Balance Machines: Computing = Balancing

Joshua J. Arulanandham, Cristian S. Calude, Michael J. Dinneen

Department of Computer Science
The University of Auckland
Auckland, New Zealand
{jaru003,cristian,mjd}@cs.auckland.ac.nz

Abstract. We propose a natural computational model called a *balance machine.* The computational model consists of components that resemble ordinary physical balances, each with an intrinsic property to automatically balance the weights on their left, right pans. If we start with certain fixed weights (representing *inputs*) on some of the pans, then the balance–like components would vigorously try to balance themselves by filling the rest of the pans with suitable weights (representing the *outputs*). This balancing act can be viewed as a computation. We will show that the model allows us to construct those primitive (hardware) components that serve as the building blocks of a general purpose (universal) digital computer: logic gates, memory cells (flip-flops), and transmission lines. One of the key features of the balance machine is its "bidirectional" operation: both a function and its (partial) inverse can be computed spontaneously using the same machine. Some practical applications of the model are discussed.

1 Computing as a "Balancing Feat"

A detailed account of the proposed model will be given in Section 2. In this section, we simply convey the essence of the model without delving into technical details. The computational model consists of components that resemble ordinary physical balances (see Figure 1), each with an intrinsic property to automatically balance the weights on their left, right pans. In other words, if we start with certain fixed weights (representing *inputs*) on some of the pans, then the balance–like components would vigorously try to balance themselves by filling the rest of the pans with suitable weights (representing the *outputs*). Roughly speaking, the proposed machine has a natural ability to load itself with (output) weights that "balance" the input. This balancing act can be viewed as a computation. There is just one rule that drives the whole computational process: *the left and right pans of the individual balances should be made equal.* Note that the machine is designed in such a way that the balancing act would happen automatically by virtue of physical laws (i.e., the machine is self-regulating).[1] One of our aims is to show that all computations can be ultimately expressed using one primitive operation:

[1] If the machine cannot (eventually) balance itself, it means that the particular instance does not have a solution.

balancing. Armed with the *computing = balancing* intuition, we can see basic computational operations in a different light. In fact, an important result of this paper is that this sort of intuition suffices to conceptualize/implement *any* computation performed by a conventional computer.

Fig. 1. Physical balance.

The rest of the paper is organized as follows: Section 2 gives a brief introduction to the proposed computational model; Section 3 discusses a variety of examples showing how the model can be made to do basic computations; Section 4 is a brief note on the universality feature of the model; Section 5 reviews the notion of *bilateral computing* and discusses an application (solving the classic SAT problem); Section 6 concludes the paper.

2 The Balance Machine Model

At the core of the proposed natural computational model are components that resemble a physical balance. In ancient times, the shopkeeper at a grocery store would place a standard weight in the left pan and would try to load the right pan with a commodity whose weight equals that on the left pan, typically through repeated attempts. The physical balance of our model, though, has an intrinsic self-regulating mechanism: it can automatically load (without human intervention) the right pan with an object whose weight equals the one on the left pan. See Figure 2 for a possible implementation of the self-regulating mechanism.

In general, unlike the one in Figure 2, a balance machine may have more than just two pans. There are two types of pans: pans carrying *fixed* weights which remain unaltered by the computation and pans with *variable* (fluid) weights that are changed by activating the filler–spiller outlets. Some of the fixed weights represent the inputs, and some of the variable ones represent outputs. The inputs and outputs of balance machines are, by default, non–negative reals unless stated otherwise. The following steps comprise a typical computation by a given balance machine:

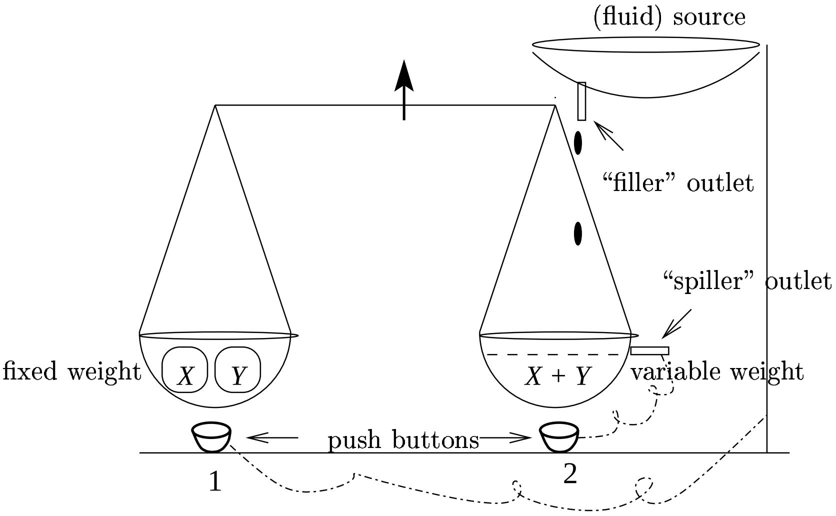


Fig. 2. A self-regulating balance. The *source* is assumed to have (an arbitrary amount of) a fluid–like substance. When activated, the *filler* outlet lets fluid from source into the right pan; the *spiller* outlet, on being activated, allows the right pan to spill some of its content. The balancing is achieved by the following mechanism: the spiller is activated if at some point the right pan becomes heavier than left (i.e., when push button (2) is pressed) to spill off the extra fluid; similarly, the filler is activated to add extra fluid to the right pan just when the left pan becomes heavier than right (i.e., when push button (1) is pressed). Thus, the balance machine can "add" (sum up) inputs x and y by balancing them with a suitable weight on its right: after being loaded with inputs, the pans would go up and down till it eventually finds itself balanced.

- Plug in the desired inputs by loading weights to pans (pans with variable weights can be left empty or assigned with arbitrary weights). This defines the initial configuration of the machine.
- Allow the machine to balance itself: the machine automatically adjusts the variable weights till left and right pans of the balance(s) become equal.
- Read output: weigh fluid collected in the output pans (say, with a standard weighting instrument).

See Figure 3 for a schematic representation of a machine that adds two quantities. We wish to point out that, to express computations more complex than addition, we would require a combination of balance machines such as the one in Figure 2. Section 3 gives examples of more complicated machines.

3 Computing with Balance Machines: Examples

In what follows, we give examples of a variety of balance machines that carry out a wide range of computing tasks—from the most basic arithmetic operations to

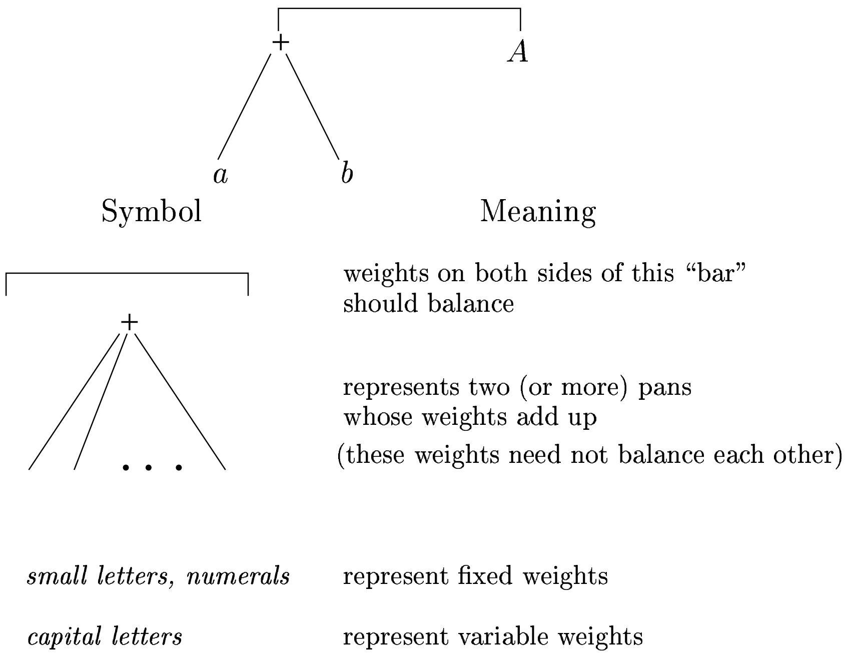

+
A
a
b
Symbol
Meaning
weights on both sides of this "bar" should balance
+
represents two (or more) pans whose weights add up
(these weights need not balance each other)
small letters, numerals
represent fixed weights
capital letters
represent variable weights

Fig. 3. Schematic representation of a simple balance machine that performs addition.

solving linear simultaneous equations. Balance machines that perform the operations *increment*, *decrement*, *addition*, and *subtraction* are shown in Figures 4, 5, 6, and 7, respectively. Legends accompanying the figures give details regarding how they work.

Balance machines that perform *multiplication by 2* and *division by 2* are shown in Figures 8, 9, respectively. Note that in these machines, one of the weights (or pans) takes the form of a balance machine.[2] This demonstrates that such a recursive arrangement is possible.

We now introduce another technique of constructing a balance machine: having a "common" weight shared by more than one machine. Another way of visualizing the same situation is to think of pans (belonging to two different machines) being placed one over the other. We use this idea to solve a simple instance of linear simultaneous equations. See Figures 10 and 11 which are self–explanatory.

An important property of balance machines is that they are *bilateral computing* devices. See [1], where we introduced the notion of bilateral computing. Typically, bilateral computing devices can compute both a function and its (partial) inverse using the same mechanism. For instance, the machines that increment

[2] The weight contributed by a balance machine is assumed to be simply the sum of the individual weights on each of its pans. The weight of the bar and the other parts is not taken into account.

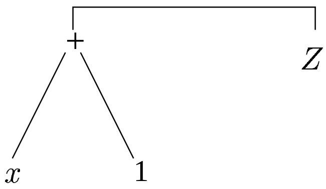


Fig. 4. Increment operation. Here x represents the input; Z represents the output. The machine computes *increment*(x). Both x and '1' are fixed weights clinging to the left side of the balance machine. The machine eventually loads into Z a suitable weight that would balance the combined weight of x and '1'. Thus, eventually $Z = x + 1$, i.e., Z represents *increment*(x).

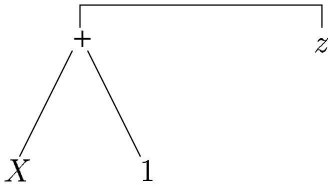


Fig. 5. Decrement operation. Here z represents the input; X represents the output. The machine computes *decrement*(z). The machine eventually loads into X a suitable weight, so that the combined weight of X and '1' would balance z. Thus, eventually $X + 1 = z$, i.e., X represents *decrement*(z).

and decrement (see Figures 4 and 5) share exactly the same mechanism, except for the fact that we have swapped the input and output pans. Also, compare machines that (i) add and subtract (see Figures 6 and 7) and (ii) multiply and divide by 2 (see Figures 8 and 9).

Though balance machines are basically analog computing machines, we can implement Boolean operations (AND, OR, NOT) using balance machines, provided we make the following assumption: *There are threshold units that return a desired value when the analog values (representing inputs and outputs) exceed a given threshold and some other value otherwise.* See Figures 12, 13, and 14 for balance machines that implement logical operations AND, OR, and NOT respectively. We represent *true* inputs with the (analog) value 10 and *false* inputs with 5; when the output exceeds a threshold value of 5 it is interpreted as *true*, and as *false* otherwise. (Instead, we could have used the analog values 5 and 0 to represent *true* and *false*; but, this would force the AND gate's output to a negative value for a certain input combination.) Tables 1, 2, and 3 give the truth tables (along with the actual input/output values of the balance machines).

4 Universality of Balance Machines

The balance machine model is capable of *universal discrete* computation, in the sense that it can simulate the computation of a practical, general purpose digital

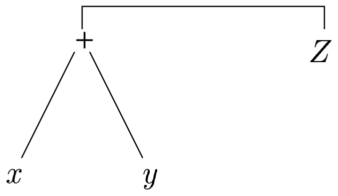


Fig. 6. Addition operation. The inputs are x and y; Z represents the output. The machine computes $x + y$. The machine loads into Z a suitable weight, so that the combined weight of x and y would balance Z. Thus, eventually $x + y = Z$, i.e., Z would represent $x + y$.

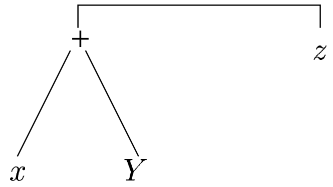


Fig. 7. Subtraction operation. Here z and x represent the inputs; Y represents the output. The machine computes $z - x$. The machine loads into Y a suitable weight, so that the combined weight of x and Y would balance z. Thus, eventually $x + Y = z$, i.e., Y would represent $z - x$.

computer. We can show that the model allows us to construct those primitive (hardware) components that serve as the "building blocks" of a digital computer: logic gates, memory cells (flip-flops) and transmission lines.

1. **Logic gates**
   We can construct AND, OR, NOT gates using balance machines as shown in Section 3. Also, we could realize any given Boolean expression by connecting balance machines (primitives) together using "transmission lines" (discussed below).
2. **Memory cells**
   The weighting pans in the balance machine model can be viewed as "storage" areas. Also, a standard S–R flip-flop can be constructed by cross coupling two NOR gates, as shown in Figure 15. Table 4 gives its state table. They can be implemented with balance machines by replacing the OR, NOT gates in the diagram with their balance machine equivalents in a straightforward manner.
3. **Transmission lines**
   A balance machine like machine (2) of Figure 16 that does nothing but a "copy" operation (copying whatever is on left pan to right pan) would serve both as a transmission equipment, and as a *delay* element in some contexts.

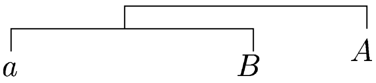


Fig. 8. Multiplication by 2. Here a represents the input; A represents the output. The machine computes $2a$. The combined weights of a and B should balance A: $a + B = A$; also, the individual weights a and B should balance each other: $a = B$. Therefore, eventually A will assume the weight $2a$.

(The pans have been drawn as flat surfaces in the diagram.) Note that the left pan (of machine (2)) is of the fixed type (with no spiller–filler outlets) and the right pan is a variable one.

Note that the computational power of Boolean circuits is equivalent to that of a Finite State Machine (FSM) with *bounded* number of computational steps (see Theorem 3.1.2 of [4]).[3] But, balance machines are "machines with memory": using them we can build not just Boolean circuits, but also memory elements (flip-flops). Thus, the power of balance machines surpasses that of mere bounded FSM computations; to be precise, they can simulate any general *sequential circuit.* (A sequential circuit is a concrete machine constructed of gates and memory devices.) Since any finite state machine (with bounded or unbounded computations) can be realized as a sequential circuit using standard procedures (see [4]), one can conclude that balance machines have (at least) the computing power of *unbounded* FSM computations. Given the fact that any practical (general purpose) digital computer with only limited memory can be modeled by an FSM, we can in principle construct such a computer using balance machines. Note, however, that *notional* machines like Turing machines are more general than balance machines. Nevertheless, standard "physics–like" models in the literature like the Billiard Ball Model[3] are universal only in this limited sense: there is actually no feature analogous to the infinite tape of the Turing machine.

5 Bilateral Computing

There is a fundamental asymmetry in the way we normally compute: while we are able to design circuits that can *multiply* quickly, we have relatively limited success in *factoring* numbers; we have fast digital circuits that can "combine" digital data using AND/OR operations and realize Boolean expressions, yet no fast circuits that determine the truth value assignment satisfying a Boolean expression. Why should computing be easy when done in one "direction", and not so when done in the other "direction"? In other words, why should *inverting* certain functions be hard, while *computing* them is quite easy? It may be

[3] Also, according to Theorem 5.1 of [2] and Theorem 3.9.1 of [4], a Boolean circuit can simulate any T–step Turing machine computation.

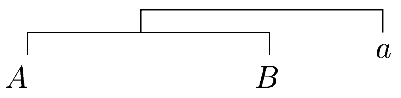


Fig. 9. Division by 2. The input is a; A represents the output. The machine "exactly" computes $a/2$. The combined weights of A and B should balance a: $A+B = a$; also, the individual weights A and B should balance each other: $A = B$. Therefore, eventually A will assume the weight $a/2$.

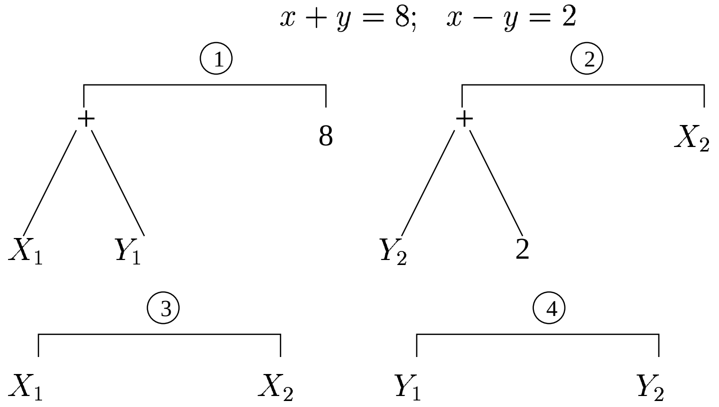


Fig. 10. Solving simultaneous linear equations. The constraints $X_1 = X_2$ and $Y_1 = Y_2$ will be taken care of by balance machines (3) and (4). Observe the sharing of pans between them. The individual machines work together as a single balance machine.

because our computations have been based on rudimentary operations (like addition, multiplication, etc.) that force an explicit distinction between "combining" and "scrambling" data, i.e. *computing* and *inverting* a given function. On the other hand, a primitive operation like *balancing* does not do so. It is the same balance machine that does both addition and subtraction: all it has to do is to somehow balance the system by filling up the empty variable pan (representing output); whether the empty pan is on the right (addition) or the left (subtraction) of the balance does not particularly concern the balance machine! In the bilateral scheme of computing, there is no need to develop two distinct intuitions—one for addition and another for subtraction; there is no dichotomy between functions and their (partial) inverses. Thus, a bilateral computing system is one which can implement a function as well as (one of) its (partial) inverse(s), using the same intrinsic "mechanism" or "structure". See [1] where we have developed bilateral computing systems based on fluid mechanics and have given a mathematical characterization of such systems.

$$x + y = 8; \;\; x - y = 2$$

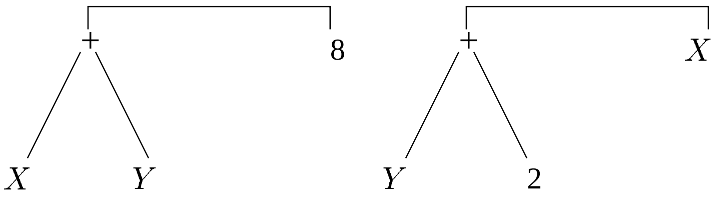


Fig. 11. Solving simultaneous linear equations (easier representation). This is a simpler representation of the balance machine shown in Figure 10. Machines (3) and (4) are not shown; instead, we have used the same (shared) variables for machines (1) and (2).

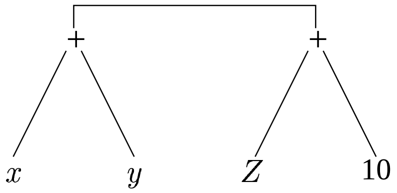


Fig. 12. AND logic operation. x and y are inputs to be ANDed; Z represents the output. The balance realizes the equality $x + y = Z + 10$.

We now show how the classic SAT problem can be solved under a bilateral computing scheme, using balance machines. For the time being, we make no claims regarding the time complexity of the approach since we have not analyzed the time characteristics of balance machines. However, we believe that it will not be exponential in terms of the number of variables (see also [1]). The main idea is this: first realize the given Boolean expression using gates made of balances; then, by setting the pan that represents the outcome of the Boolean expression to (the analog value representing) *true*, the balance machine can be made to automatically assume a set of values for its inputs that would "balance" it. In other words, by setting the output to be *true*, the inputs are forced to assume one of those possible truth assignments (if any) that generate a *true* output. The machine would never balance, when there is no such possible input assignment to the inputs (i.e., the formula is unsatisfiable). This is like operating a circuit realizing a Boolean expression in the "reverse direction": assigning the "output" first, and making the circuit produce the appropriate "inputs", rather than the other way round.

See Figure 17 where we illustrate the solution of a simple instance of SAT using a digital version of balance machine whose inputs/outputs are positive integers (as opposed to reals). Note that these machines work based on the following assumptions:

Table 1. Truth table for AND.

x	y	Z
5 (*false*)	5 (*false*)	0 (*false*)
5 (*false*)	10 (*true*)	5 (*false*)
10 (*true*)	5 (*false*)	5 (*false*)
10 (*true*)	10 (*true*)	10 (*true*)

Table 2. Truth table for OR.

x	y	Z
5 (*false*)	5 (*false*)	5 (*false*)
5 (*false*)	10 (*true*)	10 (*true*)
10 (*true*)	5 (*false*)	10 (*true*)
10 (*true*)	10 (*true*)	15 (*true*)

Table 3. Truth table for NOT.

x	Y
5 (*false*)	10 (*true*)
10 (*true*)	5 (*false*)

Table 4. State table for S–R flip-flop.

S	R	Q	Q'
0	0	previous state of Q	previous state of Q'
0	1	0	1
1	0	1	0
1	1	undefined	undefined

1. Analog values 10 and 5 are used to represent *true* and *false* respectively.
2. The filler–spiller outlets let out fluid only in (discrete) "drops", each weighing 5 units.
3. The maximum weight a pan can hold is 10 units.

6 Conclusions

As said earlier, one of our aims has been to show that all computations can be ultimately expressed using one primitive operation: *balancing*. The main thrust of this paper is to introduce a natural intuition for computing by means of a generic model, and not on a detailed physical realization of the model. We have not analysed the time characteristics of the model, which might depend on how we ultimately implement the model. Also, apart from showing with illustrative examples various possible (valid) ways of constructing balance machines, we have not detailed a formal "syntax" that governs them.

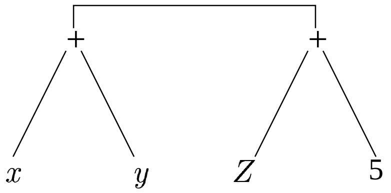


Fig. 13. OR logic operation. Here x and y are inputs to be ORed; Z represents the output. The balance realizes the equality $x + y = Z + 5$.

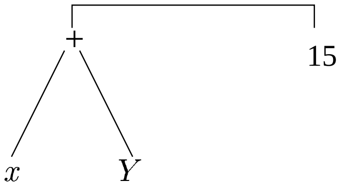


Fig. 14. NOT logic operation. Here x is the input to be negated; Y represents the output. The balance realizes the equality $x + Y = 15$.

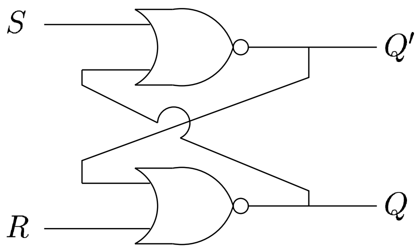


Fig. 15. S–R flip-flop constructed using cross coupled NOR gates.

Finally, this note shows that one of the possible answers to the question "What does it mean to *compute*?" is: "To *balance* the inputs with suitable outputs (on a suitably designed balance machine)."

References

1. J.J. Arulanandham, C.S. Calude, M.J. Dinneen. Solving SAT with bilateral computing, *Romanian Journal of Information Science and Technology* (2003), to appear.
2. J.L. Balcázar, J.Díaz, J. Gabarró. *Structural Complexity I*, Springer–Verlag, Berlin, 1988.
3. E. Fredkin, T. Toffoli. Conservative logic, *Int'l J. Theoret. Phys.* 21 (1982), 219–253.
4. J.E. Savage. *Models of Computation*, Addison–Wesley, Reading, Mass., 1998.

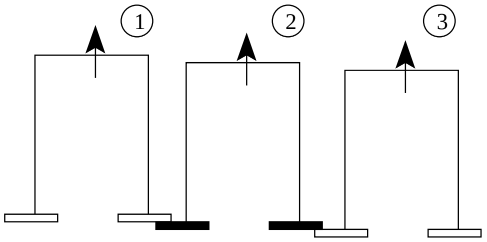
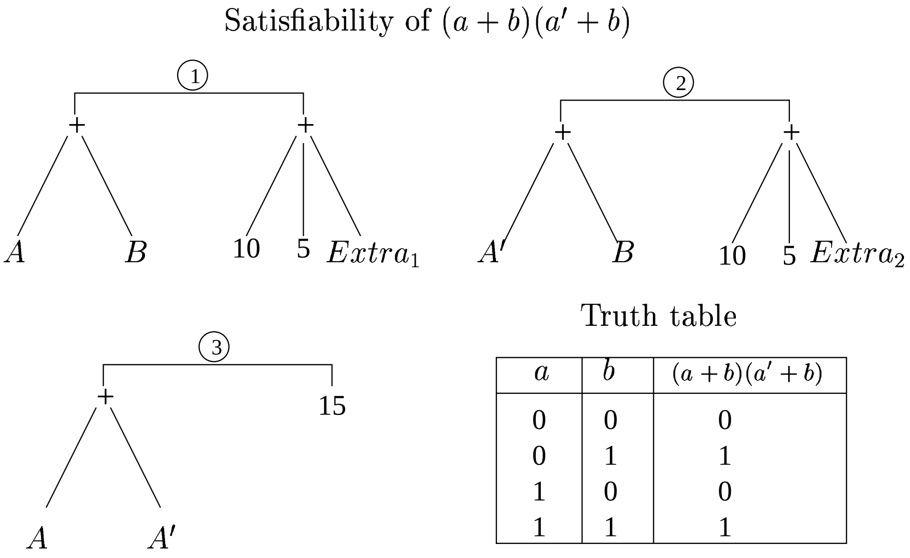


Fig. 16. Balance machine as a "transmission line". Balance machine (2) acts as a transmission line feeding the output of machine (1) into the input of machine (3).



Truth table

a	b	$(a + b)(a' + b)$
0	0	0
0	1	1
1	0	0
1	1	1

Fig. 17. Solving an instance of SAT: The satisfiability of the formula $(a + b)(a' + b)$ is verified. Machines (1), (2) and (3) work together sharing the variables A, B and A' between them. OR gates (labeled 1 and 2) realize $(a + b)$ and $(a' + b)$ respectively and the NOT gate (labeled 3) ensures that a and a' are "complementary". Note that the "output" of gates (1) and (2) are set to 10. Now, one has to observe the values eventually assumed by the variable weights A and B (that represent "inputs" of OR gate (1)). Given the assumptions already mentioned, one can easily verify that the machine *will balance*, assuming one of the two following settings: (i) $A = 5$, $B = 10$, ($Extra_1 = 0$, $Extra_2 = 5$) or (ii) $A = 10$, $B = 10$, ($Extra_1 = 5$, $Extra_2 = 0$). These are the *only* configurations that make the machine balanced. In situations when both the left pans of gate (1) assume 10, $Extra_1$ will automatically assume 5 to balance off the extra weight on the left side. ($Extra_2$ plays a similar role in gate (2).)