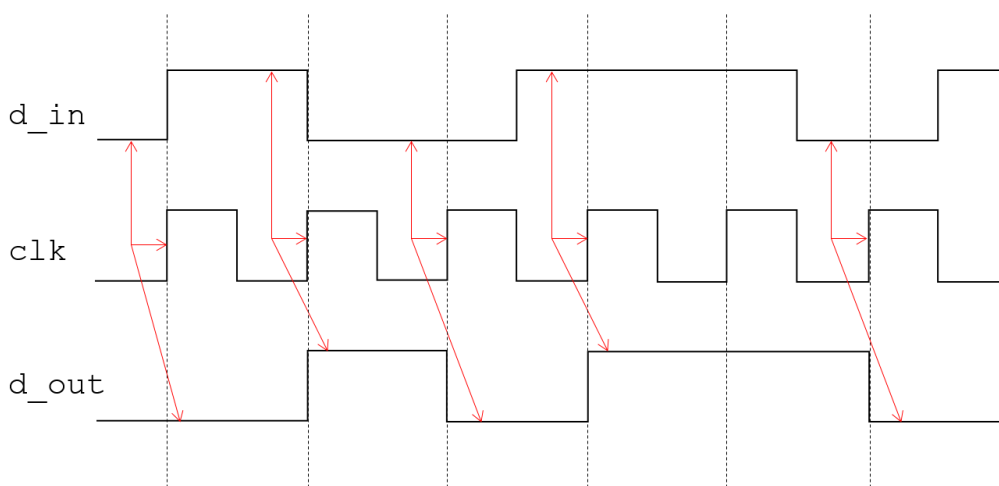# 数字同步系统设计

本节的学习目标是:

- 了解如何在给定协议下读取输入数据;
- 了解如何为具有数据路径和控制器的同步系统建模;

到目前为止,我们一直在研究构成完整系统的各种组合和顺序子组件。在本节中,我们将查看一个完整的示例,此处概述的设计实践应作为作业 2 中练习的模板。

同步系统中的定时原理

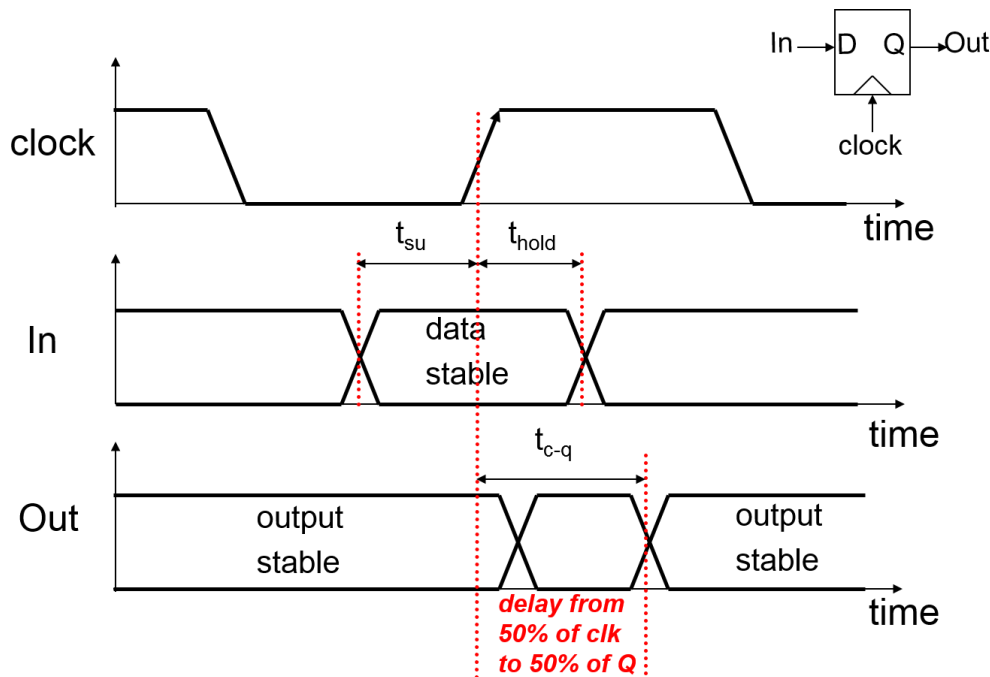当您执行模式识别器 1 的示例时,您会注意到数据需要在时钟上升沿之前的某个时间可用,以便触发器正确读取数据。例如,考虑图 6.1 的波形。

寄存器的时序波形。

箭头指示针对给定时钟边沿读取的相关数据值,以及相应的输出数据值。这里的关键点是,除非输入数据在时钟沿之前的某个时间有效,否则数据将无法被正确读取。

输入数据有效和时钟沿之间需要多少时间?

在目前的实施中,任何非零物理时间都是足够的,但实际上它需要大于称为设置时间的时间,这是晶体管级电路和制造技术的特征。实际上,寄存器(触发器)具有三个基本时序指标:建立时间 (t_{su})、保持时间 (t_{hold})和时钟到输出传播

延迟 ($t_{c\text{-}q}$)。这些时间指标如图 6.2 所示。

时序波形显示触发器的建立和保持时间以及传播延迟。

The setup time is the time that the data needs to be ready before the clock edge. If this condition is violated, it cannot be guaranteed that the data is read and saved properly. The hold time is the amount of time that the data needs to be held stable after the clock edge. If the data on the in node changes too soon after the clock edge, then the hold time condition is violated, and again it cannot be guaranteed that the data is read and saved properly. Finally, the propagation delay is the amount of time after the clock edge that it takes for the output to be valid with the new data.

In register implementations so far, the setup and hold times and propagation delay were not specified; i.e., they were ideal registers. If we knew the delay of a flip-flop when implemented in our target FPGA, we could potentially model it as an assignment delay to get a more accurate idea of the timing in a simulation. However, then we would have to model the delay of every logic gate to make the timing reasonably accurate. Afterwards, we would have to remove all assignment delays in order to synthesize the code, as clearly assignment delays are not synthesiable. Luckily, this is not necessary as all synthesis tools are able to analyse the timing automatically. After mapping the prescribed logic to hardware, the tool is able to work out all timing paths between registers to check for setup- and hold-time violations.

Reading Input Data Values

Reading Input Levels

Any data on system inputs that have to be read synchronously, such as that on x for the pattern recogniser, have to satisfy setup- and hold-time requirements with respect to the clock edge. If you recall, the output of the Mealy machine went high for a short time between essentially because the input changed close to a clock edge. Using the input to directly drive logic that determines the output results in glitches and is not satisfactory for that reason.

Instead, the accepted way to process incoming synchronous data - i.e., data that is valid one setup time before every rising clock edge (or falling clock edge for falling-edge-triggered logic), and holds its value for one hold time after the edge - is to load it into a register, so that the value stays constant for an entire cycle, regardless of whether the input changes or not outside the setup- and hold-time requirements. This can be done by simply connecting x to the input of a flip-flop, and using the output of the flip-flop as the main input signal to the rest of the system. In the example of the pattern recogniser, assume that a signal called x_reg exists, and this is the signal read by all the downstream combinational processes. The following additional code is all that's required.

```
PROCESS(clk)
BEGIN
  IF clk'event AND clk='1' THEN
    x_reg <= x;
  END IF;
END PROCESS;
```

Register the Inputs of Pattern Recogniser 1

1. Load the Mealy implementation and register the inputs as per the above explanation. Run a new simulation.
2. Note how the x_reg signal, which is the input to the rest of the system, always remnains steady for one entire clock cycle at least. However, you can also see that registering a signal always introduces a delay of one cycle, as the value is read in on a clock edge, and then passed to the rest of the system.

Reading Input Edges

In addition to registering inputs to ensure consistent timing behaviour, registering a signal is also a common technique whereby a one-cycle delay is deliberately introduced, which may be useful for various reasons. For example, you may want to monitor transitions in signals, rather than their level, for control purposes. The single phase handshaking protocol used in Assignment 2 for instance requires exactly this. Let's look at this in a bit more details.

Assume that ctrlIn is a signal that conveys information based on transitions; i.e., what is signficant is a rising or falling edge of the control signal, rather than whether the signal is high or low. How do you monitor transitions on ctrlIn?

The first idea that might suggest itself is to simply use the 'EVENT attribute on ctrlIn i.e., have a IF ctrlIn'EVENT THEN clause, such as we do for the clock. This would work fine in simulation; however, think about how this can be synthesised. The only digital gate that is edge-triggered is a flip-flop. But connecting ctrlIn to the clock (control) input on a flip-flop is by itself meaningless; what do we connect to the data input, and how do we actually monitor the transition? Further, connecting anything other than the clock to the control input of a flip-flop is fraught with danger, as we have no control over the delay from the input pin to the input of the flip-flop. The clock is a very special signal, and the clock net is routed over the entire chip with special techniques to ensure that its latency is minimised and balanced over the entire chip.

Instead, a simple and clever way to translate a signal edge to a signal level, is to take the XOR of the signal whose transitions we are monitoring, and a one-cycle delayed version of it. Consider Fig. 6.3. As can be seen, both low-to-high and high-to-low edges are indicated by the signal at the bottom, with the arrows showing the edge we want to detect, and the corresponding signal high period.
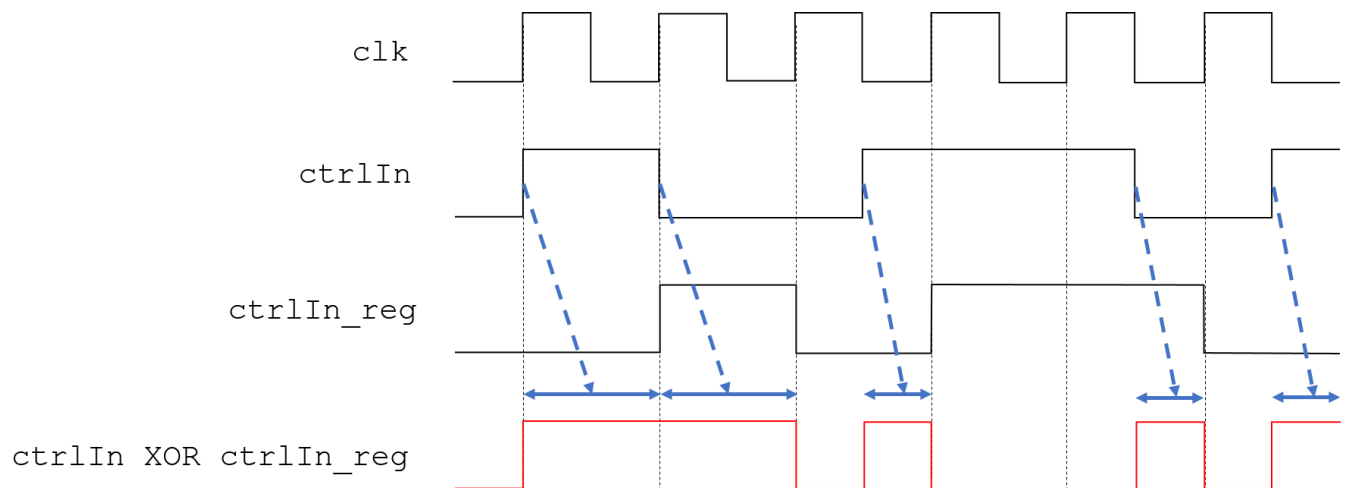
Detecting signal edges by converting edges to levels.

Implementing Pattern Recogniser 2

In the implementation of the first pattern recogniser, you most likely defined states based on recognising individual bits in the specified sequence, for example init, first, second etc. If you did indeed adopt such a scheme, you can appreciate that it is not scalable. For example, assume you had to recognise a sequence in a 64 bit word; clearly, it would be very tedious to define a state machine with 64 states, and not the best design regardless of whether or not you employed an HDL for design entry.

In this second example, the specifications are that the serial I/O interface is the same, but the pattern to recognise is a sequence of 15 zeros followed by 17 ones in a 32 bit sequence. When the pattern has been recognised, y should go high for exactly one clock cycle where a clock cycle is made up of a high and low phase with a 50% duty cycle, and go low in the next cycle.

State Transition Diagram for Pattern Recogniser 2

1. Draw the FSM state transition diagram for the above specification based on either the Mealy or Moore models.

Hint: You can use two counters with reset and enable signals to count the ones and zeros. The counters will increment on a clock edge as long as reset is deasserted, and enable is asserted. You can suspend counting by deasserting the enable signal. An example of a down counter with an enable is given in the description of a flip-flop with a clock-enable capability. These control signals

will be driven by the FSM, based on the current state for a Moore machine, and current state and inputs for a Mealy machine. To ensure that the output stays high for at least a cycle, register the x input first. You don't have to draw the state machine for the counters but show the signals that control these counters.
Answer Quiz 1, Question 15.

Implementing Pattern Recogniser 2 as a FSM

1. Construct a VHDL model of Pattern Recogniser 2 based on your designed FSM.
2. Test it with the supplied testbench, ensuring your design compiles with no changes in the testbench.

Source Code for Pattern Recogniser 2

1. Upload the source code for your module. Answer Quiz 1, Question 16.

Designing a System Comprising a Datapath and Controller

Thus far, the main examples have focused mostly on the controller and implementing it as a finite-state machine. Many digital systems have requirements for performing arithmetic operations on data that are more complex than the simple comparison operations used to check whether the input was a '0' or '1' in the pattern recognition examples. Here we will look at an example that includes arithmetic operators.

When data manipulation is required, the logic is always organised into a datapath, where the data is manipulated, and a controller, whose job it is to sequence the operations in the datapath, such as select certain inputs at certain times within the datapath, and assert signals to read and write I/O. When describing digital functionality in VHDL, the target technology (e.g.: FPGA, ASIC etc.) and synthesis tool dictate the coding style to some extent, but generally, RTL is always written following the same template, with the controller being implemented as a FSM. Certainly you should always follow this style when using Xilinx Vivado for synthesis, as you will be doing throughout the unit.

The specifications for this example is that we need to add and multiply five 8-bit operands a, b, c, d and e, that are available synchronously on five separate 8-bit input ports, where the operands are valid after a positive clock edge. The functionality (i.e., what we want to do to the data) of this data sampler is defined by f and g, where the output is to be presented on two 16-bit output ports:

f <= (a+b)*e;
g <= (a+b)*(c+d);

The interfacing is governed by a synchronous protocol (i.e., the control signals are aligned with respect to the clock). The system has an output signal, outValid, that goes high for one cycle when both f and g are valid. In the cycle following the assertion of outValid, new input data will be read.

There are many ways in which this functionality can be implemented, but let's assume that we have a constraint of one adder and one multiplier for this operation (as they are both area and power intensive components). The data path can be organised as shown in Figure 5.4.
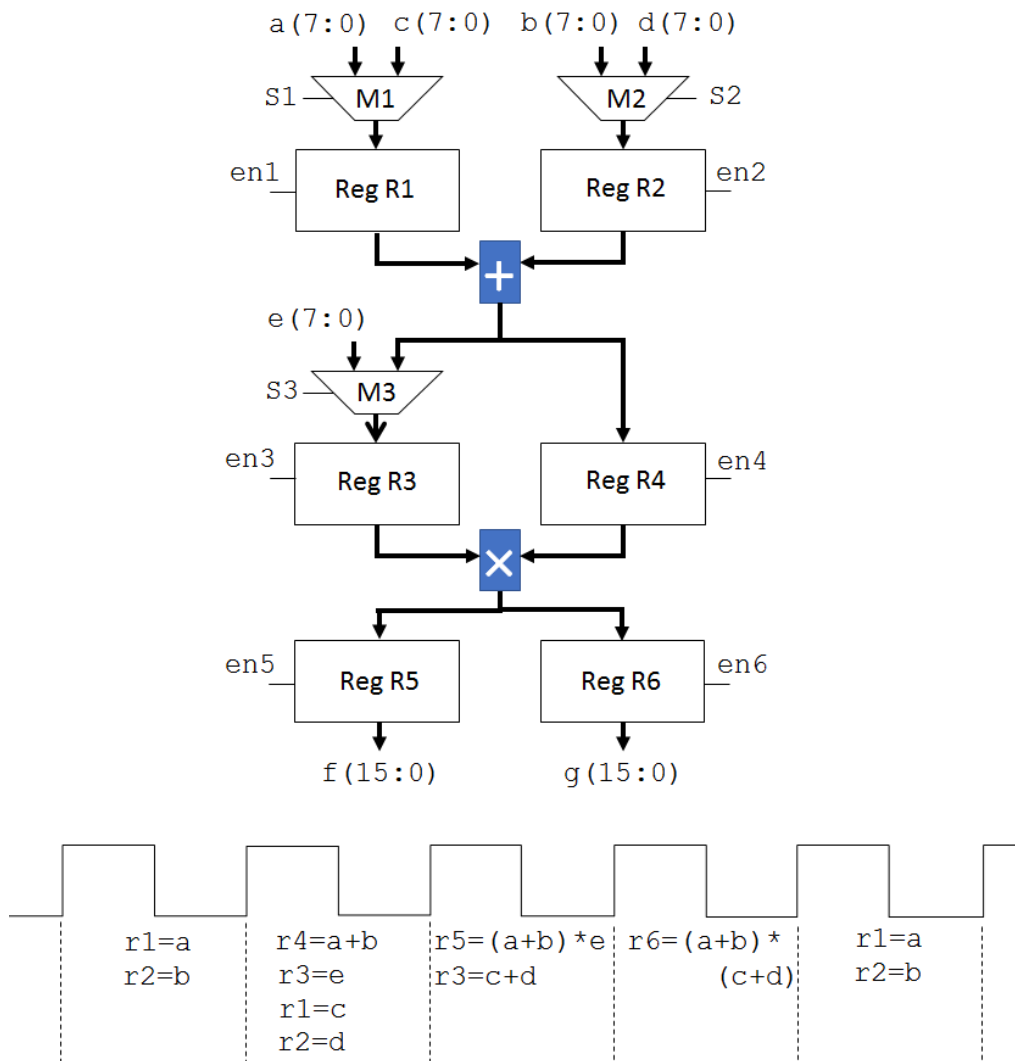
Datapath for example design with timing diagram.

- Note that these registers have enable controls designated as en. For the flip-flop to work in normal mode, the enable signal has to be asserted. If they are deasserted, the flip-flop will simply ignore the clock inputs and keep the output unchanged.

Because we have to use one adder and one multiplier, clearly we have to temporarily store some intermediate data in registers, and reroute those intermediate results to the adder and multiplier inputs in different cycles. The select signals s1, s2 and s3 controlling the MUXs and the enable signals en1, en2, en3, en4 and en5 shown in red that control the registers have to be driven to the correct values in order for the data to be correctly transformed, and their values are very precisely prescribed for any given cycle.

Assume that inputs become available on the data ports in cycle i. Therefore, in cycle i, s1 and s2 have to be driven to the values that choose the left-hand-side input, and en1 and en2 have to be asserted ensuring that a and b are loaded into the registers r1 and r2 respectively.

In cycle i+1, the control signal s3 should choose the left-hand-side input and en3 and en4 need to be asserted, so that the register R4 is loaded with the value of a+b, and R3 with e. In the same cycle, s1 and s2 will chose their right-hand side inputs, and en1 and en2 are asserted so that R1 and R2 are now loaded with c and d.

In cycle i+2, en5 is asserted so that R5 is loaded with the value of (a+b)*c. In the same cycle, s3 chooses the right-hand side input, and en3 is asserted so that R3 is loaded with c+d. R4 still holds a+b.

In cycle i+3, en6 is asserted so that R6 is loaded with the value of (a+b)*(c+d). The sequence in which the registers are loaded with the different values is shown in the timing diagram included in Fig. 6.4.

Implementing the Data Sampler

1. Create a new project and add the sources found in the folder named task5.4A.
2. The actual algorithm to generate numbers in Data Source is based on simple addition / subtraction. Look at the source code for the Data Sampler. The data path is defined in various processes, corresponding exactly to the schematic of Fig. 6.4.
3. Based on the timing diagram shown in Fig. 6.4, implement the nextStateLogic and ctrlOut processes of the FSM, where you have to define the necessary states and assign the values of the control inputs R1_sel, R2_sel, R3_sel,

R1_en, R2_en, R3_en, R4_en, R5_en and R6_en, and the output outValid in each state. Check with the supplied testbench that your implementation works.

Implementation of Controller for Data Sampler

1. Upload your source code for the Data Sampler component. Answer Quiz 1, Question 17.

General Tips and Techniques

The last example we looked at, the Data Sampler, is the most realistic example we have considered up to now, and is to be considered as showcasing best practice when writing RTL. Some of the key points to keep in mind when writing VHDL code (or Verilog for that matter) are summarised below.

Points to remember:

- Always remember that VHDL is not a programming language that is meant to be compiled and run on a processor. Its purpose is to model hardware, and any statement you write has hardware implications. The only exception is when writing testbenches. Distinguish between synthesisable constructs, and unsynthesisable constructs which have no meaning in hardware.
- Because VHDL is meant to model hardware, concurrency is built in to the execution thread. Signal assignment statements and process always execute concurrently.
- Signals are the main mechanism by which information is exchanged between communicating processes, and signal assignments always incur a delay.
- Within a process body, statements execute sequentially, and variable (declared internally within a process) assignments incur no delay.
- Inferring combinational logic requires that you define the value of the driven signal in every branch of conditional clauses, either IF-THEN-ELSE or CASE structures. Sometimes the most convenient way to do this to assign some default values at the beginning, and overwrite those default values in select branches. The WHEN OTHERS clause is also useful in this regard.
- Sequential logic is inferred by incomplete assignments, i.e., the opposite of the above, deliberately omitting a selection branch in conditional logic. However these have to be within strictly defined templates, and a level-sensitive condition for assignment results in a latch, and an edge-sensitive condition results in a flip-flop. The templates also generally support an asynchronous (or synchronous) reset, and a clock-enable.
- Do not mix combinational and sequential logic in the same process.
- Always adopt the templates introduced in this assignment. These include concurrent constructs for multiplexers that can be defined outside process bodies, behavioural style templates for inferring different types of combinational elements and behavioural style templates for sequential logic elements and behavioural style templates for implementing finite state machines. Processes implementing combinational logic should always have all read signals, for example signals that appear on the right-hand side of an assignment or are read inside a conditional clause, need to be in the sensitivity list. Only the clock and any asynchronous resets should appear in the sensitivity lists of processes modelling sequential logic. Synchronous resets are not needed in the sensitivity list as they are read inside the clocked body.
- Adopt a modular approach so that all major components are implemented as separate entities, and within the architecture body for a given component, all major subcomponents are modelled as separate processes. Avoid very long process bodies. Modular in this context essentially means that each logical component, be it a process, architecture or perhaps function or procedure, should do one thing and one thing only. Do not mix funtionalities within the same component. For example, don't have a module that either adds or shifts depending on a control signal value. There is no component in hardware that can add and shift.
- Always design your circuit (assuming it is of non-trivial complexity) to comprise a datapath and a controller, where the controller is a FSM. Wherever possible, try to draw a state-transition diagram or a timing diagram to aid in constructing the FSM.
- For numeric operations, use the NUMERIC_STD package from the IEEE library
- The top three rules to facilitate the generation of hazard free logic are, in order: avoid latches, avoid latches, and avoid latches.

Summary

The following points were covered in this section.

- Synchronous systems have storage elements or registers, i.e., edge-triggered flip-flops, that are updated once every clock cycle. For flip-flops to be updated on a clock edge, the data needs to be ready at the input one setup time before the clock edge and held steady for one hold time.
- Inputs should be registered, i.e., loaded into a register, to ensure they only change at the beginning of the clock cycle and remain steady for the remainder of the cycle to avoid glitches.
- Registering a signal effectively introduces a one-cycle delay between the input and output of the register.
- Detecting signal transitions - i.e., rising or falling edges - on a signal is accomplished by taking the exclusive-or (XOR) of the signal and a one cycle delayed (i.e., registered) version of it.

- 许多数字系统需要对数据进行算术运算。这样的数字系统总是被组织成具有包括算术、存储和开关元件的数据路径，以及对数据路径中的元件的操作进行排序的控制器。
- 现在您应该能够高效地使用 Vivado IDE 进行仿真。在作业 2 中，将探索更多功能以完成整个设计工作流程。