Iowa Logic Simulator User's Manual

Version 10

Douglas W. Jones

February 12, 2001

ACM *Computing Reviews* Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids – *hardware description languages*; D.3.2 [**Programming Languages**]: Language Classifications – *specialized application languages*

Contents

1 Introduction **1**

2 The Logic Specification Language **3**

2.1 The Circuit Name 3
2.2 The Input Connections 4
2.3 The Output Connections 4
2.4 The Parts List 5
2.5 The Wire List 6
2.6 Complete Examples 7
2.7 Advanced Features 9
2.7.1 Subcircuits 10
2.7.2 Text Inclusion from Multiple Source Files 11
2.7.3 An Example with Subcircuits 12
2.7.4 Truth Tables 13
2.7.5 Arrays and Iterators 15
2.7.6 Constant Declarations 17
2.7.7 Conditional Part and Wire lists 18
2.7.8 Parameterized Circuits 19
2.7.9 Time Delays 20
2.8 Formalities 22

3 Using the Logic Simulator **29**

3.1 The Simulation Header 30
3.2 Running the Simulation 31
3.3 Commands to the Simulator 34
3.4 Simulator Error Messages 36
3.4.1 First Pass Error Messages 36
3.4.2 Second Pass Error Messages 37

3.4.3 Rewrite Phase Messages and Warnings 40
3.4.4 Simulation Error Messages 40

4 74LS TTL Components **41**
4.1 Primitive Gates 45
4.2 Multiplexers 48
4.3 Decoders 49
4.4 Arithmetic and Comparison 51
4.5 Three-state Bus Drivers 54
4.6 Flipflops 55
4.7 Counters 58
4.8 Shift Registers 59

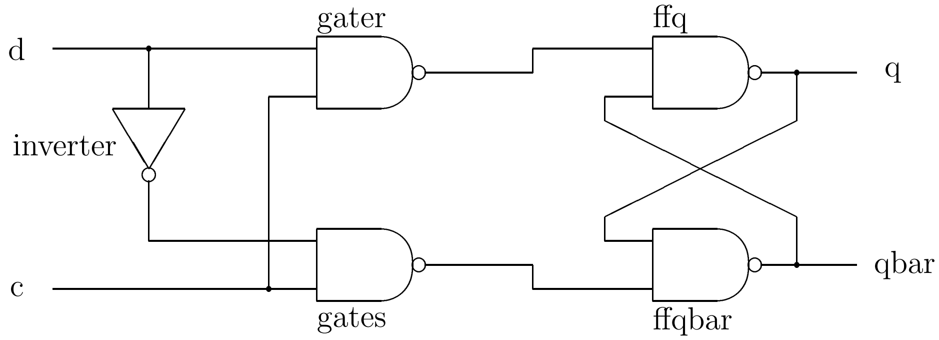
Chapter 1

Introduction

The Iowa Logic Simulator is an interactive program which allows digital circuit designs to be tested without the need to actually build them. The simulator reads circuit descriptions written in the Iowa Logic Specification Language (defined in Chapter 2); this is a textual language, so any available text editor can be used to create circuit descriptions. The syntax of this language borrows many elements from familiar programming languages, but it is not a procedural language; rather, it describes the parts of a digital system and the pattern of their interconnection.

Given an error-free circuit description, the Iowa Logic Simulator will repeatedly request simulated inputs to the circuit, and it will display the resulting outputs as waveforms on the terminal screen. As such, it emulates a logic analyzer as it would be used in debugging a real digital system. The simulator can reproduce the detailed timing of digital systems because it uses a discrete-event simulation model. These facilities are explained in Chapter 3.

The following circuit will be used as an example to illustrate the use of the logic simulator throughout this manual:



The Iowa Logic Simulator was originally written in 1983 to support the laboratory component of the course 22C:31, "Digital Systems and Computers," at the University of Iowa. The work of the original authors, Wun Chin Kao and Douglas Jones, has been improved by many others over the years. In 1985, George S. Singer added subcircuits for his MS Thesis project; later that year, Daniella Rus made major performance improvements with financial support from the University of Iowa Weeg Computing Center, and in 1987, Jing Jan added arrays and abstraction facilities as part of his MS Thesis project.

It should be noted that, as of Version 10, the Iowa Logic Simulator is not yet the tool we desire, but it is very powerful. The largest circuit it has been used to simulate (to date) is a central processor with 16 registers (6 bits each), an arithmetic logic unit, and an accumulator, with a microcoded control unit containing a 16 instruction writable microstore; this was a class assignment in 1987. Software we would like includes a preprocessor to allow truth table and equational notation in the input, a graphics editor for logic circuits, and a graphic interface to the simulation routines.

Chapter 2

The Logic Specification Language

A circuit description in the Iowa Logic Specification Language consists of 5 sections. These sections describe the circuit name, the circuit inputs, the circuit outputs, the parts used to make the circuit, and how the inputs, outputs, and parts are to be wired together. These sections must be given in the order listed.

Circuit descriptions are free format, so the indenting rules used in the following material are actually optional. The punctuation marks *comma* and *semicolon* are also optional, but their use is encouraged to improve the readability of circuit descriptions.

Comments may be included in circuit descriptions, and their use is encouraged. Blank lines may be used to delimit or separate blocks within a description. All of the comment notations found in Pascal, C and Ada are accepted, so `--` may be used to mark a marginal comment, where everything from the paired dashes to the end of line is ignored, and comments may be enclosed in braces as `{ }`, `(* *)`, or `/* */`. The closing comment brace must match the opening brace, and users should avoid the form `{= =}`, which is reserved for communication between preprocessors and the logic simulator.

Within a circuit description, a number of identifiers are used to name such things as gates and input or output connections. Identifiers may consist of any number of characters as long as the indentifier fits on one line, and as long as it consists of a letter followed by a sequence of letters and digits. All characters of an identifier are significant, and reserved words may not be redefined.

2.1 The Circuit Name

Every logic circuit description must begin with a declaration of the circuit name. Name declarations begin with the keyword `circuit`, followed by an identifier used as the circuit name. For the example circuit, which turns out to be a *type D* latch, the following declaration might be used:

```
circuit dlatch;   -- designed by D. W. Jones
```

By convention, the circuit designers name should be put in a comment in the circuit header, as shown above.

2.2 The Input Connections

Most logic circuits will have one or more input connections; if a circuit has none, this entire section may be omitted. If present, inputs must be declared immediately following the circuit name using the keyword `inputs`, followed by a list of input names. For the example circuit, the inputs may be declared as follows:

```
inputs  d, c;
```

It would be better to put a comment on each input explaining what it is for, as in the following:

```
inputs
        d {data},
        c {clock};
```

Spelling out the appropriate mnemonic names would have eliminated the need to include these comments, but long names for the inputs or outputs can lead to hard to read simulation results (see Chapter 3), and they do not improve the readability of small circuits such as the example.

2.3 The Output Connections

Most logic circuits have output connections, but there are rare occasions when it is useful to be able to declare circuits with no explicit outputs. If present, outputs are declared immediately after the inputs, using the keyword `outputs` and a list of output names. For the example circuit, the following declaration would be appropriate:

```
outputs
        q {the stored data value},
        qbar,
        dout {echo input d},
        cout {echo input c};
```

Note that `qbar` needs no comment, since (by convention), the suffix `bar` is used to indicate that this is the inverse of `q`.

2.4 The Parts List

After the inputs and outputs are declared, any parts used in the circuit must be declared, although it is sometimes useful to create circuits with no parts where the inputs are wired directly to the outputs. If parts are needed, they are declared after the keyword `parts`. Each part declaration consists of a list of new part names, a colon, and the part type to be used for those names. There are a number of predefined part types, and new part types may be defined as needed using the subcircuit mechanism. The predefined part types are listed below:

simple Boolean logic gates:

- `not` – inverter
- `and(n)` – n input and gate
- `or(n)` – n input or gate
- `nand(n)` – n input nand gate
- `nor(n)` – n input nor gate
- `xor` – exclusive or gate
- `equ` – equivalence gate

special gates:

- `tsgate` – non-inverting three-state driver
- `ntsgate` – inverting three-state driver
- `bus` – tristate bus
- `latch` – type D latch (1 bit memory)

Note that with many of the Boolean logic gates, the number of inputs is variable and must be specified as a parameter to the part type. It should also be noted that, for the purposes of logic simulation, a tristate bus is a device, even though in a real circuit, it is merely a piece of wire. For the example circuit, there are 4 two-input nand gates, and 1 not gate, so the following parts declaration would do:

```
parts   gater, gates, ffq, ffqbar: nand(2);
        inverter: not;
```

It would be better to put comments on the individual definitions as follows:

```
parts   { basic rs flipflop gates }
        ffq, ffqbar: nand(2);

        { gates used to convert d and c to r and s }
        gater, gates: nand(2);
        inverter: not;
```

Commenting could be carried further, but would probably not make the resulting circuit definition any easier to read.

2.5 The Wire List

After declaring the parts, all that is needed to finish the circuit is a description of how the parts are connected to the inputs and outputs. There are rare cases where it is useful to create a circuit containing no wires. The wire list of a circuit starts with the keyword `wires` and runs up to the keyword `end` that marks the end of the entire circuit. Each entry in the wire list consists of the name of a signal source, the keyword `to`, and a list of names of signal destinations.

Any input to the circuit is a valid signal source, as is the output connection of any previously declared part. In addition, any valid signal source declared globally to a subcircuit may be used within that subcircuit. The special signal sources `high` and `low` are declared globally for use when constant inputs are required.

The valid signal destinations include the outputs of the circuit and any unused inputs of a previously declared part. In addition, any valid signal destination declared globally to a subcircuit may be used within that subcircuit.

The following table lists the forms of the input and output connections of each of the types of gates supported by the logic simulator:

part name	type of part	inputs	outputs
a	not and(1) or(1) nand(1) nor(1)	a.in	a.out
b	and(n) or(n) nand(n) nor(n)	b.in(1) to b.in(n)	b.out
c	xor equ	c.in(1) c.in(2)	c.out
d	tsgate ntsgate latch	d.control d.data	d.out
e	bus	e.in	e.out

Thus, if `b` is the name of a 3 input nor gate, it will have inputs `b.in(1)`, `b.in(2)`, and `b.in(3)`, and it will have an output named `b.out`. The subscript on the `in` connection to a gate is only permitted if the gate has a fan-in of at least two.

The example circuit given here performs exactly the same function as the built-in `latch` part type, but with inputs named `c` and `d` instead of `control` and `data`, and with the output named `q` instead of `out`. The wire list for this example circuit could be given in the following form, but appropriate comments and blank lines (as shown in the next section) would make it easier to read:

```
wires
        d to gater.in(1), dout, inverter.in;
        inverter.out to gates.in(1);
        c to gater.in(2), gates.in(2), cout;
        gater.out to ffq.in(1);
        gates.out to ffqbar.in(1);
        ffq.out to q, ffqbar.in(2);
        ffqbar.out to qbar, ffq.in(2);
end.
```

Some care should be taken in wiring circuits that involve tristate busses. In a physical circuit, if one bus is connected to another, it is equivalent a single bus, and any changes made to one bus will be seen on all connected busses. The logic simulator does not work this way! In general, logic circuits should not be constructed with the output of one bus driving another, although the logic simulator will attempt to rewire such circuits so that they operate correctly. The logic simulator will issue warnings or error messages when it encounters circuits where it can not guarantee a successful rewrite.

One particularly useful rewrite rule within the logic simulator guarantees that if the output of a tristate bus driver is connected to both a tristate bus and to inputs of some additional gates, the result will operate exactly as if the bus driver was connected only to the bus, and the additional gates took their inputs from the bus.

2.6 Complete Examples

Combining the heading, input/output lists, parts list and wire list from the previous sections gives the following description for the example circuit:

```
circuit dlatch;   -- designed by D. W. Jones
inputs
        d {data},
        c {clock};
outputs
        q {the stored data value},
        qbar,
        dout {echo input d},
```

```
        cout {echo input c};
parts
        { basic rs flipflop gates }
        ffq, ffqbar: nand(2);

        { gates used to convert d and c to r and s }
        gater, gates: nand(2);
        inverter: not;
wires
        { distribution of inputs to rs conversion gates }
        d to gater.in(1), dout, inverter.in;
        inverter.out to gates.in(1);
        c to gater.in(2), gates.in(2), cout;

        { transfer of r and s values to basic rs flipflop }
        gater.out to ffq.in(1);
        gates.out to ffqbar.in(1);

        { basic rs flipflop }
        ffq.out to q, ffqbar.in(2);
        ffqbar.out to qbar, ffq.in(2);
end.
```

In the following sections, it will be assumed that this circuit description is stored in file `dlatch` (the convention of using the circuit name as the file name is strongly encouraged).

Unlike most computer programming languages, the order of the elements in the wire list has no effect on the meaning of the circuit description; nonetheless, careful ordering and grouping of wires is important to human readers. In this example, the wire list has been broken up into functional sections, each headed with a comment.

If the TTL library is properly installed, and is stored in the directory named `ttl`, we can rewrite this above example in terms of TTL chips:

```
circuit dlatch;   -- designed by D. W. Jones

    -- what TTL chip definitions are needed
    use ttl/LS00;
    use ttl/LS04;

inputs
        d {data},
        c {clock};
outputs
        q {the stored data value},
```

```
        qbar,
        dout {echo input d},
        cout {echo input c};
parts
        -- the chips we need
        chip1: LS00;
        chip2: LS04;
wires
        { distribution of inputs to rs conversion gates }
        d to chip1.P1, chip2.P1, dout;
        chip2.P2 to chip1.P4;
        c to chip1.P2, chip1.P5, cout;

        { transfer of r and s values to basic rs flipflop }
        chip1.P3 to chip1.P13;
        chip1.P6 to chip1.P9;

        { basic rs flipflop }
        chip1.P11 to chip1.P10, q;
        chip1.P8 to chip1.P12, qbar;

-- power to chip1.P14, chip2.P14
-- ground to chip1.P7, chip2.P7
end.
```

As is common in chip-level designs, the names given for the chips are assigned sequentially, but it would be equally reasonable to give approximate chip coordinates on a circuit board, perhaps in terms of row and column. The names P1 through Pn are reserved for the I/O pins of an n pin chip, and as with the real hardware, nothing external to the chip distinguishes input and output pins. The logic simulator does, however, detect misuse of inputs for output or visa versa.

Note that, by adding the comments giving power and ground connections, the wire list of the Iowa Logic specification language becomes, literally, the wire list needed to actually build the circuit. The logic simulator does nothing to check the correctness of these comments, but if you build it and get the power and ground connections wrong, you can destroy chips (the chips described in the TTL support library cost between 1 and 10 dollars each, depending on complexity and market conditions).

2.7 Advanced Features

The Iowa Logic Specification Language has a number of features which allow it to be used to specify large and complex systems of logic gates. Primary among these are a subcircuit mechanism and an array mechanism. The former allows a circuit to be defined once and used many times, while the latter allows input or output connections from a circuit to consist not of individual wires but of named arrays of wires.

In some applications, it is necessary to simulate not only the logical behavior of a circuit, but also the time that the circuit takes to act on changes in its inputs. The Iowa Logic Specification Language provides complete support for specification of such time delays, and the associated simulator accurately simulates them. If time delays are not specified, default delays are assumed that are typical of breadboard circuit construction with TTL gates.

2.7.1 Subcircuits

Large digital systems frequently consist of many logical subsystems, and the Iowa Logic Specification Language reflects this by allowing the declaration and use of subcircuits. Any circuit described in the Iowa Logic Specification Language may be used as a subcircuit of some larger circuit. Subcircuits of a circuit may be declared immediately after the circuit name, before the declarations of inputs and outputs. A subcircuit declaration consists of exactly the same parts as any other circuit declaration; it starts with a name, followed by an input list, an output list, a parts list, and a wire list.

When a circuit is used as a subcircuit, its name becomes a new gate type, and any number of parts of this type may be declared. For example, if the circuit defined in the previous sections is used as a subcircuit, any number of parts of type `dlatch` may be declared, as in the following example:

```
parts
        bit1, bit2, bit3, bit4: dlatch;
```

This declares the parts `bit1` through `bit4` to be instances of the circuit `dlatch`. These instances are completely independent of each other; all that they have in common is that they are described by the same circuit description.

Connections between an instance of a subcircuit and its user are made through the inputs and outputs of the subcircuit. For example, given the above definitions, it would be legal to connect `bit1` through `bit4` into a shift register as follows:

```
wires
        bit1.q to bit2.d;
        bit2.q to bit3.d;
        bit3.q to bit4.d;
```

This relies on the fact that each instance of `dlatch` has an output named `q` and an input named `d`. All of the inputs of a subcircuit must be connected for that subcircuit to function correctly; for example, `bit1.c` and `bit1.d` must be connected to make the subcircuit instance `bit1` work. Some subcircuits may have inputs which are not needed in a particular application; these should be connected to the global inputs `high` or `low`, as appropriate. The outputs of `bit1` are `bit1.q`, `bit1.qbar`, `bit1.cout`, and `bit1.dout`. It is not necessary to use all of the outputs of a subcircuit.

In many ways, subcircuits are similar to procedures in a conventional programming language. Thus, it is correct to talk about the main circuit and the subcircuits it uses. The inputs and outputs of a subcircuit are similar to procedure parameters, since they allow communication between the user of a subcircuit and the subcircuit. The Iowa Logic Specification Language has scope rules similar to those of Pascal.[1] If subcircuit `X` is declared within subcircuit `Y`, then `X` may only be used in `Y` and in other subcircuits declared in `Y`. That is, `X` is considered to be a local subcircuit of `Y`.

There are two recommended approaches to interfacing subcircuits to tristate busses. In all cases, the bus itself should be declared globally, not within a subcircuit. The simplest is to use one input pin and one output pin of each subcircuit to interface to each bus wire. The bus driver in the subcircuit should directly drive the output pin, and all components in the subcircuit that take inputs from the bus should do so through the input pin.

The second recommended approach takes advantage of the tristate rewrite rules implemented by the simulator. In this case, a subcircuit that must both read and write the bus can be connected to the bus with just an output pin. The circuit should be wired with the bus driver driving this pin and all of the bus receivers, and the logic simulator rewrite rules will move the inputs to the bus receivers from the bus driver to the bus itself.

2.7.2 Text Inclusion from Multiple Source Files

The Iowa Logic Specification Language provides a text inclusion construct as an alternative to physically including the text of a subcircuit in the definition of a circuit which uses it. The text inclusion construct consists of the keyword `use` followed by the name (or pathname) of the file where the desired subcircuit definition is to be found. Use constructs are only allowed in places where a subcircuit definition would be legal. For example, to use the circuit `dlatch` as a subcircuit, assuming it is stored in a file with the same name, all that would be needed is the following construct:

```
use dlatch
```

The use construct textually inserts the contents of the named file into the original source file as the logic simulator reads it. The file referenced by a use construct may contain a sequence of subcircuit or constant definitions, and each subcircuit in the referenced file may contain local subcircuits and other definitions.

Files referenced by use constructs may contain other use constructs, but such nesting should be limited to 3 levels beyond the original source file (the logic simulator can have no more than 4 open input files at any time). Circuits with deeply nested use constructs can

[1] Strictly speaking, the language has dynamic scope rules, while Pascal has static scope rules. This is an error in the current implementation of the language, but most users will never notice the difference.

frequently be fixed by putting all of the use constructs in the main circuit, in effect, making all of the used subcircuits global. Note that use constructs *do not* introduce any scope rules!

Try to avoid multiple use constructs referencing the same file, since this will result in a waste of memory and time when the logic simulator processes a circuit. If two subcircuits both require definitions from the same file of definitions, put a use construct for that file global to both subcircuits instead of two use constructs, one local to each subcircuit.

2.7.3 An Example with Subcircuits

The examples given above suggest the construction of a shift register made of *type D* latches. The complete specification of this circuit would have the following structure:

```
circuit dregister;   -- designed by D. W. Jones

    circuit dlatch;
    inputs
            d {data},
            c {clock};
    outputs
            q {the stored data value},
            qbar,
            ... rest of definition from Section 2.6 ...
    end;    -- dlatch

-- start of dregister
inputs
        i { the data to shift into the register },
        c { the clock signal to shift the register };
outputs
        o1, o2, o3, o4 { output the 4 bits of the register };
parts
        invert: not { used to invert the clock };
        bit1, bit2, bit3, bit4: dlatch { storage for bits };
wires
        { wires for the data flowing through the register }
        i to bit1.d;
        bit1.q to o1, bit2.d;
        bit2.q to o2, bit3.d;
        bit3.q to o3, bit4.d;
        bit4.q to o4;

        { wires for the clock signals of the register }
```

```
          c to bit1.c, bit3.c, invert.in;
          invert.out to bit2.c, bit4.c;
end.      -- dregister
```

In this example, note that there is no conflict between the input `c` of the circuit `dregister` and the input `c` of the subcircuit `dlatch`. Within the subcircuit, the name `c` is the input to that subcircuit; outside the subcircuit, `c` is the input to the entire circuit, and `bit1.c` through `bit4.c` are the inputs to the instances of the subcircuit. Within the subcircuit, `c` is a source of data; outside the subcircuit, `bit1.c` through `bit4.c` are destinations.

As with the previous example, commenting and indenting have been used here to help the reader find logically related wires and to set off the subcircuit from the main circuit. Note the comment at the end of each circuit and subcircuit which gives the circuit or subcircuit name. Also note that a comment is included after the end of the subcircuit definition to help the reader find the start of the main circuit.

This example of the use of subcircuits could be written in more compact form with the aid of the `use` construct. Assuming that the circuit `dlatch` is stored in a file with that name, the file defining the shift register would have the following form:

```
circuit dregister;   -- designed by D. W. Jones

     use dlatch;

-- start of dregister
inputs
        ... rest of definition as given above ...

end.    -- dregister
```

2.7.4 Truth Tables

The Iowa Logic Specification Language allows tabular specification of the function of a circuit or subcircuit. The table specification language allows fairly conventional looking truth table notation. Because of the graphical nature of such tables, truth tables must be entered in a fairly strictly indented format.

Tables begin with the keyword `table` and end with the keyword `end`. The `table` keyword defines the margin that must be used in indenting the table body up to and including the `end` keyword. Comments and white space are the only material allowed to the left of this margin. Note that the alignment constraints are interpreted under the assumption that tab stops are set every 8 columns.

Tables consist of two halves separated by a vertical line composed of a series of aligned `|` (vertical bar) characters. Blank lines and comments may interrupt this vertical line, but the vertical bar characters must be aligned exactly.

Inputs to the truth table are listed on the left side of the vertical line, while outputs are listed on the right. The first line of the table lists the names of each input to the left of the vertical bar, and the name of each output to the right. Each successive line, excepting comments, specifies a combination of the inputs, and for that combination, gives the corresponding output. Inputs and outputs are given in binary, with `0` or `L` meaning low and `1` or `H` meaning high. In addition, the symbols `x`, `X` and `-` are allowed as indications of don't care conditions. Don't care inputs are ignored, while don't care outputs have unspecified behavior.

Users should beware that use of `-` for don't care conditions may cause problems where no spaces are included between table columns because `--` is interpreted as a comment marker! Inclusion of blanks between table columns eliminates this problem, but there are contexts where this may be inconvenient, for example, where a sequence of table entries is grouped in order to emphasize its interpretation as a binary number.

The following example illustrates the tabular specification of a full adder.

```
table adder
  { binary full adder }
     A B Cin | S Cout
    ---------+--------
     0 0  0  | 0  0
     0 0  1  | 1  0
     0 1  0  | 1  0
     0 1  1  | 0  1

     1 0  0  | 1  0
     1 0  1  | 0  1
     1 1  0  | 0  1
     1 1  1  | 1  1
end
```

Note, in this specification, that the line of dashes below the column headers is interpreted as an end-of-line comment because it begins with two dashes. A circuit with exactly the same behavior could be specified as follows:

```
table adder
  { binary full adder }
     A B Cin | S Cout
    ---------+--------
     0 0  1  | 1  -
     0 1  0  | 1  -
     1 0  0  | 1  -
     1 1  1  | 1  -
```

```
    x 1  1  | -  1
    1 x  1  | -  1
    1 1  x  | -  1
end
```

In either case, an equivalent circuit could have been specified using parts and wires, with the specification beginning as follows:

```
circuit adder;
  { binary full adder }
  inputs  A, B, Cin;
  outputs S, Cout;
  parts
    ...
```

2.7.5 Arrays and Iterators

The Iowa Logic Specification Language allows inputs, outputs or parts to be declared as arrays. This is done by declaring a named range of allowed subscripts with a `range` declaration, and then using the range identifier as a subscript on the input, output, or part definition. Unnamed ranges may also be used by giving the bounds of the range in the array declaration, but it is usually preferable to declare named ranges. Declarations of ranges and named constants may appear anywhere a subcircuit declaration is allowed, that is, between the circuit header and the `input` keyword, or in a file included by a `use` construct. The outputs `o1` through `o4` and the parts `bit1` through `bit4` used in the `dregister` example in the previous section would much more appropriately have been declared as arrays. The following example illustrates this:

```
range   nibble = 1 .. 4;
inputs  i, c;
outputs o(nibble);
parts
        invert: not;
        bit(nibble): dlatch;
```

In this example, the output `o` is a 4 bit wide output array, with subscripts ranging from 1 to 4. Similarly, `bit` is a part array. The subscripts in a range declaration must be listed in ascending order!

Arrays are allowed in the specification of a truth table. For example, a 2-input 4-output decoder might be specified as follows:

```
table decoder;
 in(1..0) | out(1..4)
```

```
----------+----------
    0 0   |  1 0 0 0
    0 1   |  0 1 0 0
    1 0   |  0 0 1 0
    1 1   |  0 0 0 1
end.
```

The order of subscripts given in a table column is significant! In the above table, the inputs are, from left to right, `in(1)` and `in(0)` while the outputs are `out(1)` up to `out(4)`, also listed from left to right.

Note that the subscript bounds for a truth table must be integer constants! Use of symbolic names for such bounds poses problems, for example, because changes to the dimensions of a table typically involve adding rows or columns and are not a simple matter of renumbering.

Arrays of inputs, outputs, and parts may be interconnected one wire at a time, explicit iterators may be used, or implicit iterators may be used. In the context of the above definitions, it would be legal to make individual connections as follows:

```
wires
        i to bit(1).d;
        bit(1).q to o(1), bit(2).d;
        bit(2).q to o(2), bit(3).d;
        bit(3).q to o(3), bit(4).d;
        bit(4).q to o(4);
```

If such an interconnection pattern is desired, a much more direct expression of the pattern can be made with an iterator.

Iteration in the Iowa Logic Specification Language is done with a `for` construct; `for` constructs control the iteration of a wire list, and they may appear anywhere in the body of a wire list. This implies that `for` constructs may be nested. The following example illustrates the use of a `for` construct equivalent to the above:

```
wires
        i to bit(1).d;
        for j in first(nibble)..last(nibble)-1 do
            bit(j).q to o(j), bit(j+1).d;
        endfor;
        bit(4).q to o(4);
```

Note that the `for` construct declares an identifier, `j` in this example. The range of values over which this identifier is iterated is given by the expression between the keywords `in` and `do`.[2] The order in which values are assigned to `j` is not specified and has no effect on the meaning of the result.

[2] `in` is not really a keyword; it only appears that way in the context of the `for` construct.

The identifier declared by a `for` construct must not have been declared as a local identifier of the current circuit or of an enclosing `for` construct. The identifier may override an identifier of the same name declared in an outer block, and two `for` constructs in the same block may use the same identifier if neither encloses the other.

Implicit iteration may be applied to wiring outputs of one subcircuit to inputs of another. It is applicable whenever the source pin and destination pin of a wire list entry are arrays with numerically equal bounds. For example, if `byte` is a global range, perhaps declared as 8 bits, the following would be legal:

```
inputs  a(byte);
outputs b(byte);
wires
        a to b;
```

This single wire list entry actually implies the following `for` loop:

```
wires
        for i in byte do
            a(i) to b(i);
        endfor;
```

The expression giving the range over which a `for` loop iterates may be the name of a range or an explicit statement of the upper and lower bounds of the range. Although numerical bounds may be used, it is usually preferable to state the bounds in symbolic terms. The predicates `first`, `last`, and `size` are provided in order to simplify the construction of new ranges from old. These extract the lower bound of a range, the upper bound of a range, and the size of a range. For example, the following would be a legal iterator equivalent to the one used in the array version of `dregister`:

```
for i in first(nibble) .. first(nibble) + size(nibble) - 2 do
```

For the example declaration of `nibble`, this iterates over the range 1..3. Note that would have been legal to use the explicit range 1..3 in this context, but that this would make it hard to change the size of the register. A simple change to one range declaration is much easier than a search through the code for all related range expressions.

2.7.6 Constant Declarations

Constant declarations allow commonly used constants and the values of commonly used expressions to be named. The keywords `range`, `integer`, `real`, `boolean` and `time` introduce constant declarations for constants of the associated types. As already mentioned in the context of range constants, these may be put anywhere a subcircuit declaration is allowed. The following example illustrates a typical use of constant declarations:

```
range   word = 0 .. 15;
integer wordsize = size(word);
integer doublewordsize = 2 * wordsize;
range   doubleword = 0 .. doublewordsize - 1;
```

The types of numerical expressions follow from the types of their components in the usual ways. Values of type `time` obey the usual rules for time intervals; thus, times may be added or subtracted, but times may not be multiplied, and the ratio of two times is a real number, not a time. Ranges may be combined using the `|` operator; this is legal only when the upper bound of one range is exactly one less than the lower bound of the other.

The identifiers created by `range` and constant declarations obey all of the hierarchic scope rules of the specification language. Thus, it is legal, and in fact, common, to declare a range in the main circuit and then use that range in subcircuits. This provides a major abstraction facility, since it allows circuits to be interconnected without knowledge of the actual numbers of wires needed to interconnect them.

When declarations make heavy use of definition in terms of other declarations, it is handy to be able to document the assumptions made about them, and to have the system check those assumptions. The assert construct is used for this. Consider the following example:

```
circuit build( range word );
  assert  first(word) = 0;
  integer wordsize = size(word);
  assert  last(word) = wordsize - 1;
  integer doublewordsize = 2 * wordsize;
  range   doubleword = 0 .. doublewordsize - 1;
  assert  doubleword = (0 .. 2 * size(word) - 1);
```

This example does perhaps belabors the assertions, but documentating the assumptions made about the values can uncover a host of errors in a complex circuit specification.

2.7.7 Conditional Part and Wire lists

In some contexts, a subcircuit may need to expand differently depending on the context of the expansion; for example, additional gates may be needed if the number of elements in an array is greater than some threshold. In this case, conditional part and wire lists can be added. The notation used for this is based on the familiar `if then else endif` constructs of programming languages. Note that an `if` construct must always end with the keyword `endif`. Within an `if` construct, the phrase `else if` is allowed for testing secondary Boolean conditions; the phrase `else if` *does not* introduce a new `if` construct, it only introduces an alternate `then` clause.

For example, consider a circuit for adding two numbers. This might use the global range `word` to describe the structure of a word. Suppose that a simple adder, with ripple carry, is preferable when there are fewer than K bits in a word, but that fancy carry anticipation logic is desired for larger word sizes. A generic adder circuit could include conditional sections in its part and wire lists so that it selects one of the two implementations depending on the word size. This might be done as follows:

```
parts
        if size(word) < K then { use ripple carry }
                adders(word): fulladder;
        else { use carry anticipation }
                adders(word): addanticipate;
                lookahead(1 .. size(word)/4): anticipator;
        endif;
wires
        if size(word) < 1 then { something is wrong }
                error;
        else if size(word) < K then { use ripple carry }
                ...
        else { use carry anticipation }
                ...
        endif;
end { adder };
```

Note that the conditional structure of the parts list will frequently mirror that of the wire list. This is because each change in the parts list frequently requires a corresponding change in the wire list. It is legal to declare parts with the same name in both the `then` and `else` clauses, but this can be very confusing unless the parts correspond in some way. In the above example, the part `adders` has two possible definitions depending on the size of a word. This would be sensible if the circuit `addanticipate` is similar to `fuladder` in both function and input/output pin naming conventions. For example, `addanticipate` might have the same pin names as `fulladder` for the two addends, the sum, and the carry in and out, but it might have additional input and output pins for communication with the anticipation units.

The Boolean expression controlling an `if` construct must be composed using the usual relational and Boolean operators; these include `>`, `<`, `=`, `<=`, `>=` and `<>` (not equal), as well as `&` (and), `|` (or) and `\` (not), with parentheses allowed to control the order of evaluation. Integers, reals and times may be compared, each with its own types, and integers and reals may be freely mixed in comparison. Ranges may be compared only for inequality and equality.

2.7.8 Parameterized Circuits

It is sometimes useful to make generic subcircuits which can be parameterized when they are used in a parts list. For example, the predefined gates are parameterized with the number of input pins they allow, and they have an optional time delay as a parameter. Subcircuits may have a formal parameter list following the circuit name. This may include formal parameters of types `integer`, `real`, `time`, `range` or `circuit`. The following example illustrates this:

```
circuit generic( circuit x; integer n );
inputs  i, c;
outputs o(1..n);
parts   bit(1..n): x;
        invert: not;
wires
        i to bit(1).d;
        for j in 1 .. n-1 do
            bit(j).q to o(j), bit(j+1).d;
        endfor;
        bit(n).q to o(n);

        c to invert.in;
        for j in 1 .. n / 2 do
            c to bit(2 * j - 1).c;
            invert.out to bit(2 * j).c;
        endfor;
end;
```

This example will only work if the subcircuit passed as parameter `x` has inputs called `c` and `d` and an output called `q`. The subcircuit `dlatch` meets these criteria, so it would be legal to include `a: generic(dlatch,4)` in a parts list where `generic` and `dlatch` are visible. As a result of this, `a` would have exactly the same behavior as an instance of circuit `dregister` given previously.

The combination of parameterized circuits with conditional part and wire lists allows recursive circuit descriptions. For example, an n input decoder with k auxiliary enable inputs can be described either as an inverter and a pair of $k+1$ input and gates, in the case where $n = 1$, or as an inverter and a pair of $n-1$ input decoders with $k+1$ enable inputs, in the case where $n > 1$.

2.7.9 Time Delays

For most purposes, a logic description such as is given above will be sufficient, but there are times when it is desirable to simulate not only the static interconnection of a circuit, but also the timing characteristics of that circuit. The Iowa Logic Specification Language allows timing characteristics to be specified for both logic gates and wires. If timing characteristics are not given, the logic simulator will assume defaults typical of kluge card or breadboard implementations of TTL circuits (gates are given a nominal propagation delay

10ns (nanoseconds), and wires are given delays between 0.5ns and 1.5ns, corresponding to wires from about 15cm to 45cm long).

If the default gate propagation delay is not appropriate, an explicit delay can be specified as an additional parameter on the part type. This is illustrated in the following example:

```
parts
        a: not;
        b: not(1.5 * ns);
        c: nand(2);
        d: nand(2, 2 * ns);
```

In this example, gates `a` and `c` have the default delay, while gates `b` and `d` are relatively fast gates, with delays of 1.5 and 2 nanoseconds.

Delays are expressions or constants of type `time`. These are usually constructed by multiplying or dividing a predefined time constant by a factor, as in the above examples. Times may be added or subtracted to yield new times, but they may not be multiplied, and division of two times yields a real number, not a time. The predefined constants of type `time` are:

`s` – second
`ms` – millisecond (`ms` = `s`/1000)
`us` – microsecond (`us` = `ms`/1000)
`ns` – nanosecond (`ns` = `us`/1000)

The symbol `u` in the abbreviation of microseconds is used to approximate the shape of the Greek letter μ (mu) which is a standard abbreviation for the prefix *micro*.

It should be noted that, during the simulation of a circuit, the propagation delay specified for a gate is modified by a small random factor each time it is used. Thus, a gate with a nominal 10ns delay will have actual delays ranging from 9.5ns to 10.5ns. This *jitter* in the propagation delay results in slightly unpredictable results for some circuits, but it is necessary to make flipflops decay realistically to one or the other of their stable states after being simultaneously set and reset.

If the default signal delay along a wire is not appropriate, a specific delay can be given as a parameter on the `to` keyword used in the wire list part of the circuit description, as shown in the following example.

```
wires
        a.out to(15 * ns) c.in(1);
        a.out to(1 * ns) c.in(2), b.in;
```

Here, note that all of the destination connections listed on any particular `to` clause have the same delay, so a separate `to` clause must be used for each different delay which is specified. In the above example, `a.out` is connected to three different inputs, using two different delays.

If subcircuits are used, delays may be specified in both the connections to an instance of a subcircuit, and in the connections from the inputs and outputs of the subcircuit to its components. The Iowa Logic Simulator will correctly sum these delays in order to determine the total time taken by a signal as it travels from one component to another.

Some circuits require periodic clocks for their operation; these may be constructed from an inverter with its output fed back into its input. The period of the clock will be twice the delay of the inverter, so a clock with a period of 1μs may be constructed from an inverter with a delay of 500ns; the same effect may be obtained by using an inverter with a shorter delay and putting extra delay in the wire connecting the output of the inverter back to its input. Note that the technology used to construct a gate determines its delay, and thus, that real gates have nothing analogous to a delay parameter. Furthermore, long wires used to achieve long delays are impractical: a 500ns delay would require about 150 meters of wire, or perhaps 50 meters of coaxial cable. Thus, the periodic clocks used in real systems usually rely on more complex electronic solutions.

By default, the time taken to evaluate a truth table will be the composite of the default times taken to propagate signals through 3 levels of logic gates plus 4 layers of interconnecting wires; this default behavior can be overridden by specifying a time formal parameter to the table, as follows:

```
table inverter( time );
   in | out
  ----+-----
    0 |  1
    1 |  0
end
```

Given such a specification, each instance of the part defined by the table must be given a specific time delay, for example, as follows:

```
parts
not1: inverter( 10 * ns );
not2: inverter( 2 * s ); -- a very slow inverter!
```

2.8 Formalities

The Iowa Logic Specification Language includes the following reserved words:

```
assert          for             range
boolean         if              real
circuit         inputs          table
do              integer         then
else            mod             time
```

end	outputs	to
endfor	parts	use
endif		wires

In addition to these words, the identifier `in` appears to be a keyword in the context of the third lexeme of a `for` construct, but not elsewhere. The identifier `tally` may be placed before the keyword `circuit` in the heading of the main circuit to request a tally of all parts and subcircuits used in building that circuit. No predefined meaning is ascribed to `tally` in any other context, and `tally` may be freely redefined.

The following identifiers are predefined as input or output pins of predefined gates. With the exception of `in`, they have no meaning elsewhere, and their redefinition will not have any effect on their meaning in the context of the predefined gates or in the context of the `for` construct.

in	control	data	out

The following identifiers are the names of predefined global inputs to the circuit; they may be redefined, but such redefinition will mask their predefined meaning.

high	low

The following identifiers are predefined constants; they may be redefined, but such redefinition will mask their predefined meaning. Of these, `true` and `false` have Boolean values, the others are time units.

true	s	ms
false	ns	us

The following identifiers are names of predefined gates; they may be redefined, but such redefinition will mask their predefined meaning.

and	latch	not	tsgate
bus	nand	ntsgate	xor
equ	nor	or	

Note that `and`, `or`, and `not` are not legal operators in Boolean expressions, but that the symbols `&`, `|` and `\` are used in expressions. The `and`, `nand`, `or` and `nor` gates have a required parameter, the number of inputs; all gates other than the bus have a second optional parameter giving the time delay of the gate.

The following identifiers are names of predefined functions which may be used within expressions; they may be redefined, but such redefinition will mask their predefined meaning. The first three take a single range expression as an argument and return, respectively, the first element of the range, the last element of the range, and the number of elements in the range. The last two take integer arguments and return, respectively, whether the integer was odd and the number of one bits in the binary representation of the integer.

```
first          last          size
odd            onebits
```

Predefined identifiers are considered to be declared globally to the main circuit. Local declarations within any circuit or subcircuit may redefine identifiers declared outside that circuit or subcircuit. It is illegal to redefine an identifier which was locally defined within a circuit or subcircuit. Although the identifier declared by a `for` construct has a scope limited to the `for` construct body, it may not have the same name as any locally declared identifier within the same circuit or subcircuit, and it may not have the same name as the identifier declared by any enclosing `for` construct. It is legal for two `for` constructs in the same wire list to declare the same identifier as long as the two are not nested.

Expressions may be composed using the following operators, listed in order of decreasing precedence:

```
      (x) f(x) \      – nested parentheses, functions, Boolean not
          **          – integer exponentiate
      * / mod &       – multiply, divide, remainder, Boolean and
        + - |         – add, subtract, Boolean or
.. < <= = <> >= >     – range construction, comparison
```

The logic specification language recognizes expressions of type integer, real, time, Boolean, and range. Constants and formal parameters of these types may be declared using the corresponding keywords. In addition, the keyword `circuit` may be used to declare formal parameters of that type.

The following Extended BNF grammar describes the Iowa Logic Specification Language. The notation used here is traditional: angle brackets surround nonterminal symbols, square brackets surround optional constructs, curly braces surround constructs that may be repeated more than one time, and a vertical bar is used to separate alternatives. When one of these metasymbols also occurs as a symbol in the language, it is quoted. Lexical details are not covered by this grammar. There are two top-level constructs, files containing a main circuit, and files intended to be included by a `use` directive; these are defined first.

```
<main file> ::=
        [ tally ] <circuit or table declaration> [ . ]

<included file> ::=
        <subcircuit and constant declarations> [ . ]

<circuit or table declaration> ::=
        <circuit declaration> | <table declaration>

<circuit declaration> ::=
        circuit <identifier> [ <formal parameter list> ] [ ; ]
```

```
        [ <subcircuit and constant declarations> [ ; ] ]

        [ inputs
                <identifier list> [ ; ] ]
        [ outputs
                <identifier list> [ ; ] ]
        [ parts
                <part list> ]
        [ wires
                <wire list> ]
        end

<formal parameter list> ::=
        ( <formal declaration> { [ ; ] <formal declaration> } )

<formal declaration> ::=
        <formal keyword> <identifier list>

<formal keyword> ::=
        boolean | integer | real | time | range | circuit

<identifier list> ::=
        <identifier> { [ , ] <identifier> }

<subcircuit and constant declarations> ::=
        <declaration> { [ ; ] <declaration> }

<declaration> ::=
        <constant declaration>
      | <use directive>
      | <assertion>
      | <circuit or table declaration>

<constant declaration> ::=
        <constant keyword> { <identifier> = <expression> }

<constant keyword> ::=
        boolean | integer | real | time | range

<use directive> ::=
        use <file name>
```

```
<assertion> ::=
        assert <expression>

<part list> ::=
        { <part declaration> | <part list if> }

<part declaration> ::=
        <part names> : <part type> [ ; ]

<part names> ::=
        <part name> { [ , ] <part name> }

<part name> ::=
        <identifier> [ ( <expression> ) ]

<part type> ::=
        <identifier> [ ( <actual parameter list> ) ]

<actual parameter list> ::=
        <expression> { [ , ] <expression> }

<part list if> ::=
        if <expression> then
            <part list>
        [ else if <expression> then
            <part list> ]
        [ else
            <part list> ]
        endif [ ; ]

<wire list> ::=
        { <wire list element> | <wire list if> | <wire list for> }

<wire list element> ::=
        <pin> to [ ( <expression> ) ] <pin> { [ , ] <pin> } [ ; ]

<pin> ::=
        <identifier> [ ( <expression> ) ]
             [ . <identifier> [ ( <expression> ) ] ]

<wire list if> ::=
        if <expression> then
```

```
            <wire list>
        [ else if <expression> then
            <wire list> ]
        [ else
            <wire list> ]
        endif [ ; ]

<wire list for> ::=
        for <identifier> in <expression> do
            <wire list>
        endfor [ ; ]

<expression> ::=
        <simple expression> [ <relop> <simple expression> ]

<relop> ::=
        < | <= | <> | = | > | >= | ..

<simple expression> ::=
        [ + | - ] <term> { <addop> <term> }

<addop> ::=
        + | - | "|"

<term> ::=
        <factor> { <mulop> <factor> }

<mulop> ::=
        * | / | mod | &

<factor> ::=
        <fact> [ ** <fact> ]

<fact> ::=
        <identifer> | <function> | <number> | \ <fact> | ( <expression> )

<function> ::=
        <identifier> ( <expression> )

<table declaration> ::=
        table <identifier> [ ; ]
```

```
        <input names> | <output names>

        {  <inputs>   |   <outputs>  }

        end

<input names> ::=
        <table pin> { <table pin> }

<output names> ::=
        <table pin> { <table pin> }

<table pin> ::=
<identifier> [ ( <number> .. <number> ) ]

<inputs> ::=
        <table value> { <table value> }

<outputs> ::=
        <table value> { <table value> }

<table value> ::=
0 | 1 | x | X | -
```

This grammar is ambiguous. One of the major problems is the keyword sequence `else if`; the parser is greedy, and as a result, it cannot parse an `if` construct at the start of a list of items in an `else` clause. The next version of the language will correct this problem by introducing the keyword `elseif`.

Chapter 3

Using the Logic Simulator

After having edited or created a circuit description file, if the file does not use truth tables introduced by the keyword `table`, you can execute the logic simulator by typing

```
logicsim
```

on the system you are using. In response to this, the logic simulator will output the following:

```
IOWA Logic Simulator,  Ver. 10
Source file name:
```

At this point, type in the file name of the file describing the circuit you wish to test. For example, type `dlatch` to simulate the example circuit, assuming the example circuit description is in file `dlatch`. After the file name is given, the system will compile your circuit description file. If there are syntax errors, the system will print out the error messages and terminate. When this happens, correct the errors and try again.

Circuits or subcircuits containing truth tables must be preprocessed through a table preprocessor. This preprocessor compiles the tabular specification of a circuit or subcircuit into an equivalent circuit implemented using parts and wires. This compilation is reasonably efficient, but no optimization is done. It is up to the designer to optimize boolean functions if the number of gates or interconnecting wires is to be minimized.

Note that the logic simulator compiles the circuit description into an internal representation quite different from the textual representation of the specification language. Users of the simulator need not understand this representation, but the following description may be useful in understanding how the simulator works. The internal representation is not machine code or assembly code; instead, each gate is represented by a record with information about the state of that gate and a pointer to a linked list of wires connected to the output of that gate. Each wire record in this list holds a pointer to some other gates and a marker indicating which input of that gate it drives. This data structure is designed to support very fast simulation.

In the internal representation of a circuit, subcircuits and arrays have been replaced by data structures that are the same as the data structures that would have been used had the same circuit been described without subcircuits or arrays. Although the output of the compiler is not textual, the process by which subcircuits are processed resembles macro expansion. As a result, the decision to use or not use subcircuits or arrays should not be influenced by any consideration of simulation efficiency.

3.1 The Simulation Header

If there are no errors in the circuit, the logic simulator will print out the simulation header. For the example test file, this would be as follows:

```
==========================================================
   circuit name   = dlatch
   input interval = 1.0us    If you need help, type h.
   output interval= 500ns
----------------------------------------------------------
TIME:OUTPUT                 :INPUT
----:------                 :-----
500 :q  :qbar               : d
 NS : | : | :do :co         : |c
    : | : | : | : | :       : ||
==========================================================
   0:|  :|  :|  :|  :INPUT:
```

This header tells you the initial status of the simulator. The components of the header are explained in the following paragraphs:

`circuit name:` Gives the circuit name specified in the circuit description.

`input interval:` Gives the simulated time interval between consecutive `INPUT:` prompts in the following simulation. The default input interval value is 1 microsecond. That is, when the next `INPUT:` prompt comes out, 1μs of simulated time will have elapsed since the previous prompt. (See the advanced features section if you wish to override this default.)

`output interval:` Gives the simulated time interval between two consecutive output display lines. The default output interval value is 500 nanoseconds. That is, when the next output display line comes out, 500ns of simulated time will have elapsed. (See the advanced features section if you wish to override this default.)

The rest of the header is a table with three columns, a `TIME` column, a `OUTPUT` column, and a `INPUT` column. These serve as the labels on the three columns of simulation output.

`TIME:` The first column of the simulation output gives the time elapsed since the start of the simulation, in multiples of the output interval.

`OUTPUT:` The second column of the output has a sub-column to display each output pin of the circuit being simulated. The header includes the name of each output pin at the top of the appropriate sub-column; these names are taken from the text of the circuit description. Note that output pin names longer than three characters force the use of additional lines in the header.

`INPUT:` The last column is reserved for input to the simulator. The header for this column lists the names of each input pin of the circuit being simulated. These are listed one per line and one per column.

It is worth noting that on an 80 column display, there is room for only a limited number of inputs and outputs. The fixed parts of the display format require 12 columns, each output requires 4 columns, and each input requires 1 column. There is guaranteed to be room for 4 inputs, leaving room for a maximum of 16 outputs. Circuits can be constructed with more inputs and outputs, but their output will only be readable if printed on a line printer.

Following the header, the simulator prints the initial values of each output and the initial `INPUT:` prompt. The format for these is explained in the next section. Outputs are initially low (0 or false) when the simulated power is turned on, and all inputs are low prior to your entering the first input value. This initial state is unstable, since it means that some gates may start out with outputs that are not right for their inputs. As a result, even if you don't change the input values, some of the outputs will immediately change from 0 to 1. If your circuit has any feedback loops in it, for example, if it contains flipflops, it may oscillate between 0 and 1 for some time as a result of these initial changes; this oscillation may consume considerable computing time and it may delay the `INPUT:` prompt significantly.

3.2 Running the Simulation

When an `INPUT:` prompt appears, the system is in simulation mode. Each time this prompt appears, you can give new input values for each input of the circuit. The legal input values are:

`0` or `L`	– a low (false) logic level.
`1` or `H`	– a high (true) logic level.
`p`	– a pulse.
`x`,blank	– repeats the previous logic level.
empty line	– all input values remain unchanged.

If the input interval is 1 microsecond, there will be a new `INPUT:` prompt every microsecond of simulated time. Note that simulated time has nothing to do with real time. Making the input interval smaller will not necessarily make the simulator do any more or less work. This is because the simulator uses what is called a *discrete event* simulation model; in this model, it takes a (roughly) constant amount of computer time to simulate each change in the input of a gate from high to low or from low to high, completely independent of the input or output intervals. Every transition which takes place within a circuit is simulated, whether or not that transition has any observable effect on the outputs of the circuit. Note that this implies that the accuracy of the simulation is not influenced by the input or output interval! A long output interval only limits the accuracy with which simulation results are reported, not the accuracy with which they are computed. Some other methods of simulation involve an internal clock; such systems tend to give more accurate results if the clock is run at higher speeds.

Typing `11` in response to the `INPUT:` prompt for the example circuit indicates that both inputs `d` and `c` are assigned the value 1 (high), since a `1` appears in each column. After being given values, the system simulates all logic transitions which occur as a result of input changes. It also samples all the output values at the instant when one output interval expires and displays the sampled data. After the input `11`, the following will be on the display screen:

```
OK, logicsim
IOWA Logic Simulator,  Ver. 10
source file name: dlatch
=========================================================
   circuit name   = dlatch
   input interval = 1.0us    If you need help, type h.
   output interval= 500ns
---------------------------------------------------------
TIME:OUTPUT               :INPUT
----:------               :-----
 500:q   :qbar            : d
 ns : | : | :do :co       : |c
    : | : | : | : | :     : ||
=========================================================
   0:|  :|  :|  :|  :INPUT: 11
   1:|= :|= :|_ :|_ :     : 11
   2:  |:|  :  |:  |:INPUT:
```

There are two new lines of output because the input interval is twice the output interval. The input interval should usually be an integer multiple of the output interval.

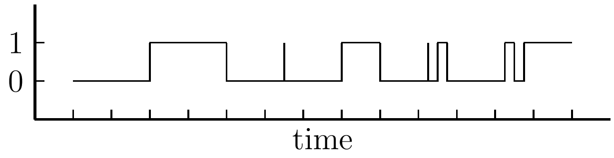
Typing a `p` as an input value causes the simulator to switch the indicated input from its previous value to the inverse of that value for roughly half of an input inverval, and then switch it back to the original value. Thus, a positive pulse will result if the previous value was low, and a negative pulse will result if the previous value was high. The pulse created by `p` will be roughly centered in the interval between successive `INPUT:` prompts.

The input/output formats of the logic simulator are similar to those of a tool called a *logic analyzer*. Logic analyzers are used in testing and debugging digital hardware; they have a graphical display which will show the outputs of a digital system plotted as a function of time, and they are able to inject prepared sequences of test data into the inputs of the circuit. Because common computer display screens can scroll vertically, but not horizontally, the output of the simulator is turned 90 degrees from its conventional orientation. Because of character set limitations, transitions resulting from input changes will be displayed at the end of the next output interval and not immediately after the change. The latter leads to an apparent delay of one output interval in all outputs from the circuit. (If the ASCII character set had an overbar as well as an underline, this problem would be eliminated.)

The values of circuit outputs are displayed by the logic simulator as follows:

```
:|   : – a low (false, 0) logic level.
:|_  : – a transition from low to high.
:   |: – a high (true, 1) logic level.
:  _|: – a transition from high to low.
:|   :
:|-  : – a short spike or pulse (2 transitions).
:|_  :
:  _|: – a longer pulse (1 transition per line).
:|   :
:|=  : – multiple spikes (4 or more transitions).
:|   :
:|=  : – a dirty transition (3 or more transitions).
:   |:
```

On a logic analyzer, the above output trace might have been shown as follows:



Note that, when a spike, multiple spikes, or a dirty transition is shown, repeating the experiment with a smaller output interval will show more detail, allowing each transition to be shown on a different line. This corresponds to stretching the time axis of the display.

When a the simulation of a circuit begins, flipflops will usually produce multiple spikes for a considerable time; this is a simulation of the behavior of real flipflops when the power is turned on. Flipflop outputs will usually settle into a stable state after a few microseconds, but they can be explicitly set or reset immediately, as was done with the example input for testing the `dlatch` circuit. It may take a considerable amount of computer time to simulate the power-on behavior of circuits with many flipflops, but this can usually be avoided by using the initial low value of some input to explicitly clear all flipflops in the circuit. Of course, if you do this, the circuit won't work at all until you enter a high value for this input.

The globally predefined `high` and `low` inputs provided by the simulator can be used to make an automatic power-on-reset circuit. The key to this is that the simulator provides a low-to-high transition on `high` immediately after simulating the power-on-transient at each gate. If an appropriate delay element is connected to the `high` input, a signal can be produced which will remain low for a period after the simulation begins, and then go high for the remainder of the simulation.

3.3 Commands to the Simulator

There are a number of special commands which the simulator will accept at the `INPUT:` prompt. With the exception of the `quit` command, these commands will be executed and then the output from the simulator will be re-displayed and a new `INPUT:` prompt will be given as if no simulated time had elapsed. The legal commands may all be abbreviated to one letter; thus, `quit` may be abbreviated `q`.

`quit:` To stop the simulator, type `q` or `quit`.

`input:` To change the input interval, type `input␣nn␣uu`, where `nn` is a positive number and `uu` is one of `s`, `ms`, `us`, or `ns`. Large input intervals will allow many output lines to be displayed between `INPUT:` prompts. The input interval should usually be an integer multiple of the output interval.

`output:` To change the output interval, type `output␣nn␣uu`, where `nn` is a positive number and `uu` is one of `s`, `ms`, `us`, or `ns`. Small output intervals allow detals of multiple transitions to be observed with each transition on a line by itself. Note that the input interval should not be smaller than output interval.

`read:` To read test data from a file other than the terminal, type `read␣ff`, where `ff` is the name of the new input file. This command is useful because it allows predefined test sequences to be used over and over, for example, while debugging a circuit, without the need to retype the data each time the circuit is tested. For example, consider the following test sequence stored in a file called `example`.

```
o 100ns
i 0.2us
11
10 -- a comment
01
11
q
```

The output from a session in which this command is used is shown below:

```
OK, logicsim
IOWA Logic Simulator,  Ver. 10
source file name: dlatch
==============================================================
```

```
   circuit name   = dlatch
   input interval = 1.0us    If you need help, type h.
   output interval= 500ns
--------------------------------------------------------------
TIME:OUTPUT               :INPUT
----:------               :-----
 500:q  :qbar             : d
 NS : | : | :do :co       : |c
    : | : | : | : | :     : ||
==============================================================
   0:|  :|  :|  :|  :INPUT: r example
   1:|  :|  :|  :|  :INPUT: o 100ns
   1:|  :|  :|  :|  :INPUT: i 0.2us
   1:|  :|  :|  :|  :INPUT: 11
   2:|= :|= :|_ :|_ :     : 11
   3:  |:|  :  |:  |:INPUT: 10 -- a comment
   4:  |:|  :  |: _|:     : 10
   5:  |:|  :  |:|  :INPUT: 01
   6: _|:|_ : _|:|_ :     : 01
   7:|  :  |:|  :  |:INPUT: 11
   8:|_ : _|:|_ :  |:     : 11
   9:  |:|  :  |:  |:INPUT: q

  Simulation ends.
OK,
```

Note that each line of the file contains only one command or one set of input values. Note also that comments on an input line are allowed as long as there is at least one space separating the comment from the last column associated with an input pin. If you want a paper copy of the output produced by a session with the simulator, redirect the output before you start the simulator.[1]

`title`: The `title` command reprints the simulation header. This would have been useful in the above example after the input and output intervals had been reset. This is also useful if the header scrolls off the screen and you forget which inputs are which.

`help`: Help messages for all system facilities are available. Whenever you need help, try typing `h` and see what help the logic simulator can give you.

[1]use the Prime `como` command, or the UNIX `script` command for this.

3.4 Simulator Error Messages

All error messages produced by the Iowa Logic Simulator during the compilation of a circuit description have the following format:

```
ERROR on line 999 of file XXX
      ---> MMM
```

The first line indicates the line number of the file on which the error was detected, and the second line indicates the nature of the error. The compiler operates in two passes, and if the first pass detects any errors, the second pass will be aborted. Within a pass, the compiler will usually report all errors it detects, tempered by an attempt to avoid reporting the same error more than once; for some errors, however, compilation will be aborted with the following message:

```
Compilation aborted; fatal error.
```

3.4.1 First Pass Error Messages

`Pascal run-time execution errors:` An attempt to read from a file which does not exist will raise a Pascal run-time execution error in the machine independent version of the logic simulator. Other run-time errors should be reported, and any code that causes them should be saved so that the error can be recreated.

`string pool overflow:` A fatal error. There were too many total characters in all identifiers in the circuit description. The best fix for this is to recompile the logic simulator with a larger string pool, but if this is impractical, use shorter identifier names, or re-use local identifier names from one subcircuit to the next.

`packed text overflow:` A fatal error. The total number of lexemes in the entire circuit description, including the text of all subcircuits, was too large. The best fix is to recompile the logic simulator with a larger packed text pool, but if this is impractical, try to make more effective use of subcircuits, arrays, and iterators to shorten the circuit description.

`symbol table overflow:` A fatal error. There were too many distinct identifiers in the entire circuit description. The best fix is to recompile the logic simulator with a bigger symbol table, but if this is impractical, use fewer distinct identifiers by re-using local names from one subcircuit to the next.

`unexpected EOF:` A fatal error. An end-of file was found while scanning a circuit body for the keyword `end`.

`"circuit" or "inputs" expected:` During the scan for subcircuit and constant declarations at the head of a circuit, something was found other than one of the keywords that can start a declaration (`circuit`, `boolean`, `integer`, `real`, `time`, `time` or `use`) or a keyword indicating start of of the body of that circuit (`inputs`, or `outputs`).

`identifier expected:` The keyword `circuit` was not followed by an identifier giving the circuit name.

`too many use levels; limit = 4:` A fatal error. The number of levels of nesting of files included by the `use` directive was too large. The logic simulator can have no more than 4 open input files at a time. The solution is to move `use` directives to the global level, so that instead of having the main file include another, which in turn includes yet another, all included files are directly referenced by the main file. Alternately, simply replace some of the more deeply nested `use` directives by the text of the included files.

`"circuit" expected:` The main circuit did not begin with the keyword `circuit`.

`EOF expected:` There was something after the end of the main circuit, or a file included by the `use` directive had something in it other than declarations of constants and subcircuits. Keep in mind that nothing but blank space may follow the (optional) period at the end of a file.

`bad number:` A number was specified with too many digits.

`")" expected in parameter list:` The formal parameter list of a circuit included a keyword illegal in the formal parameter list, or the end parenthesis was missing from the formal parameter list causing the entire circuit body to be interpreted as part of the list.

`unindentifiable lexeme:` A strange character was found in the input text.

3.4.2 Second Pass Error Messages

`x found; y expected:` This message is used for almost all syntax errors. The token `x` was found in a context where it is not legal; `y` is one of the tokens (or the name of a token class) that would be appropriate in this context.

`x: illegal redefinition:` An attempt was made to redefine the identifier `x` in a context where that identifier already had a local definition.

`too many (actual)parms:` The number of actual parameters provided in the declaration of a part was more than the number expected by the part type being instantiated.

`too few (actual)parms:` While parsing a formal parameter list and binding the actual parameters to formal parameters, there were not enough actual parameters to bind with each formal parameter. Or, a subcircuit was declared with over 20 formal parameters, in which case, the logic simulator will need to be recompiled to increase this limit.

`x: invalid gate/circuit name:` The identifier `x` was found after the colon in a part declaration and `x` was not the name of a subcircuit or a predefined gate type.

`x: invalid pin name:` The identifier `x` was found after the dot in a pin name in a wire list and `x` was not the name of an input or output of that subcircuit or predefined gate.

`x: invalid part name:` The identifier `x` was found before the dot (if there was one) in a pin name in a wire list and `x` was not the name of a declared part.

`x: unmodifiable name:` The identifier `x` was found before a dot in a pin name, but was not modifiable by the suffix provided, for example, because `x` was the name of an input or output to the current circuit or subcircuit.

`x.y: is illegally reused:` The pin `x.y` used as the destination of a wire already had another wire connected to it.

`assertion failure:` The boolean expression on an assert construct had a negative value.

`no destination given:` In parsing a wire list entry, a source pin was found and the keyword `to` was expected but missing.

`is not a source:` In the wire list of a circuit, the input of a gate or subcircuit instance was used as a source pin, or an output of the current circuit was used as a source pin.

`is not a destination:` In the wire list of a circuit, the output of a gate or subcircuit instance was used as a destination pin, or an input of the current subcircuit was used as a destination pin.

`x.y: has an open source connection:` This error message indicates that no wire listed in the current subcircuit provided a source of data for `x.y`. After parsing the wire list of a circuit or subcircuit, a check is made to see that all outputs of that circuit or subcircuit were connected to some source of data, and a check is made that all inputs of all subcircuits of that circuit or subcircuit are connected to a source of data. As a result, this error message always lists the line number of the keyword `end` marking the end of the subcircuit where the wire is missing.

`x: undeclared:` The identifier `x` was used in an expression but had no declared meaning.

`x: value is of wrong type:` The identifier `x` was used in an expression and was declared as the name of an instance of a gate or subcircuit, or the name of an input or output pin. Or, in the declaration of the constant `x`, the type of the expression giving the value did not match the keyword used in the declaration. In the latter case, an extra line is added to the error message giving the value of the expression that was found.

`operands of " " incompatible:` An attempt was made to apply one of the arithmetic operations to values that could not be operated on by those operations; for example, adding an integer to a time, or multiplying a subcircuit times a range.

`incomparable operands:` An attempt was made to apply a comparison operator to values that could not be compared; for example, times cannot be compared with anything other than times, and ranges and circuits cannot be compared at all.

`boolean expected:` A Boolean operator such as and, or or not was applied to a value which was not Boolean, or the expression governing an `if` construct in a parts list or wire list was not Boolean, or the expression on an assert construct was not Boolean. In both cases, an additional line gives the actual value of the expression or operand that was found.

`integer expected:` The predicate `odd` or `onebits` was applied to a non integer, an operand of the `mod` operator was not an integer, an attempt was made to exponentiate a non-integer, a range constructor was used with non-integer bounds, or a non-integer array subscript was used. In all cases, an additional line gives the actual value of the expression or operand that was found.

`positive integer expected:` An attempt was made to raise a number to a negative or non-integer power, an attempt was made to raise zero to the zero power, integer division or `mod` involving negative numbers was attempted, division by zero was attempted, or an attempt was made to instantiate a pre-defined gate type with a negative or zero number of inputs. In all cases, an additional line gives the actual value of the expression or operand that was found.

`nonzero real expected:` Division by a real or time value of zero was attempted. An additional line gives the value of the expression or operand that was found.

`positive time expected:` The delay specified for a gate or wire was less than zero or not a time. In all cases, an additional line gives the actual value of the expression or operand that was found.

`range expected:` The `size`, `first` or `last` operator was applied to a value that was not a range, or in the declaration of a part or an input or output pin of a circuit, the syntax of an array specification was used with a value that was not a range, or the expression governing the iteration in a `for` construct was not a range. In all cases, an additional line gives the actual value of the expression or operand that was found.

`index out of bounds:` The pin number used to select an input of a predefined gate was out of bounds, or an array index was out of bounds. In both cases, an additional line gives the actual value of the expression or operand that was found.

`x: not an array name:` A subscript was used on `x`, a part name or pin name that was not the name of an array of parts or pins.

`x: unmatched parameter types:` An attempt was made to bind the formal parameter `x` to an actual parameter of the wrong type. An additional line gives the value of the actual parameter.

`unmatched bounds:` An attempt was made to wire a source to a destination where the array bounds of the two did not match or where one was an array and one was not an array, or an attempt was made to concatenate two ranges where the upper bound of one range was not exactly one less than the lower bound of the other.

`x: index required for this part:` The name `x` of an array of parts was used without a subscript in specifying a pin to be connected in a wire list.

3.4.3 Rewrite Phase Messages and Warnings

`"x" is involved in a circular interconnection of tristate busses:` indicates that x is involved in a loop where one tristate bus drives another tristate bus that in turn drives the first. The logic simulator cannot properly simulate loops of busses, so the simulation will be aborted.

`"x" drives more than one tristate bus in parallel:` indicates that the gate named x had its output connected to more than one tristate bus. Sometimes, the logic simulator can successfully rewrite the model so that the results will be correct, but there is no guarantee that the rewrite rules accomplish this goal.

3.4.4 Simulation Error Messages

Most interactive simulator error messages should be self-explanatory and offer interactive help in correcting the error. The following error message will sometimes result from actual errors in the circuit being simulated, however, and can only be eliminated by changes to the circuit design:

`Warning, bus "x" was driven with conflicting inputs:` The tristate bus named `x` had more than one bus-driver enabled at the same time, and two or more of the enbled drivers were attempting to drive the bus to differing values at the same time. In a real system, this can cause physical damage, overheating, or electrical noise and transient malfunctions; in the simulated system, it will cause no damage but the value on the three-state bus will be unpredictable until the conflict is resolved.

Chapter 4

74LS TTL Components

A standard set of subcircuits is provided with the logic simulator for designing logic circuits using standard TTL chips. The TTL family of integrated circuits was introduced about 20 years ago by Texas Instruments. TTL stands for *Transistor Transistor Logic*, which signifies that two transistors are used to drive each output of each chip, one for pulling the output down to a low level, and one for pulling the output up to a high level. Chips made using TTL technology are faster than the older RTL (*Resistor Transistor Logic*) and DTL (*Diode Transistor Logic*) families of integrated circuits, and they consume more power than the MOS (*Metal Oxide Semiconductor*) technology used in most VLSI (*Very Large Scale Integrated circuit*) chips.

The TTL family has at least 6 subfamilies which offer different speed/power tradeoffs. These are summarized in the following table, where they are listed in roughly the order in which they were introduced:

family		delay (ns)	power (mW)
basic		10	10
low-power	L	35	1
Schottky	S	3	18
low-power Schottky	LS	9	2
advanced Schottky	AS	1.5	10
advanced low-power Schottky	ALS	4	1

The term *Schottky* refers to a technology for making faster transistors. It is interesting to note that, in each generation of the TTL family, the low power representative of that generation is about 3 times slower than the other member, but consumes about 1/10 the power. Today, the low-power Schottky subfamily is the most widely used member of the TTL family; chips from this family are available from a wide variety of manufacturers

All manufacturers of TTL chips use a common naming system, as exemplified by the chip name "SN74LS00". The prefix SN indicates that the chip was made by Texas Instruments; other manufacturers have their own prefix codes, but if the remainder of the chip name matches, the chips should perform exactly the same function. Additional one letter codes may be added as prefixes or suffixes to this code, for example, RSN indicates radiation hardened chips made by Texas Instruments, and SNM indicates the use of quality control procedures specified by the military specification MIL-STD-883. The numeric code 74 indicates that the chip conforms to the requirements of the civilian computer industry, being able to operate over a temperature range of 0° to 70°C, while the code 54 indicates the ability to operate over the more extreme temperature range of -55° to 125°C required by many military and industrial applications. The letters LS indicate which subfamily the chip belongs to. Finally, the last two digits indicate the logical function performed by the chip.

Each Iowa Logic Specification Language TTL subcircuit description corresponds to one standard chip, and these chips are packaged one per file, in a form appropriate for inclusion using the `use` construct. The actual file names of these circuits will depend on your installation, but in general, the prefix indicating the manufacturer and military rating will not be included in the name, since it conveys nothing about the logical function of the chip. The following circuit descriptions are currently available (listed in numeric order):

LS00 Quad 2-input nand gates.

LS02 Quad 2-input nor gates.

LS04 Hex inverters.

LS08 Quad 2-input and gates.

LS10 Triple 3-input nand gates.

LS11 Triple 3-input and gates.

LS20 Dual 4-input nand gates.

LS21 Dual 4-input and gates.

LS27 Triple 3-input nor gates.

LS30 8-input nand gate.

LS32 Quad 2-input or gates.

LS40 Dual 4-input nand buffers.

LS42 BCD-to-decimal decoder (or 3-line to 8-line decoder with enable).

LS74A Dual positive-edge-triggered D flipflop.

LS75 Quad D latch

LS85 4-bit binary magnitude comparator.

LS86 Quad 2-input exclusive-or gates.

LS109A Dual positive-edge-triggered J-$\overline{\text{K}}$ flipflop.

LS125A Quad bus-buffer gates with three-state outputs.

S133 13-input nand gate.

LS139 Dual 2-line to 4-line decoders/demultiplexers.

LS153 Dual 4-line to 1-line data selectors/multiplexers.

LS157 Quad 2-line to 1-line data selectors/multiplexers.

LS158 Quad 2-line to 1-line mux (as LS157), with inverted outputs.

LS161A Synchronous 4-bit binary counter.

LS164 8-bit serial to parallel shift register.

LS166 8-bit parallel to serial shift register.

LS174 Hex positive-edge-triggered D flipflops with common clock and clear.

LS175 Quad positive-edge-triggered D flipflops with common clock and clear.

S182 Look-ahead carry generators (see LS381A).

LS183 Dual full adders.

LS240 Octal inverting three-state driver.

LS244 Octal non-inverting three-state driver.

LS273 Octal positive-edge-triggered D flipflops with common clock and clear.

LS352 Dual 4-line to 1-line mux (as LS153), with inverted outputs.

LS374 Octal three-state positive-edge-triggered D flipflops with

LS381A Arithmetic logic unit with outputs for look-ahead carry.

LS382 Arithmetic logic unit with outputs for ripple carry.

The descriptions of these circuits are based on the details given in *The TTL Data Book* published by Texas Instruments. Equivalent data books are published by all of the major chip manufacturers such as Signetics and National Semiconductor. An effort has been made to make the simulated timings of these circuits match the documented timings of the corresponding chips. In general, the *Data Book* should be used as a source for information about these chips, while the information in the circuit descriptions should be considered secondary and subject to error.

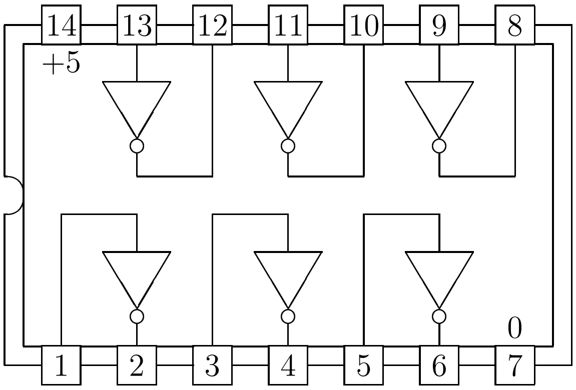
Each circuit description in the TTL collection follows a common scheme. The input and output pins of the chip have numeric names corresponding to the pin numbers used for a dual-in-line package; for example, for a 14 pin chip, these are `p1`, `p2`, `p3`, and so on up to `p14`. For any given chip, 2 of these pins are reserved for power (+5 volts) and ground (0 volts); these pins are not included in the chip descriptions, since they serve electronic purposes below the level of abstraction with which the logic simulator deals. The remaining pins are divided between input and output connections.

For example, consider the SN74LS00 chip; this has 4 nand gates. The inputs of the first nand gate are `p1` and `p2`, and its output is `p3`. The inputs of the second nand gate are `p4` and `p5`, and its output is `p6`. The remaining nand gates are connected to `p8` through `p13`, and pins 7 and 14 are power and ground, and so are not given names in the specification.

The input and output pins of the various chips are documented in the following sections, where the gates are listed in order by function. When a group of chips all have similar input and output connections, a diagram is given for only one of them, with the others listed below it. The diagrams on these pages are abbreviated and somewhat simplified compared with those in the data books; the data books should be used whenever an authoritative chip description is needed and these descriptions should only be used as a guide.

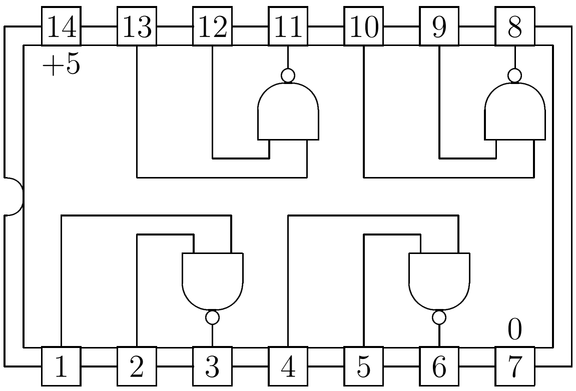
4.1 Primitive Gates

Inverters



LS04: Hex inverters. 9.5ns

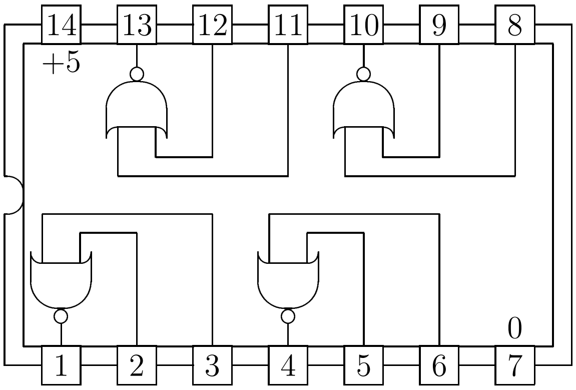
Two Input Gates



LS00: Quad 2-input nand gates. 9.5ns

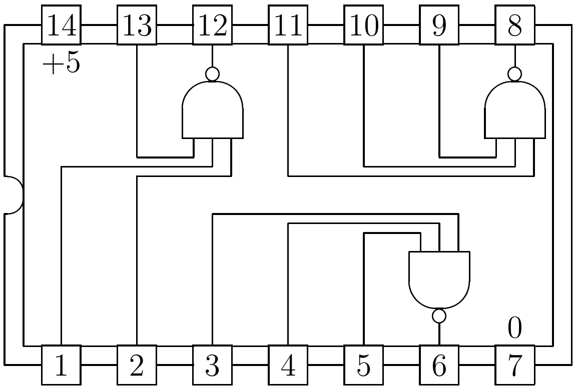
LS08: Quad 2-input and gates. 9.0ns
LS32: Quad 2-input or gates. 14ns
LS86: Quad exclusive or gates. 10ns

Two Input Nor Gates



LS02: Quad 2-input nor gates. 10ns

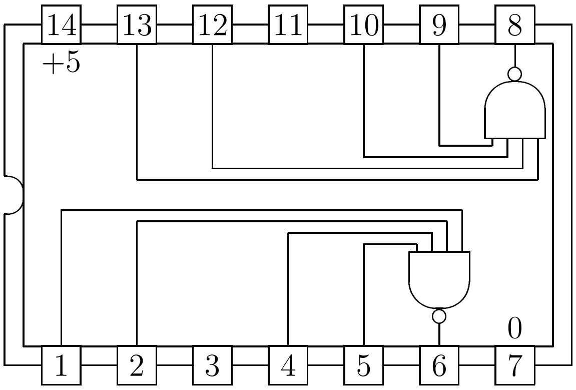
Three Input Gates



LS10: Triple 3-input nand gates. 9.5ns

LS11: Triple 3-input and gates. 9.0ns
LS27: Triple 3-input nor gates. 10ns

Four Input Gates

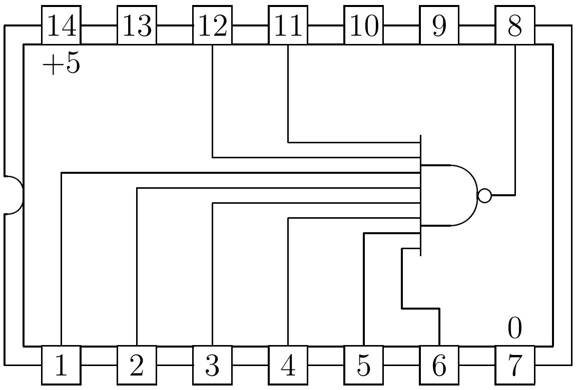


LS20: Dual 4-input nand gates. 9.5ns

LS21: Dual 4-input and gates. 9.0ns

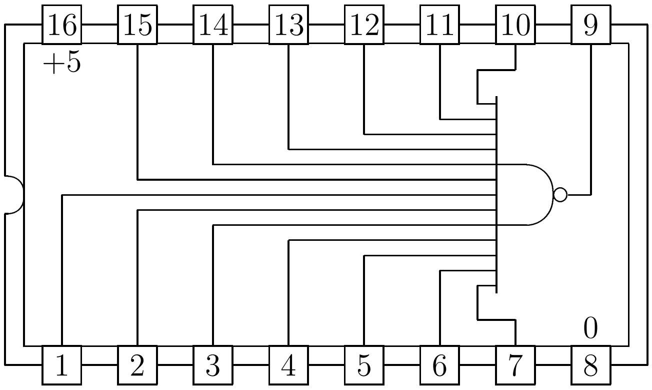
LS40: Dual 4-input nand buffers. 12.0ns

Eight Input Nand Gates



LS30: Single 8-input nand gates. 10.5ns

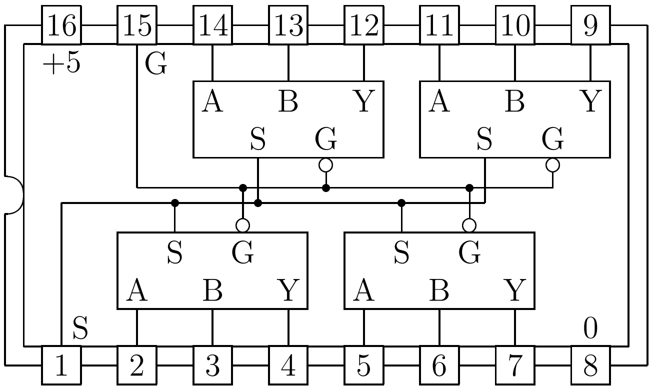
Thirteen Input Nand Gates



S133: Single 13-input nand gates. 6ns

4.2 Multiplexers

Two Input Multiplexers

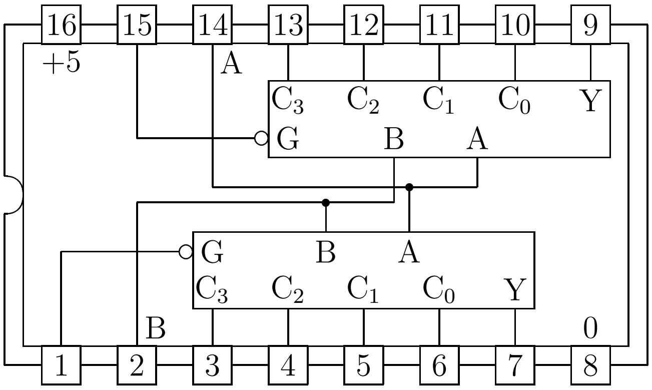


inputs				output Y	
G	S	A	B	LS157	LS158
H	x	x	x	L	H
L	L	L	x	L	H
L	L	H	x	H	L
L	H	x	L	L	H
L	H	x	H	H	L

LS157: Quad 2-line to 1-line multiplexers. 9ns

LS158: Quad 2-line to 1-line inverting multiplexers. 7ns

Four Input Multiplexers



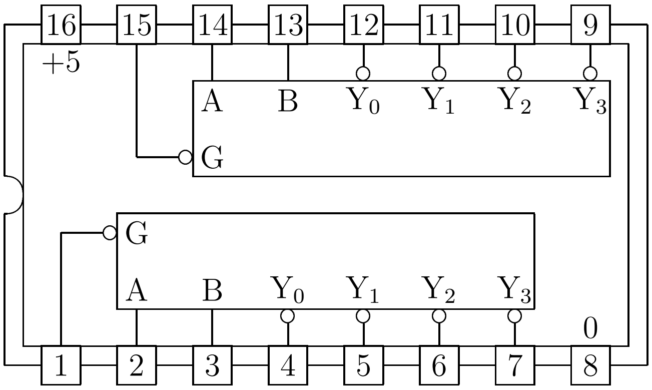
inputs			output Y	
G	B	A	LS153	LS352
H	x	x	L	H
L	L	L	C_0	$\overline{C_0}$
L	L	H	C_1	$\overline{C_1}$
L	H	L	C_2	$\overline{C_2}$
L	H	H	C_3	$\overline{C_3}$

LS153: Dual 4-line to 1-line multiplexers. 22ns

LS352: Dual 4-line to 1-line inverting multiplexers. 22ns

4.3 Decoders

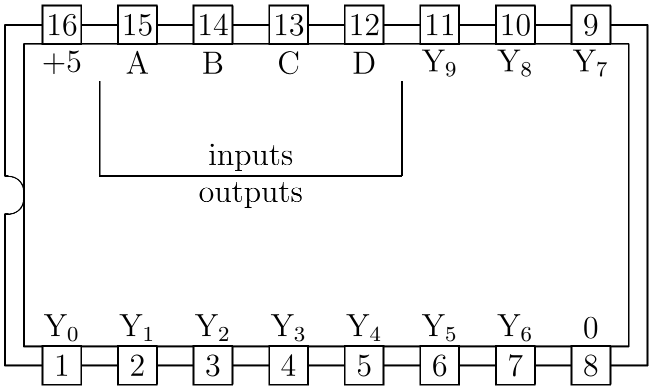
Four Output Decoders



inputs			outputs			
G	B	A	Y_0	Y_1	Y_2	Y_3
H	x	x	H	H	H	H
L	L	L	L	H	H	H
L	L	H	H	L	H	H
L	H	L	H	H	L	H
L	H	H	H	H	H	L

LS139: Dual 2-line to 4-line decoders. 22ns

Ten (or Eight) Output Decoders



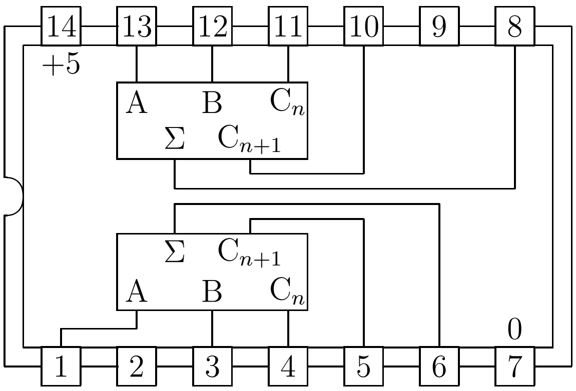
inputs				outputs									
D	C	B	A	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8	Y_9
L	L	L	L	L	H	H	H	H	H	H	H	H	H
L	L	L	H	H	L	H	H	H	H	H	H	H	H
L	L	H	L	H	H	L	H	H	H	H	H	H	H
L	L	H	H	H	H	H	L	H	H	H	H	H	H
L	H	L	L	H	H	H	H	L	H	H	H	H	H
L	H	L	H	H	H	H	H	H	L	H	H	H	H
L	H	H	L	H	H	H	H	H	H	L	H	H	H
L	H	H	H	H	H	H	H	H	H	H	L	H	H
H	L	L	L	H	H	H	H	H	H	H	H	L	H
H	L	L	H	H	H	H	H	H	H	H	H	H	L
H	L	H	x	H	H	H	H	H	H	H	H	H	H
H	H	x	x	H	H	H	H	H	H	H	H	H	H

LS42: BCD-to-decimal or binary-to-octal decoder. 17ns

To use the LS42 as a binary-to-octal decoder, ignore outputs Y_8 and Y_9, apply the binary input to C, B and A, and use input D as a negative-logic enable signal.

4.4 Arithmetic and Comparison

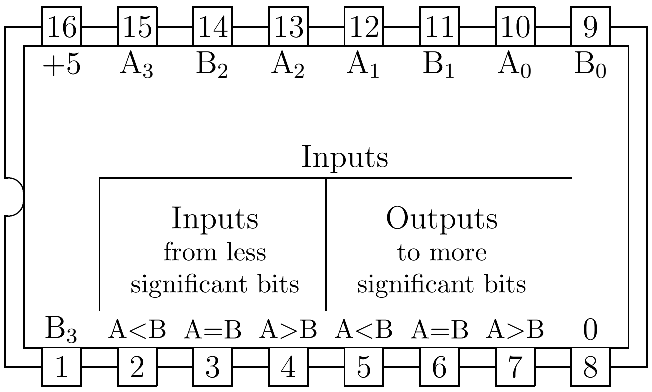
Dual Full Adders



inputs			outputs	
A	B	C_n	Σ	C_{n+1}
L	L	L	L	L
L	L	H	H	L
L	H	L	H	L
L	H	H	L	H
H	L	L	H	L
H	L	H	L	H
H	H	L	L	H
H	H	H	H	H

LS183: Dual full adders. 15ns

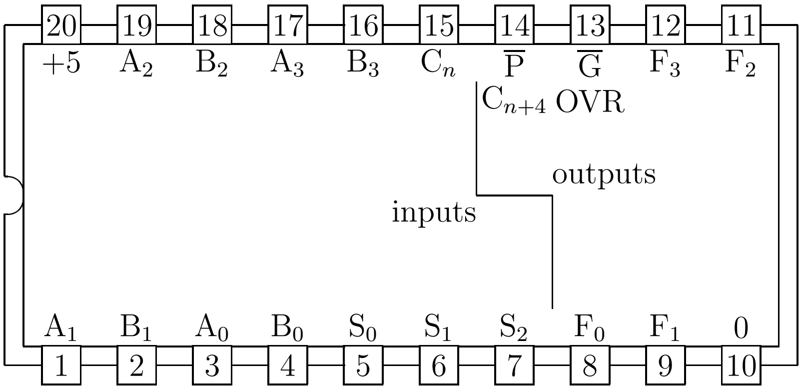
Binary Magnitude Comparators



LS85: 4-bit binary magnitude comparators 24ns

In A and B, the two words to be compared, bit 0 is the least significant bit, and positive logic is used. The <, =, and > inputs to the least significant end of a chain of comparators should be L, H, and L respectively; the outputs from the most significant end are positive logic.

Arithmetic Logic Units



select			output
S_2	S_1	S_0	F
L	L	L	0000
L	L	H	B minus A
L	H	L	A minus B
L	H	H	A plus B
H	L	L	A xor B
H	L	H	A or B
H	H	L	A and B
H	H	H	1111

LS381A: 4-bit ALU slice with look-ahead carry outputs. 17ns

LS382 : 4-bit ALU slice with ripple carry outputs. 17ns

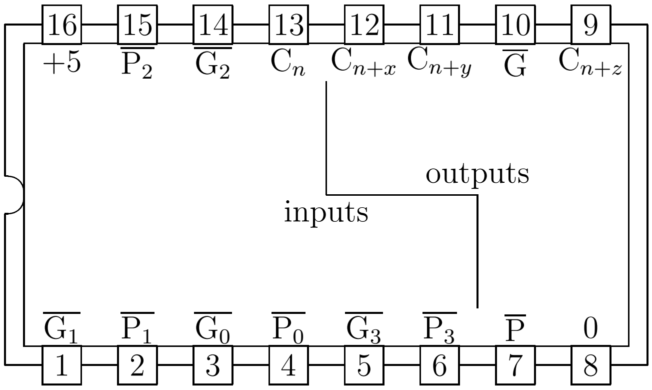
In A and B, the two operands, as well as in F, the result, bit 0 is the least significant bit, and positive logic is used. The delay given is from A and B to F; the function select input S must typically be set up at least 38ns prior to examining the outputs, while the carry input C_n typically takes only 16ns to reach the outputs.

The LS381A produces $\overline{\mathrm{P}}$ and $\overline{\mathrm{G}}$ outputs on pins 14 and 13 which can be used as inputs to S182 chips for look-ahead carry generation. When $\overline{\mathrm{P}}$ is low, the 4-bit slice will propagate a carry, while when $\overline{\mathrm{G}}$ is low, the 4-bit slice will generate a carry.

The LS382 produces C_{n+4} on pin 14, allowing ripple carry between 4-bit slices. The OVR output on pin 13 indicates signed two's complement overflow; effectively, it indicates that $C_{n+3} \neq C_{n+4}$.

Note that when using either ALU to perform subtraction, both the carry in and carry out signals indicate the absence of a borrow. Thus, C_0 (the carry in to the least significant bit) should be 1 during subtraction.

Look-Ahead Carry Generators



S182: Look-ahead carry generators. 6ns

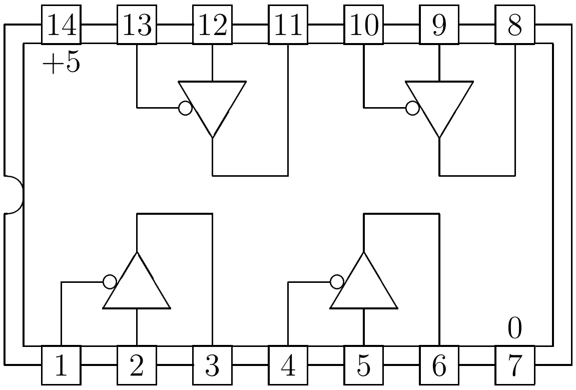
The S182 look-ahead carry generator can provide look-ahead carry propagation for as many as 4 ALUs such as the LS381A, and a tree of 5 S182 chips will serve to provide fast carry across 16 LS381A ALUs, allowing 64 bit arithmetic operations in typically 45ns (from data inputs to data outputs, assuming that the ALU function has been selected in advance).

For both inputs and outputs, $\overline{P}$ will be low if an ALU or group of ALUs will propagate a carry, and $\overline{G}$ will be low if an ALU or group of ALUs will generate a carry. $\overline{P_0}$ through $\overline{P_3}$ and $\overline{G_0}$ through $\overline{G_3}$ are inputs from ALUs or groups of ALUs, with increasing subscripts indicating more significant ALUs. $\overline{P}$ and $\overline{G}$ are outputs from this group of ALUs to higher levels in the tree.

The C_n input (positive logic) indicates carry in to this group of ALUs, and it should be the same as the carry in to the least significant ALU in the group. C_{n+x}, C_{n+y} and C_{n+z} are outputs to the 3 more significant ALU's in the group, in order of increasing significance. These do not depend on $\overline{P_3}$ or $\overline{G_3}$, so the most significant ALU used for a full word operation need not produce look-ahead signals.

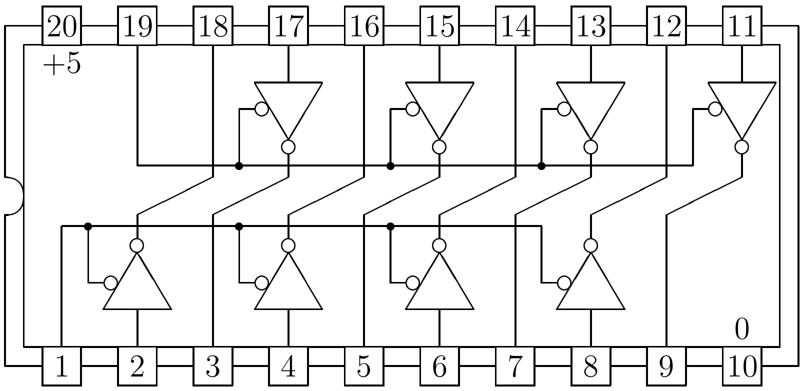
4.5 Three-state Bus Drivers

Quadruple Bus Drivers



LS125A: Quadruple non-inverting three-state buffers. 9ns

Octuple Bus Drivers

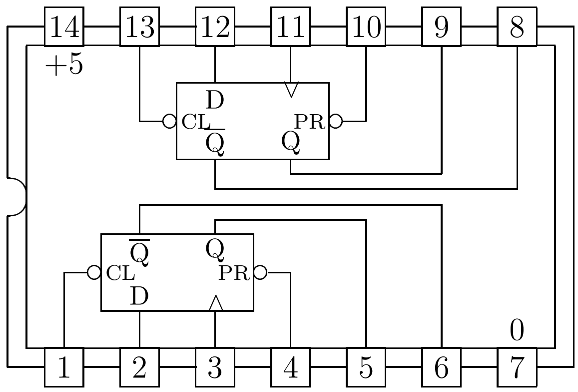


LS240: Octal inverting three-state buffers. 10.5ns

LS244: Octal non-inverting three-state buffers. 12ns

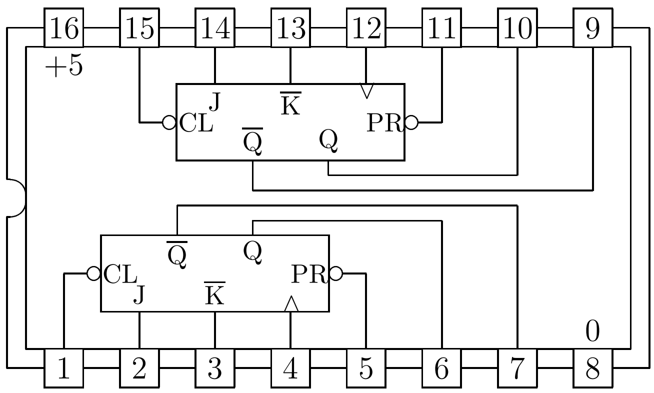
4.6 Flipflops

Dual D Flipflops



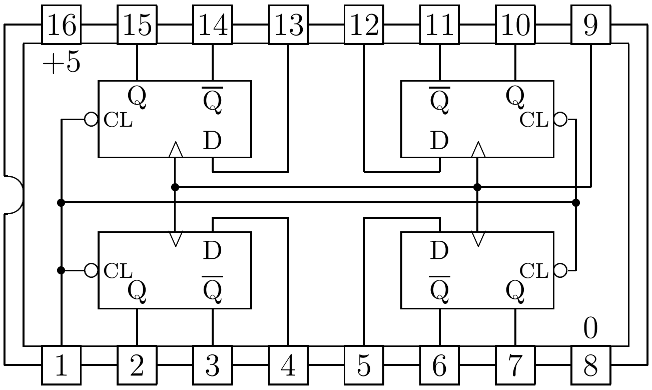
LS74A: Dual D positive-edge-triggered flipflops. 25ns

Dual J-K Flipflops



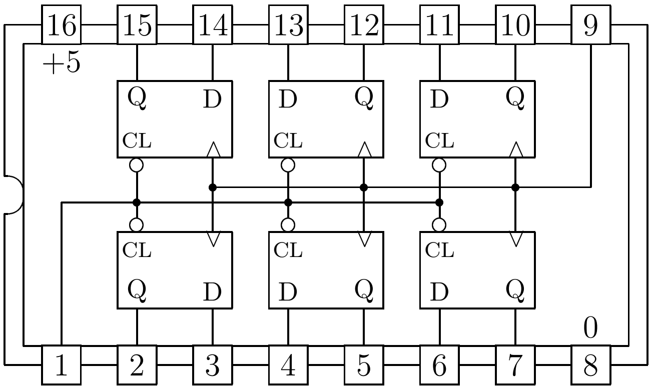
LS109A: Dual J-$\overline{K}$ positive-edge-triggered flipflops. 24ns

Quadruple D Flipflops



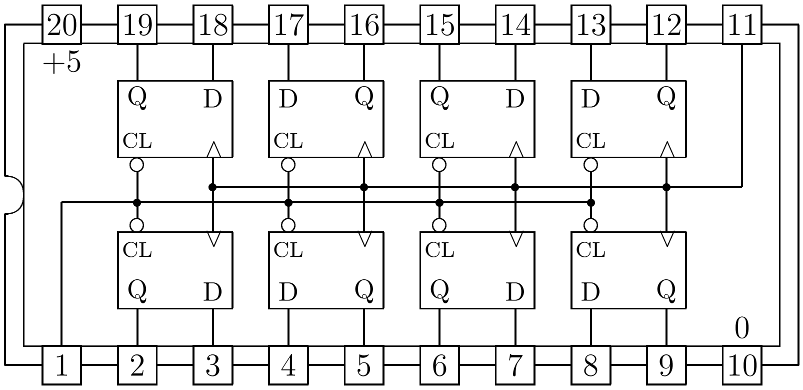
LS175: Quad D positive-edge-triggered flipflops. 20ns

Hextuple D Flipflops



LS174: Hex D positive-edge-triggered flipflops. 20ns

Octuple D Flipflops



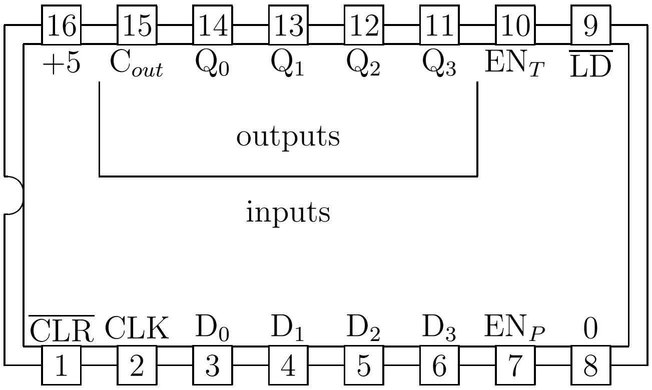
LS273: Octal D flipflops with clear. 18ns

LS374: Octal D flipflops with three-state outputs. 20ns

The clear input CL to each flipflop on the LS273 is replaced on the LS374 by an output-enable input called OE. Both CL and OE use negative logic, and all of these flipflops are positive edge triggered.

4.7 Counters

Synchronous 4-bit Binary Counters



LS161A: Synchronous 4-bit binary counter. 14ns

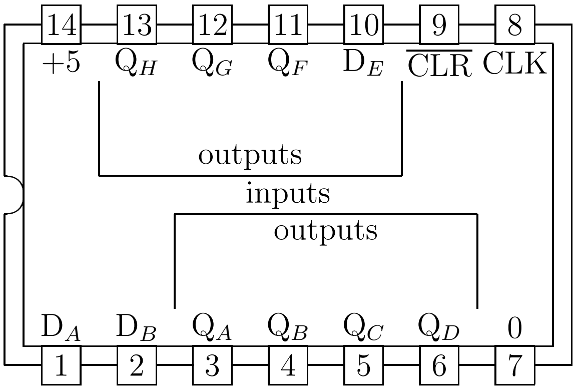
The LS161A binary counter has a positive edge triggered clock input CLK, and a negative logic asynchronous clear input $\overline{\mathrm{CLR}}$. The value stored in the counter is continuously presented on the Q outputs (Q_0 is least significant, positive logic is used).

When $\overline{\mathrm{LD}}$ is held low, data from the D inputs will be loaded into the counter when it is clocked. When EN_T, EN_P and $\overline{\mathrm{LD}}$ are all held high, the counter will increment when clocked. Holding either EN_T or EN_P inputs low will inhibit counting. The C_{out} output indicates that the counter holds the value 1111 and the EN_T input is high.

To build high precision synchronous counters, wire the C_{out} output of each stage to the EN_T input of the next more significant stage, and wire the CLK, $\overline{\mathrm{CLR}}$, $\overline{\mathrm{LD}}$ and EN_P inputs in parallel across all stages.

4.8 Shift Registers

8-bit serial to parallel shift registers



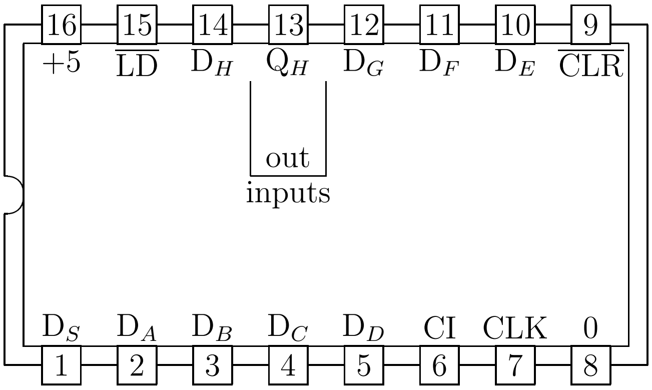
LS164: 8-bit serial to parallel shift register. 28ns

The LS164 shift register has a positive edge triggered clock CLK and a negative logic asynchronous clear $\overline{\text{CLR}}$. All 8 bits stored in the register are continuously displayed on the data outputs Q_A to Q_H, where Q_A was the most recent value shifted into the register.

The serial data inputs D_A and D_B are anded together; both must be high to shift a high into the register; either or both may be low to shift a low into the register.

To make a longer shift register, wire the Q_H output of one stage to the serial data inputs of the next stage, and wire all clock and clear inputs in parallel across all stages.

8-bit Parallel to Serial Shift Registers



LS166: 8-bit parallel to serial shift register. 28ns

The LS166 shift register has a positive edge triggered clock CLK and a positive logic clock-inhibit input CI. CI should not change from low to high except when CLK is already high. $\overline{\text{CLR}}$ is a negative logic asynchronous clear. The value stored in the 8th bit of the register is continuously presented on the output Q_H.

When $\overline{\text{LD}}$ is held low, data from the inputs D_A to D_H will be loaded into the register when it is clocked. When $\overline{\text{LD}}$ is held high, the register will shift when it is clocked, with new data entering stage A of the register from the serial data input D_S.

Longer shift registers can be made by chaining the Q_H output of one stage to the D_S input of the next, and connecting the CLK, CI, $\overline{\text{CLR}}$, and $\overline{\text{LD}}$ inputs in parallel across all stages.