Microprogramming Choices Explained (Part 1)
The Microprogram Control Unit
– Addendum –

Two flavors of programmability – it's about how to make changes
Programmability allows for altering and updating again and again, even at the customers. You can have this effect in two ways, by programming microcontrollers or RISC processors or by programming CPLDs or FPGAs.

The latter is a choice even for people not that accustomed to gates and flip-flops. What we are talking about here is programming by describing the behavior. Hardware description languages are similar to familiar programming languages. Some development environments support even synthesizing circuitry out of source programs written in one of the popular high-level languages.

There is, however, an essential difference. To run a usual program, it must be compiled and loaded into the memory. A circuitry, however, has to be synthesized. This process relies on highly complex Boolean algorithms. Therefore, we speak of the two flavors of programmability.

What we want to emphasize here is the problem of making changes.

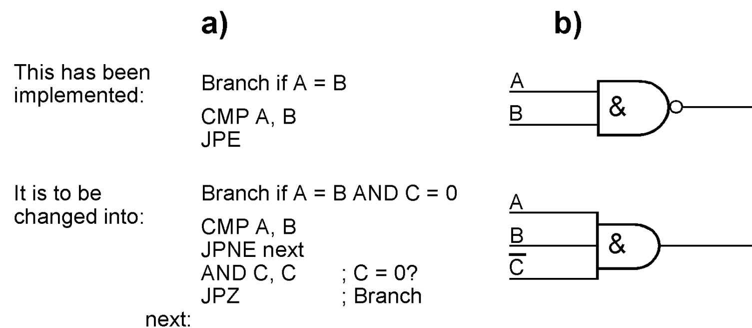

Figure A1 Two examples of changing minor details. a) software; b) hardware.

Changing a program is straightforward. Enter the statements or instructions, let the machine program be built again, load it and try whether the change shows to be effective or not. Sometimes, it is even feasible to alter the machine instructions residing in the memory. (There was a reason for the operating panels of the vintage computers. Veterans accustomed to the PDP-8 and the like will wistfully remember ...)

However, implementing an ECO (Engineering Change Order) in the hardware is much different.

On a vintage PCB, you had to identify available gates or IC sockets. Then traces had to be cut, wires soldered on, and the like. A small ECO would require, say, half an hour if you were allowed to tinker. If it was required to be approved by your superiors and done in a centralized repair facility, the turnaround time was unpredictable.

Within an FPGA, however, you cannot simply change a gate or flip-flop. Each ECO, even the slightest, requires running the Boolean synthesis once again. When the hardware is implemented with FPGAs, it can be modified over and over.It is tempting to write the solution of the application problem as a program and to leave it to the FPGA development system to synthesize the hardware. Here, however, the intricacies of the circuit synthesis can become noticeable, especially concerning the depth of the combinational circuitry and the associated clock slowdown, not to speak of the turnaround time.

As a remedy, we try to rely less on the Boolean synthesis and to increase the share of usual programming, in other words, solving the problems by natural intelligence and cunning.

We see microprogramming as a fundamental principle to achieve this objective. This way, we may build simple hardware platforms and solve the application tasks mainly by usual programming. (For a principal alternative, I refer to my CC article [7] and my web page *realcomputerarchitecture.com*.)

It is comparatively easy to design and debug such circuitry. The complexity of the solution of the application problemis not in the hardware but a memory content. Microprogramming brings usual programming down to the register transfer level. Thus we may be able to eliminate most design flaws and bring in most updates by programming instead of Boolean synthesis.

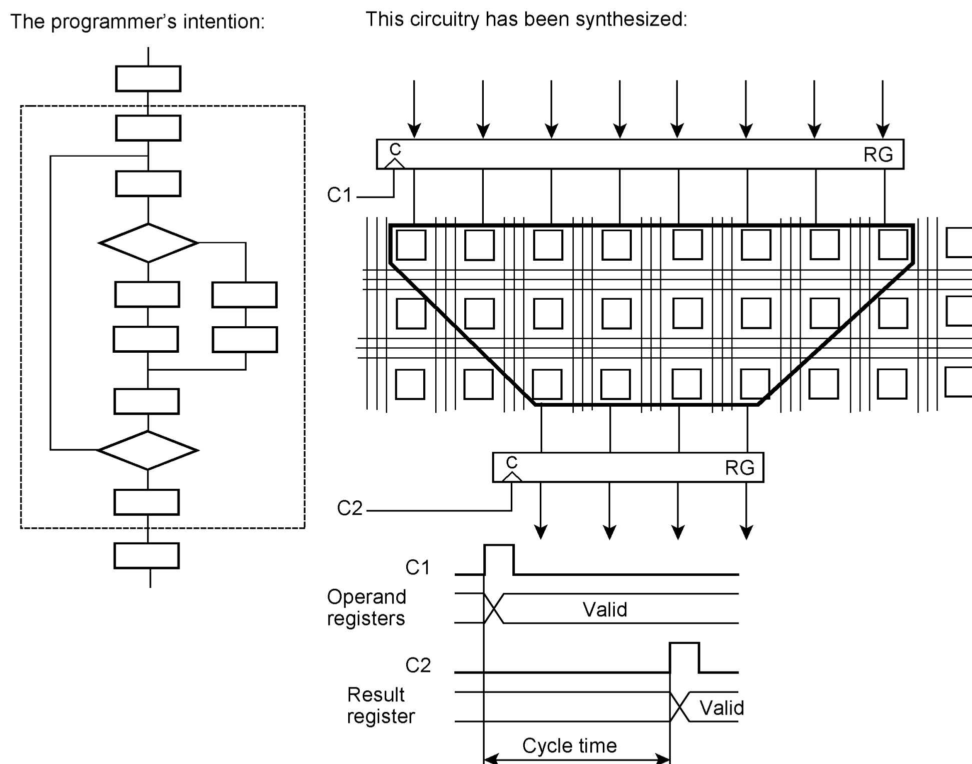

Figure A2 An application problem has been solved by hardware-software co-design. The engineer has written a program; the development system has synthesized a circuit to program an FPGA.

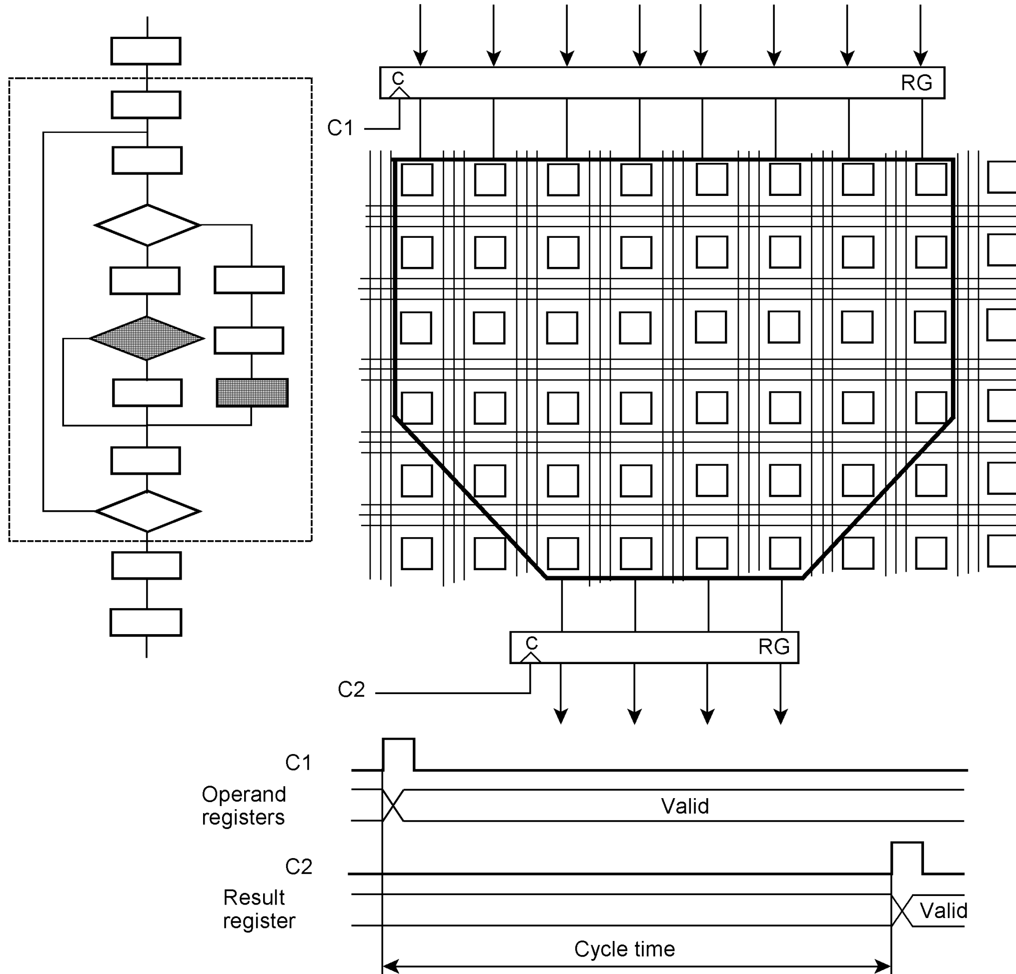

Figure A3 Now, the program has been changed. Therefore, the circuit must be synthesized again. However, this can yield deeper combinational networks. Consequently, the cycle time has to be increased accordingly.

Opportunities where microprogramming could step in

We expect opportunities on both ends of the performance and complexity spectrum.

The low end

It is about emulating state machines, even somewhat more complex ones. This is, so to speak, a natural domain of the small microcontrollers. Think, for example, about PICs, AVRs, 8051s, and the like.

Problem-solving may be easy. Only select a well-suited device and write a program. Sometimes, however, this will not fly.

In bygone times, the microcontroller I/O ports were plain registers and open-drain or tri-state driver stages. Nowadays, microcontroller manufacturers devote a considerable share of silicon real estate to complex programmable peripheral circuitry where each pin has its own register file.

Nevertheless, occasionally there may be no way to fit the microcontroller's peripherals to the intricacies of the application environment.

Therefore, you will have to choose a considerably more expensive platform (for example, a 32-bit RISC) or develop some application-specific circuitry.

Sometimes, the perfect IC would be a microcontroller core surrounded by arrays of FPGA-like logic cells, allowing you to design the peripherals yourself. If you need a 19-bit counter, then do not piffle around with 16-bit counter/timer units, interrupts, and so on, but don't hesitate to simply design one.

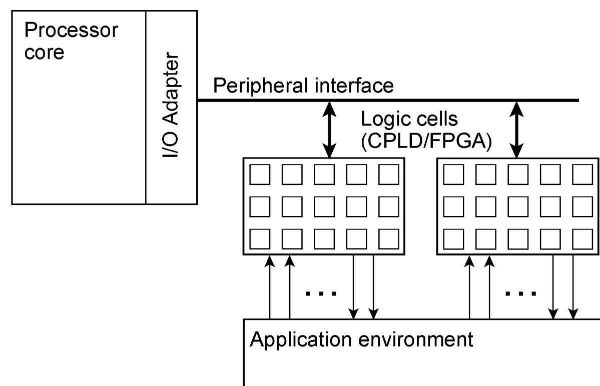

Figure A4 How an ideal microcontroller could look. A programmable array of logic cells is the most versatile I/O interface. Application-specific I/O devices are synthesized as required.

Making good use of the principles of microprogramming

This approach could lead to programmable peripherals or I/O processors or even replace the industry-standard microcontroller with a programmable core adapted to the requirements of the application.

Imagine a simple microcontroller core, as shown below. It is a single-address Harvard machine centered around an accumulator (or working register, respectively), somewhat similar to a renowned microcontroller family ([8] to [11]).

We may, however, confidently state that the principal idea of an accumulator-based single-address machine is free. It goes back to the pioneering work of John von Neumann and others.

The microinstruction is the generously dimensioned single-address instruction enhanced by additional functions. What we want to stress is that we can all details and dimensions tailor to our needs. Our design is, for example, not confined to 8 bits word length but may be synthesized for 19 bits if adequate to the application task. Furthermore, the microinstructions could be as long as appropriate, combining, for example, arithmetic functions, I/O accesses, and branching.

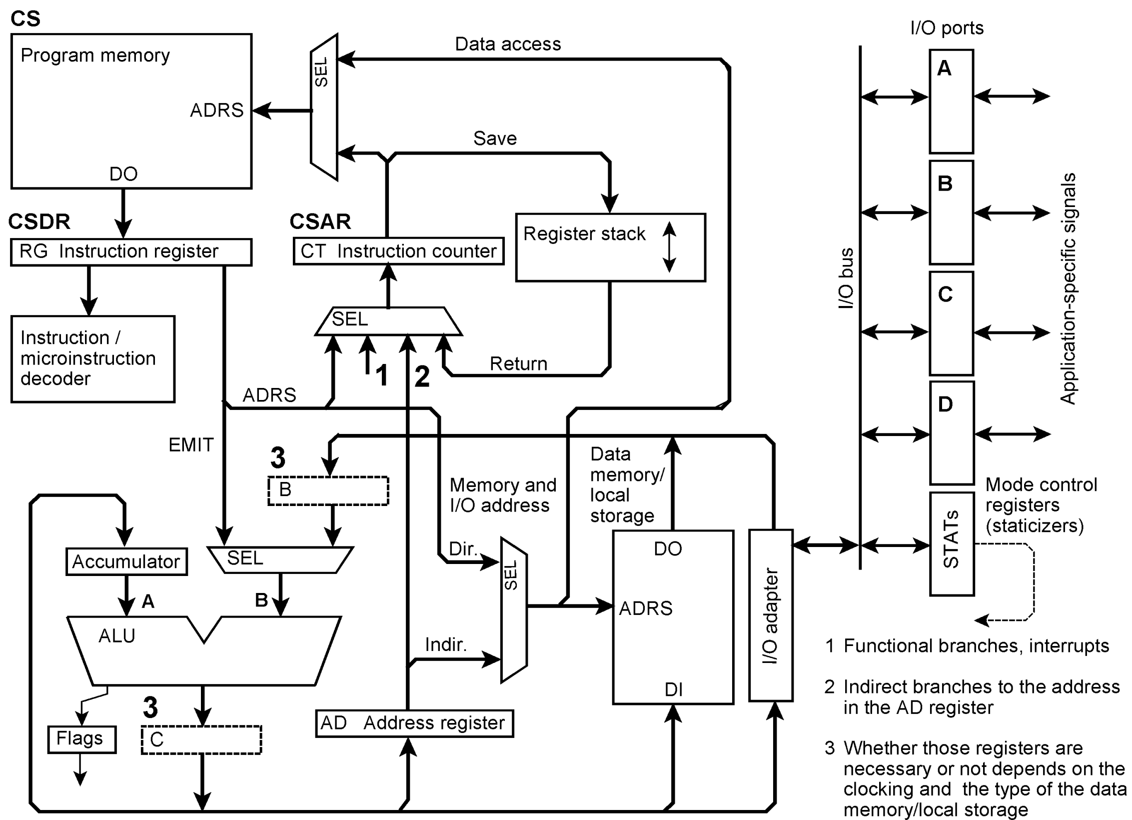

Figure A5 A small microprogrammable platform. It is a matter of opinion to call it a microcontroller and to liken it to a well-known architecture. Here, however, we emphasize the freedom to tailor the hardware to our needs. Typical examples are the word length, the format of the microinstructions, and application-specific peripherals.

The high end

Large FPGAs are populated by complex application-specific functional units, accelerators, and the like. Conventionally, RISC IP cores provide for initialization, parameter passing, communication, diagnostics, and other housekeeping work.

A microprogram control unit, appropriately and generously dimensioned, would show less overhead in dealing with the functional units, shorter latencies, and so on. It could be a companion to the RISC core or even replace it.

Within a functional unit, the complexity is not in the data paths but in the control section. The more straightforward the principles of operation are, the fewer opportunities may occur to commit design errors. To make the control sections of the functional units as straightforward as possible, the more complex functions may be assigned to the microprogram. It has extremely low latencies and can be tailored to particular requirements. So we can expect the decline of performance to be low, often even negligible. This is illustrated here by an example from the past.

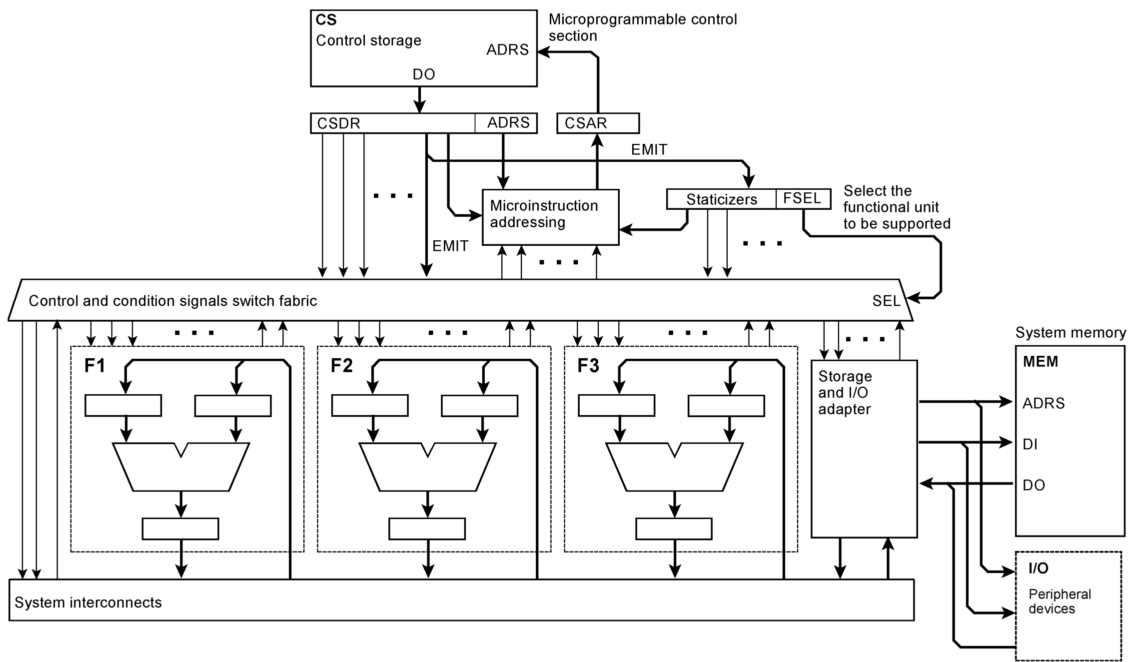

Figure A6 A microprogram control unit acting as some kind of conductor.

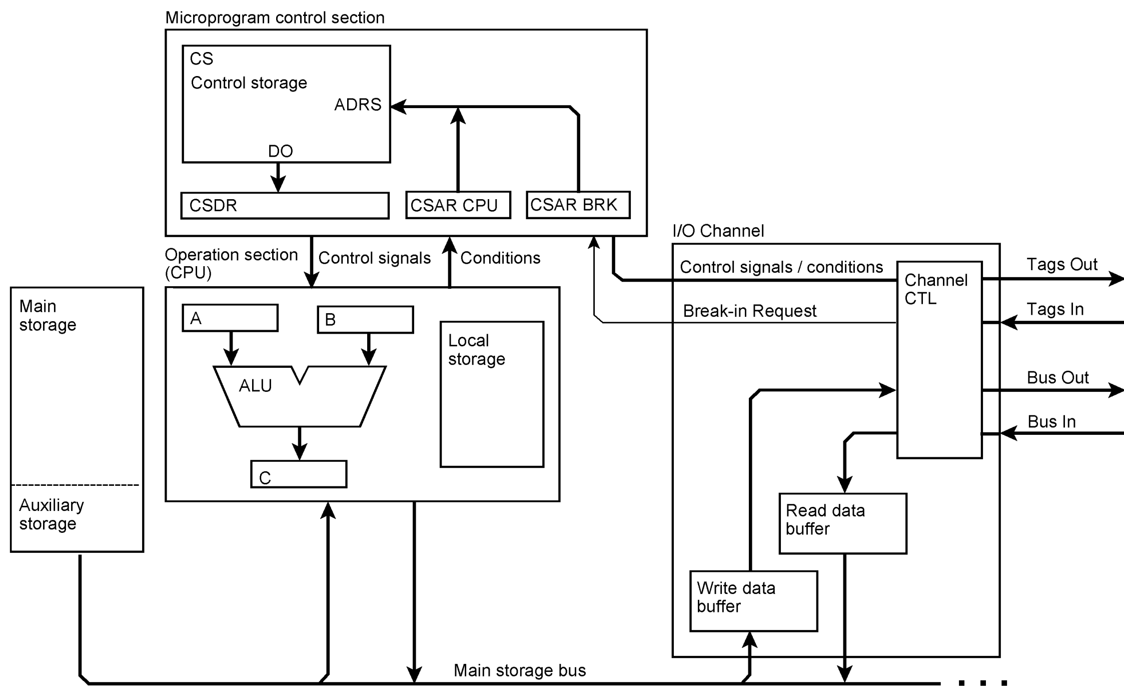

Figure A7 An example from the past. The I/O channels of the smaller models of IBM's system /360 and /370 are no completely independent functional units. Some channel functions have been assigned to the operation section (CPU, ALU) and the microprogram control section (some details may be found in [23] to [32]).

Our block diagram shows the basic functional units or sections of such a machine. The architecture provides for autonomous operation of the I/O channels. Once activated, for example,

by a Start I(O (SIO) instruction, they transfer data to and from the peripheral devices without programmed intervention. Both read and write data are buffered.

When a read buffer is full or a write buffer is empty, memory accesses are required. The channels control the I/O interface autonomously. Memory accesses, however, are executed via the ALU and are controlled by microprograms.

To empty or fill a buffer is not that simple. The memory address must be supplied and, after the access, incremented, the bytes transferred must be counted, and so on. Addressing the memory, incrementing the address, counting the bytes transferred, and so on is done via the ALU and controlled by microprograms. Thus the channels need no main storage interface, address registers, byte counters, and so on.

To call a supporting microprogram, a particular interrupt mechanism has been implemented. Such microprogram interrupts are called break-ins. They have nothing to do with the interrupts specified in the architecture. In our block diagram, the microprogram control unit has two control storage address registers (CSARs), one for the CPU and one for handling the break-ins (BRK).

The main storage is extended by an auxiliary storage area. There the CPU registers are saved, and the channels' memory addresses, byte counts, and so on are stored.

If a buffer is to be emptied or filled, the channel issues a break-in request, causing the second CSAR to address the control storage. The break-in microprogram saves the CPU registers, loads the channel address, the byte count, and the channel status, writes or reads the buffer content, does the housekeeping, and swaps the register contents again. Then it resumes CPU operation by switching back to the first CSAR.

The principal idea could be applied to FPGA-based machines, too. The control sections within the functional units are limited to comparatively straightforward sequential control tasks. The more complex control activities are assigned to microprograms called via break-in requests. The break-in mechanism may be a viable design idea to keep functional units not overly complicated. More details will be described in the 2nd article on this subject.

Functional branching

The microinstruction delivers only a part of the address of its successors. The remaining bits are contributed by condition signals.

In the follwoing example, a single condition signal is inserted into the lowest-order address bit position. Those block diagrams supplement Figure 13 in the printed article.

Besides the conditions, one needs the immediate values 0 and 1 to access the next microinstruction unconditionally. These values can be included in the set of selectable conditions (a), or their insertion can be controlled by a particular field in the microinstruction, as shown in (b) or in Figure 13. A separate encoding has the advantage that the COND SEL field may be freely available if the next microinstruction is to be addressed unconditionally.

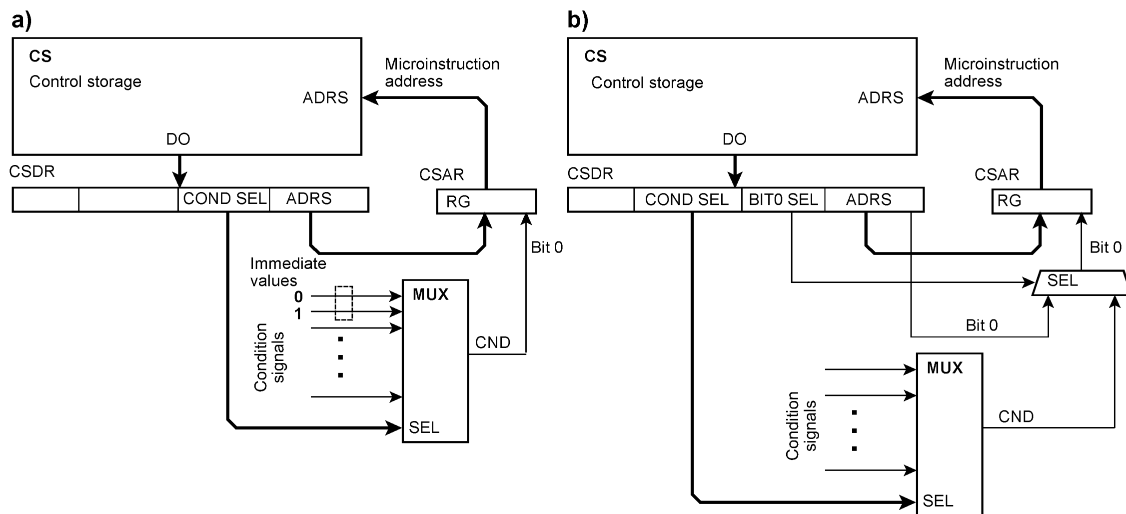

Figure A8 Inserting immediate values 0 and 1 in the microinstruction address. Two alternatives.

Multiway branching

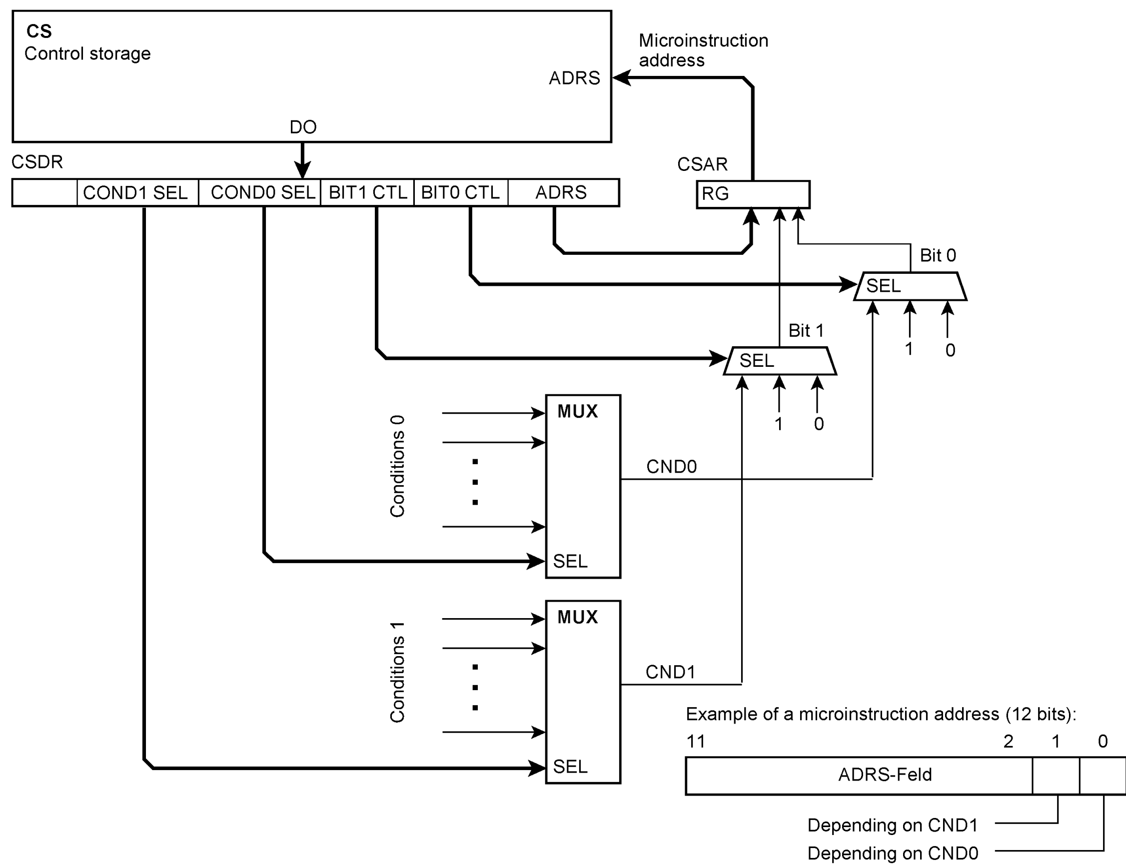

Figure A9 Inserting two condition signals into the microinstruction address allows branching in four directions.

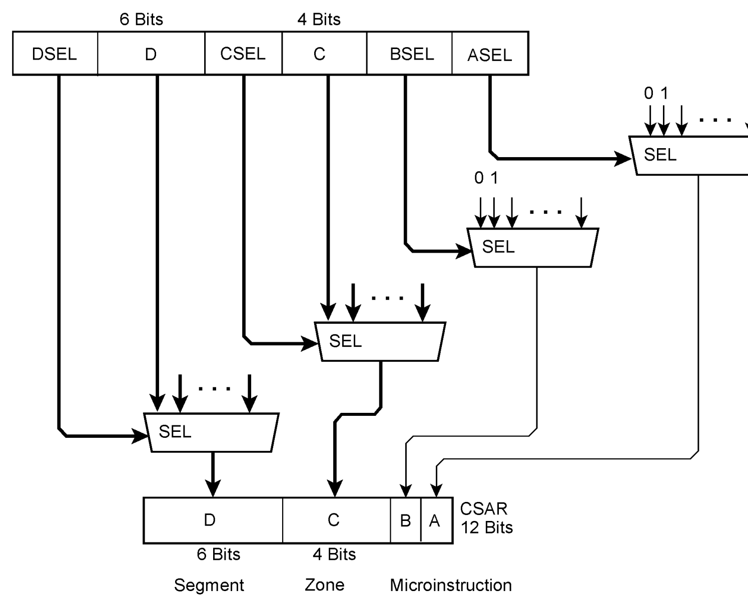

Figure A10 An example from the past. Details, for example, in [31] and [32].

This diagram depicts 4-way and functional (multiway) branching as implemented in IBM's S/360 model 50. Four-way branching works as described above. The higher-order address bits may be immediates out of the microinstruction or taken from registers, staticizers, or other condition signals from various parts of the machine.

Inserting bits into an address means that an entire subspace of the address space may be occupied by microinstructions being potential successors. Hence microinstructions cannot be placed simply one behind the other.

When n bits are inserted into the lowest-order (rightmost) address bit positions, one of 2^n potential successors may be addressed. To avoid squandering address space, addresses are to be generated selectively.

The principal solution is to split up the total address space into segments, partitions, or the like. The address format depends on the number of potential successors (for example, whether the successor is to be selected within a block of 4, 16, or 64 microinstructions or within the whole address space).

In our example, the microinstruction address space is divided into 64 segments of 16 zones of four microinstructions, allowing for selectively placing microinstruction blocks of different sizes.

The segment, zone, and microinstruction addresses may be immediate values or put together from various signals and register contents.

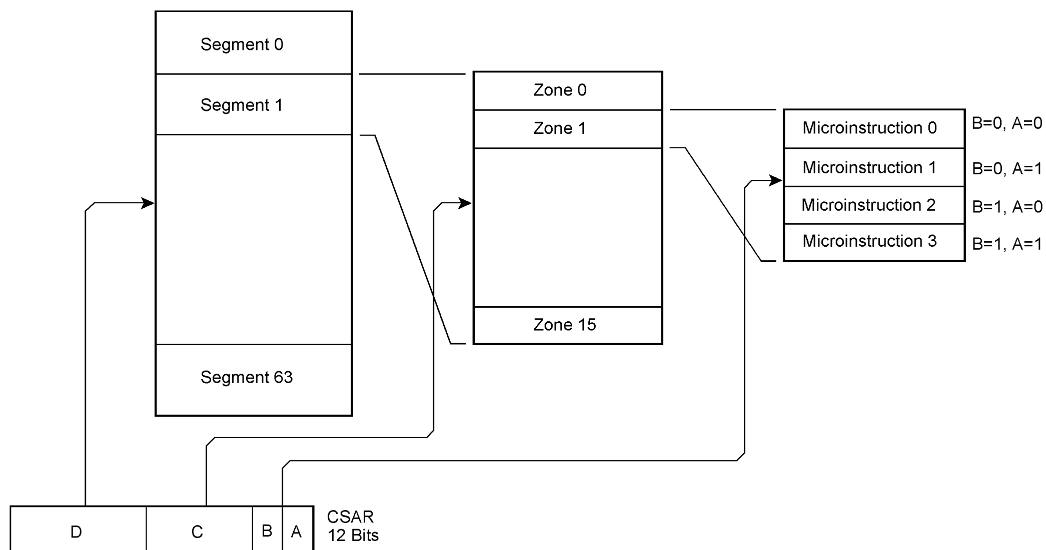

Figure A11 How the address space is split up in segments, zones, and microinstructions.

Clocking

Clocking is a fundamental task in circuit design. The microprogram-controlled machines of the past had several clock phases and clock pulses committed to particular functions. They were generated even with monostable multivibrators or delay lines. This cannot be applied to the clock systems of the FPGAs. When logic was built on printed circuit boards, clock pulses could be generated in whatever way was deemed appropriate. You also had the choice between latches and flip-flops. In the FPGA, on the other hand, we have to use what is prefabricated, the logic cells with edge-controlled flip-flops, the clock signal paths, and the clock generation and management. But we can work with extremely high clock frequencies.

Our circuits are operated by clocks running continuously. The registers, counters, and flip-flops are controlled with enable signals (Clock Enable CE, Load Enable LD, and so on). A CLR signal is not an erase pulse but a signal allowing the register to be cleared; an LD signal is not a load pulse but a signal allowing the register to be loaded. What these signals enable or allow, respectively, becomes effective with the next clock edge. Most of our block diagrams do not show the clock signals and the clock inputs of the components.

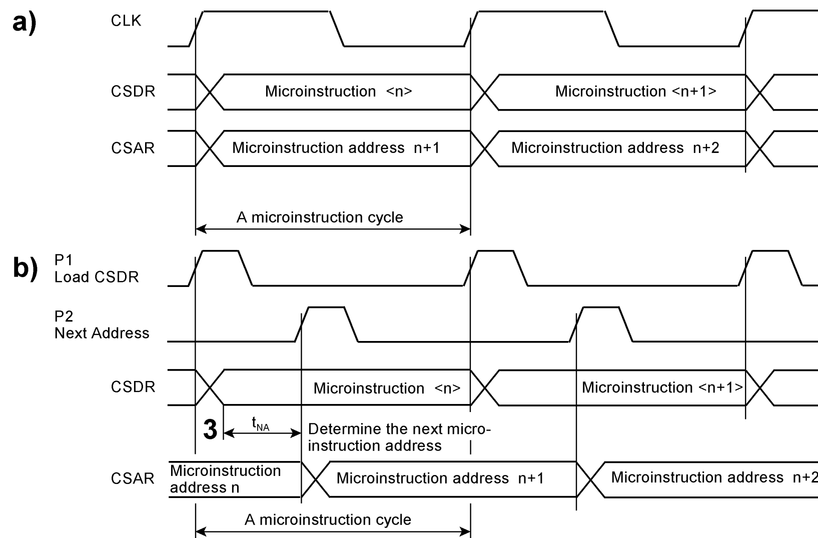

Figure A12 Single-phase and multi-phase microinstruction cycles.

a) The single-phase microinstruction cycle. A single clock signal is applied to all flip-flops. The clock cycle is the microinstruction cycle. When flip-flops switch with a particular clock edge, all enable signals must be valid before this edge occurs. The successor to the current microinstruction must be selected before loading the microinstruction register CSDR (1). The corresponding control signals cannot be attached to the microinstruction register because then they would be effective only in the next microinstruction cycle (2). Hence they must be attached to the outputs of the control storage.

b) The multi-phase microinstruction cycle. It makes sense to divide the microinstruction cycle into at least two phases. Then the control signals may be connected to the microinstruction register CSDR. At the beginning of the first phase (P1), the microinstruction register is loaded. Then the microinstruction fields are decoded. The control signals pass through the signal paths and combinational circuits. This way, the address of the next microinstruction has been obtained too. In the second phase (P2), it is loaded into the microinstruction address register CSAR.

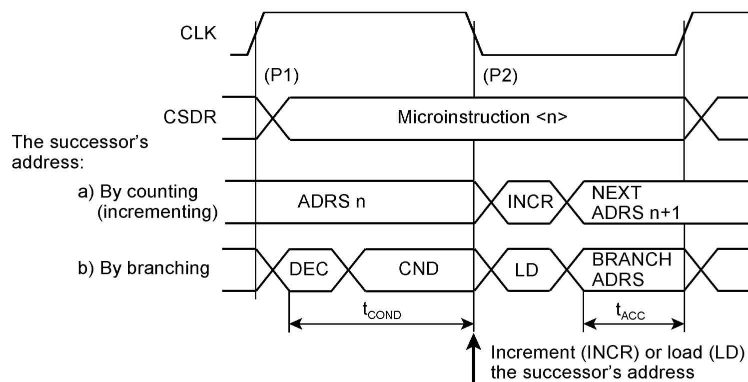

Figure A13 The simplest multiphase clock is the two-phase clock, which results when both edges of a single clock signal are used.

Condition signals

It must be possible to branch on conditions. The microprogram control unit is a clock-synchronous state machine. Therefore, condition signals must be synchronized when not generated inside. It is essential when the condition signals deciding about the next microinstruction must be valid. There are three basic alternatives.

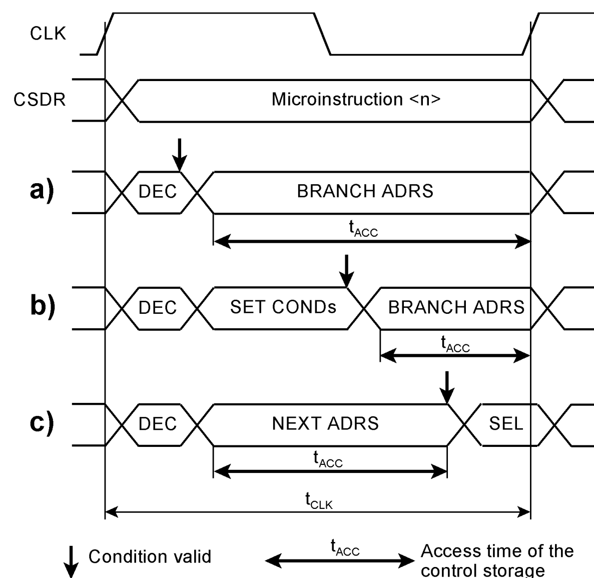

Figure A14 When must the condition signals be valid? Three basic alternatives.

a) At the beginning of the microinstruction. The microinstruction evaluates conditions that have been selected before. Fetching the next microinstruction can then begin immediately. Many microprogram control units work this way. Processing or I/O microinstructions select the conditions; branch microinstructions decide about the next microinstruction.

b) In the same microinstruction. The fields of the current microinstruction determine how the conditions are obtained. Therefore, the successor's address will be available only later in the cycle. If the cycle time is given, the control storage must have a correspondingly short access time. Otherwise, the microinstruction cycle must be longer. Since the microinstruction selects or generates the branch conditions itself, we will often get by, however, with a single microinstruction where otherwise we would have needed two or more microinstructions.

c) At the end of the microinstruction cycle (late branching). When the current cycle begins, the microinstruction causes all successors to be read in parallel. In the meantime, the conditions are queried. Both activities take place at the same time. Calculations, comparisons, querying of conditions, and the like overlap the fetching of the next microinstructions. At the end of the microinstruction cycle, the conditions have become valid too. Accordingly, at the beginning of the new microinstruction cycle, the successor is selected from the microinstructions that have been read in advance.

Synchronization
Synchronization means to sample signals coming from outside, making them fit the timing requirements of a clocked (in other words, synchronous) circuitry ([12] to [16]).

Synchronization circuits (synchronizers)
Concerning the I/O ports of the microcontrollers, synchronizers are usually considered self-evident. In many datasheets and manuals, they are not even mentioned and not shown in the block diagrams. When designing circuitry, however, you will have to solve the problem by yourself. (At that, you should heed a particular pitfall: I/O ports have built-in synchronizers, the bus systems of the microprocessors typically do not.)

The most straightforward synchronizer is a D-type flip-flop connected to the asynchronous input signal and a clock.

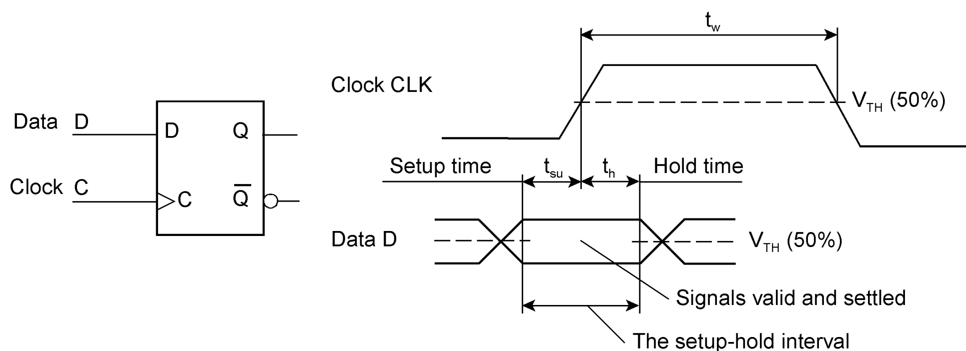

Figure A15 The most straightforward synchronizer.

Clock and Data – the Setup-Hold Interval
There are two data sheet values relating to an interval in which the data must have settled, the setup time and the hold time.

The setup time is the minimum interval in which the signals must be valid and settled before the clock edge triggers the flip-flop.

The hold time is the minimum interval the signal must be kept valid and settled after the clock edge. To many flip-flops, a hold time of zero is specified. Then the input signal may change once the clock edge has passed the threshold voltage.

Metastable states
The synchronization is a sampling process, the signal being sampled, for example, by the low-to-high edge of the synchronization clock. But what happens when the input signal of a flip-flop changes in the setup-hold interval surrounding a clock edge? Sometimes nothing special will happen; the flip-flop will either change its state or keep the previous one. However, there is a critical time interval within the setup-hold interval. Its width depends on the circuit technology and the structure of the flip-flop (we speak of picoseconds here; approx. 1 to 150 ps are typical).If the input signal changes within this interval, the flip-flop may enter an intermediate state, called the metastable state. In such a state, the flip-flop emits output signals that do not correspond to one of both logic levels.

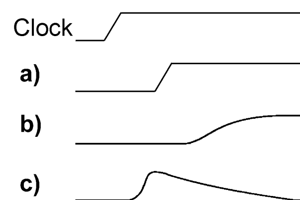

Figure A16 Output signals of a synchronization flip-flop. a) correct output signal; b), c) output signals typical of metastable states. After a while, the metastable state fades away; the signal then enters one of the two logic levels.

Such metastable states are unavoidable. One can only wait a particular time (settling time), hoping the metastable state has then vanished.

Occasionally, it may happen that it does not vanish. This is deemed a failure. How often will it occur? The relevant characteristic parameter is the mean time between failures (MTBF). If it exceeds the typical lifetime of the hardware considerably or meets the customer's requirements, then we may be content.

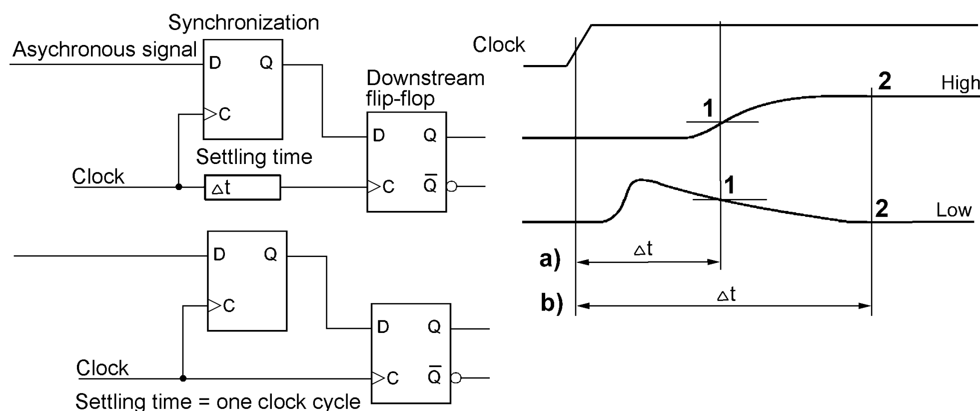

Figure A17 Metastable states will settle if we wait long enough. a) The settling time Δt is obviously too short. b) The settling time is long enough; the downstream flip-flop sees a valid logic level at its input. 1 - invalid, 2 - valid logic levels. The settling time is usually not implemented by a delay line but by a clock cycle.

It is essential to synchronize all signals from outside with separate flip-flops and to provide enough settling time between synchronization and the clocks of the downstream flip-flops. Often, a clock cycle will suffice. FPGA manufacturers mention millions of years MTBF if allowing for a settling time of 5 ns, for example.

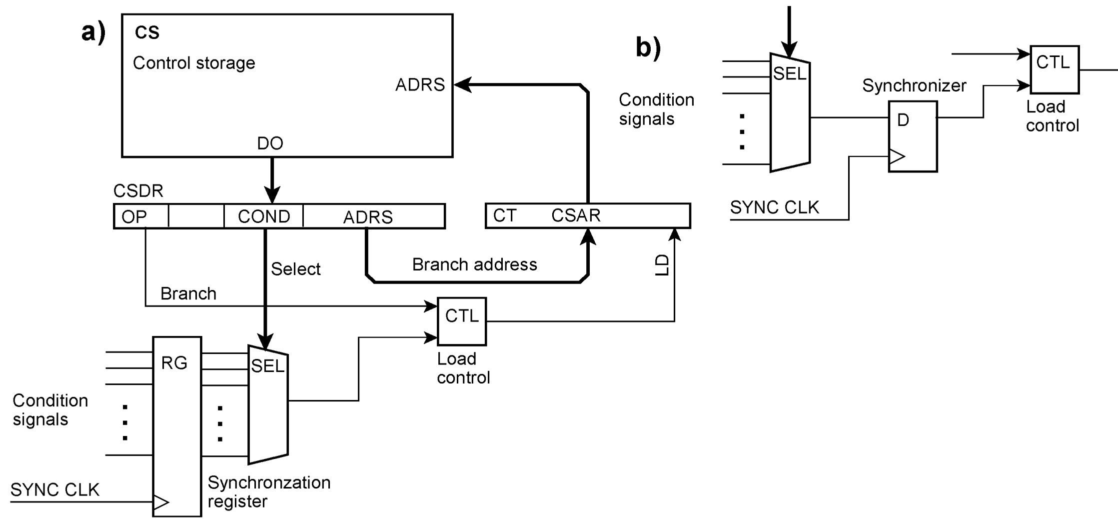

Figure A18 A typical synchronization problem in a microprogram control unit.

a) All condition signals are synchronized. Synchronization does not depend on condition selection. SYNC CLK could be the clock pulse at the beginning of the cycle.

b) The selected condition signal is synchronized. Three clock phases are needed to make this circuit work. The first loads the microinstruction into the microinstruction register. The second synchronizes the selected condition. The third causes the microinstruction address to be loaded or incremented (depending on whether the branch is to be taken or not).

Typical design flaws:

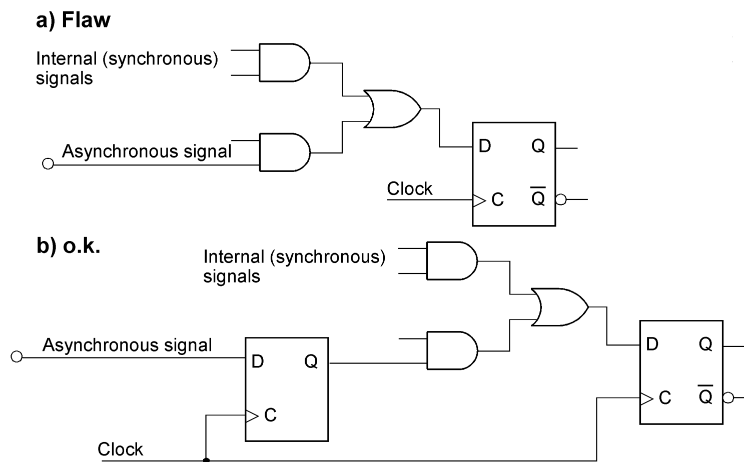

Figure A19 Synchronize all the asynchronous signals. Do not connect them unsynchronized to downstream flip-flops.

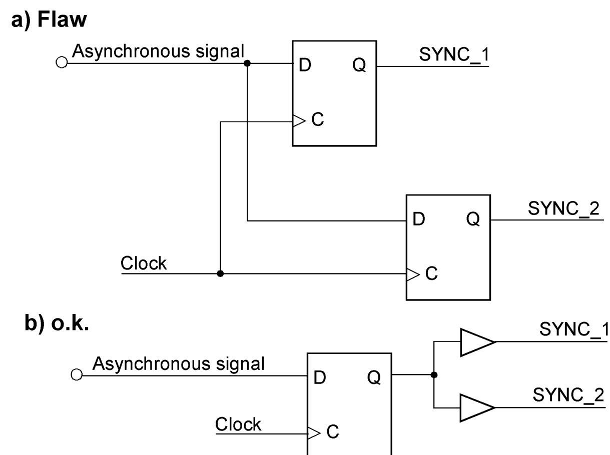

Figure A20 Synchronize each asynchronous signal with only one flip-flop. Otherwise, SYNC_1 might show a different signal level than SYNC_2. Add driver stages if you need a higher fan-out.

We presuppose that you are familiar with this basic tenet of digital technology and consider it when designing. For the sake of clarity, we have therefore omitted the synchronization circuits in most of the figures.

Writing microprograms

Pursuing a small project, we cannot expect a fully-fledged compiler. Instead, we must be content with an assembler. Principally, this is a program that converts symbolic names into bit patterns. There are several ways to provide such a development tool, the meta-assembler, the macro-assembler, and the homemade assembler.

Meta-assemblers are designed to generate assemblers for any machine code. You have only to set up the corresponding tables.

Any somewhat advanced macro-assembler can generate any bit pattern from any number of parameters. It is thus possible to define the microinstructions as macros. The parameters of the macros can be numerical values, symbolic addresses (labels), or symbolic identifiers (the latter are to be defined using EQ statements).

A homemade assembler is not that difficult to write. At its core, it's just a program searching in tables. If the tables can get large, searching should be programmed adequately, for example, by hashing algorithms. However, if the tables are not too large (at most a few thousand entries), they can also be scanned item by item (linear search).

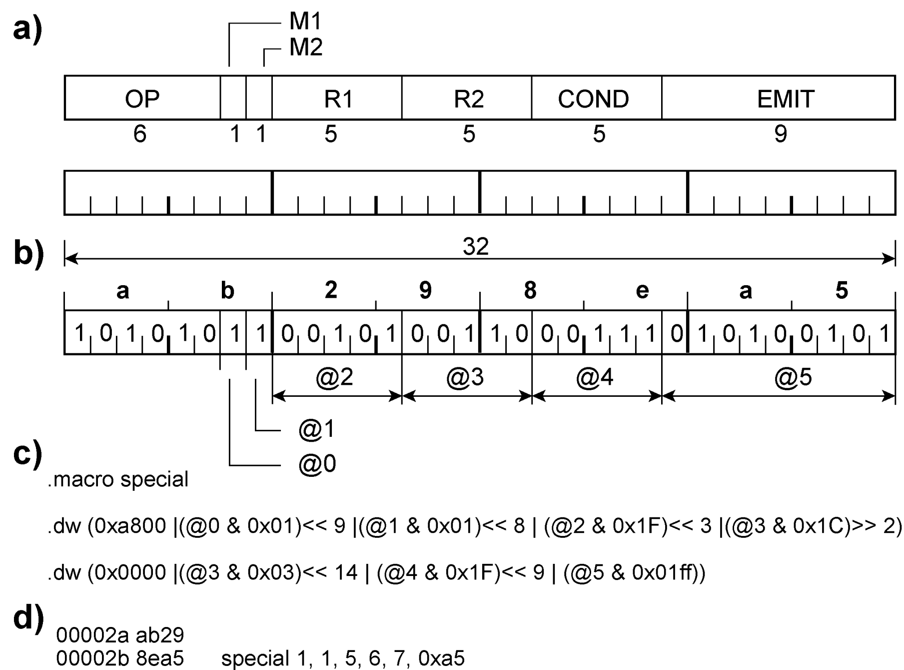

Figure A21 Declaring a microinstruction as a macro. The 6-bit opcode is 2AH. Being the 6 leftmost bits in a byte, it is to be captured as A8H.

In our example, we use a comparatively basic macro-assembler. It is part of the AVR Studio (Atmel / Microchip). a) shows a fictitious microinstruction format, b) the bit pattern of the call example given under d). The microinstruction is 32 bits long. It is captured as a macro called *special* (c). The contents of the microinstruction fields are the macro parameters @ 0 to @ 5. The 32 bits are split up into two 16-bit words. The bit positions to be inserted are cut out of the transferred parameters (by AND-ing; &), shifted appropriately (<< or >>), and inserted into the respective word (by OR-ing; |). d) shows an example of a call. Instead of the numerical values, symbolic addresses (labels) or identifiers (mnemonics) can be entered too.

References

Our list begins with the pioneering paper of Maurice V. Wilkes. Then we mention two vintage textbooks. The literature of the past deals mainly with details of the technology at the time. College textbooks address the topic merely superficially. The most authoritative and inspiring sources are the handbooks and manuals of machines really built. [18] to [33] are a small selection, here restricted to vintage mainframe machines, above all IBM's S/360. Devouring such sources, you may skip the machine-specific details and concentrate on the principles. In [6], I have tried to cover the subject comprehensively, of course with state-of-the-art implementations and applications in mind.

[1] Wilkes, Maurice V.: The Best way to Design an Automatic Calculation Machine. Report of Manchester University Computer Inaugural Conference, July, 1951, p. 16–18.

[2] Wilkes, Maurice V.; Stringer, J. B.: Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer. Proceedings Cambridge Philosophical Society, Vo. 49, No. 2, 1953, p. 230–238.

[3] Wilkes, Maurice V.: The Growth of Interest in Microprogramming: A Literature Survey. Computing Surveys, Vol. 1, No. 3, September 1969, p. 139–145.

[4] Husson, Samir S.: Microprogramming. Principles and Practices. Prentice-Hall, 1970.

[5] Agrawala, Ashok; Rauscher, Tomlinson G.: Foundations of Microprogramming: Architecture, Software, and Applications. Academic Press, 1975.

[6] Matthes, Wolfgang: Mikroprogrammierung. Prinzipien, Architekturen, Maschinen. ISBN 978-3-8325-5234-3. Logos, 2021.

[7] Matthes, Wolfgang: Resource Algebra and the Future of FPGA Technology. Circuit Cellar, Issue 317, December 2016, p. 18-27.

Small microcontrollers:

[8] PICmicro Mid-Range MCU Family. Microchip Technology Inc., 1997.

[9] PIC16(L)F1508/9 20-Pin Flash, 8-Bit Microcontrollers with XLP Technology. Microchip Technology Inc., 2011–2014.

[10] PIC17C7XX High Performance 8-bit CMOS EPROM Microcontrollers with 10-bit A/D. Microchip Technology Inc., 1998–2013.

[11] PIC18(L)F67K40 64-Pin, Low Power, High Performance Microcontrollers with XLP Technology. Microchip Technology Inc., 2016–2017.

Synchronization and metastability:

[12] Johnson, Howard; Graham, Martin: High-Speed Digital Design. A Handbook of Black Magic. Prentice-Hall, 1993.

[13] Dally, William J.; Poulton, John W.: Digital Systems Engineering. Cambridge University Press, 1998.

[14] Becke, Georg; Haseloff, Eilhard: Das TTL-Kochbuch. Digitaler Schaltungsentwurf in Theorie und Praxis. Texas Instruments, 1996.

[15] Metastable Response in 5-V Logic Circuits. SDYA006. Texas Instrumens, 1997.

[16] Alfke, Peter; Philkofsky, Brian: Metastable Recovery. XAPP094. Xilinx, 1997.

[17] Metastability in Altera Devices. AN-042-04. Altera, 1999.

Microprogrammed machines:

[18] An Introduction to Microprogramming. IBM Corporation, 1971.

[19] IBM System/360 Model 25 Functional Characteristics. IBM Corporation, 1972.

[20] IBM System/360 Model 25 Microprogram Listing System/360 Emulator. IBM Field Engineering Education Supplementary Course Material. IBM Corporation, 1970.

[21] 2025 Processing Unit. IBM Field Engineering Education Student Self-Study Course. IBM Corporation, 1969.

[22] 2025 Processing Unit. IBM Field Engineering Theory of Operation. IBM Corporation, 1968.

[23] 2030 Processing Unit IBM Field Engineering Manual of Instruction. IBM Corporation, 1965.

[24] System /360 Model 30 IBM Field Engineering Handbook. IBM Corporation, n. d.

[25] System /360 Model 30 2030 Processing Unit. IBM Field Engineering Theory of Operation. IBM Corporation, 1967.

[26] System /360 Model 40 Functional Units. IBM Field Engineering Manual of Instruction. IBM Corporation, 1970.

[27] System /360 Model 40 CPU and Channels. IBM Field Engineering Supplementary Course Material. IBM Corporation, 1970.

[28] System /360 Model 40 2040 Processing Unit. IBM Field Engineering Diagram Manual. IBM Corporation, 1970.

[29] System /360 Model 40 Comprehensive Introduction. IBM Field Engineering Theory of Operation. IBM Corporation, 1970.

[30] System /360 Model 40 IBM Field Engineering Handbook. IBM Corporation, n. d.

[31] System /360 Model 50 Multiplexor Channel Field Engineering Theory of Operation. IBM Corporation, 1966.

[32] System /360 Model 50 2050 Processing Unit. IBM Field Engineering Diagram Manual. IBM Corporation, 1966.

[33] Spectra 70 System 7045 Processor EO Flow Charts. RCA Corporation, 1966.

Sources

The author's homepages:

https://www.realcomputerprojects.dev
https://www.controllersandpcs.de/projects.htm
https://www.realcomputerarchitecture.com

The ultimate archive concerning computer architecture and vintage computers:

http://bitsavers.trailing-edge.com/pdf/

Here you may find the references [18] to [33] – and much more ...