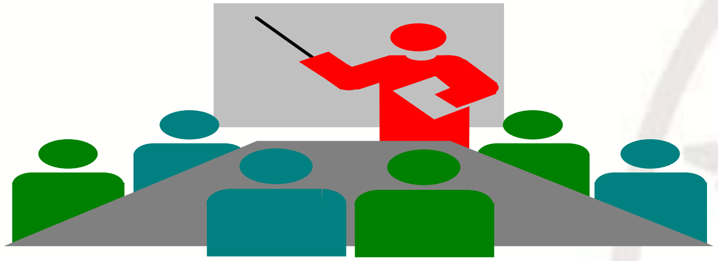


数字逻辑设计

LOGIC and Computer Design Fundamentals

CHAPTER 6

Register & Register Transfers

Registers, Microoperations & Implementations (part I)

施青松
Asso. Prof. Shi Qingsong
College of Computer Science and Technology, Zhejiang University
zjsqs@zju.edu.cn

Overview

- **Part 1-Registers, Microoperations and Implementations**
  - Registers and load enable
  - Register transfer operations
  - Microoperations - arithmetic, logic, and shift
  - Microoperations on a single register
    - Multiplexer-based transfers
    - Shift registers
- **Part 2 - Counters, Register Cells, Buses, & Serial Operations**
- **Part 3 – Control of Register Transfers**

Course Outline

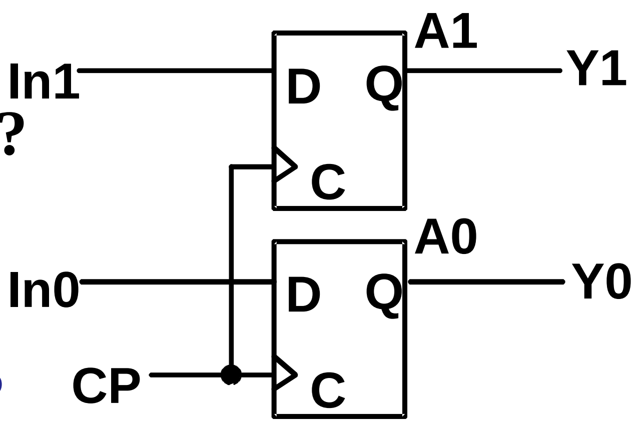
- What is Registers
- Register transfer operations
- Microoperations arithmetic & logic
- Microoperations on a single register

What is Registers

- **Register**
  - a collection of binary storage elements
  - In theory, a register is sequential logic which can be defined by a state table
  - More often, think of a register as storing a vector of binary values
- **Operating**
  - Frequently used to perform simple data storage and data movement and processing operations

Example: 2-bit Register

- How many states are there?
- How many input combinations?
  Output combinations?
- What is the *output function*?
- What is the *next state function*?
- Moore or Mealy?



State Table:

Current State A1 A0	Next State A1(t+1) A0(t+1) For In1 In0 = 00 01 10 11	Output (=A1 A0) Y1 Y0
0 0	00 01 10 11	0 0
0 1	00 01 10 11	0 1
1 0	00 01 10 11	1 0
1 1	00 01 10 11	1 1

- What are the *quantities* above for an *n*-bit register?

Register Design Models

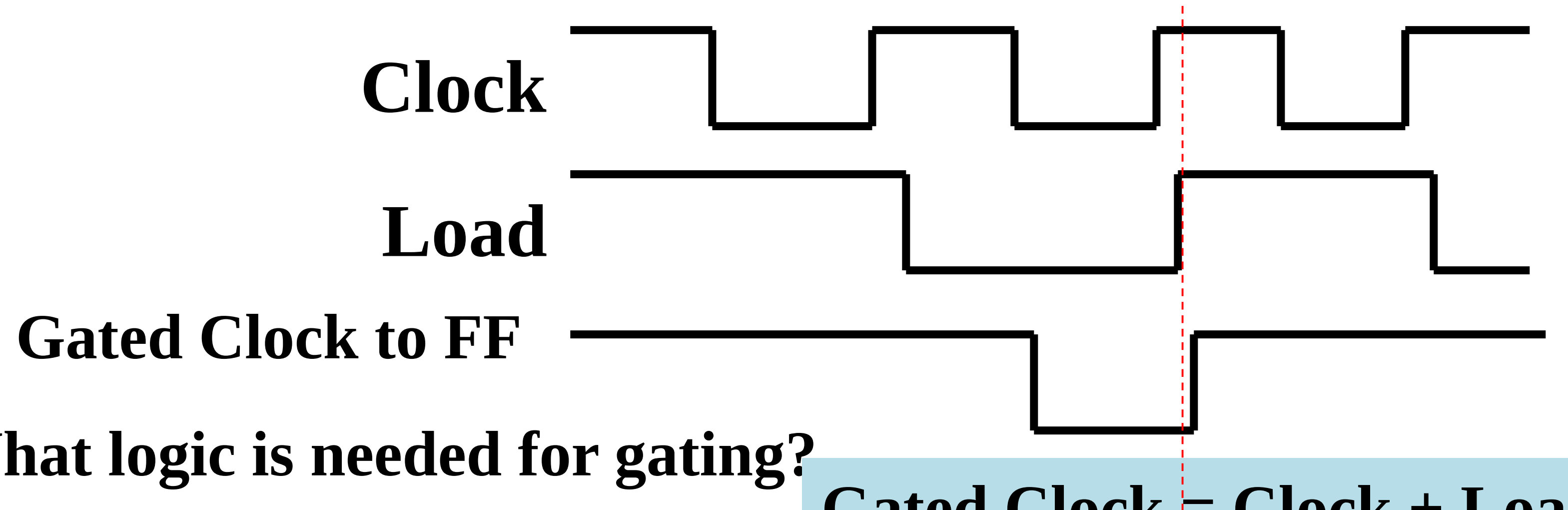
- **Due to the large numbers of states and input combinations as *n* becomes large, the state diagram/state table model is not feasible!**
- **What are methods we can use to design registers?**
  - Add predefined combinational circuits to registers
    - Example: To count up, connect the register flip-flops to an incrementer
  - Design *individual cells* using the state diagram/state table model and combine them into a register
    - A 1-bit cell has just two states
    - Output is usually the state variable

Register Storage

- **Expectations:**
  - A register can store information for multiple clock cycles
  - To "store" or "load" information should be controlled by a signal
- **Reality:**
  - A D flip-flop register loads information on every clock cycle
- **Realizing expectations:**
  - Use a signal to *block* the clock to the register,
  - Use a signal to control *feedback* of the output of the register back to its inputs, or
  - Use other SR or JK flip-flops, that for (0,0) applied, store their state
- **Load is a frequent name for the signal that controls register storage and loading**
  - Load = 1: Load the values on the data inputs
  - Load = 0: Store the values in the register

Clock Skew：Registers with Clock Gating

- The $\overline{Load}$ signal enables the clock signal to pass through if 1 and prevents the clock signal from passing through if 0.
- Example: For Positive Edge-Triggered or Negative Pulse Master-Slave Flip-flop:



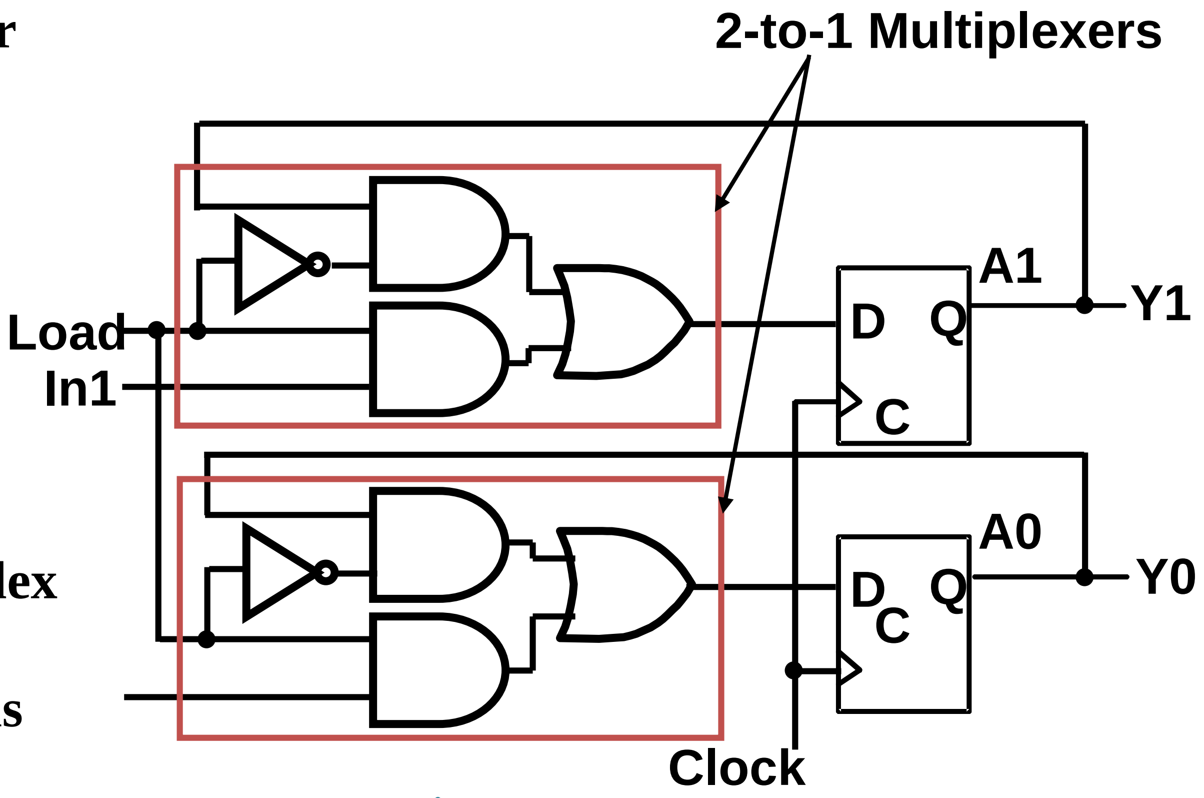
- What logic is needed for gating?
- What is the problem?

Gated Clock = Clock + Load

Clock Skew of gated clocks with respect to clock or each other

Registers with Load-Controlled Feedback

- **A more reliable way to selectively load a register:**
  - Run the clock continuously, and
  - Selectively use a load control to change the register contents.
- **Example: 2-bit register with Load Control:**
- **For Load = 0, loads register contents (hold current values)**
- **For Load = 1, loads input values (load new values)**
- **Hardware more complex than clock gating, but free of timing problems**

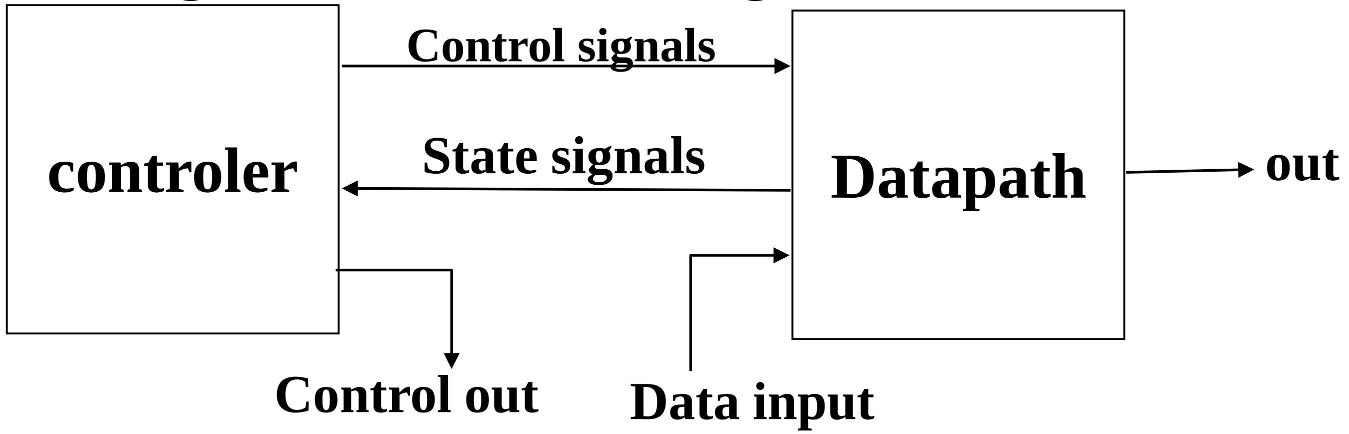


Course Outline

- What is Registers
- Register transfer operations
- Microoperations arithmetic & logic
- Microoperations on a single register

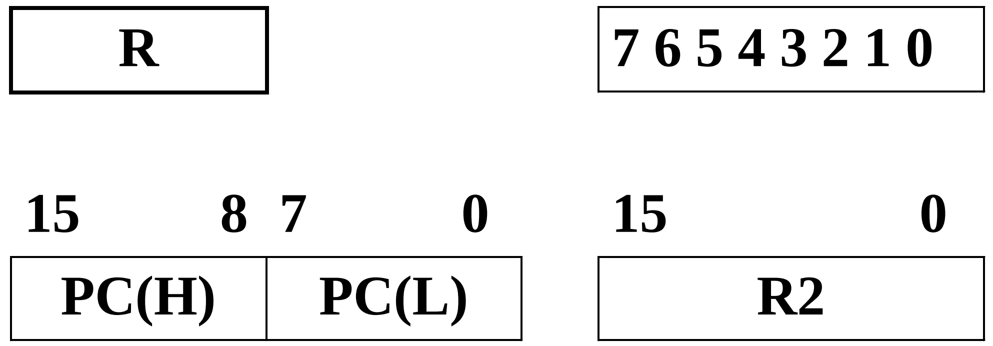
Register Transfer Operations

- Register Transfer Operations – The movement and processing of data stored in registers



- Three basic components:
  - Set of registers
  - Operations
  - control of operations
- **Elementary Operations -load, count, shift, add, bitwise "OR", etc.**
  - Elementary operations called *microoperations*

Register Notation



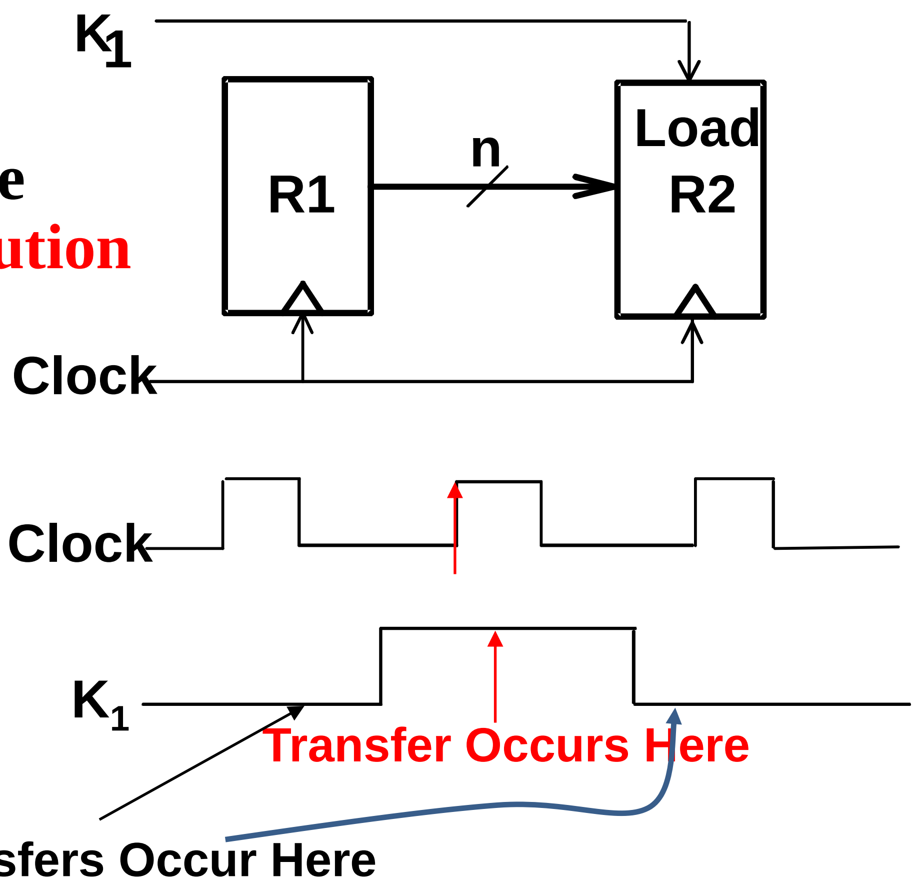
- Letters and numbers – denotes a register (ex. R2, PC, IR)
- Parentheses ( ) – denotes a range of register bits
  - ex. R1(1), PC(7:0), PC(L)
- Arrow (←) – denotes data transfer
  - ex. R1 ← R2, PC(L) ← R0
- Comma – separates parallel operations
- Brackets [ ] – Specifies a memory address
  - ex. R0 ← M[AR], R3 ← M[PC]

Conditional Transfer

- If (K1 =1) then (R2 ← R1) is shortened to

  K1: (R2 ← R1)

  where K1 is a control variable specifying a conditional execution of the microoperation.



Microoperations

- **Logical Groupings:**
  - Transfer - move data from one register to another
  - Arithmetic - perform arithmetic on data in registers
  - Logic - manipulate data or use bitwise logical operations
  - Shift - shift data in registers

Arithmetic operations	**Logical operations**
+ Addition	**$\vee$ Logical OR**
– Subtraction	**$\wedge$ Logical AND**
*** Multiplication**	**$\oplus$ Logical Exclusive OR**
/ Division	**$\overline{}$ Not**

Example Microoperations

- Add the content of R1 to the content of R2 and place the result in R1.

  $R1 \leftarrow R1 + R2$

- Multiply the content of R1 by the content of R6 and place the result in PC.

  $PC \leftarrow R1 * R6$

- Exclusive OR the content of R1 with the content of R2 and place the result in R1.

  $R1 \leftarrow R1 \oplus R2$

Example Microoperations (Continued)

- Take the 1's Complement of the contents of R2 and place it in the PC.

$$PC \leftarrow \overline{R2}$$

- On condition K1 **OR** K2, the content of R1 is **Logic bitwise OR** with the content of R3 and the result placed in R1.

$$(K1 + K2):\ R1 \leftarrow R1 \vee R3$$

- NOTE
  - "+" (as in $K_1 + K_2$) means "OR"
  - In $R1 \leftarrow R1 + R3$, + means "plus."

Control Expressions

- The control expression for an operation appears to the left of the operation and is separated from it by a colon
- Control expressions specify the logical condition for the operation to occur
- Control expression values of:
  - Logic "1" -- the operation occurs.
  - Logic "0" -- the operation is does not occur.

- Example:

  $\overline{X}\,K1:\ R1 \leftarrow R1 + R2$

  $X\,K1:\ R1 \leftarrow R1 + \overline{R2} + 1$
- Variable K1 enables the add or subtract operation.
- If X =0, then $\overline{X}$ =1 so $\overline{X}$ K1 = 1, activating the addition of R1 and R2.
- If X = 1, then X K1 = 1, activating the addition of R1 and the two's complement of R2 (subtract).

Course Outline

- What is Registers
- Register transfer operations
- Microoperations arithmetic & logic
- Microoperations on a single register

Arithmetic Microoperations

- From Table 7-3:

Symbolic Designation	Description
$R0 \leftarrow R1 + R2$	Addition
$R0 \leftarrow \overline{R1}$	Ones Complement
$R0 \leftarrow \overline{R1} + 1$	Two's Complement
$R0 \leftarrow R2 + \overline{R1} + 1$	R2 minus R1 (2's Comp)
$R1 \leftarrow R1 + 1$	Increment (count up)
$R1 \leftarrow R1 - 1$	Decrement (count down)

- Note that any register may be specified for source 1, source 2, or destination.
- These simple microoperations operate on the whole word

Logical Microoperations

- From Table 7-4:

Symbolic Designation	Description
R0 $\leftarrow \overline{R1}$	Bitwise NOT
R0 $\leftarrow$ R1 $\vee$ R2	Bitwise OR (sets bits)
R0 $\leftarrow$ R1 $\wedge$ R2	Bitwise AND (clears bits)
R0 $\leftarrow$ R1 $\oplus$ R2	Bitwise EXOR (complements bits)

Logical Microoperations (continued)

- Let R1 = 10101010, and R2 = 11110000
- Then after the operation, R0 becomes:

R0	Operation
01010101	$R0 \leftarrow \overline{R1}$
11111010	$R0 \leftarrow R1 \vee R2$
10100000	$R0 \leftarrow R1 \wedge R2$
01011010	$R0 \leftarrow R1 \oplus R2$

Shift Microoperations

- From Table 7-5:
- Let R2 = 11001001
- Then after the operation, R1 becomes:

Symbolic Designation	Description
R1 ← sl R2	Shift Left
R1 ← sr R2	Shift Right

R1	Operation
10010010	R1 ← sl R2
01100100	R1 ← sr R2

- Note: These shifts "zero fill". Sometimes a separate flip-flop is used to provide the data shifted in, or to “catch” the data shifted out.
- Other shifts are possible (rotates, arithmetic) (see Chapter 10).

Transfers and Assigning

- **assign(流描述)**

  assign r0 = {0, r0[15:1]}; //Right Shift

- **always(过程描述语句)**
  - 对于复杂的电路行为，用assign赋值无法描述
  - always可以用来描述复杂的信号传输过程
  - 赋值必须是reg变量，但结果由电路综合决定，不一定是reg

```
Always @ (触发表达式)begin
        //过程赋值
        //条件语句
        ……
end
```

- 触发事件表达式有：
  - 组全电路使用变量：

    (a)、(a or b)，可用“*”代替
  - 时序电路用时钟边沿：

    (posedge clk) //上升沿

    (negdge clk) //下降沿

    (posedge clk or negdge res) //上下边沿

  - timing
    - 非阻塞赋值：<=，触发边沿同时赋值(并行)
    - 阻塞赋值： =，先赋值再同步，较难把握(组合电路，assign)

Course Outline

- What is Registers
- Register transfer operations
- Microoperations arithmetic & logic
- Microoperations on a single register

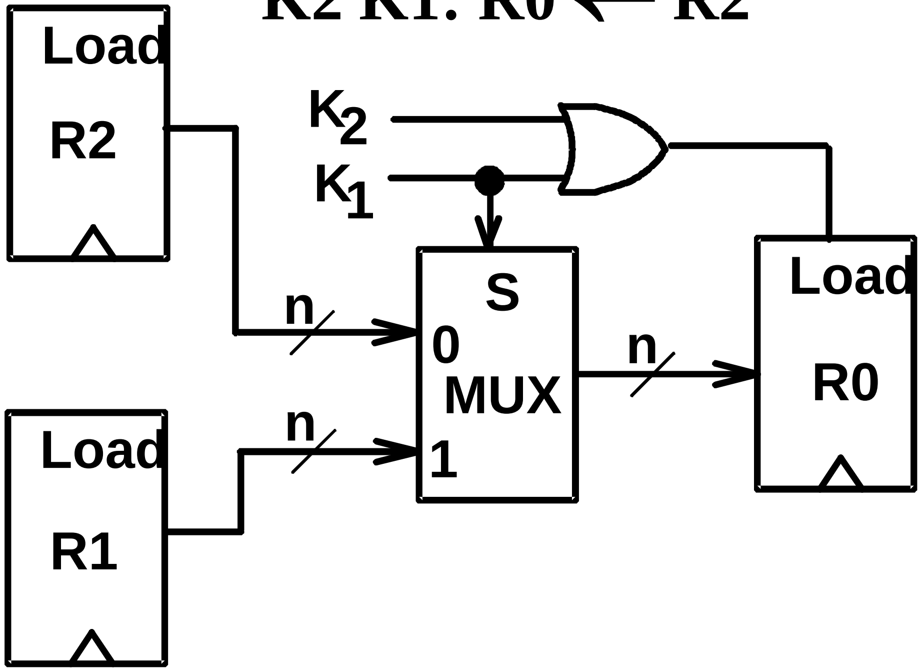
Register Transfer Structures

- **Multiplexer-Based Transfers - Multiple inputs are selected by a multiplexer dedicated to the register**
- **Bus-Based Transfers - Multiple inputs are selected by a shared multiplexer driving a bus that feeds inputs to multiple registers**
- **Three-State Bus - Multiple inputs are selected by 3-state drivers with outputs connected to a bus that feeds multiple registers**
- **Other Transfer Structures - Use multiple multiplexers, multiple buses, and combinations of all the above**

Multiplexer-Based Transfers

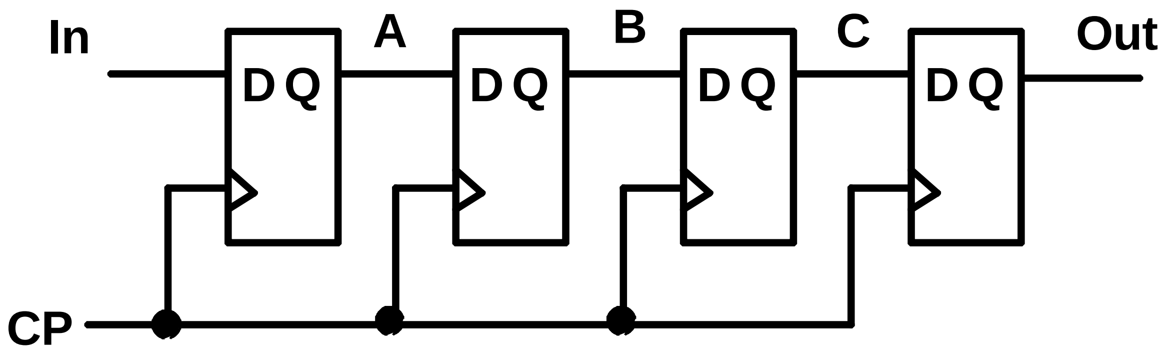
- Multiplexers connected to register inputs produce flexible transfer structures (Note: Clocks are omitted for clarity)
- The transfers are: K1: R0 $\leftarrow$ R1

  $K2\,\overline{K1}$: R0 $\leftarrow$ R2



Shift Registers

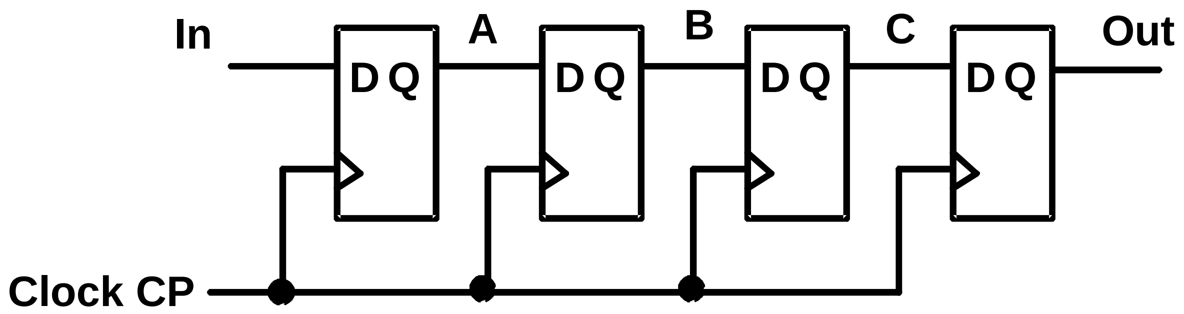
- Shift Registers move data laterally within the register toward its MSB or LSB position
- In the simplest case, the shift register is simply a set of D flip-flops connected in a row like this:



- Data input, In, is called a *serial input* or the *shift right input*.
- Data output, Out, is often called the *serial output*.
- The vector (A, B, C, Out) is called the *parallel output*.

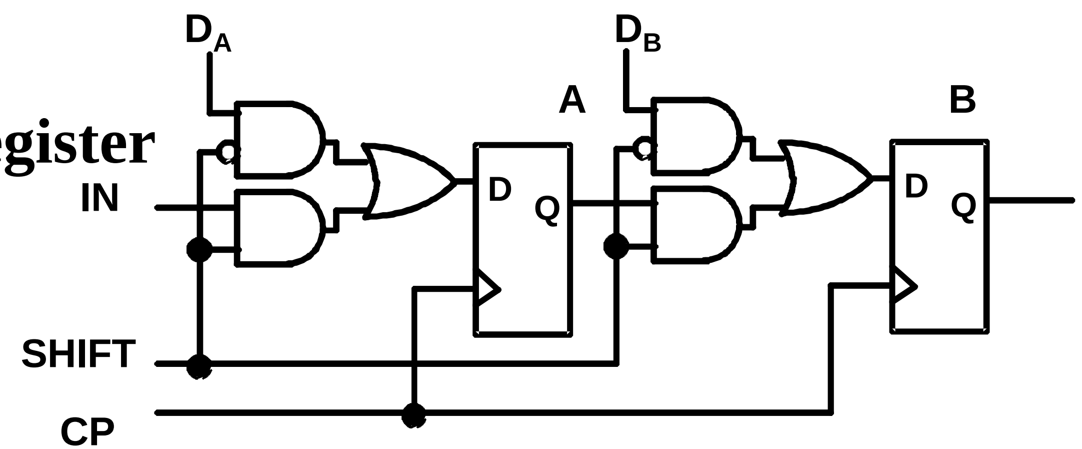
Shift Registers Running

- The behavior of the serial shift register is given in the listing on the lower right
- T0 is the register state just before the first clock pulse occurs
- T1 is after the first pulse and before the second.
- Initially unknown states are denoted by "?"
- Complete the last three rows of the table



CP	In	A	B	C	Out
T0	0	?	?	?	?
T1	1	0	?	?	?
T2	1	1	0	?	?
T3	0	1	1	0	?
T4	1				
T5	1				
T6	1				

Parallel Load Shift Registers

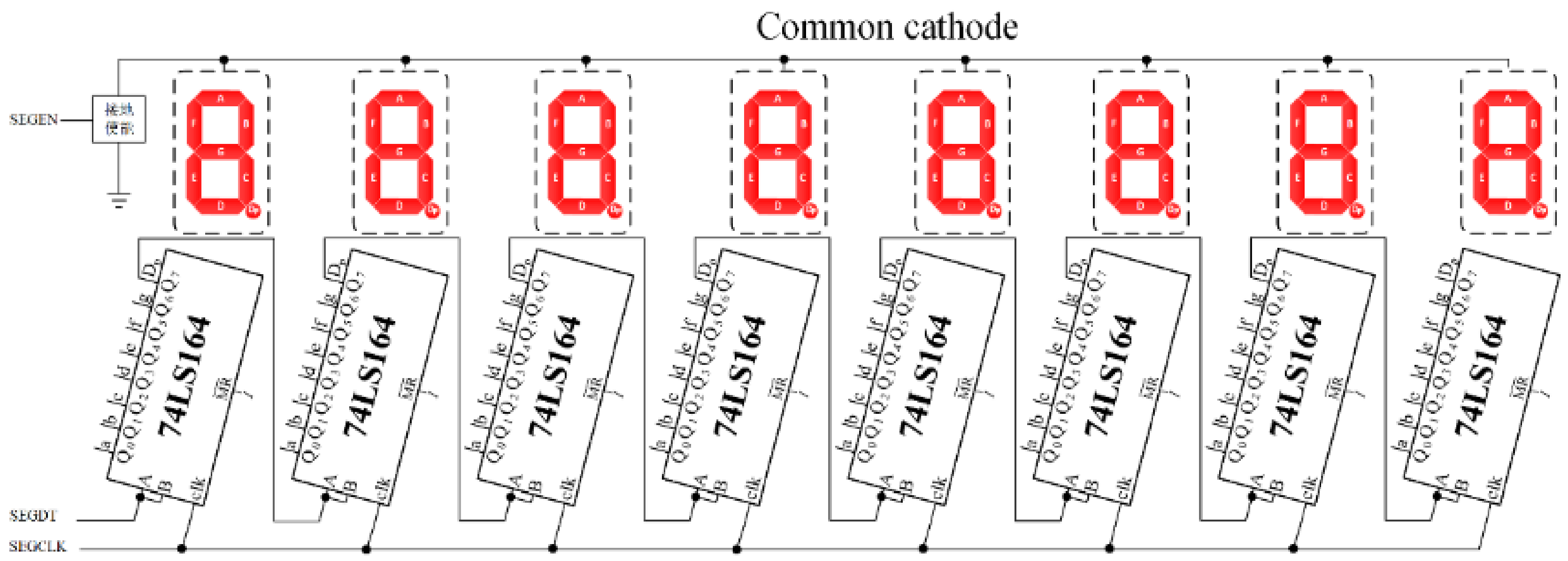


- By adding a mux between each shift register stage, data can be shifted or loaded
- If SHIFT is low, A and B are replaced by the data on D_A and D_B lines, else data shifts right on each clock.
- By adding more bits, we can make *n*-bit parallel load shift registers.
- A parallel load shift register with an added “hold” operation that stores data unchanged is given in Figure 7-10 of the text.

Shift Registers with Additional Functions

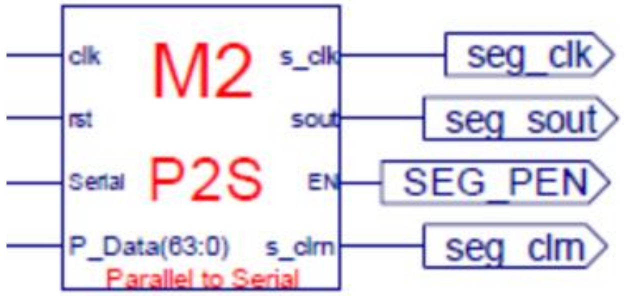
- By placing a 4-input multiplexer in front of each D flip-flop in a shift register, we can implement a circuit with shifts right, shifts left, parallel load, hold.
- Shift registers can also be designed to shift more than a single bit position right or left
- Shift registers can be designed to shift a variable number of bit positions specified by a variable called a *shift amount*.

Application of the shift register



通过P2S模块输出：

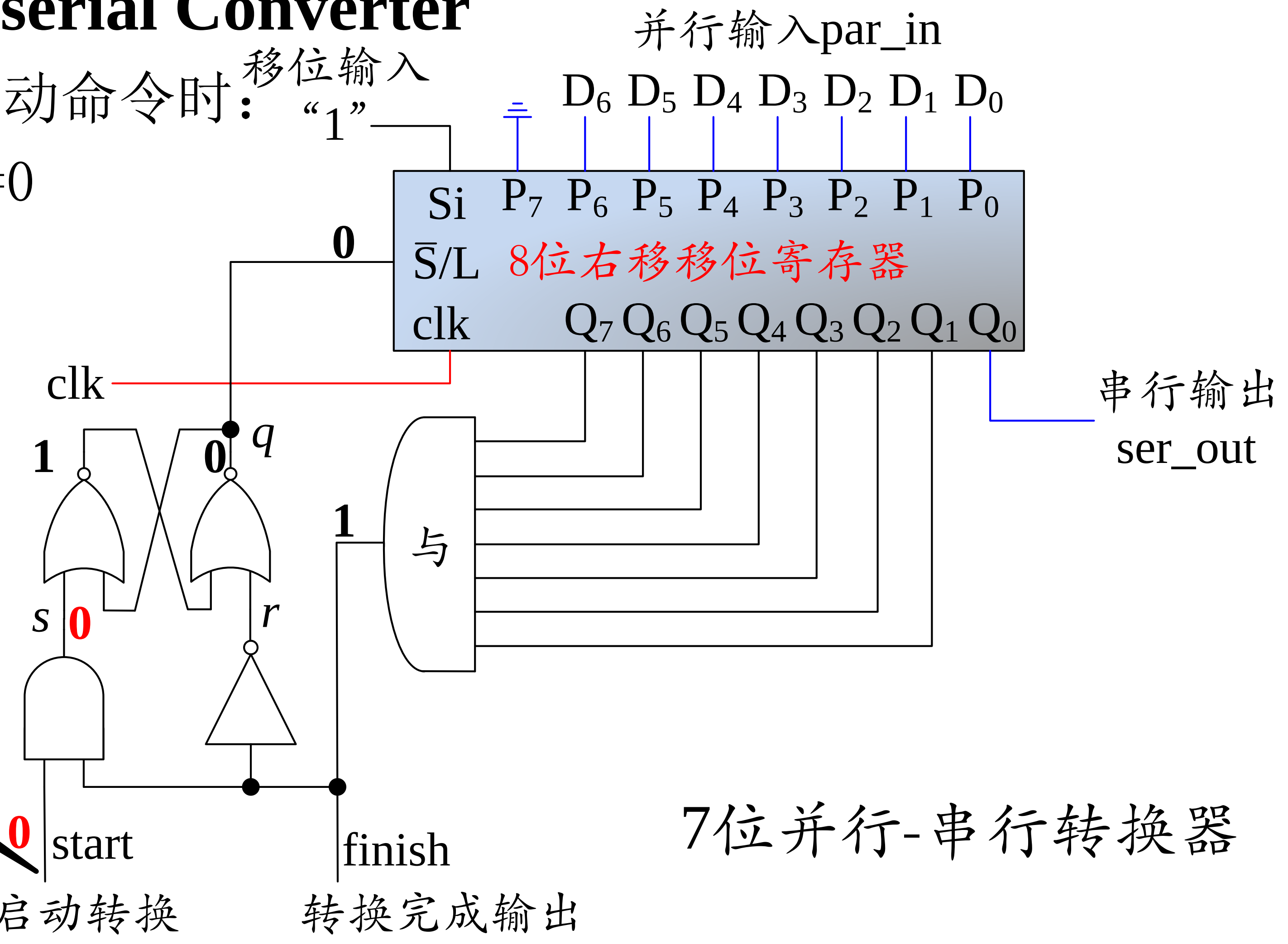
P_Data[63:0]=SEGMENT[63:0]



Synchronous serial transmission: Parallel-serial Converter

◎ Parallel-serial Converter

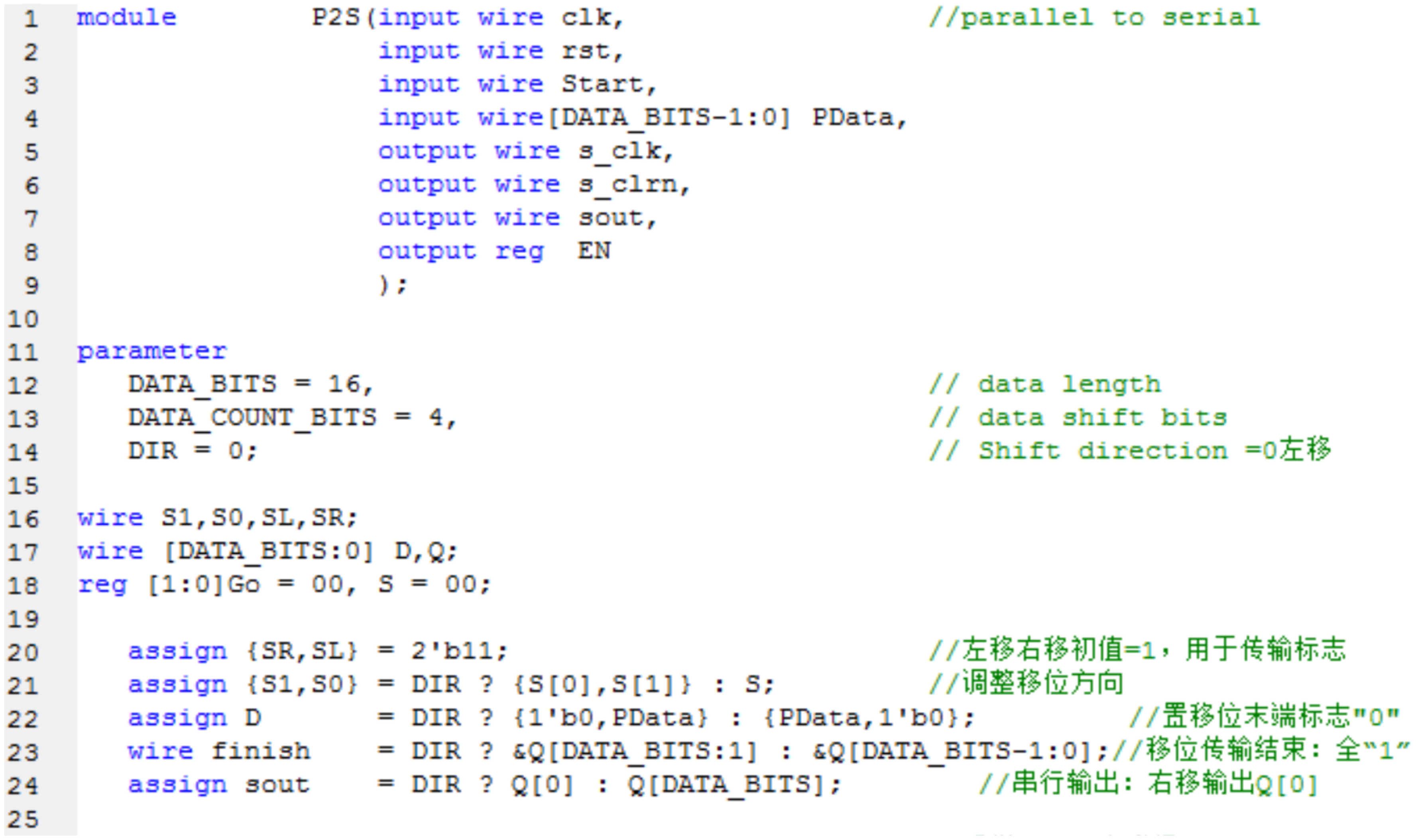
- 没有启动命令时：
  - Start=0
  - S/L=0



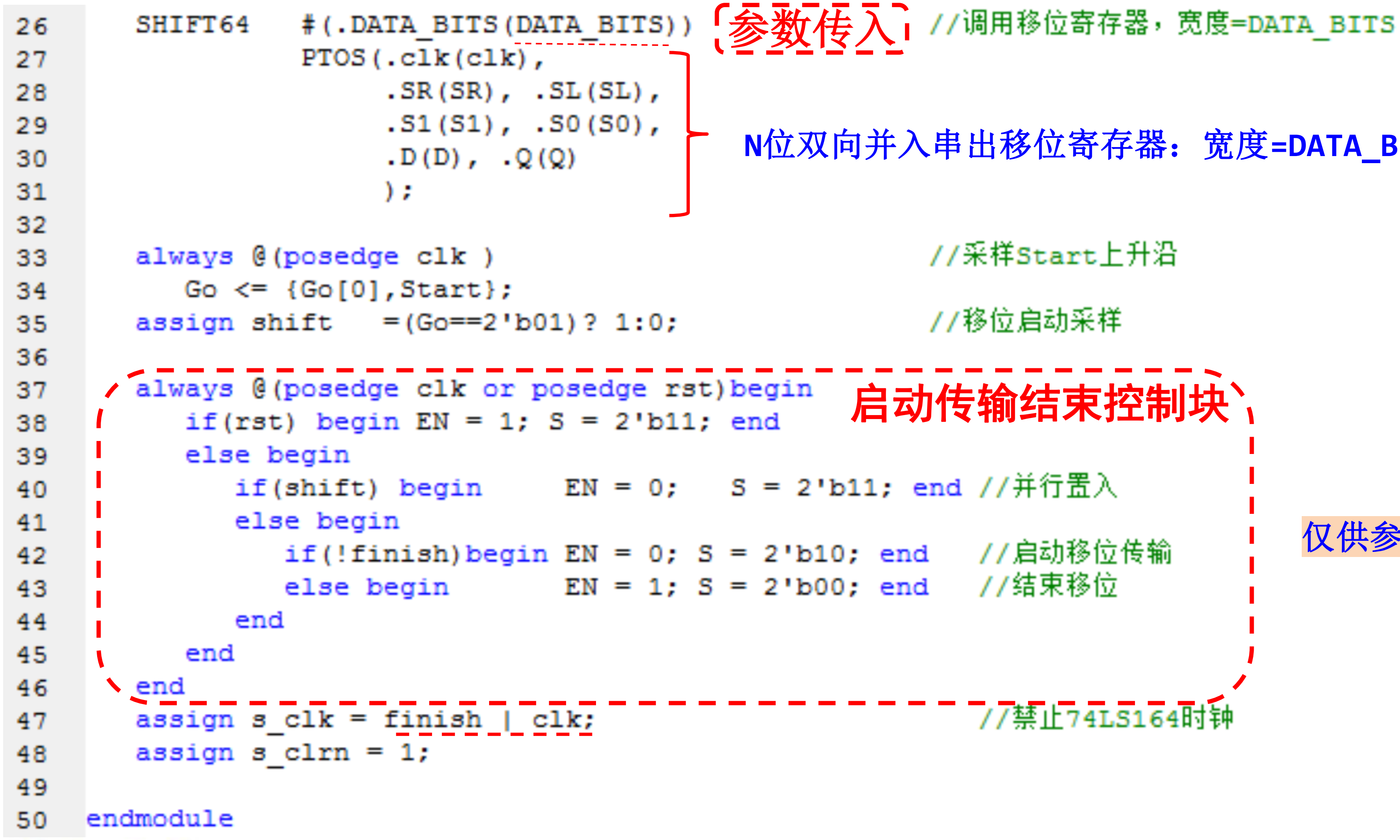
start=0

7位并行-串行转换器

移位寄存器器应用：P2S模块设计



```
module          P2S(input wire clk,                    //parallel to serial
                    input wire rst,
                    input wire Start,
                    input wire[DATA_BITS-1:0] PData,
                    output wire s_clk,
                    output wire s_clrn,
                    output wire sout,
                    output reg  EN
                    );

parameter
   DATA_BITS = 16,                                     // data length
   DATA_COUNT_BITS = 4,                                // data shift bits
   DIR = 0;                                            // Shift direction =0左移

wire S1,S0,SL,SR;
wire [DATA_BITS:0] D,Q;
reg [1:0]Go = 00, S = 00;

   assign {SR,SL} = 2'b11;                             //左移右移初值=1，用于传输标志
   assign {S1,S0} = DIR ? {S[0],S[1]} : S;             //调整移位方向
   assign D       = DIR ? {1'b0,PData} : {PData,1'b0};             //置移位末端标志"0"
   wire finish    = DIR ? &Q[DATA_BITS:1] : &Q[DATA_BITS-1:0];//移位传输结束: 全“1”
   assign sout    = DIR ? Q[0] : Q[DATA_BITS];              //串行输出: 右移输出Q[0]

```



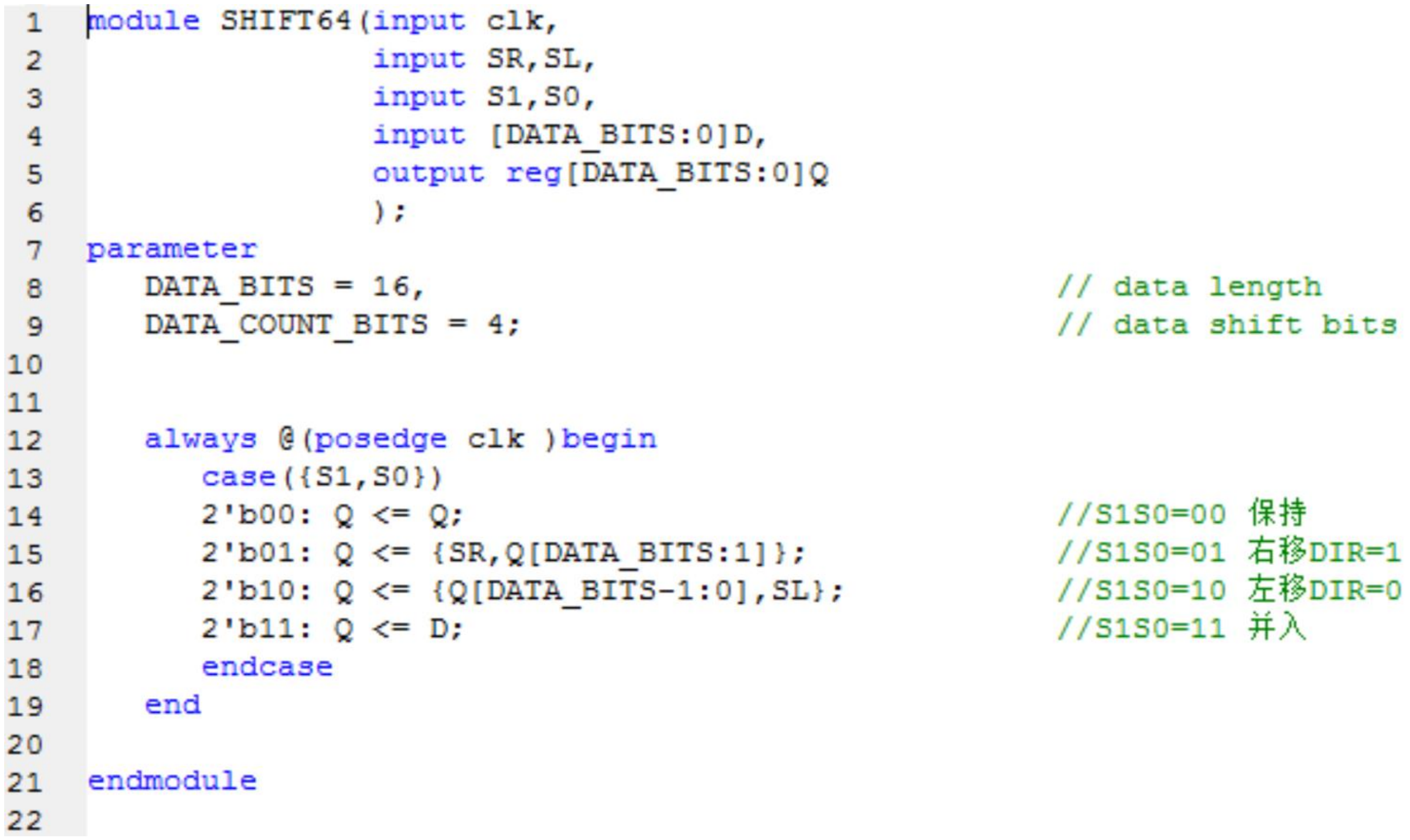
```
   SHIFT64   #(.DATA_BITS(DATA_BITS))                  //调用移位寄存器，宽度=DATA_BITS
             PTOS(.clk(clk),
                  .SR(SR), .SL(SL),
                  .S1(S1), .S0(S0),
                  .D(D), .Q(Q)
                  );

   always @(posedge clk )                              //采样Start上升沿
      Go <= {Go[0],Start};
   assign shift   =(Go==2'b01)? 1:0;                   //移位启动采样

   always @(posedge clk or posedge rst)begin
      if(rst) begin EN = 1; S = 2'b11; end
      else begin
         if(shift) begin     EN = 0;   S = 2'b11; end //并行置入
         else begin
            if(!finish)begin EN = 0; S = 2'b10; end   //启动移位传输
            else begin       EN = 1; S = 2'b00; end   //结束移位
         end
      end
   end
   assign s_clk = finish | clk;                          //禁止74LS164时钟
   assign s_clrn = 1;

endmodule
```

参数传入

N位双向并入串出移位寄存器：宽度=DATA_BITS

启动传输结束控制块

仅供参考



```verilog
module SHIFT64(input clk,
               input SR,SL,
               input S1,S0,
               input [DATA_BITS:0]D,
               output reg[DATA_BITS:0]Q
               );
parameter
   DATA_BITS = 16,                                        // data length
   DATA_COUNT_BITS = 4;                                   // data shift bits


   always @(posedge clk )begin
      case({S1,S0})
      2'b00: Q <= Q;                                      //S1S0=00 保持
      2'b01: Q <= {SR,Q[DATA_BITS:1]};                    //S1S0=01 右移DIR=1
      2'b10: Q <= {Q[DATA_BITS-1:0],SL};                  //S1S0=10 左移DIR=0
      2'b11: Q <= D;                                      //S1S0=11 并入
      endcase
   end

endmodule

```

第六章作业 一

Ch6-1

page391-392：

6-1，6-2，6-5

With Registers , It is More easy!

Thank you!