

Design of large-scale systems

1 Universal system

2 Chained systems

3 Communication of asynchronous systems

4 Protocols

5 Asynchronous pipelines

6 Classification of design methods for asynchronous systems

Design of large-scale systems

When working with digital systems, it is necessary to master two main tasks - the specification of the system and its subsequent design. System specification means a description of its activity, regardless of the means by which the required activity is performed.

A **hierarchical description** is used for large systems. The system is divided into subsystems; the subsystem is divided into functional blocks, which are further divided into individual components. The simplest component is understood to be a logic device (gate), realizing one of the elementary logical functions - OR, AND, NOT and XOR. The last (lowest) level of elementary electronic components and their interconnection - the so-called "physical level" consisting of individual transistors - is not considered here. It is explicitly a matter of the integrated circuit manufacturer and his technological processes.

Working with standard function blocks (decoders, multiplexers, flip-flops, counters, registers, etc.) substantially facilitates the design. Function blocks, however, may be much more complex units, conveniently defined for a designed system.

Such a solution allows the design to proceed **top-down** from the division of systems into subsystems, which can be further divided into functional blocks. The disadvantage of this approach is that optimal design at higher levels can lead to problems at lower levels - for example, it requires function blocks or components that are not available.

The other process is called **bottom-up**. In the case of a bottom-up approach, it is based on existing components and function blocks, which are interconnected to fulfill the assignment. It is therefore guaranteed that suitable components are available. The disadvantage is that it is not possible to guarantee in advance that the optimal design for the lower levels will not lead ultimately to a system, whose activity is not optimal and possibly deviates from the specification.

The disadvantages of both methods can be suppressed by a **combined** procedure, which is based on the principle of "top-down", but at the same time it is checked whether the chosen variant of division at a higher level allows a suitable implementation of a lower level. If not, another option will be tried. The task is probably ambiguous; usually the system can be designed in several variants, all of which meet the requirements. The selection of the most suitable variant then takes into account other criteria, such as price, reparability, the possibility of easy modification in the future, etc.

1 Universal system

Large-scale systems are designed from **functional blocks** (FBs) - interconnected and cooperating. The data gradually passes through a number of function blocks and is gradually processed. This creates a **data path**. However, the individual function blocks must also be controlled in some way to perform the required operations and to transmit data at the right time. To do this, there must exist their **control**.

The individual function blocks can be both completely single-purpose and very universal - for example, the arithmetic-logic unit of a computer. We will consider universal blocks. Figure 1 shows such a function block. The individual inputs and outputs - **control and status** signals - have the following meaning:

WR - register input
 EN - writing disable/enable
 OP - selection of data operation
 ST - processing status
 OE - disabling/enabling data output, three-state output
 DI - data input
 DO - data output

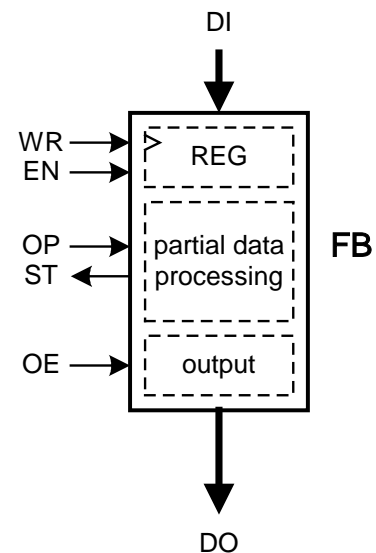


Fig. 1: Universal function block

It is assumed that the writing pulses *WR* are generated at regular intervals (clock pulses) and are not interrupted. The *EN* input is used to enable or disable writing - see the chapter "Triggers and metastability". The *OP* input determines the **operation** to be performed by the FB. The signal *ST* indicates the status of the FB. Basically, there are three: the readiness for the new operations, operation running, and operation completed. This information is important for the cooperation of several FBs. A three-state output is enabled via the *OE* input, so that the FBs can be grouped into a bus structure. All these inputs and outputs do not have to be implemented for all FBs - but here we will assume that the FBs are fully equipped.

The function blocks can be connected to the bus with their three-state outputs as required, and by arranging them, a **data path** is created. A chosen operation is started by writing into the input register, and during and after the data processing the **status signal** is issued - it can signal the end of the operation, but also its closer details - e.g. error in the function, data out of range, etc.). The data from the output of one block is then written to the register of the next block and further processed. Everything takes place under the control of central control circuits and with precise timing - see Fig. 2.

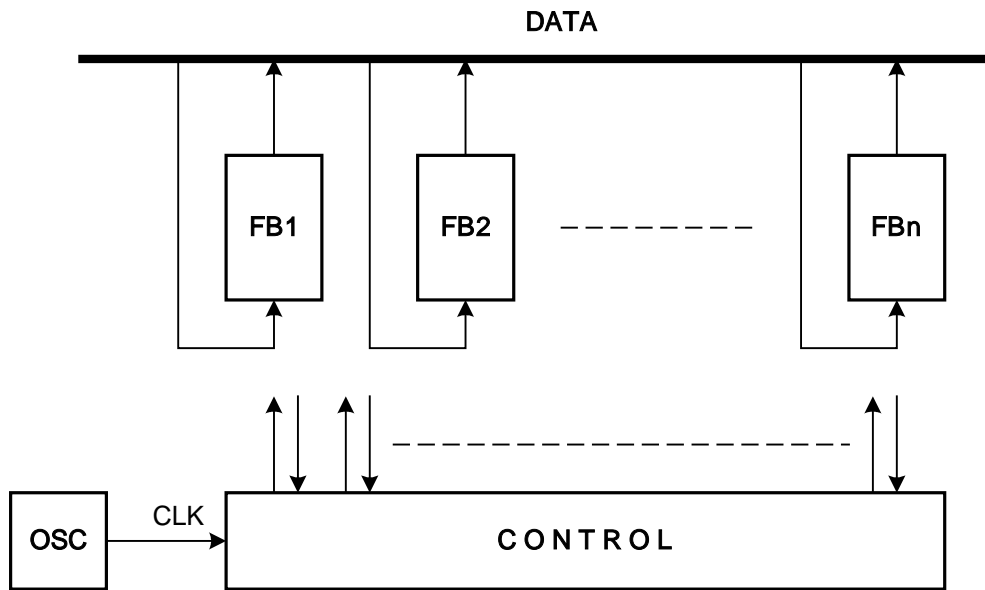


Fig. 2: Group of universal FBs

In the simplest case, the timing is fixed, derived from the parameters of the slowest FB - but then any faster FB with faster operation will not be fully utilized. Even with the same FBs, different operations may take different time to perform.

Total acceleration is possible through **variable timing**. The timing is adapted to the **state of the** individual function blocks. The simplest method is reporting the completion of the operations in the function blocks. The control block then simply delays the start of operation operations with the other FBs until the current operation is completed. Because the entire system is synchronized by the oscillator, all time intervals will be a multiple of the oscillator clock period.

However, the bus itself limits the use of function blocks, as their three-state outputs allow only one connection at a time - only one block works at a time and the rest is unused. However, if the FBs can be divided into two groups connected to two buses, two FBs can be active at the same time - each from one group. It is also possible to increase the number of groups and thus buses. Then a larger number of FBs will be used at the same time. But it is necessary to provide the **bus switches**, so that data can be transmitted between FBs from different groups. CMOS switches are used for this (see chapter "Digital components and technologies"). Each bus conductor has its own switch. Fig. 3 shows the interconnection of the four buses A, B, C, D. The CMOS switches are marked symbolically. By suitable control of groups of switches it is possible to connect AB buses and simultaneously CD, or AC and BD, or AD and BC.

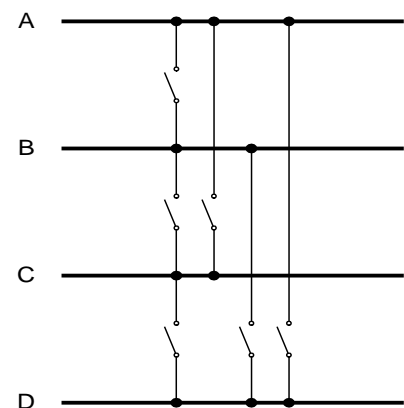


Fig. 3: Interconnection of four buses

The described system with its possibilities of control of individual FBs and interconnection of buses can be very universal. To change the overall function, it is sufficient to change the operation of the control circuits without the need to interfere with the FB system. In this way were constructed **processors** in computers called **CISC** - Complex Instruction Set Computer. The processor executes the instructions sequentially, and each instruction determines the operation of the **control block**. This also determines the selection and operation of the processor's function blocks. After completing the instruction, other instruction is read from the program memory, etc.

The disadvantage of the described arrangement is its **low efficiency**, measured as the number of FBs currently in operation in relation to their total number. The result of low efficiency is low data processing speed. In many applications, such as processors, digital signal processing devices, etc., a different FB arrangement and control is preferred.

2 Chained system

An effective method for accelerating the operation of the system is the introduction of a **chained structure** - "**pipeline**". The basic idea is shown in Fig. 4. The original system is divided into sections in suitable places. In the first approximation, we can imagine the sections as combinational circuits, although the method can be well applied to sequential circuits. The delays of individual sections indicated in the figure are generally different. The boundaries cannot be made at any place, but rather between the functional blocks, integrated circuits, connectors, etc..

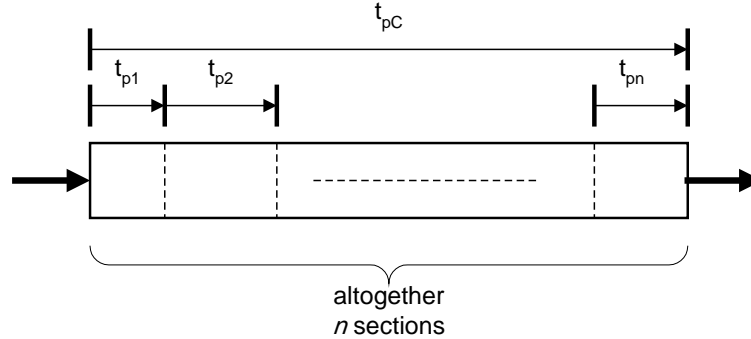


Fig. 4: Dividing the system into sections in cascade

The sum of the delays of all sections gives the total delay t_{pc} :

$$t_{pc} = \sum_{i=1}^n t_{pi}$$

Between sections are inserted **registers**. These serve to remove potential false impulses and to store data on the inputs of sections for a short time - see Fig. 5. All registers are loaded at the same time by common clock pulses.

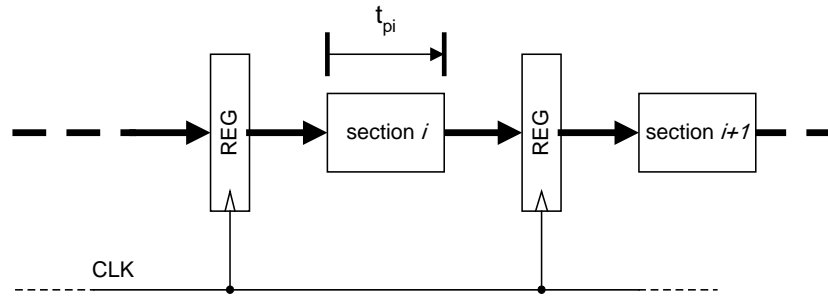


Fig. 5: Inserting registers between sections

The registers form a temporary memories or **buffers** that isolate the sections from each other for the duration of the clock cycle. Let us follow the input data stored in the first register and processed by the first section. Thus, they are available for the second section, and by the next clock pulse they are stored in the second register and further processed. At the same time the first section accepts and processes next input data. This way, input data is shifted through the chain of sections and gradually processed. All sections work in every clock cycle and the data processing speed is therefore high. The operation of this structure resembles some liquid transported through the pipeline - from this originates its name **Pipeline structure**.

The timing of the signals in one section is shown in Fig. 6. The transit time of the signals in sections t_{pi} also respects the stabilizing of possible false impulses. The signals at the register input must be stabilized at least by the setup time t_s **before** the active edge of the clock pulse CLK . The flip-flops in the register toggle after the time t_{pCQ} from the edge CLK . The sum of these three times determines the minimum period of clock pulses. However, the individual sections do not have the same delays and therefore the period of the common clock pulses must be determined according to the section with the **longest delay** t_{p_max} :

$$T_{CLK} \geq t_{p_max} + t_s + t_{pCQ}$$

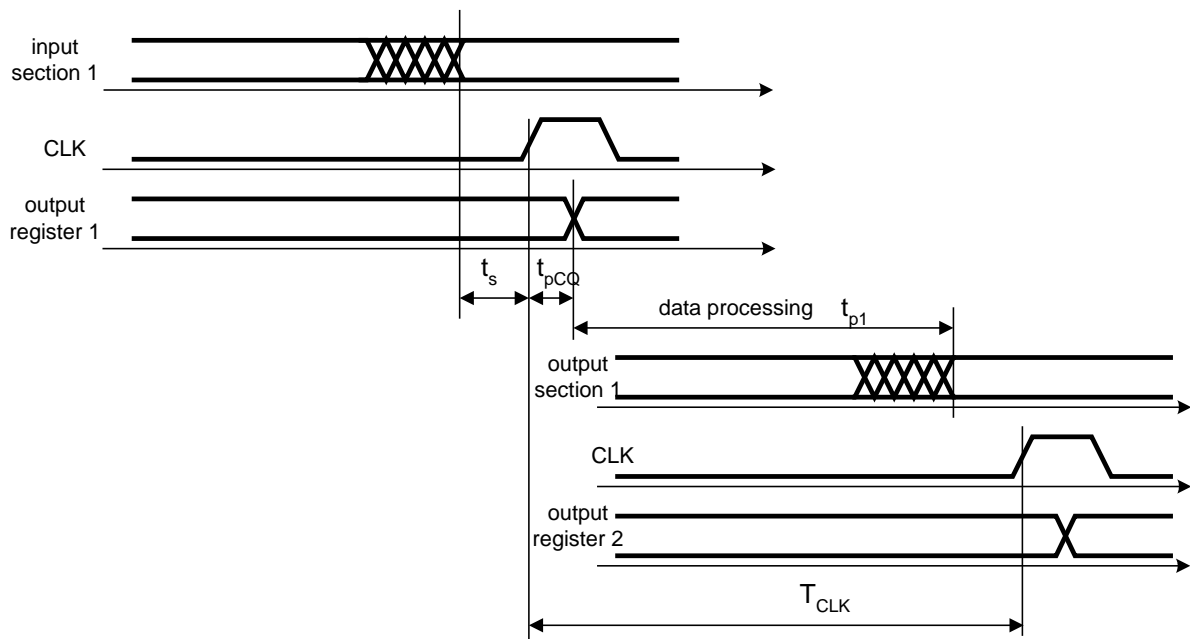


Fig. 6: Timing of signals in one section

It is a synchronous system in which the timing of signals is critical - especially delays in connections and possibly repeaters of signals in the distribution network of clock pulses (see chapter "Synchronous sequential circuits").

At the maximum frequency of the clock pulses (i.e. the minimum T_{CLK}), the transit time of the first data through the whole string t_{DD} is given as:

$$t_{DD} = N \cdot t_{p\max} + N(t_s + t_{pCQ}) ,$$

which is even longer than the delay time of the original (un-chained) system. Thus, pipelining does not seem to bring any improvement. Note, however, that the data is supplied to the input of the original (un-chained) system at **regular intervals**. The supplied data is processed and after the time t_{pc} sent to the output. Only now can new input data be brought in. Thus, the input data can be delivered with a period of at least t_{pc} . In pipelined processing, data can be delivered with a T_{CLK} period. The output data will change with the same period. In the continuous processing of the data supplied to the input clock by clock, the data at the output is based on the period T_{CLK} , which is a much shorter time than the time t_{pc} in the original system. It is then possible to calculate the **acceleration** of the operation due to pipelining. It is the ratio of the time interval between the two following data words at the output of the original (un-chained) system and at the output of the pipeline. **Acceleration** a_{pl} is given by the relation

$$a_{pl} = \frac{\sum t_{pi}}{T_{CLK}} = \frac{\sum t_{pi}}{t_{p\max} + t_s + t_{pCQ}}$$

As an example of chaining efficiency, assume a system with five sections with a delay of 7 ns, 7 ns, 6 ns, 8 ns and 10 ns. Flip-flops have $t_s = 2$ ns, $t_{pCQ} = 4$ ns. The period of clock pulses then results in 16 ns ($f_{CLK} = 62.5$ MHz). Acceleration $a_{pl} = 38/16 = \mathbf{2.375}$. This is certainly a significant result.

In the ideal case, i.e. with the same delay for all N sections and with registers with zero delay, the result is $a_{pl} = N$. The reality is always a little worse. A necessary condition for the successful chaining is **continuous supply** of input data with each CLK period.

Very often, the principle of pipelining is used in processors and digital signal processing systems. If the instructions are executed by the processor continuously one after another and are of the same length, pipelining is very effective. However, if there is irregularity in the supply of instructions, some time is lost.

The sections in the chain are just simplified function blocks from the previous text (see Fig. 1). The simplification is due to the tight interconnection of the sections, so that three-state output circuits drop out. Other signals (EN, ST, OP) are also not needed in simple chains and then there is no common control block.

However, the control of individual sections is needed for more complex pipeline structures that occur in practice. There are sections with branching or joining functions, there are also feedbacks. Several variants are shown in Fig. 7. Registers are marked as R and data

processing circuits as Z. The distribution net of clock pulses is not drawn for simplicity - it would lead to all registers simultaneously.

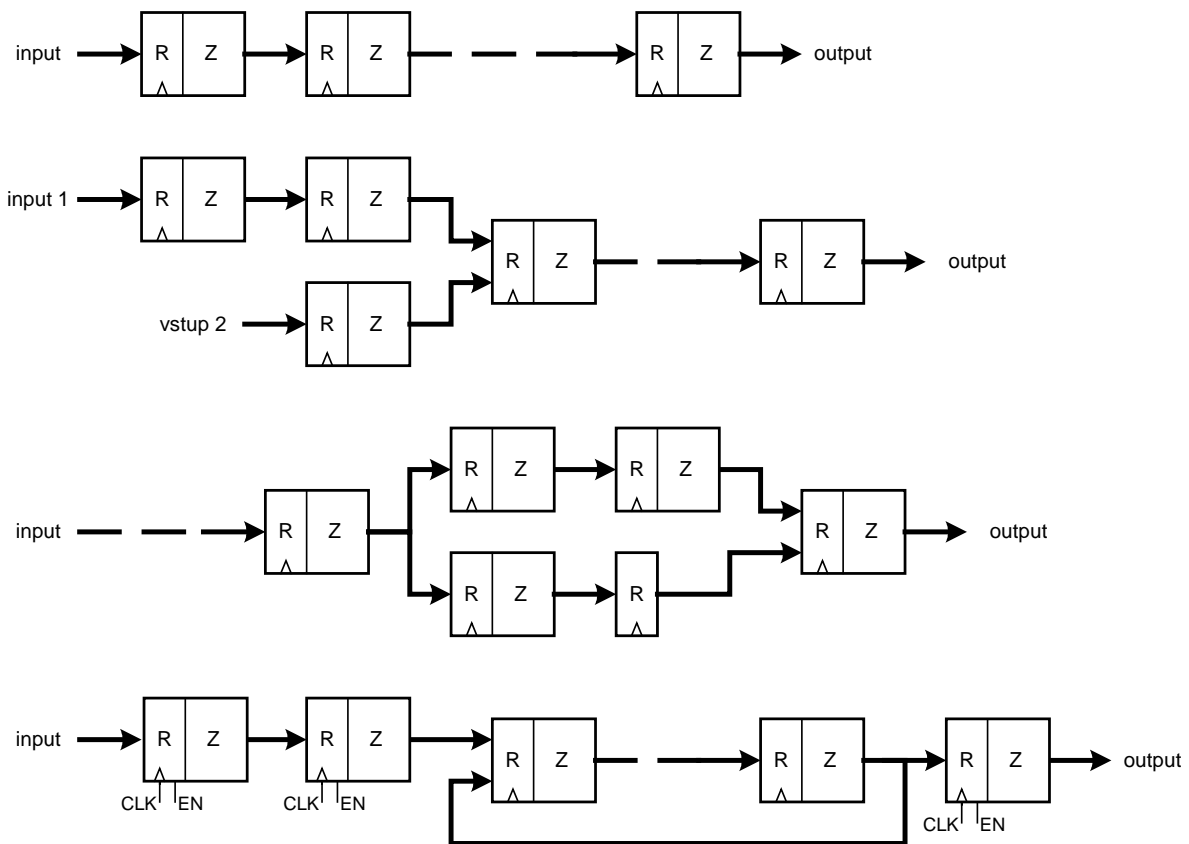


Fig. 7 : Examples of different shapes of pipelines

The first image above shows a simple **linear pipeline**. The second figure shows the collection of data from the two inputs and their **merging**. In the merging point, data from input 1 is one clock behind data from the input 2.

The third picture shows the **division of a** data path into two parts in a "**fork**", their different processing, and then merging again in a "**join**". The upper path leads through two stages of processing and the lower originally through only one. Therefore, a single register is inserted in the lower path so as to **equalize the number** of clock cycles of the two parts of the data path when they are merged.

The last figure shows a pipeline with a **feedback loop**, which is a structure used for iterative calculations and also in processors. In more complex forms, it may be necessary to delay the passage of data in some sections so that the individual data is correctly ordered.

Some signals do not have to be processed in the given section - they only proceed to the next section. It is then necessary that such signals pass through **the same number** of registers as signals that are processed. This ensures the simultaneous validity of all signals at the output of the section in a given clock cycle - see Fig. 8.

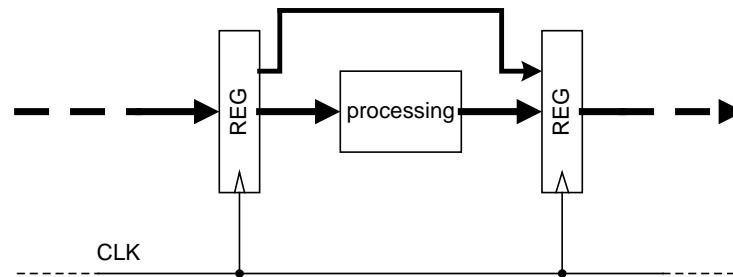


Fig. 8: All signals pass through the same registers

In some cases it is necessary to control the activities of the section. It is possible to connect control circuits to the function blocks as was shown in the universal solution (Fig. 2). However, a simpler arrangement, shown in Fig. 9, may suffice. The control signals are introduced into the pipeline at the input, but are used as needed in subsequent sections. They pass through the same registers as the processed data. However, this simple method has one limitation - the operations of individual sections must be determined in advance as soon as the data enters the pipeline.

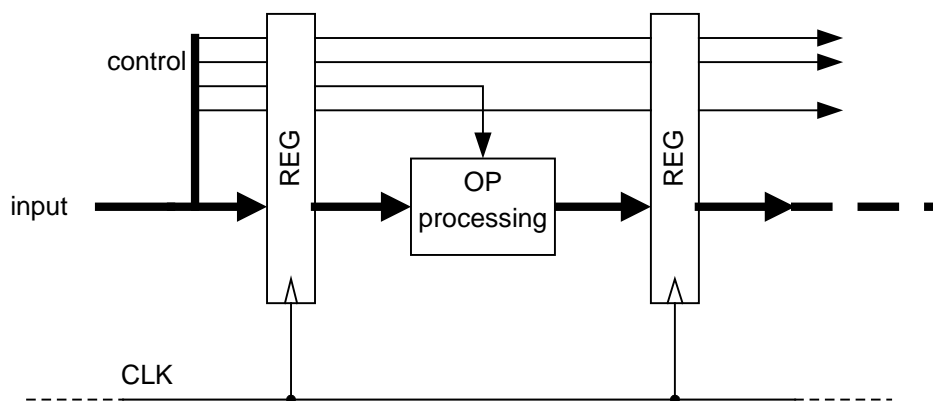


Fig. 9: Control signals passing through a chain

3 Communication of asynchronous systems

Errors can occur when transferring data between two digital systems. At the input, the data is always stored in a register composed of flip-flops. However, improper timing of signals can cause metastability.

Metastability can be eliminated by introducing **samplers** in place of single flip-flops. This principle can be successfully used if the individual signals at the system input are mutually independent and can change at any time. Typically, these can be signals from push-button switches, limit switches, sensors, etc. It is necessary to take into account the specific behavior of the sampler, which suppresses metastability, but introduces an unnecessary delay of the signal by one period of synchronization pulses. In the above cases, a possible delay of one period is irrelevant.

However, if the signals at the input have the character of a vector - i.e. **groups of signals** that belong are related (expressing a number, code, etc.), the delay of some signals in the group will be quite significant. For a transitional period, combinations of values may exist that do not belong to the previous state or to the next state, but form a kind of intermediate value. E.g. when the binary number changes from 0111 to 1000 (which is only the minimum numeric change from the decimal value 7 to 8), all bits change. If the order of changes in the individual bits is unknown, any temporary combinations of the values of the four bits are possible, so there are 6 possible erroneous intermediate states. A faulty intermediate state will not take the form of a short false pulse, but will last a full clock cycle, just as the correct state. These phenomena are called signal **races**. Therefore, the undesirable consequences of races **cannot be eliminated** by the sampler.

However, enhancement of reliability of data transfer between two systems that are not synchronized with each other is needed. The issue is called "**Clock Domain Crossing**" - **CDC**. The solution is the introduction of a "**handshake**" - **HS** - in mutual communication. In addition to the data signals, it is necessary to introduce other signals for controlling communication.

For a pair of systems that communicate with each other, we will distinguish between the data **sender** and the data **receiver**. In addition to data signals, there are other - usually two - control signals. They are most often called **RQ** - Request and **AC** - Acknowledge. One of them is generated by the data sender, the other by the data receiver. It should be emphasized that not only the data but also the control signals are non-synchronous with each other.

The initiator of the data transfer can be both the data sender and the data receiver. There are two variants of the meaning of **RQ** and **AC** signals and two types of operation:

1. Transmission initiated by **data sender**.

Once the data sender has the data ready, it prompts the receiver to receive it by the **RQ** signal. As soon as the receiver receives the data, i.e. loads it into its **input register**, it confirms it with an **AC** signal. The sender can then prepare next data. The data is therefore pushed from the sender to the receiver and this arrangement is called "**Push Channel**".

2. Transmission initiated by the **data receiver**.

As soon as the data receiver can receive the new data, the **RQ** signal prompts the sender to output it. As soon as the sender makes the data available, it confirms it with an **AC** signal. The receiver can then load the data into its input register. The data is therefore pulled by the receiver from the source and this arrangement is called "**Pull Channel**".

The situation is illustrated in Fig. 10. The dot on the data arrow indicates the type of the channel.

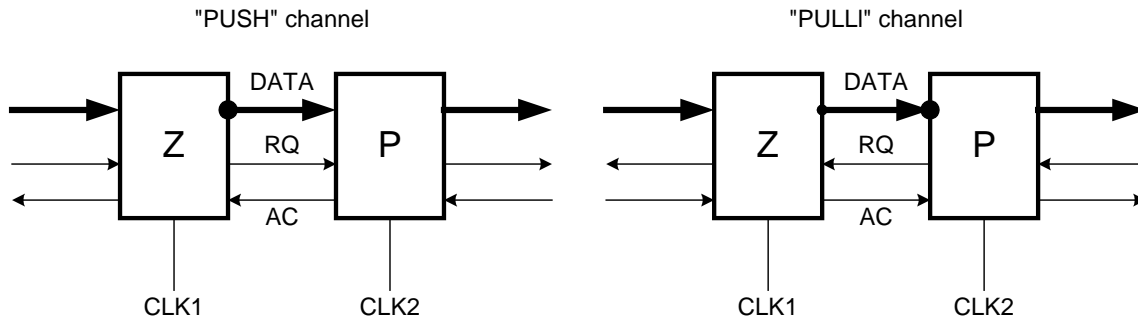
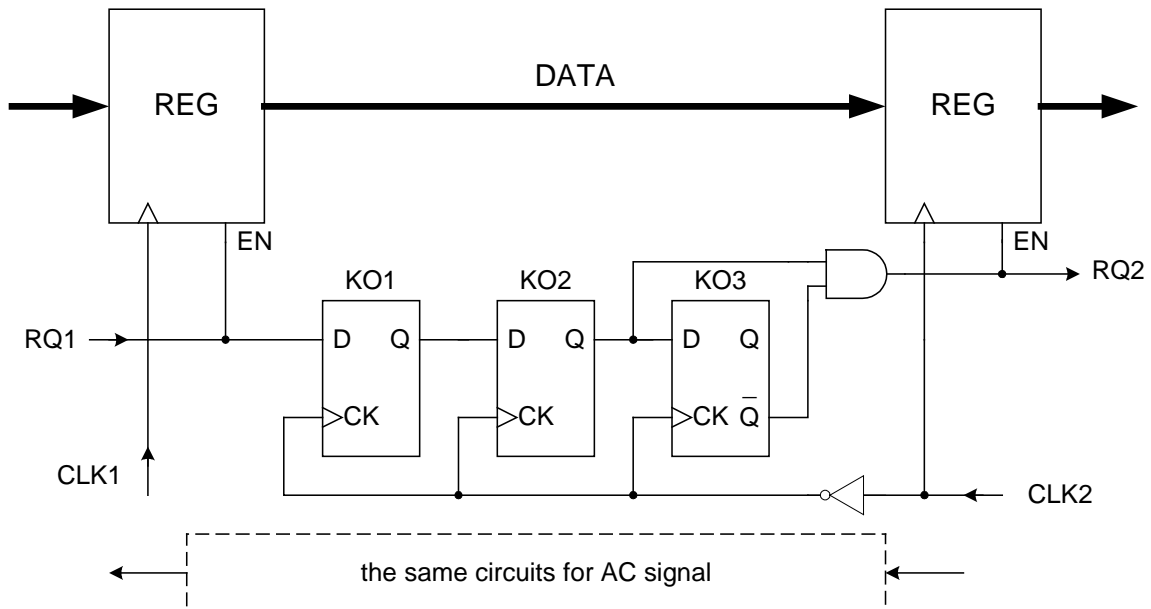


Fig. 10: "Push" and "Pull" channel layout

Push or pull communication channels are not limited to large systems; they are very frequent also between the pipeline **sections**. This is common with **asynchronous pipelines** that do not have a common clock distribution (see next text).

In case that both sender and receiver are synchronous, but with independent clock generators, the processing of RQ and AC signals is complicated by the fact that they are generated in one system and introduced into another, with which they are not synchronized - this creates a risk of metastability in subsequent circuits. For reliable operation, it is necessary to synchronize the RQ and AC signals to the clock pulses **at the point of action**. The solution is shown in Fig. 11.

Fig. 11: Synchronization of RQ with $CLK2$

At the output of the data sender and at the input of the receiver, the registers are equipped with an EN input for blocking or enabling loading. The RQ signal is routed through three flip-flops - KO_1 and KO_2 form a **double sampler**, KO_2 and KO_3 work as a **raising edge detector**. Loading of the sender register is synchronous with CLK_1 . The KO group is

synchronous with CLK_2 and the sampler ensures a smooth passage of the RQ signal between the two clock domains. Finally, the output of the edge detector is an EN_2 pulse of the length T_{CLK_2} , which allows data to be written to the receiver's register during one CLK_2 period. This pulse is shifted by half a clock cycle (see inversion in the CLK_2 supply), so that the signal EN_2 is guaranteed to be constant at the moment of the edge CLK_2 - this is necessary to avoid **metastability** in the register. The AC signal in the opposite direction is not marked in the figure for simplicity, but would be treated in the same way as the RQ signal shown.

4 Protocols

There are several typical waveforms that are standardized as **protocols**. Figures 12 and 13 show the waveforms for the "push" channel, which is a much more common arrangement than the "pull" channel. Arrows indicate cause and effect.

Figure 12 shows a **four-phase "early"** protocol. After preparing the data, the receiver is requested to accept it (i.e. load to the register) with the signal $RQ = 1$. This is followed by acknowledgment of data reception by the signal $AC = 1$. Immediately afterwards, the data expires - therefore **"early"**. The data sender then terminates the RQ and the data receiver terminates the AC . This allows the data source to prepare new data - it has a relatively long time to prepare it. The combination of the values of both control signals determines the **four phases** of the transmission process: 1. waiting for a request, 2. request to take over data, 3. taking over data, 4. completing the cycle.

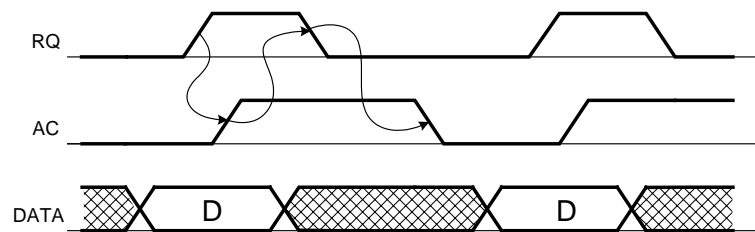


Fig. 12: Four-phase "early" protocol

Figure 13 shows a four-phase **"broad"** protocol. The waveforms differ only in the period of validity of the data. Here it is valid until the AC ends, i.e. for a long time (hence "broad").

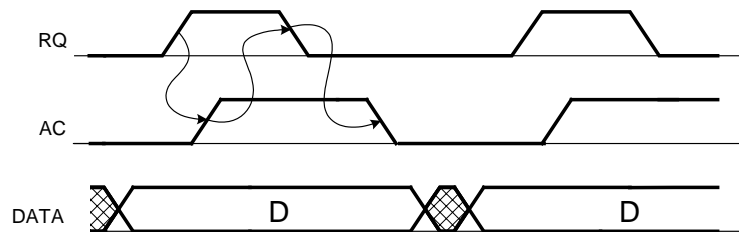


Fig. 13: Four-phase "broad" protocol

Both 4-phase protocols have their justification. For **"early"**, valid data exists only shortly after the leading edge of the AC , and if the entire cycle is not to be delayed, it must be stored in the

input register of the data recipient as soon as possible - i.e. the data register should be **edge controlled**. On the contrary, the data source has almost the entire cycle available for the preparation of new data. Its output is insignificant during this time. With the "**broad**" protocol, data exists for a relatively long time, and there is enough time to store it in the recipient's input register - so a simpler latch (level-controlled register) is sufficient - writing to it is typically controlled directly by the *AC* signal. But if the cycle is not to be significantly slowed down, the data source has only a short time to release new data.

In the case of Fig. 14 it is a **two-phase** protocol, also "**transition signaling**". Request and acknowledge are signaled by a change of signal, regardless of whether $0 \rightarrow 1$ or $1 \rightarrow 0$. After the data is ready, the receiver is prompted to accept it by changing the *RQ*. The receiver then indicates acceptance by changing the *AC* and by this is the transfer completed. By "accepting data" is meant to load data into the input register. Important feature of this protocol is that both control signals are changed only once in a single data transmission - the demands on the speed of these signals are smaller than in the 4-phase protocols.

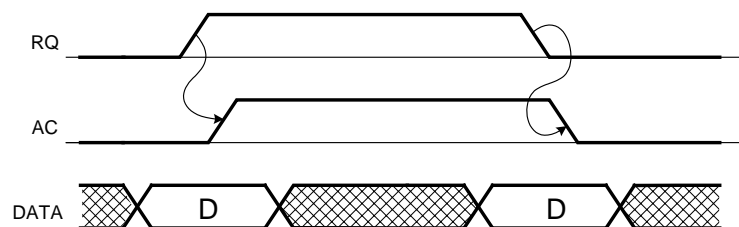


Fig. 14: Two-phase "push" protocol

Until now, all channels had one signal per bit of transmitted data. However, an arrangement **with two signals per bit** is also possible. One signal (D_{i_1}) signals state 1 with a 0-1-0 pulse and the other (D_{i_0}) signals state 0 with a 0-1-0 pulse. Simultaneous pulses in both signals are prohibited. Since the data is transmitted in the form of pulses, it is not necessary to accompany them with an *RQ* signal. This is easily derived in the data receiver as the logical sum of all signals. The register can be composed of latches RS, which is the most economical variant. An impulse on one data signal turns the latch to one state and an impulse on the other signal turns it to the other state - see Fig. 15.

The disadvantage is the double number of data conductors, but the advantage is a significant simplification of all circuits, better speed and reliability. By added simple hardware it is possible to detect errors.

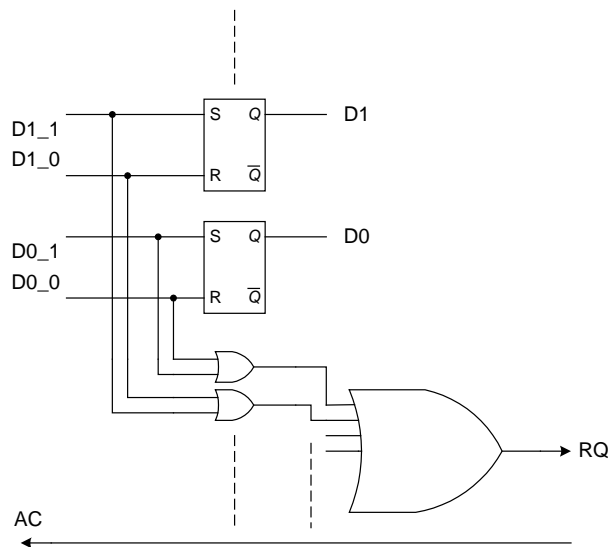


Fig. 15: Two-wire per bit transmission

5 Asynchronous pipelines

Pipelines are commonly found in synchronous versions. In the continuous processing of the data supplied to the input clock by clock, the output data is output with a period of synchronization pulses, which is a much shorter time than the data processing time of the original system. Thus, the pipelining significantly accelerated the operation of the system.

However, if we monitor the time for processing **one** data word, i.e. the total time for its passage through the string, the situation is worse. The time is given by the *CLK* period and the number of sections. Due to the common timing of the synchronous circuits, the *CLK* period must be determined from the **slowest** section of the chain. But then, if the input data is fed irregularly, there is an unnecessary delay in the faster sections.

In asynchronous chains, the write pulses for the section input registers are not generated centrally, but **locally** in each section and are independent of each other - see Fig. 16. They are generated by a **section controller** (CTL), with which each section is equipped. The controller controls both the transfer of new data from the previous section and the transfer of processed data to the next section. The controller always takes the form of an asynchronous **sequential** circuit.

Once the data in the section is processed, it is passed on. Thus, faster sections are not waiting for centrally generated synchronization pulses. The result is a faster passage of data through the chain in the event of an **irregular** supply of input data. However, with regular data delivery, as with a synchronous chain, it is necessary to respect the slowest section. But irregular operations are very common, e.g. in processor circuits.

With the asynchronous variant, the distribution of synchronization pulses throughout the system is eliminated. Instead, adjacent sections are interconnected by signals that ensure that the data is moved at the right time.

The division of the entire asynchronous system into small sections and their pipelining is a completely logical intervention aimed at increasing the speed. However, it has another effect, which is to significantly simplify the design. Instead of a large asynchronous system that is difficult to solve, only small units need to be solved, which is much simpler. **Pipelined systems are therefore a viable method for solving complex asynchronous systems.**

An important part of the solution is the mutual communication of sections in the chain. It is always in a form of Request - Acknowledge ("handshake"). The section that is currently the initiator of the transmission sends an *RQ* signal. Then the data is transferred. Finally, the transmission is confirmed by an *AC* signal from the receiving section. The sum of the transmitted data and signals controlling the communication is called the **channel**. We will assume that we will deal with a "push channel".

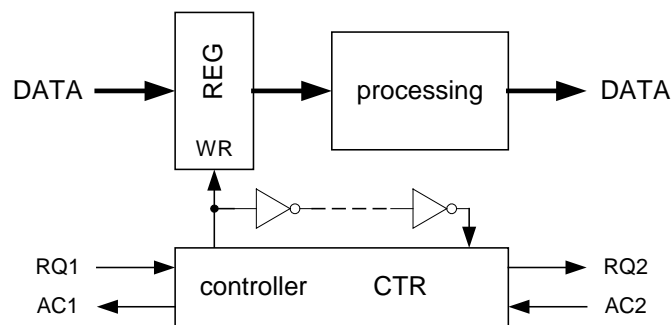


Fig. 16: Asynchronous pipeline section

Each pipeline section contains a **controller** that generates a write pulse to the section's input register and provides a link to the previous section and the next section of the pipeline. When designing the controller, it is necessary to respect the selected communication protocol between adjacent sections and also the design of the register - whether it is edge- or level-controlled. There are a number of recommended connections in the literature, differing in complexity and function. What is important is how far the controller is able to control the transfer from the previous section independently of the transfer to the next section. Of course, complete independence is not possible, as each section may pass just processed data to the next section only when it is ready to receive new data from the previous section.

After writing the input data to the register, the data at the output of the section changes with a certain delay; the total delay is given by the delay of the register and the processing (combinational) circuits. The section is ready to receive new input data only when its just processed output data is inserted into the register of the next section - otherwise the data would be **lost**.

Assume that data transfers take place according to a four-phase "**push**" protocol with *RQ* and *AC* signals. The waveforms correspond to the "**broad**" protocol from Fig. 13. The state diagram showing the operation of the section controller is shown in Fig. 17. The notation common for sequential circuits in **BM** mode is used (see chapter Asynchronous sequential circuits), namely for both input and output variables. A **Moore** - type sequential circuit is assumed.

The controller consists of two circuits, partially independent of each other. The first circuit with states S_0 and S_1 cooperates with the previous section and responds to signals RQ_1 , AC_2 , and generates AC_1 . The second circuit cooperates with the next section and responds to signals K_1 , AC_2 , and generates RQ_2 . The K_1 signal is delayed by a time that reliably overlaps the delay of the register and combinational circuits. Then the communication with the next section is started.

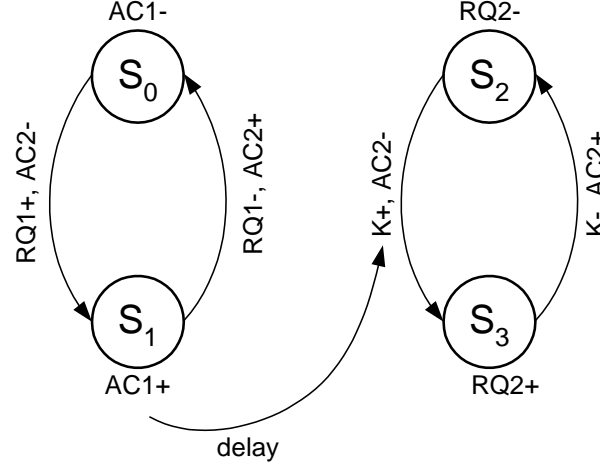
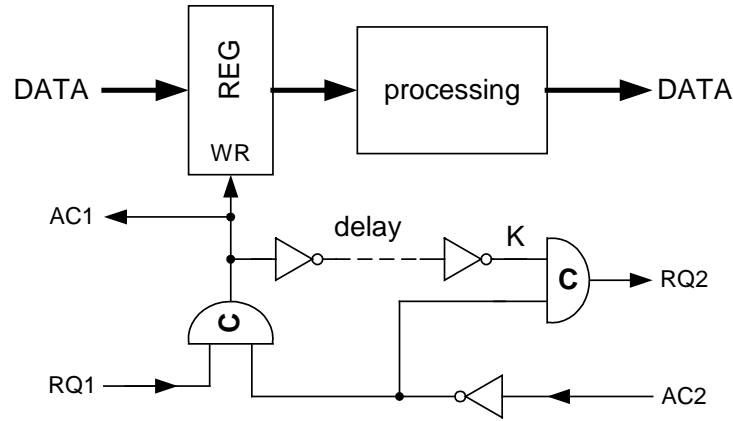


Fig. 17: Flowchart of the controller from the example

The implementation of this activity is provided by the circuit according to Fig. 18. It is based on C-elements, which are the core of all controllers mentioned in the literature. The delay of the K signal is realized by a delay line, which can be easily assembled from a series of gates of suitable length.



About no. 18: Controller diagram from the example

The initial state is "0" on RQ_1 , AC_1 , RQ_2 , AC_2 . Signal changes will be marked using BM notation. When $AC_2 = 0$ and RQ_1+ , AC_1+ occurs, thus writing new data to the register and starting the processing. The previous section will release RQ_1- , otherwise nothing will change yet. When the delay elapses, there will be $K+$ and thus also RQ_2+ . The controller of the following section issues AC_2+ and writes data in its register. Now, when AC_2+ , appears AC_1-

and the cycle is completed in this section. Then it is possible to write next data to the input register.

Throughout the chain, RQ-AC communication runs from left to right and from right to left, while the data is shifted to the right. Interconnection of controllers ensures reliable data processing without data loss, even when a section is stopped (blocked).

The described controller was simple. More complex controllers are needed for some chains as marked in Fig. 7.

6 Classification of design methods for asynchronous systems

Design methods are classified according to how they respect delays in components and connections. Asynchronous systems have their special features over synchronous systems, which is reflected in their design. The basic difference is the **absence of synchronization** pulses. When designing a synchronous system, individual delays within one cycle are checked and, if necessary, the length of the tact or frequency of clock pulses are determined so as to meet the requirements of reliable operation - refer to the signal timing in a synchronous sequential circuit in the chapter "Synchronous sequential circuits". In asynchronous systems, the concept of clocks does not exist. The signal travels through the circuit at a speed corresponding to the delay along its path. There is no need to check the delay with respect to the clock pulses. Due to the fact that several signals are processed in the circuit at the same time, it is necessary to control their **mutual timing**, which can be quite a difficult task. Its solution affects the reliability of the whole system.

The term **isochronous branching** will appear in the description of the design methods. It is a branching whose all branches have the same delays - often the delays are compensated by different meanders of signal traces. If the delays in the traces are the same, they can be covered under one value and "shifted" to the device that feeds the branching. This greatly simplifies the situation. We distinguish three main approaches to design in terms of timing:

SI - Speed Independent. There are **unknown** (but limited) delays in components and **zero** delays in connections. The method requires all branches to be isochronous and also known order of changes of input signals. It is suitable for small systems with limited speed. It is not suitable for integrated circuit design where delays in connections cannot be neglected.

DI - Delay Insensitive. There are **unknown** (but limited) delays in components and **in connections**. It is a very robust design, but difficult to implement. All circuits composed only of **C-elements** and **inverters** are **DI**.

QDI - Quasi DI. With the exception of some isochronous branches, QDI is suitable to DI. It is the method that best suits reality. It basically uses the principle of question-answer (RQ-AC). Different signal sequences in the circuit are tolerated.