计算机组织与结构 II：POC 实验报告

李勃璘
东南大学

日期：2025 年 3 月 18 日

摘　要

本文为计算机组织与结构 II 课程设计中，8 位并行输出控制单元（Parallel Output Controller, POC）的说明文档。该控制单元的状态和输入数据受一 8 位处理器控制，并可以与打印机交互，将内部数据打印出来。项目源代码已附在文末。

1　整体功能

本设计实现了一个 8-bit 的并行输出控制单元，其通过一个简易的 8-bit 处理器获取数据和控制信息，并将输出结果传输给打印机模块。打印机经过一段时间的延时后将数据打印出来。项目的总体结构如图 1 所示。

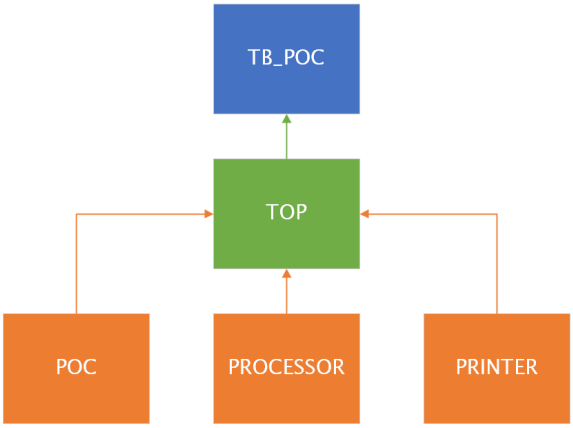


图 1: 项目模块结构

2　POC 模块

该模块完成并行输出控制单元（下文称 POC）的描述，以及和打印机（下文称 PRINTER）、处理器（下文称 PROCESSOR）的交互。

2.1　时钟与复位信号

该模块的时钟和复位信号如表 1。

信号名	位宽	流向	描述
i_clk	1	输入	时钟信号
i_rst_n	1	输入	全局异步复位

表 1: POC 模块的时钟与复位信号

2.2 模块接口

该模块的对外接口如表 2。

端口	位宽	流向	描述	来源/通向模块
i_addr	1	输入	地址信号，当值为 0 时，`i_din` 被解释为状态信号；当值为 1 时，被解释为要打印的数据	PROCESSOR
o_dout	8	输出	输出自身状态信号	PROCESSOR
i_din	8	输入	数据信号，值意义参考 `i_addr` 的值	PROCESSOR
i_rw	1	输入	读写信号，当值为 1 时 PROCESSOR 向 POC 写值	PROCESSOR
i_mode	1	输入	模式选择，1 为中断模式，0 为查询模式	/
o_irq	1	输出	中断信号，为 1 时向 PROCESSOR 发出中断请求（低有效）	PROCESSOR
i_rdy	1	输入	信号为 1 时允许向打印机写数据	PRINTER
o_tr	1	输出	信号为 1 时向打印机发送请求	PRINTER
o_pd	8	输出	向打印机输出数据	PRINTER

表 2: POC 模块端口说明

2.3 模块功能描述

该模块描述了查询模式和中断模式下的 POC 状态转换和信号输入输出。模块功能见算法 1。[1]实际实现时，为调试方便以及区分两种模式下的功能，为中断模式下的数据更新和打印机交互各设立了一个状态，但功能完全相同。

[1]表述和代码不完全相同。

Algorithm 1 POC 模块

Note: System turns query mode when SR_0 is 0, otherwise it turns to interrupt mode.
Input: $i\_clk, i\_rst\_n, i\_mode, i\_rw, i\_addr, i\_din, i\_rdy$
Output: $o\_dout, o\_irq, o\_tr, o\_pd$
Internal Registers: $state, buffer, status, printer\_data, enable\_printer, ready$
procedure INITIALIZE
    $state \leftarrow$ Wait for PROCESSOR data
    $ready \leftarrow 0$
    $status \leftarrow 8'b10000000$
    $enable\_printer \leftarrow 0$
    $interrupt \leftarrow SR_7 \,\&\, SR_0$
    $SR_0 \leftarrow i\_mode$
end procedure
procedure WAIT FOR PROCESSOR DATA
    State Transition:
    if $i\_rw == 1 \,\&\&\, SR_0 == 0 \, or \, interrupt$ **then**
        $state \leftarrow$ Update buffer and status from PROCESSOR
    end if
end procedure
procedure UPDATE BUFFER AND STATUS FROM PROCESSOR
    if $i\_rw == 1$ **then**
        if $i\_addr == 1$ **then**
            $buffer \leftarrow i\_din$
        else if $i\_addr == 0$ **then**
            $status \leftarrow i\_din$
        end if
    end if
    State Transition:
    if $SR_7 == 0$ **then**
        $state \leftarrow$ Push buffer to PRINTER
    end if
end procedure
procedure PUSH BUFFER TO PRINTER
    if PRINTER Ready **then**
        $printer\_data \leftarrow buffer$ (Until the end of the state)
        $o\_tr \leftarrow$ pulse
        State Transition:
        $state \leftarrow$ Wait for PROCESSOR data
    end if
end procedure

3 PROCESSOR 模块

3.1 时钟与复位信号

该模块的时钟和复位信号如表 3。

信号名	位宽	流向	描述
i_clk	1	输入	时钟信号
i_rst_n	1	输入	全局异步复位

表 3: POC 模块的时钟与复位信号

3.2 模块接口

该模块的对外接口如表 4。

端口	位宽	流向	描述	来源/通向模块
o_addr	1	输出	地址信号，当值为 0 时，`i_din` 被解释为状态信号；当值为 1 时，被解释为要打印的数据	POC
i_dout	8	输入	POC 的 8 位状态信号（SR）	POC
o_din	8	输出	数据信号，值意义参考 `o_addr` 的值	POC
o_rw	1	输出	读写信号，当值为 1 时 PROCESSOR 向 POC 写值	POC
i_irq	1	输入	来自 POC 的中断信号（低有效）	POC
i_data	8	输入	系统输入数据（将被打印）	/

表 4: PROCESSOR 模块端口说明

3.3 模块功能描述

该模块描述了一个中央处理器（模块功能如算法 2 ）。它向 POC 传递数据，并在传递数据结束后将 SR_7 归零以表征传送结束。处理器使用地址线选择要写入的寄存器，使用数据线写入数据。地址译码与数据分配由 POC 完成。

由于状态和数据共用一条传输线，有必要规定传送时序。规定当 POC 空闲时（SR_7 为 1），先写入数据到传输线端口，然后打开写模式以更新传输线；更新之后等待一个周期，更改地址为“状态”，并写入更新 SR_7 后的状态数据。

Algorithm 2 PROCESSOR 模块

Note: We denote STATUS as 0 and BUFFER as 1, as they represent the actual bit allocation.
Input: $i\_clk, i\_rst\_n, i\_irq, i\_dout, i\_data$
Output: o_din, o_addr, o_rw
Internal Registers: $poc\_status, address, rw, state, next\_state, set\_data\_done, read\_status\_done$
procedure INITIALIZE( )
    $state \leftarrow IDLE$
    $poc\_status \leftarrow 8'b0$
    $read\_status\_done \leftarrow 0$
end procedure
procedure STATE TRANSITION( )
    if $state == IDLE$ **then**
        $(Mode == Interrupt)?state \leftarrow READ\_FROM\_POC : state \leftarrow SET\_DATA$
    end if
    $state \leftarrow READ\_FROM\_POC \leftarrow SET\_DATA$
    $state \leftarrow WRITE\_DATA \leftarrow DELAY \leftarrow WRITE\_STATUS$
    $state \leftarrow IDLE$
end procedure
procedure CONTROL LOGIC FOR ADDRESS AND READ-WRITE( )
    state == IDLE: $address \leftarrow STATUS, rw \leftarrow 0$
    state == READ_FROM_POC: $rw \leftarrow 0, address \leftarrow STATUS$
    state == SET_DATA: $rw \leftarrow 0, address \leftarrow BUFFER$
    state == WRITE_DATA: $rw \leftarrow 1, address \leftarrow BUFFER$
    state == WRITE_STATUS: $rw \leftarrow 1, address \leftarrow STATUS$
end procedure
procedure DATA OUTPUT CONTROL( )
    if $address == STATUS$ **then**
        $o\_din \leftarrow poc\_status$
    else
        $o\_din \leftarrow i\_data$
    end if
end procedure
procedure STATUS OPERATIONS( )
    if $state == READ\_FROM\_POC$ **then**
        $poc\_status \leftarrow i\_dout$
    else if $state == SET\_DATA$ **then**
        $poc\_status[7] \leftarrow 0$
    end if
end procedure

4 PRINTER 模块

4.1 时钟与复位信号

该模块的时钟和复位信号如表 1。

信号名	位宽	流向	描述
i_clk	1	输入	时钟信号
i_rst_n	1	输入	全局异步复位

表 5: PRINTER 模块的时钟与复位信号

4.2 模块接口

该模块的对外接口如表 6。

端口	位宽	流向	描述	来源/通向模块
i_tr	1	输入	冲激信号，象征打印开始	POC
i_pd	8	输入	POC 的 8 位待打印数据，打印机将取冲激信号为 1 时刻的输入	POC
o_rdy	1	输出	打印机工作时为 0，否则为 1	POC
o_data	8	输出	打印机输出打印数据	/

表 6: PRINTER 模块端口说明

4.3 模块功能描述

该模块是一个打印机模块。在接收到来自 POC 的冲激信号之后，打印机将自身状态设置为 BUSY。打印机需要 8 个时钟周期进行打印，打印的结果会延迟 4 个时钟。

Algorithm 3 PRINTER 模块

Note: We denote STATUS as 0 and BUFFER as 1, as they represent the actual bit allocation.
Input: $i\_clk, i\_rst\_n, i\_tr, i\_pd$
Output: o_data, o_rdy
procedure IDLE( )
    Do nothing.
    $o\_rdy \leftarrow 1$
    State Transition:
    if $i\_tr == 1$ **then**
        $o\_rdy \leftarrow 0$ (with one clock cycle delay)
        $state \leftarrow BUSY$
    end if
end procedure
procedure BUSY( )
    if $counter == 8$ **then**
        End of printing.
        $state \leftarrow IDLE$
    else
        Print data with a delay of 4 clock cycles.
        counter++;
    end if
end procedure

5 仿真验证

5.1 时延分析

定义一个打印周期为从 POC 重置自身 SR_7 到一轮打印结束所经历的时钟上升沿个数。那么：

1. 由 POC 重置到 PROCESSOR 置零 SR_7，使用 6 个时钟，分别为读取 POC 信号的一个时钟和状态机的五个时钟；[2]
2. POC 此时开始与打印机进行交互。写入 `o_tr` 和 `o_pd` 共使用 1 个时钟，`o_tr` 信号持续 1 个时钟；
3. 打印机收到信号后，耗时 8 个时钟打印数据；
4. 打印结束后，POC 重置 SR_7，使用 1 个时钟。

故一个打印周期为 17 个时钟（查询模式）或 16 个时钟（中断模式）。

5.2 激励设置

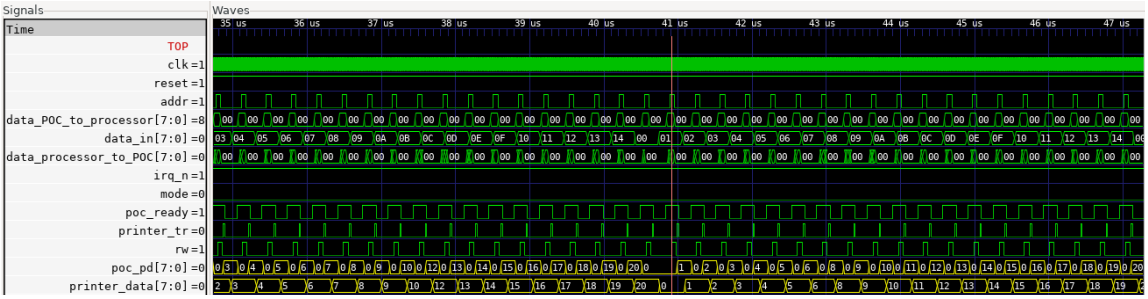
为了制造连续数据，避免数据丢包，输入数据的持续时间（延迟）应当和打印周期相同。仿真设置时钟以 20ns 为周期，故查询模式下，每一个数据更新之间的延迟应为 340ns；中孤单模式下，每一个数据更新之间的延迟应为 320ns。为简单起见，对两个不同的模式测试时，使用相同的测试样例。最终，选取无符号十进制数 0-20 按照 340ns/320ns 延时循环出现，作为仿真的输入数据。

[2]中断模式较查询模式少用一个时钟。因为中断信号由组合逻辑生成，中断模式下 PROCESSOR 可立刻读取中断信号，无需读取 POC 信号。

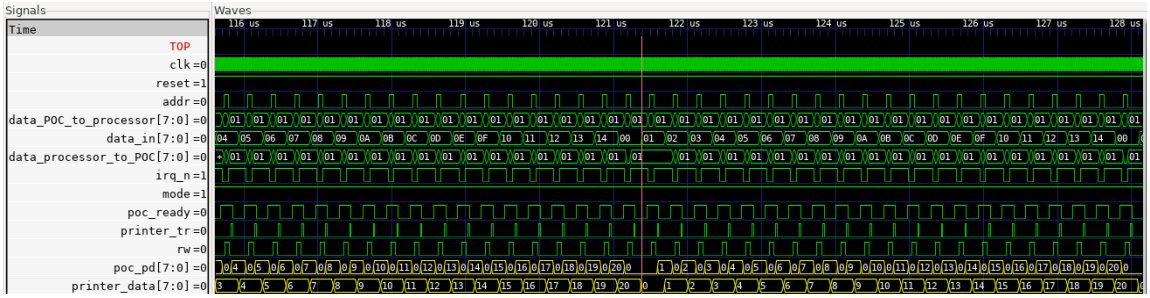
5.3 仿真结果

依据激励设置，撰写 POC 模块的 Testbench 如附录 A.5。使用 iverilog+vvp+gtkwave 运行仿真，运行结果为所有模块完整的波形图。

观察 TOP 模块的波形，可以看到查询和中断模式下（图 2 ），信号都能够流水输出，这证明激励设置正确。中断模式下中断信号在每个周期开始时由 POC 置零一次，标志一个打印周期的开始；而查询模式下中断信号始终为高电平，这符合我们的预期。



(a) 查询模式波形图



(b) 中断模式波形图

图 2: TOP 模块仿真波形图

接下来观察各个子模块的波形，验证时延分析的分析结果。`i_dout` 是 POC 目前的状态，当 POC 将状态更新为 1 时，查询模式下 PROCESSOR 使用了 6 个时钟周期将 SR_7 更新为 0。波形结果如图 3 所示。

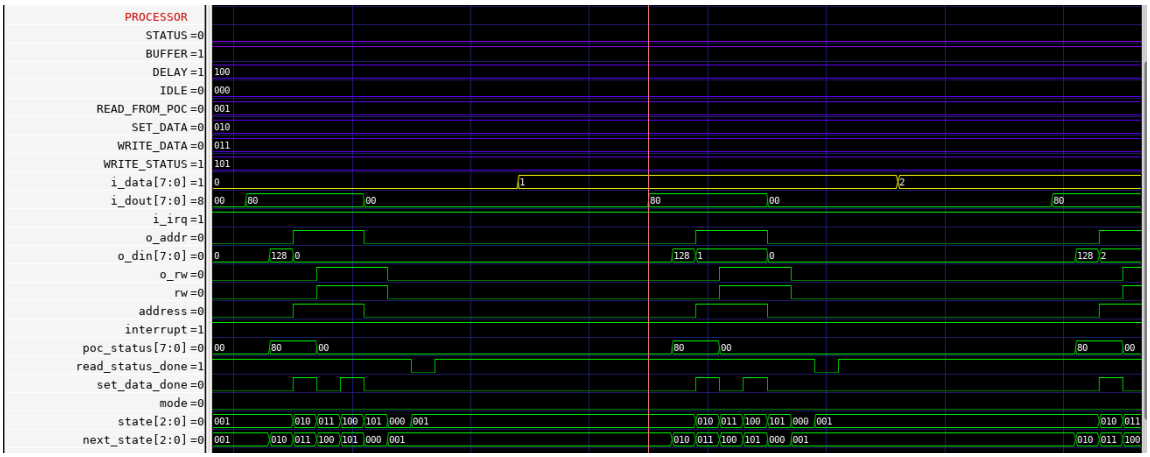


图 3: PROCESSOR 模块一个打印周期内的波形（查询模式）

中断模式下，PROCESSOR 使用了 5 个时钟周期将 SR_7 更新为 0（图 4 ）。

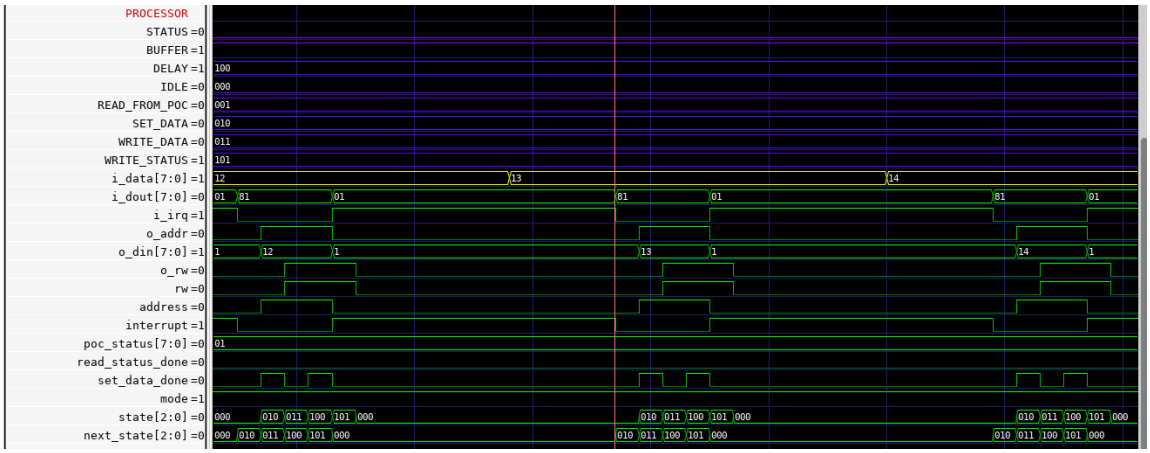


图 4: PROCESSOR 模块一个打印周期内的波形（中断模式）

POC 的 SR_7 被更新为 0 后，POC 使用一个时钟周期将 `o_tr` 值赋为 1。`o_tr` 持续了一个时钟周期（图 5 ）。

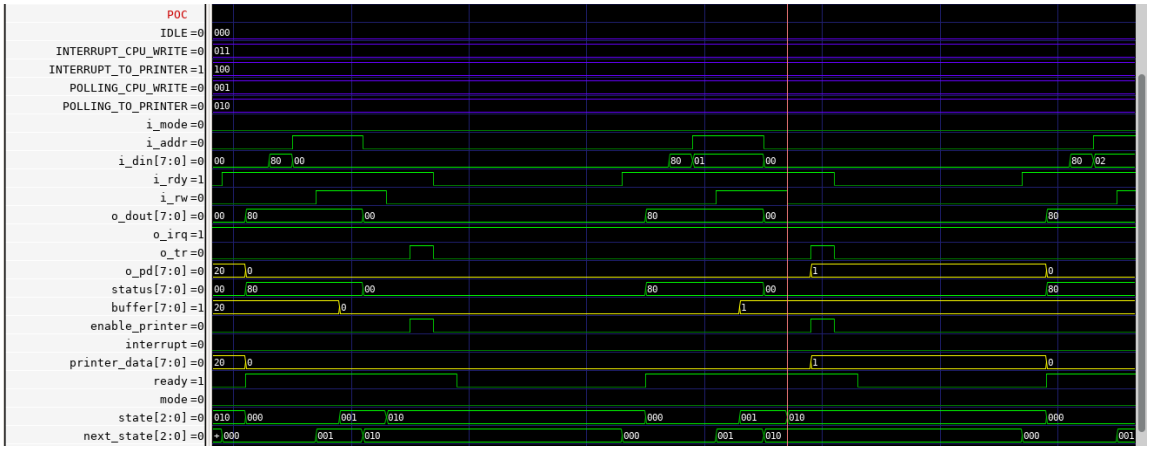


图 5: POC 模块与 PRINTER 模块的交互

`o_tr` 的脉冲使得打印机开始打印。打印机选取了 `o_tr` 为 1 时的待打印数据，延迟四个周期后输出到 `o_data` 中。打印机在开始打印时即开始计数。计数器 `counter` 满 8 之后归零，打印机的 `ready` 信号重新置 1。（图 6 ）

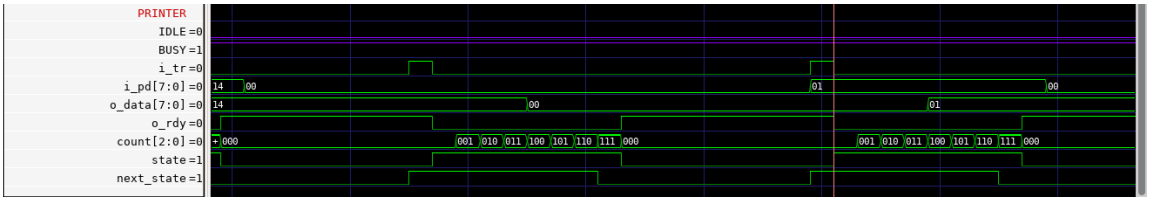


图 6: PRINTER 模块打印捕获的数据

POC 模块发现 ready 信号和寄存器中 ready 信号不同，即可判定打印结束。再经过一个周期后，POC 模块将自身 SR_7 置 1，打印周期结束。（图 7 ）

综上所述，总时钟周期和理论分析相同，这代表硬件实现的结果和预期相符。

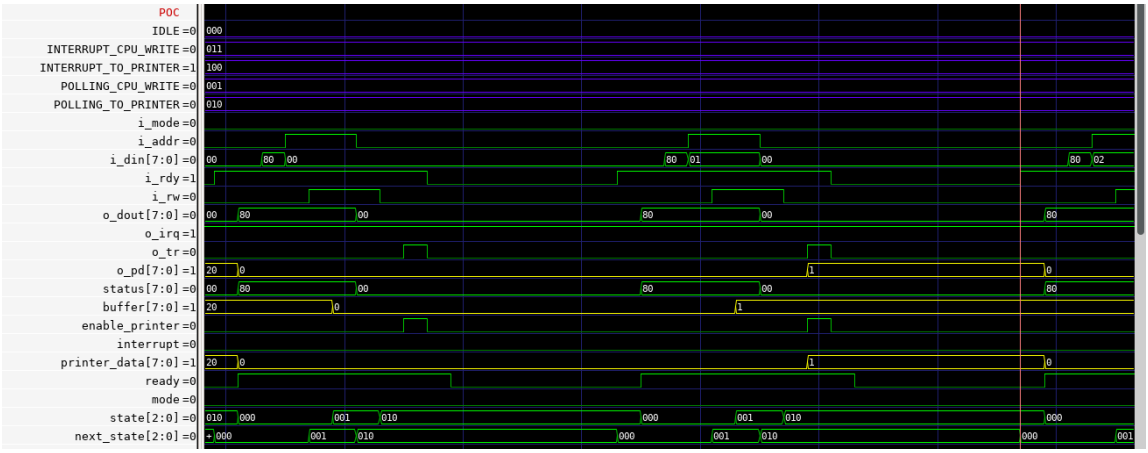


图 7: POC 模块更新 SR_7

A 完整设计代码

A.1 top.v

```
module TOP (
    i_clk,
    i_rst_n,
    i_data,
    i_mode,
    o_tr,
    o_pd,
    o_rdy,
    o_data,
    o_rw,
    o_addr,
    o_irq,
    o_data_poc_to_processor,
    o_data_processor_to_poc
);
// I/O

input wire i_clk;
input wire i_rst_n;
input wire i_mode;
input wire [7:0] i_data;

output wire [7:0] o_data;
output wire o_irq;
output wire o_addr;
output wire o_rw;
output wire o_tr;
output wire [7:0] o_pd;
output wire o_rdy;
output wire [7:0] o_data_poc_to_processor;
output wire [7:0] o_data_processor_to_poc;
// CPU/POC connections

wire irq;
```

```
wire addr;
wire [7:0] data_processor_to_POC;
wire [7:0] data_POC_to_processor;
wire rw;

// POC/Printer connections

wire tr;
wire [7:0] pd;
wire rdy;

// Instantiations

PROCESSOR u_processor (
    .i_clk(i_clk),               // 时钟输入
    .i_rst_n(i_rst_n),           // 复位信号，低有效
    .i_irq(irq),                            // 中断
    .i_data(i_data),                        // Directly from TB
    .i_dout(data_POC_to_processor),         // 从 POC 读取的数据
    .o_din(data_processor_to_POC),          // 发送到 POC 的数据
    .o_addr(addr),                          // 地址信号
    .o_rw(rw)                  // 读/写信号
);

POC u_poc (
    .i_addr(addr),
    .o_dout(data_POC_to_processor),
    .i_din(data_processor_to_POC),
    .i_rw(rw),
    .i_clk(i_clk),
    .i_rst_n(i_rst_n),
    .i_mode(i_mode),
    .o_irq(irq),
    .i_rdy(rdy),
    .o_tr(tr),
    .o_pd(pd)
);

PRINTER u_printer (
    .i_tr(tr),
    .i_pd(pd),
    .o_rdy(rdy),
    .i_clk(i_clk),
    .i_rst_n(i_rst_n),
    .o_data(o_data)         // directly output to tb
);

// assignments

// CPU/POC
assign o_irq = irq;
assign o_addr = addr;
assign o_rw = rw;
assign o_data_processor_to_poc = data_processor_to_POC;
```

```
assign o_data_poc_to_processor = data_POC_to_processor;

// POC/Printer
assign o_tr = tr;
assign o_pd = pd;
assign o_rdy = rdy;


endmodule
```

A.2 poc.v

```
module POC (
    i_addr,
    o_dout,
    i_din,
    i_rw,
    i_clk,
    i_rst_n,
    i_mode,
    o_irq,
    i_rdy,
    o_tr,
    o_pd
);

  // From Top module

  input wire i_mode;  // 0 = query, 1 = interrupt

  // From/to Processor

  input wire i_addr;  // 0: status, 1: buffer
  input wire i_clk;
  input wire i_rst_n;
  input wire i_rw;  // 0: read, 1: write
  input wire [7:0] i_din;  // status/data from Processor
  output wire [7:0] o_dout;  // status to CPU
  output wire o_irq;  // 0: interrupt

  // From/to Printer
  input wire i_rdy;
  output wire o_tr;
  output [7:0] o_pd;

  // Inside POC
  reg mode;
  reg [7:0] status;
  reg [7:0] buffer;
  reg [7:0] printer_data;
  reg enable_printer;
  wire interrupt;
  reg ready;
```

```
// State Params

parameter IDLE = 3'b0;
// Wait for CPU response
parameter POLLING_CPU_WRITE = 3'b001;
// POC will receive data from processor using polling method
parameter POLLING_TO_PRINTER = 3'b010;
// POC will transmit data to printer
parameter INTERRUPT_CPU_WRITE = 3'b011;
// POC will receive data from processor using interrupt method
parameter INTERRUPT_TO_PRINTER = 3'b100;
// POC will transmit data to printer


reg [2:0] state, next_state;


// initial
always @(posedge i_clk or negedge i_rst_n) begin
  if (!i_rst_n) begin
    state <= IDLE;
    ready <= 'b0;  // maybe there is a more beautiful solution
    mode  <= 'b0;
  end else begin
    state <= next_state;
    ready <= i_rdy;
    mode  <= i_mode;
  end
end

// state transition
always @(*) begin
  case (state)
    IDLE: begin
      if (i_rw == 1'b1) begin
        if (status[0] == 0) begin  // query
          next_state = POLLING_CPU_WRITE;
        end else begin
          if (interrupt == 1'b1) begin
            next_state = INTERRUPT_CPU_WRITE;
          end
        end
      end else begin
        next_state = IDLE;
      end
    end
    POLLING_CPU_WRITE: begin
      if (status[7] == 0) begin
        next_state = POLLING_TO_PRINTER;
      end else begin
        next_state = POLLING_CPU_WRITE;
```

```
      end
    end
    POLLING_TO_PRINTER: begin
      if (i_rdy == 'b1 && ready == 'b0) begin
        next_state = IDLE;
      end else begin
        next_state = POLLING_TO_PRINTER;
      end
    end
    INTERRUPT_CPU_WRITE: begin
      if (status[7] == 0) begin
        next_state = INTERRUPT_TO_PRINTER;
      end else begin
        next_state = INTERRUPT_CPU_WRITE;
      end
    end
    INTERRUPT_TO_PRINTER: begin
      if (i_rdy == 'b1 && ready == 'b0) begin
        next_state = IDLE;
      end else begin
        next_state = INTERRUPT_TO_PRINTER;
      end
    end
    default: begin
      next_state = IDLE;
    end
  endcase
end


// status and buffer
always @(*) begin
  case (state)
    IDLE: begin
      status         = {7'b1000000, mode};  // reset to POC ready
      printer_data   = 8'b0;  // every cycle reset the printer data
      enable_printer = 1'b0;  // every cycle reset the printer status
    end
    POLLING_CPU_WRITE: begin
      // CPU needs to write both buffer and status in this state
      if (i_rw == 1'b1 && i_addr == 1'b1) begin
        buffer = i_din;
      end
      if (i_rw == 1'b1 && i_addr == 1'b0) begin
        status = i_din;
      end
    end
    INTERRUPT_CPU_WRITE: begin
      if (i_rw == 1'b1 && i_addr == 1'b1) begin
        buffer = i_din;
      end
      if (i_rw == 1'b1 && i_addr == 1'b0) begin
        status = i_din;
      end
```

```
      end
    endcase
  end

  always @(posedge i_clk or negedge i_rst_n) begin
    if (!i_rst_n) enable_printer <= 1'b0;
    else begin
      if (state == POLLING_TO_PRINTER || state == INTERRUPT_TO_PRINTER) begin

        if (ready == 1'b1 && enable_printer == 1'b0) begin
          printer_data   <= buffer;
          enable_printer <= 1'b1;
        end else if (enable_printer == 1'b1) begin
          enable_printer <= 1'b0;
        end
      end
    end
  end

  // Assignments
  assign o_irq = ~interrupt;
  assign o_tr = enable_printer;
  assign o_pd = printer_data;
  assign o_dout = status;

  assign interrupt = status[7] & status[0];

endmodule
```

A.3 processor.v

```
module PROCESSOR (
    i_clk,
    i_rst_n,
    i_irq,
    i_dout,
    i_data,
    o_din,
    o_addr,
    o_rw
);


  input wire i_clk;
  input wire i_rst_n;
  input wire i_irq;
  input wire [7:0] i_dout;  // Data from POC (specifically status info)
  input wire [7:0] i_data;  // Data from top module

  output reg [7:0] o_din;  // Data to POC
  output wire o_addr;
  output wire o_rw;
```

```
// Internal registers
reg [7:0] poc_status;
reg address;
reg rw;

// State registers
reg [2:0] state, next_state;

// Status tracking signals
reg set_data_done;
reg read_status_done;

// Mode signal from poc_status[0]
wire mode = poc_status[0];
wire interrupt = i_irq;

// State encoding
parameter STATUS = 1'b0;
parameter BUFFER = 1'b1;

parameter IDLE = 3'b000;
parameter READ_FROM_POC = 3'b001;
parameter SET_DATA = 3'b010;
parameter WRITE_DATA = 3'b011;
parameter DELAY = 3'b100;
parameter WRITE_STATUS = 3'b101;

// State transition
always @(posedge i_clk or negedge i_rst_n) begin
  if (!i_rst_n) state <= IDLE;
  else state <= next_state;
end

// Next state logic
always @(*) begin
  case (state)
    IDLE: begin
      if (mode == 1'b0) next_state = READ_FROM_POC;
      else if (interrupt == 1'b0) next_state = SET_DATA;
      else next_state = IDLE;
    end
    READ_FROM_POC: begin
      if (read_status_done && poc_status[7] == 1'b1) next_state = SET_DATA;
      else next_state = READ_FROM_POC;
    end
    SET_DATA: begin
      if (set_data_done) next_state = WRITE_DATA;
      else next_state = SET_DATA;
    end
    WRITE_DATA: begin
      next_state = DELAY;  // wait one clock cycle for poc read delay
    end
    DELAY: begin
      next_state = WRITE_STATUS;
```

```
    end
    WRITE_STATUS: begin
      next_state = IDLE;
    end
    default: next_state = IDLE;
  endcase
end

// Control logic for address and rw
always @(*) begin
  case (state)
    IDLE: begin
      address = STATUS;
      rw = 1'b0;
    end
    READ_FROM_POC: begin
      rw = 1'b0;
      address = STATUS;
    end
    SET_DATA: begin
      rw = 1'b0;
      address = BUFFER;
    end
    WRITE_DATA: begin
      rw = 1'b1;
      address = BUFFER;
    end
    WRITE_STATUS: begin
      rw = 1'b1;
      address = STATUS;
    end
  endcase
end

// Output data control
always @(*) begin
  set_data_done = 1'b0;
  if (state == WRITE_DATA || state == IDLE) begin
    set_data_done = 1'b0;
  end else begin
    if (address == STATUS) o_din = poc_status;
    else begin
      o_din = i_data;
      set_data_done = 1'b1;
    end
  end
end

// poc_status and read_status_done update
always @(posedge i_clk or negedge i_rst_n) begin
  if (!i_rst_n) begin
    read_status_done <= 1'b0;
    poc_status <= 8'b0;
  end else begin
```

```
      case (state)
        IDLE: begin
          read_status_done <= 1'b0;
        end
        READ_FROM_POC: begin
          poc_status <= i_dout;
          read_status_done <= 1'b1;
        end
        SET_DATA: begin
          poc_status[7] <= 1'b0;
        end
      endcase
    end
  end

  // Assign output signals
  assign o_rw   = rw;
  assign o_addr = address;

endmodule
```

A.4 printer.v

```
module PRINTER (
    i_tr,
    i_pd,
    o_rdy,
    i_clk,
    i_rst_n,
    o_data
);
  // interfaces
  input wire i_clk;
  input wire i_rst_n;
  input wire i_tr;
  input wire [7:0] i_pd;
  output wire o_rdy;
  output wire [7:0] o_data;

  // inside printer
  reg [7:0] delay_buffer[0:3];  // for delaying 4 cycles
  reg [2:0] count;
  reg state, next_state;

  parameter IDLE = 1'b0;
  parameter BUSY = 1'b1;


  always @(posedge i_clk or negedge i_rst_n) begin
    if (!i_rst_n) begin
      state <= IDLE;
    end else begin
      state <= next_state;
```

```
  end
end

// state transition
always @(*) begin
  case (state)
    IDLE: begin
      if (i_tr) begin
        next_state <= BUSY;
      end else begin
        next_state <= IDLE;
      end
    end
    BUSY: begin
      // we will wait 8 clock cycles for printer to print, even though the printing process
          only needs 4.
      if (count == 3'b111) begin
        next_state <= IDLE;
      end else begin
        next_state <= BUSY;
      end
    end
    default: begin
      next_state <= IDLE;
    end
  endcase

end

// counter
always @(posedge i_clk or negedge i_rst_n) begin
  if (!i_rst_n) begin
    count <= 3'b0;
  end else begin
    case (state)
      IDLE: begin
        count <= 3'b0;
      end
      BUSY: begin
        count <= count + 1;
      end
      default: begin
        count <= 3'b0;
      end
    endcase
  end
end

// delay four cycles to simulate printing process
always @(posedge i_clk) begin
  if (state == BUSY) begin
    delay_buffer[0] <= i_pd;
  end
end
```

```
  always @(posedge i_clk) begin
    if (state == BUSY) begin
      delay_buffer[1] <= delay_buffer[0];
    end
  end

  always @(posedge i_clk) begin
    if (state == BUSY) begin
      delay_buffer[2] <= delay_buffer[1];
    end
  end

  always @(posedge i_clk) begin
    if (state == BUSY) begin
      delay_buffer[3] <= delay_buffer[2];
    end
  end

  assign o_data = delay_buffer[3];
  assign o_rdy  = (state == IDLE) ? 1'b1 : 1'b0;

endmodule
```

A.5 tb.v

```
    // Date: 2025.3.10
    // Author: LiPtP
    // Description: Testbench for TOP module
    `timescale 1ns / 1ps
    module tb_POC;

      // Clock and reset
      reg clk;
      reg reset;

      // Input to modules
      reg [7:0] data_in;
      reg mode;

      // Output from modules
      wire [7:0] printer_data;
      wire printer_tr;
      wire [7:0] poc_pd;
      wire poc_ready;

      wire rw;
      wire addr;
      wire [7:0] data_POC_to_processor;
      wire [7:0] data_processor_to_POC;
      wire irq_n;

      // Clock generation (50MHz, 20ns period)
```

```
always #10 clk = ~clk;

// Reset sequence
initial begin
  clk = 0;
  reset = 0;
  mode = 0;
  data_in = 8'b0;

  #50 reset = 1;  // 50ns delay to allow reset propagation
end

// Mode switching after 100,000 ns
initial begin
  #100000 mode = 1;
end

// Periodic Data Input
initial begin
  #100;
  while (1) begin
    if (mode == 0) begin
      data_in = 8'b00000000;
      #340;
      data_in = 8'b00000001;
      #340;
      data_in = 8'b00000010;
      #340;
      data_in = 8'b00000011;
      #340;
      data_in = 8'b00000100;
      #340;
      data_in = 8'b00000101;
      #340;
      data_in = 8'b00000110;
      #340;
      data_in = 8'b00000111;
      #340;
      data_in = 8'b00001000;
      #340;
      data_in = 8'b00001001;
      #340;
      data_in = 8'b00001010;
      #340;
      data_in = 8'b00001011;
      #340;
      data_in = 8'b00001100;
      #340;
      data_in = 8'b00001101;
      #340;
      data_in = 8'b00001110;
      #340;
      data_in = 8'b00001111;
      #340;
```

```
            data_in = 8'b00010000;
            #340;
            data_in = 8'b00010001;
            #340;
            data_in = 8'b00010010;
            #340;
            data_in = 8'b00010011;
            #340;
            data_in = 8'b00010100;
            #340;
          end else begin
            data_in = 8'b00000000;
            #320;
            data_in = 8'b00000001;
            #320;
            data_in = 8'b00000010;
            #320;
            data_in = 8'b00000011;
            #320;
            data_in = 8'b00000100;
            #320;
            data_in = 8'b00000101;
            #320;
            data_in = 8'b00000110;
            #320;
            data_in = 8'b00000111;
            #320;
            data_in = 8'b00001000;
            #320;
            data_in = 8'b00001001;
            #320;
            data_in = 8'b00001010;
            #320;
            data_in = 8'b00001011;
            #320;
            data_in = 8'b00001100;
            #320;
            data_in = 8'b00001101;
            #320;
            data_in = 8'b00001110;
            #320;
            data_in = 8'b00001111;
            #320;
            data_in = 8'b00010000;
            #320;
            data_in = 8'b00010001;
            #320;
            data_in = 8'b00010010;
            #320;
            data_in = 8'b00010011;
            #320;
            data_in = 8'b00010100;
            #320;
```

```
        end
      end
    end

    // Dump waveforms
    initial begin
      $dumpfile("poc.vcd");
      $dumpvars(0, tb_POC);
      #200000 $finish;
    end

    // Instantiate DUT
    TOP u_dut (
        .i_clk(clk),
        .i_rst_n(reset),
        .i_data(data_in),
        .i_mode(mode),
        .o_tr(printer_tr),
        .o_pd(poc_pd),
        .o_rdy(poc_ready),
        .o_data(printer_data),
        .o_rw(rw),
        .o_addr(addr),
        .o_irq(irq_n),
        .o_data_poc_to_processor(data_POC_to_processor),
        .o_data_processor_to_poc(data_processor_to_POC)
    );

endmodule
```