

Counting NOR Switches in the TOY-8 CPU

Jeffery Osagie

Abstract

TOY-8 is a stored-program computer with one accumulator, a 4-bit program counter, and sixteen bytes of memory. Its arithmetic, storage, and control circuits can be expressed using NOR gates and input-controlled switches. This document follows a summation program through the CPU's datapath and derives the hardware cost of each component. Expanding the implemented Boolean circuit into NOR/NOT circuits gives 2038 NOR gates and 3371 switches. Memory is the SRAM derived in Section 7.4 and accounts for the largest share, mostly in its 120 storage cells.

1 Introduction

The TOY-8 CPU has a single 8-bit register R, an 8-bit instruction register IR, a 4-bit program counter PC, and a 16×8 memory. An instruction's upper nibble specifies the operation; its lower nibble supplies an address or immediate value. Memory location 0 supplies a constant zero. Location F provides input and output: LDM F reads an input byte, and STM F emits the byte held in R.

The datapath connects memory and R to the ALU. The ALU result returns to R through its input multiplexer, which also accepts memory data, the zero-extended low nibble of IR, and the zero-extended PC. IR supplies branch targets to PC; R supplies the return address for RET. During BAL, the already-incremented PC passes into R while PC receives the branch target. These paths let the same accumulator serve arithmetic, memory operations, and subroutine calls.

The phase ring advances through Fetch (F), Fetch Write (FW), Execute (E), and Execute Write (EW). F selects the instruction address and computes the sequential PC. FW captures the instruction in IR and advances PC. E selects operands and prepares an operation's result; EW writes the result to its destination. Separating selection from capture keeps each source stable while the receiving register or memory cell takes its value.

Figure 1 shows the complete CPU during an addition in the summation program. Memory supplies the running sum, R supplies the new input, and the ALU computes their sum for capture in R. A program changes the stored values and switch states; the hardware count remains the same throughout execution.

2 The demonstration program

By default, `CpuComponentsDrawer` initializes an embedded copy of `sum.toy` and starts at $PC = 1$. The program adds input bytes until it reads zero, then outputs the accumulated sum. The default inputs are 8, 5, and 0, so the output is 13. Addresses and instruction words in Table 1 are hexadecimal.

Address	Word	Instruction	Operation
01	A0	LDM 0	$R \leftarrow 0$
02	CE	STM E	$M[E] \leftarrow R$
03	AF	LDM F	Read the next input into R
04	E9	BRZ 9	If $R = 0$, branch to 9
05	2E	ADD E	$R \leftarrow R + M[E]$
06	CE	STM E	$M[E] \leftarrow R$
07	A0	LDM 0	$R \leftarrow 0$
08	E3	BRZ 3	Branch to the input at 3
09	AE	LDM E	Load the accumulated sum
0A	CF	STM F	Output R
0B	00	HALT	Stop the clock

Table 1: The summation program loaded by the drawing.

Location E holds the running sum. The first pass adds 8 to its initial zero; the second adds 5 to the stored 8. Loading zero at address 7 makes the following BRZ unconditional. When the input itself is zero, the branch at address 4 exits the loop and the program outputs the value in E. Addition retains the low eight bits, so accumulated sums wrap modulo 256.

Consider the second execution of ADD E, at address 5. R holds 5 and $M[E]$ holds 8. The following snapshots show how the CPU produces 13:

Frame	Phase completed	Result
41	F	Read instruction 2E; compute $PC + 1$
42	FW	Capture $IR = 2E$ and $PC = 6$
43	E	Compute $5 + 8 = 13$; R still holds 5
44	EW	Capture 13 in R

Figure 1 uses frame 43. IR selects ADD; its low nibble selects memory E. The ALU receives $R = 5$ and $M[E] = 8$, and the R mux routes the sum to R's data inputs. EW is the next phase, so the control unit selects R write; the next capture pulse stores the sum. The crimson and steel-blue wires show the settled logic values at this point in the run.

The drawing records the run once and opens paused. Space advances one phase, B moves back, and P plays the recorded phases. `--inputs` sets the decimal input-byte sequence; exhausted input supplies zero. From the section directory, Figure 1 can be reproduced with:

```
java CpuComponentsDrawer.java --frame=43 \
  --scale=1 --save=cpu-components.png
```

2.1 Selecting a program

`--program` loads a TOY-8 memory image from a file, and `--pc` selects the initial address in hexadecimal. Program files contain hexadecimal address and byte pairs, such as `01: A0`. The shared `Toy8Program` loader is also used by the TOY8 simulator. Location 0 remains constant zero; code occupies addresses 1 through F. For example, the subroutine demonstration can be opened with:

```
java CpuComponentsDrawer.java --program=bal.toy --pc=1
```

Loaded programs start with an empty input queue. `--inputs` supplies decimal bytes from 0 to 255; exhausted input supplies zero. The drawing identifies the selected program and initial PC in its header.

3 Counting method

The circuit built by `CpuComponentsDrawer` contains 1231 Boolean gates. Expressing each gate as a NOR/NOT circuit gives 2038 NOR gates containing 3371 input-controlled switches.

A k -input NOR contains k normally-closed switches. A NOT is a one-input NOR and contains one switch. OR adds an output NOT to NOR; AND applies NOR to inverted inputs; NAND adds an output NOT to that AND circuit:

$$\begin{aligned}\text{OR}(x_1, \dots, x_k) &= \text{NOT}(\text{NOR}(x_1, \dots, x_k)), \\ \text{AND}(x_1, \dots, x_k) &= \text{NOR}(\text{NOT}(x_1), \dots, \text{NOT}(x_k)), \\ \text{NAND}(x_1, \dots, x_k) &= \text{NOT}(\text{NOR}(\text{NOT}(x_1), \dots, \text{NOT}(x_k))).\end{aligned}$$

These replacements are applied gate by gate, retaining the assembly's gate instances and shared signals. Wires and hardwired constants contribute zero. Let G_{logic} count Boolean gates, G_{NOR} count NOR gates, and S_{NOR} count switches.

For example, a 2-input AND becomes two input NOTs followed by a 2-input NOR. The NOTs contribute one switch each and the NOR contributes two, giving three NOR gates and four switches. This same conversion applies to the AND terms used in register write inputs, muxes, and control logic.

Boolean gate	NOR gates	Switches
NOT	1	1
k -input NOR	1	k
k -input OR	2	$k + 1$
k -input AND	$k + 1$	$2k$
k -input NAND	$k + 2$	$2k + 1$

3.1 XOR

The XOR circuit consists of four 2-input NANDs:

$$u = \text{NAND}(A, B), \quad v = \text{NAND}(A, u), \quad w = \text{NAND}(B, u), \quad A \oplus B = \text{NAND}(v, w).$$

Each NAND contributes four NOR gates and five switches, so one XOR uses 16 NOR gates and 20 switches.

4 Registers, clock, and control

4.1 R and IR

Each stored bit has a gated set/reset input and two cross-coupled NORs:

$$\bar{D} = \text{NOT}(D), \quad S = \text{AND}(E, D), \quad T = \text{AND}(E, \bar{D}), \quad \bar{Q} = \text{NOR}(S, Q), \quad Q = \text{NOR}(T, \bar{Q}).$$

Here E enables capture, and S and T are the set and reset signals. With capture disabled, both inputs to the storage pair are inactive and feedback preserves the bit. Enabling capture selects the set or reset input according to D . The five Boolean gates expand to nine NOR gates and 13 switches. Each 8-bit register also has a shared AND combining its write signal with the clock pulse. Thus, for either R or IR,

$$G_{\text{logic}} = 8(5) + 1 = 41, \quad G_{\text{NOR}} = 8(9) + 3 = 75, \quad S_{\text{NOR}} = 8(13) + 4 = 108.$$

4.2 Clock and phase ring

The ring holds one active phase bit and transfers it to the next latch on each enabled pulse. Its four outputs let the control unit distinguish operand selection from register capture. The oscillator consists of one 2-input NOR and three NOTs. Run/halt logic controls its pulse, and the phase ring sequences Fetch, Fetch Write, Execute, and Execute Write.

Part	Logic gates	NOR gates	Switches
Oscillator	4	4	5
Run/halt, reset, and phase enable	4	8	11
Next-phase terms and reset injection	5	14	19
Phase latches	20	36	52
Clock and ring	33	62	87

4.3 Control unit

The opcode decoder uses four shared NOTs and sixteen 4-input NORs. An 8-input NOR detects zero in R, and a 4-input AND detects address F in IR[3:0]. Phase and opcode terms drive sixteen gated control outputs and three wire aliases.

Part	Logic gates	NOR gates	Switches
Opcode decoder	20	20	68
Zero detector	1	1	8
Address-F detector	1	5	8
Phase/opcode terms	8	18	35
Control-output gates	16	49	67
Control	46	93	186

5 ALU and R input selection

5.1 ALU

The adder accepts one bit from R, one bit from memory, and a carry from the preceding slice. Its sum and carry are

$$S = A \oplus B \oplus C_{\text{in}}, \quad C_{\text{out}} = (A \wedge B) \vee (A \wedge C_{\text{in}}) \vee (B \wedge C_{\text{in}}).$$

Each full-adder slice therefore uses two XORs, three 2-input ANDs, and two 2-input ORs:

$$G_{\text{logic}} = 2(4) + 3 + 2 = 13, \quad G_{\text{NOR}} = 2(16) + 3(3) + 2(2) = 45, \quad S_{\text{NOR}} = 2(20) + 3(4) + 2(3) = 58.$$

Eight slices form the ripple adder. Bit 0's carry-in is grounded, and bit 7's carry-out is discarded.

For each output bit, three AND terms gate the ADD, AND, and XOR results; a 3-input OR combines them. Shared selector guards enable the chosen operation.

Part	Logic gates	NOR gates	Switches
Full adders	104	360	464
Bitwise ANDs	8	24	32
Bitwise XORs	32	128	160
Selector NOTs and guards	6	15	21
Output selection	32	88	128
ALU	182	615	805

5.2 R mux

R receives the ALU result, a memory word, or the zero-extended low nibble of IR or PC. Let s_i select source X_i . The selector guard h_i and output bit D_b are

$$h_i = s_i \wedge \bigwedge_{j \neq i} \neg s_j, \quad D_b = \bigvee_{i=0}^3 (h_i \wedge X_{i,b}).$$

The shared selector logic uses four NOTs and four 4-input ANDs, contributing 24 NOR gates and 36 switches. Each output bit uses four 2-input ANDs and one 4-input OR, contributing 14 NOR gates and 21 switches. Across eight bits,

$$G_{\text{logic}} = 8 + 8(5) = 48, \quad G_{\text{NOR}} = 24 + 8(14) = 136, \quad S_{\text{NOR}} = 36 + 8(21) = 204.$$

During the illustrated ADD, the ALU guard is active and its result reaches R's data pins. R captures that value when its write pulse arrives. During BAL, the PC guard instead passes the sequential return address into R.

6 Program counter and memory address

6.1 Program counter

The 4-bit incrementer computes $PC + 1$:

$$I_0 = \neg Q_0, \quad I_1 = Q_1 \oplus Q_0, \quad I_2 = Q_2 \oplus (Q_0 Q_1), \quad I_3 = Q_3 \oplus (Q_0 Q_1 Q_2).$$

Fetch captures this result in four staging latches. Fetch Write commits it to PC together with the fetched instruction in IR. The load path selects IR[3:0] for a branch target or R[3:0] for RET. BAL saves the sequential PC in R and loads its target into PC.

Increment and load terms feed the PC input OR. A reset mask precedes each PC latch, and a shared AND supplies the write pulse.

Part	Logic gates	NOR gates	Switches
Incrementer	23	23	34
IR/R load selection	13	33	45
Fetch staging and capture enable	21	39	56
Increment/load selection	12	32	44
Reset masks	8	8	12
PC latches and write enable	21	39	56
PC	98	174	247

6.2 Memory address

The address selector passes PC[3:0] during Fetch and IR[3:0] during Execute. Four latches capture the selected address. A second selector passes that held address during Fetch Write and Execute Write, keeping the memory address stable through capture.

Part	Logic gates	NOR gates	Switches
Selector NOTs and guards	4	8	10
PC/IR selection	12	32	44
Capture logic and address latches	22	41	59
Hold-phase OR and NOT	2	3	4
Live/hold selection	12	32	44
Memory-address path	52	116	161

7 Memory

Memory is the SRAM derived in Section 7.4 ([Gate_Count_Analysis.pdf](#)), with $m = 4$ address bits and $n = 8$ data bits. Each inversion is made once and shared: four address NOTs serve the decoder, one NOT of the write pin WE serves every word, and eight data NOTs serve every column. Word j has a decoder output W_j and three row gates:

$$\overline{W_j} = \text{NOT}(W_j), \quad WS_j = \text{NOR}(\overline{W_j}, \overline{WE}), \quad \overline{WS_j} = \text{NOT}(WS_j).$$

With these complements available, every gate in a cell is a 2-input NOR:

$$S = \text{NOR}(\overline{D}, \overline{WS_j}), \quad T = \text{NOR}(D, \overline{WS_j}), \quad read_j = \text{NOR}(\overline{Q}, \overline{W_j}),$$

plus the latch's two cross-coupled NORs. The read gate takes the latch's own $\overline{Q}$, so no cell needs an inverter. A cell costs five NOR gates and ten switches. Each output bit ORs its column's read terms with a 15-input NOR followed by a NOT. Apart from the AND that forms WE , the circuit is already in NOR form, so the replacements of Section 3 change nothing else.

M[0] is wired to zero, so it has no cells, decoder output, or row gates. Figure 1 still draws word 0 as an ordinary row holding zero.

Part	Logic gates	NOR gates	Switches
Address NOTs and decoder	19	19	64
Write pin and its NOT	2	4	5
Data NOTs	8	8	8
Word and write-select lines	45	45	60
Storage cells (S , T , latch)	480	480	960
Cell read gates	120	120	240
Column NORs and NOTs	16	16	128
Memory	690	692	1465

Section 7.4's formulas give $(5n + 2)2^m + n = 680$ NORs and $m + 2^{m+1} + 2n + 1 = 53$ NOTs at this size, 733 gates in all. Wiring M[0] to zero removes 44 of them: its eight cells (40 NORs), its decoder output, its write select, and its two row inverters. The CPU adds one AND, which forms WE from the clock pulse and the memory-write control. That leaves the 690 gates in the table.

7.1 The read-tree alternative

The column NOR puts fifteen switches in series, one per word. The switch count charges for those switches but not for the delay of a long series chain, and Section 7.4 noted that such wide fan-in scales poorly. A balanced tree of 2-to-1 muxes avoids it. In NOR form a mux needs three 2-input NORs,

$$\text{mux}(s; a, b) = \text{NOR}(\text{NOR}(a, s), \text{NOR}(b, \bar{s})),$$

with $\bar{s}$ taken from the shared address NOTs. A 16-to-1 tree has $8 + 4 + 2 + 1 = 15$ muxes, so eight trees cost $8 \cdot 15 \cdot 6 = 720$ switches. They would replace the read gates, column NORs, and output NOTs, which cost $240 + 128 = 368$. The row gates stay, because writes still need $\overline{W_j}$. Memory would grow to $1465 - 368 + 720 = 1817$ switches. No gate would then have more than two switches in series, but a stored bit would pass through eight gate levels to reach the output instead of three. `Toy8MemoryDrawer` draws this tree for one bit.

8 Total

Module	Logic gates	NOR gates	Switches	Share
Clock and four-phase ring	33	62	87	2.58%
Control	46	93	186	5.52%
ALU	182	615	805	23.88%
R mux	48	136	204	6.05%
R	41	75	108	3.20%
IR	41	75	108	3.20%
PC	98	174	247	7.33%
MA mux and address hold	52	116	161	4.78%
Memory	690	692	1465	43.46%
Total	1231	2038	3371	100.00%

$$G_{\text{logic}} = 1231, \quad G_{\text{NOR}} = 2038, \quad S_{\text{NOR}} = 3371.$$

Memory remains the largest module because it has 120 cells; they hold 1200 of its 1465 switches.

Source. `CpuComponentsDrawer.java` defines the counted circuit. `SwitchCountAudit.java` enumerates its gates; `SwitchCount-counts.csv` records each gate's inputs and NOR costs.

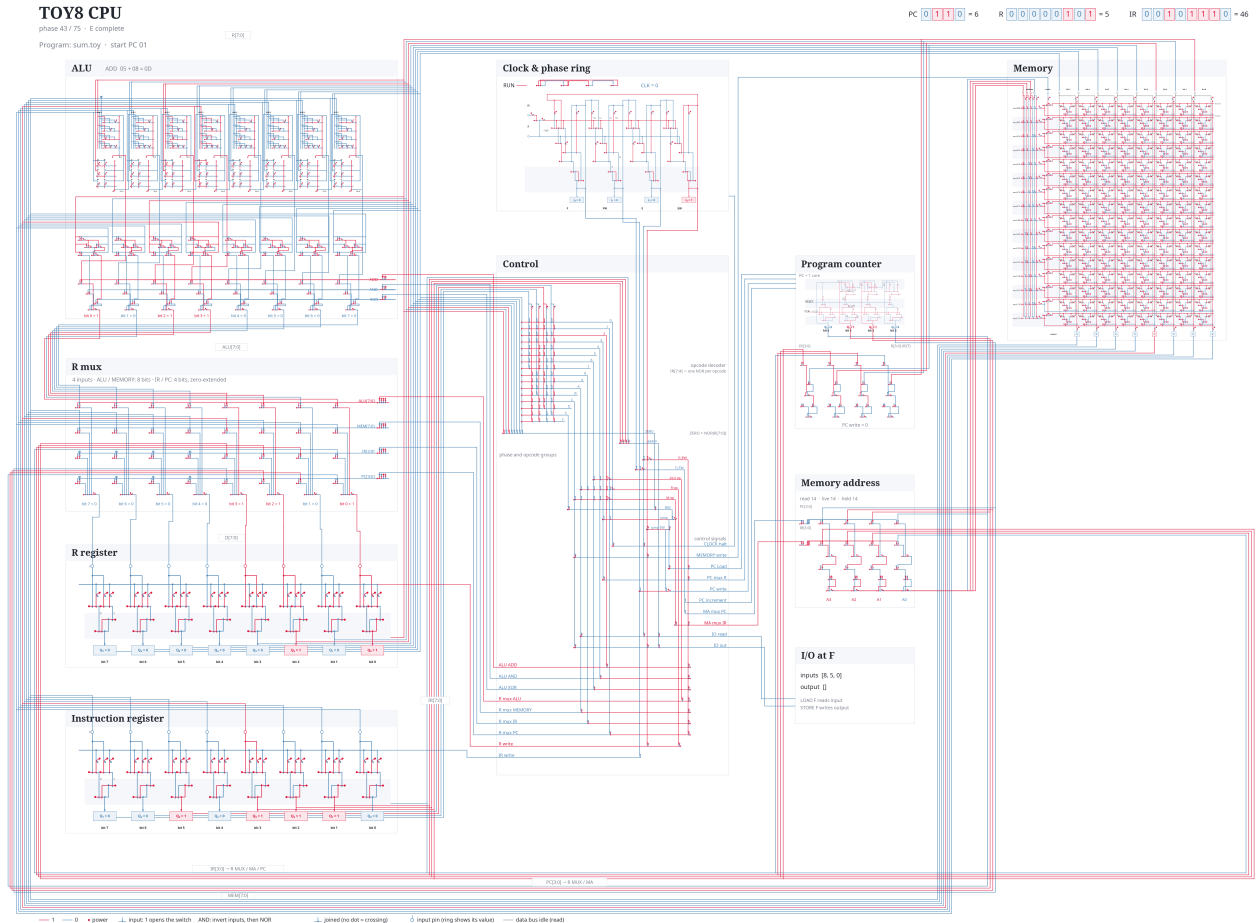


Figure 1. TOY-8 executing ADD E in the summation program, frame 43: $R = 5$, $M[E] = 8$, and the ALU result is 13. The legend is along the bottom edge of the drawing.