# Contents

**1 Introduction** **3**

**2 VLSI circuits** **5**
2.1 Definition of objects . . . . . . 6

**3 Circuit editor** **10**
3.1 Display . . . . . . 10
3.2 Basic editing . . . . . . 11
3.3 Advanced features . . . . . . 11

**4 Circuit file format** **14**
4.1 Description of objects . . . . . . 15

**5 Valid circuits** **17**

**6 Simulation** **19**

**7 Automated testing** **24**
7.1 The AUCH$^{TM}$ language . . . . . . 24
7.1.1 Stop . . . . . . 27
7.2 Examples . . . . . . 27

**8 Circuit images** **29**

**9 Circuit optimization** **31**

**10 High-level circuit description** **32**
10.1 HLOOCH$^{TM}$ files . . . . . . 32
10.1.1 Subcircuits . . . . . . 36
10.2 Building the circuits . . . . . . 37

**11 Generating circuits** **38**

**12 Circuit War** **41**
12.1 Introduction . . . . . . 41
12.2 Objective . . . . . . 41
12.3 Definitions . . . . . . 41
12.4 Rules of the game . . . . . . 43
12.5 Turns . . . . . . 43
12.6 Time limits . . . . . . 44
12.7 End of game . . . . . . 44
12.8 Communication protocol . . . . . . 45
12.8.1 Logging in . . . . . . 45

12.8.2 Conducting a turn . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
12.8.3 End of game . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
12.8.4 End of game session . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
12.9 Operating the reference server . . . . . . . . . . . . . . . . . . . . . . . . . . 48
12.9.1 Environment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
12.9.2 Starting the server . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
12.9.3 Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
12.10 Rules of the Championship . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
12.10.1 Qualification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

Chapter 1

# Introduction

The single most significant advance in the history of computers so far was the introduction of integrated circuits. This technology allows us to put more computing power on a single chip than was ever imagined in the era of room-sized computers. A VLSI (Very Large Scale Integration) circuit can contain millions of transistors, therefore designing such circuits is quite a challenging task. In this contest, you will have to develop a VLSI circuit designer system that allows the user to create, edit, simulate, optimize, and generate circuits.

Chapter 2 defines the simplified circuit model used in the contest: we consider 2-layer circuits containing only NAND gates and flip-flops. Knowledge of electronics is not required, but some experience with digital circuits would help.

The most important part of the VLSI designer system is the graphical circuit editor. In Chapter 3 we specify how the editor should work and what editing features should be supported. The circuit editor can save and load circuits. Chapter 4 defines the XML file format for circuit files. In some of the other tasks you are required to implement additional features in the editor (such as simulation, optimization, etc.)

A circuit can contain errors, such as a gate not being connected to power, or the output of two gates being connected. In Chapter 5 you are asked to write a program that will detect such errors in a circuit.

In Chapter 6 your job is to write a circuit simulator. The user sets the values on the inputs of the circuit, and selects which information will be displayed. The program simulates the operation of the circuit, and displays the required information continuously during the simulation.

When we want to test that a circuit indeed works as specified, then we might have to perform a large number of simulations. Chapter 7 asks you to implement a script language that allows the user to automatically run multiple simulations and evaluate the results of these simulations.

In Chapter 8 you have to write a program that converts a circuit into an image file. Your program also has to be able to reverse engineer a circuit: it has to turn an image file into a circuit description.

The production cost of a circuit is largely determined by the length of the wires and the number of gates/flip-flops in the circuit. In Chapter 9 you have to write a program that optimizes a circuit, reducing its cost as much as possible.

Designing a large circuit by hand is impossible, there are too many wires, gates, flip-flops to consider. Chapter 10 defines a high-level circuit description language that gives only the connections between the gates, but it does not specify how the gates are placed or how the wires are routed. Moreover, it allows us to define "subcircuits", and use multiple copies of these subcircuits later when building larger circuits. Your task will be to write a program that turns a high-level circuit description into an actual circuit.

In Chapter 11 the circuit design process becomes even more automated. You have to write a program that can automatically generate circuits for certain specialized task, such as for evaluating arithmetic and Boolean circuits.

Finally, in Chapter 12, you are presented with a less serious (but nevertheless challenging) task: you have to write a program that can play the game “Circuit Wars”. At the end of the contest the programs developed by the different teams will compete in a championship. Your score will be determined by the results of your program in the contest, so be sure to make it as tough as you can!

Good work and have fun!

Chapter 2

# VLSI circuits

In this chapter we define the VLSI circuit model that will be used in the problem set. This model might be somewhat simplified and inaccurate, but our aim is not to consider every aspect of real-world circuit design, but to create fun and challenging problems.

The circuit can be imagined as a grid. Each node of the grid is either empty, or contains a device (a gate or a flip-flop). These devices are connected by wires, the wires can go only along the lines of the grid. The circuit has two layers, therefore the wires can cross each other without actually being connected. If necessary, the two layers can be connected by placing a *via* at some node of the grid. Furthermore, the wires on layer 1 can only go horizontally, while the wires on layer 2 can only go vertically (this is the so-called *Manhattan model*). To simplify the model, we assume that each device is connected to the ground, so we do not have to worry about that.

Figure 2.1 shows an example VLSI circuit.

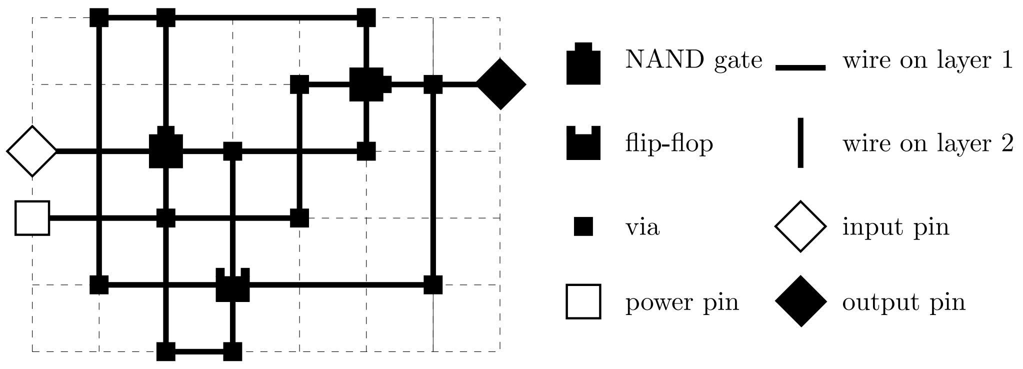


Figure 2.1: A VLSI circuit

Below we summarize the objects that can be found on a circuit. The next section contains more details on these objects.

- horizontal wire segment: wire on layer 1.
- vertical wire segment: wire on layer 2.
- via: a via is a connection between the two layers.
- NAND gate: a logical NAND gate with two data inputs, $a$ and $b$, and an output $c =$ NOT ($a$ AND $b$) (or using mathematical notation: $c = \overline{a \wedge b}$). The gate also has a "power" input: in order to operate the gate has to be connected to the power source (see below and Chapter 5).

- flip-flop: the flip-flop is a device that can store one bit of information. It has two inputs, "data" and "control". If *control* is high, then the value of the data input appears on the output of the flip-flop. If *control* becomes low, then the value of the output "freezes", remaining the same even if the data input changes (i.e., the flip-flop stores the last value of the data input). Like the NAND gates, the flip-flops have to be connected to the power source.
- input and output pins: certain nodes of the grid can be designated as "input pins" or "output pins". When using the circuit, we put some signal on the input pins and we expect that the circuit produces some result on the output pins.
- power pin: as described above, the gates and the flip-flops need power and so have to be connected to the power pin.

You have to write a graphical VLSI circuit editor program that allows the user to view, edit, and design VLSI circuits (see Chapter 3). In Chapter 6 you are asked to write a program that simulates how a circuit works with the given inputs. In order to simplify the problem, the simulation should be a *discrete time simulation.* That is, at each time step every gate and flip-flop has a current state, and the state of a gate or flip-flop in the next time step depends only on the state of its inputs (see below for more details).

## 2.1 Definition of objects

In this section we define those objects that can be found in a circuit and how they will work during the simulation.

### Wires

Each line segment of the grid is either a wire or not. If the line segment is horizontal, then the wire is on layer 1, if it is vertical, then the wire is on layer 2. The wires on the two layers do not touch each other, therefore a horizontal wire can cross a vertical wire at a node on the grid (see Figure 2.2a). In the simulation we assume that the signals are propagated

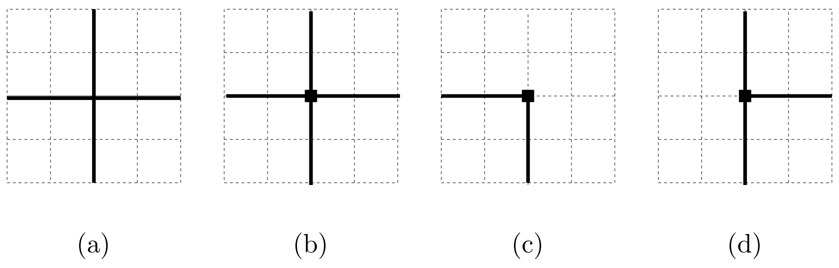


Figure 2.2: (a) A horizontal and a vertical wire can cross each other without being connected, (b) or they can be joined with a *via.* (c) If a wire turns, then it has to go to the other layer using a via. (d) Similarly, if a vertical wire has a horizontal branch, then a via must be used.

instantaneously through the wires (in the real world it takes some time for an electric signal to reach the other end of the wire). For example, assume that the output of gate $A$ is connected by a wire to the input of gate $B$. If at time step $i$ gate $A$ sends a signal down the wire, then we assume that regardless of the length of the wire, the signal reaches the input of gate $B$ before we begin the simulation of time step $i + 1$.

### Vias

Any node of the grid (which is not a NAND gate, flip-flop or pin) can contain a via. A via connects the two layers of the circuit, thus it connects the wires that go through the node. Since layer 1 contains only horizontal wires and layer 2 contains only vertical wires, a via is required if a wire turns (see Figure 2.2c): a horizontal wire on layer 1 has to go to layer 2 using a via if it wants to go vertically.

## NAND gates

A NAND gate has two inputs, $a$ and $b$, and one output, $c$. The output is the logical NAND function of the two inputs: $c = \overline{a \wedge b}$. The following table shows the value of $c$ depending on the inputs, $a$ and $b$:

| input $a$ at step $i$ | input $b$ at step $i$ | $c = a$ NAND $b$ at step $i+1$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

It is well-known that every boolean function can be expressed by NAND gates.

The signal on the output of a NAND gate does not appear instantaneously as it takes one time step for the gate to calculate the correct output. The value of the output of the gate at time step $i+1$ is determined by the values of the inputs at time step $i$. The output at time step $i+1$ does not depend on the inputs at time step $i+1$ (the inputs at time step $i+1$ will determine the value of the output at time step $i+2$).

As shown on Figure 2.3, the NAND gate can be rotated so the output can leave the gate in any of the four possible directions. On the diagram, the small "head" of the gate shows the direction of the output. The power input is always opposite to the output. The two data inputs enter on the remaining two sides of the gate, input $a$ is left (counterclockwise), input $b$ is right (clock-wise) to the output. The power input has to be connected to the power pin of the circuit by a wire, otherwise the gate will not work.

If the power pin is connected to one of the inputs of the NAND gate, then this input will always have the value 1.

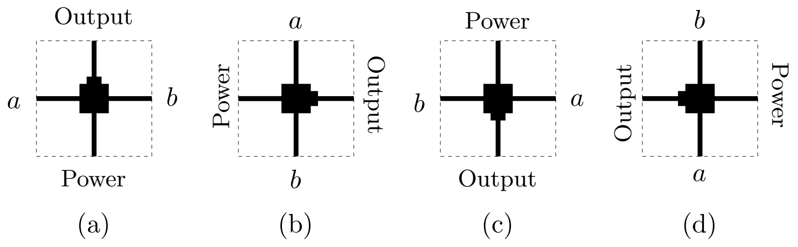


Figure 2.3: The four different positions of the NAND gate. The output wire leaves the gate (a) to Up, (b) to Right, (c) to Down, (d) to Left.

## Flip-flops

A flip-flop has one output and two inputs *data* and *control*. The value of the output at time step $i+1$ is determined by the values of the inputs at time step $i$. The following table summarizes how the output of the flip-flop is determined. For each possible combination of the values of the control input, data input, and the output of the flip-flop we show what will be the value of the output at the next time step.

| Control at step $i$ | Data at step $i$ | Output at step $i$ | Output at step $i+1$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

That is, if control is 0, then the output is the same as in the previous time step (the flip-flop stores a bit of information). If control is 1, then the new state of the flip-flop is set by the data input.

Like the NAND gate, the flip-flop can be rotated and the output can leave the gate in any of the four directions (see Figure 2.4). On the diagram, the small "mouth" of the flip-flop shows the direction of the output. In order to operate, the flip-flop has to be connected to the power source. The power input is always opposite to the output. The data and control input wires enter on the remaining two sides: data input is in the clockwise direction to the output, while the control input is in the counterclockwise direction. The power input has to be connected to the power pin of the circuit by a wire, otherwise the flip-flop will not work.

If the power pin is connected to the data or control input of the NAND gate, then this input will always have the value 1.

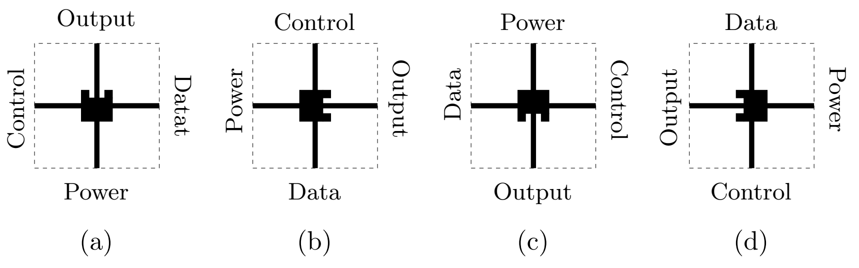


Figure 2.4: The four different positions of a flip-flop. The output wire leaves the gate (a) to Up, (b) to Right, (c) to Down, (d) to Left.

## Input and output pins

In a circuit there are some special nodes on the grid which will be called "input pins" and "output pins". A circuit can have multiple input pins and multiple output pins. When the circuit is simulated (see Chapter 6), the user can set the values appearing on the input pins. During the simulation, the values appearing on the output pins are continuously displayed.

An input or output pin acts as a via, it connects the two layers of the circuit. Therefore it is possible that both a horizontal and a vertical wire starts from an input pin.

Each input/output pin has a name, which can be a character sequence of up to 20 characters in length. The name can only contain the characters 'a'-'z', 'A'-'Z', and '0'-'9'. The names has to be unique, there cannot be two pins with the same name.

## Power pin

The power input of each NAND gate and flip-flop has to be connected by a wire to the power pin, otherwise it will not work. Note that the power input has to be connected directly to the power pin. That is, a gate will not work if its power input is connected to the output of some other gate (even if that other gate receives power).

A power pin acts as a via, connecting the two layers of the circuit. Therefore it is possible for both a horizontal and a vertical wire to start from a power pin. A circuit can contain only one power pin.

In Chapter 5 you are asked to check whether each power input is connected to the power pin, and in Section 3.3 you have to connect each power input to the power pin, minimizing the total length of the wires used. When simulating a circuit (Chapter 6), it should be verified that the circuit is correct and each gate is connected to the power, otherwise we cannot start the simulation. (Chapter 5 gives more details on when we say that a circuit is “correct”.)

If the power pin is connected to the input of a NAND gate or flip-flop, then it acts as a constant value 1.

Chapter 3

# Circuit editor

In the circuit editor program the user can design and edit VLSI circuits using a graphical user interface. The editor should be user-friendly and easy to use with a mouse. In this chapter we define how the editor displays the circuit, and what editing features it has. The circuit editor is the central, most important part of this contest. In the further chapters you will be asked to give additional functionality to the editor: checking the circuit (Chapter 5), simulating the circuit (Chapter 6), optimizing the circuit (Chapter 9), etc.

## 3.1 Display

The editor should display the circuit in a format that is similar to Figure 2.1. The background is white, the grid lines should be displayed in light gray and the wires in black. The NAND gates, flip-flops, vias, and input/output/power pins should be displayed using the symbols on Figure 2.1 (see also Chapter 8). Note that these are only guidelines that you don't have to follow these specifications pixel-by-pixel, but it is important that the user should be able to distinguish the different objects on the circuit.

When starting a new circuit, the user sets the width and the height of the circuit. The width is the number of grid nodes in a row, the height is a number of grid nodes in a column. Figure 3.1 shows a $3 \times 3$ circuit. The editor cannot place/manipulate objects outside this area, unless the user enlarges the circuit (see "Resize circuit" below).

Your program should be able to manage circuits of size $500 \times 500$, in most of the tests we will use circuits not greater than that. However, to receive maximum score in the contest, your program should be able to handle circuits of size up to $20000 \times 20000$ (see the end of the chapter for details on scoring). In any case, it can be assumed that there are no more than 200000 objects in a circuit.

The name of the input/output pins should be displayed somewhere near the pin. The user can select whether these names are displayed/hidden. Similarly, the light gray grid lines can be also displayed/hidden.

The editor should support several different zoom levels. At the highest level of magnifi-

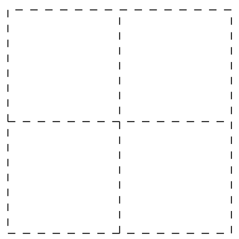


Figure 3.1: A circuit with width 3, height 3, and area 9.

cation the distance between the grid lines should be at least 50 pixels. At the lowest level of magnification the distance should be at most 2 pixels (that is, there are 2 white pixels between two adjacent grid line). When the magnification is very low it might not be possible to accurately recognize the objects and the pin names in the circuit. However, even at the lowest level of magnification, the user should be able to tell where the objects are and where the wires are (even if the type of the different objects cannot be distinguished).

If the circuit does not fit into the screen at the current zoom level, then the user should be able to scroll the visible area of the circuit. The scrolling should be smooth and pleasant to use, but you do not have to make extreme efforts to achieve hyper-smooth, videogame-like scrolling.

When the user moves the mouse over some object/grid node, then the editor should display the row/column number of the selected grid node.

## 3.2 Basic editing

The editor should support the following basic editing functions:

- **Add object:** add a new NAND gate/flip-flip/via/pin to the grid node pointed by the user. In the case of a NAND gate or flip-flop, the user can also set the direction of the object, i.e., the direction the output leaves the object (see Figures 2.3 and 2.4). Note that there can be at most one power pin in a circuit, then cannot add more than one power pins.
- **Edit pin name:** sets the name of the selected input/output pin. The name can be an arbitrary character sequence of up to 20 characters in length, and it can contain only the characters 'a'-'z', 'A'-'Z', and '0'-'9'.
- **Add wire:** add a new wire segment to the grid segment pointed by the user.
- **Add long horizontal/vertical wire:** the user selects two grid nodes, and the editor connects the two grid nodes by a wire. The two grid nodes selected by the user should be either in the same row or in the same column, and the wire should be fully horizontal or fully vertical. If it is not possible to connect the two grid nodes (because something occupies the row/column between the two nodes), then an error message should be presented to the user.
- **Delete:** the object or wire segment selected by the user is deleted from the circuit.
- **Delete region:** every object and wire in the rectangular region selected by the user is deleted.
- **Copy/move region:** the user selects a rectangular region and a destination position and the contents of the selected region is copied/moved to the destination position. If copying a region creates two power pins, then one of them should be deleted.
- **Resize circuit:** the circuit has a width and height and the user can add objects only in this region. The user can set a new size to the circuit, if it is smaller than the original size (in one dimension), then a part of the circuit is lost, if it is larger than the original size, then new, empty regions are added to the circuit.

## 3.3 Advanced features

The following advanced editing features should be implemented in your program:

### Statistical information

If the user requests it, the program should display the following information:

- **Number of objects:** the number of gates/flip-flops/pins/vias should be displayed, separately for each type.
- **Total length of wires:** total length of wires in the circuit, 1 unit is the distance between two adjacent grid nodes.
- **Size of the circuit:** the width, the height, and the area of the circuit. The area of the circuit is the number of grid nodes contained in it, i.e., the area equals the width multiplied by the height. Figure 3.1 shows a circuit whose area is 9.

### Overview image

The program should be able to display an overview image of the circuit that fits within the screen. Of course, the user might not be able to see all the details on such an overview image, but some meaningful representation has to be given. Draw all the wires and all the gates/flip-flops, but do not draw the gridlines.

### Connecting objects

The user selects two objects and the editor connects them by a wire. Unlike in the add long wire feature, here the two objects do not have to be in the same row or column. Therefore the wire might have to turn; possibly several times, if lots of objects "block" the area between the two selected objects.

If one of the selected objects is a NAND gate or flip-flop, then the user also has to select one side of the object (up, down, left or right), and the wire has to connect to that side of the gate/flip-flop. If a wire is already connected to that side of the gate/flip-flop, then the user should be notified that the connection cannot be made. If the selected object is a via or pin, then the new wire can connect to it from any of the four directions.

The new wire cannot touch any objects other than the two selected by the user, and it cannot touch any wire already in the circuit. However, it is possible that the wires cross another wire already in the circuit without touching it, as on Figure 2.2a. The wire has to connect directly to the selected objects, it is not an acceptable solution to use a wire already connected to the object.

If the two objects can be connected, then the program should add this wire to the circuit, and display the length of the wire and the number of vias used by the wire. The program should find an optimal wire routing. More precisely, the user can select whether he wants to find a connection of minimal length, or a connection that uses as few vias as possible (for example, it is possible that there is a connection of length 100 that uses 13 vias, and a connection of length 170 that uses 8 vias; if the user wants to minimize the length, then the first solution should be presented, and if the user wants to minimize the number of vias, then the second solution should be presented.)

### Connecting to power

As described in Chapter 2, each NAND gate/flip-flop has to be connected to the power pin. If a gate/flip-flop is not connected to the power pin, then the circuit is invalid, and it cannot be simulated (see Chapter 5). This feature connects the power input of all the gates/flip-flops that do not yet receive power from the power pin. If there is no power pin in the circuit, then this operation cannot be performed.

The program has to add wires and vias to the circuit in such a way that each gate/flip-flop receives power. The new wires have to connect the power input either to the power pin or to some wire in the circuit already connected to the power pin. The new wires cannot touch any other objects, and cannot touch any wire not connected to the power pin. Each new wire and via has to be connected to the power pin. When connecting the gates/flip-flops to power, the position of the power pin cannot be changed, and the objects and wires already in the circuit cannot be moved or deleted.

The program should display the total length of the new wires ($\ell$) and the total number of new vias used ($n_v$). The program should find a solution the cost of which is minimal, and where the cost is calculated as

$$cost = \ell + 3n_v.$$

| | |
|---|---|
| Display | **30 points** |
| **Basic editing** | |
| Add object | **6 points** |
| Edit pin name | **2 points** |
| Add wire | **2 points** |
| Add long wire | **2 points** |
| Delete | **2 points** |
| Delete region | **2 points** |
| Copy/move region | **2 points** |
| Resize circuit | **2 points** |
| **Advanced features** | |
| Statistical information | **5 points** |
| Overview image | **5 points** |
| Connecting objects | **20 points** |
| Connecting to power | **20 points** |

Chapter 4

# Circuit file format

Your program has to be able to read and write circuit definition files. A circuit definition file is an XML file that contains the description of a VLSI circuit. Don't worry if you are not familiar with XML format, it will be easy to understand the syntax from the examples of this document.

The XML file should start with the XML declaration:

```
<?xml version="1.0"?>
```

The root element of the circuit definition file is `circuit`, hence the whole file is included in a `circuit` element. It has two mandatory attributes, `rows` and `columns`, that give the number of rows and columns of the grid.

```
<?xml version="1.0"?>
<circuit rows="40" columns="40">
...
</circuit>
```

The circuit has a short name and a longer description which can be an arbitrary text:

```
<?xml version="1.0"?>
<circuit rows="40" columns="40">
<name>8-bit adder</name>
<description>
This circuit adds two 8-bit numbers and the output is a 9-bit
number.
</description>
...
</circuit>
```

Don't forget that element and attribute names are *case sensitive* in XML, and the quotes are mandatory around attribute values! The order of the attributes are arbitrary. The end of the line could be either LF or CR LF, your program should handle both cases. The XML file can contain comments:

```
<?xml version="1.0"?>
<!--

  This is just a comment
-->
<circuit rows="40" columns="40">
...
</circuit>
```

## 4.1 Description of objects

Below we summarize the XML elements corresponding to the object of the circuit. All of the elements have empty content, hence they are closed by the characters '`/>`', and no end tag is required.

### NAND gates

The XML element `NAND` corresponding to a NAND gate has three attributes: the row number (`row`), the column number (`col`), and the direction (`dir`). Row 1 is the topmost row of the circuit, column 1 is the leftmost column of the circuit. Direction can be U, D, L or R (for up, down, left or right), it is the direction in which the output leaves the gate.

```
<NAND row="12" col="20" dir="U"/>
<NAND row="13" col="21" dir="U"/>
<NAND row="21" col="48" dir="D"/>
<NAND row="42" col="49" dir="L"/>
```

### Flip-flops

The XML element `ff` corresponding to a flip-flop has three attributes: the row number (`row`), the column number (`col`), and the direction (`dir`). Row 1 is the topmost row of the circuit, column 1 is the leftmost column of the circuit. Direction can be U, D, L or R (for up, down, left or right), it is the direction in which the output leaves the flip-flop.

```
<ff row="22" col="20" dir="R"/>
<ff row="23" col="15" dir="L"/>
```

### Vias

The XML element `via` corresponding to a via has two attributes: the row number (`row`) and the column number (`col`).

```
<via row="22" col="20"/>
<via row="23" col="15"/>
```

### Power pin

The XML element `power` corresponding to the power pin has two attributes: the row number (`row`) and the column number (`col`). There can be only one power pin in a circuit.

```
<power row="15" col="12"/>
```

### Input pins

The XML element `input` corresponding to an input pin has three mandatory attributes: the row number (`row`), the column number (`col`), and the name of the pin (`name`). The name can be up to 20 characters in length, and can contain only the characters '`a`'-'`z`', '`A`'-'`Z`','`0`'-'`9`'.

```
<input row="12" col="22" name="a0"/>
<input row="14" col="22" name="a1"/>
<input row="16" col="22" name="b0"/>
<input row="18" col="22" name="b1"/>
<input row="20" col="22" name="select"/>
```

## Output pins

The XML element `output` corresponding to an output pin has three mandatory attributes: the row number (`row`), the column number (`col`), and the name of the pin (`name`). The name can be up to 20 characters in length, and can contain only the characters '`a`'-'`z`', '`A`'-'`Z`','`0`'-'`9`'.

```
<output row="12" col="80" name="r0"/>
<output row="12" col="80" name="r1"/>
<output row="12" col="80" name="r2"/>
<output row="12" col="80" name="r3"/>
```

## Wires

The XML element `wire` corresponds to a horizontal or vertical wire segment of any length. The element has four attributes `row1`, `col1`, `row2`, `col2` that describe the positions of the two ends of the wire. Either `row1` equals `row2` (the wire segment is horizontal, it is on layer 1), or `col1` equals `col2` (the wire segment is vertical, it is on layer 2). Note that `row1` can be smaller than `row2`, and similarly with the column numbers.

```
<wire row1="10" col1="40" row2="10" col2="32"/>
<wire row1="10" col1="40" row2="28" col2="40"/>
<wire row1="28" col1="40" row2="28" col2="48"/>
<wire row1="28" col1="48" row2="20" col2="48"/>
```

A wire segment defined by a `wire` element can touch other objects only at its two end nodes. If the length of the segment is more than one unit, then there cannot be objects between the end nodes. For example, the following lines *cannot* appear in a valid circuit definition file:

```
<wire row1="10" col1="8" row2="10" col2="15"/>
<via row="10" col="9"/>
<nand row="10" col="13" dir="L"/>
```

| | |
|---|---|
| Save circuit in XML format | **15 points** |
| Load circuit from file | **15 points** |

Chapter 5

# Valid circuits

In Chapter 6 you are asked to write a program that simulates how a given circuit works. However, there are invalid circuits that do not work at all, so they cannot be simulated. In the circuit editor (Chapter 3), the user can ask the program to check the circuit. If one of the following problems occurs in the circuit, then the program should notify the user, showing which type of problem is occurring and where. As described below, the positions of the errors should be shown in a different color. However, if the circuit is large (larger than the screen), then it might be difficult for the user to find the location of the errors. Therefore your program should contain a "go to error" command that scrolls into view the location of at least one error.

Before simulating the circuit the program should check if the circuit is a valid one, and the simulation can be started only if none of the following problems occurs in the circuit.

## No power

As explained in Chapter 2, each NAND gate and flip-flop has to be connected to the power pin (see Figure 5.1a). If there is a NAND gate or flip-flop whose power input is not connected to the power pin by a wire, and the user asks the circuit editor to verify the circuit, then these gates/flip-flops should be shown in a different color. Moreover, the program has to display the number of gates and number of flip-flops not connected to the power pin.

## Connected outputs

The outputs of two NAND gates/flip-flops cannot be connected by a wire, since in this case we cannot determine the state of the wire (see Figure 5.1b). More generally, in a valid circuit the wires cannot connect any two of the following:

- the output of a NAND gate,
- the output of a flip-flop,
- an input pin,
- the power pin.

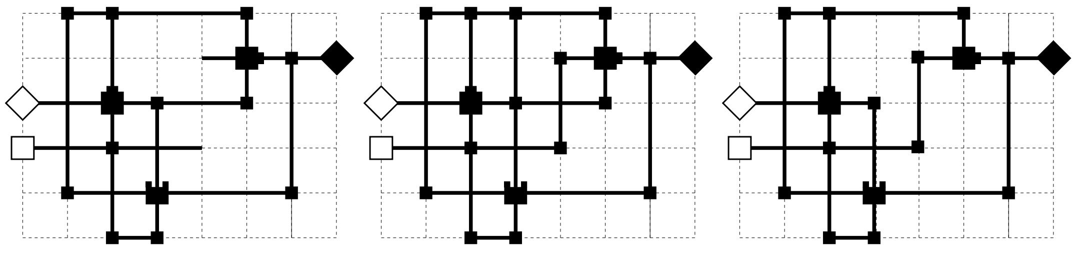
If there are such invalid connections in the circuit, and the user checks the circuit in the editor, then the program should show these invalid connections (wires) in a different color.

## Floating inputs

If the inputs of a NAND gate/flip-flop/output pin is not connected to some signal, then the value appearing on the input is undefined, and we cannot simulate that gate/flip-flop/output pin (see Figure 5.1c). More precisely, an input of a gate/flip-flop/output pin "floats", if it is not connected to any of the following:

- the output of a NAND gate,
- the output of a flip-flop,
- an input pin,
- the power pin.

If there are gates/flip-flops/output pins with floating input, and the user checks the circuit in the editor, then the program should show these gates/flip-flops/output pins in a different color.



(a) (b) (c)

Figure 5.1: (a) The NAND gate on the right is not connected to power. (b) The output of the NAND gate on the left and the output of the flip-flop are connected by a wire. (c) The NAND gate on the right has a floating input.

| | |
|---|---|
| No power | **10 points** |
| Connected outputs | **10 points** |
| Floating inputs | **10 points** |

Chapter 6

# Simulation

You have to write a program that simulates how the circuit works with the given input signals. When the simulation starts, each gate and each flip-flop is in state 0, there are non-zero values only on the input pins. The program has to simulate the circuit step-by-step, and it has to display the information requested by the user at each time step.

Before the simulation starts, the user has to set several parameters that will affect how the simulation is executed, and how the results will be displayed. Here we briefly summarize the information supplied by the user, the following section describes these parameters in detail.

- **Input signals:** for each input pin, the user sets whether the value of this pin is 0 or 1 during the simulation.
- **Data items:** the user can define up to 10 data items, each data item is displayed separately during the simulation. A data item can be a single bit (such as the value of an output pin or a flip-flop), or it can be more than one bit that form together an integer number (for example, the user might want to display the value of the 4 bit number appearing on output pins 1–4).
- **Format of data items:** the data items can be displayed as a number (in binary, decimal, or hexadecimal format), or as an ASCII character.
- **Numerical display:** The standard way of displaying the results of the simulation is to display the results row-by-row, with one time step in each row.
- **Graphical display:** Additionally, your program has to be able to plot the results graphically, where the horizontal axis is the time, and the vertical axis is the value of the data item. The different data items should be drawn on separate graphs. Moreover, there is also a plotter mode where a moving point is displayed whose $x$- and $y$-coordinates are given by a data item (see below for details).
- **Smart simulation:** If smart simulation is turned on in numerical display mode, then you do not have to display the data items for each simulation step, only for those steps where a data item is changed.
- **Log file:** The user might want to save the results of a simulation to a file, which your program must allow.

## Input signals

For each input pin, the users sets the value of the pin (0 or 1) during the simulation. The value of an input pin never changes during a simulation.

## Data items

The user can define up to 10 data items. Each data item contains 1 to 8 bits. Each bit can be one of the following:

- The current value of an output pin.
- The current value of a gate.
- The current value of a flip-flop.
- The current value of a wire segment.

For example, the user might specify that data item 1 is the value of output pin 3; data item 2 is a 3-bit number composed of the value of output pins 10–12; data item 3 is the current value of a given gate; and data item 4 is an 8-bit number whose bits are determined by 8 flip-flops in the circuit.

When a new data item is defined, the user sets how many bits belong to this data item. Then for each bit the user either selects the name of an output pin, or selects a gate/flip-flop/wire segment by clicking on it in the circuit. Note that it should be possible for a data item to contain bits of different types: for example, it is possible that bit 0 of a data item is the value of output pin 5; bit 1 is the value of a flip-flip, and bit 2 is the value of a wire segment. When the user defines the bits of a data item, it should be clearly indicted which bit is the highest (most significant, MSB), and which bit is the lowest (least significant, LSB) in the data item.

## Display format

Separately for each data item, the user can select one of the following display formats:

- binary format (e.g., 101010),
- decimal format (e.g., 42),
- hexadecimal format (e.g., 2a),
- ASCII format (e.g., '*').

In ASCII format the data item should be displayed as a single character, using standard ASCII character encoding. However, you should display the data item only if its value is not less than 32 and not greater than 127. If the value of the data item does not fall into this range, then do not display anything for that data item.

## Numerical display

In numerical display mode, you have to display one row for each step of the simulation. The first entry in each row is the number of simulation steps (in decimal format) elapsed since the beginning of the simulation. This number is followed by the values of the data entries *after* the given number of simulation steps. Therefore in the first line the first entry is 1, and the first line contains the values after the first step of the simulation. The format of the data items should be as specified by the user (binary, decimal, hexadecimal, or ASCII). If the list of data items is so long that they are wider than the screen, then the user should be able to scroll the results horizontally.

The user can choose to run the simulation in one of the following three ways:

- **One-step-at-a-time:** When the simulation is started, only the first step is simulated, the result is printed in the first row of the output. Then the program waits for an action from the user (pressing a key, clicking a button), simulates the second step, prints the result below the first row, waits again, and so on. When the screen (or window) is full, the results should be scrolled up one row, the next result is printed again into the last row of the screen.

- **One-screen-at-a-time:** This should be similar to the previous mode, but the program does not wait after each step of the simulation. When the screen (or window) is full, the program waits for an action from the user, clears the output, and continues the simulation. After clearing the output, the next result will be printed starting in first row.
- **Continuous mode:** Again similar to the first mode, but the program does not wait after each step of the simulation. When the screen (or window) is full, the results are scrolled up one row (without waiting for any user action). The user can control the speed of the simulation (even during the simulation). The slowest simulation should be about 1 step per second, the fastest should be as fast as the computations permit. The results scrolled out at the top of the screen are lost forever, you do not have to implement a "scrollback" feature (but you can, if you want).

## Graphical display

In graphical display mode your program has to draw a separate graph for each data item, showing how the value of the data item changes during the simulation. The horizontal axis represents the time, the user can set how many pixels correspond to each time step (between 4 and 40 pixels, and it is the same for each data item). The vertical axis represents the value of the data item. The scale of the vertical axis is different for each data item, and it should be possible for the user to set the scale of the vertical axis individually for each data item. If the graphs do not fit into the screen vertically, then the user should be able to scroll the display. The width of the line should be 1 pixel. Draw the line continuously: draw vertical lines when the value of the data item changes, as shown on Figure 6.1.

You should implement the three options, One-step-at-a-time, One-screen-at-a-time and Continuous, in graphical display mode as well.

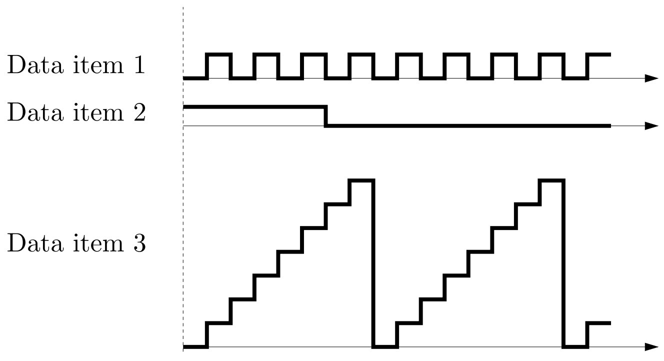


Figure 6.1: Graphical display of two data items. The first two data items consistin of one bit only, their values are 0 or 1. The third data item consists of 3 bits, its value is between 0 and 7.

## Plotter mode

In plotter mode the user selects two data items, and the values of these items will be interpreted as the horizontal and vertical coordinates of a moving object. There are two modes: either the moving object leaves a trace (all the points are shown that were visited by the moving object) or only the current position of the object is shown. The user should be able to zoom the display, independently for the horizontal and the vertical axis. To help the user determine the position of the object, the program should display a grid (see the Figure 6.2).

You have to implement the One-step-at-a-time and the Continous modes in plotter mode.

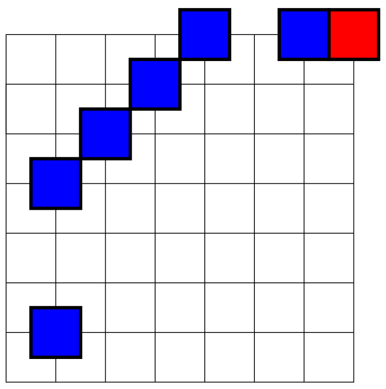


Figure 6.2: Plotter mode. The current position of the moving object is shown in red, the positions visited are shown blue. The values of the two selected data items where $(1,1)$, $(1,4)$, $(2,5)$, $(3,6)$, $(4,7)$, $(6,7)$, and currently $(7,7)$.

### Smart simulation

In numerical display mode the user should be able to turn on smart simulation. This means that not every step of the simulation is displayed, but only those steps where a data item changed its value.

If smart simulation is turned on in One-step-at-a-time mode, then the program does not wait after each step, it waits only after those steps that were displayed (only after those steps where some data item changed). To help the user understand the output, draw a line between two rows if there are "missing" rows between them, if one or more time steps were skipped.

### Log file

In numerical display mode the user can specify a log file where the results of the simulation are written. Each line of the log file corresponds to one step of the simulation. The log file has to satisfy the following formatting requirements:

- The lines (including the last line) are terminated by a single LF character (decimal code 10).
- Each line begins with the number of steps elapsed since the beginning of the simulation, followed by the values of the data items, separated by tab characters (decimal code 8).
- The numbers appearing in the output do not have extra leading zeros or spaces.
- The hexadecimal numbers in the output use lower case letters.
- Do not print any leading zeros in binary, decimal or hexadecimal format.
- If a data item is printed in ASCII format, and the value is less than 32 or greater than 127, then do not print anything for that data item, not even a space (but do not forget the tab characters separating the data items).

It is very important to follow these requirements, since the results of your simulation will be compared to the reference solution character-by-character! Please have an MD5 checksum generator utility at hand for the judging process.

| | |
|---|---|
| **Parameters** | |
| Input signals | **5 points** |
| Data items | **5 points** |
| Format data items | **5 points** |
| Numerical display | **20 points** |
| Graphical display | **35 points** |
| Smart simulation | **5 points** |
| Log file | **25 points** |

Chapter 7

# Automated testing

The simulator program (Chapter 6) allows us to test a circuit before realizing it in hardware. With the help of the simulator, we can check whether the circuit produces correct output for a given input. However, in the case of a complex circuit, several tests may be required, to ensure that it indeed works according to the specifications. Moreover, in some situations we might want to change the input values dynamically during the simulation. AUCH$^{TM}$ (Automated Universal Circuit Hyperlanguage) helps us in this task. This programming language can be used to define several tests, and to automatically evaluate the results of the tests. Given a circuit and an AUCH$^{TM}$ script, your program has to simulate the circuit using the instructions in the script.

## 7.1 The AUCH$^{TM}$ language

An AUCH$^{TM}$ script is composed of instructions, separated by semi-colons (';'). A line can contain several instructions, and an instruction can be broken into more than one line (a space character can be replaced by any other white space character, CR and LF characters included). The language is case-insensitive in the variable names and keywords ('ELSE' and 'eLSe' are the same). An empty instruction is also a valid instruction, hence it is possible that there is no instruction between two semi-colons.

### Variables

In the AUCH$^{TM}$ language every variable is a 32-bit signed integer. Variable names can be up to 16 characters in length, and can contain only the characters 'a'-'z', 'A'-'Z', '0'-'9', and '_'. The first character of a variable cannot be a digit, and variable names cannot coincide with the keywords defined in this section. Variable names are case-insensitive: 'first_TEST1' and 'FIRST_test1' refer to the same variable. Variables do not have to be declared, and the value of each variable is 0 before its first assignment.

### Numerical expressions

Numerical expressions can be built from decimal constants (like '42' or '-123456'), variables, operators '+', '-', '*', '/', and the parenthesis symbols '(', ')'. Note that '-' is a binary operator, hence it *cannot* be used as '-a+b' or 'a+-b' (however, 'a--1' is a valid expression with the meaning '$a$ minus $-1$'). The operators and the parentheses can be surrounded by white space characters. Some examples of valid numerical expressions: '( (a) +(b))', '(a*(1+(c+2)))-3', '((a1)+( b1 )+(c1)) +(a1-b1-c1)'. The usual precedence and left to right rules apply when the expressions are evaluated. The result is always an integer, and it can be negative. The result of the '/' operator is rounded towards zero (i.e., rounded down if the result is positive, rounded up if the result is negative). The execution of the AUCH$^{TM}$ script is terminated on a division by zero.

There is one more thing that can appear in a numerical expression: a list of input or output pins, enclosed in square brackets ('[', ']'), for example '[I3,I4,O6,O1]'. Its value is a number, whose bits are composed of the values of the listed pins. For example, the value of '[I3,I4,O6,O1]' is a 4 bit number whose bit 3 (highest bit, MSB) takes the value of input pin 3, bit 2 takes the value of input pin 4, bit 1 takes the value of output pin 6, and bit 0 (lowest bit, LSB) takes the value of output pin 1.

## Boolean expressions

A boolean expression is composed of two numerical expressions, separated by a comparison operator. The boolean operators are '=', '!=', '<>', '<', '>', '<=', '=<', '=>', '>='. The operators '!=' and '<>' both mean 'not equals'. The operators can be surrounded by white space characters. Examples: 'a<max', 'a+b >= b*c'.

## Assignments

An assignment instruction starts with a variable name, followed by the symbol '=', and a numerical expression. The numerical expression is evaluated, this value will be the new value of the variable. Examples: 'i=i+1', 'a = (b) + (c)', 'd2 = (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2)'.

Another use of the assignment instruction is to set the value of some input pins. For example the instruction

```
[I5,I4,I10]=a+b
```

calculates the value of the expression 'a+b', interprets it as a 3 bit number, and assigns bit 2 (highest bit) of this value to input pin 5, assigns bit 1 to input pin 4, assigns bit 0 (lowest bit) to input pin 10. In such an assignment, if we set the value of $k$ input pins, then the numerical expression has to be between 0 and $2^k - 1$ (boundaries included). For example, in the above example, 'a+b' cannot be smaller than 0 or larger than 7. If the numerical expression is not in this range, then the execution of the program should be terminated with an error message. As shown in the example, the order of the input pins do not have to be increasing, and it can contain 'gaps' (pin 4 can be followed by pin 10).

## Print

The 'print' statement is followed by one or more items, separated by commas (','). Each item is either a string or a numerical expression. The string should be enclosed in double quotes ('"') and should not contain a line break. A string should be printed as it appears in the source, while the value of a numerical expression should be printed in decimal form, without any leading spaces or zeros. If there are more than one item, then in the output the items should be separated by one space. After the last item, a new line character (LF, decimal code 10) should be written. Example: if the value of variable $a$ is 5, then the instruction 'print "Result:",a+1,a*a' produces 'Result: 6 25'.

## While loops

The only iteration structure in AUCH$^{TM}$ is the while loop. The 'while' statement is followed by a boolean expression, and the body of the loop enclosed in braces '{', '}'. The body of the loop can contain one or more instructions. At the beginning of each iteration, the boolean expression is evaluated, and if it is false, then the loop is terminated, execution is continued with the next instruction after the loop. If the boolean expression evaluates to true, then the body of the loop is executed.

Example: the following program

```
x=1; while x<=4 {print x;x=x+1}
```

outputs

```
1
2
3
4
```

## Conditional statements

The if... then, and the if... then... else conditional structures common in other programming languages can be found in AUCH$^{TM}$ as well. The '`if`' keyword is followed by a boolean expression, which is followed by one or more instructions enclosed in braces '{', '}'. The instructions between the braces are executed only if the boolean expression is true. Example: '`if a<0 {print "Negative:  ",a;a=-1*a}`' tests if $a$ is negative, and if it so, then prints its value, and sets variable $a$ to its absolute value.

The instructions in braces can be followed by the keyword '`else`' and by some additional instructions in braces. In this case, the block after '`else`' is executed only if the boolean expression after '`if`' evaluates to false. Example:

```
if a>0 {print "Positive"} else {if a<0 {print "Negative"}else{print "Zero"}}
```

determines the sign of variable $a$.

## Simulation

The main feature of the AUCH$^{TM}$ language is its ability to interact with the circuit simulator developed in Chapter 6. During the execution of the AUCH$^{TM}$ program, we store the current state of the circuit: the values of the input and output pins, and the values of the gates and the flip-flops. When the execution of the program starts, the value of each pin, each gate, and each flip-flop is 0. The '`simulate x steps`' instruction takes the current state of the circuit, and simulate what happens after $x$ time steps, and sets the state of the circuit accordingly. Here $x$ can be any numerical expression. Since the instruction modify the state of the circuit, performing a '`simulate 1 steps`' instruction 3 times is equivalent to a '`simulate 3 steps`'.

This command can be used to run the simulation not only for a fixed number of steps, but while some condition is true. If the '`simulate`' keyword is followed by '`while`' and a boolean expression, then we simulate the circuit repeatedly, and before each iteration, we evaluate the boolean expression. If it evaluates to *false*, then the simulation is stopped, the execution is continued on the next instruction. If the boolean expression true then the simulation continues. If the expression is false even before this '`simulate while`' instruction, then no simulation is performed. It is possible that the execution of the script enters an infinite loop if the boolean expression will never be false in the simulation.

Example: the instruction '`simulate while [O1,O2,O3]<>a`' simulates the circuit repeatedly, it stops only if the value appearing on the first 3 output pins becomes equal to the value of variable $a$.

## Restarting the simulation

During the execution of the program, we store the current state of the circuit. The '`restart`' instruction resets the state of the circuit: the value of each pin, each gate, and each flip-flop becomes 0. If you perform multiple simulations of the same circuit with different input values, then be sure to reset the state of the circuit between simulations: the circuit might not work as expected if it is started from a state different from the initial state. Do not forget that '`restart`' resets the values on the input pins as well, thus you have to set the input values between the '`restart`' instruction and the '`simulate`' instruction.

### Comment

If the first non-space character of a line is an at sign ('@'), then the line is a comment, and is not interpreted.

### 7.1.1 Stop

The 'stop' instruction terminates the execution of the script.

## 7.2 Examples

Here we give some example AUCH$^{TM}$ scripts to demonstrate the language constructions defined above. Moreover, it will give you some feeling of how this language can be used in testing circuits.

**Example 1.** The following example script tests a circuit that is supposed to be an adder. The circuit should add the 3 bit number on input pins 1–3 and the 3 bit number on pins 4–6, the result should appear on output pins 1-4. The script tries all $2^3 \times 2^3$ possible combination of input values, and simulates the circuit for 100 steps. If the correct answer appears on the output pins after 100 steps, then the script tries the next combination. If the answer is incorrect, then then the script is terminated with error.

```
@ Testing an adder circuit
@ Specification:
@ [O1,O2,O3,O4]=[I1,I2,I3]+[I4,I5,I6]

a=0; while a<=7 {
b=0; while b<=7 {
restart;
[I1,I2,I3]=a; [I4,I5,I6]=b;
simulate 100 steps;
if [O1,O2,O3,O4]<>a+b
{print "Error:",a,"+",b,"=",a+b,"instead of",[O1,O2,O3,O4];stop};
b=b+1
}
a=a+1}
print "Circuit is correct."
```

**Example 2.** This example is similar to Example 1, but here we test a multiplier. The main difference in the script is that we do not simulate the circuit for exactly 100 steps, but we wait until output pin 7 signals that the calculation is ready.

```
@ Testing a multiplier circuit
@ Specification:
@ [O1,O2,O3,O4,O5,O6]=[I1,I2,I3]*[I4,I5,I6]
@ Calculation is ready when O7=1

a=0; while a<=7 {
b=0; while b<=7 {
restart;
[I1,I2,I3]=a; [I4,I5,I6]=b;
simulate while O7=0;
if [O1,O2,O3,O4,O5,O6]<>a*b
{print "Error:",a,"*",b,"=",a*b,"instead of",[O1,O2,O3,O4,O5,O6];stop};
b=b+1
}
a=a+1}
print "Circuit is correct."
```

**Example 3.** The following script tests an “impulse generator”: a circuit that generates the given number of impulses. If the input pins 1–4 define a number $n$, the circuit generates $n$ impulses: output pin 1 turns on/off $n$ times. After the $n$ impulses are generated, pin 2 turns on to indicate that the circuit is ready.

```
@ Testing an impulse generator
@ Specification:
@ [I1,I2,I3,I4]: number of impulses on O1; O2 becomes high when ready

a=0; while a<=15 {
restart;
[I1,I2,I3,I4]=a;
c=0;
while (c<a) {
while [O1]=0 {simulate 1 steps; if [O2]=1 {print "Error";stop}};
while [O1]=1 {simulate 1 steps; if [O2]=1 {print "Error";stop}};
c=c+1;
};
while [O2]=0 {simulate 1 steps; if [O1]=1 {print "Error";stop}};
a=a+1;
}
```

Test will be performed by executing several, increasignly difficult scripts.
Total score: **100 points**

Chapter 8

# Circuit images

In Chapter 4, we defined a text file format for describing VLSI circuit. However, sometimes it is better to store the circuit not as a text file, but as a bitmap image that shows how the circuit looks. Your circuit editor (Chapter 3) has to be able to convert a circuit into an image file. Moreover, the program should be able to read an image file and (if it represents a correct circuit) convert it into a circuit.

If the circuit has $n$ rows and $m$ columns, then you have to generate a black and white image $12m + 20$ pixels wide and $12n + 40$ pixels tall. Pixel $(12i + 4, 12j + 4)$ of the image corresponds to the grid node in the intersection of row $i$ and column $j$ (recall that the numbering of the rows and the columns starts from 1, and the pixel in the upper left corner is $(1, 1)$). Below we describe what has to be drawn for each object that appears in the circuit.

- Wires: for each wire segment, draw a line 1 pixel wide between the pixels that correspond to the two end nodes of the segment.
- Vias: if there is a via at same grid node, then draw a filled rectangle centered on the pixel that corresponds to the grid node. The size of the rectangle should be $5 \times 5$ pixels.
- NAND gates: a NAND gate is represented by a $7 \times 7$ filled rectangle, with a $3 \times 2$ rectangle attached to it (see Figure 8.1b). The small rectangle shows the direction of the output (as in Figure 2.3).
- flip-flops: a flip-flop is represented by a $7 \times 7$ rectangle, with a $3 \times 2$ rectangle "cut out" from one of its sides (see Figure 8.1c). This small mark shows the direction of the output (as in Figure 2.4).
- power pin: the power pin is represented by a $7 \times 7$ empty rectangle (see Figure 8.1d).
- input pins: an input pin is represented by a rotated empty rectangle, as shown in Figure 8.1e.
- output pin: an output pin is represented by a rotated filled rectangle, as shown in Figure 8.1f.

Figure 8.1 shows how these objects look in a bitmap. In Figure 8.2 the same objects are repeated, but it also shows how the wires are connected.

Your program has to follow these specifications carefully, pixel by pixel, otherwise we cannot accept the images produced by your program. The sample files on the CD can help you test the correctness of your program.

When converting an image file into a circuit, then it can be assumed that every pixel of the image is correct, there are no errors, noise or inaccuracies. In the circuit each input pin has to receive a number. Assign the numbers to the input pins in top-to-bottom, left-to-right, row major order. That is, assign number 1 to the pin whose row number is minimal, if there are more than one such pin, then assign number 1 to leftmost among them. The output pins are numbered similarly.

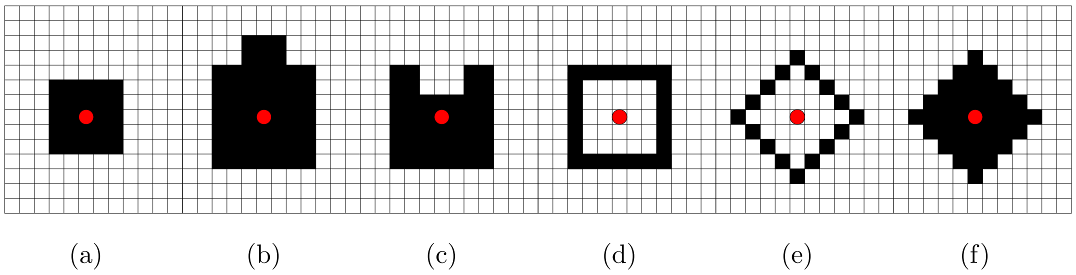


Figure 8.1: The representation of the different objects on the image. (a) A via, (b) a NAND gate, (c) a flip-flop, (d) the power pin, (e) an input pin, (f) an output pin. The red circles do not appear in the image file, they show the pixels corresponding to the nodes of the grid.

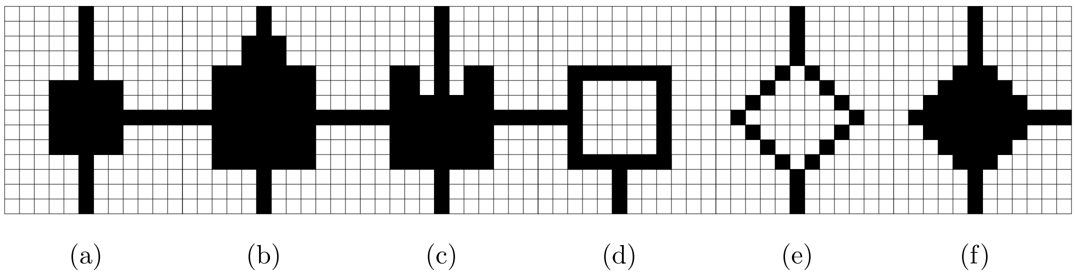


Figure 8.2: The representation of the different objects on the image, with some wires attached to each object. (a) A via, (b) a NAND gate, (c) a flip-flop, (d) the power pin, (e) an input pin, (f) an output pin.

The output should be presented in a 24 bit TGA file. The file format is very simple: the 18 byte header is followed by the BGR ordered color components of the pixels. The header should be the following:

| | |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 2 |
| 3 | 0 |
| 4 | 0 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |
| 9 | 0 |
| 10 | 0 |
| 11 | 0 |
| 12 | low order byte of width |
| 13 | high order byte of width |
| 14 | low order byte of height |
| 15 | high order byte of height |
| 16 | 24 |
| 17 | 32 |

| | |
|---|---|
| Save to image file: | **15 points** |
| Load from image file: | **25 points** |

Chapter 9

# Circuit optimization

When designing a VLSI circuit it is important to keep the number of gates/flip-flops as low as possible. Using lots of wires in the circuit also increases the production costs. Your job is to write a program that optimizes a circuit: the program should produce a circuit that works exactly like the original, but has fewer gates/flip-flops/wires.

During the judging process, you will be presented several circuits with different complexity. Your program has to construct an optimized version **within 30 seconds** for each circuit. Your optimized circuit is simulated with different inputs, and the values appearing on the output pins are compared to the values appearing on the output pins in the original circuit. Your solution is accepted only if it produces exactly the same output as the original circuit.

The optimized circuit will be assigned a cost as follows. The cost depends on the number of NAND gates ($n_{\mathrm{NAND}}$), number of flip-flops ($n_{\mathrm{ff}}$), the total wire length ($\ell$), the number of vias ($n_{\mathrm{via}}$) and the area of the circuit ($A$). The area of the circuit is the number of grid nodes in it, that is, the number of rows multiplied by the number of columns. The cost is calculated by

$$\mathrm{cost} = n_{\mathrm{NAND}} + 4n_{\mathrm{ff}} + 0.01\ell + 0.2n_{\mathrm{via}} + 0.001A \ .$$

For each circuit, we order the solutions of the different teams by increasing cost. The number of points your team receives for a given circuit depends on your position in the list. The team with the lowest cost receives 100% of the maximum points for that circuit, the second 90% of the points and so on (see the table below for more details). From the 11th place on, if your program manages to achieve any optimization (the cost is lower than the cost of the original circuit), then you will get 10% of the maximum points. In case of a tie, the points will be averaged.

| Position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Points received | 100% | 90% | 80% | 70% | 60% | 50% | 40% | 30% | 20% | 10% |

| | |
|---|---|
| Total score of the test cases: | **100 points** |

Chapter 10

# High-level circuit description

When we want to design a circuit for a job, then the most important thing to do is to decide how the NAND gates and the flip-flops are connected, that is, for each object we have to specify where its inputs and its output are connected (the "high-level description of the circuit"). The actual placement of the gates and the flip-flops on the circuit and the exact routes of the wires are less important, since they do not affect the operation of the circuit. Of course, if we want to build the circuit, then we need the coordinates of the objects and the routes of the wires. Your job is to write a program that turns a high-level description of the circuit into a working circuit within the time limit of **30 seconds**.

Designing a very large circuit with thousands of gates and flip-flops is impossible by hand, even if the placement of the objects and the routing of the wires are done automatically by a program. Fortunately, most circuits are well-structured, there are certain types of subcircuits that are repeated multiple times in the circuit. The HLOOCH$^{TM}$ (High-Level Object Oriented Circuit Hyperlanguage) lets us define subcircuits, and gives us a convenient way to reuse a subcircuit multiple times in a circuit. We can even build larger subcircuits from several smaller subcircuits (the nesting can be of arbitrary depth). Your program should be able to turn a HLOOCH$^{TM}$ file into a circuit, that is, it should

- set the place of the input/output/power pins,
- add as many NAND gates and flip-flops to the circuit as necessary,
- connect the gates/flip-flops as described in the file, and
- connect each gate/flip-flop to the power pin.

## 10.1 HLOOCH$^{TM}$ files

A HLOOCH$^{TM}$ file is an XML file that describes one circuit. Additionally, the file can define several subcircuits that are used in the definition of the main circuit.

The HLOOCH$^{TM}$ XML file starts with a standard XML declaration and the rest is contained in a `hlooch` element. The `hlooch` element contains a `circuit` element describing the circuit. Optionally, the `hlooch` element may contain a `definitions` element. The `definitions` element contains one or more `circuit` elements, each element describes a subcircuit, which can be used in the definition of the main circuit. The definition of a subcircuit can use a subcircuit defined previously.

```
<?xml version="1.0"?>
<hlooch>
<definitions>
<circuit id="c1">
...
definition of subcircuit c1
```

```
...
</circuit>
<circuit id="c2">
...
definition of subcircuit c2
can use subcircuit c1
...
</circuit>
</definitions>
<circuit>
...
definition of the main circuit
can use subcircuits c1 and c2
...
</circuit>
</hlooch>
```

As shown in the example, each subcircuit defined in the `definitions` element has a name, which is set by the `id` attribute of the `circuit` element.

The contents of the `circuit` element describe the circuit. It contains the description of the following:

- an optional name for the circuit,
- all the NAND gates,
- all the flip-flops,
- all the inputs,
- all the outputs,
- all the connections between the gates/flip-flops/inputs/outputs.

The connections are specified in the following way. Each input/NAND gate/flip-flop is assigned a unique name. For each gate/flip-flop we have to give the name of the objects connected to its inputs. Similarly, each output has to be connected to some input/gate/flip-flop, which is identified by its name.

In the following we summarize the elements that can appear in a `circuit` element.

## Name of the circuit

The `name` element contains the name of the circuit, and it is optional:

```
<name>8-bit counter</name>
```

## Inputs

There is one `input` element for each input of the circuit. The `input` element has one mandatory parameter `id`, which assigns a unique name to the input.

```
<input id="a0"/>
<input id="a1"/>
<input id="a2"/>
<input id="a3"/>
```

## Output

There is one `output` element for each output of the circuit. The `output` element has one mandatory parameter, `from`, which contains the name of the object that is connected to this output.

```
<output from="gate3"/>
<output from="flip-flop11"/>
```

## NAND gates

One `nand` element corresponds to a NAND gate of the circuit. It has three mandatory attributes: `data1`, `data2`, and `id`. The attributes `data1` and `data2` set the name of the objects connected to the two inputs of the NAND gate. The `id` tag assigns a unique name to the gate.

For example, the following HLOOCH$^{TM}$ file defines a circuit with 4 inputs, 1 output, and 3 NAND gates. Gate `nand1` is connected to the first two inputs, gate `nand2` is connected to the last two inputs, while gate `nand3` is connected to the output of gates `nand1` and `nand3`. Finally, the output of the circuit is the output of gate `nand3`. That is, the circuit computes the value $\overline{(\overline{i1 \wedge i2}) \wedge (\overline{i3 \wedge i4})}$.

```
<?xml version="1.0"?>
<!-- First example -->
<hlooch>
<circuit>
<input id="i1"/>
<input id="i2"/>
<input id="i3"/>
<input id="i4"/>
<output from="nand3"/>
<nand data1="nand1" data2="nand2" id="nand3"/>
<nand data1="i1" data2="i2" id="nand1"/>
<nand data1="i3" data2="i4" id="nand2"/>
</circuit>
</hlooch>
```

Notice that the input of a NAND gate can be a NAND gate defined later. Figure 10.1 shows a possible circuit corresponding to these definitions.

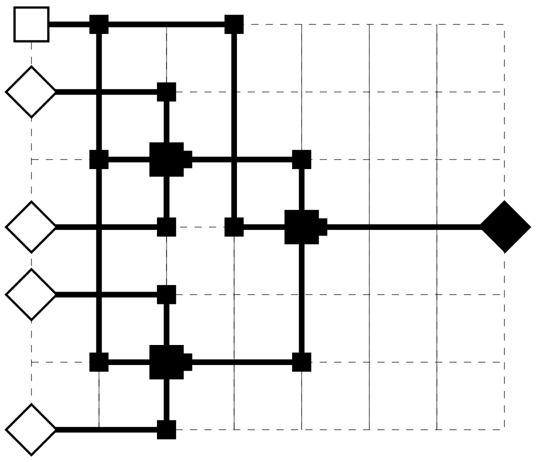


Figure 10.1: A VLSI circuit corresponding to the first example

## Flip-flops

One `ff` element corresponds to each flip-flop of the circuit. It has three mandatory attributes: `data`, `control`, and `id`. The attributes `data` and `control` set the name of the objects connected to the data and control inputs of the flip-flop gate. The `id` tag assigns a unique name to the gate.

For example, the following HLOOCH$^{TM}$ file implements a 4-bit register (a circuit that stores a 4-bit number). If the first input (`control`) is 1, then the register stores the 4-bit number given by inputs `d0`, `d1`, `d2` and `d3`. If the value of `control` is 0, then the stored 4-bit number does not change.

```
<?xml version="1.0"?>
<hlooch>
<!-- Second example -->
<circuit>
<name>4-bit register</name>
<input id="control"/>
<input id="d0"/>
<input id="d1"/>
<input id="d2"/>
<input id="d3"/>
<output from="b0"/>
<output from="b1"/>
<output from="b2"/>
<output from="b3"/>
<ff data="d0" control="control" id="b0"/>
<ff data="d1" control="control" id="b1"/>
<ff data="d2" control="control" id="b2"/>
<ff data="d3" control="control" id="b3"/>
</circuit>
</hlooch>
```

Figure 10.2 shows a possible circuit corresponding to these definitions.

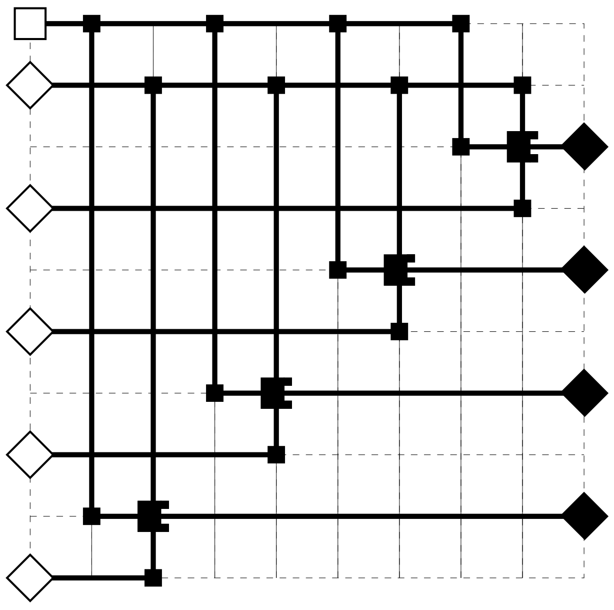


Figure 10.2: A VLSI circuit corresponding to the second example

### 10.1.1 Subcircuits

The circuits defined in the `definitions` element can be used as subcircuits to build larger circuits. The `subcircuit` element can be used to insert a copy of a subcircuit into the circuit. It has one mandatory attribute `id`, which has to be the `id` value of a `circuit` element defined earlier.

The `subcircuit` element contains `input` and `output` elements, whose number has to match the number of inputs and outputs in the definition of the subcircuit. The `input` element has one mandatory attribute `from`, which gives the name of the object connected to this input. The `output` element has one mandatory attribute `id`, which gives a unique name to each output. If an object in the circuit is connected to this output, then this unique name has to be given in the definition of the object.

For example, the following HLOOCH$^{TM}$ file defines three subcircuits, `and`, `or` and `xor`, that implement the corresponding three boolean functions. The definition of `xor` uses the definitions of `and`, `or`.

```
<?xml version="1.0"?>
<hlooch>
<definitions>
 <circuit id="and">
  <input id="i1"/>
  <input id="i2"/>
  <output from="g2"/>
  <nand data1="i1" data2="i2" id="g1"/>
  <nand data1="g1" data2="g1" id="g2"/>
 </circuit>
 <circuit id="or">
  <input id="i1"/>
  <input id="i2"/>
  <output from="g3"/>
  <nand data1="i1" data2="i1" id="g1"/>
  <nand data1="i2" data2="i2" id="g2"/>
  <nand data1="g1" data2="g2" id="g3"/>
 </circuit>
 <circuit id="xor">
  <input id="d1"/>
  <input id="d2"/>
  <output from="and-out"/>
  <subcircuit id="or">
   <input from="d1"/>
   <input from="d2"/>
   <output id="or-out"/>
  </subcircuit>
  <nand data1="d1" data2="d1" id="nand-out"/>
  <subcircuit id="and"/>
   <input from="or-out"/>
   <input from="nand-out"/>
   <output id="and-out"/>
  </subcircuit>
 </circuit>
</definitions>
<circuit>
 <input id="i1"/>
 <input id="i2"/>
 <input id="i3"/>
 <input id="i4"/>
 <output from="xor-out-3"/>
 <subcircuit id="xor">
```

```
  <input from="i1"/>
  <input from="i2"/>
  <output id="xor-out-1"/>
 </subcircuit>
 <subcircuit id="xor">
  <input from="xor-out-1"/>
  <input from="i3"/>
  <output id="xor-out-2"/>
 </subcircuit>
 <subcircuit id="xor">
  <input from="xor-out-2"/>
  <input from="i4"/>
  <output id="xor-out-3"/>
 </subcircuit>
</circuit>
</hlooch>
```

The `and` circuit is composed from two NAND gates. Notice that if the two inputs of a NAND gate are the same, then the gate acts as an inverter. Therefore gate `g2` inverts the output of the NAND gate `g1`, thus the circuit computes the function 'and'. The `or` circuit works similarly: the two inputs are inverted with gates `g1` and `g2`, then the NAND gate `g3` computes the result. By De Morgan's identity the NAND of the inverted inputs will be the 'or' of the inputs: $\overline{\overline{a} \wedge \overline{b}} = a \vee b$.

The `xor` circuit is built from a nand gate and two subcircuits. Here we use the fact that 'xor' can be expressed as $a \oplus b = (a \vee b) \wedge \overline{(a \wedge b)}$. That is, $a$ xor $b$ is true if and only if at least one of them is true $(a \vee b)$ but not both of them are true $(\overline{a \wedge b})$.

Finally, the main circuit computes the 'xor' of the four inputs with the help of three `xor` circuits: $i1 \oplus i2 \oplus i3 \oplus i4 = ((i1 \oplus i2) \oplus i3) \oplus i4$.

## 10.2 Building the circuits

Your program has to turn a HLOOCH$^{\text{TM}}$ file into a circuit description file (see Chapter 4 for the format of circuit description files). You have to follow the following rules:

- The power pin should be at the upper left corner of the circuit (i.e., it is in row 1 and column 1).
- The input pins should be named `I1`, `I2`, ..., in the order in which they are defined in the HLOOCH$^{\text{TM}}$ file.
- All the input pins should be in the first column.
- The output pins should be named `O1`, `O2`, ..., in the order in which they are defined in the HLOOCH$^{\text{TM}}$ file.
- All the output pins should be in the rightmost column of the circuit.

Other than these rules, the circuit can be arbitrary, as long as it satisfies the definitions in the HLOOCH$^{\text{TM}}$ file, and it is a valid circuit (see Chapter 5). Figures 10.1 and 10.2 show possible solutions for the first two examples. Of course, there are many other valid solutions, but they have to work the same as these circuits.

| Total score of the test cases: | **125 points** |
|---|---|

Chapter 11

# Generating circuits

Designing large circuits by hand is very difficult, even with the high-level language defined in Chapter 10. Therefore it is very important to automate the circuit design process as much as possible. In this chapter you have to write a program that can automatically generate certain types of circuits. For example, the user enters a Boolean formula (such as `(NOT X1 OR (X2 AND X3)) AND X2`), and the program generates a circuit that evaluates this formula. Below we describe the different types of circuits that your program should be able to generate.

## Evaluating Boolean formulas

A Boolean formula can contain the following elements:

- variables `X1`, `X2`, . . . ,
- the binary Boolean operators `AND`, `OR` (with the usual meaning),
- the unary negation operator `NOT`,
- paranthesis symbols '`(`' and '`)`'.

There can be any number of spaces between the elements. The `NOT` operator has the highest precedence, followed by `AND` and finally by `OR`. Therefore the formula `X1 OR NOT X2 AND X3` is equivalent to `X1 OR ((NOT X2) AND X3)`.

Your program has to convert a Boolean formula to a circuit description. The circuit should have as many input pins as the number of distinct variables appearing in the formula, and they should be named X1, X2, etc. The circuit has one output pin, the value appearing on this pin is determined by the Boolean formula: it should be the value obtained by replacing the variables in the formula by the values of the corresponding input pins. The output pin should reach its final value in at most 2000 steps. For example if, the formula is `(NOT X1 OR (X2 AND X3)) AND X2`, then the following table shows the final value of the output pin for every combination of the inputs:

| X1 | X2 | X3 | output |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

It should be possible for the user to enter the formula on the keyboard, or load it from a text file. The formula can contain up to 32 distinct variables. If your program can handle only formulas with few (up to 7) variables, then you will get a reduced score.

## Periodic impuluse generator

In many situations, it is useful to have a periodic signal of a given frequency. Your program has to be able to generate a circuit that outputs multiple periodic signals, each with a given frequency.

The user gives $n$, the number of outputs ($1 \leq n \leq 10$), and $n$ integers $p_1, \ldots, p_n$. Each $p_i$ is greater than 1 and less than 40. The generated circuit has no inputs and has $n$ outputs. The $i$th output is periodic with period length $p_i$: it outputs a value 1 for one time step, and the value 0 for $p_i - 1$ time steps, and this is repeated forever. After the first step of the simulation all the outputs has to be 1.

For example, if the user gives the four numbes 6, 2, 10, 3, then the following table shows how the values of the output pins change after each step of the simulation:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Output 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| Output 2 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| Output 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Output 4 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |

## Arithmetic expressions

The user gives (using the keyboard or a text file) a series of arithmetic assignments, for example

```
D1=X1*(X2+X3+3)
D2=(X3-D1)*(X2+D1+1)-8
D3=D1*D2+X2*X1+17
```

Your program has to generate a circuit that outputs the value of the last expression, if the values of the variables `X1`, `X2`, ... are given as inputs.

Each assignment is on a separate line, and begins with a variable name D$i$ and the assignment operator `=`. The right hand side of the assignment is arithmetic expression that can contain the following:

- input variables `X1`, `X2`, ...,
- newly defined variables `D1`, `D2`, ... (has to be defined earlier),
- decimal integers (e.g. `28`, `0`, `-42`),
- binary operators '`+`','`-`' and '`*`',
- parentheses symbols '`(`' and '`)`',
- space characters.

If an expression contains a newly defined variable, then it has to be a variable defined earlier.

If the assignments contain $n$ input variables `X1`, `X2`, ..., X$n$, then your program has to generate a circuit with $8n$ input pins and 8 output pins. The $8n$ inputs are interpreted as $n$ 8-bit numbers, corresponding to the $n$ input variables. Input pin 1 is the highest bit (MSB) of input variable `X1`, input pin 8 is the lowest bit (LSB) of `X1`, input pin 9 is the highest bit (MSB) of input variable `X2`, and so on. The value of the last expression (last line) should appear on the output pins, output pin 1 is the highest bit (MSB) of the expression.

All calculations should be done using 8-bit arithmetic, in case of an overflow retain only the the lowest 8 bits. Negative numbers are represented using standard two complement notation, i.e., -1 is represented as 11111111, and -2 is represented as 11111110, etc.

For example, if the user gives the following assignments

```
D1=X1+X2
D2=X2+D1
D3=D1+D2
D4=D2+D3
```

then your program has to generate a circuit with 16 inputs and 8 outputs. If the two numbers appearing on the input are both 1, then the output should be 8 (binary 00000100). If both of them are 0, then the output should be 0. If `X1` = 7 and `X2` = 1, then the output is 26 (decimal).

| | |
|---|---|
| Evaluating Boolean formulas | **45 points** |
| Impulse generator | **40 points** |
| Arithmetic expression | **65 points** |

# Chapter 12

# Circuit War

## 12.1 Introduction

In the not so far distant future, the base material for high performance integrated circuits will be a new element, Selidium, discovered and available only on the 5th moon of planet Jupiter. As the Selidium stock price rockets sky-high, and each square nanometer of Selidium worth thousands of Nollars (new dollars), no wonder that a new race of computer viruses is born: the circuit virus. A circuit virus will modify the existing circuit on any IC board to gain control of it, re-burning gates and connections and re-programming circuit behavior. A truly effective circuit virus will not only crawl into computer motherboards, but into other electronic devices like microwave ovens and electric razors, turning them into TV broadcast jammers and mobile phones with highly annoying ringtones, respectively.

## 12.2 Objective

Your task is to write a counter circuit virus, which will stop the spreading of a circuit virus on any IC board. You can stop the spreading of a circuit virus by applying the same (or more) aggressive behavior as the virus displays: you have to occupy the largest possible territory on a given circuit board. The counter viruses, written by all participating teams, will be tested for their efficiency in a championship, where your team's program has to beat the others' in a multiple set of games.

## 12.3 Definitions

A circuit `board` is represented by a square grid, its size ranging from 3 to 21 units. The championship will be held on a fixed size board of 11x11, however for testing purposes your program has to be able to "adapt" to different board sizes.

The key terms for the competing programs are the `squares` and the `segments` on the board. A `square` is the smallest area on the board enclosed by gridlines. The upper left `square` of the `board` is represented by the (0, 0) coordinates, the lower right square is represented by the (n-1, n-1) coordinates, where n is the size of the `board`. In the (x,y) coordinate pair, x advances from left to right, while y advances from top to bottom. (see Figure 12.1)

A `segment` is an elementary part of the grid connecting two neighboring grid points together, creating a line segment on any `border` of any `square`. Hence, any segment on the board can be represented as a coordinate triple (x, y, border), where x and y are the coordinates of the given `square`, and `border` is one of the following elements: top, right, bottom, left.

Note: this logical representation is redundant. Assuming that $x$, $y$, $x+1$, $y+1$, $x-1$, $y-1$ are all within the range of 0 and $n-1$ ($n$ meaning the size of the `board`), the following

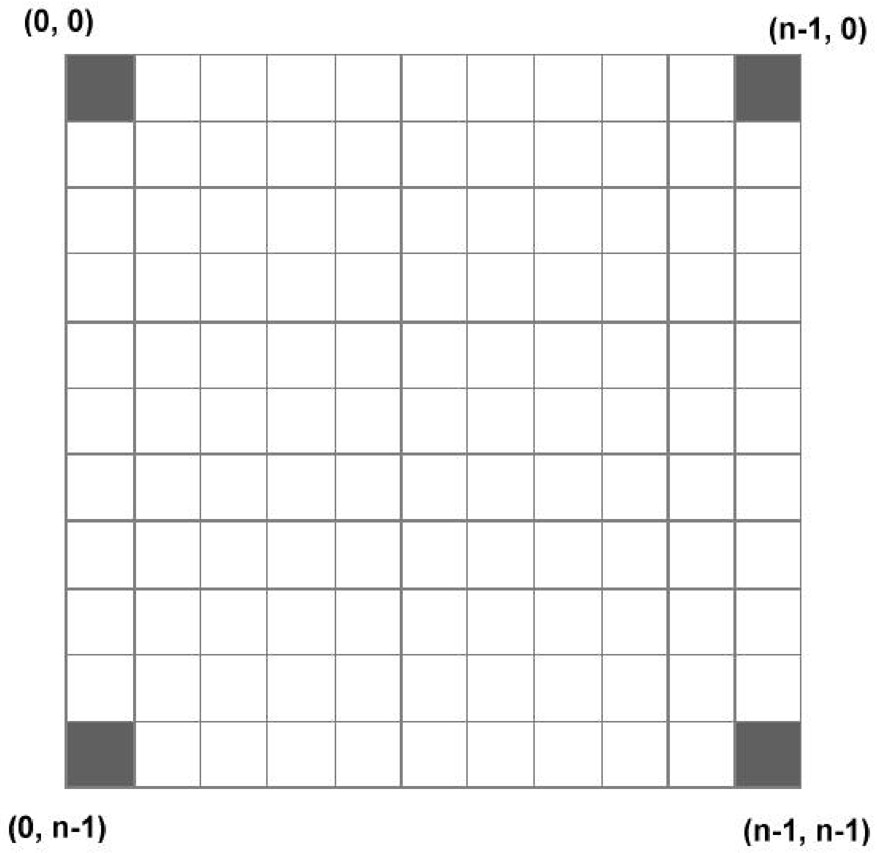


Figure 12.1: Squares on the board

borders are identical:
(x, y, top) and (x, y-1, bottom),
(x, y, right) and (x+1, y, left),
(x, y, bottom) and (x, y+1, top) and
(x, y, left) and (x-1, y, right).

During the flow of the game, for every turn the two competing programs will select a `free segment` on the board. Once a given side of a given `square` is `occupied` (selected) by any of the competing programs, it will remain `occupied` until the end of the game, hence it will not count as a `free` side of that `square`. At the start of the game every `segment` is `free`, including those on the `board`'s boundaries.

Two `squares` on the board are considered `neighboring squares`, if they have a common side. Therefore, each `square` on the board has four `neighboring squares` (top neighbor, right neighbor, bottom neighbor, left neighbor), except for the `squares` on the board's boundaries (the top left `square` on the `board`, having (0, 0) coordinates, has no top neighbor and no left neighbor, for example).

An `area` on the board is defined as a selected set of squares, where any given square from the set has at least one neighboring square also present within the set. In other words, an area is an interconnected set of neighboring squares on the board.

A `closed area` is an area that has a continuous chain of occupied segments around its borders. In other words, in a closed area set for all the borders of every square (top, right, bottom and left)

- either there is a neighboring square also present in the set
- or the respective segment is occupied by any of the players.

See Figure 12.2 for several examples of occupied segments and closed areas.

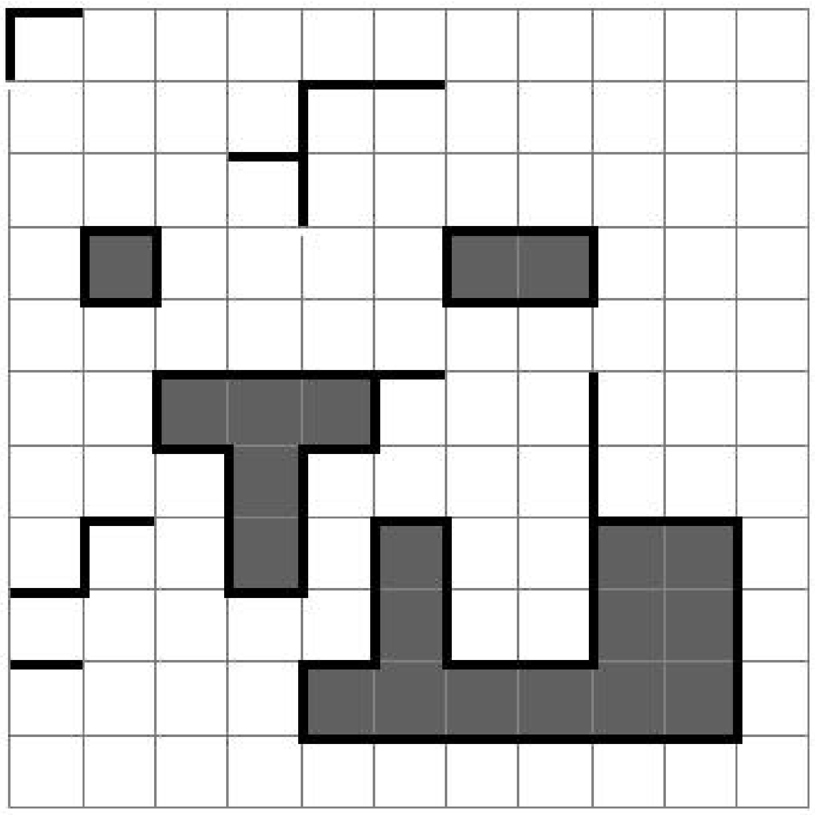


Figure 12.2: Occupied segments and closed areas

## 12.4 Rules of the game

A game session starts when two programs successfully log in to the game server. A game session consists of one or more games with the same two client programs, each game starting with an alternate client. The first game always starts with the client that logged in first to the server, the second game will start with the opponent's move, and so on. During the championship, game sessions will always contain two games.

A game is composed of alternate turns of the clients, until one of the "end of game conditions" is met. At the end of game, either a new game will start with the same client programs, or the game session will end, depending on the total number of games in the game session and the number of games already played.

## 12.5 Turns

A game on the board will break into turns: one program conducts one step at a time. A step is interpreted as occupying one of the free segments on the board. If the given segment creates a new closed area on the board, the squares in that closed area will increase the program's score, each square counting 1 point. If a new closed area is created during a turn, all borders of all squares within the closed area automatically become occupied. Therefore, even if no player occupied them previously, segments within a closed area will not count as valid steps for the rest of the game.

Any program, while its turn lasts, can present more than one possible step to the server, however, only the last step will be taken into account.

A program's turn ends (i.e. the other program will continue with its turn), if any of the following conditions are met:

- the respective program sends the "done" message to the server, meaning that it has no intention to present additional alternative steps.
- the time limit is reached for the given turn.
- the program presented an invalid segment (occupied or out of the table boundaries), or violated the communication protocol in any way.

`Warning`: The last from the above listed conditions (invalid step or communication protocol violation) automatically implicates the end of game as well.

## 12.6 Time limits

For each turn, there is a time limit, within which the oncoming program should present a valid step, otherwise a `timeout event` will occur. If timeout occurs, but the program already presented at least one valid step towards the server, the timeout has no effect, the server will simply accept the last step presented by the program and end the current turn.

However, if before the timeout event there was no step sent to the server, the following events will take place:

- If this is the n. occasion for the respective program that a timeout has occurred during the current game, where n equals the `maximum number of timeouts`, the game will end immediately.
- If there were less timeout occasions for the respective program during the current game so far, than the `maximum number of timeouts`, the server will generate a random step for that program (chosen from any free segment on the board)

Note: As there is a possibility to present several steps during one turn, it is highly recommended to present a first step as soon as possible, and only after that should the program try to find the best possible solution(s). In that way it is easy to avoid the very frustrating event of a timeout, not to mention the event of too many timeouts - and therefore possibly loosing the game (see end of game rules).

## 12.7 End of game

The game will continue turn by turn, until one of the following conditions is met:

- there are no more available free segments on the board. In that case the winner of the game is the program that has the higher score. Note: as the size of the board is odd, there is no possibility of achieving a draw.
- one of the programs had a timeout event, and it was the n. occasion for the respective program, where n equals the `maximum number of timeouts`. In that case, that program has "used up" all its timeout possibilities, and the other program will gain all the remaining free squares on the board (its score will be increased by the number of free squares). For example, on a 5x5 board, `player1` having a score of 4, and `player2` having a score of 9, if `player2` uses all its timeout possibilities, the ending score will be 16:9 for `player1`. Obviously, a program using up all its timeout possibilities and therefore ending the game prematurely, can still win, if it had a large advantage over the other program or if there are only a few free squares left.
- one of the programs sends an invalid step to the server or violates the communication protocol. The end of game scoring and the decision of the winner occurs exactly the same way, as it was described in the previous point.

## 12.8 Communication protocol

The communication between clients (i.e. the programs written by the participating teams) and the server utilizes TCP/IP sockets. As the championship will take place on more than one server in parallel, and more server instances may run on the same physical hardware, the clients should be configurable in terms of server IP address and communication port number. The clients and the server will use the defined socket for both outgoing and incoming communication. The communication consists of exchanged text messages, each message terminated with the end of line sequence. Integer values should be sent and received as the String representation of the given integers. To test the communication protocol, you can use the telnet application on the test server, provided for each team.

All communication messages are case sensitive, therefore "login" does not equal "Login". In the following examples, the messages sent by the client will be shown in *italic characters*, while the messages sent by the server will be shown in `normal monospaced characters`.

### 12.8.1 Logging in

After opening the TCP socket at the provided server IP address and communication port number, the program has to log in. For logging in to a game, the client should send the following message to the server:

*login <team ID> <team name>*

where <team ID> is the identification code provided for each team by the organisers, and <team name> is the name of the competing team. <team name> may contain any characters including space.

The server will reply to a valid login with the

```
<team ID> accepted
```

message. If there was an error during the login, the server will close the socket, but will still accept further logins as nothing has happened.

The login session ends when two programs managed to log in. At the end of the login dialog, the server will notify the players about different parameters related to the current game session, like the size of the board or the time limit value for a turn. The server will send the following messages to **both** players in the following order:

```
opponent <opponent team ID>
boardsize <size of the board>
timeout <time limit for a turn>
maxtimeoutnum <maximum number of possible timeouts during the game>
numofgames <number of games to be played within this game session>
```

The above messages are pretty self-explanatory, however the timeout value ( <time limit for a turn> ) needs some explanation: it is given in milliseconds. Therefore the message `timeout 1000` means that there is exactly one second in every turn to find the best possible step.

A possible login session would look like the following:

*login B-22 Bithunter warriors*

```
B-22 accepted
```

now there may be some pause until the other client logs in, assuming that it has not logged in already.

```
opponent B-105
boardsize 11
timeout 500
maxtimeoutnum 3
numofgames 2
```

### 12.8.2 Conducting a turn

The program that logged in first starts the first game. At the beginning of every turn, the program whose turn is on, will be notified as follows:

```
turn <team ID>
```

Please note that this message will only be received by the program whose turn is just staring, the opponent will not be notified about this event.

Now the program, whose turn is on, has the possibility to send one or more possible steps to the server in the following format:

```
<x> <y> <border>
```

Where the value of x and y should be positive integer numbers in the range of 0..n-1 (n being the size of the board), representing the coordinate of a square on the board, and the value of border should be an integer number in the range of 0..3, representing the four borders as follows:
0 means top,
1 means right,
2 means bottom,
3 means left.
Remember, values outside the above given ranges, or (x,y,border) coordinate triples pointing to an occupied segment will be considered invalid, and will terminate the current game.
As mentioned before, within the specified time limit the client can send further steps towards the server. All of the steps but the last one will be automatically discarded by the server, the last one will be considered as the only step.

```
<x2> <y2> <border2>
<x3> <y3> <border3>
```

The turn can be ended by the client by sending the following message:

```
done
```

or by the server sending one of the following messages:

```
timeout
```

or

```
invalid
```

meaning respectively a timeout event or an invalid step. Note that theoretically it is possible to receive a timeout message, even after sending done to the server, if these two events happen at almost the same moment. If, due to an invalid step or due to the number of timeouts the game ends, the message described in the next section will be sent to the players. Otherwise, the server will broadcast (i.e. send to both clients) the step of the current player the following way:

```
<team ID> <x> <y> <border>
```

Note that this is the only message received by the program that is waiting for its turn during the opponent's turn. After that the game will advance to the next turn, and the next player will be notified as specified in the beginning of this section.

A sample segment of communication between one of the clients and the server:

```
turn B-22
2 2 0
2 2 1
3 2 2
done
B-22 3 2 2
B-105 3 2 3
turn B-22
timeout
B-22 2 2 0
```

### 12.8.3 End of game

At the end of the game the following message will be broadcasted:

```
gameover <winner team ID> <winner score> <looser team ID> <looser
score>
```

Following the previous example, and end of game communication could look like:

```
B-105 4 1 1
turn B-22
timeout
gameover B-105 17 B-22 8
```

After the gameover message, if there are more games to be played (according to the game session), a new game starts. The client that has not started the previous game will be appointed to make the first move. Obviously, at the start of every game the starting score of both programs will be zero, and the board will be empty (each segment will be free on the board). Within one game session all parameters (like the board size, or the time limit value) remain the same.

### 12.8.4 End of game session

After finishing the n. game, where n is the number of games in the current game session, the server will say goodbye to the clients and close the socket. The following message will be broadcasted at the end of game session:

```
bye
```

No answer is required from the clients for this message. The server will start a new game session and will accept new clients for logging in.

## 12.9 Operating the reference server

### 12.9.1 Environment

For development and test purposes, each team will receive a reference server on a CD. The server is written in Java, developed under the 1.4.1_02 environment. Although there may be other Java versions, under which the server operates without flaws, we strongly suggest using the above mentioned environment. The server was tested only under Windows98 and Linux operating systems—we do trust Java's portability. However, if you are using the 1.4.1_02 Java Runtime Environment under a different operating system, and you happen to encounter some problems while running the reference server, please contact the staff immediately.

### 12.9.2 Starting the server

After setting the classpath correctly (hopefully done during the Java environment installation), the server should be started with the

```
java -jar CircuitWar.jar <optional parameters>
```

command.

### 12.9.3 Parameters

The following parameters will be recognized by the server:

-port <port number>

Defines the communication port between client and server. <port number> should be an integer value between and including 1025 and 65535. If this parameter is omitted, the default port number, 2400, will be used for communication.

-boardsize <size of the board>

Defines the size of the gameboard. <size of the board> should be an odd integer value between and including 3 and 21. Even board size number should not be passed to the server, as it permits a draw at the end of the game. If this parameter is omitted, the default board size, 11, will be used for the game.

-timeout <time limit for a turn in milliseconds>

Defines the time limit value for a turn. The <time limit for a turn in milliseconds> should be an integer between and including 100 and 60000. The default timeout value is 5000 (i.e. 5000 milliseconds, or 5 seconds).

-numoftimeouts <maximum number of possible timeouts during the game>

Defines the maximum number of possible timeout events for a player during one game. The value for this number should be an integer between and including 0 and 1000. Passing 0 for this parameter will result in a zero tolerance game: at the first timeout event the server will end the current game. The default value for this parameter is 5.

-numofgames <number of games to play in the current session>

Defines the number of games to be played in the current game session. The value for this number should be an integer between and including 1 and 100. The default value for this variable is 2.

-endofgamepause <length of forced pause at the end of each game>

Defines for how long should suspend the server the flow of game at each game's ending. This parameter is interpreted as milliseconds, allowing the players to examine the end of game situation on the board. Passing 0 (zero) as argument will require user interaction to advance to the next game: the game will pause until a mouse click in the graphical display. The default value for this parameter is 5000.

-nodisplay

Passing this parameter will result in a "blind" game: the graphical display will not be shown during the game. The default value for the graphical display is true: if you do not pass "-nodisplay" to the server, the game will be shown on the graphical display.

Possible invocations of the server could look like:

```
java -jar CircuitWar.jar -boardsize 3
```

or

```
java -jar CircuitWar.jar -port 3333 -boardsize 5 -timeout 1000 -numoftimeouts 3
-numofgames 1
```

## 12.10 Rules of the Championship

The championship will take place in the first level of the hosting building. Shortly before the start of the championship each qualifying team will be asked to place one network capable computer along the corridor, on which the team intends to run its client. The computers network settings should be set up as follows:

- IP address: 192.168.0.team_id , where team_id is the identification code of the participating team.
- Gateway: 192.168.0.254
- Netmask: 255.255.255.0

The qualifying teams will be organized into groups, the teams within the groups will compete to get in to the best group, where the finals will take place. During the championship, the server will be run with the following parameters:

- -boardsize 11
- -timeout 1000
- -numtimeouts 3
- -numofgames 2

### 12.10.1 Qualification

To enter the championship, your program should prove that it is capable of playing a full game from logging in until all segments on the board become occupied, without prematurely ending the game with communication error or too many timeout events. The opponent will be appointed by the staff. There is no requirement on winning the game to get qualified.

| | |
|---|---|
| Passing the qualification | **30 points** |
| **First phase** | |
| 1st place in the group | **20 points** |
| 2nd place in the group | **10 points** |
| 3rd place in the group | **5 points** |
| **Second phase—direct elimination** | |
| For each game won | **20 points** |
| **Third phase—best four** | |
| 1st place | **85 points** |
| 2nd place | **50 points** |
| 3rd place | **25 points** |