

AC 2007-745: A TURING MACHINE FOR THE 21ST CENTURY

Christopher Carroll, University of Minnesota-Duluth

CHRISTOPHER R. CARROLL Christopher R. Carroll earned his academic degrees from Georgia Tech and from Caltech. He is Director of Undergraduate Engineering in the College of Science and Engineering at the University of Minnesota Duluth and serves in the department of Electrical and Computer Engineering. His interests include special-purpose digital systems, VLSI, and microprocessor applications, especially in educational environments.

A Turing Machine for the 21st Century

Christopher R. Carroll
Director of Undergraduate Engineering
Associate Professor and Assistant Head
Electrical and Computer Engineering, 271 MWAH
University of Minnesota Duluth 55812-3009

Abstract

The Turing Machine, originally proposed in 1936, is a primitive model which nevertheless embodies concepts that form foundations for modern computer design.¹ Described in this paper is an implementation of a Turing Machine core that is useful as a vehicle for teaching finite state machines. It is adaptable to many levels of state machine design, from introductory digital circuit implementations with gates and flip-flops through industrial Programmable Logic Controllers (PLCs) to advanced designs using Programmable Logic Devices (PLDs) and other high-end components. The Turing Machine excels as a vehicle for learning finite state machine design due to its simple structure and easily-understood operation. The particular Turing Machine core described here is especially useful due to the visible display of the machine operation produced on a standard oscilloscope to portray how the user's design works.

A basic Turing Machine consists of a theoretically infinite *tape* on which information is stored in *cells*, and a *head* that moves back and forth across the tape reading and modifying information found there. By controlling the head movement and what it writes to the cells in response to information present on the tape, various computing tasks can be demonstrated.

In the Turing Machine core described here, there are *two* parallel tapes each holding sixteen cells. In each cell is stored one of four symbols (**O**, **I**, **—**, or **•**) for use in the computation. The single head reads information at corresponding cells on both tapes, writes updated information to both cells, and then moves either left or right along the tapes, in response to instructions from the controlling state machine.

The unique characteristic of this design is the display produced on a standard oscilloscope to record operation of the Turing Machine. The oscilloscope display shows the contents of the two tapes as two parallel strings of sixteen symbols each, chosen from **O**, **I**, **—**, or **•**, with one of those symbols in each string raised on the screen to indicate the head position. The display is similar to that produced by other instruments designed by the author and reported in earlier ASEE papers.^{2,3} As the Turing Machine operates, the head position moves back and forth, and the tape contents are updated as controlled by the state machine being used.

Background

The Electrical and Computer Engineering curriculum at the University of Minnesota Duluth includes an introductory “digital circuit design” class covering topics that parallel those covered in similar classes in any other electrical or computer engineering program across the country. The course teaches students the basics of digital circuit design, covering combinational circuits built from logic gates, and synchronous sequential circuits that additionally include flip-flops to implement memory. Beyond the basic “gates and flip-flops” design strategies, the course explores higher level circuit structures built from the basic components, such as decoders, data selectors, counters, and shift registers. None of this is noteworthy. The same outline describes introductory digital design courses at most any university.

At the University of Minnesota Duluth, this course is required both for engineering students pursuing the Electrical and Computer Engineering degree, and for students pursuing the Computer Science degree. This, too, is common practice at universities across the country. The presence of both computer science students and engineering students together in the same class requires careful structuring of the course and the course material. The engineering students are eager to start wiring circuits right away to make things work. The computer science students are more interested in the theory behind the circuit structures. Student backgrounds in digital hardware are also vastly different. Engineering students often have deep hobby-level experience in working with digital circuits. Computer science students often have hardly put their fingers across the terminals of a battery, and may be wary of wiring anything called a “circuit.”

Fortunately, the material included in the “digital circuit design” class can be presented in a way that satisfies the needs of both engineering students and computer science students. Nothing electrical in nature is required of students in this course. The material begins like a math course, covering Boolean algebra and the various postulates and theorems related to that, and then builds into the sequential arena by adding the concepts of memory implemented with flip-flops. Finite state machine design is presented as a mechanical exercise with specific design steps that lead to a solution in a very methodical, structured way, again just like a math course. The course can avoid both electrical details that are unwanted by the computer science students and also the computer structure details that may be unwanted by the electrical engineering students. Students do design and build digital electrical circuits, but no discussion of the electrical nature of the circuits is included. The power supply is just a source of 1’s and 0’s for the computation, and the wires are just physical implementations of pencil lines on paper. Of course, eventually the two groups of students must diverge and pursue those details related to their major disciplines. Conveniently, that divergence comes right at the end of this introductory “digital circuit design” course. The engineering students follow this course with a “digital computer circuits” course that does teach them the electrical properties and limitations of digital circuits, and further explores circuit structures that contribute to the foundations of computer circuit design. Computer science students follow this course with a “computer organization” course that explores higher level aspects of computer architecture and structure.

The topic of this paper, the Turing Machine, is an ideal topic to end the introductory course and to launch students into computer design in either the engineering direction or the computer science direction. The Turing Machine is a simple example of a digital computer that is easily

understood and manipulated, but that models vastly more sophisticated digital computer designs. The core of the Turing Machine can be implemented in hardware in many ways, one of which is demonstrated in this paper. *Programming* the Turing Machine to perform a computation simply requires designing a finite state machine to control the head movement and data written by the head. That state machine design can be realized at many levels, from basic gate and flip-flop design techniques preferred by engineering students to abstract, table-driven techniques that isolate the designer from hardware altogether, which may be preferred by computer science students. Regardless of the design approach used to control the Turing Machine core, the result is a computing machine that solves a computational problem in a visible, easily understood way.

Turing Machine Core

As mentioned earlier, the core of a Turing Machine consists of a theoretically infinite *tape* containing *cells*, and a tape *head* that reads and modifies the contents of those cells. Each cell stores one of a finite number of *symbols*, and as the head moves over the tape, it reads the symbol in the cell under the head, writes a new symbol there to replace the original one, and then moves either left or right to the adjacent tape cell. Computation is performed by controlling the movement of the head and the symbols written by the head in response to information stored in the symbols present on the tape.

In the particular implementation of the Turing Machine core described here, *two* parallel tapes are implemented, each holding sixteen cells, and each cell stores one of four symbols, selected from **O**, **I**, **—**, or **•**. A single head reads information stored at corresponding cells on both tapes, writes new symbols into those cells, and then moves left or right as directed by the controlling finite state machine during each clock cycle of operation. Different computations are performed on the tape's data by designing appropriate finite state machines to control the head.

The implementation of the Turing Machine core described in this paper produces a unique display of the contents of the two tapes in the machine, as two rows of symbols generated on a standard oscilloscope screen. The head position is indicated in the display by elevating one of the symbols (at corresponding locations on each tape), so that as the head moves left and right during a computation, the symbols under consideration at that step of the computations rise on the screen, then are modified to new symbols by the head, and then fall back to their original positions as the head moves left or right, where the neighboring symbols in that direction then rise and the process repeats for the new clock cycle of computation. Figure 1 shows a photograph of the oscilloscope screen during a computation on this Turing Machine.

Tape Implementation

The tape portion of this Turing Machine implementation is built using four 74LS170 chips, each of which is a 4-by-4 dual-port RAM. The 74LS170 includes four RAM locations, each storing four bits of data. Four of these chips are used, for a total of sixteen locations, or cells, in which the Turing Machine tape data are stored, with the upper two bits of data in each cell storing the character from the upper tape, and the lower two bits of data in each cell storing the character from the lower tape on the display. The 74LS170 is used here because there are two separate ports, or access mechanisms for the locations in the memory, one for reading data from the

locations, and one for writing new data to those locations. The two ports are totally independent and allow reading from one location while writing to an entirely different location in the memory. Separate address inputs to the 74LS170 chips allow specifying separate “read” addresses and “write” addresses, and data leaves the chip and enters the chip on different pins under control of separate output enable and write enable signals, so that read and write processes are indeed truly independent. With a total of sixteen locations among the four chips, a 4-bit address is used to identify a location for reading or writing. The top two bits of each of these addresses are decoded to activate the output enable or the write enable input on the appropriate chip, according to the read or write addresses specified.

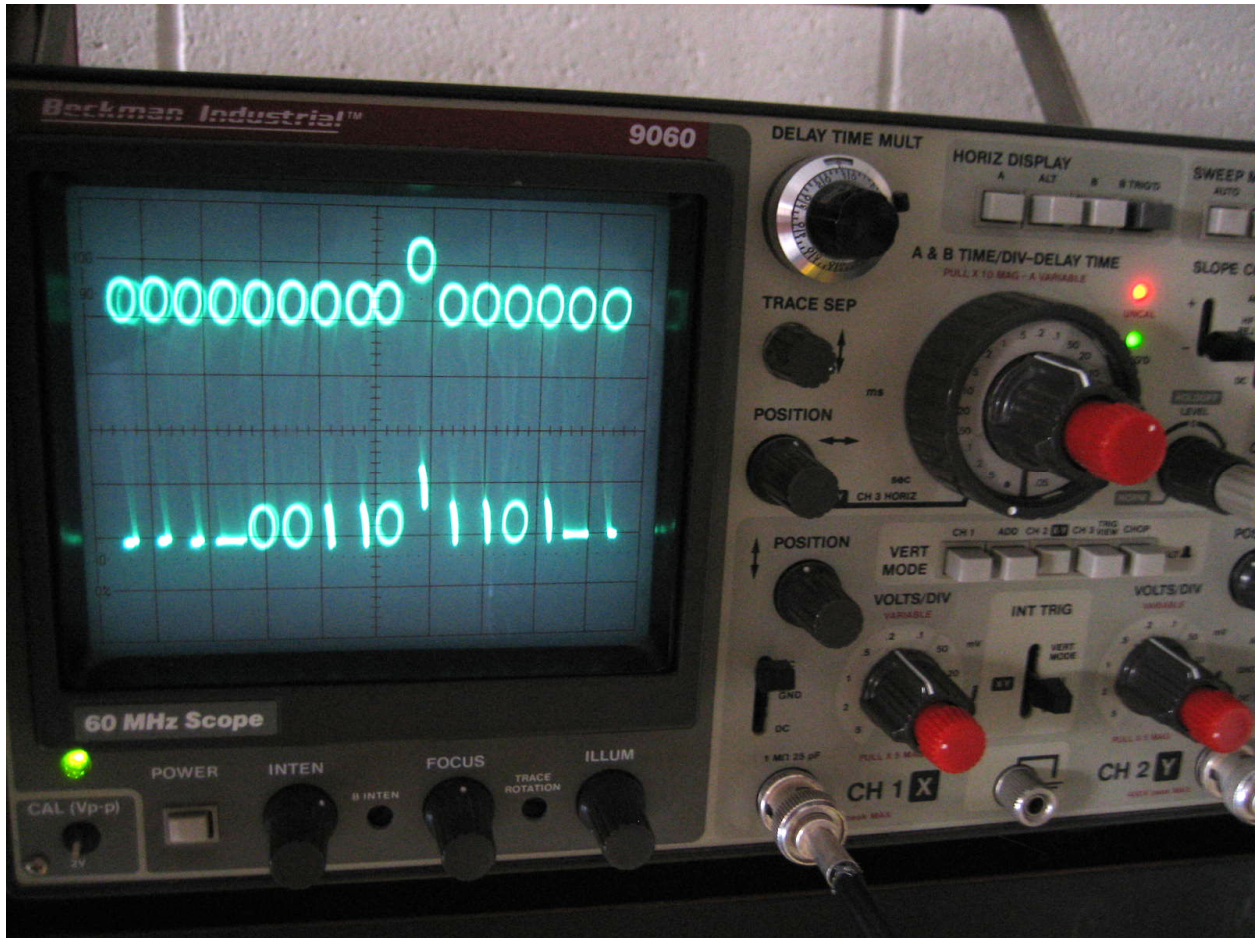


Figure 1. Turing Machine display in the midst of a computation. (Upper tape is not used here.)

The display of symbols contained on the tapes is produced continuously by constantly incrementing a counter through the RAM locations storing tape information. A 5-bit counter is used here, with the least significant bit identifying the tape being read, upper or lower, during each clock cycle, and the other four bits identifying the cell number on the tapes. The least significant bit of this 5-bit address gates either the upper two or lower two bits of data read from the RAM to the symbol-producing circuits to be described next, so that as the 5-bit counter advances, a symbol from the lower tape is displayed, then the symbol from the corresponding location on the upper tape is displayed, then the next symbol to the right on the lower tape, etc.,

until all 32 symbols have been displayed. The process repeats continuously, independently of what is being written by the head through the other RAM port.

The symbols presented on the oscilloscope are produced as Lissajous figures. To produce the **O** symbol, the 60 Hz sine wave from the power supply modulates one axis of the oscilloscope display, while a second sine wave, phase shifted by nearly 90 degrees, modulates the other axis, producing an approximate circle on the screen. To achieve the other three symbols (**I**, **—**, or **•**), one or both of the axis modulation signals are turned off by analog switch components, effectively squashing the circle to a vertical line, a horizontal line, or a dot for the other symbol representations. The analog switches are controlled by the two bits of data stored in the upper or lower half of the RAM location addressed by the 5-bit counter mentioned above, cycling through the locations on the RAM that implement the Turing Machine tape cells. The four upper bits of that 5-bit counter are converted to an analog signal by an R-2R network of resistors to position the symbol at one of sixteen locations horizontally on the oscilloscope screen. The least significant bit of the 5-bit counter modulates the vertical deflection of the oscilloscope to position the symbol on the upper or lower tape display.

Head Implementation

The tape head of the Turing Machine is located at one of the sixteen cells of the tapes, identified with a 4-bit address. That address is compared with the top four bits of the 5-bit counter that cycles through the RAM locations to produce the display, and when they match, a signal is produced that deflects the oscilloscope display vertically, raising the symbols at that horizontal location to show the position of the Turing Machine head. The 4-bit head address is held in an up/down counter, so that the head position can move either right or left as the computation progresses, as controlled by the user's state machine.

The operation of the circuitry implementing the Turing Machine head is quite simple. On each clock cycle of the computation, the 4-bit head address identifies the tape location to be modified. Data at that location from the Turing Machine tape is read through the process of updating the display, described above. The user's finite state machine, based on the data read, determines new data to be written to that cell, and in the middle of the clock cycle (on the falling edge of the system clock) a pulse is generated on the write enable of the selected 74LS170 RAM chip to write that new data. Then, at the end of the clock cycle, the 4-bit head address either increments or decrements, as determined by the user's finite state machine, and the Turing Machine enters the next cycle of the computation. The process repeats, until the user's state machine activates a special "halt" signal which stops the operation of the Turing Machine when the computation is completed.

Example Computations

The Turing Machine core described above does nothing without a finite state machine to control it. The core provides four input variables to the state machine (two bits each from the symbols being read by the head on each of the two tapes). There are six output variables from the state machine to control the core. Four variables are the two symbols to be written to the two tapes (two bits each). One variable controls the direction of movement for the head, left or right. The

last output variable is the “halt” signal that stops the Turing Machine operation when the computation is completed. The Turing Machine control unit is just this synchronous finite state machine with the four input variables and six output variables identified above.

One example computation is to increment a binary number stored on the tape (most significant bit on the left) using symbols **O** and **I** to represent the bits of the number, and using the other two symbols, **—** and **•**, to frame the number on the tape on both sides. A state diagram description of the controlling finite state machine is shown in Figure 2. In this case, only one tape in the Turing Machine is used. The labels on the arcs show before the slash the input symbol read from the tape, the output symbol that replaces the input symbol on the tape, the direction the head is to move (L or R), and the Halt signal if activated. The head begins and ends on the cell containing the least significant bit of the number to be incremented.

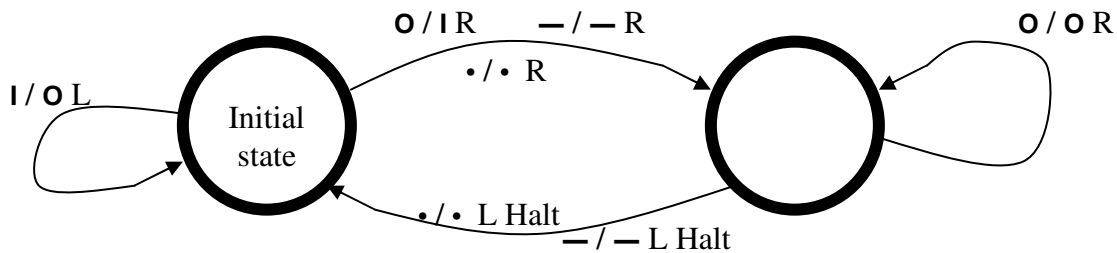


Figure 2. State diagram for finite state machine to increment a number on the tape.

A second example that demonstrates how having two parallel tapes in the Turing Machine can be useful adds a binary number stored on the upper tape (using **O** and **I** symbols, framed with **—** and **•** symbols) to a second number (of the same number of bits) stored in the same position on the lower tape. Again, the most significant bit of the numbers is to the left, and the head starts on the cell containing the least significant bits of the numbers, but here the head ends on the cell containing the most significant bits. The state diagram for this example is in Figure 3. In this case there are two input symbols (upper tape symbol followed by lower tape symbol), then the slash, the two output symbols, the head direction, and Halt signal as before. On the arc labels, the symbol Φ indicates a “don’t care” condition.

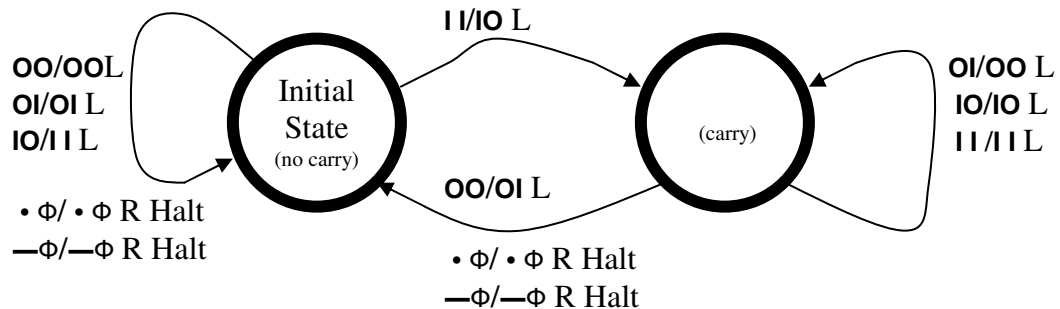


Figure 3. State diagram for finite state machine to add upper tape number to lower tape.

In both of these examples, the finite state machine involved is described with a state diagram, which is the way an engineering student normally begins state machine designs. Alternatively, a table representation could be used to convey the same information, with the rows in the table identified by the current state of the state machine and the input symbols present. Each row of the table would include next state and output information to be produced by the state machine. This less graphic representation of the state machine design may be more suited to design approaches in which the designer uses a programmable logic controller or other programmable device to implement the state machine, rather than building it directly from digital components. Any approach is fine, as long as the resulting state machine interacts with the input variables and output variables of the Turing Machine core in the same way.

Conclusion

The Turing Machine design described in this paper serves several functions in an introductory digital circuit design class. First, it is an excellent foundation for exploring synchronous finite state machine design, one of the primary topics included in any beginning digital course. The visible representation of the Turing Machine operation produced by the graphic display on a standard oscilloscope aids students in understanding the functioning (and *malfunctioning!*) of the state machines they design. Second, the Turing Machine model serves to unite engineering students and computer science students around a common design foundation, one in which each group of students can discover techniques useful in their distinct disciplines. Finally, the design of the Turing Machine core is accessible to students, and the unique oscilloscope display inspires them to ask “Hey, how’d they do that?” which is always a goal in educational settings. This tool will be a valuable addition to the digital circuit design lab.

References

1. web page, http://en.wikipedia.org/wiki/Turing_machine
2. Carroll, C. R., “The Chipmonk: An Inexpensive Digital Circuit Tester,” *Proceedings of the 1999 North Midwest Section Meeting of ASEE*, Winnipeg, Canada (1999).
3. Carroll, C. R., “Digital Logic Lab Experiments Using the Chipmonk Instrument,” *Proceedings of the 2001 ASEE North Midwest Section Meeting of ASEE*, Grand Forks, ND (2001).