Constructing a Sandpile Machine

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Rebecca Schoenberg-Jones

May 2008

Approved for the Division
(Mathematics)

_____

David Perkinson

Acknowledgements

I extend my deepest gratitude to those without whom I would not be here:

My parents, for giving me the space to make my own mistakes and learn from them–even when they knew better all along.

My advisor, David Perkinson, for guiding me through the dual processes of independent investigation and academic collaboration and also for continuing to work with me despite all the hurdles I erected.

My professors, for challenging my reasoning and developing my thought, both in and outside of the classroom.

The staff of Student Services, in particular Sarah Parshley and Lisa Moore, for offering me help before I even knew to ask.

My friends, for listening to my stories and laughing when I needed you to–especially when they weren't particularly funny.

Finally, were it not for the daily encouragement of Jessica Gill, Suzanna Goldblatt, Marianna Mullens, and Amanda Ufheil-Somers, I doubt that I could have come this far.

Table of Contents

List of Figures

Abstract

The Turing machine is an abstraction of what is needed to build a computing machine. Circuits are based in logic but are often applied in the construction of computers. Sandpiles are dynamical systems that can model circuits, as described by Goles and Margenstern [GM96]. The construction of a Turing machine from sandpiles is described herein.

Chapter 1

Machine

First described by Alan Turing in 1936, the Turing machine is a simple-to-visualize symbol-manipulating device. The physical machinery includes a semi-infinite tape divided into cells and a read/write tape head that moves either left or right one cell. Additionally, there is a finite set of symbols; a set of machine states; and a program that determines the next symbol of the cell, the next state of the machine, and whether the tape head moves left or right depending on the current symbol of the cell and state of the machine.

This simple construction is incredibly versatile in its applications. Turing machines can decide whether an input satisfies some condition, e.g. whether a number is even. Alternatively, we can use a Turing machine to mechanize the computation of a desired function; for instance, we can construct a Turing machine that computes the sum of two numbers.

Definition 1. A *Turing machine* is denoted

$$M = (Q, \Gamma, \Sigma, B, \delta, q_1, \text{halt}),$$

where

Q is the finite set of *transition states*,

Γ is the finite set of allowable *tape symbols*,

$\Sigma \subseteq \Gamma$ is the set of *input symbols*,

$B \in \Gamma$ is the *blank symbol*, $B \notin \Sigma$,

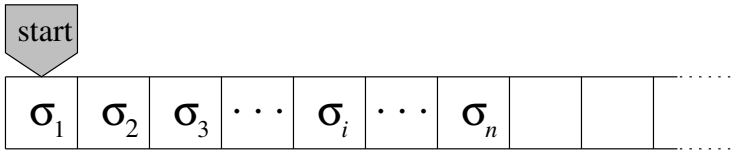

Figure 1.1: An arbitrary Turing machine with an input string $\sigma_1\sigma_2 \ldots \sigma_n$.

δ, the *next move function*, is a mapping:

$$\delta : \mathscr{I} \to \mathscr{O},$$

where $\mathscr{I} \subseteq Q \times \Gamma$ and $\mathscr{O} \subseteq (Q \cup \{\text{halt}\}) \times \Gamma \times \{L, R\}$,

$q_1 \in Q$ is the *initial state*,

halt is a distinguished *final state*, and halt $\notin Q$.

The δ-function describes *transitions* of the machine; if the machine is in a state q and the tape head reads a symbol σ, then the machine transitions to a state q' and the tape head writes a symbol σ' and moves to the next cell (either one to the left or to the right).

Example 2. *Decision machine: Is this number odd?*
Let $i \in \mathbb{N}$. Is i odd? Consider the function: parity : $\mathbb{N} \to \{0, 1\}$, where

$$i \mapsto \begin{cases} \text{halt} & \text{if} \quad i \bmod 2 = 1 \\ \text{fail to halt} & \text{if} \quad i \bmod 2 = 0 \end{cases}$$

Let us consider the image of the parity function to be the boolean set, where we identify "halt" with *true* and "fail to halt" with *false*. Then it is natural to extend this interpretation to the function itself; the parity function decides the truth value of the statement, "The number i is odd."

We can construct a Turing machine, M_{odd}, that decides the parity of the unary representation of i as follows:

$$M_{odd} = (\{\text{even}, \text{odd}, \text{fail}\}, \{1, B\}, \{1\}, B, \delta, \text{even}, \text{halt}),$$

where

$$\begin{aligned} \delta(\text{even}, 1) &= (\text{odd}, B, R) \\ \delta(\text{odd}, 1) &= (\text{even}, B, R) \\ \delta(\text{even}, B) &= (\text{fail}, B, R) \\ \delta(\text{odd}, B) &= (\text{halt}, 1, R) \\ \delta(\text{fail}, B) &= (\text{fail}, B, R). \end{aligned}$$

We can also describe the next-move function, δ, using the state diagram of M_{odd} (see Figure 1.2). In a state diagram, each state is depicted as a circle. Machine transitions are depicted as arrows; if $\delta(q, \sigma) = (q', \sigma', \Delta)$, then there is an arrow from state q to q' labeled "$\sigma \to \sigma', \Delta$". The start state is the distinguished state taking an arrow from nowhere.

For example, M_{odd} starts in the even state and so there is an arrow pointing to the circle labeled "even" that does not originate at another circle. Also, we see that there is an arrow from "odd" to "halt" labeled "$B \to 1, R$", matching the next move function $\delta(\text{odd}, B) = (\text{halt}, 1, R)$.

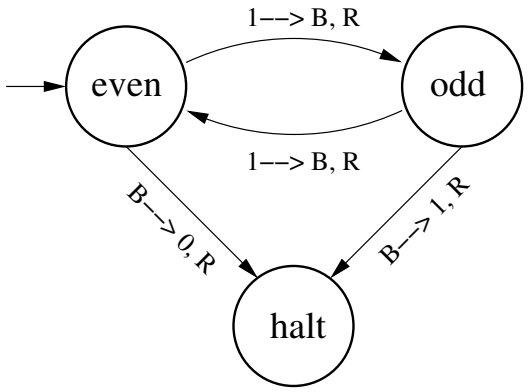

Figure 1.2: The state diagram of the Turing machine M_{odd}

Definition 3. A Turing machine *configuration* is an element

$$(\ell, q, r) \in \Gamma^* \times (Q \cup \text{halt}) \times \Gamma^*,$$

where Γ^* is the set of all *strings*, or finite sequences, of symbols in Γ. The *initial configuration* of the Turing machine is (ε, q_1, r), where ε denotes the *empty string* and $r \in \Sigma^*$.

Let c_1, c_2, and c_n be machine configurations. Then $c_1 \vdash c_2$ denotes the *one-step transition* from c_1 to c_2. Analogously, $c_1 \vdash^* c_n$ denotes the transition from c_1 to c_n in zero or more steps.

From a given configuration (ℓ, p, r), where $\ell = \ell_1\ell_2 \ldots \ell_m \in \Gamma^m$ and $r = r_1r_2 \ldots r_n \in \Gamma^n$, the Turing machine *transitions* to the next configuration as follows:

if $\delta(p, r_1) = (q, \sigma, L)$, then $(\ell, p, r) \vdash (\ell_1\ell_2 \ldots \ell_{m-1}, q, \ell_m\sigma r_2 \ldots r_n)$,

if $\delta(p, r_1) = (q, \sigma, R)$, then $(\ell, p, r) \vdash (\ell_1\ell_2 \ldots \ell_m\sigma, q, r_2r_3 \ldots r_n)$.

For example, let us return to the Turing machine M_{odd} described in Example 2. Consider the transitions of the configurations of M_{odd} given the input string 5=11111:

$$\begin{aligned}(\varepsilon, \text{even}, 11111) &\vdash (\varepsilon, \text{odd}, 1111)\\ &\vdash (\varepsilon, \text{even}, 111)\\ &\vdash (\varepsilon, \text{odd}, 11)\\ &\vdash (\varepsilon, \text{even}, 1)\\ &\vdash (\varepsilon, \text{odd}, \varepsilon)\\ &\vdash (1, \text{halt}, \varepsilon).\end{aligned}$$

Then $(\varepsilon, \text{even}, 11111) \vdash^* (1, \text{halt}, \varepsilon)$ and so we can conclude that 5 is odd. Figure 1.3 depicts the transitions of the physical machine tape and head given the input 11111.

Definition 4. The *language* of a decision Turing machine, $\mathscr{L}(M)$, is the set of all input strings such that the Turing machine transitions into the *final configuration*

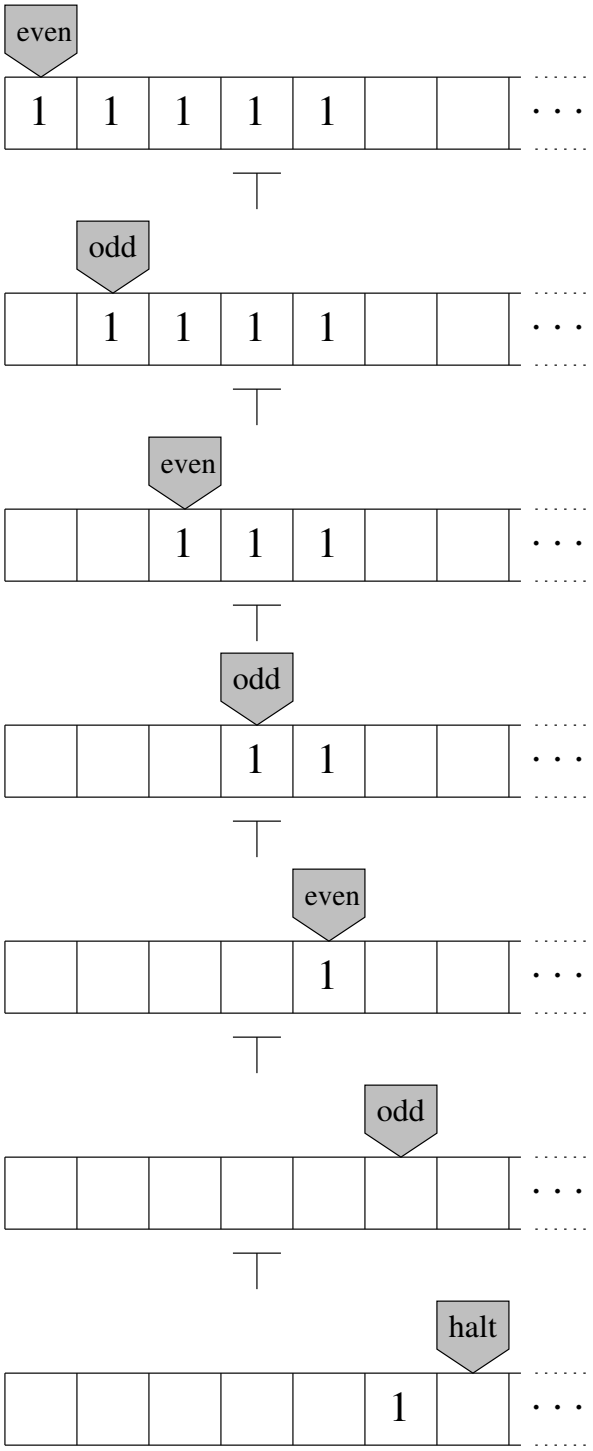

Figure 1.3: Configurations of the parity-decision machine on a string 11111

$(1, \text{halt}, \varepsilon)$, i.e.,

$$\mathscr{L}(M) = \{x \in \Sigma^* | (\varepsilon, q_1, x) \vdash^* (1, \text{halt}, \varepsilon)\}.$$

A string in the language of the machine is called an *accepted string*.

Example 5. *Parity decision* (continued)
We know that M_{odd} accepts 11111, the number equivalent to 5 in unary notation.

Figure 1.4 shows that the machine transitions into the fail state given the input 1111. M_{odd} will not transition out of the fail state (see Figure 1.2) and so it is clear that 1111 is not in the language of the machine.

Earlier we associated the boolean values 1 and 0 with *true* and *false*, respectively. We can also choose to interpret 1 to mean *accepts* and 0 to mean *fails to accept*. Then the parity-deciding machine M_{odd} accepts the string of 1s of length i if i is odd and fails to accept i if i is even, or

$$\mathscr{L}(M_{odd}) = \{1^n \mid n \bmod 2 = 1\}.$$

Let us examine a more complicated example of a decision-type Turing machine.

Example 6. *Decision machine: Is this number a power of 2?*
Let $i \in \mathbb{N}$. Is i a power of 2? Consider a function defined as follows:

$$i \mapsto \begin{cases} 1 & \text{if } i \text{ is a power of 2} \\ 0 & \text{otherwise} \end{cases}$$

Similar to Example 2, let us consider the image of the function to be the boolean set, $\{0, 1\}$. We can then say that the function decides the truth value of the statement, "The number i is a power of 2."

We can construct a Turing machine M_{pow} that decides, given the unary representation of i, if i a power of 2:

$$M_{pow} = (\{q_1, q_2, q_3, q_4, q_5, q_6, q_7, \text{fail}\}, \{0, 1, x, B\}, \{1\}, B, \delta, q_1, \text{halt}),$$

where the next move function is described by the state diagram in Figure 1.5. The language of M_{pow} is the set of all i-length strings of 1s, where i is a power of two, i.e.,

$$\mathscr{L}(M_{pow}) = \{1^n \mid n = 2^k \text{ for } k \in \mathbb{N}\}.$$

For decision-type Turing machines, the output of the machine is fairly uninteresting because every accepted string has the same output, namely 1. However, for function-computing Turing machines the output string is desired because it is the result of the computation for a given input.

Let M be a computation-type Turing machine. Then for an input $x \in \Sigma^*$, M computes $y \in \Gamma^*$, denoted $x \overset{M}{\longmapsto} y$, whenever $(\varepsilon, q_1, x) \vdash^* (y, \text{halt}, \varepsilon)$.

Example 7. *Computation machine: What is the sum of i and j?*
Let $i, j \in \mathbb{N}$. Consider the following Turing machine:

$$M_{sum} = (\{q_1, q_2, q_3, q_4\}, \{0, 1, B\}, B, \delta, \text{sum}\},$$

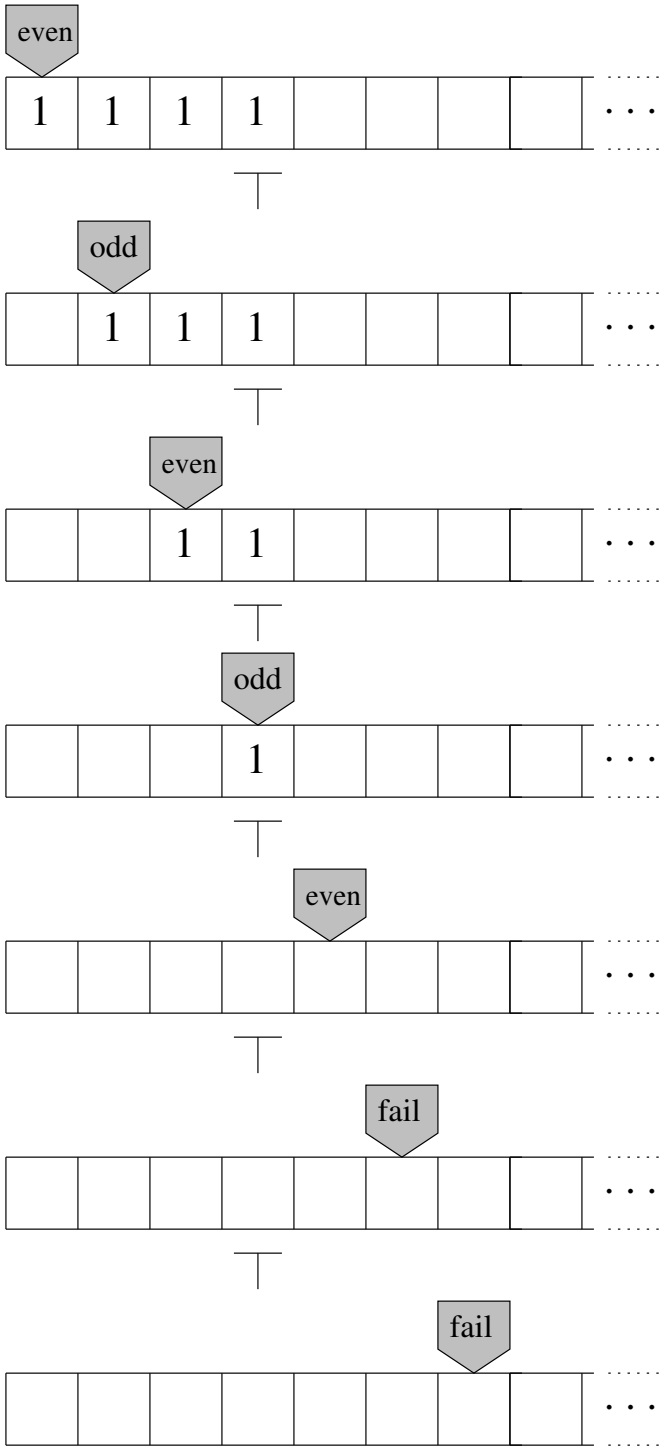

Figure 1.4: Configurations of M_{odd} on a string 1111

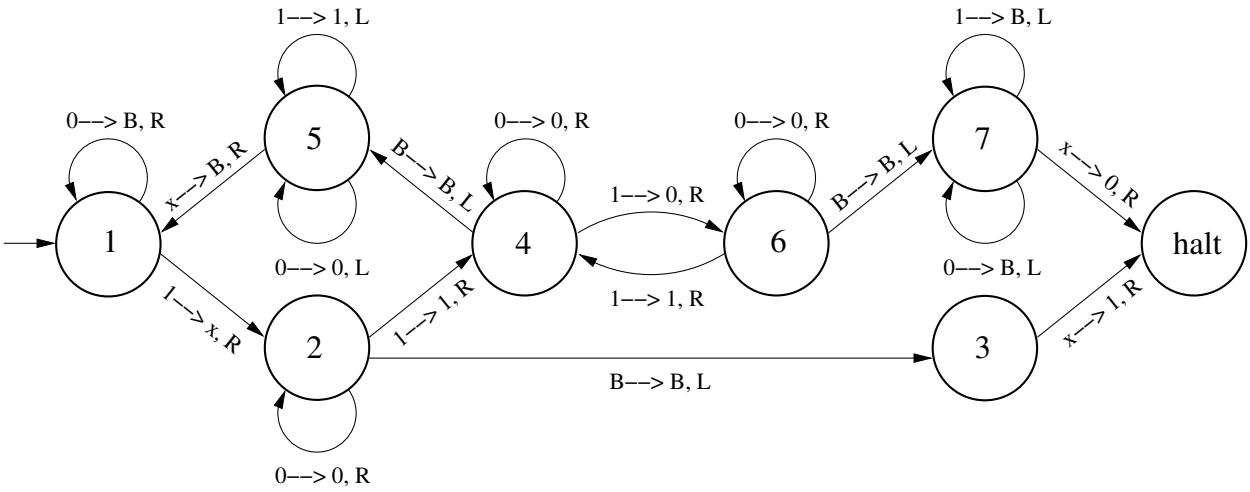

Figure 1.5: The state diagram of the power-of-2 decision Turing machine M_{pow}

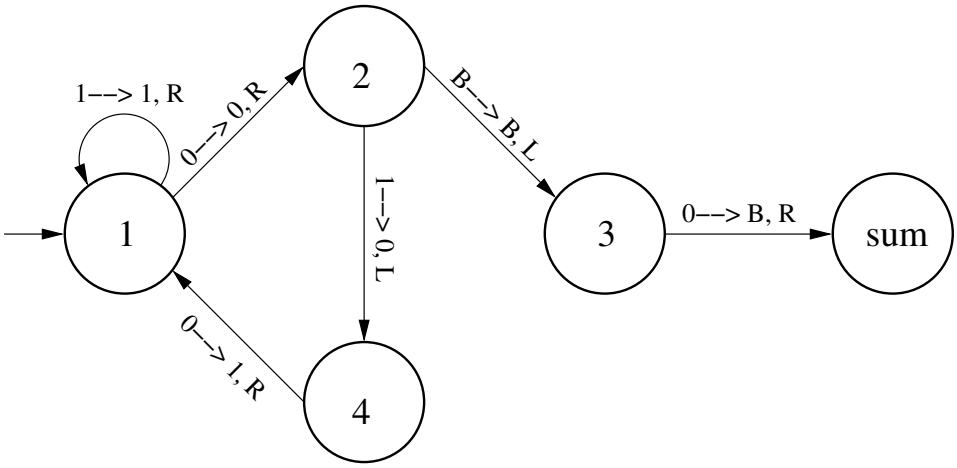

Figure 1.6: The state diagram of the sum-computing Turing machine M_{sum}

where δ is described according to the state diagram depicted in Figure 1.6. Then $1^i01^j \overset{M_{sum}}{\longmapsto} 1^{i+j}$, i.e., M_{sum} computes the sum (in unary notation) of two numbers in unary notation separated by 0.

Let us observe the transitions of M_{sum} given the input string 110111:

$$\begin{aligned}(\varepsilon, q_1, 110111) &\vdash (1, q_1, 10111)\\ &\vdash (11, q_1, 0111)\\ &\vdash (110, q_2, 111)\\ &\vdash (11, q_4, 0011)\\ &\vdash (111, q_1, 011)\\ &\vdash (1110, q_2, 11)\\ &\vdash (111, q_4, 001)\\ &\vdash (1111, q_1, 01)\\ &\vdash (11110, q_2, 1)\\ &\vdash (1111, q_4, 00)\\ &\vdash (11111, q_1, 0)\\ &\vdash (111110, q_2, \varepsilon)\\ &\vdash (11111, q_3, 0)\\ &\vdash (1111, q_4, 1)\\ &\vdash (11111, \text{sum}, \varepsilon)\end{aligned}$$

Then $(\varepsilon, q_1, 110111) \vdash^* (11111, \text{sum}, \varepsilon)$. Figure 1.7 depicts the physical machine tape and head in the computation of the sum of 2 and 3.

Example 8. *Computation machine: What is the product of i and j?*
Let $i, j \in \mathbb{N}$. Consider the following Turing machine:

$$M_{mult} = (\{q_k \mid k \in \{1, 2, \ldots, 16\}\}, \{0, 1, x, y, B\}, \{0, 1\}, B, \delta, q_1, \text{halt}),$$

where δ is defined according to the state diagram in Figure 1.8. M_{mult} computes the product of two unary numbers separated by 0, i.e.,

$$1^i01^j \overset{M_{mult}}{\longmapsto} 1^{i \cdot j}.$$

For a machine with so many states, it becomes tedious to represent the steps of the computation. Instead, we can give a general description of the behavior of M_{mult}:

1. Decrement i.
2. Write a 0 at the end of the input string; move to the right and write j 1s.
3. Repeat steps (1) and (2) $i - 1$ times.
4. Remove the 0s separating the i groups of j ones.

We are left with $i \cdot j$ 1s, the product of i and j in unary notation.

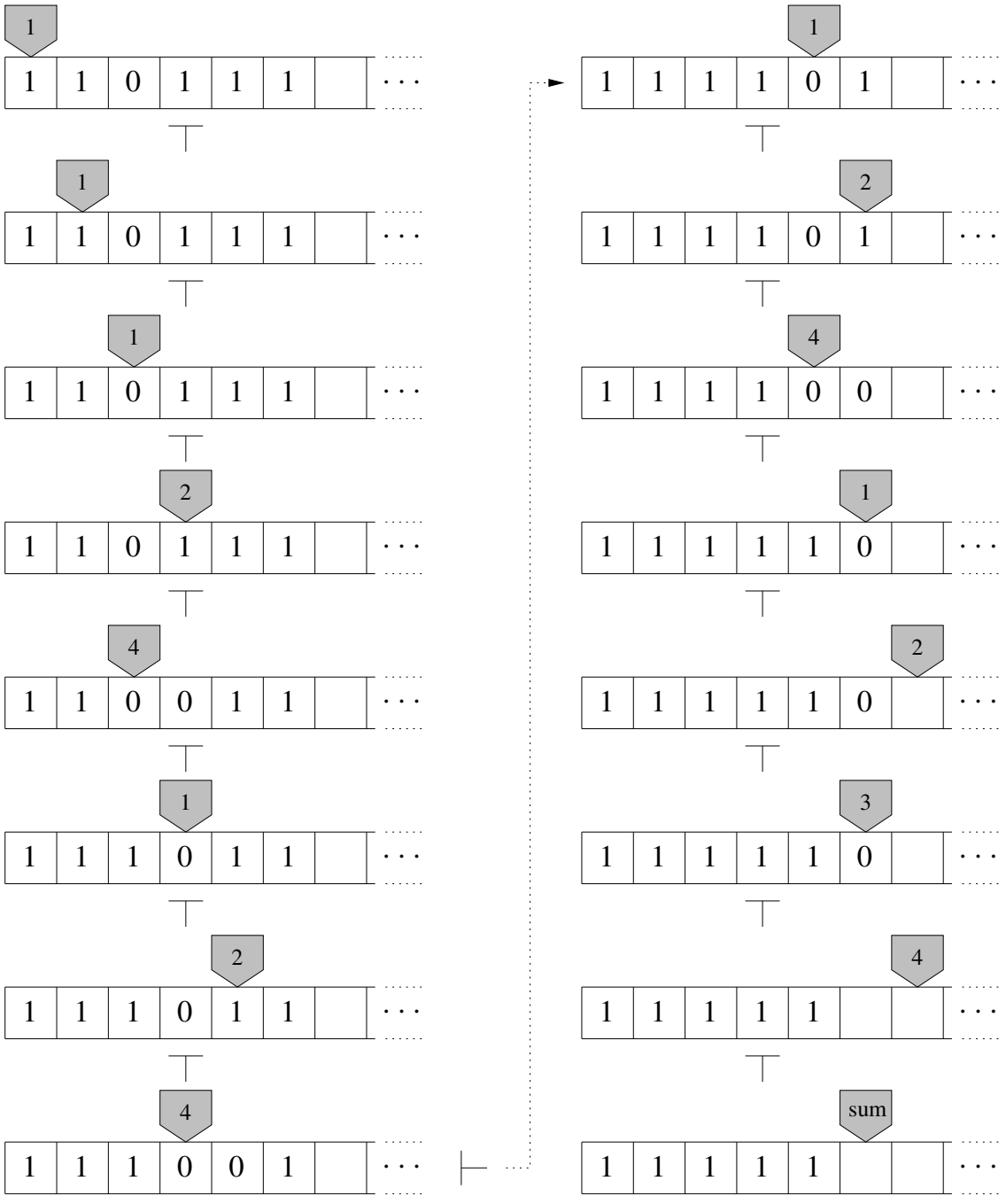

Figure 1.7: Configurations of the Turing machine M_{sum} on a string 110111

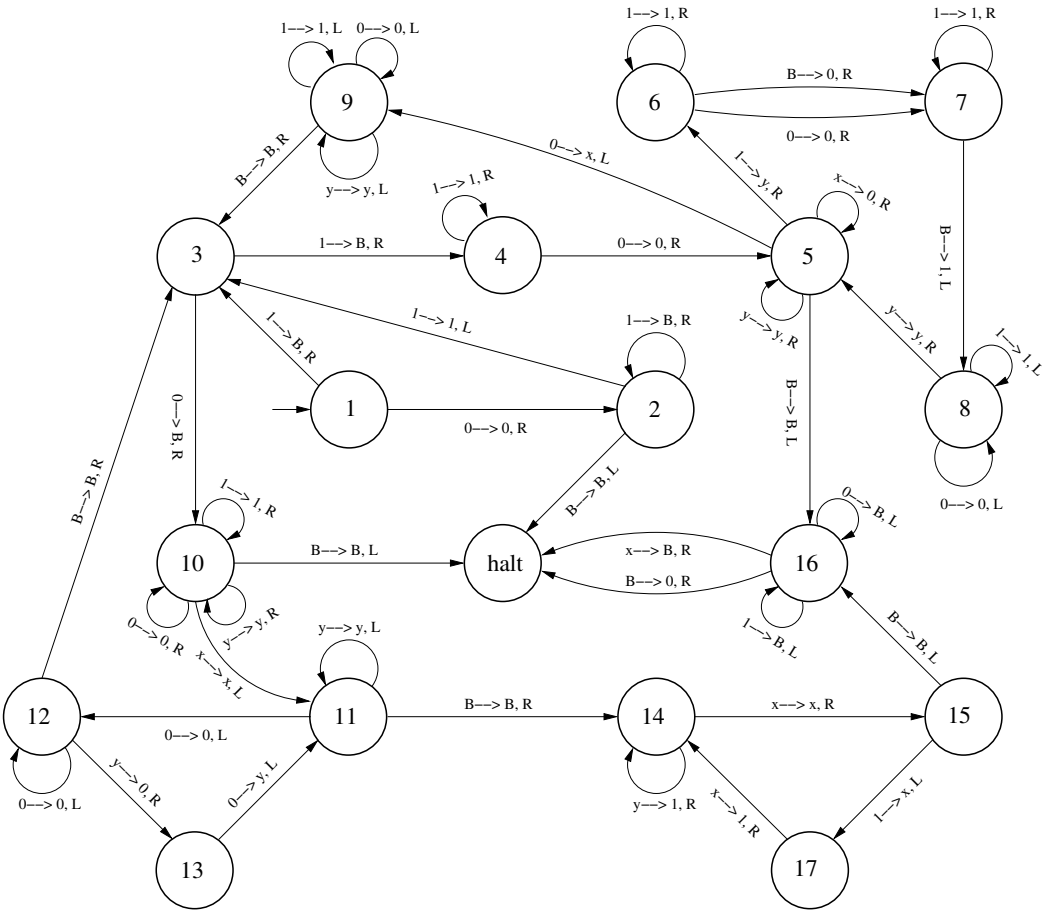

Figure 1.8: The state diagram of the Turing machine M_{mult}

Chapter 2

Circuit

2.1 Boolean operations

We are interested in the *Boolean algebra* over the set $\{0, 1\}$ having the following three operations:

1. *Conjunction*, also expressed as $\wedge$ and AND, is a commutative binary operation that evaluates to 1 only when both arguments are 1:

$$0 \wedge 0 = 0$$
$$0 \wedge 1 = 0$$
$$1 \wedge 0 = 0$$
$$1 \wedge 1 = 1$$

2. *Disjunction* ($\vee$ and OR) is a commutative binary operation that evaluates to 0 only when both arguments are 0:

$$0 \vee 0 = 0$$
$$0 \vee 1 = 1$$
$$1 \vee 0 = 1$$
$$1 \vee 1 = 1$$

3. *Negation* ($\neg$ and NOT) is a unary operation that evaluates the complementary element of the set:

$$\neg 0 = 1$$
$$\neg 1 = 0$$

2.2 Logic gates

A *logic gate* computes a Boolean operation on one or more *inputs*, producing one *output*. Inputs and outputs are *Boolean variables*, i.e., elements of the set $\{0, 1\}$.

Let x and y be Boolean variables. Consider the following three logic gates:

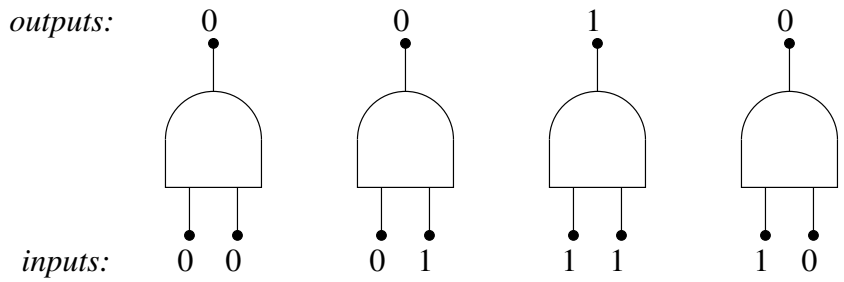

Figure 2.1: The AND gate computes $x \wedge y$

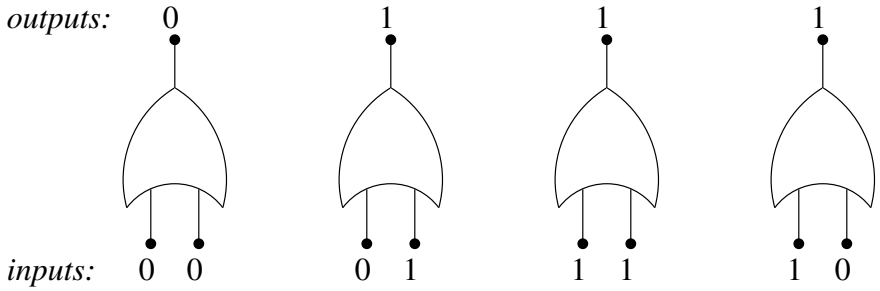

Figure 2.2: The OR gate computes $x \vee y$

1. an AND gate computes $x \wedge y$,
2. an OR gate computes $x \vee y$, and
3. a NOT gate computes $\neg x$.

Figures 2.1 and 2.2 depict the AND and OR gates, respectively, with the inputs $(x, y) = (0, 0), (0, 1)$ and $(1, 1)$. Figure 2.3 depicts the NOT gate with the inputs 0 and 1.

Because the output of a logic gate is a boolean variable, we are able to connect logic gates together, where the output of one logic gate connects to the input of another logic gate. Then it is possible to compute more complicated Boolean

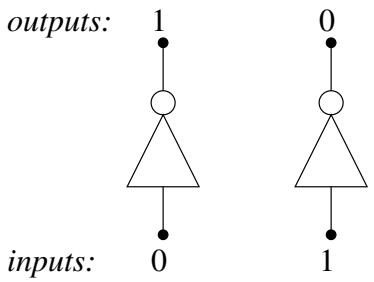

Figure 2.3: The NOT gate computes $\neg x$

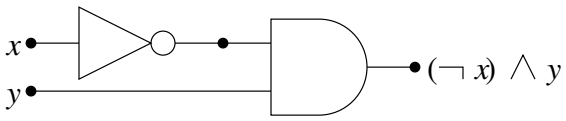

Figure 2.4: Connecting two logic gates to compute $(\neg x) \wedge y$

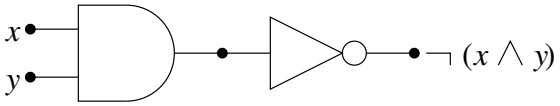

Figure 2.5: Connecting two logic gates to compute $\neg(x \wedge y)$

expressions; for example, given the boolean variables x and y, we can compute $(\neg x) \wedge y$ by connecting the output of a NOT gate to the input of an AND gate (see Figure 2.4). Figure 2.5 shows an alternative connection of a NOT gate and an AND gate that computes $\neg(x \wedge y)$.

We can think of logic gates as *directed graphs*, where the vertices are either Boolean operations or variables and the edges act as wires, carrying the values of the variables to logical gates to be evaluated, forming boolean expressions.

Consider connecting n AND gates (see Figure 2.6). If any of the n input Boolean variables is 0, then the output of the connected AND gates is 0. For example, if $n-1$ inputs are 1 and one input is 0, then:

$$\begin{aligned}(0 \wedge (1 \wedge (\cdots \wedge (1 \wedge (1 \wedge 1)) \cdots))) &= (0 \wedge (1 \wedge (\cdots \wedge (1 \wedge 1) \cdots))) \\ &\vdots \\ &= (0 \wedge 1) \\ &= 0\end{aligned}$$

Then we can define an *n-AND gate* to be the logic gate with a vertex having n in-edges and one out-edge where the value of the output is 1 if and only if all the inputs have value 1. Similarly, an *n-OR gate* is a logic gate having n in-edges and

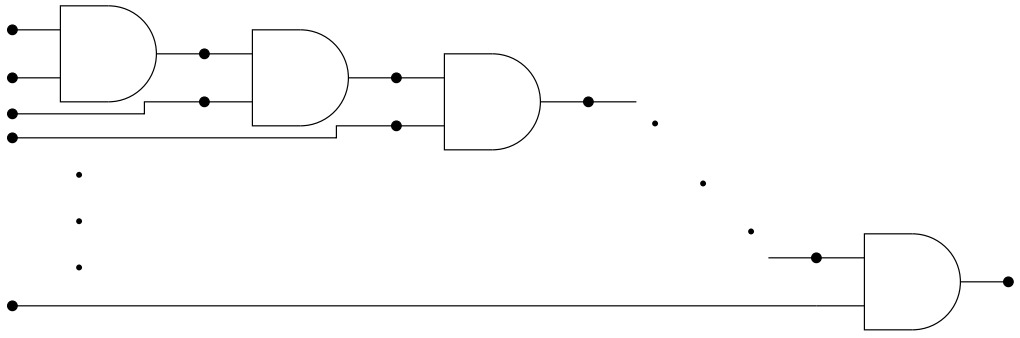

Figure 2.6: A connection of n AND gates

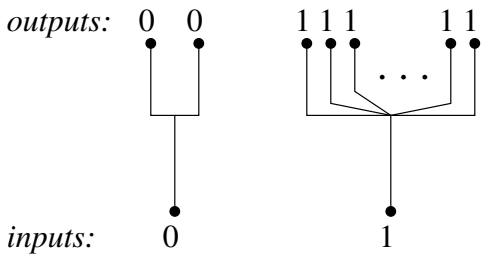

Figure 2.7: A 2-splitter and an n-splitter

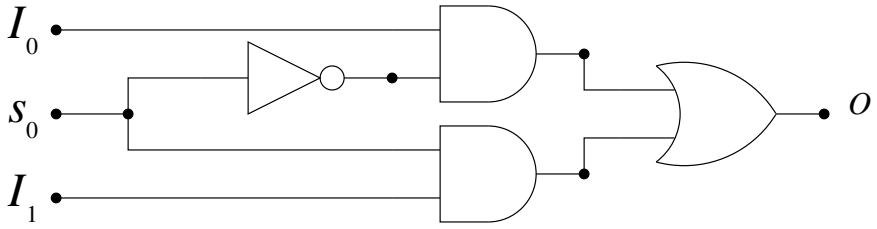

Figure 2.8: A 2:1-multiplexer circuit

one out-edge where the value of the output is 0 if and only if all the inputs have value 0.

An *n-splitter* is a directed graph with a vertex having one in-edge and n out-edges where the value of every output is equal to the value of the input. For example, Figure 2.7 shows that a 2-splitter with a 0-valued input has two 0-valued outputs and an n-splitter with a 1-valued input has n 1-valued outputs.

A *circuit* is any acyclic directed graph composed of some combination of inputs, outputs, logic gates, and splitters.

Example 9. *Multiplexer*
An n:1-*multiplexer* is a circuit having $n+k+1$ inputs, $I_0, I_1, \ldots, I_{n-1}$ and $s_0, s_1, \ldots, s_k$, where $2^k < n \leq 2^{k+1}$, and one output. The s inputs are called *selector bits* because we output the value of I_p, where $\sum_{i=0}^{k} s_i 2^i = p$.

Figure 2.8 shows the construction of a 2:1-multiplexer, a circuit with three inputs, I_0, I_1 and s_1, and an output O. The circuit outputs $O = I_0$ when $s_0 = 0$ and $O = I_1$ when $s_0 = 1$.

Chapter 3

Circuit machine

3.1 Preparation for circuit construction

If we are to emulate a Turing machine with circuits, then we will first need to establish a convention of encoding states, symbols, and the directions left and right. We will also develop an alternative understanding of how a Turing machine can be "built" in such a way that transitions do not rely on a tape head.

3.1.1 Encoding the machine

We will encode symbols as boolean values as follows:

1. Index the set $\Gamma_B = \Gamma \setminus \{B\}$, i.e., $\Gamma_B = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$.
2. Define the set β_Γ of $m + 1$ vectors of boolean values of length m such that there is at most one 1 in any vector.
3. We will identify Γ_B with the subset β_Γ of vectors having exactly one 1 as follows:
$$\sigma_i = (0, 0, 0, \ldots, 0, 1, 0, \ldots, 0),$$
where the i^{th} boolean value of the vector is 1.

   We will identify the blank symbol B with the zero vector in β_Γ:
$$B = (0, 0, \ldots, 0).$$

   Then for every tape symbol there is a corresponding encoding of the symbol as a vector of boolean variables of length m, which we will call the *symbol vector*.

We will encode machine states as boolean values similarly:

1. Index the set of transition states Q, i.e., $Q = \{q_1, q_2, \ldots, q_n\}$.
2. Define a set β_Q of $n + 1$ vectors of boolean values of length n such that there is at most one 1 in any vector.

3. We will identify Q with the subset of vectors of β_Q having exactly one 1 as follows:

$$q_i = (0, 0, 0, \ldots, 0, 1, 0, \ldots, 0),$$

where the i^{th} boolean value in the vector is 1.

We will identify the halt state with the zero vector in β_Q:

$$\text{halt} = (0, 0, \ldots, 0).$$

Then for every state of the machine there is a corresponding encoding of the state as a vector of n boolean variables, which we will call the *state vector*.

To encode the directions the tape head moves as boolean values, we will identify L with 1 and R with 0.

3.1.2 Reinterpreting machine transitions

Let each cell of a Turing machine have both a written symbol $\sigma \in \Gamma$ and a written state $p \in Q \cup \{\text{halt}\}$. At most one cell of the Turing machine is in a transition state, but it is possible that every cell is in the halt state.

Let c be a configuration having a cell with a transition state q and a tape symbol σ such that $\delta(q, \sigma) = (q', \sigma', \Delta)$, where $\Delta \in \{L, R\}$. Then in the next configuration c', the cell is in the halt state and has symbol σ' and the Δ-side neighboring cell is in state q'. If a cell is in the halt state with symbol σ in configuration c, then the cell has symbol σ in the next configuration c'.

3.2 Circuit of cell components

We will begin with a *cell-state vector* and a *cell-symbol vector* that encode the state q and symbol σ of the k^{th} cell in a configuration c, which we denote $v_q(c)(k)$ and $v_\sigma(c)(k)$, respectively. For the moment, our discussion is limited to the configuration c and the k^{th} cell and so we will abridge our notation to v_q and v_σ.

In order to emulate a tape cell of the Turing machine, we will need four main components:

1. a *transition circuit* that decides if the cell is in a transition state,

2. a *δ-circuit* based on the next move function,

3. a *symbol circuit* that computes the symbol of the cell in the next configuration, and

4. a *state circuit* that computes directed state vectors.

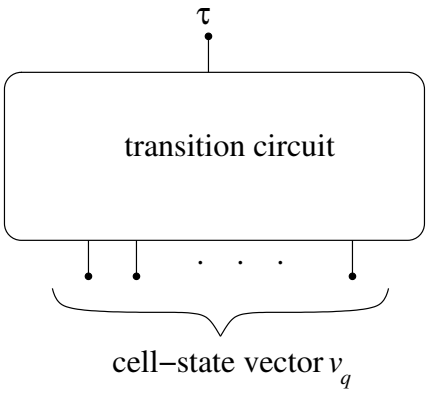

Figure 3.1: The transition circuit for an arbitrary Turing machine

3.2.1 Transition circuit

The transition circuit takes the cell-state vector and produces a single boolean variable output τ. The component computes a 1 if the cell-state vector is non-zero; otherwise the transition circuit computes 0. Then we say either:

- the cell is in a transition state if $\tau = 1$, or
- the cell is in the halt state if $\tau = 0$.

Figure 3.1 depicts the transition circuit for an arbitrary Turing machine.

3.2.2 δ-circuit

The δ-circuit takes the cell-state and -symbol vectors and computes a boolean direction variable d and state and symbol vectors, denoted the *δ-state vector* d_q and the *δ-symbol vector* d_σ, respectively (see Figure 3.2).

If the cell is in a transition state q and reads a symbol σ, then the component computes $(q', \sigma', \Delta) = \delta(q, \sigma)$. If the cell is in the halt state and reads σ, then the component computes (halt, σ, R).

3.2.3 Symbol circuit

The symbol circuit takes as inputs:

- the cell-symbol vector,
- the δ-symbol vector d_σ of the δ-circuit, and
- τ, the boolean variable output of the state circuit.

The component computes the *next cell-symbol vector* v'_σ, i.e., the symbol vector in the configuration c' where $c \vdash c'$. If $\tau = 1$, then the next cell-symbol vector is equal to d_σ; if $\tau = 0$, then the next symbol vector is equal to v_σ. Figure 3.3 depicts the symbol circuit for an arbitrary Turing machine.

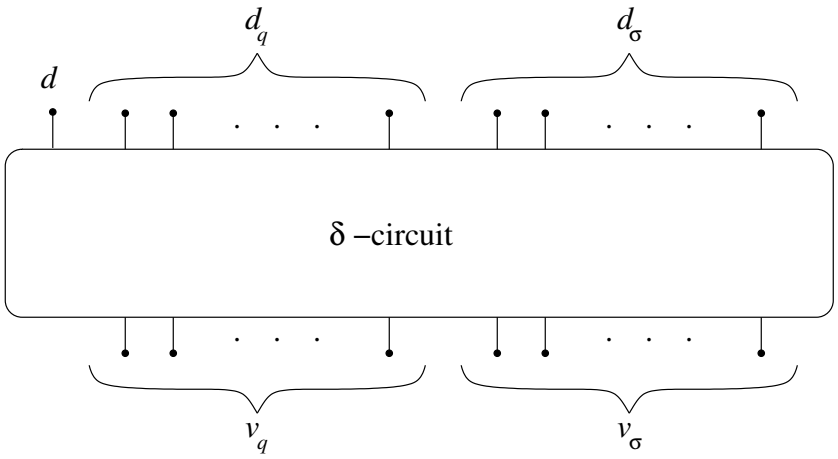

Figure 3.2: The δ-circuit for an arbitrary Turing machine

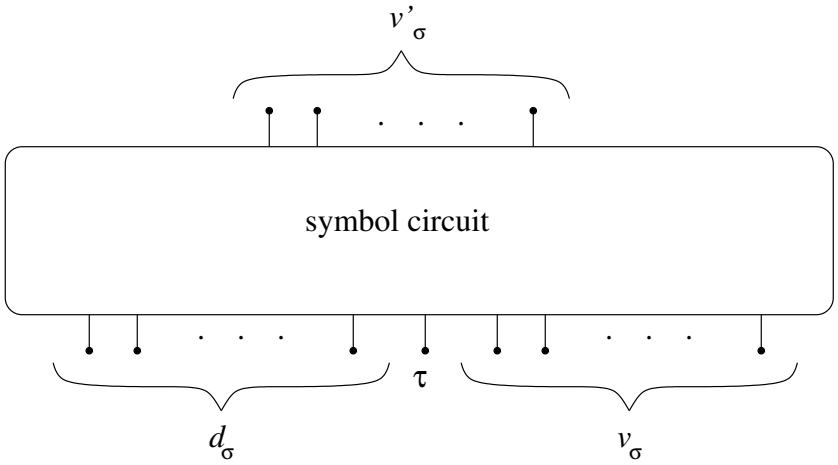

Figure 3.3: The symbol component for an arbitrary Turing machine

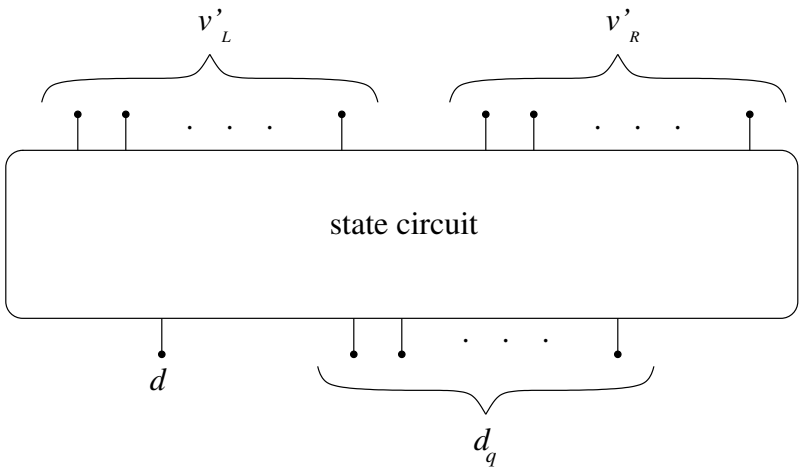

Figure 3.4: The state component for an arbitrary Turing machine

3.2.4 State circuit

The state circuit takes the δ-state vector d_q and the direction variable output, d, of the δ-circuit and computes the *left-state vector* v'_L and the *right-state vector* v'_R.

If d=1, then the component computes the left-state vector equal to the δ-state vector and the right-state vector encodes the halt state. If $d = 0$, the direction-state vectors are switched, i.e., the left-state vector encodes the halt state and the right-state vector equals the δ-state vector.

Figure 3.4 depicts the state circuit for an arbitrary Turing machine.

3.3 The cell components of the Turing Machine M_{odd}

Before we describe the construction of circuit components for arbitrary Turing machines, we will examine the circuit components for the decision-style Turing machine M_{odd}.

- Let q_1 = even, which we will encode as (1,0,0).
- Let q_2 = odd, which we will encode as (0,1,0).
- Let q_3 = fail, which we will encode as (0,0,1).
- The halt state is encoded as (0,0,0).

Let $\gamma_1 = 1$, which we will encode as 1. The blank symbol B is encoded as 0. Figure 3.5 depicts the transition circuit.

Consider the δ-circuit, pictured in Figure 3.6. For example, given state and symbol inputs we can compute the outputs:

- If $v_q = (1, 0, 0)$ and $v_\sigma = 1$, then $d = 0$, $d_q = (0, 1, 0)$, and $d_\sigma = 0$.

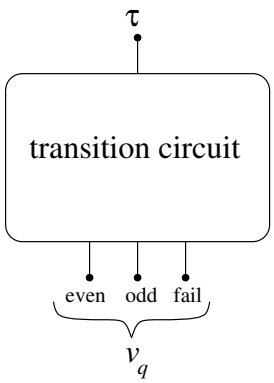

Figure 3.5: The transition circuit for M_{odd}

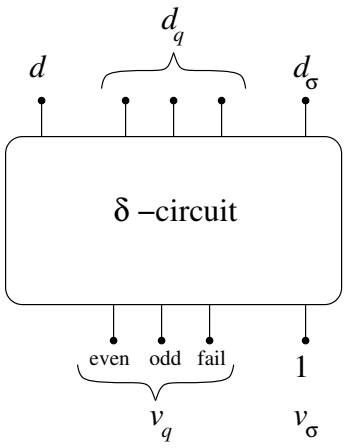

Figure 3.6: The δ-circuit for M_{odd}

- If $v_q = (0, 1, 0)$ and $v_\sigma = 1$, then $d = 0$, $d_q = (1, 0, 0)$, and $d_\sigma = 0$.
- If $v_q = (0, 0, 0)$ and $v_\sigma = 1$, then $d = 0$, $d_q = (0, 0, 0)$, and $d_\sigma = 1$.

Recall the state diagram of M_{odd} (Figure 1.2); the tape head of the machine moves right in every transition and so the circuit ouputs $d = 0$ when a cell is in a transition state. The general description of a δ-circuit requires that $d = 0$ if the cell is in the halt state. Therefore, the δ-circuit for the cell of the Turing machine M_{odd} always outputs $d = 0$.

Consider the symbol circuit. Given sample inputs d_σ, τ, and v_σ, we can compute the output v'_σ:

- If $d_\sigma = 0$, $\tau = 1$, and $v_\sigma = 1$, then $v'_\sigma = 0$.
- If $d_\sigma = 0$, $\tau = 0$, and $v_\sigma = 1$, then $v'_\sigma = 1$.

Figure 3.7 depicts the symbol circuit.

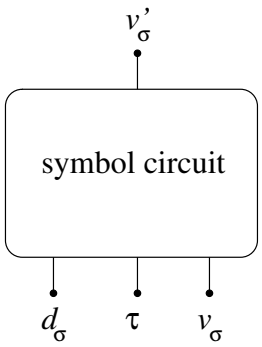

Figure 3.7: The symbol circuit for M_{odd}

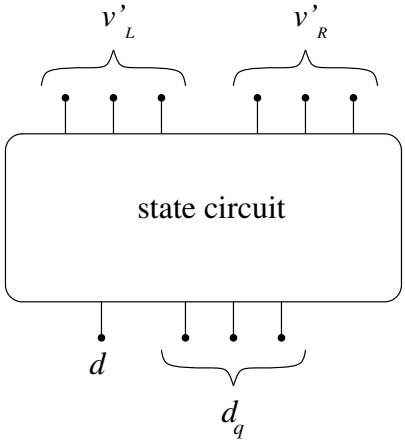

Figure 3.8: The state circuit for M_{odd}

Consider the state circuit for M_{odd}, depicted in Figure 3.8. The left-state vector v'_L always encodes the halt state because the δ-circuit always outputs $d = 0$, for example:

- If $d_q = (1, 0, 0)$, $d = 0$, then $v'_L = (0, 0, 0)$ and $v'_R = (1, 0, 0)$.
- If $d_q = (0, 0, 0)$, $d = 0$, then $v'_L = (0, 0, 0)$ and $v'_R = (0, 0, 0)$.

We can then wire the components together to build the cell circuit; given the state q and a symbol σ of a cell in a configuration c, the circuit will compute left- and right-state vectors for the neighboring cells and the symbol of the cell in the next configuration (see Figure 3.9).

3.4 Component construction for the arbitrary machine

Let us consider the physical representation of an arbitrary Turing machine, M (see Figure 3.10), and let $c = (\ell, q, r)$ be a machine configuration where $\ell = (\ell_1 \ell_2 \ldots \ell_{k-1})$,

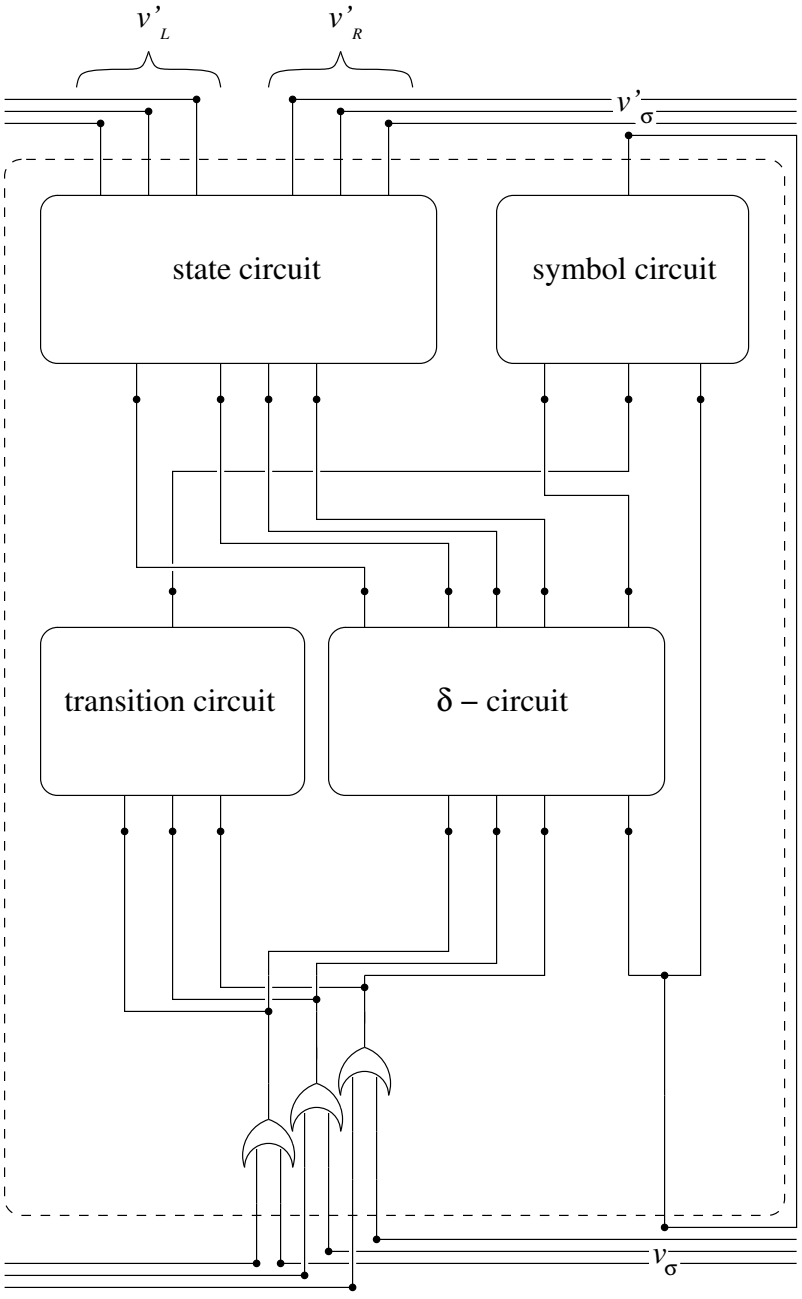

Figure 3.9: The cell circuit for M_{odd}

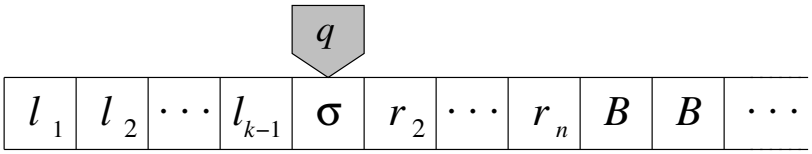

Figure 3.10: The machine tape head at the k^{th} cell

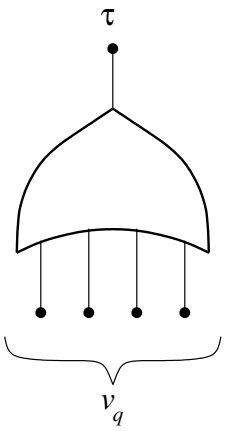

Figure 3.11: Transition circuit for M_{sum}

which is to say that the tape head is located at the k^{th} cell of the tape. To better illustrate the general circuit construction, we include component constructions for the particular Turing machine, M_{sum}.

3.4.1 Transition circuit

The transition circuit is an n-OR gate on the cell-state vector v_q. We connect a wire from each of the n boolean variable inputs composing v_q to the n-OR gate.

The OR gate having a non-zero input computes 1, but if every input is 0 then the OR gate computes 0.

Thus if we have a non-zero cell-state vector, the circuit component computes 1, i.e., the cell is in a transition state, and if the cell-state vector encodes the halt state, the component computes 0. Figure 3.11 depicts the transition circuit for M_{sum}.

3.4.2 δ-circuit

Processing inputs

First we build a circuit that indicates whether the tape cell has the blank symbol. Basing our construction on the transition circuit, if the symbol vector is non-zero then the cell has a non-blank symbol. Lay out an m-OR gate connected to the symbol vector input. The OR gate computes 0 if the cell reads the blank symbol. Connect the input of a NOT gate to the output of the OR gate.

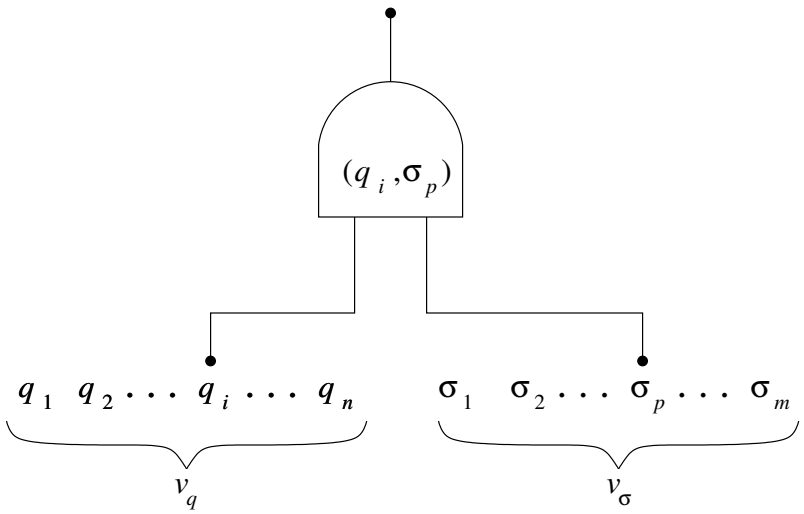

Figure 3.12: The input for $(q_i, \sigma_p) \in \mathscr{I} \setminus \mathscr{I}_0$

This subcomponent, composed of a m-OR gate and a NOT gate, computes 1 when the cell has the blank symbol and 0 otherwise. For the construction of the δ-circuit only, we will consider the *symbol vector* to be a vector of length $m + 1$ having exactly one 1 as follows:

- for $1 \leq i \leq m$, $\sigma_i = (0, 0, \ldots, 0, 1, 0, \ldots, 0)$ where the i^{th} boolean value of the vector is 1, and
- $B = (0, \ldots, 0, 1)$.

Inputs

We want subcomponents that determine whether the current state and symbol of the cell (q_i, σ_p) should lead to the direction, state, and symbol (q_j, σ_u, Δ).

Recall that δ is defined over $\mathscr{I}$, a set of transition state-and-symbol pairs. Define a set $\mathscr{I}_0 = \{(q, \sigma) \in \mathscr{I} \mid \delta(q, \sigma) = (\text{halt}, B, R)\} \subseteq \mathscr{I}$.

Lay out a 2-AND gate for each state and symbol pair $(q_i, \sigma_p) \in \mathscr{I} \setminus \mathscr{I}_0$. Connect wires from the i^{th} boolean variable of the state vector and the p^{th} boolean variable of the symbol vector to the AND gate, which we will call the (q_i, σ_p)-AND gate (see Figure 3.12).

Outputs

Let $k = |\{(q, \sigma) \in \mathscr{I} \setminus \mathscr{I}_0 \mid \delta(q, \sigma) = (q_i, \sigma', \Delta)\}|$. We lay out a k-OR gate (the 1-OR gate simply outputs the input variable) with its output wire connected to the i^{th} boolean variable of the δ-state vector. Similarly, let $r = |\{(q, \sigma) \in \mathscr{I} \setminus \mathscr{I}_0 \mid \delta(q, \sigma) = (q', \sigma_p, \Delta)\}|$. We lay out an r-OR gate with its output wire connected to the j^{th} boolean variable of the δ-symbol vector. We lay out an s-OR gate connected to the direction variable, d, where $s = |\{(q, \sigma) \in \mathscr{I} \setminus \mathscr{I}_0 \mid \delta(q, \sigma) = (q', \sigma', L)\}|$.

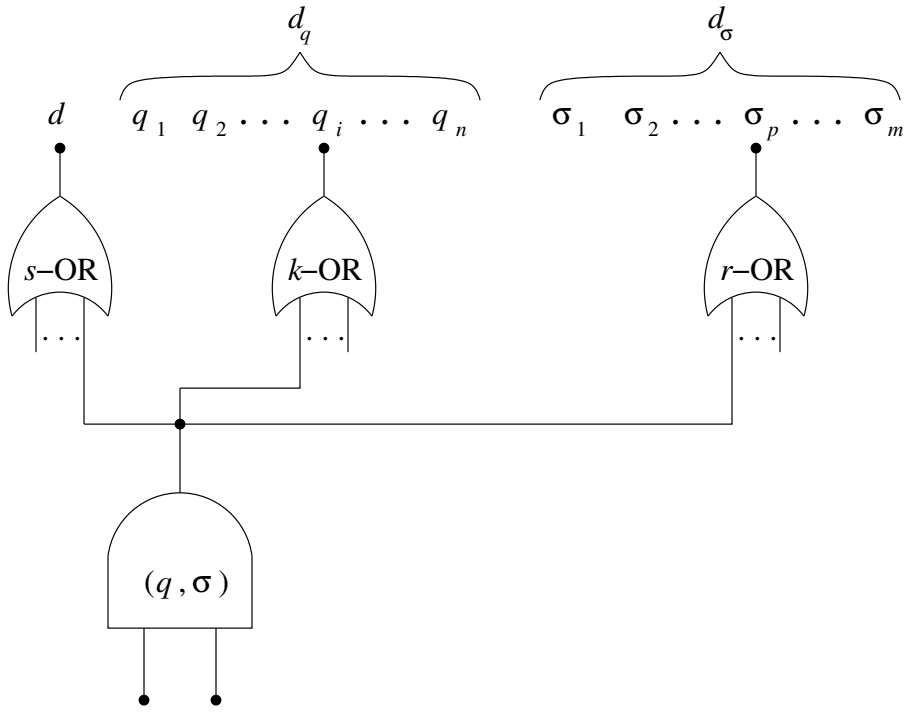

Figure 3.13: An output where $\delta(q, \sigma) = (q_i, \sigma_p, L)$

For each i, p, if $\delta(q, \sigma) = (q_i, \sigma_p, \Delta)$, we connect one wire from the (q, σ)-AND gate to the k-OR gate and another wire from the (q, σ)-AND gate to the r-OR gate. Furthermore, if $\Delta = L$, we also connect a wire from the (q, σ)-AND gate to the s-OR gate. This output is depicted in Figure 3.13.

Figure 3.14 depicts the δ-circuit for M_{sum}.

3.4.3 Symbol circuit

The symbol circuit has m 2:1 multiplexers. The i^{th} multiplexer has the selector bit τ, computed in the transition circuit, and I_0 is the i^{th} variable of the symbol vector and I_1 is the i^{th} variable of the δ-symbol vector.

Therefore, if $\tau = 1$, the symbol circuit computes the next symbol vector equal to the δ-symbol vector and if $\tau = 0$, the symbol circuit computes the next symbol vector equal to the symbol vector.

Figure 3.15 depicts the symbol circuit for M_{sum}.

3.4.4 State circuit

If the direction variable $d = 1$, then the state circuit computes the left-state vector equal to the δ-state vector and the right-state vector equal to the zero state vector. If $d = 0$, then the state circuit computes the right-state vector equal to the δ-state vector and and the left-state vector equal to the zero state vector.

Figure 3.16 depicts the state circuit for M_{sum}.

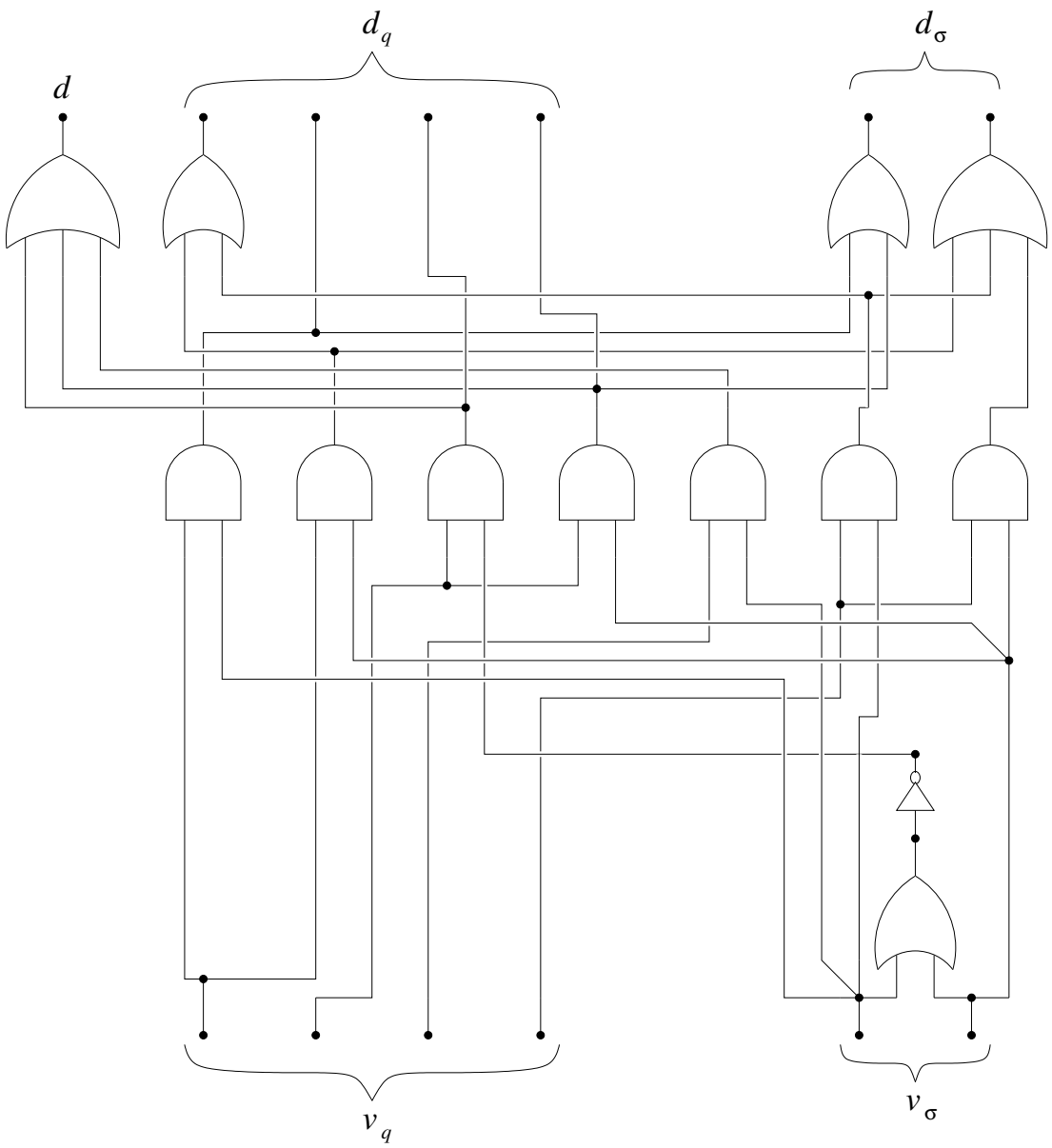

Figure 3.14: The δ-circuit for M_{sum}

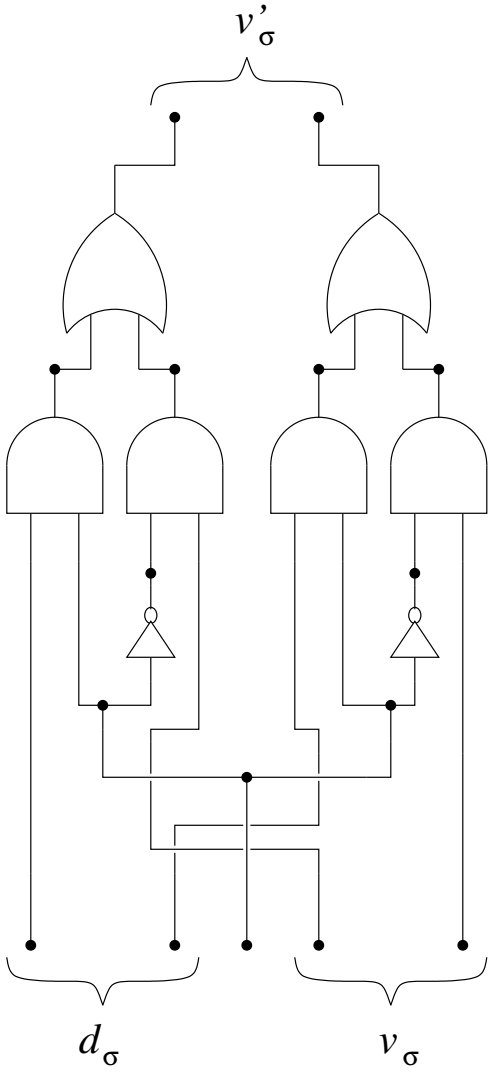

Figure 3.15: Symbol circuit for M_{sum}

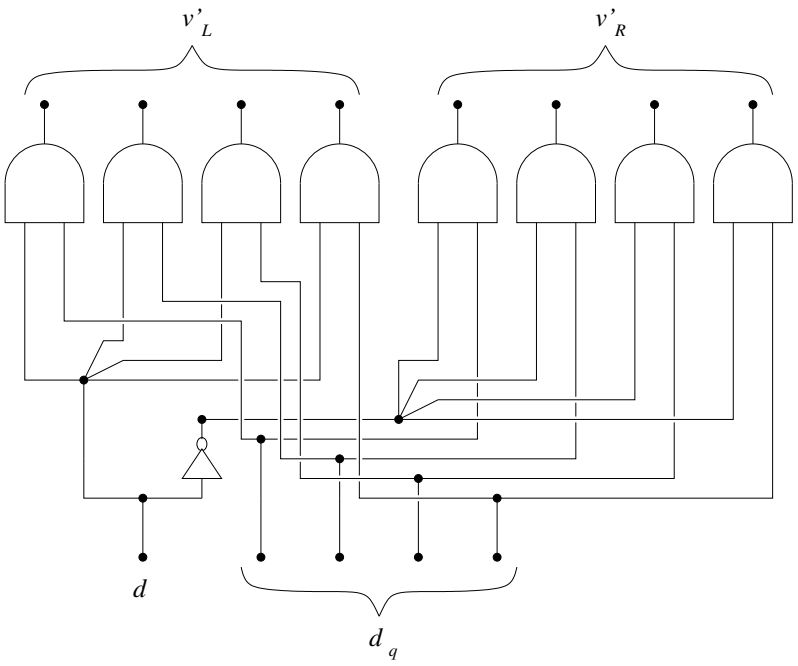

Figure 3.16: State circuit for M_{sum}

Left-state vector

Lay out n AND gates. The i^{th} AND gate has a wire from the direction variable and a wire from the i^{th} variable of the δ-state vector. Then if $d = 1$ the AND gates compute the values of the δ-state vector and if $d = 0$ the AND gates compute all 0s.

Right-state vector

Connect a NOT gate to the direction variable d. Lay out n AND gates. The i^{th} AND gate has a wire from the negated direction variable $\neg d$ and a wire from the i^{th} variable of the δ-state vector. Then if $d = 1$ each AND gate computes 0 because $\neg d = 0$, and if $d = 0$ the AND gates compute the values of the δ-state vector.

3.5 Connecting the circuits

By this construction, in the next configuration each cell will input "state of the cell" information from its neighbors to the left and right. Only one tape cell is transitioning at a time and so at most one cell of the machine will receive a non-zero state value; every other left and right state variable vector encode the halt state. Then we build a subcomponent that computes the state vector of the cell.

Lay out m OR gates; the i^{th} OR gate has a wire from the i^{th} variable of the left-state vector and a wire from the i^{th} variable of the right-state vector. This component computes the state vector of the cell.

The cell-symbol vector in the next configuration is computed by the symbol

circuit component, i.e., $v_\sigma(c')(k) = v'_\sigma(c)(k)$. Then we need only connect the i^{th} variable of the next cell-symbol vector v'_σ to the i^{th} boolean variable of the cell-symbol vector v_σ.

We have thus described a method by which we may construct an arbitrary Turing machine using circuits.

Figure 3.17 is a component representation of a cell of M_{odd} among neighboring cells and Figure 3.18 illustrates the circuitry of a machine cell of M_{sum} also in the context of its neighbors.

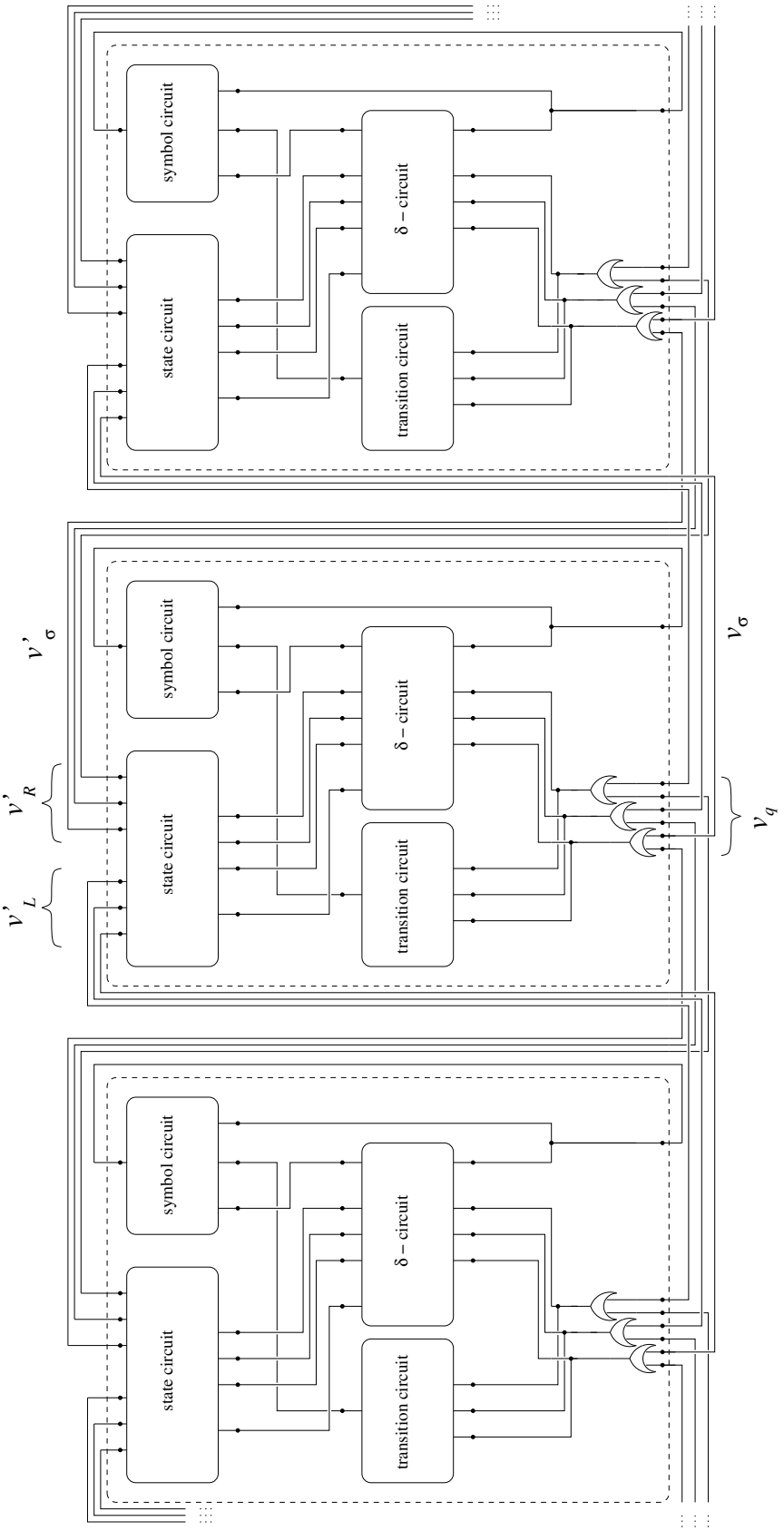

Figure 3.17: A cell of the machine M_{odd} presented as circuit components

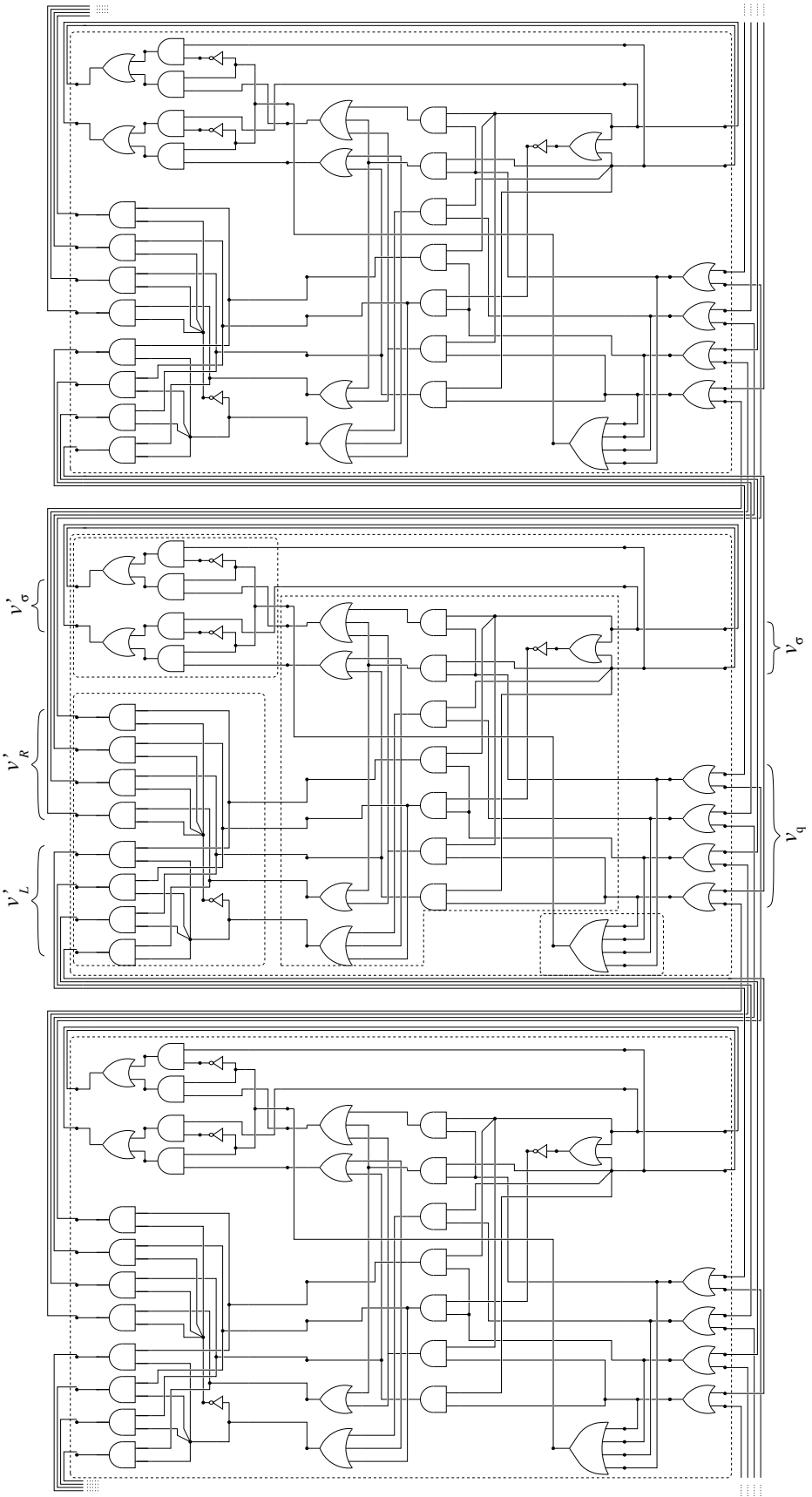

Figure 3.18: A cell circuit of the machine, M_{sum}

Chapter 4

Sand

Sandpiles as models of self-organized criticality were first studied by Bak, Tang, and Wiesenfeld in their attempts to understand chaos underlying organized systems [BTW88]. As we add grains of sand to a sandpile, we observe that at a certain point, the *critical point*, grains of sand cascade down the slope until the pile stabilizes. The pile repeats this behavior; as we add sand, the slope of the pile increases until we reach the critical point, when adding any additional sand causes the pile to cascade. There has been a lot of recent study of the sandpile model (e.g. [HLM$^+$08] and [Lev07]).

4.1 Sandpile

Let $G = (V, E)$ be a *graph*. The vertex set is V and the edge set is $E \subseteq (V \times V)$. We will assume that G has no loops, i.e., $(v, v) \notin E$ for any $v \in V$. The *degree* of a vertex is the number of edges associated with a vertex, i.e., for a vertex $v \in V$,

$$\mathrm{d}(v) = \sum_{w \in V} I(v, w),$$

where I is the following *indicator function*:

$$I(v, w) = \begin{cases} 1 & \text{if } (v, w) \in E \\ 0 & \text{otherwise} \end{cases}.$$

A vertex v is a *sink* if $\mathrm{d}(v) = 0$.

A *sandpile configuration* c of the graph G is an element of the set of mappings from V to $\mathbb{Z}$, i.e., $c \in \mathbb{Z}^V$.

Intuitively, we can think of a configuration c as labeling each vertex v with the number of grains of sand $c(v)$ being held by the vertex. We say a configuration is *unstable* if for some $v \in V$, $c(v) \geq \mathrm{d}(v)$. We define a function reporting whether a vertex in a given configuration c is unstable:

$$\mathrm{unstable}(v, c) = \begin{cases} 1 & \text{if } c(v) \geq \mathrm{d}(v) \\ 0 & \text{otherwise} \end{cases}.$$

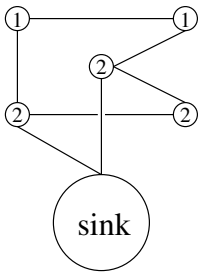

Figure 4.1: A stable sandpile configuration for a graph

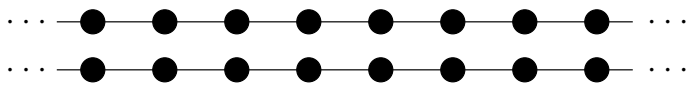

Figure 4.2: An infinite wire

We define the next configuration $c' = next(c)$ of the graph:

$$\text{next}(c)(v) = c(v) - \text{d}(v)\text{unstable}(v, c) + \sum_{(v,w)\in E} \text{unstable}(w, c).$$

Thus, to get the next sandpile configuration, each unstable vertex *fires*, delivering one grain of sand to each of its neighboring vertices. The *sink* of the graph is only stable without sand, but since there are no out-edges the sink never fires. Consequently, we often distinguish the sink from the other vertices and do not include the sink in the configurtion of the graph.

4.2 Sandpile circuit

We will now describe how to build circuits out of sandpiles, as presented by Goles and Margenstern [GM96].

Let us consider a graph composed of vertices arranged in two parallel lines, with an edge between horizontal neighbors, as in Figure 4.2. We will call this graph a *wire*. The *neutral wire* is the stable wire configuration where each vertex holds one grain of sand (see Figure 4.3).

Consider the configuration of an infinite line of vertices having a pair of vertices holding 0 and 2 grains of sand (from left to right) amid vertices holding 1 grain of sand (see Figure 4.4). The vertex with 2 grains of sand is unstable and fires,

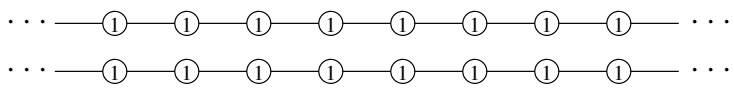

Figure 4.3: A neutral wire

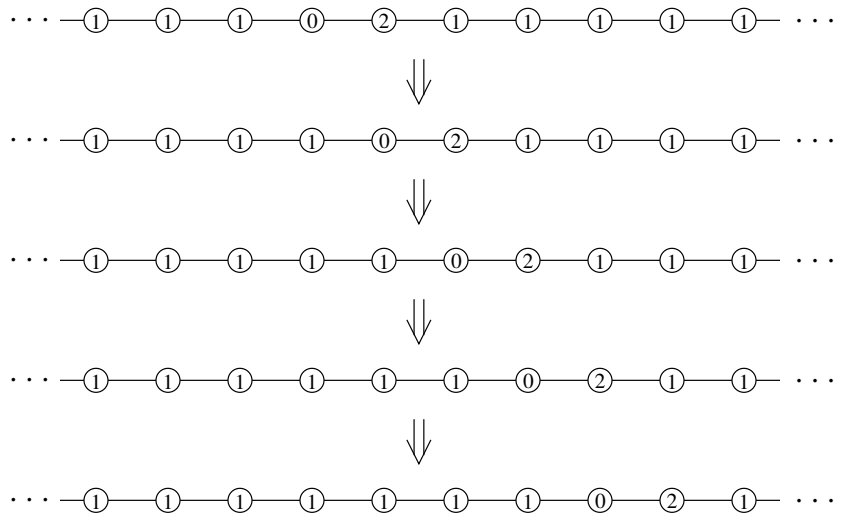

Figure 4.4: The signal is directed along the length of the wire due to the unstable vertex firings

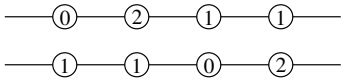

Figure 4.5: A 0-valued wire

translating the sand one vertex to the right; similarly the 02 will travel down the line of vertices amid vertices having 1 grain sand. We will interpret this 02 as a signal along the line of vertices and we can exploit this signal to ascribe a boolean value to a wire in a subgraph configuration by staggering 02 signals in the top and bottom lines of vertices.

A wire, as in Figure 4.5, with a top signal two vertices ahead of a bottom signal is said to be carrying the boolean value 0. A wire, as in Figure 4.6, with a bottom signal two vertices ahead of a top signal is said to be carrying the boolean value 1. Then the 0-valued wire is simply the 1-valued wire with the lines of vertices inverted.

Sandpile gates

As we noted, the wires carrying 0 and 1 are inversions of one another and so the construction of a NOT-gate is straightforward; we will invert the lines of vertices to

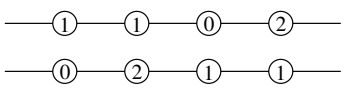

Figure 4.6: A 1-valued wire

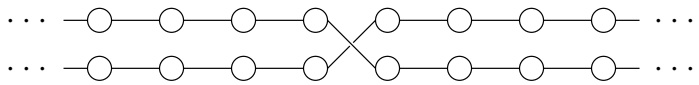

Figure 4.7: The NOT-gate graph

reverse the value of the wire (see Figure 4.7).

Intuitively, we can think of the sandpile AND- and OR-gates as composed of vertices that either delay or send signals forward without a delay. The vertex with 3 out-edges and 1 grain of sand will only fire after two of its neighboring vertices have fired but the vertex with 3 out-edges and 2 grains of sand fires when any of its neighboring vertices fires (see Figures 4.8 and 4.9). The AND-gate has a delaying vertex above and a not-delaying vertex below; then the top 02 "signal" is sent ahead of the bottom 02 "signal" (encoding 1) only when both input wires carry 1, otherwise the bottom signal is propagated first because it is not delayed. The OR-gate has a non-delaying vertex above and a delaying vertex below; the bottom signal is sent ahead (encoding 0) only when both input wires carry 0, otherwise the top signal is propagated first.

An AND-gate takes two input wires, say X and Y, and outputs one wire carrying the truth value of $X \wedge Y$. Figure 4.10 depicts the construction of the sandpile AND-gate.

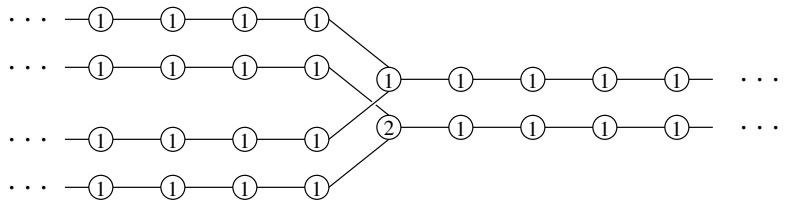

Figure 4.10: The sandpile AND-gate with neutral wire inputs and output

As we might expect, an OR-gate is an inverted AND-gate (see Figure 4.11).

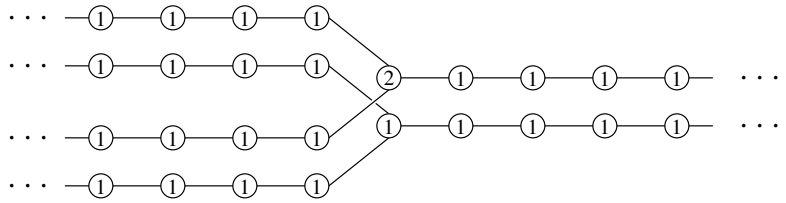

Figure 4.11: The sandpile OR-gate with neutral wire inputs and output

We can verify that the sandpile gates really do output the truth value of the input expression, i.e., $1 \wedge 1 = 1$ or $1 \wedge 0 = 0$ (see Figures 4.12 and 4.13, respectively).

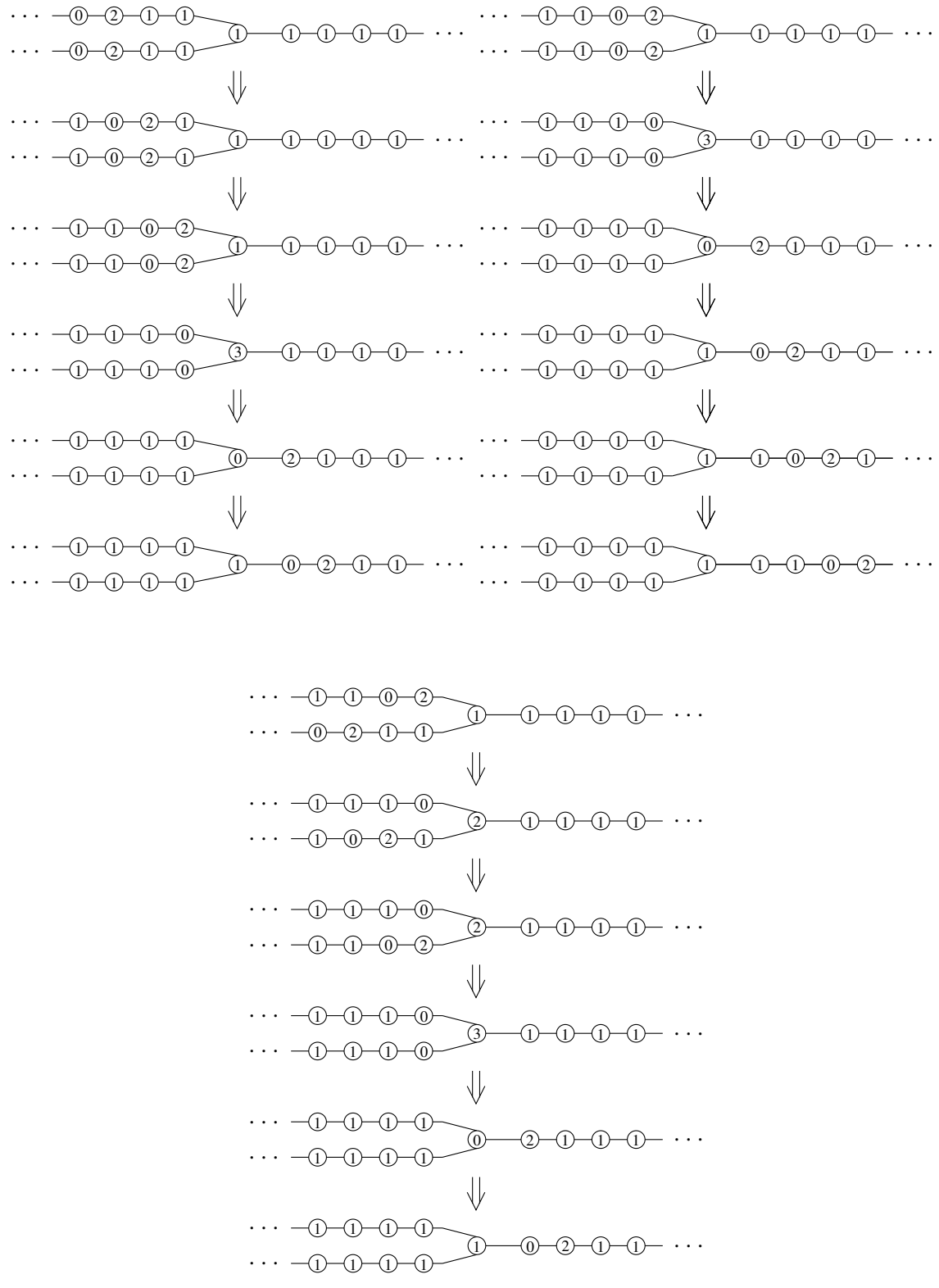

Figure 4.8: The delaying vertex in action

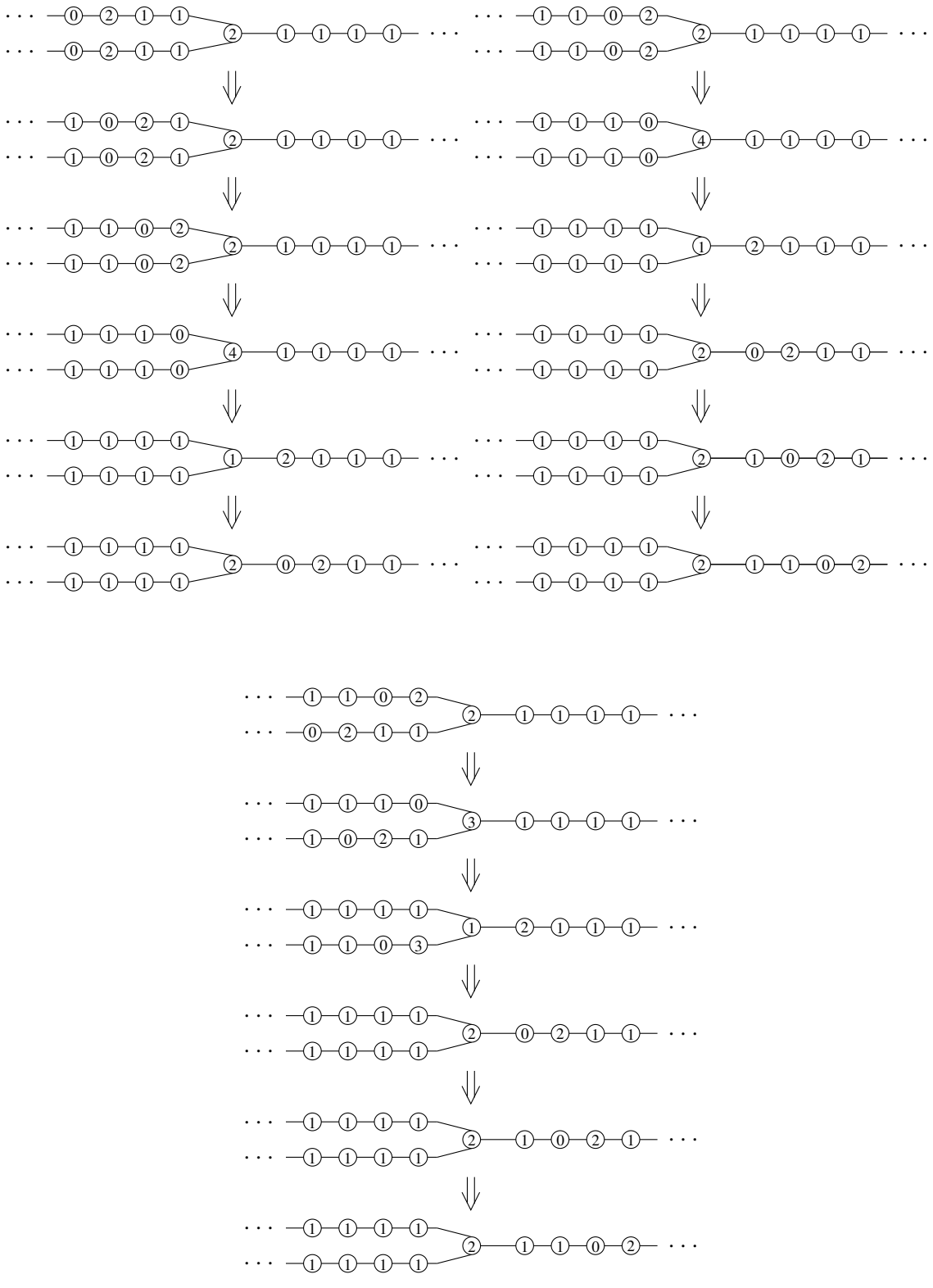

Figure 4.9: The non-delaying vertex in action

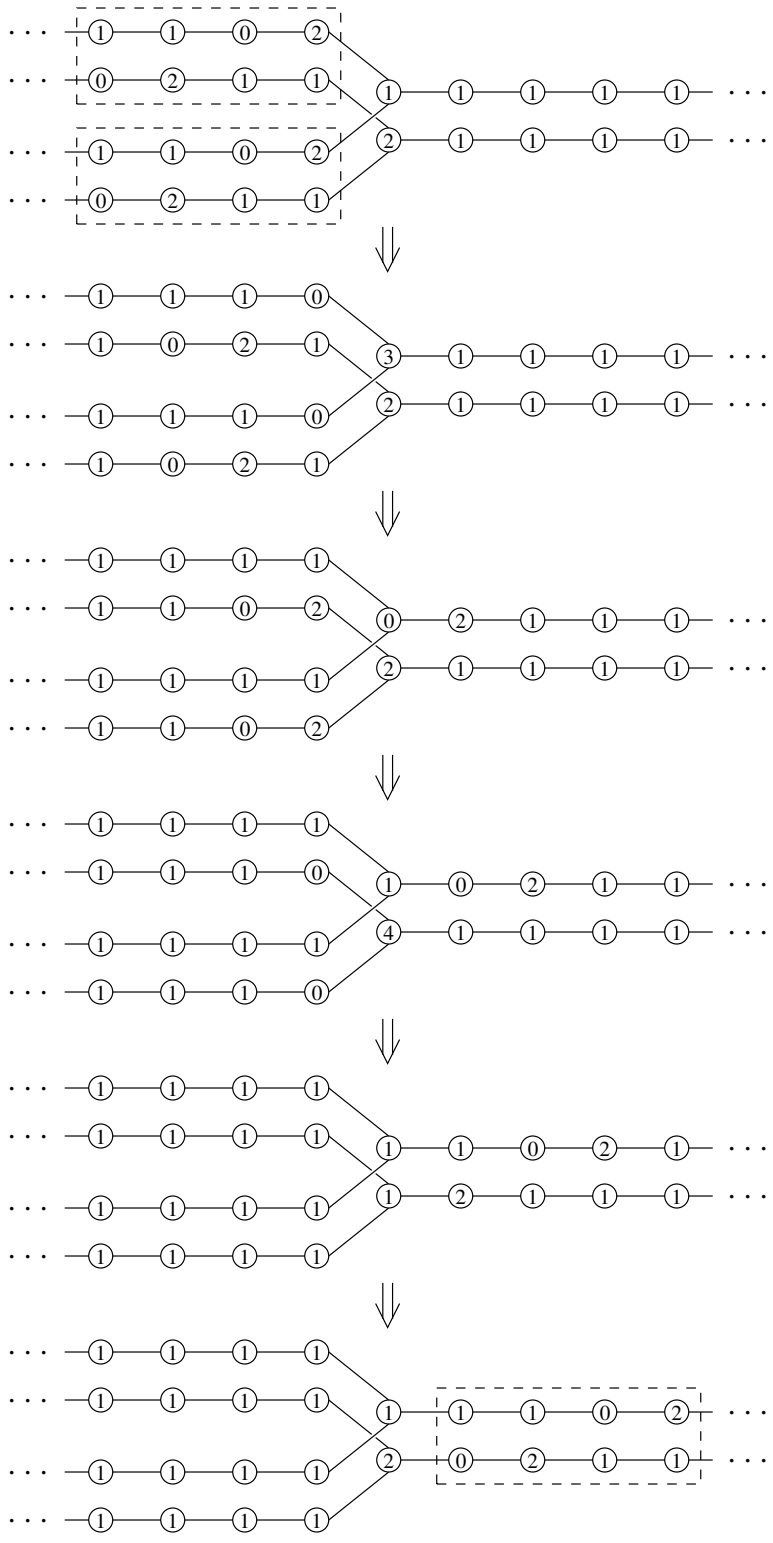

Figure 4.12: The sandpile AND-gate evaluates $1 \wedge 1 = 1$

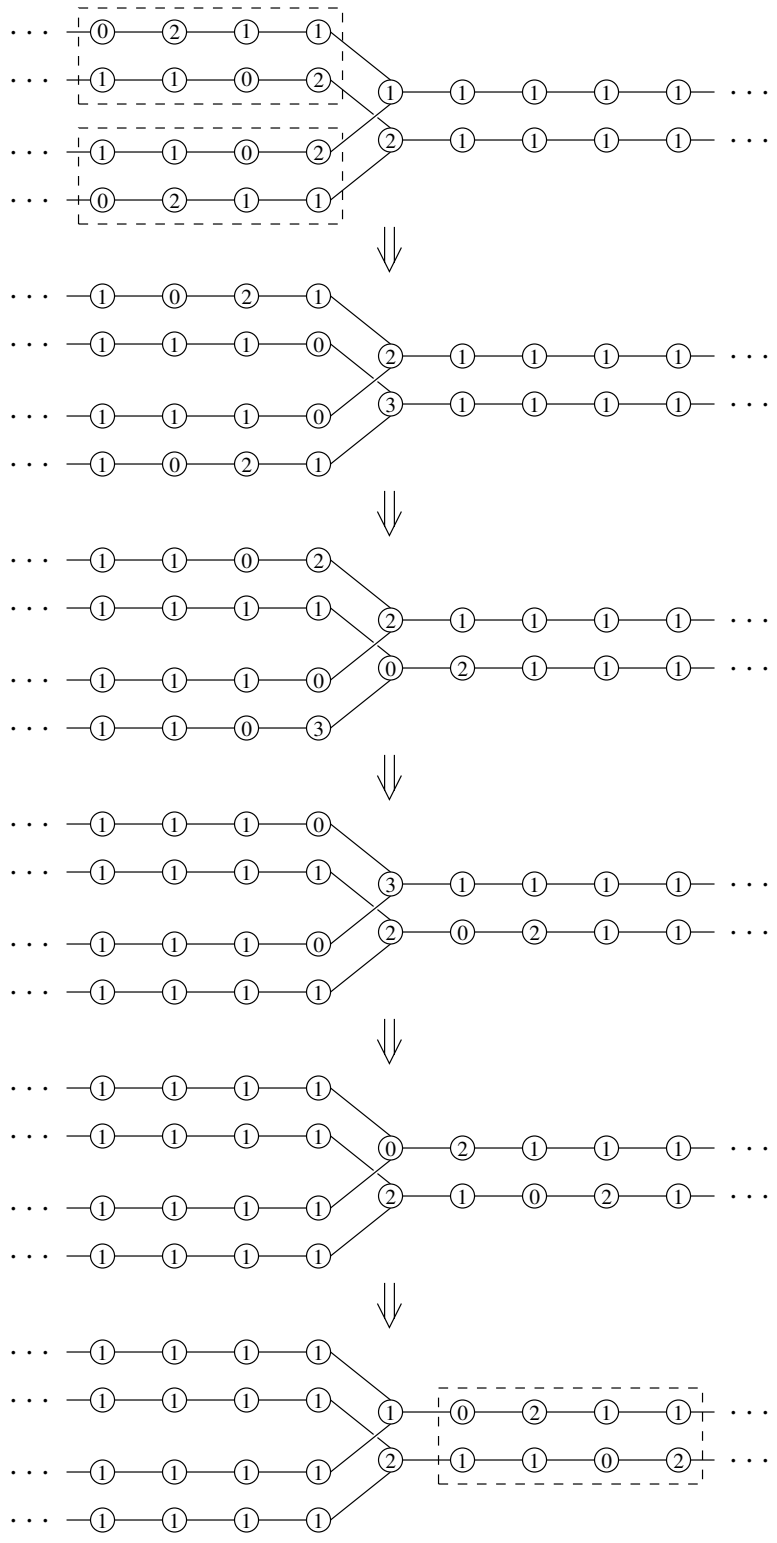

Figure 4.13: The sandpile AND-gate evaluates $1 \wedge 0 = 0$

We now have most of the components we need in order to build our Turing machine, but we must also describe the k-splitter. In order to propagate k signals from an original signal, we need only have a vertex with k grains of sand and where one line of vertices comes from the left, for instance, k lines of vertices come from the right (see Figure 4.14).

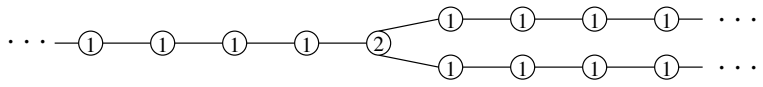

Figure 4.14: A 2-splitter sandpile in a neutral line.

4.3 Sandpile machine

We can now build a cell of the Turing machine out of sandpile gates, splitters, and neutral wires, i.e., doubled neutral lines. We will specify that the wires be at least four vertices long to allow us to see the value being transmitted. We will ensure that the cells are synchronized by standardizing the length of each vertex path through the machine cell. In fact, we can have a path length of 61 vertices.

In order to accommodate for the first cell, there is no OR-gate for state inputs because there is no left-neighboring cell; the first cell is in a transition state either when the machine begins its computation or, when in the previous configuration the second cell was in a transition state, the next-move function moves the tape head left (see Appendix A).

Once we have built the machine hardware, we are able to compute an input by adding and subtracting a grain of sand to alter the wires from neutral configurations to unstable configurations, coding boolean values. The signals propagate along the wires as the unstable vertices topple simultaneously; our Turing machine has started the computation. We will consider the computation to be completed when the state at each cell is halted, i.e., the state output wires are all 0-valued. Then the output of the machine is simply the symbol output at each machine cell.

Appendix A

Sandpile M_{odd}

The sandpile representation of the Turing machine M_{odd} is composed of the four sandpile circuit subcomponents (see Figures A.1–A.4). We then depict the configurations of the first two cells of the sandpile machine computing the input 1 and halting in 118 steps. The halted sandpile machine encodes a blank in the first cell, a 1 in the second cell, and blanks in the remaining tape cells; thus the machine decides that 1 is odd.

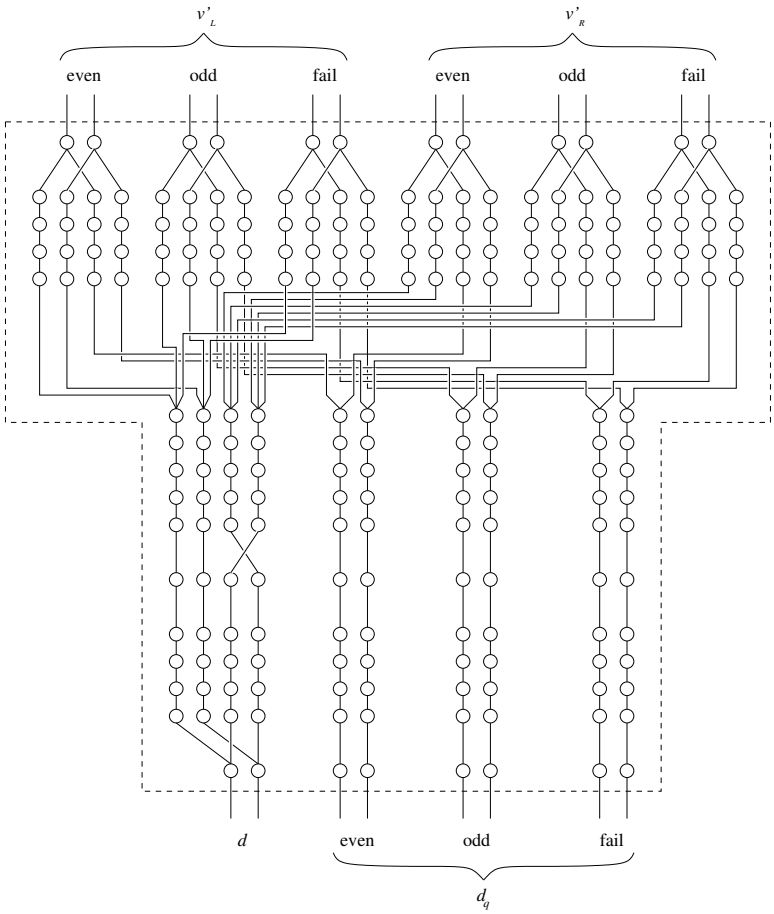

Figure A.1: The sandpile state circuit for M_{odd}

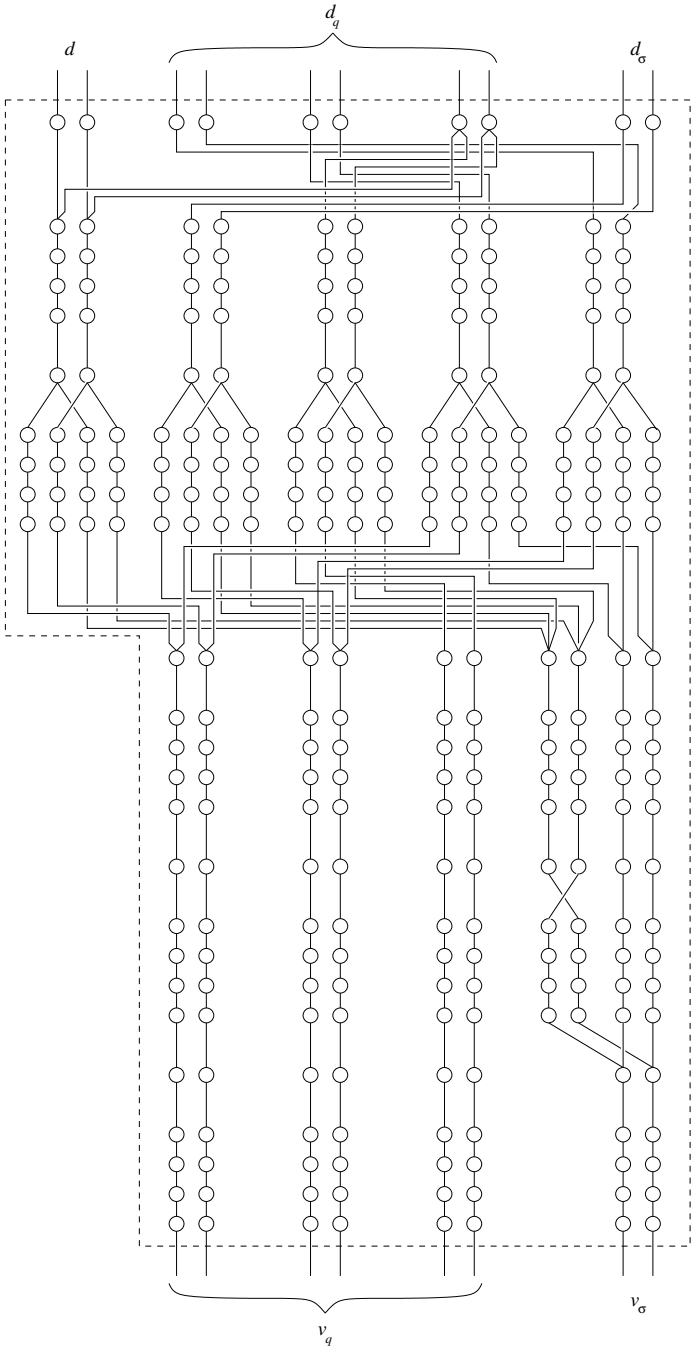

Figure A.2: The sandpile δ-circuit for M_{odd}

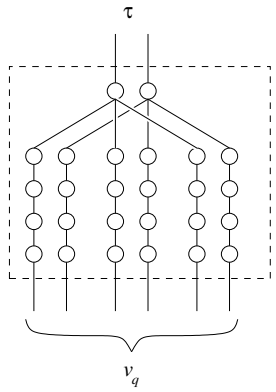

Figure A.3: The sandpile transition circuit for M_{odd}

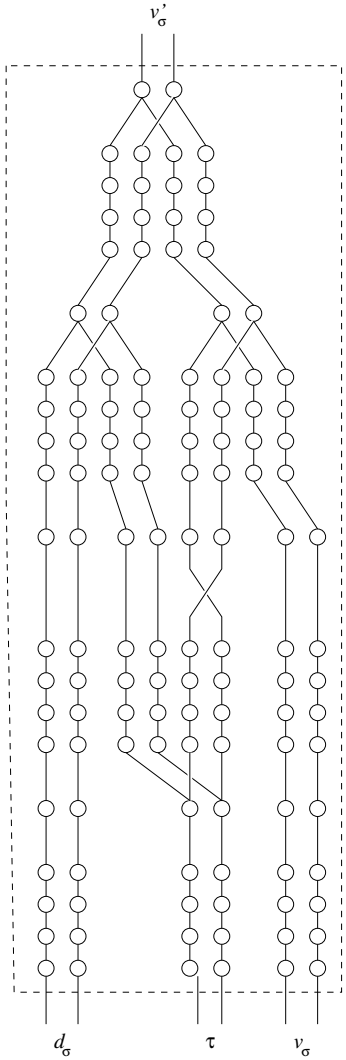

Figure A.4: The sandpile symbol circuit for M_{odd}

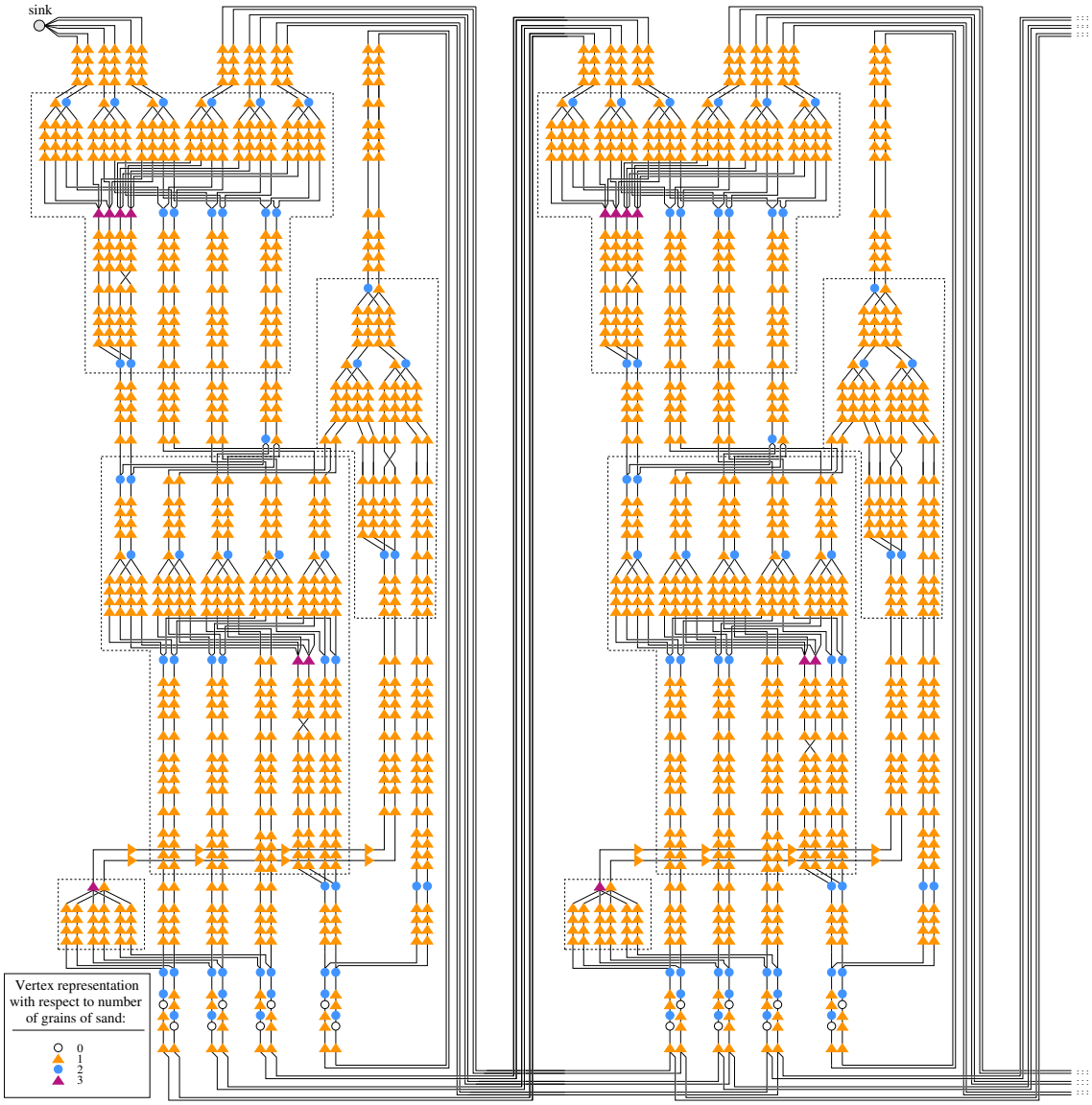

Figure A.5: The initial configuration of the sandpile M_{odd} with input 1

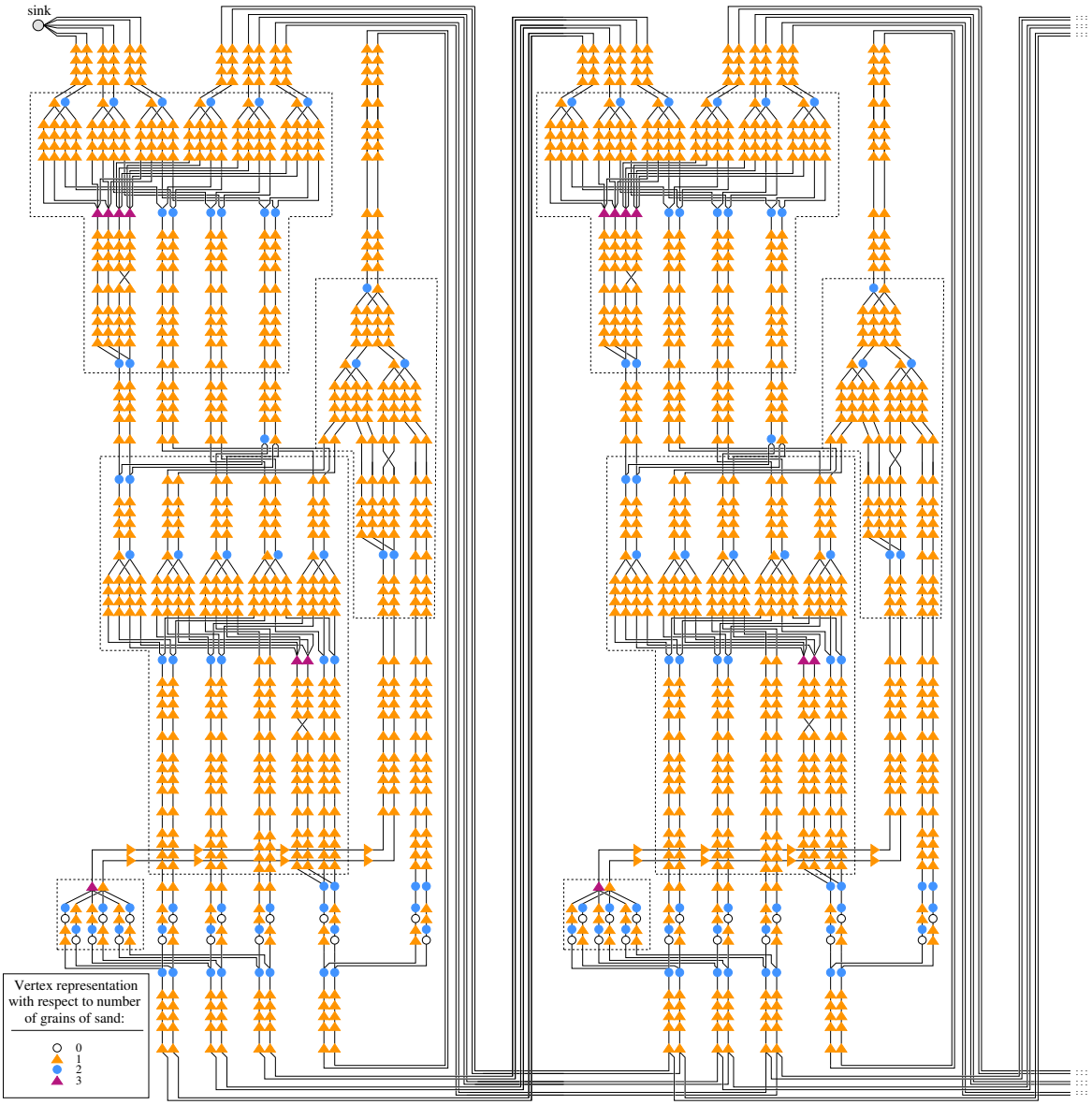

Figure A.6: The 6^{th} sandpile configuration of M_{odd} with input 1

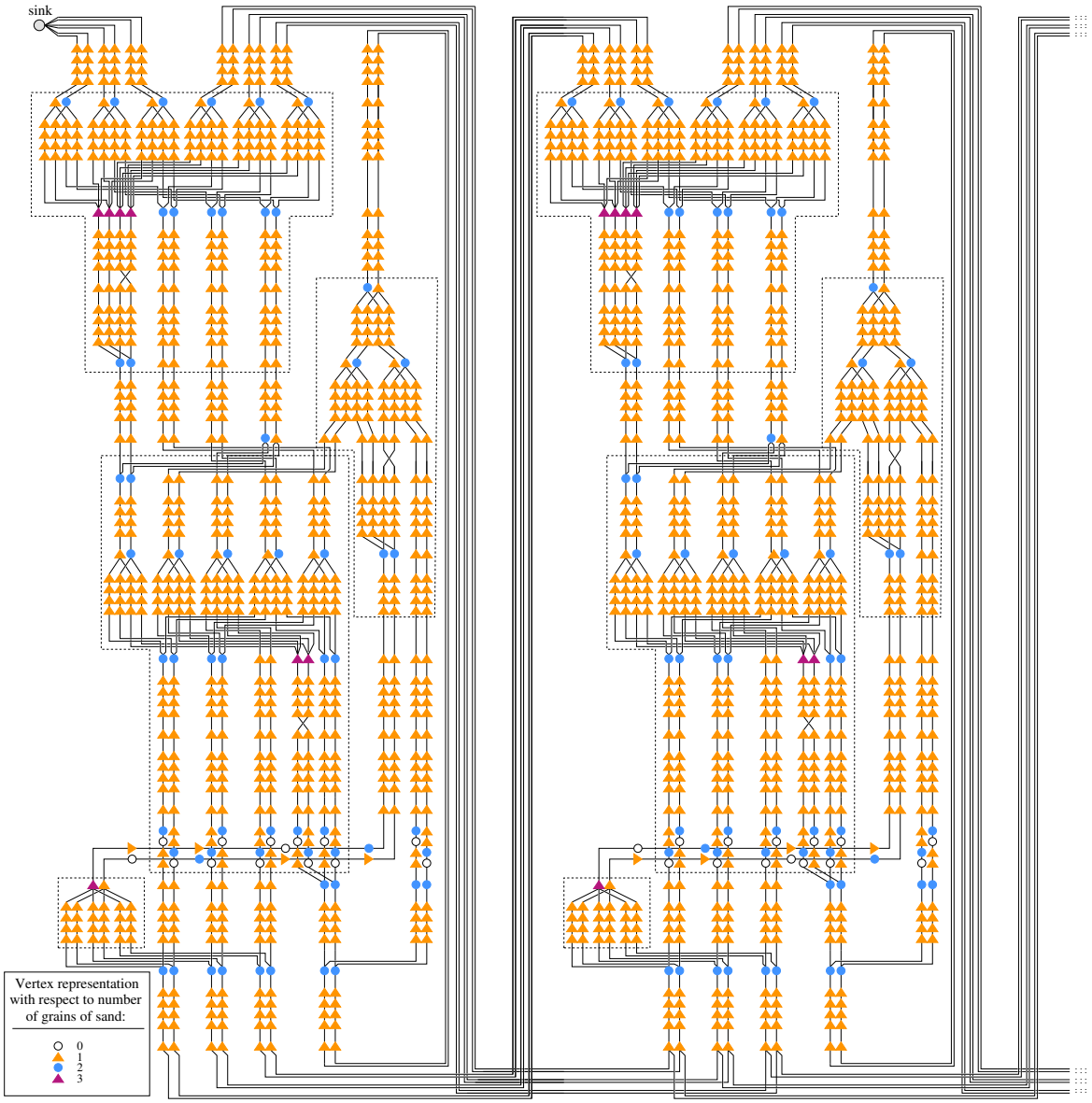

Figure A.7: The 11^{th} sandpile configuration of M_{odd} with input 1

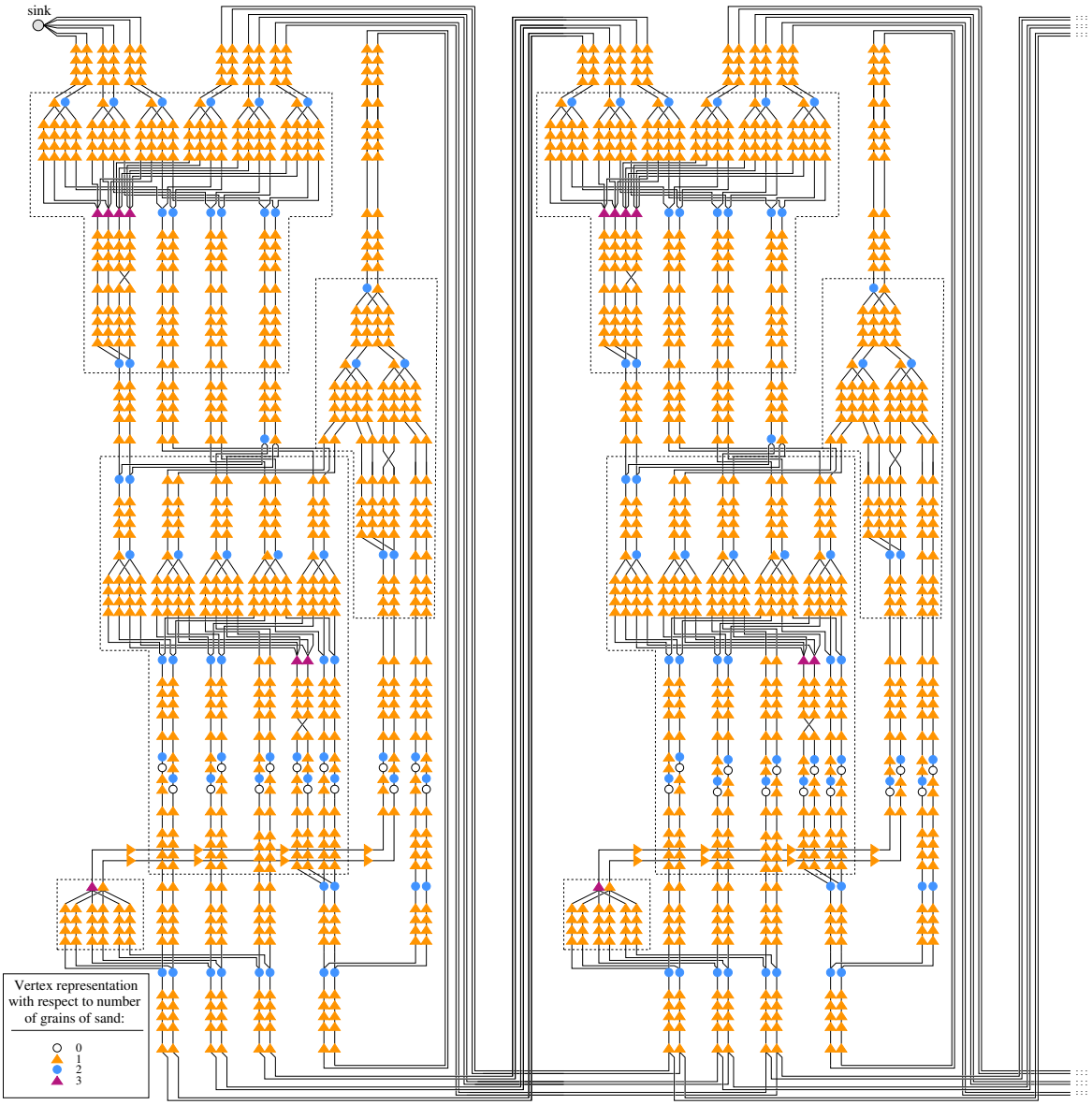

Figure A.8: The 16^{th} sandpile configuration of M_{odd} with input 1

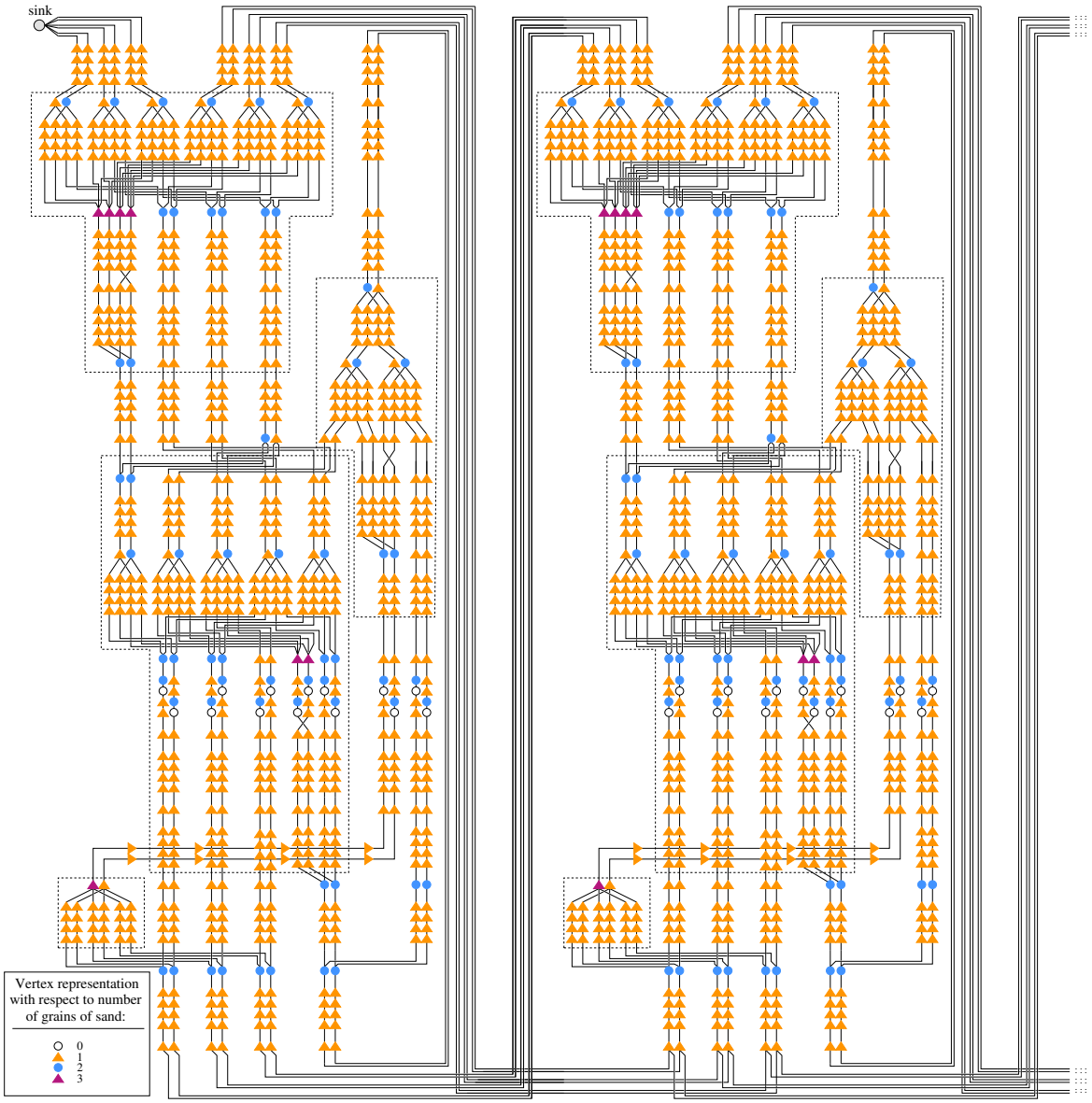

Figure A.9: The 21^{th} sandpile configuration of M_{odd} with input 1

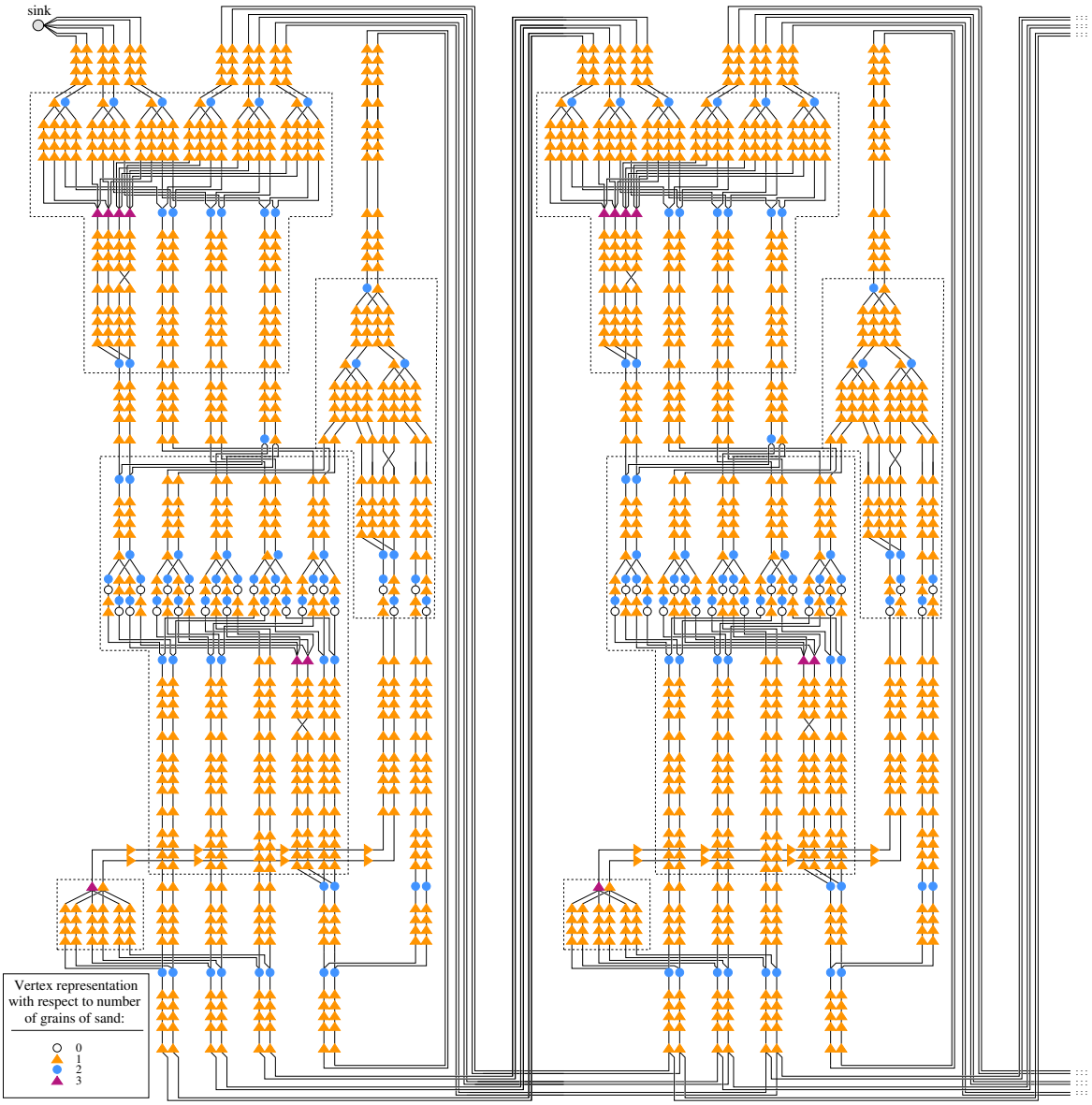

Figure A.10: The 26^{th} sandpile configuration of M_{odd} with input 1

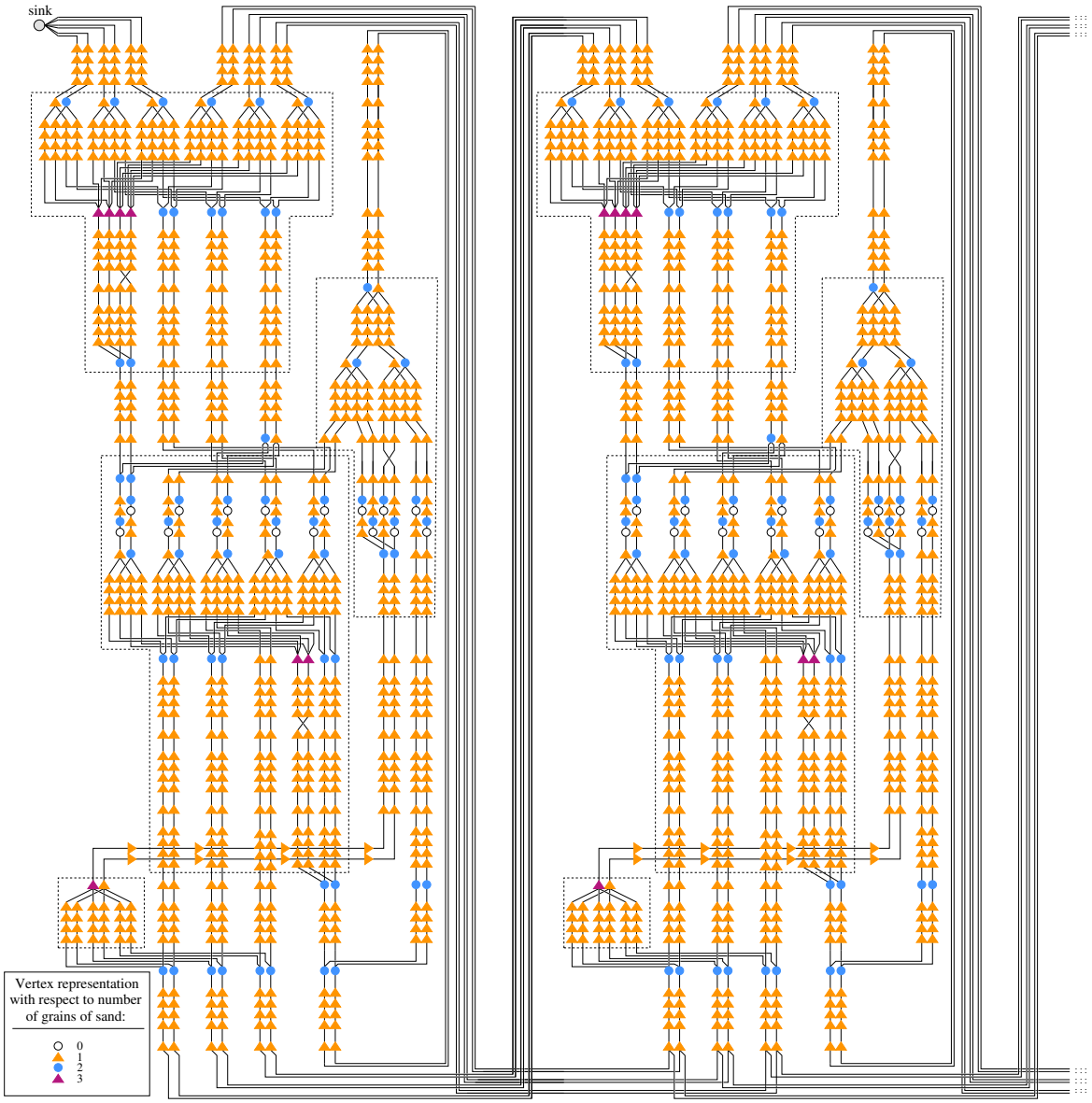

Figure A.11: The 31^{st} sandpile configuration of M_{odd} with input 1

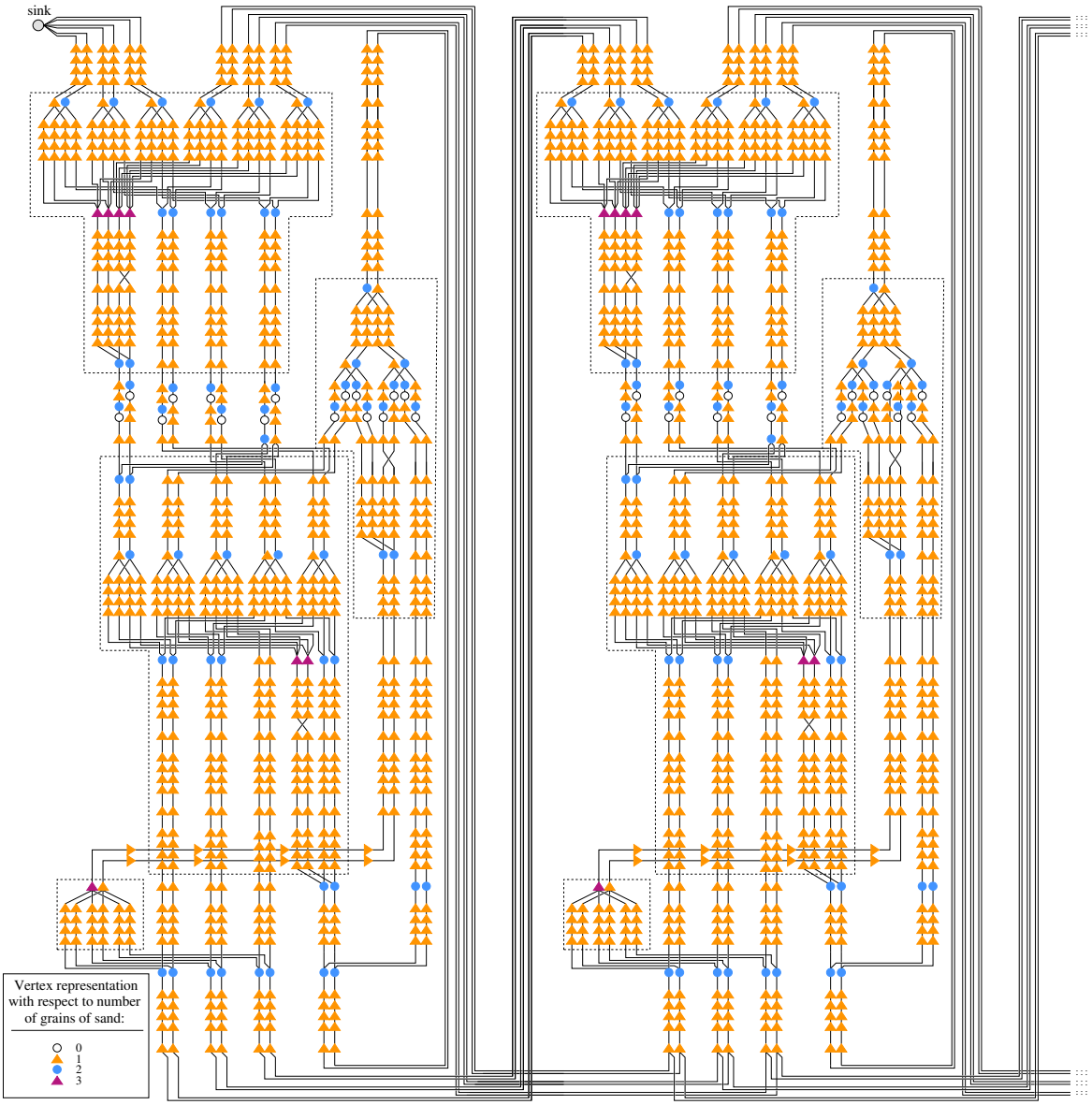

Figure A.12: The 37^{th} sandpile configuration of M_{odd} with input 1

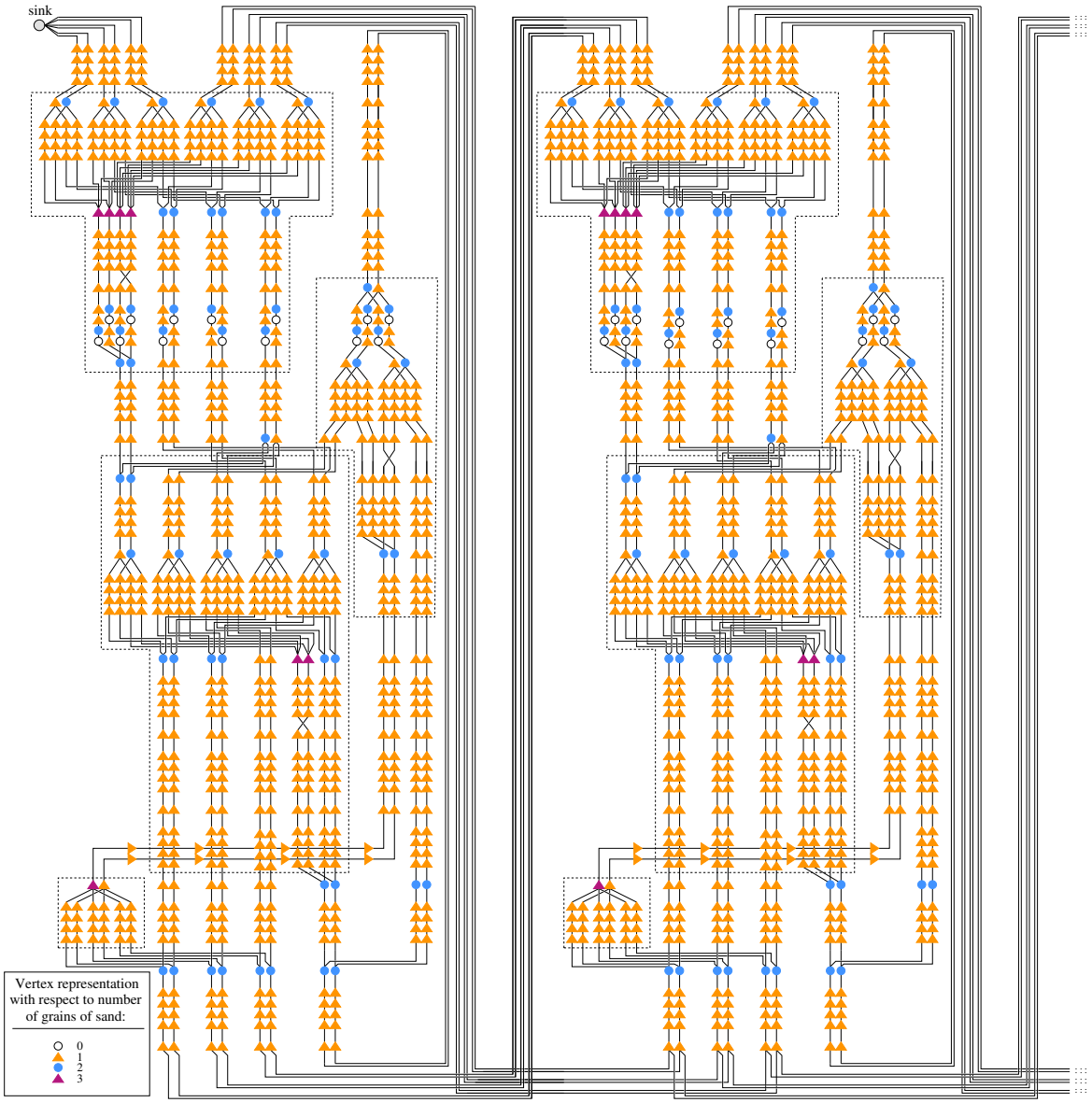

Figure A.13: The 42^{nd} sandpile configuration of M_{odd} with input 1

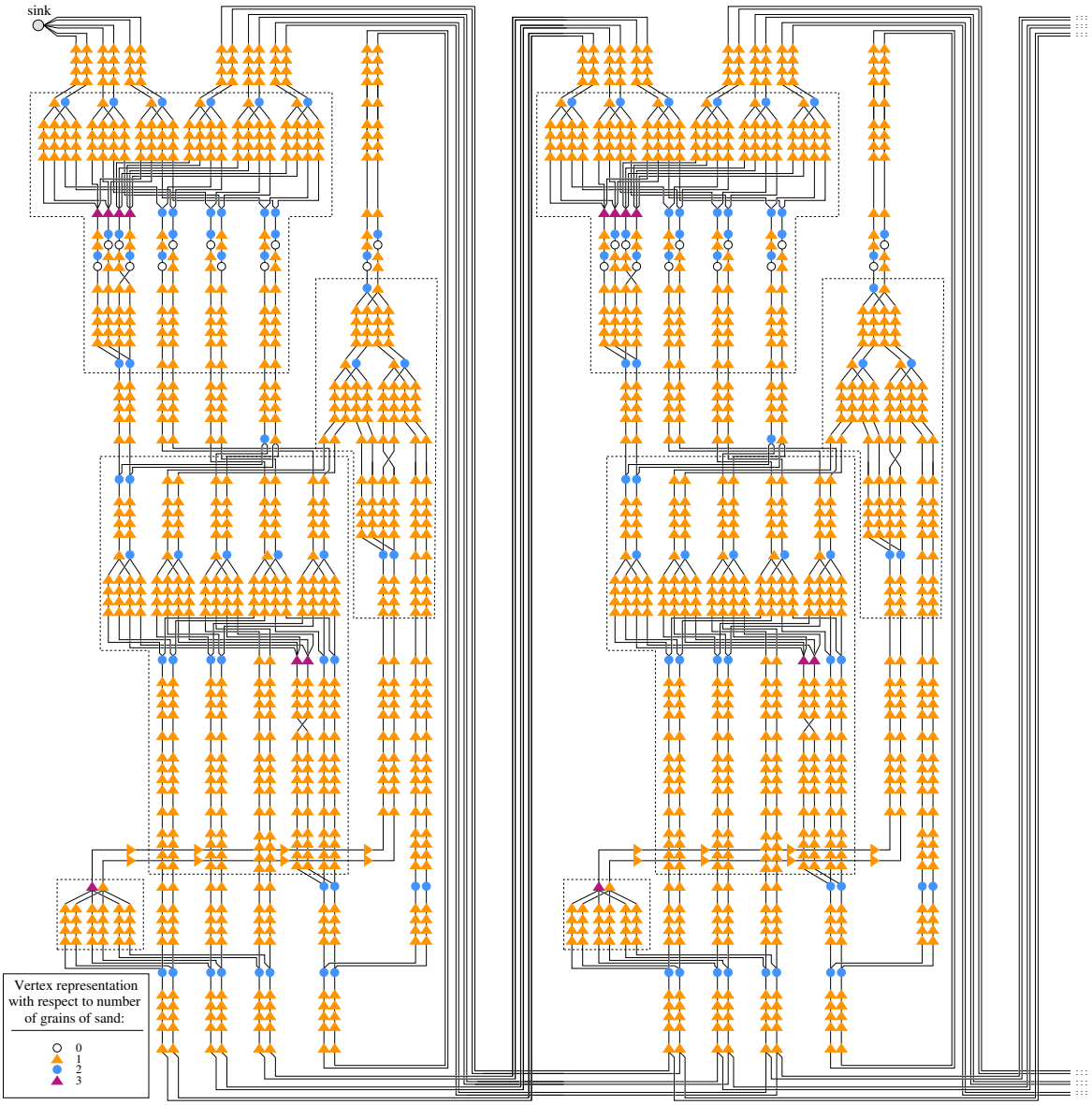

Figure A.14: The 47^{th} sandpile configuration of M_{odd} with input 1

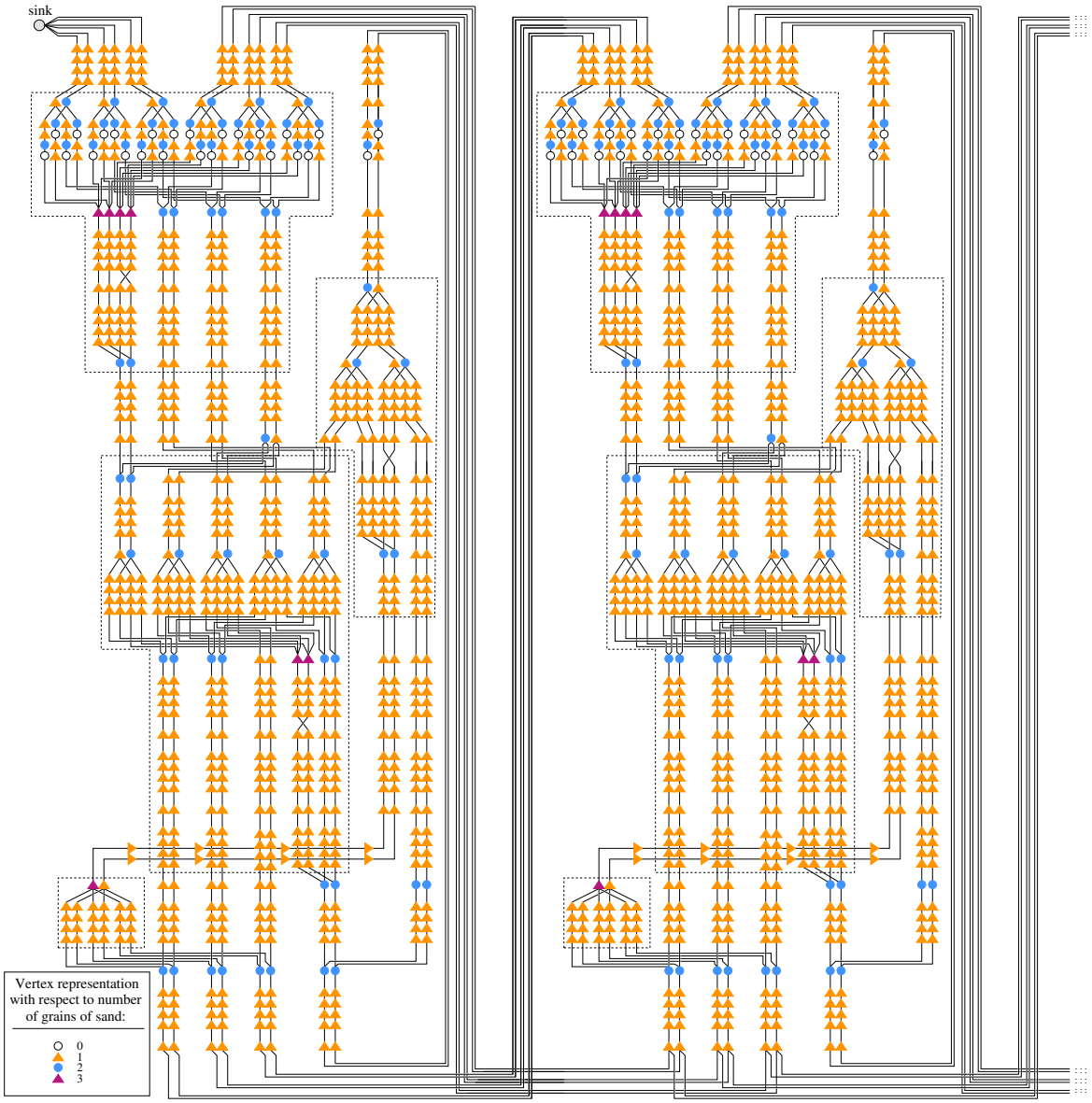

Figure A.15: The 52^{nd} sandpile configuration of M_{odd} with input 1

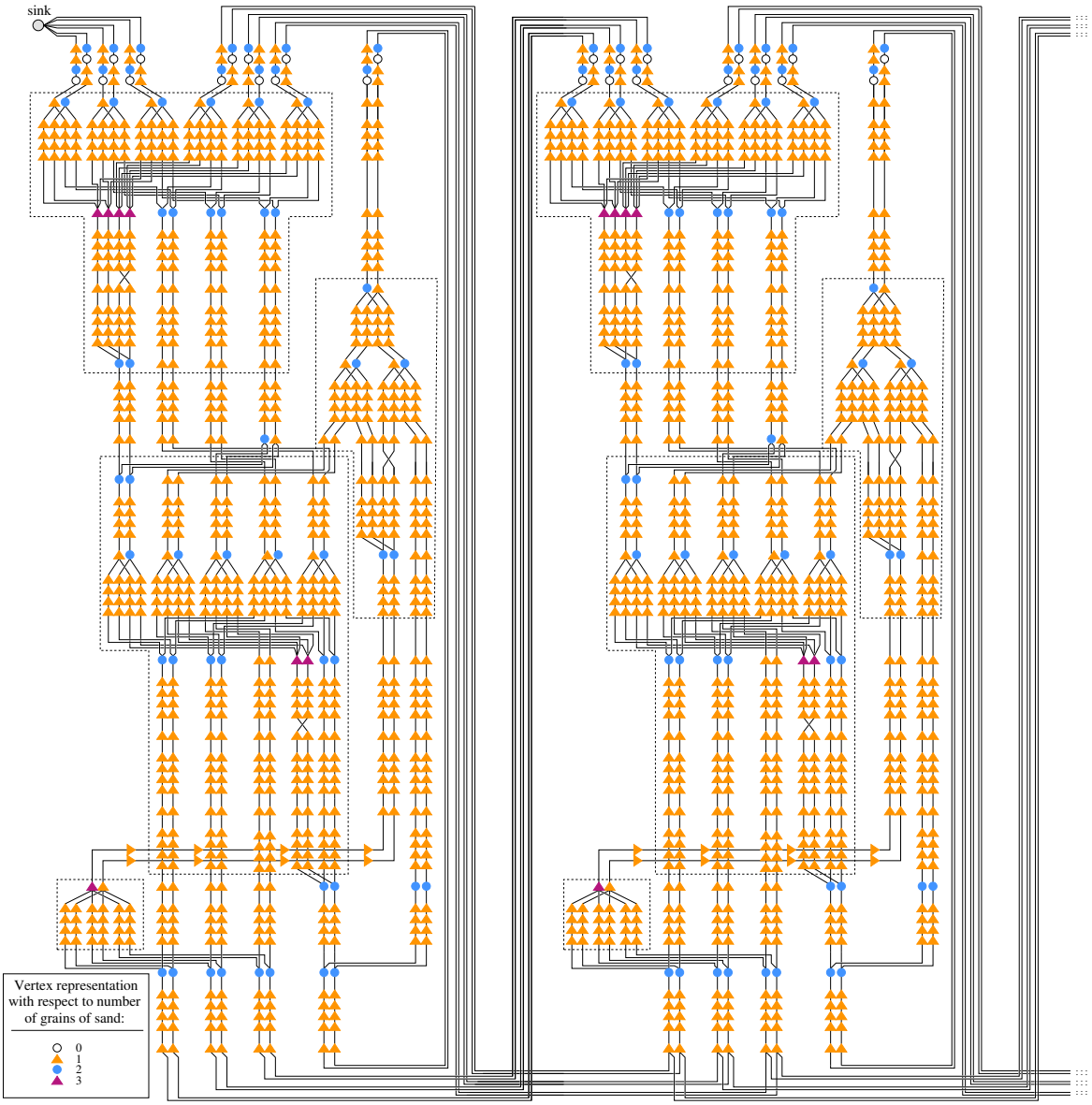

Figure A.16: The 57^{th} sandpile configuration of M_{odd} with input 1

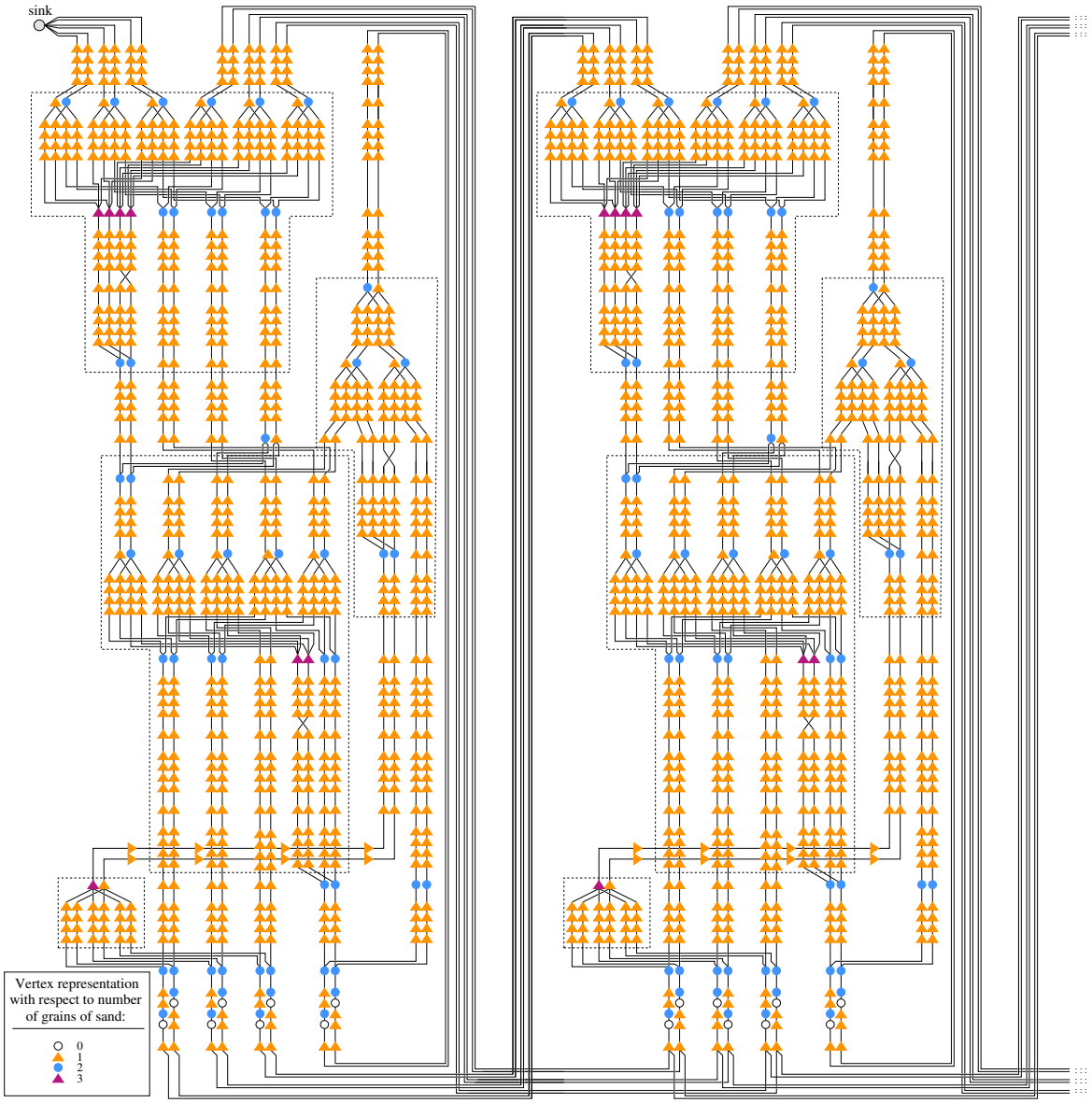

Figure A.17: The 62^{nd} sandpile configuration of M_{odd} with input 1

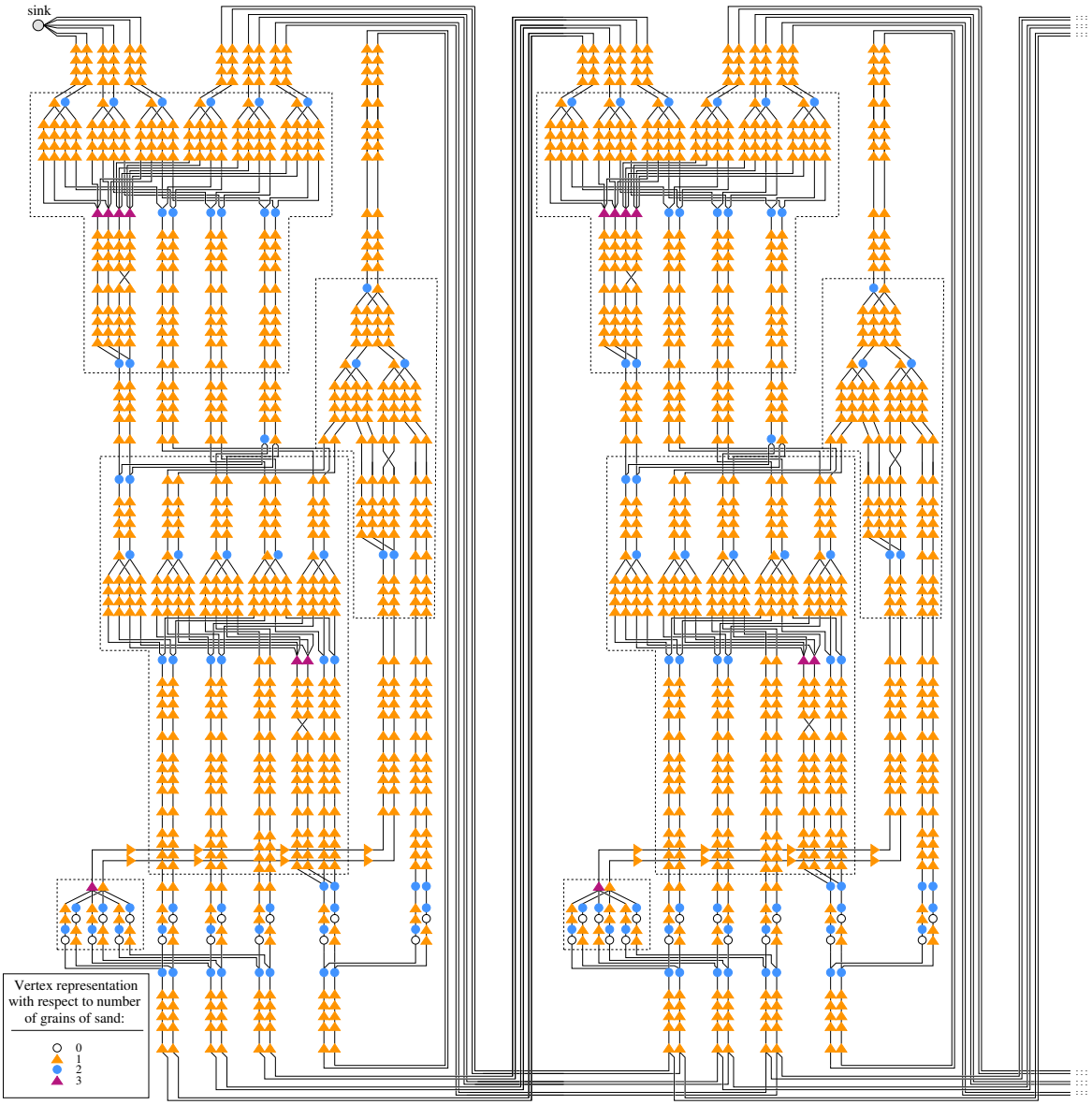

Figure A.18: The 67^{th} sandpile configuration of M_{odd} with input 1

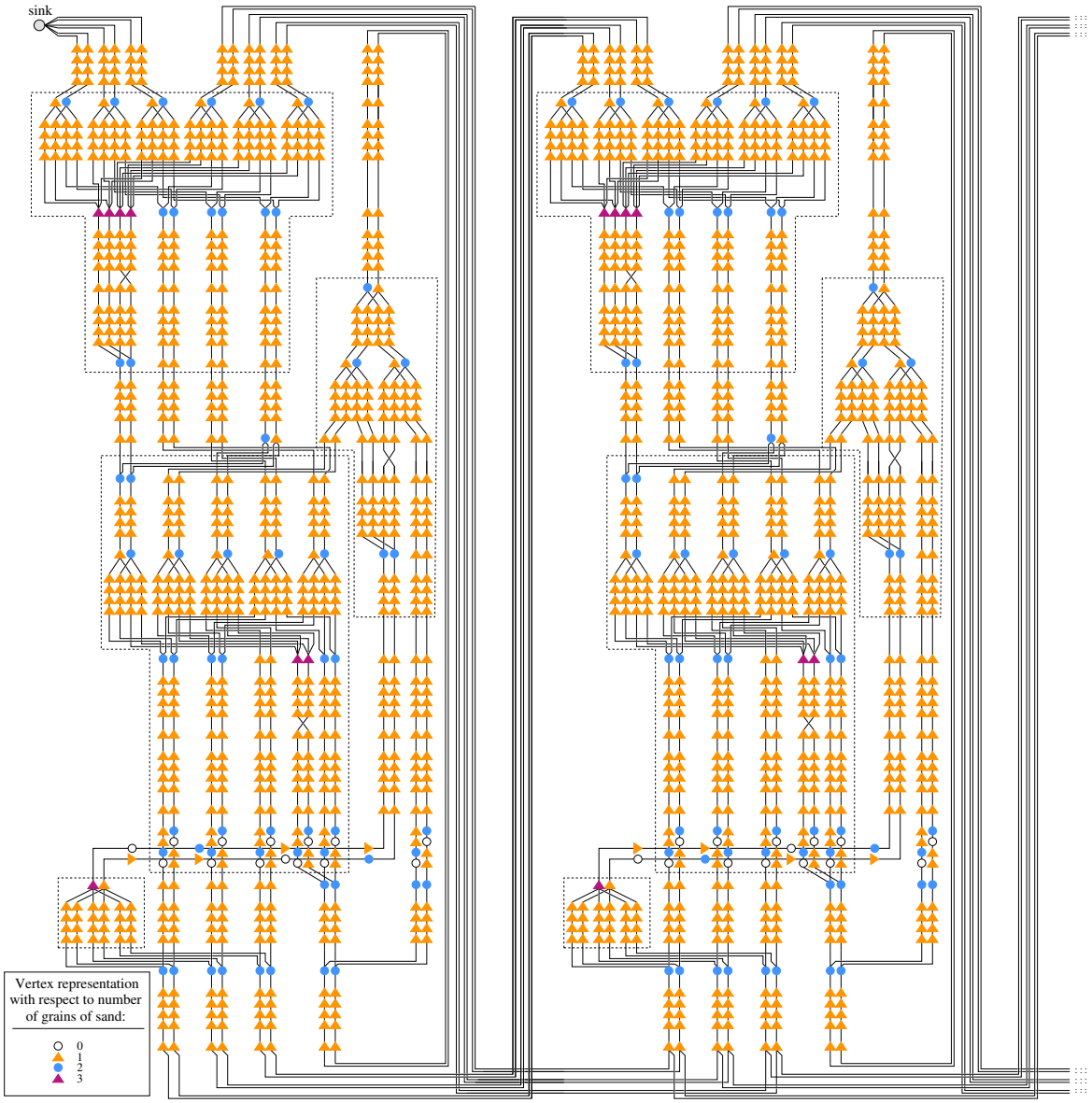

Figure A.19: The 72^{nd} sandpile configuration of M_{odd} with input 1

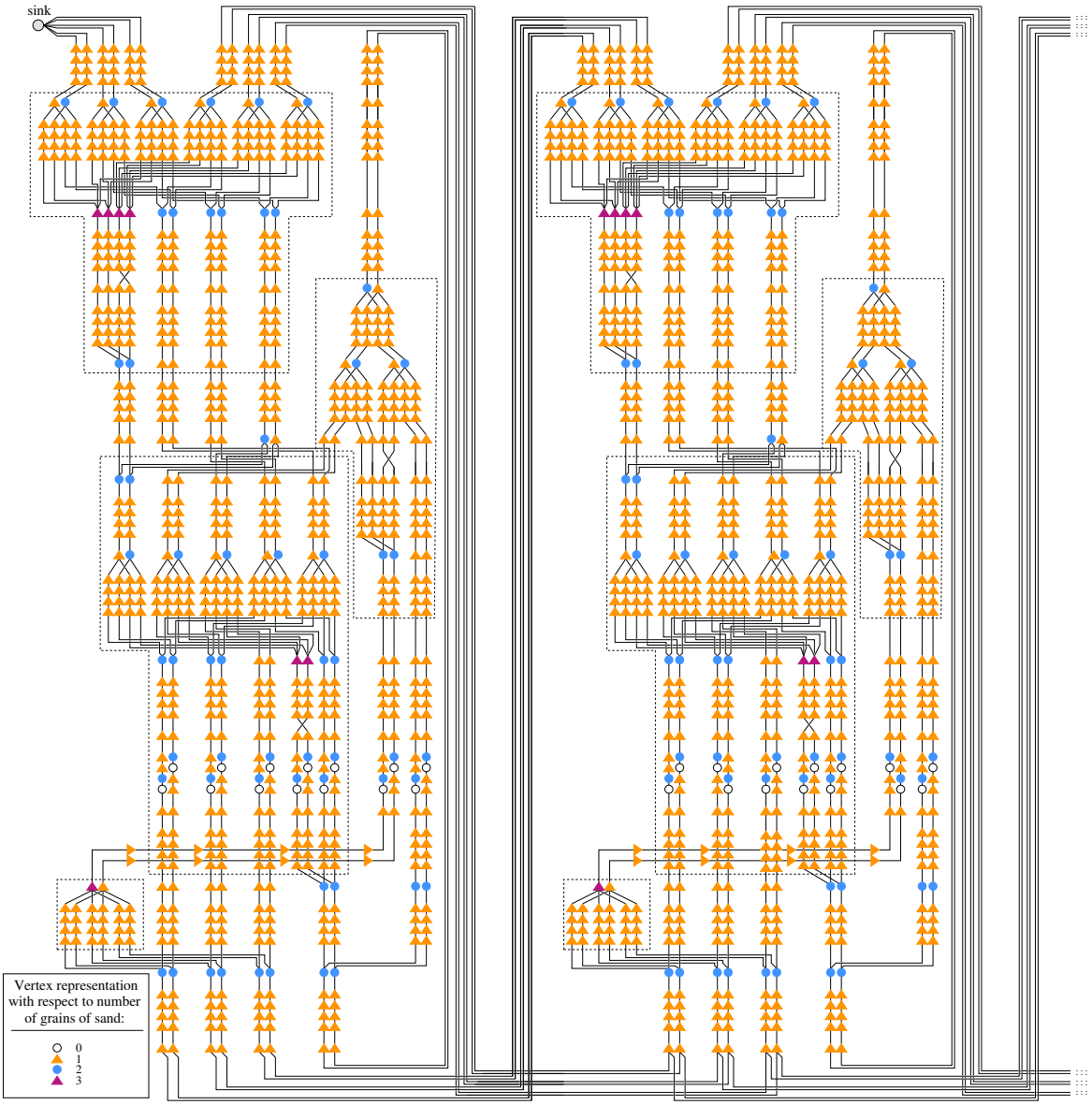

Figure A.20: The 77^{th} sandpile configuration of M_{odd} with input 1

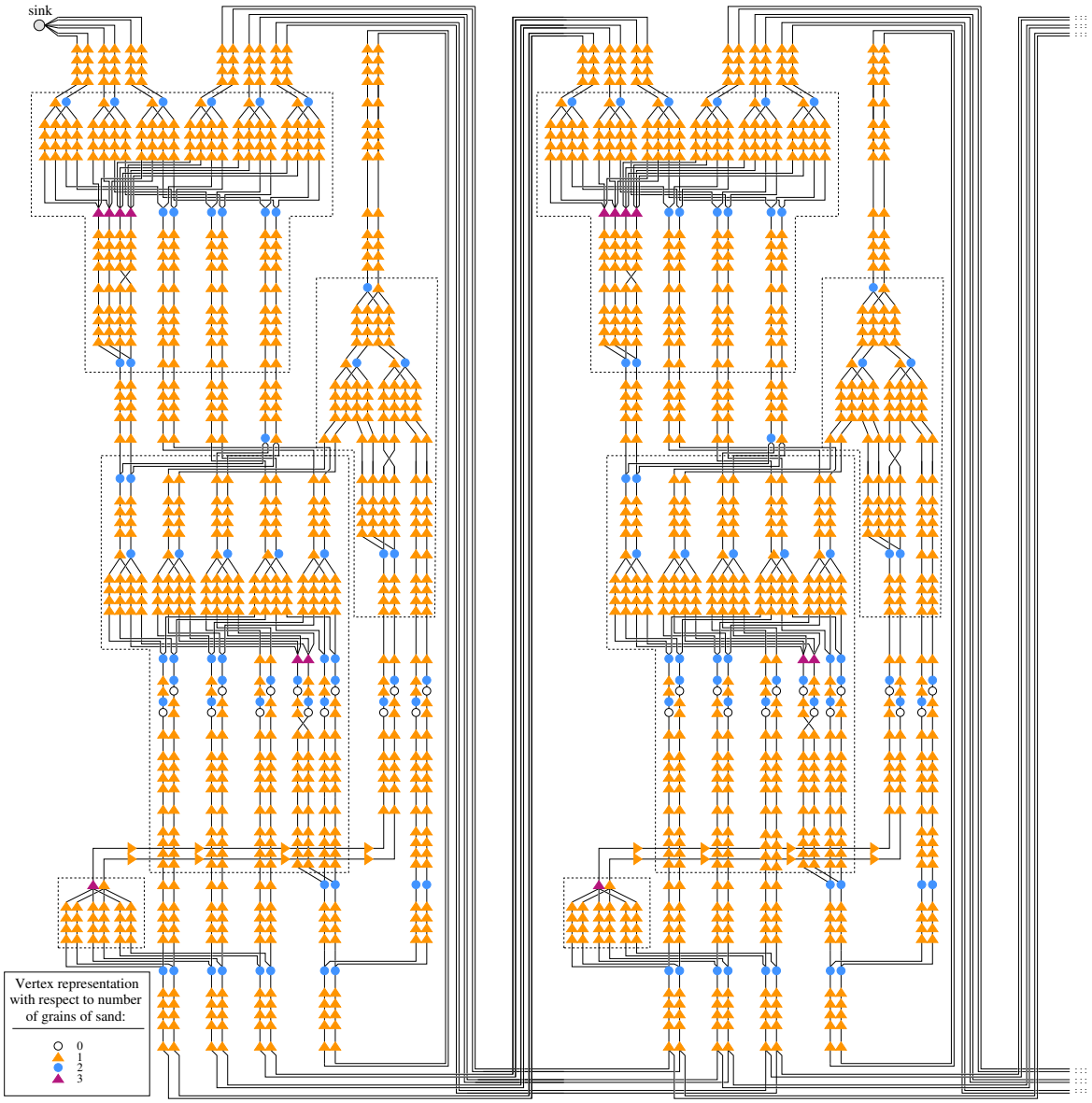

Figure A.21: The 82^{nd} sandpile configuration of M_{odd} with input 1

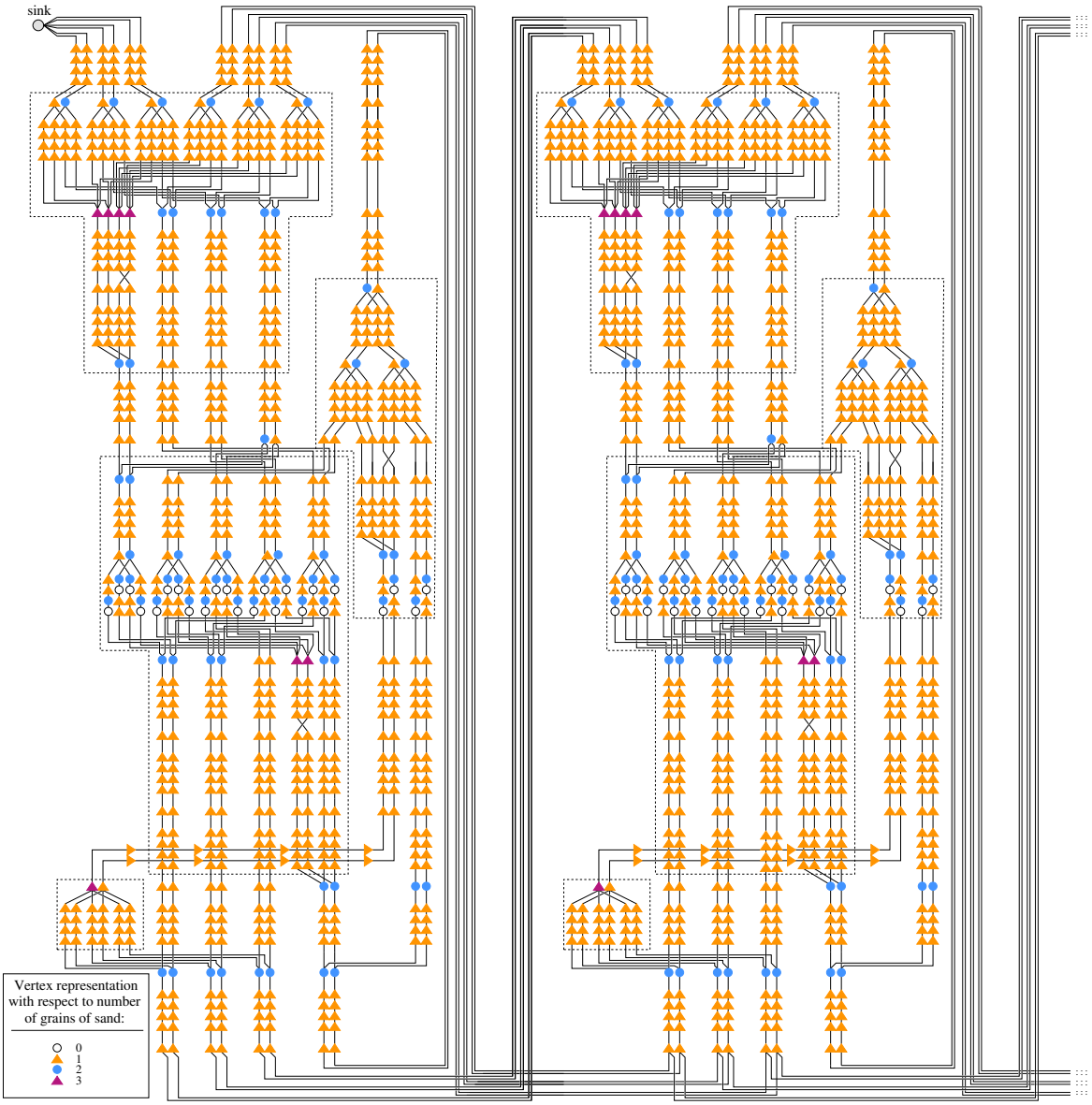

Figure A.22: The 87^{th} sandpile configuration of M_{odd} with input 1

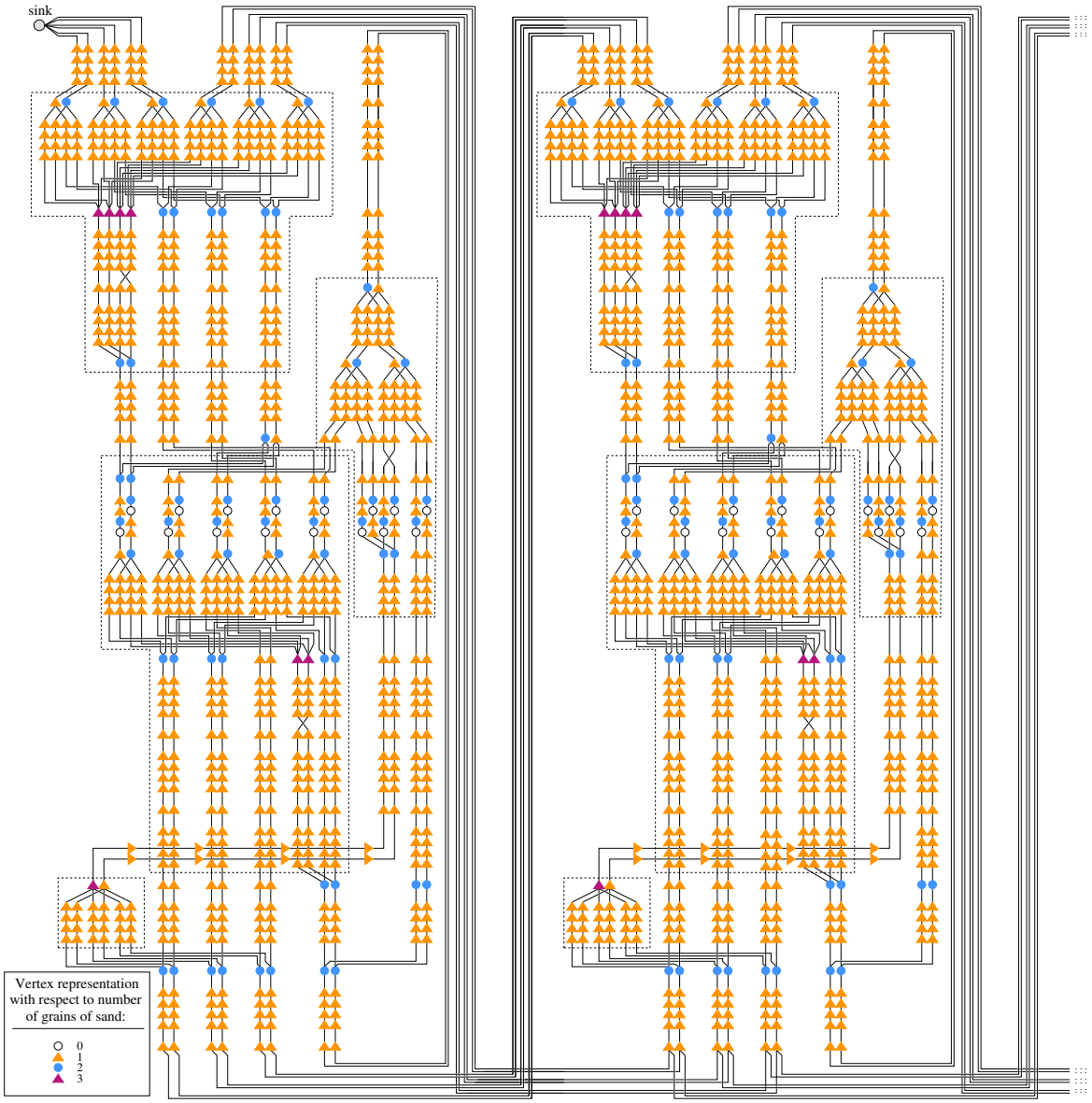

Figure A.23: The 92^{nd} sandpile configuration of M_{odd} with input 1

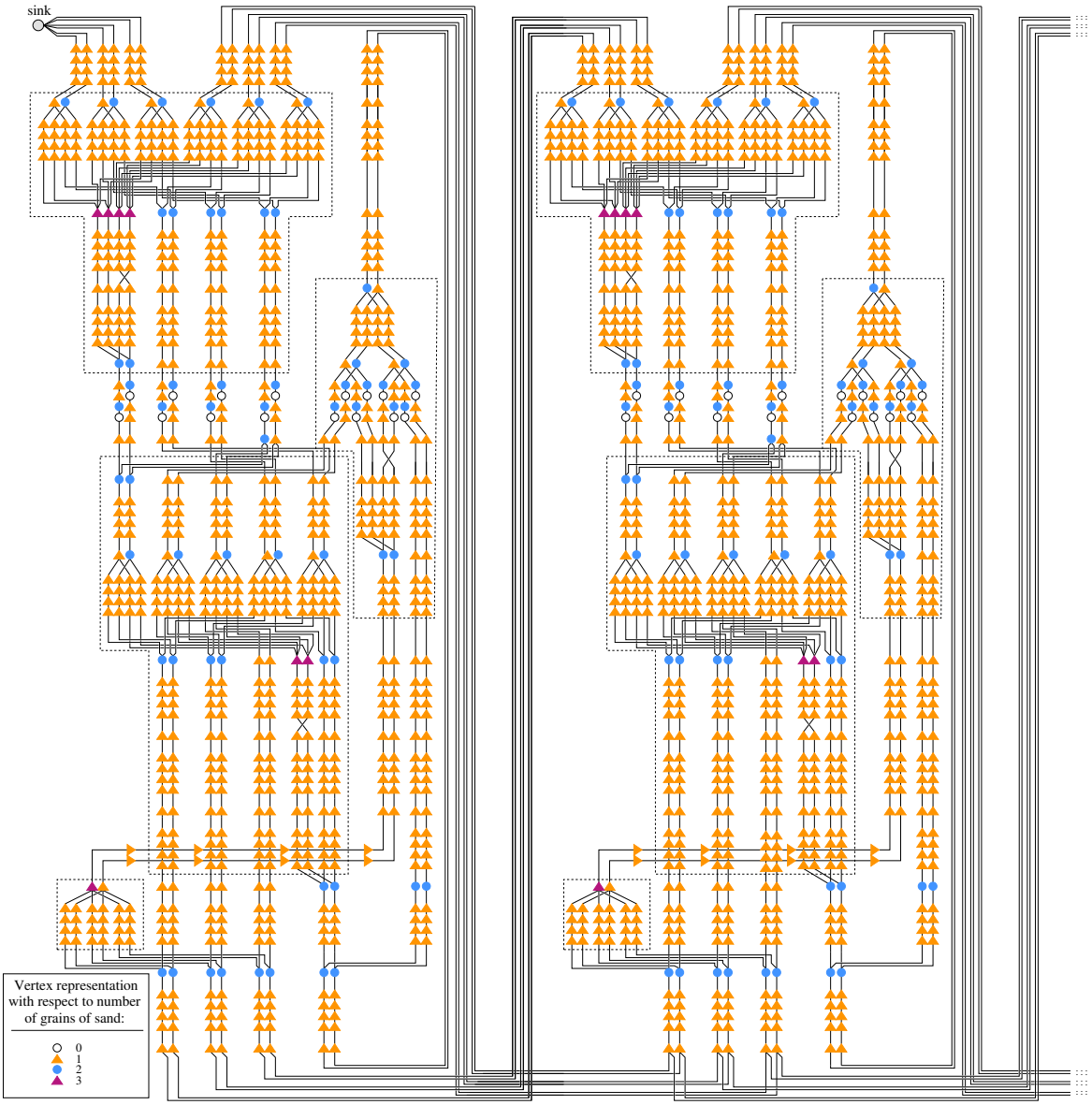

Figure A.24: The 98^{th} sandpile configuration of M_{odd} with input 1

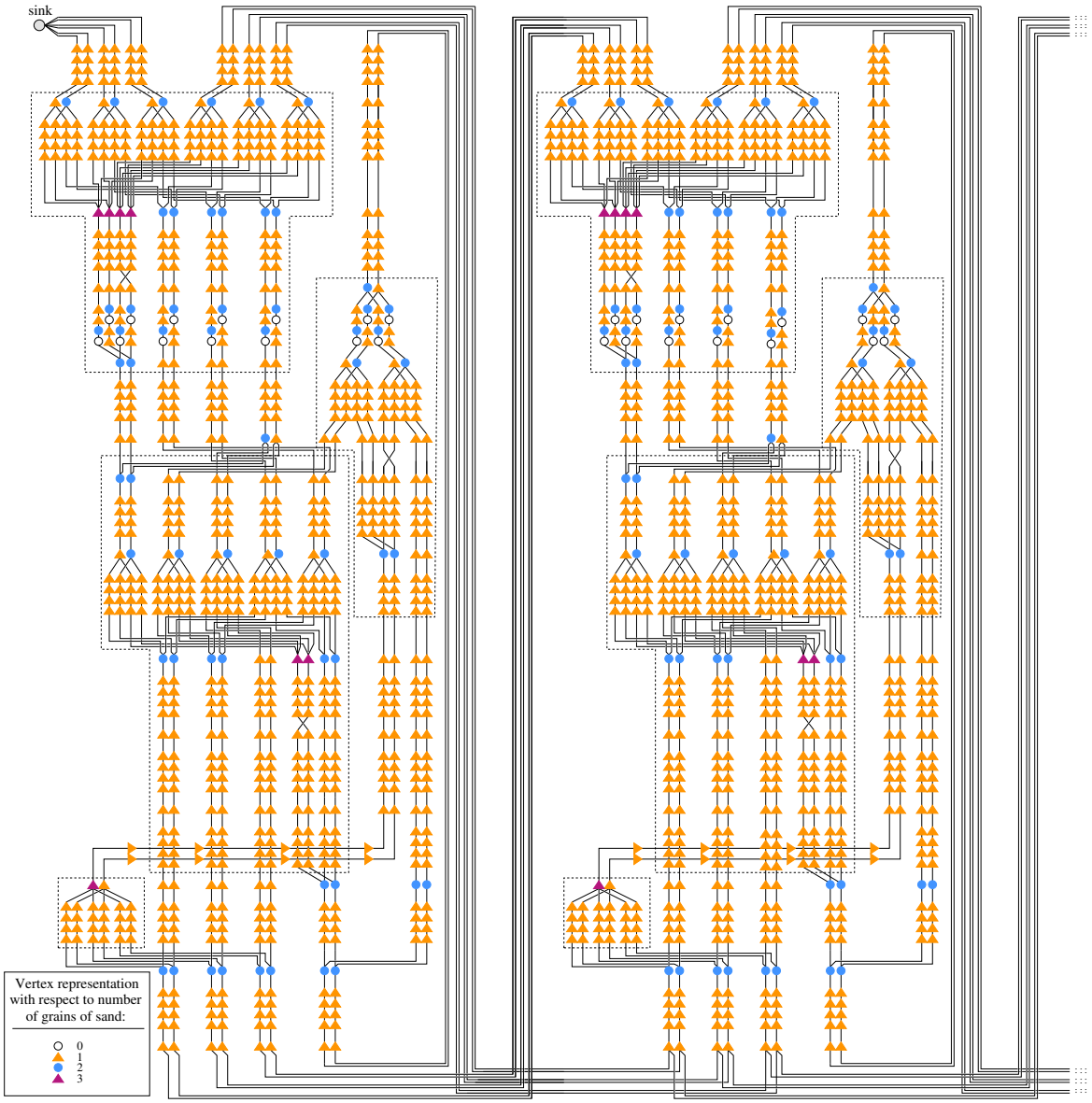

Figure A.25: The 103^{rd} sandpile configuration of M_{odd} with input 1

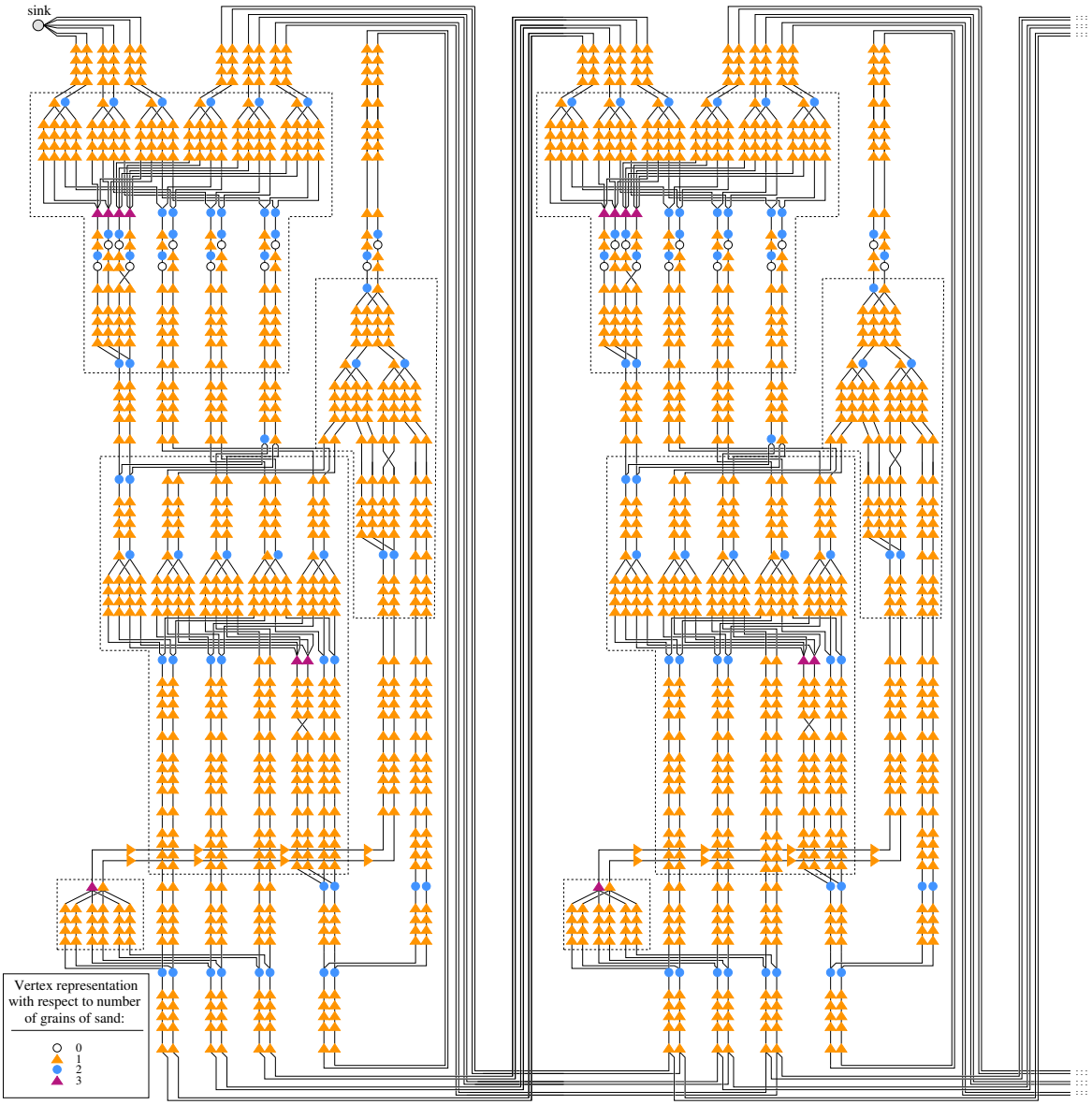

Figure A.26: The 108^{th} sandpile configuration of M_{odd} with input 1

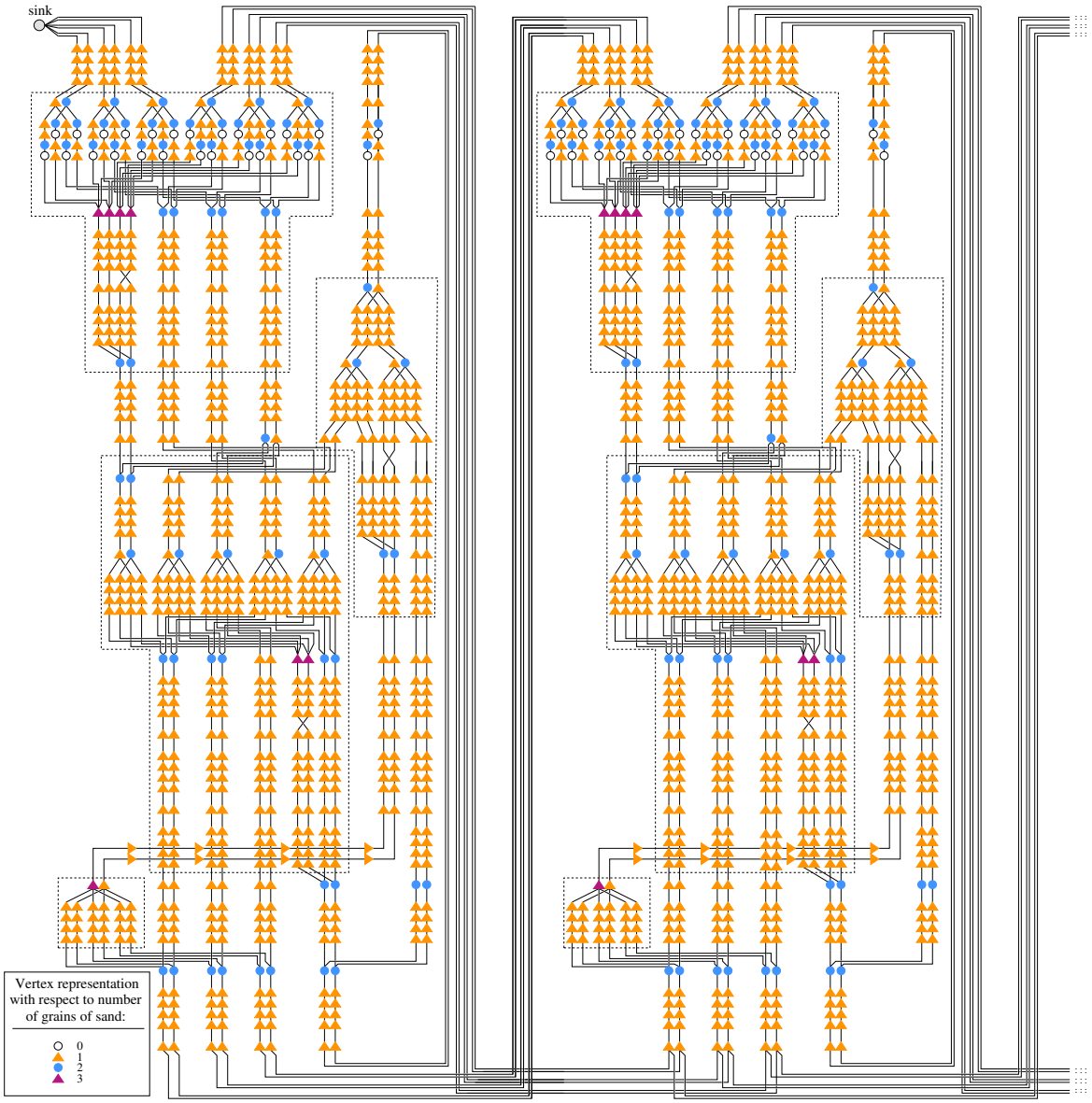

Figure A.27: The 113^{th} sandpile configuration of M_{odd} with input 1

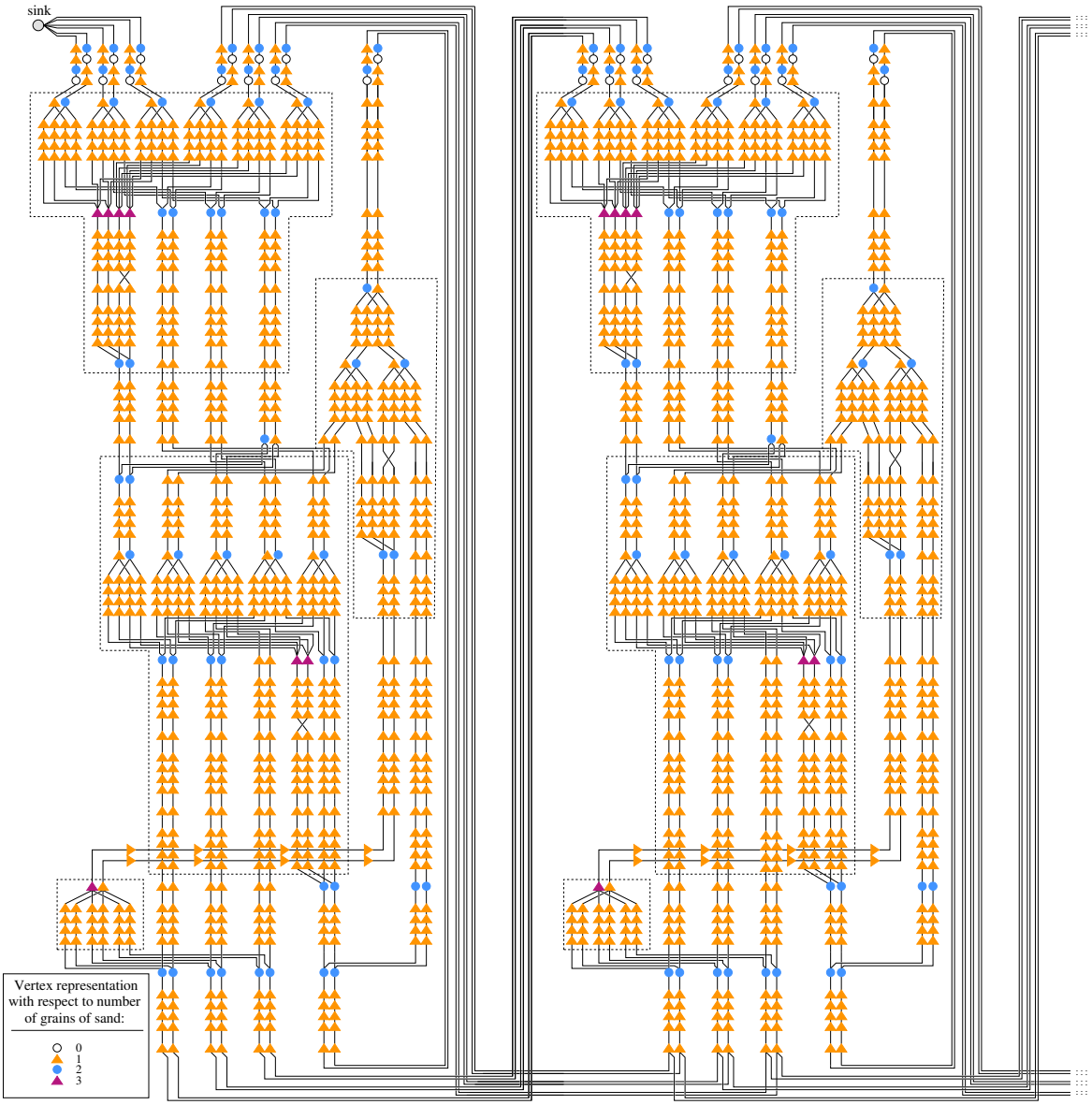

Figure A.28: M_{odd} halts after the 118^{th} sandpile configuration with output 1

References

[Bak96] Per Bak. *How Nature Works: The Science of Self-Organized Criticality*. Copernicus, 1996.

[BTW88] P. Bak, C. Tang, and K. Wiesenfeld. Self-organized criticality. *Physical Review A*, 364, 1988.

[GM96] Eric Goles and Maurice Margenstern. Sandpile as a universal computer. *International Journal of Modern Physics C*, 7:113–122, 1996.

[HLM+08] Alexander E. Holroyd, Lionel Levine, Karola Meszaros, Yuval Peres, James Propp, and David B. Wilson. Chip-firing and rotor-routing on directed graphs. 2008.

[Lev07] Lionel Levine. The sandpile group of a tree, 2007.

[Min67] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., 1967.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation, Second Edition*. Thompson Course Technology, 2006.