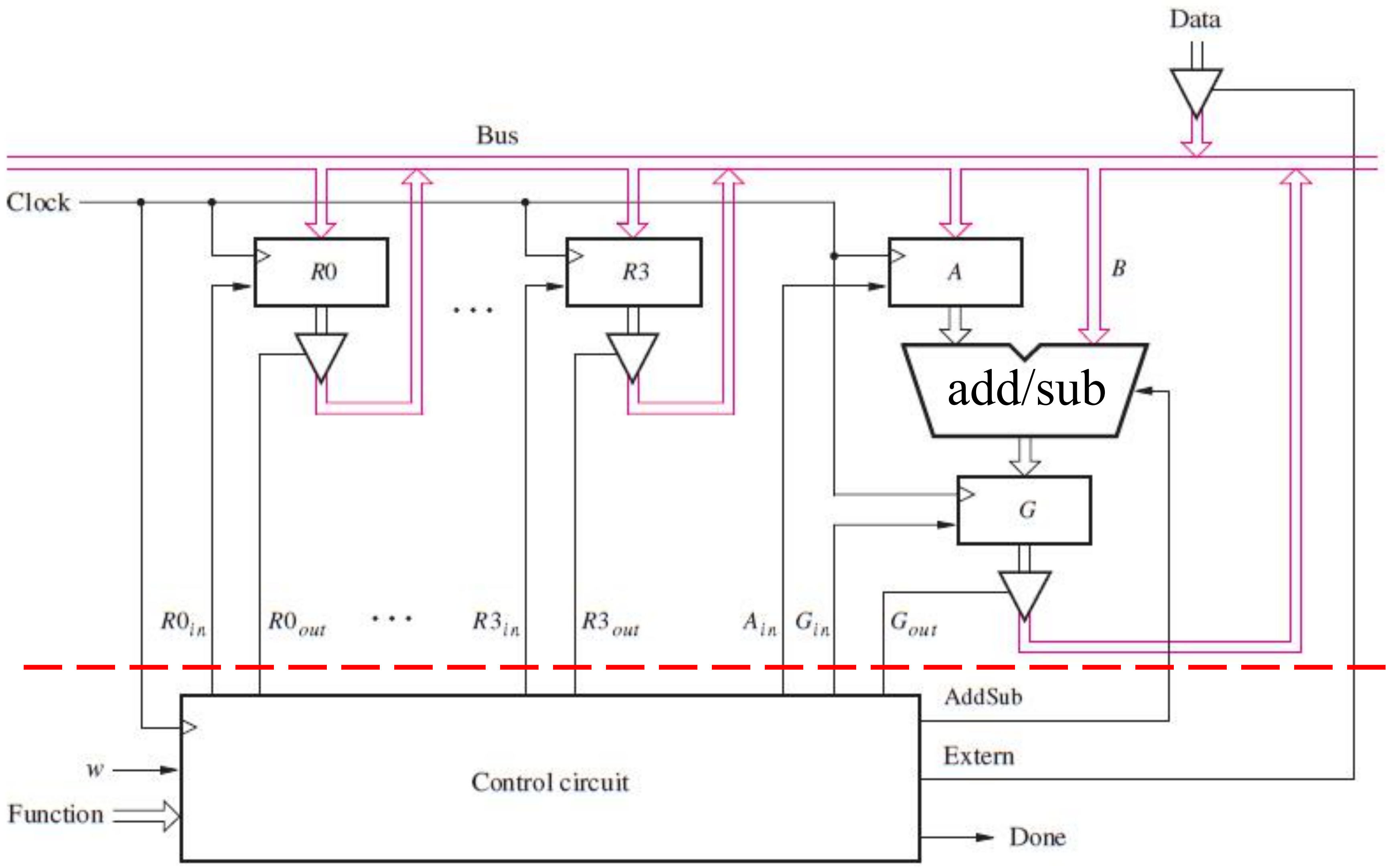
$$G_{in} = (I_2 + I_3)\ T_2$$

$$R0_{in} = (I_0 + I_1)T_1X_0 + (I_2 + I_3)T_3X_0$$

$$G_{out} = (I_2 + I_3)\ T_3$$

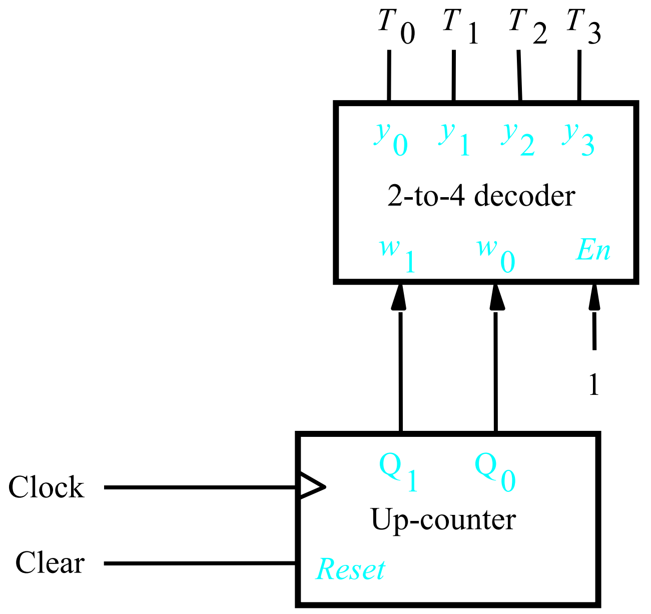
$$R0_{out} = I_1T_1Y_0 + (I_2 + I_3)(T_1X_0 + T_2Y_0)$$

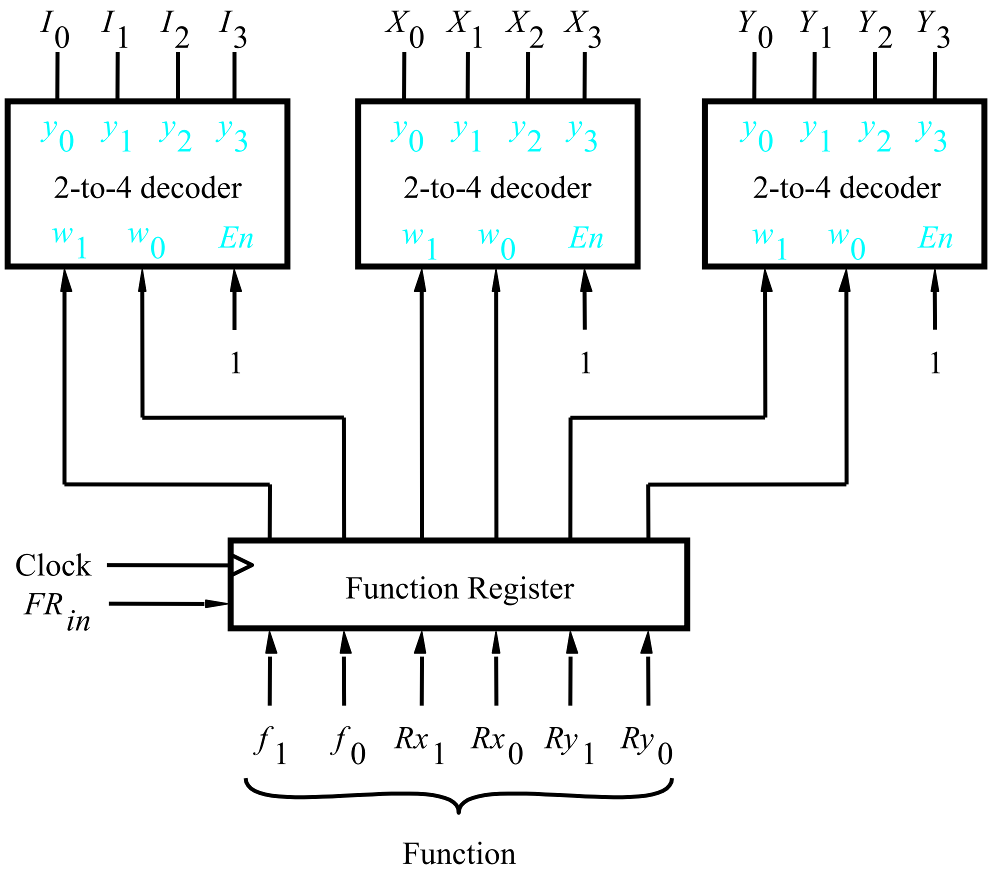
处理器模块框图



Data
Bus
Clock
R0
R3
A
B
add/sub
G
$R0_{in}$
$R0_{out}$
$R3_{in}$
$R3_{out}$
A_{in}
G_{in}
G_{out}
AddSub
Extern
w
Function
Control circuit
Done

操作控制（周期控制）





$$Clear = \overline{w}T_0 + Done$$

$$FR_{in} = \overline{w}T_0$$

操作控制器（周期控制）

```
module upcount (Clear, Clock, Q);
    input Clear, Clock;
    output reg [1:0] Q;

    always @(posedge Clock)
        if (Clear)
            Q <= 0;
        else
            Q <= Q + 1;

endmodule
```

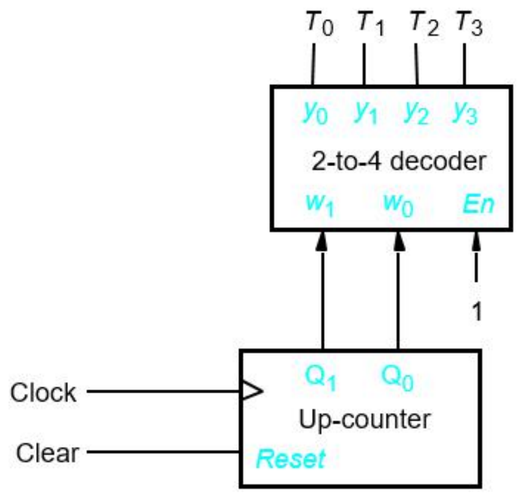
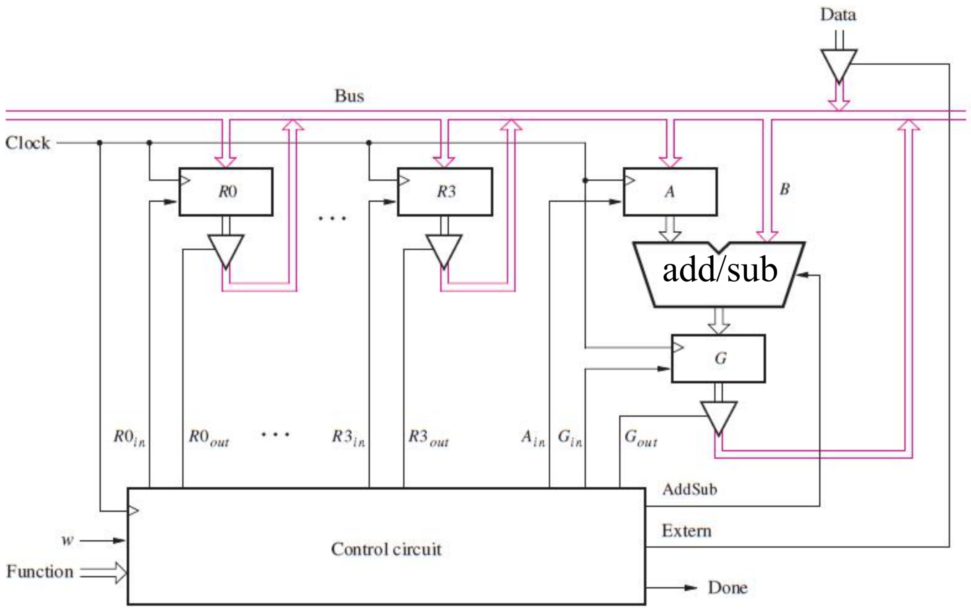


Figure 7.12. A two-bit up-counter with synchronous reset.

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires
    input [7:0] Data;
    input Reset, w, Clock;
    input [1:0] F, Rx, Ry;
    output wire [7:0] BusWires;
    output Done;
    reg [0:3] Rin, Rout;
    reg [7:0] Sum;
    wire Clear, AddSub, Extern, Ain, Gin, Gout, FRin;
    wire [1:0] Count;
    wire [0:3] T, I, Xreg, Y;
    wire [7:0] R0, R1, R2, R3, A, G;
    wire [1:6] Func, FuncReg;
    integer k;
    //控制电路
    upcount counter (Clear, Clock, Count);
    dec2to4 decT (Count, 1'b1, T);

    assign Clear = Reset | Done | (~w & T[0]);
    assign Func = {F, Rx, Ry};
    assign FRin = w & T[0];

    regn functionreg (Func, FRin, Clock, FuncReg);
       defparam functionreg.n = 6;
    dec2to4 decI (FuncReg[1:2], 1'b1, I);
    dec2to4 decX (FuncReg[3:4], 1'b1, Xreg);
    dec2to4 decY (FuncReg[5:6], 1'b1, Y);

    assign Extern = I[0] & T[1];
    assign Done = ((I[0] | I[1]) & T[1]) | ((I[2] | I[3]) & T[3]);
    assign Ain = (I[2] | I[3]) & T[1];
    assign Gin = (I[2] | I[3]) & T[2];
    assign Gout = (I[2] | I[3]) & T[3];
    assign AddSub = I[3];
```



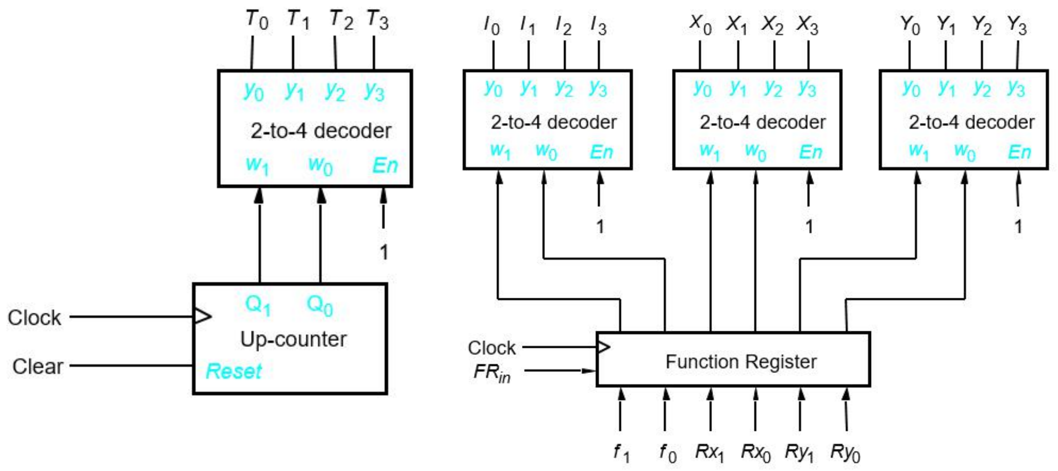
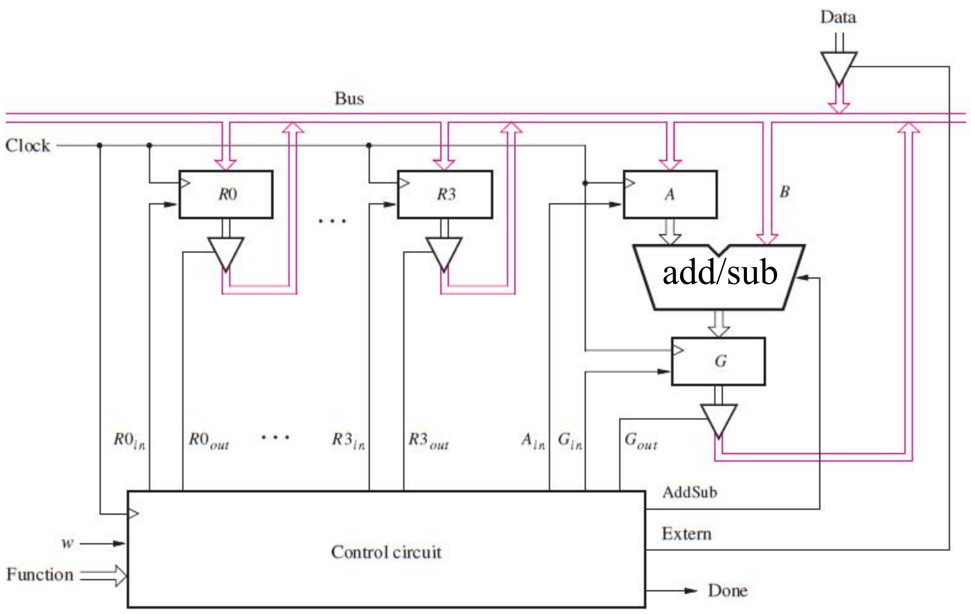


Figure 7.13. Code for the processor (Part *a*).

```
// RegCntl
always @(I, T, Xreg, Y)
    for (k = 0; k <  4; k = k+1)
    begin
        Rin[k] = ((I[0] | I[1]) & T[1] & Xreg[k]) |
            ((I[2] | I[3]) & T[3] & Xreg[k]);
        Rout[k] = (I[1] & T[1] & Y[k]) | ((I[2] | I[3]) &
            ((T[1] & Xreg[k]) | (T[2] & Y[k])));
    end
//数据通路
trin tri_ext (Data, Extern, BusWires);
regn  reg_0 (BusWires, Rin[0], Clock, R0);
regn  reg_1 (BusWires, Rin[1], Clock, R1);
regn  reg_2 (BusWires, Rin[2], Clock, R2);
regn  reg_3 (BusWires, Rin[3], Clock, R3);

trin  tri_0 (R0, Rout[0], BusWires);
trin  tri_1 (R1, Rout[1], BusWires);
trin  tri_2 (R2, Rout[2], BusWires);
trin  tri_3 (R3, Rout[3], BusWires);
regn  reg_A (BusWires, Ain, Clock, A);

// alu
always @(AddSub, A, BusWires)
    if (!AddSub)
        Sum = A + BusWires;
    else
        Sum = A – BusWires;

regn  reg_G (Sum, Gin, Clock, G);
trin  tri_G (G, Gout, BusWires);

endmodule
```



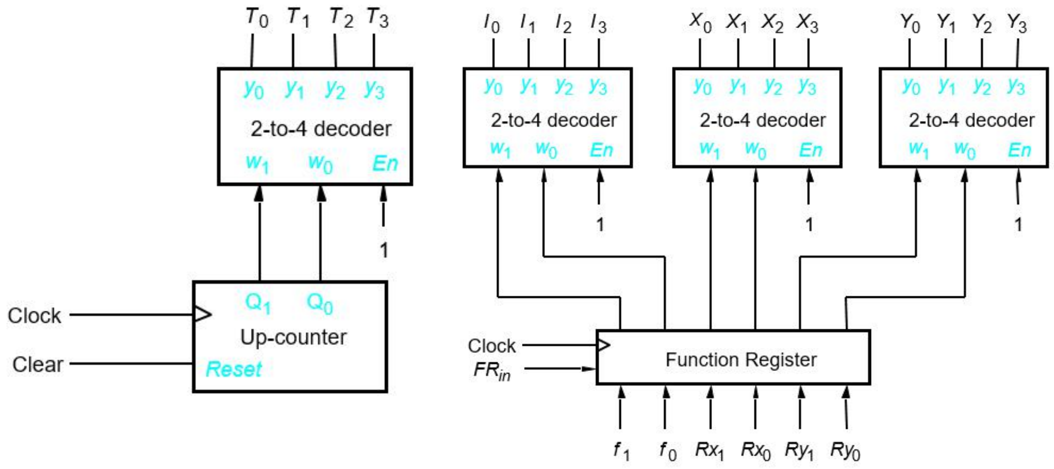


Figure 7.13. Code for the processor (Part *b*).

目录

1 同步数字系统结构

2 总线结构

3 简单处理器

4 乘法器

5 设计思想

例：乘法器设计

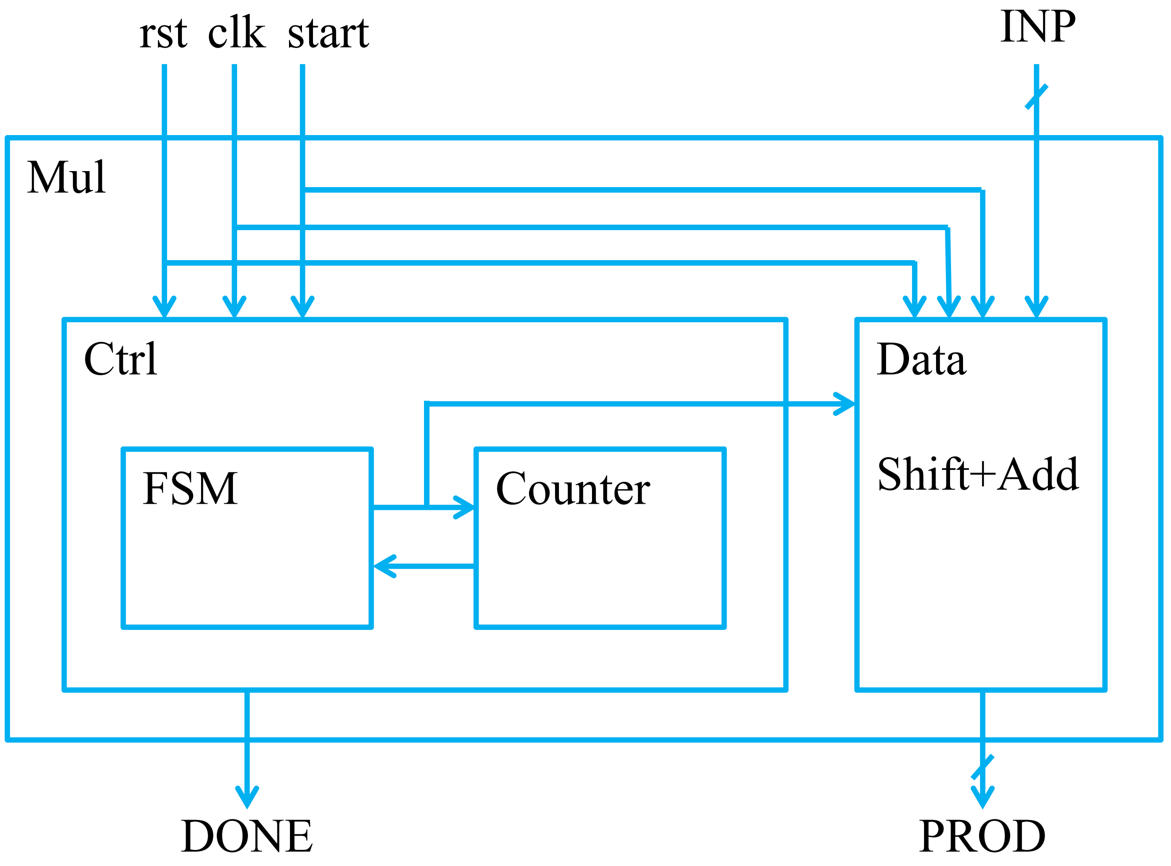
Decimal

```
    13
 ´  11
 ------
    13
   13
 ------
   143
```

Binary

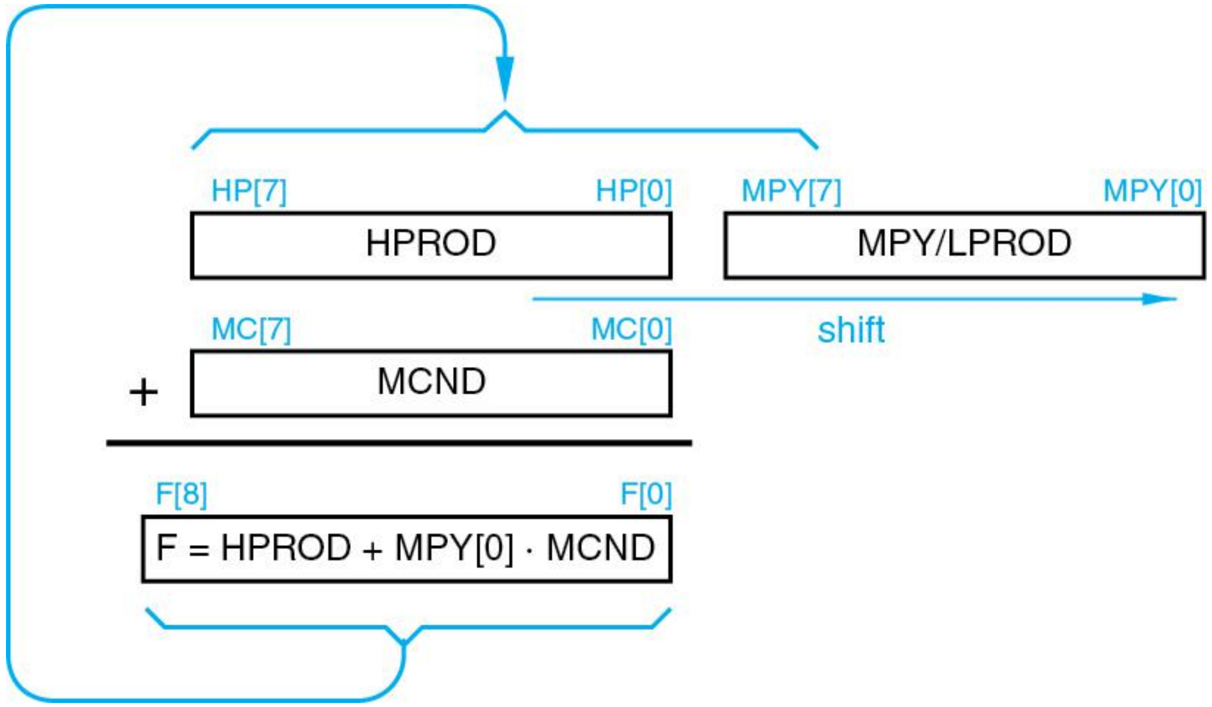
```
         1 1 0 1     Multiplicand
       ´ 1 0 1 1     Multiplier
     -------------
         1 1 0 1
       1 1 0 1
     0 0 0 0
   1 1 0 1
 -----------------
 1 0 0 0 1 1 1 1     Product
```

系统设计图



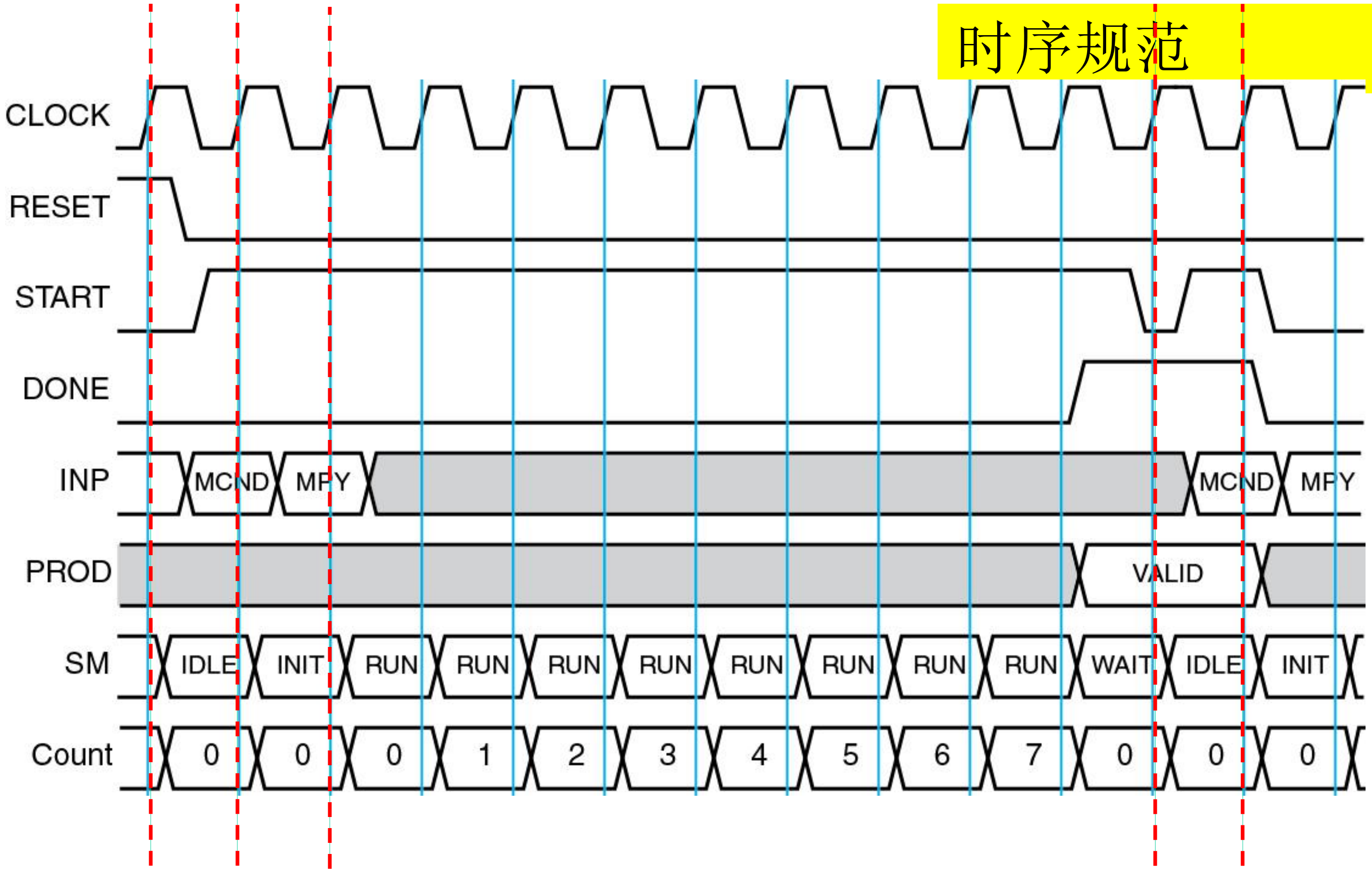
例：乘法器设计

功能说明


HP[7]
HP[0]
MPY[7]
MPY[0]
HPROD
MPY/LPROD
shift
MC[7]
MC[0]
+
MCND
F[8]
F[0]
F = HPROD + MPY[0] · MCND

例：乘法器设计

时序规范


CLOCK
RESET
START
DONE
INP
MCND
MPY
MCND
MPY
PROD
VALID
SM
IDLE
INIT
RUN
RUN
RUN
RUN
RUN
RUN
RUN
RUN
WAIT
IDLE
INIT
Count
0
0
0
1
2
3
4
5
6
7
0
0
0

例：乘法器设计

`include "Muldef.v"

```
parameter IDLE  = 2'b00,                // State-machine states
        INIT  = 2'b01,
        RUN   = 2'b10,
        WAIT  = 2'b11,
        SMmsb = 1,                      // SM state-reg size [SMmsb:SMlsb]
        SMlsb = 0;
parameter MPYwidth = 8,                 // Operand width
        MPYmsb   = MPYwidth-1,          // Index of operand MSB
        PRODmsb  = 2*MPYwidth-1,                // Index of product MSB
        MaxCnt   = MPYmsb,              // Number of shift-and-add steps
        CNTRmsb  = 2;                   // Step-counter size [CNTRmsb:0]
```

例：乘法器设计

状态机

```
module MPYsm ( RESET, CLK, START, MAX, SM );
  input RESET, CLK, START, MAX;
  output [1:0] SM;
  reg [1:0] Sreg, Snext;
  always @ (posedge CLK)                    // state memory
    if (RESET) Sreg <= IDLE;
    else Sreg <= Snext;
  always @ (START or MAX or Sreg)           // next-state logic
    case (Sreg)
      IDLE : if (START)              Snext <= INIT;
             else                    Snext <= IDLE;
      INIT :                         Snext <= RUN;
      RUN : if (MAX && ~START)               Snext <= IDLE;
            else if (MAX && START)           Snext <= WAIT;
            else                             Snext <= RUN;
      WAIT : if (~START)                     Snext <= IDLE;
             else                            Snext <= WAIT;
      default :                              Snext <= IDLE;
    endcase
  assign SM = Sreg;                         // output logic
endmodule
```

例：乘法器设计

计数器

```
module MPYcntr ( RESET, CLK, SM, MAX );
  input RESET, CLK;
  input [SMmsb:SMlsb] SM;
  output MAX;
  reg [CNTRmsb:0] Count;

  always @ (posedge CLK)
    if (RESET) Count <= 0;
    else if (SM==RUN) Count <= (Count + 1);
    else Count <= 0;

  assign MAX = (Count == MaxCnt);
Endmodule
```

例： 乘法器设计

控制单元

```
module MPYctrl ( RESET, CLK, START, DONE, SM );
  input RESET, CLK, START;
  output reg DONE;
  output [SMmsb:SMlsb] SM;
  wire MAX;
  wire [SMmsb:SMlsb] SMi;

  MPYsm   U1 ( RESET, CLK, START, MAX, SMi );
  MPYcntr U2 ( RESET, CLK, SMi, MAX);
  always @ (posedge CLK)                  // DONE logic
    if (RESET) DONE <= 1'b0;
    else if ( ((SMi==RUN) && MAX) || (SMi==WAIT) ) DONE <= 1'b1;
    else DONE <= 1'b0;
  assign SM = SMi;                        // Output
endmodule
```

例：乘法器设计

数据单元

```
module MPYdata (RESET, CLK, START, INP, SM, PROD );
  input RESET, CLK, START;
  input [MPYmsb:0] INP;
  input [SMmsb:SMlsb] SM;
  output [PRODmsb:0] PROD;
  reg [MPYmsb:0] MPY, MCND, HPROD;
  wire [MPYmsb+1:0] F;

  always @ (posedge CLK)
    if (RESET)
      begin MPY  <= 0; MCND <= 0; HPROD <= 0; end
    else if ((SM==IDLE) && START)                // load MCND, clear HPROD
      begin MCND <= INP; HPROD <= 0; end
    else if (SM==INIT) MPY <= INP;               // load MPY
    else if (SM==RUN) begin                      // shift registers
      MPY <= {F[0], MPY[MPYmsb:1]};
      HPROD <= F[(MPYmsb+1):1];  end

  assign F = (MPY[0]) ? ({1'b0, HPROD} + {1'b0, MCND}) : {1'b0, HPROD};
  assign PROD = {HPROD, MPY};
endmodule
```

例：乘法器设计

顶层模块

```
module MPY8x8 (RESET, CLK, START, INP, DONE, PROD );

  input RESET, CLK, START;
  input [MPYmsb:0] INP;
  output DONE;
  output [PRODmsb:0] PROD;
  wire [SMmsb:SMlsb] SM;

  MPYdata U1 ( RESET, CLK, START, INP,SM, PROD );
  MPYctrl U2 ( RESET, CLK, START, DONE, SM );
endmodule
```

目录

1 同步数字系统结构

2 总线结构

3 简单处理器

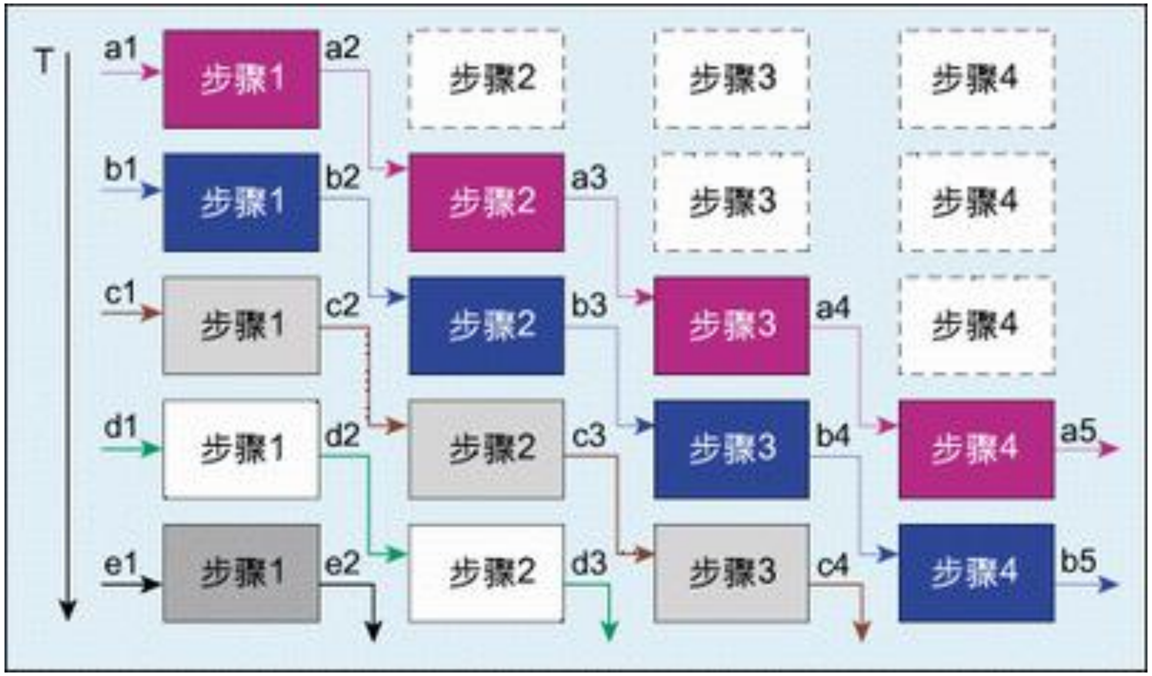
4 乘法器

5 设计思想

流水线操作的概念

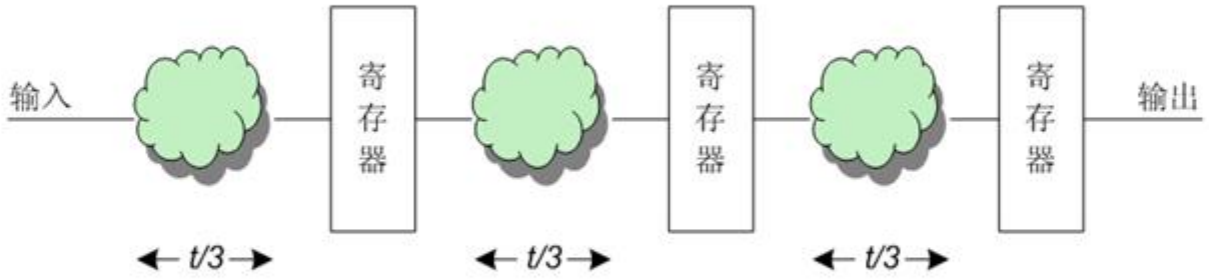
•如果某个设计的处理流程分为若干步骤，而且整个数据处理是“单流向”的。

•即没有反馈或者迭代运算，前一个步骤的输出是下一个步骤的输入，则可以考虑采用流水线设计方法来提高系统的工作频率。



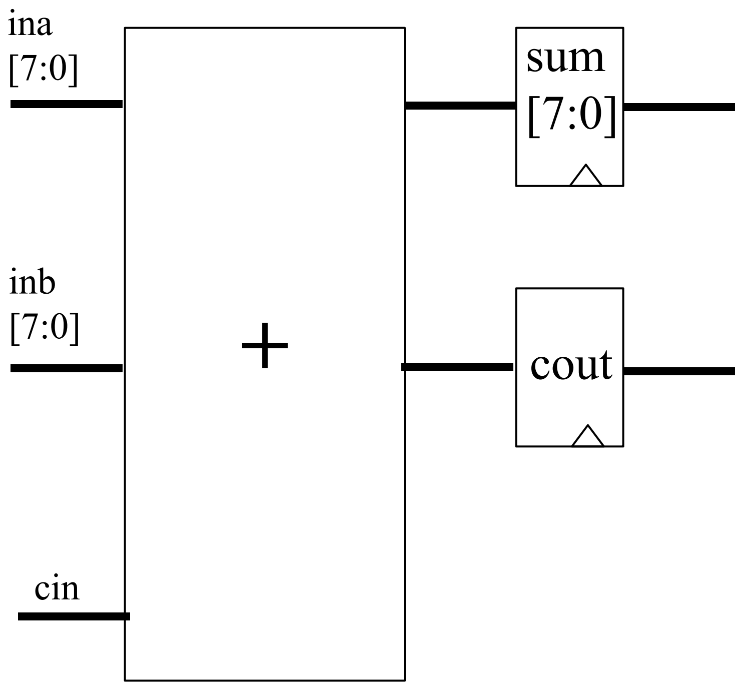
流水线操作的概念

- 如某个复杂逻辑功能的实现需较长的延时，可将其分解为几个（如3个）步骤来实现，每一步的延时变小，在各步间加入寄存器，以暂存中间结果，这样可大大提高整个系统的最高工作频率。

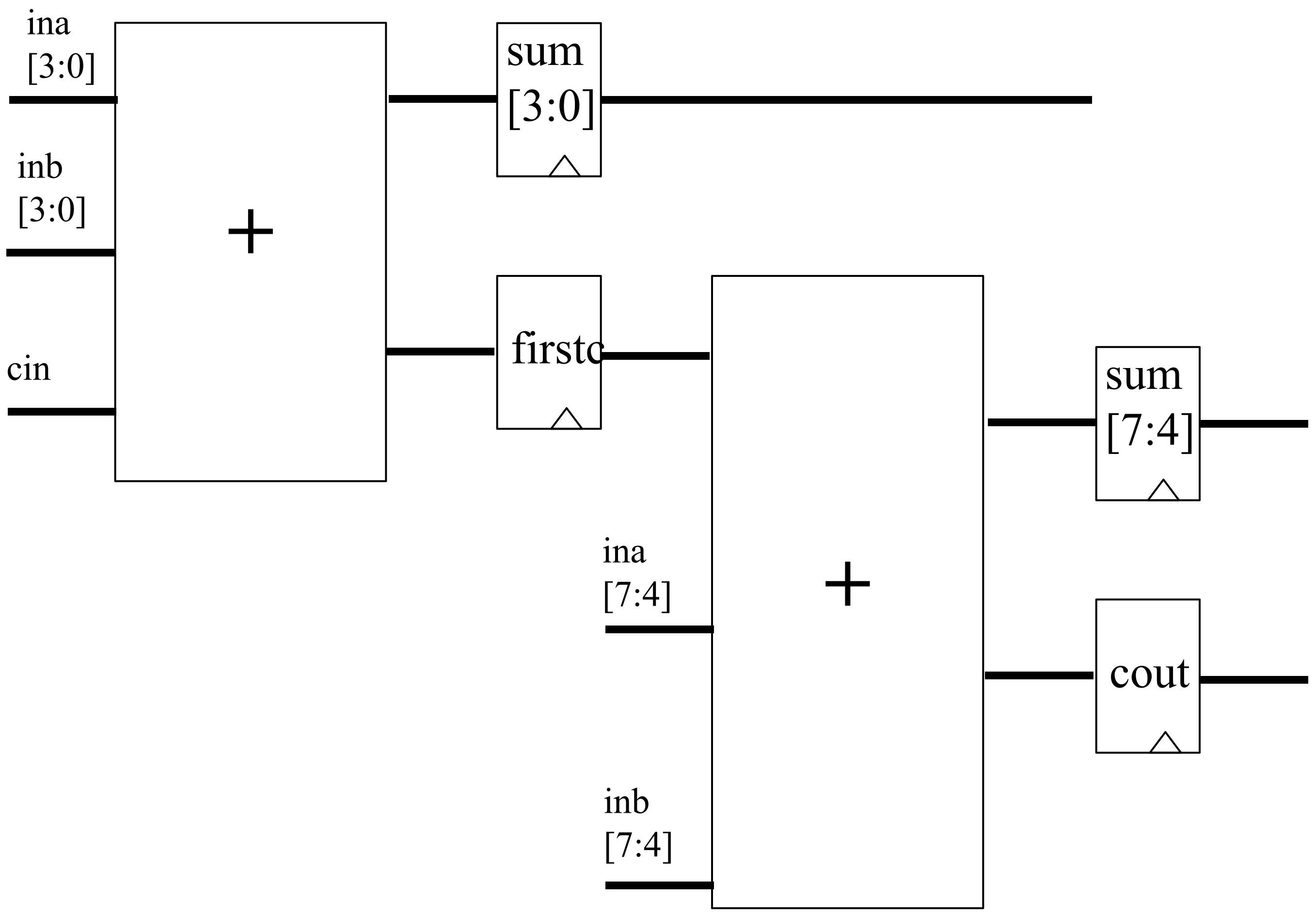


流水线操作的概念示意图

非流水线方式8位全加器

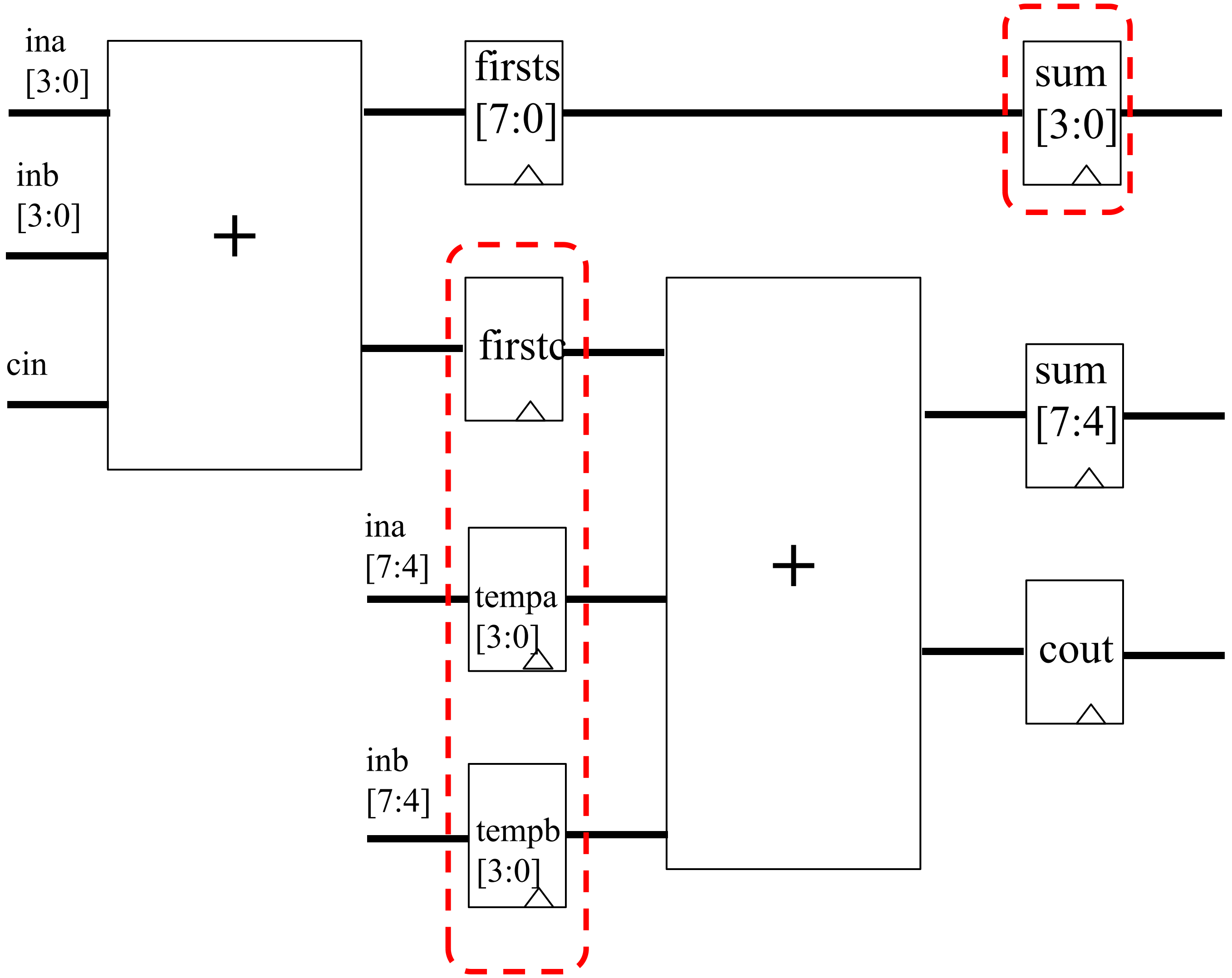


两级流水实现的8位加法器



这个实现有什么问题吗？

两级流水实现的8位加法器

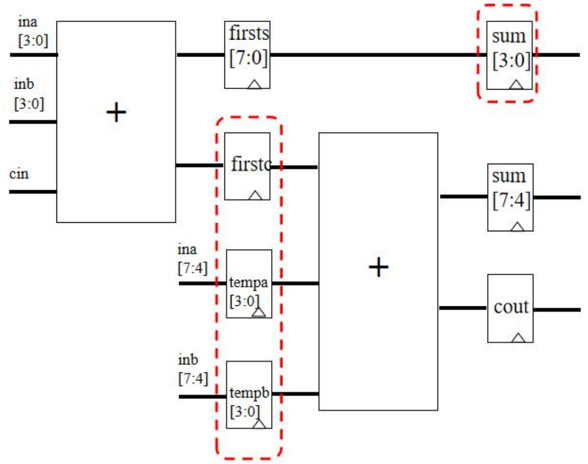


非流水线方式8位全加器

```
 module adder8(cout,sum,ina,inb,cin,clk);
 input[7:0] ina,inb; input cin,clk; output[7:0] sum;
output cout;
 reg[7:0] tempa,tempb,sum; reg cout,tempc;

 always @(posedge clk)
 begin
     tempa=ina;tempb=inb;tempc=cin;
 end
         //输入数据锁存
 always @(posedge clk)
 begin
         {cout,sum}=tempa+tempb+tempc;
 end
 endmodule
```

两级流水实现的8位加法器

```
module adder_pipe2(cout,sum,ina,inb,cin,clk);
input[7:0] ina,inb; input cin,clk; output reg[7:0] sum;
output reg cout; reg[3:0] tempa,tempb,firsts; reg firstc;

always @(posedge clk)
begin
   {firstc,firsts}=ina[3:0]+inb[3:0]+cin;
   tempa=ina[7:4];
   tempb=inb[7:4];
end

always @(posedge clk)
begin
   {cout,sum[7:4]}=tempa+tempb+firstc;
   sum[3:0]=firsts;
end
endmodule
```



将8位数每四位分2次相加，形成两级流水线运算过程。

四级流水线实现的8位加法器

$\boxed{76}$ $\boxed{54}$ $\boxed{32}$ $\boxed{10}$

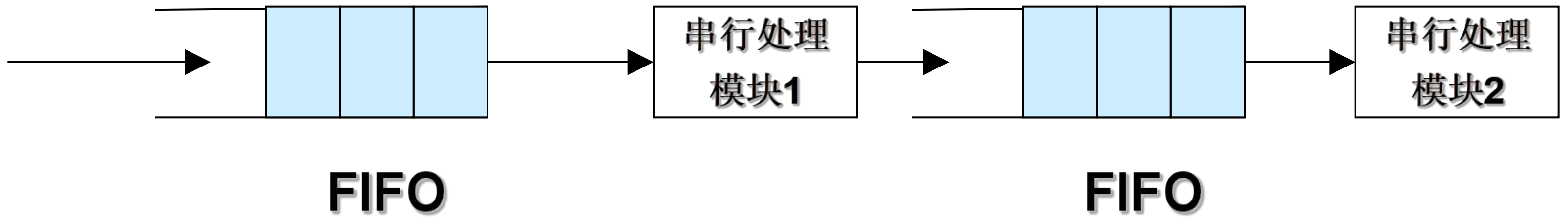
```
module  pipeline(cout,sum,ina,inb,cin,clk);
output[7:0]  sum;output  cout;
input[7:0] ina,inb;input  cin,clk; reg[7:0] tempa,tempb,sum;
reg  tempci,firstco,secondco,thirdco, cout;
reg[1:0]  firsts, thirda,thirdb;
reg[3:0]  seconda, secondb, seconds; reg[5:0]  firsta, firstb, thirds;

always @(posedge clk)
begin tempa=ina;  tempb=inb;  tempci=cin; end              //输入数据缓存
always @(posedge clk)
begin {firstco,firsts}=tempa[1:0]+tempb[1:0]+tempci;  //第一级加（低2位）
firsta=tempa[7:2]; firstb=tempb[7:2];          //未参加计算的数据缓存
end
always @(posedge clk)
begin {secondco,seconds}={firsta[1:0]+firstb[1:0]+firstco,firsts};
seconda=firsta[5:2]; secondb=firstb[5:2];       //数据缓存
end
always @(posedge clk)
begin {thirdco,thirds}={seconda[1:0]+secondb[1:0]+secondco,seconds};
thirda=seconda[3:2];thirdb=secondb[3:2];                 //数据缓存
end
always @(posedge clk)
begin
{cout,sum}={thirda[1:0]+thirdb[1:0]+thirdco,thirds}; //第四级加（高两位相加）
end  endmodule
```

将8位数每两位分4次相加，形成四级流水线运算过程。

串行设计

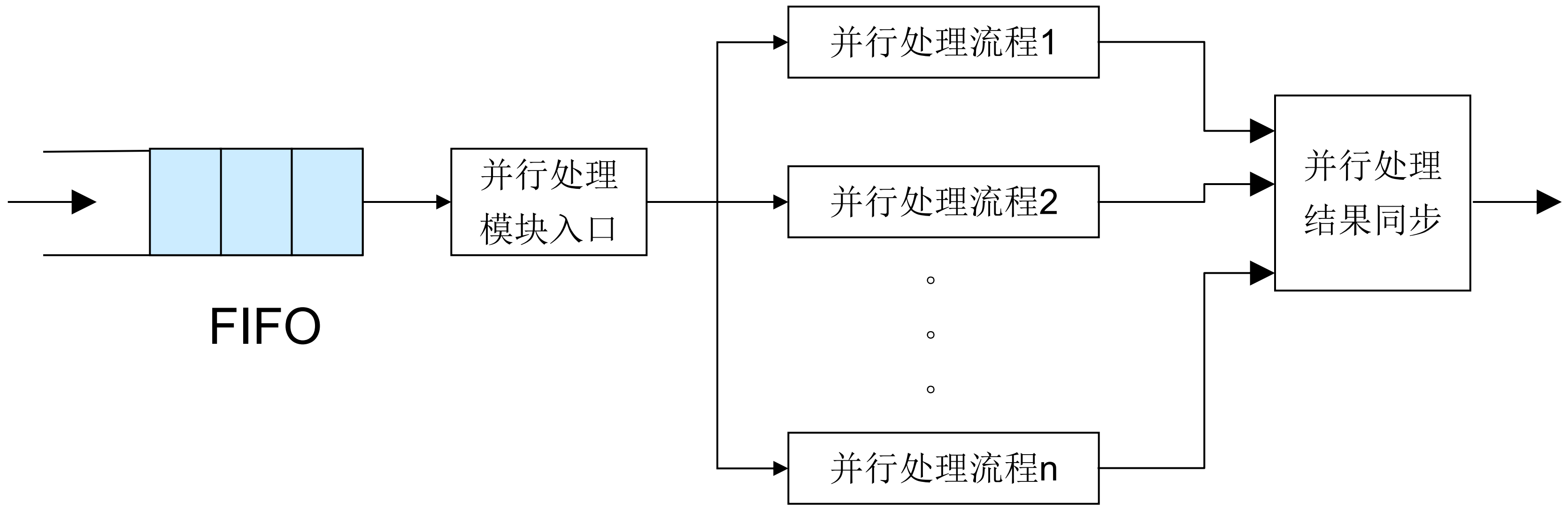
串行设计是最常见的一种设计；

- 当一个功能模块对输入的处理是分步骤进行的，并且后一个步骤只依赖前一个步骤的结果时，功能模块的设计就需要采用串行设计的思想。
- 一般采用FIFO（First In First Out）进行缓冲处理



并行设计

并行设计采用几个处理流程同时处理到达的负载，提高处理的效率，并行处理要求这些处理之间是独立的。



考 试

- 笔试（选择、简答、分析、设计）
- 以PPT知识点为主线展开复习
- 16周周四第4大节
  - 逸夫楼204：计181+计182
  - 逸夫楼205：物联 + 重修
  - 逸夫楼206：信安
  - 逸夫楼207：计183+计184+辅修