Microprogramming Choices Explained (Part 2)
Microprogrammable Machines
– Addendum –

Most basic sequencers

They cannot branch but only advance from state to state. They go through their sequence of states either cyclically or only once (single-cycle operation). More advanced sequencers can remain in the current state until a particular conditon is met. Such states are called wait states.

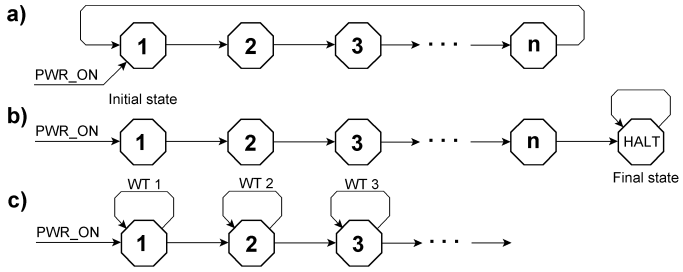


Figure A1 Some state diagrams of basic sequencers, showing cyclic operation (a), singe-cycle operation (b) and wait states (c).

Electromechanical sequencers

In the printed article, we have already mentioned the liquid-propellant rocket as a somewhat exotic but impressive and perspicuous example. In the past, such control tasks have been tackled by electromechanical devices, magnetic tapes, and the like ([3]; [4]). In [3], for example, you may find some vivid descriptions of how such electromechanical sequencers work. By the way, there are situations in which it is just right to go stubbornly through an unsophisticated sequence of states. For example, after developing full thrust, the rocket must move away from the launchpad as fast as possible, with deliberate disregard of trajectory corrections and optimization of the combustion processes. The designers of the N1 (the Soviet counterpart to the Saturn V) had to learn this the hard way (as described in [3]).

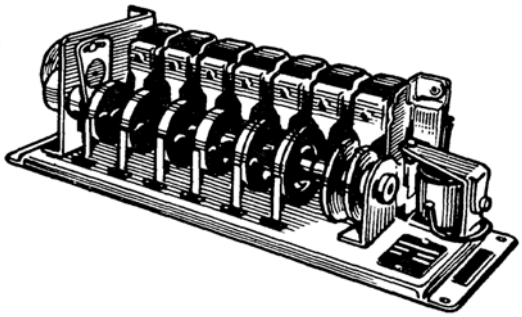


Figure A2 Example of an electromechanical sequencer (cam timer). A DC motor drives a shaft carrying interchangeable cams. Such devices have been offered for continuous (recycling) and single-cycle operation ([5]). Wait states could be introduced by disconnecting the motor's supply voltage until the wait condition is satisfied. In alternative implementations (as described, for example, in [3]), the camshaft is driven by a stepper motor. During wait states, stepping pulses occur only if the wait condition is satisfied.

The binary counter as control storage address register (CSAR)
Our counterpart to the camshaft is a binary counter (the CSAR) addressing a memory (the control storage CS) containing bit patterns energizing the control signals. At least one bit position serves to encode what to do in the last state. For cycling operation, it will reset the counter. For single-cycle operation, it will inhibit further counting.

In our circuit diagrams, we use fully synchronous counters (as you may invoke in a CPLD/FPGA development environment or get synthesized from a behavioral description). The clock runs continuously. All control signals are hi-active. The priorities are 1. reset, 2. load, 3. count.

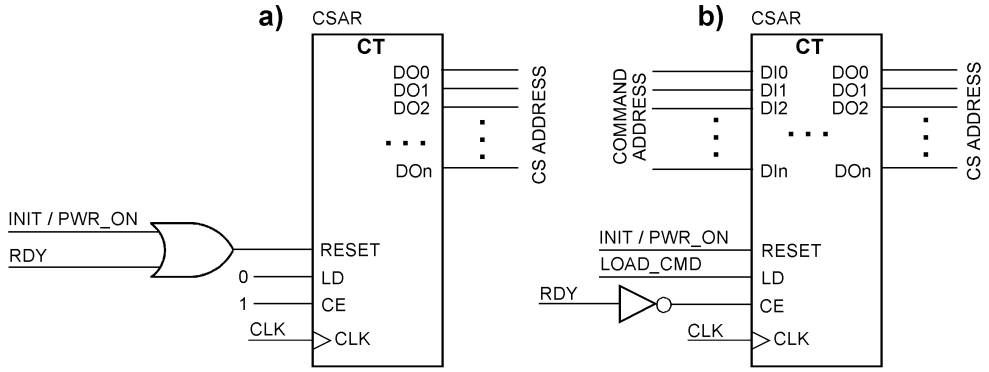


Figure A3 Binary counters act as the CSAR. a) shows continuous (recycling) operation, b) single-cycle operation, supplemented by provisions to load a start address (referred to as a command address) from outside.

A general-purpose programmable sequencer
In old times, to get a sequencer showing the desired behavior, you had to order an appropriate device (for example, as offered in [5]). In CPLD and FPGA times, you have to synthesize the circuit accordingly.

Some microinstruction bits more will lead to a general-purpose sequencer that is to be synthesized only once, allowing to change the desired behavior on the fly simply by altering the memory content.

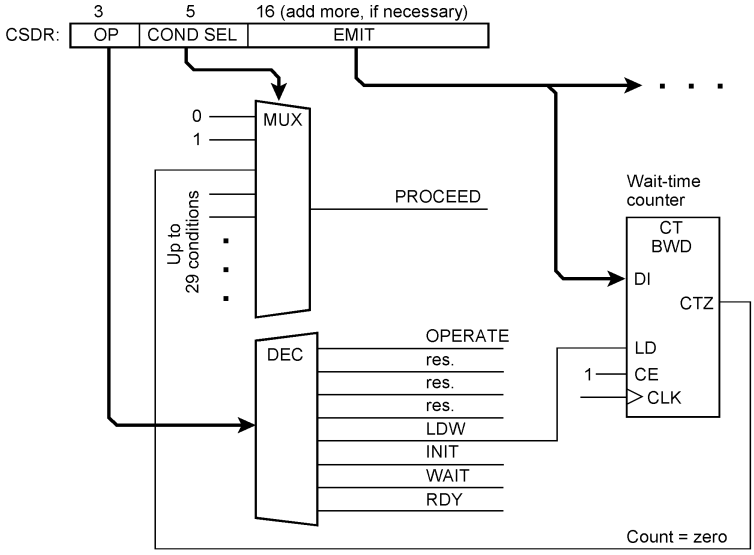


Figure A4 Example of a general-purpose microinstruction format. This sequencer can wait, cease its operation, or operate cyclically. All the variants and modifications described in the printed article are combined here. The EMIT field may accommodate application-specific fields and control bits.

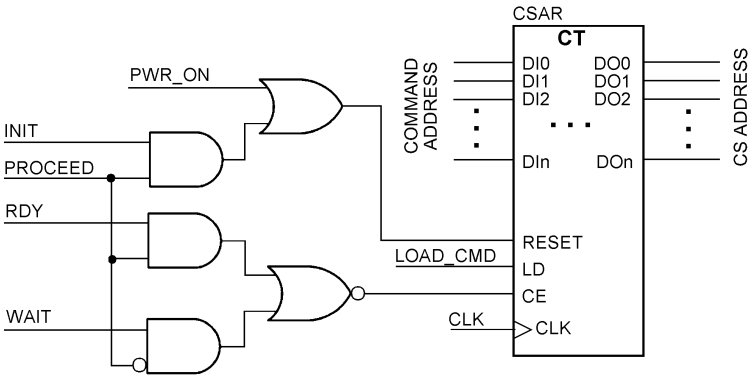


Figure A5 Here the CSAR is shown. It is also illustrated how the selected condition (the PROCEED signal) may control whether the micro-operation is executed or skipped.For example, the sequencer will cycle (INIT) or halt (RDY) only if the selected condition is met. Otherwise, the next microinstruction will be read. For unconditional execution, the condition "1" is to be selected. It is some kind of predication as known from the ARM architecture.

A general-purpose branch sequencer

The state transitions: advance to the next state, retain the current state (wait), branch to another state, and go to the initial state may be implemented by a binary counter and some multiplexers.

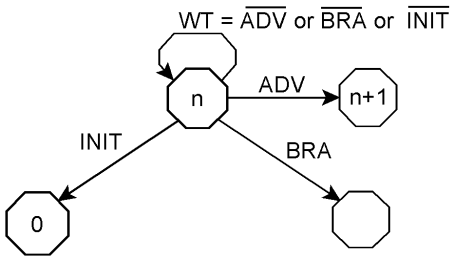


Figure A6 A versatile pattern of state transitions.

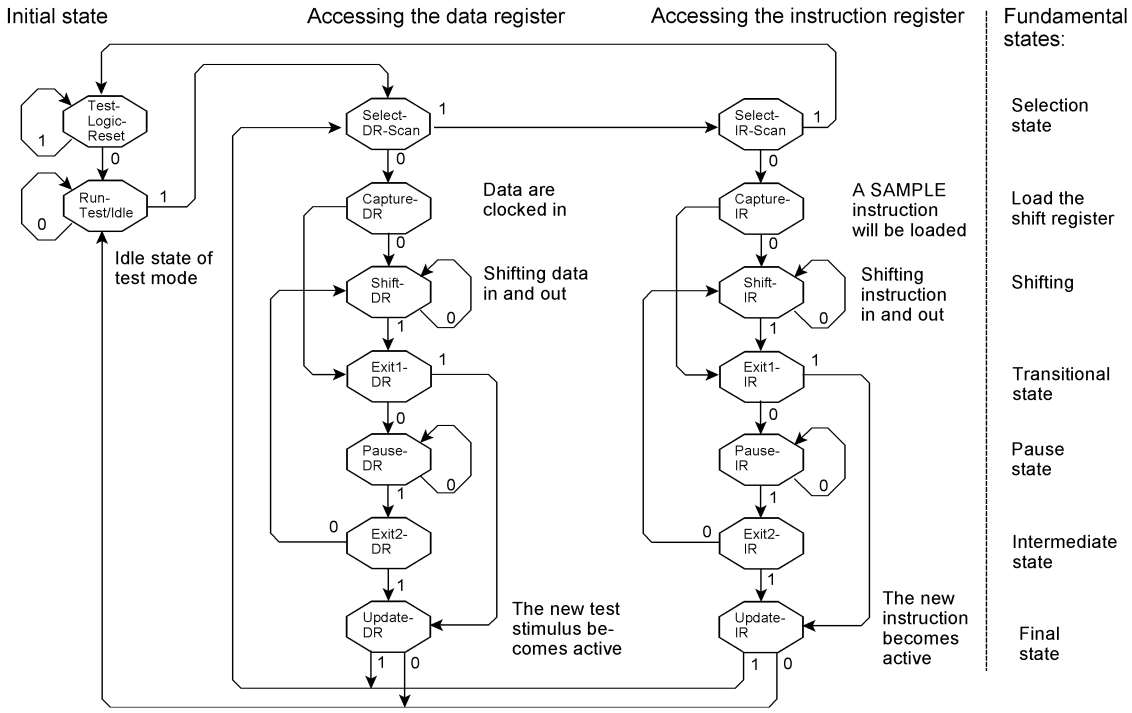


Figure A7 This example demonstrates that the principal pattern of state transitions may be viable. Shown here is the state diagram of a Test Access Point (TAP) according to the IEEE standard 1149.1 (refer to, for example, [6] and [7]).

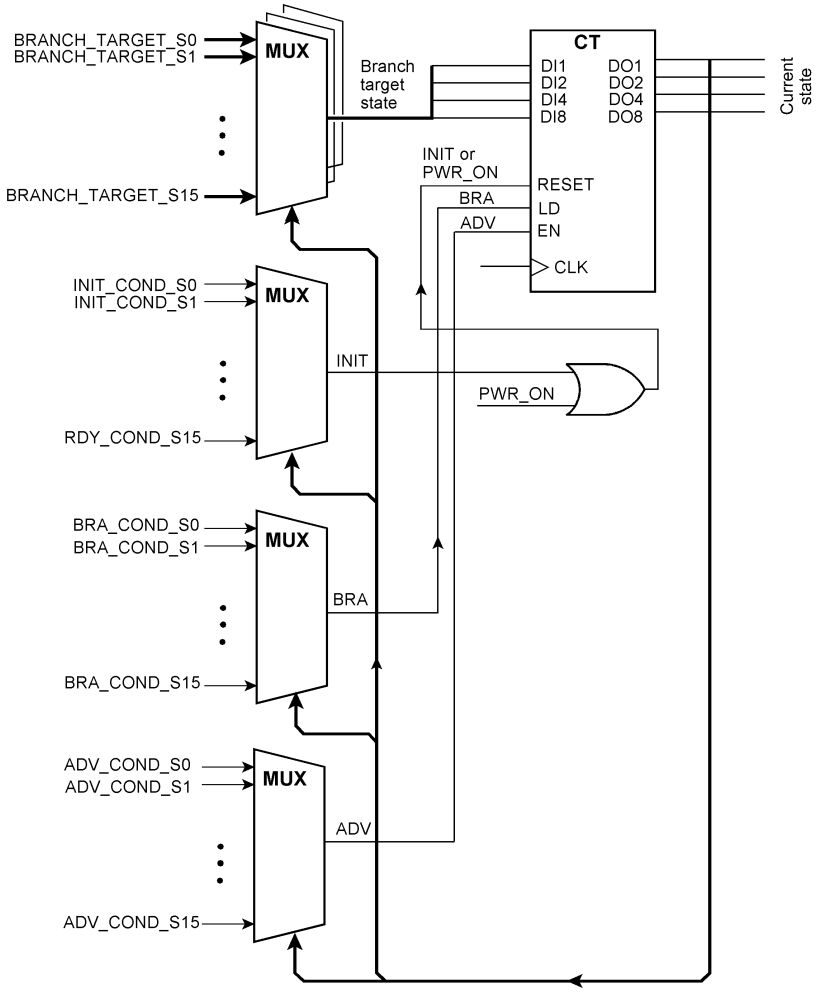


Figure A8 The branch sequencer centered around a binary counter.

For each of the states S0...S15 and for each kind of transition, the appropriate condition signals are to be connected to the particular multiplexer. If a transition should occur unconditionally, connect a logical 1. If the transition should never take place, connect a logical 0. The topmost multiplexers in the diagram constitute a ROM, addressed by the current state and delivering the state number of the branch target.

Outputs are not shown here. They could be added by decoding the state number or by combining the state number with input signals.

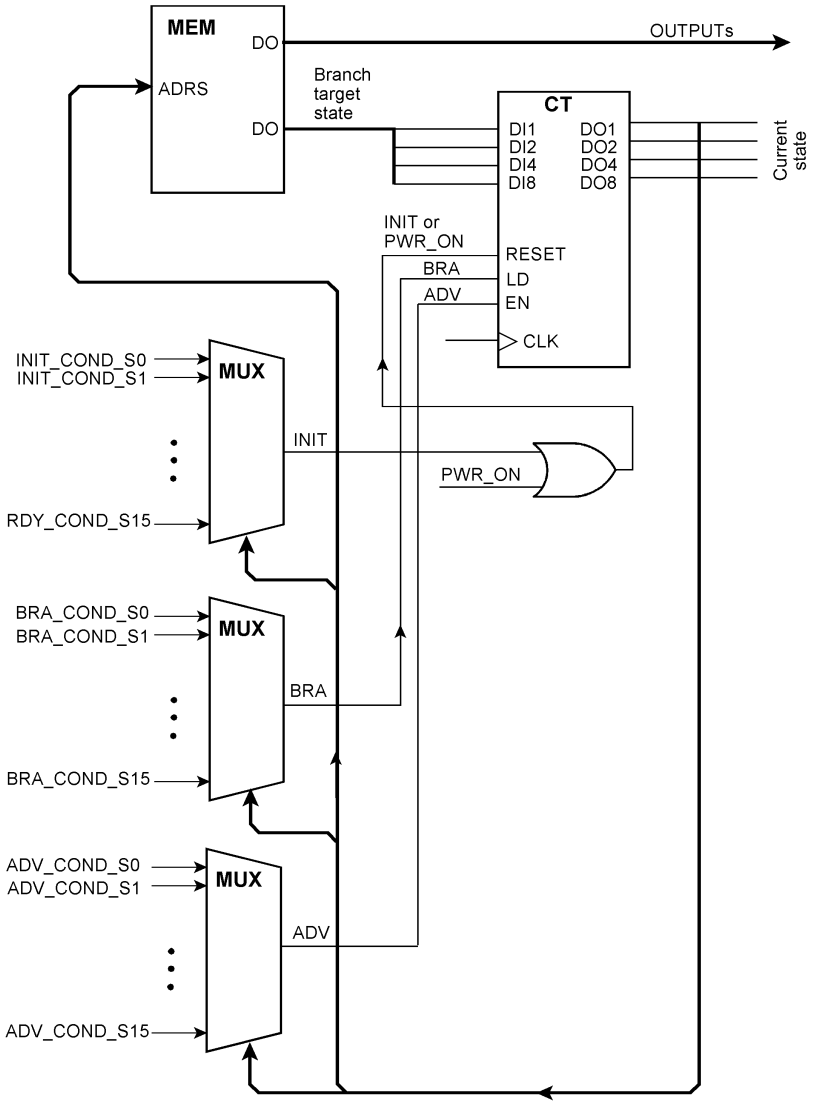


Figure A9 The branch target multiplexers have been replaced by a ROM (or – in the FPGA – a block RAM). It acts as a state decoder, too, delivering output signals.

In the diagrams shown above, the multiplexers select condition signals in dependence on the current state. The condition signals are wired to the multiplexer inputs. Attaching another signal to a particular input requires an ECO. In its course, the circuitry has to be synthesized again.

The alternative is to select the condition signals by memory contents too. The circuitry must be synthesized only once. All subsequent changes are done simply by altering the memory content.

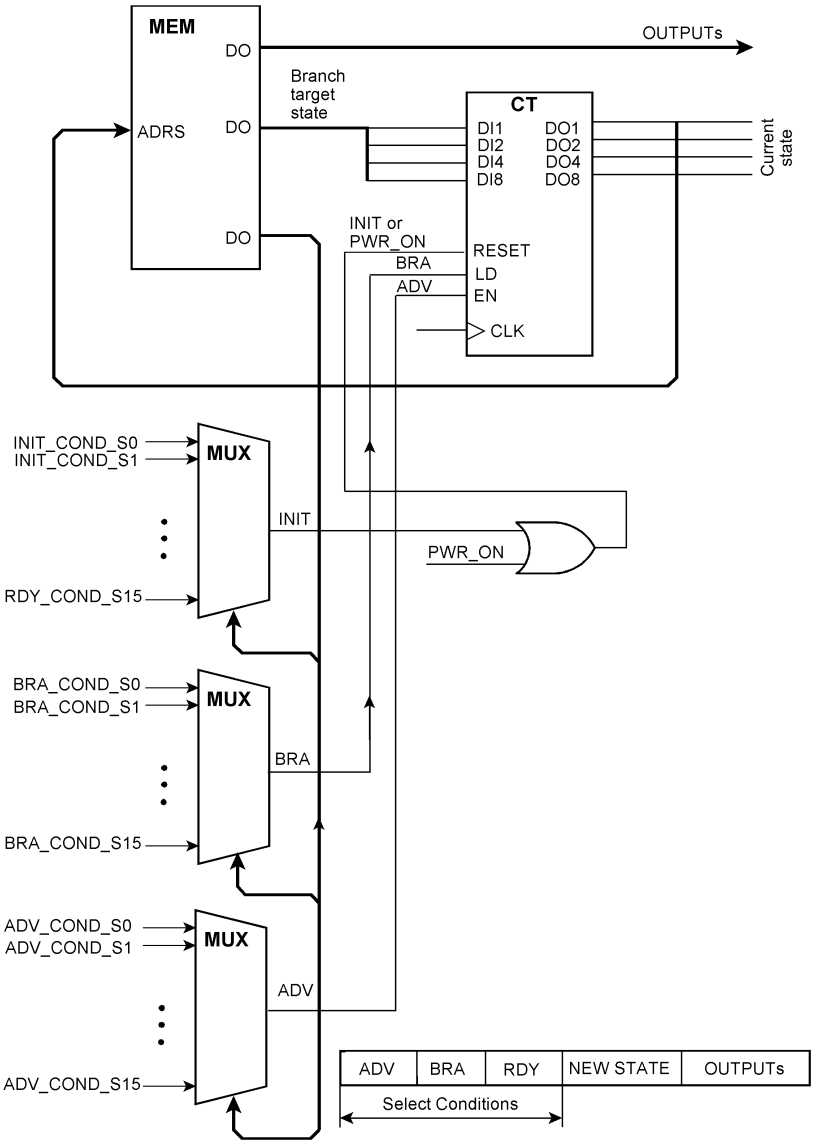


Figure A10 The condition signals are selected by memory contents too. In each state, each transition may be caused by a selected condition. Whether such a device is called a memory-based branch sequencer or a microprogram control unit is a matter of opinion.

Clocking

The sequencers depicted above employ only one clock edge. The state counter equals the CSAR. A memory data register is missing. Instead, the control signals are taken immediately from the memory's outputs. Microprogrammable branch sequencers may be laid out the same way.

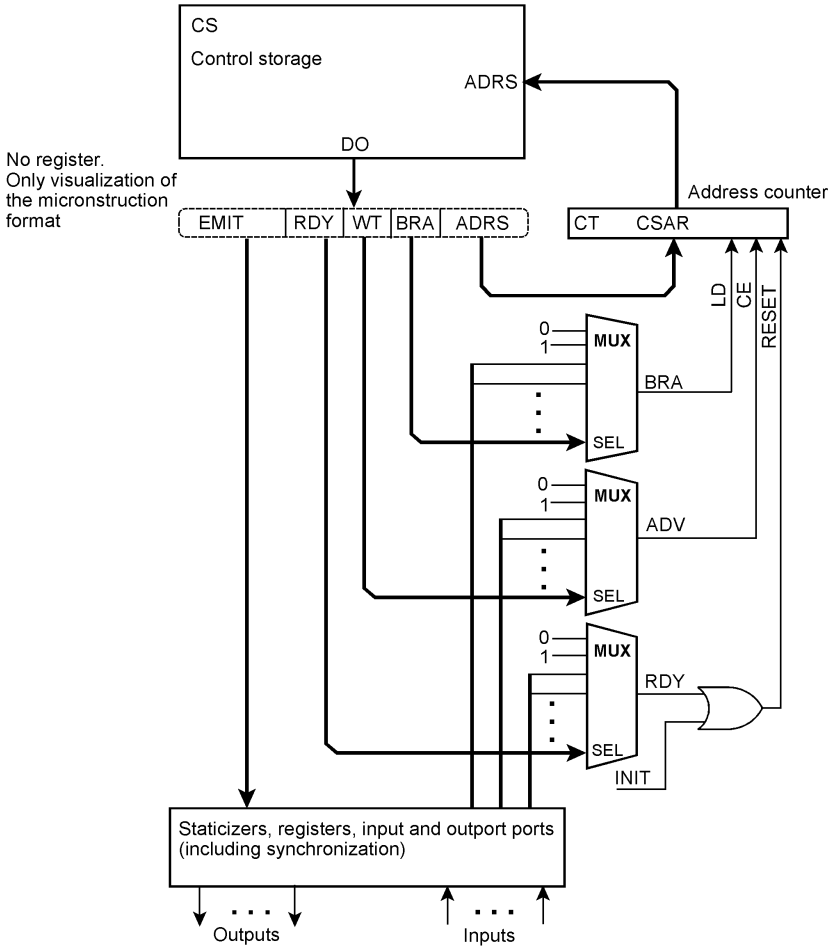


Figure A11 A microprogrammable branch sequencer without a CSDR. Principally, it is the same circuit as shown above.

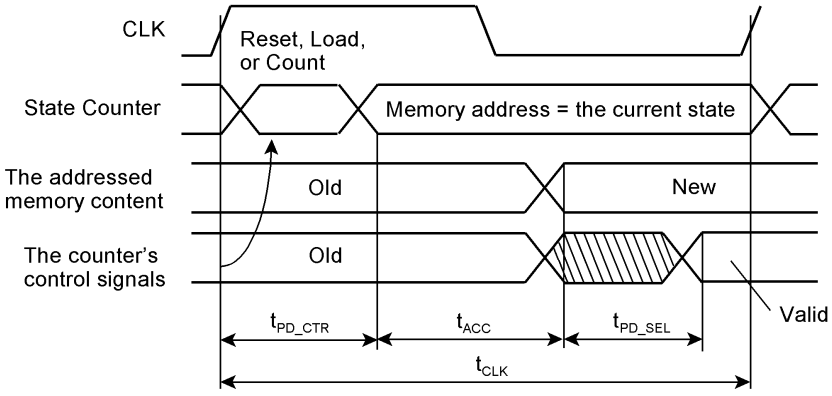


Figure A12 The single-phase clock cycle of the memory-based or microprogrammed branch sequencer without memory data register or CSDR, respectively. t_{PD_CTR} is the propagation delay of the counter; t_{ACC} is the memory access time; t_{PD_SEL} is the propagation delay of the multiplexers selecting the condition signals.

All the clocking into the flip-flops must be done by the leading edge of the clock signal. Beware that all conditions coming from outside must be synchronized, and a sufficient settling time between clocking in the asynchronous signal and evaluating the condition is to be ensured.

When a memory data register (the CSDR) is inserted, we need at least two clock phases. The most straightforward two-phase clock is the single-phase clock with both edges employed.

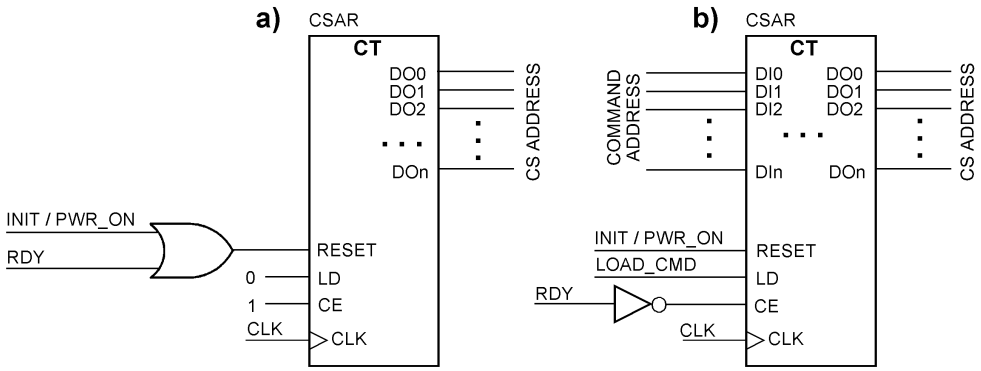


Figure A13 A dual-phase clock cycle employing both edges of the clock signal. P1 causes the CSDR, P2 the CSAR to be loaded.

An often implemented alternative is to retain the single-phase clock. A memory data register allows for some kind of pipelining. The leading edge of the clock causes the CSDR to be loaded with the new microinstruction and the address in the CSAR to be incremented, thus addressing the next microinstruction. It is already read while the current microinstruction is executed. Hence the microinstruction cycle could be nearly as short as the memory access time. Branching, however, will not fit in this scheme. Because the branch target address and the code which selects the branch condition come from the CSDR, branching will require two cycles. Many microcontrollers and RISC processors operate this way.

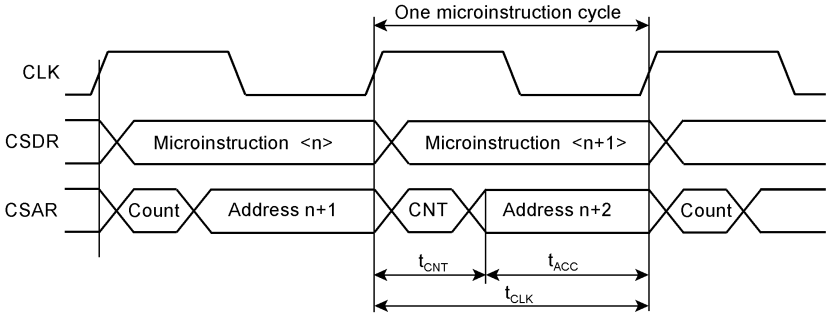


Figure A14 A single-phase clock allows for some kind of pipelining.

An enhanced single-address machine

Additional data paths support operations embracing an immediate operand out of the instruction, a memory operand, and the content of the address (AD) register. The new types of operations are imm OP <mem>, <AD> : = <AD> OP imm, and <AD> : = <AD> OP <mem>. (imm is the immediate operand out of the emit-field, mem is the memory operand.)

Reading data out of the program memory of a Harvard machine is somewhat intricate, requiring

a hard-wired sequencer. Here, we hint only at the address path. Data will flow via the CSDR to the ALU. Such data accesses are encoded in the microinstruction's operation field.

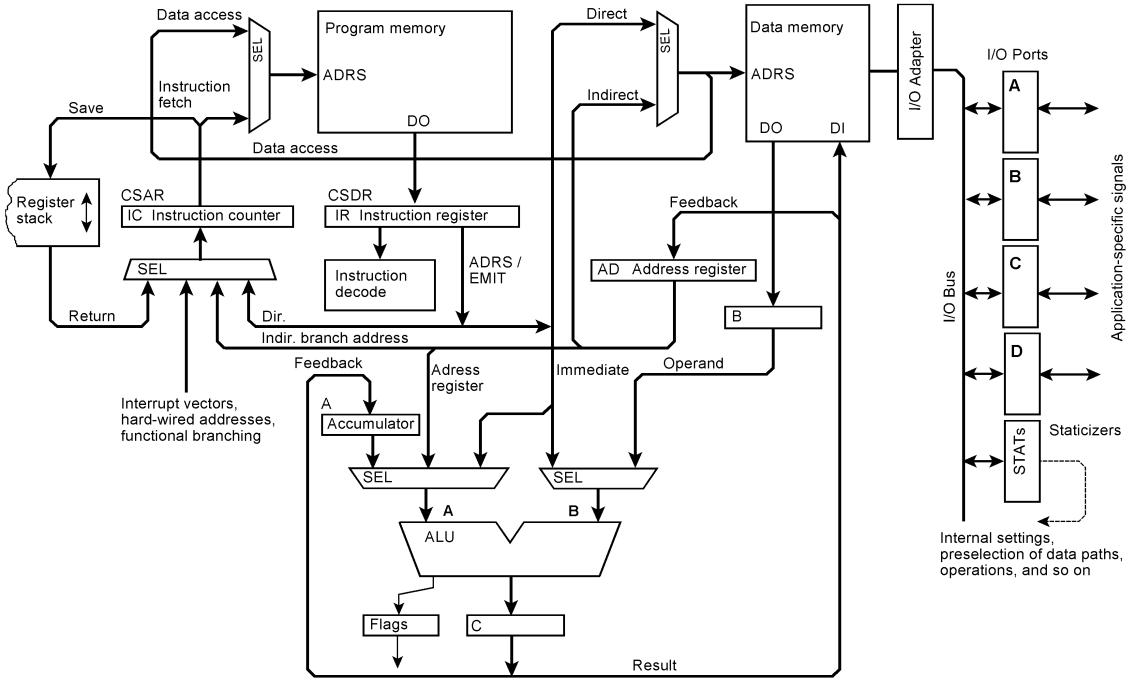


Figure A15 The single-address machine shown in Figure 13 has been enhanced by additional data paths and provisions to read data out of the program memory.

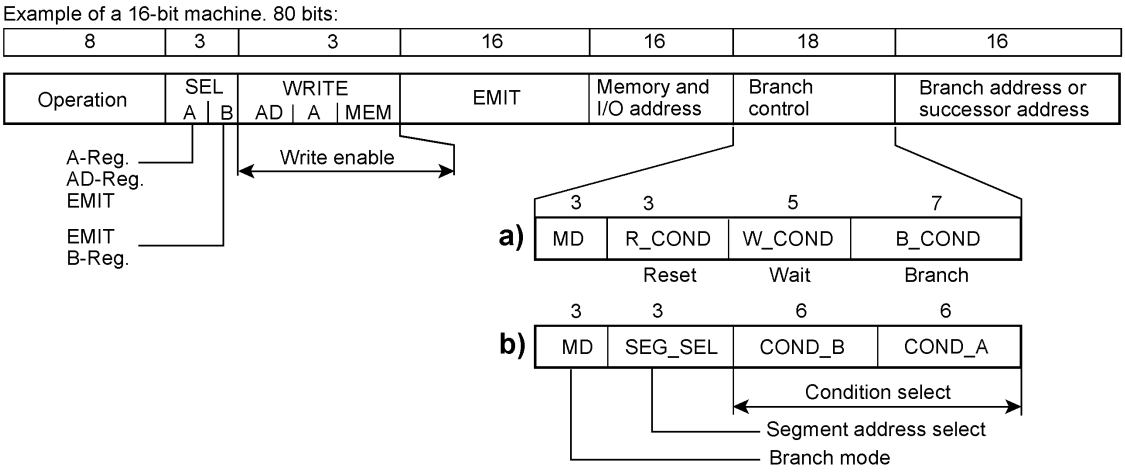


Figure A16 A tentative sketch of a horizontal microinstruction format. Optimization and savings were not even tried. The example concerns a 16-bit machine (16-bit words, 16-bit addresses).

The MD field encodes the branching mode. (a) shows a format that supports the state transitions of the general-purpose branch sequencer, as depicted in Figure 2d or A6. (b) is a format of multiway branching by inserting two condition signals into the successor address. The SEG_SEL

field encodes in which bit positions the conditions are to be injected. Other MD codes concern functional branching, subroutine call, and return.

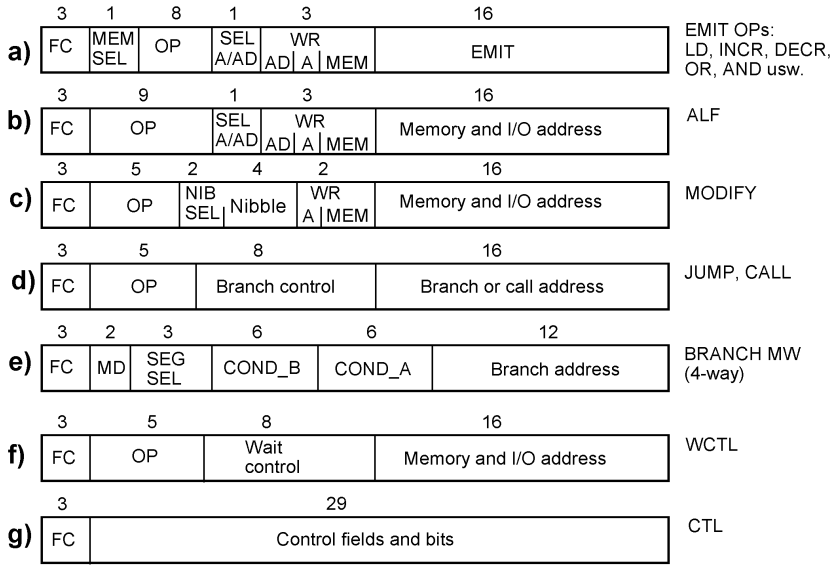


Figure A17 32-bit vertical – or if you will – diagonal formats.

For a small, humble microprogrammed core, 80-bit microinstructions are way too much. Here we have tried to get along with 32 bits.

The format (a) contains the emit-field. The memory must be addressed via the AD register. MEMSEL designates an operation imm OP <<DA>>. Then the content of the emit-field flows to the ALU's A input. Otherwise, only operations <A> OP imm or <AD> OP imm are possible.

The format (b) is the conventional single-address format supporting operations <A> OP <mem> or <AD> op <mem>. Indirect addressing via the AD register is encoded by address = 0.

Format (c) supports operations with the addressed memory operand alone. These operations affect only a selected 4-bit nibble within the memory operand. Typical operations set and clear bits, clear nibbles or whole operands, shift operands, and so on.

The formats (d), (e), and (f) have been provided to support the different flavors of branching, including waiting and multiway.

Format (g) may accommodate application-specific control fields and bits.

Microprogram control of autonomous accelerators
Basically, there are two kinds of accelerators, the application-specific operation unit attached to the processor core and the fully-fledged special-purpose processor. The latter runs complex algorithms that process voluminous data structures (think, for example, of rendering graphics or supporting machine learning, vector processing, matrix algebra, and the like).

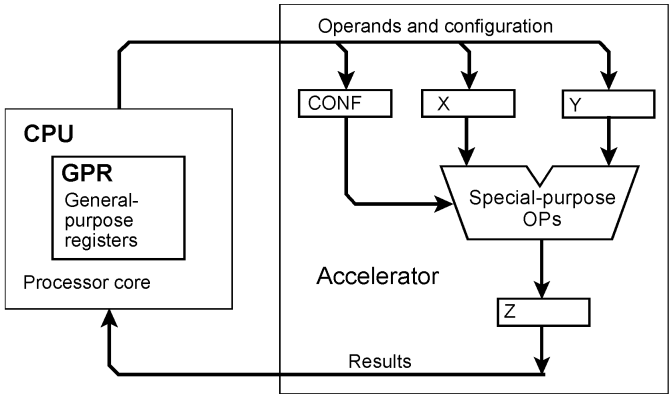


Figure A18 A small accelerator accompanies the operation unit of a RISC core.

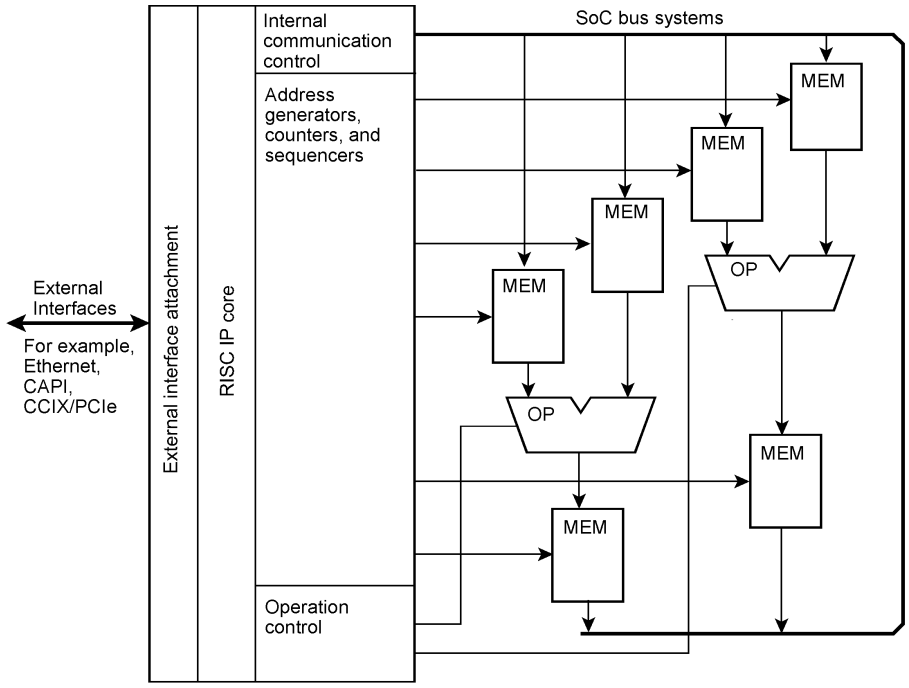


Figure A19 Large accelerators are special-purpose processors working autonomously. The less performance-critical tasks are dealt with by a RISC IP Core.

The accelerator's memories, data paths, operation units, and so on are adapted to the requirements of the particular algorithms. The utmost goal of accelerator design and programming should be that each machine cycle contributes to the final result according to the processing width of its circuitry (think, for example, that a vector or matrix component will be stored during each of the machine cycles).

Complex algorithms are encoded by nested loops. It is essential to accelerate the innermost loop.

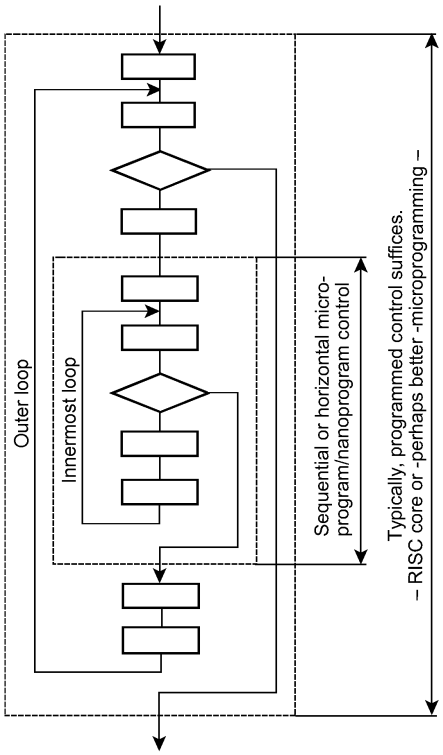


Figure A20 Nested loops. Above all, the innermost loop has to be accelerated.

Accelerating the innermost loop requires an appropriate RTL design of the data paths and operation units, including dedicated memories. This circuitry is to be controlled by synthesized sequencers or horizontal microprograms or nanoprograms, respectively. (For nanoprogramming, refer to [2], [7], and [8].) Nanoinstructions are wide horizontal microinstructions that control the data paths and registers with minimum decoding delay. Thus they are well-suited for very short machine cycles.)

When our demand is met (that is, when the innermost loop produces one partial result in each machine cycle), it is typically not necessary to accelerate outer loops to the same degree. The industry-standard solution is to let a RISC IP core take over. Here, microprogram control could come into play again. Microinstructions may control the address generators, operation units, and other functional units right away, whereas RISC cores can query and control the application-specific circuitry only via I/O accesses. Furthermore, microprogram interrupts, the so-called break-ins (see [1] and [2]), have considerably shorter latencies than typical RISC interrupts.

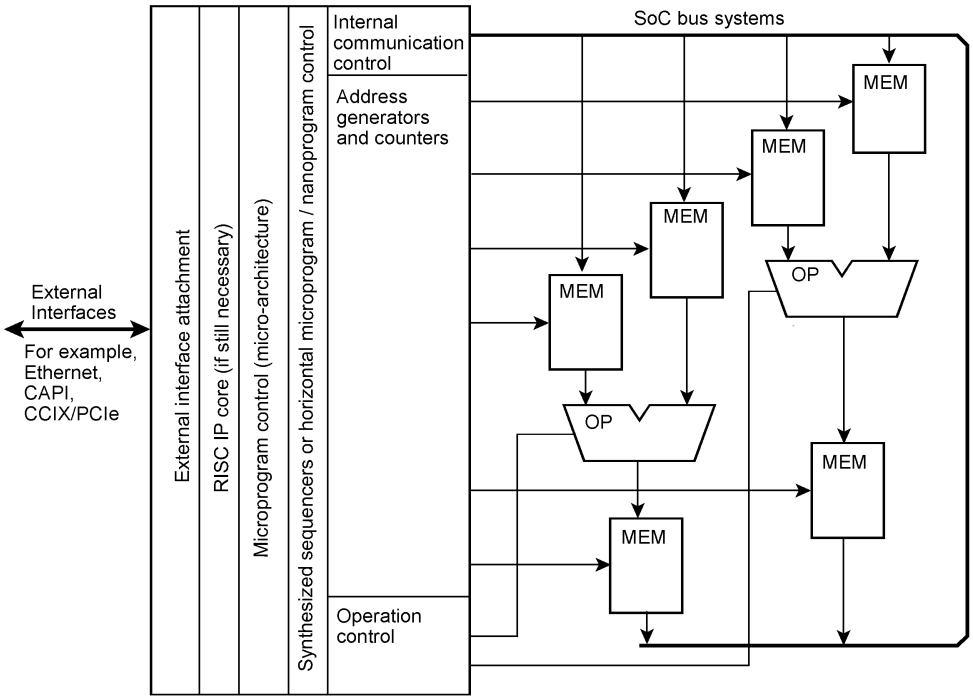


Figure A21 Only the innermost loops need utmost speed and seamless operation. Here, horizontal microprogramming or nanoprogramming may be a viable alternative to hard-wired (synthesized) sequencers and be it because such control sections may be altered so easily. In controlling outer loops and housekeeping activities, microprogrammed control units may have advantages too.

References

[1] Matthes, Wolfgang: Microprogramming Choices Explained (Part 1). Circuit Cellar, Issue 378, January 2022, p. 26-35.

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

[3] Chertok, Boris Evseevich: Rockets and People. For volumes. NASA History Series SP-2005-4110, SP-2006-4110, SP-2009-4110, SP-2011-4110. National Aeronautics and Space Administration NASA History Office, 2005 bis 2011.

[4] Stakem, Patrick H.: The History of Spacecraft Computers from the V-2 to the Space Station. PRRB Publishing, 2009 und 2011.

[5] 1976 Engineering Manual & Purchasing Guide No. 760. Allied Electronics, 1976. http://bitsavers.trailing-edge.com/components/alliedElectronics/catalogs/1976_Allied_Electronics_Catalog.pdf

[6] IEEE Std 1149.1 (JTAG) Testability Primer. SSYA002C, Texas Instruments, 1996.

[7] Parker, Ken P.: Boundary Scan Handbook. Kluwer Academic Publishers, 2002.

[8] QM-1 Hardware Level User's Manual. Nanodata Corporation, 1972. http://bitsavers.trailing-edge.com/pdf/nanodata/QM1HW.pdf

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

Sources

The author's project homepages:

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

The ultimate archives concerning computer architecture, vintage computers, and components:

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