High-level circuit design

Eric C. R. Hehner[1], Theodore S. Norvell[2], Richard F. Paige[3]

[1] Department of Computer Science, University of Toronto ON M5S 3G4 Canada (hehner@cs.utoronto.ca)

[2] Faculty of Engineering, Memorial University of Newfoundland, St.John's, NF A1B 3X5 Canada (theo@engr.mun.ca)

[3] Department of Computer Science, York University, Toronto ON M3J 1P3 Canada (paige@cs.yorku.ca)

Received: 7 November 1995 / 15 September 1997 / 21 November 1998

Dedication This paper is dedicated to the memory of Jan van de Snepscheut, 1953-1994.

Abstract We present two new ways to implement ordinary programs with logic gates. One, like imperative programs, has an associated memory to store state; the other, like functional programs, passes the state from one component to the next. Circuit design can be done more effectively by using a standard programming language to describe the function that a circuit is intended to perform, rather than by describing a circuit that is intended to perform that function. The resulting circuits are produced automatically; they behave according to the programs, and have the same structure as the programs. For timing we use local delays, rather than a global clock or local handshaking. We give a formal semantics for both programs and circuits in order to prove our circuits correct. By simulation, we also demonstrate that the circuits perform favorably compared to others.

1 Introduction

The design methods for digital circuits that are commonly found in current textbooks resemble the low-level machine-language programming methods of forty years ago. Manually selecting individual logic gates in a circuit is something like selecting individual machine instructions in a program. These methods may have been adequate for small circuit design when they were introduced, and they may still be adequate for large circuits that are simply repetitions of a small circuit (such as a memory), but they are not adequate for circuits that perform complicated custom algorithms.

In general, we do not build circuits to perform complicated custom algorithms. We build general-purpose processors, and customize them for a particular algorithm by writing a program. That we can do so was a fundamental insight, due to Turing, upon which computer science is based. But for some applications, particularly where speed of execution or security is important, a custom-built circuit has some advantages over the usual processor-and-software combination. The speed is improved by the absence of the "machine-language" layer of circuitry with its "fetch-execute" cycle of interpretation, and

by the ease with which we can introduce parallelism. Security is improved by the impossibility of reprogramming. In addition, unless the application requires a lengthy algorithm, there are area savings compared to a combination of software and processor.

The VHDL [8] and Verilog [14] languages are presently being used by industry. These languages allow circuit designers to describe circuits more conveniently. There are interactive synthesis tools to aid in the construction of circuits from subsets of these languages. The circuits are then "verified" by simulation.

We do not present a new language for circuit design. Instead, we advocate using a standard programming language (for example, C), not to describe circuits, but to describe algorithms. The resulting circuits are produced automatically; they behave according to the programs, and have the same structure as the programs. For timing we use local delays, rather than a global clock (synchronous) or local handshaking (asynchronous). We give a formal semantics for both programs and circuits in order to prove our circuits correct, using a theory presented in [5]. By simulation, we also demonstrate that the circuits perform favorably compared to others.

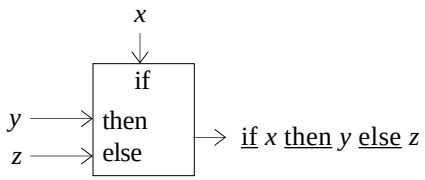
There are other high-level circuit design techniques being developed and reported in the literature. Early work includes [12], [13], and [4]. In [3,7], a circuit is specified in a subset of CSP as a set of communicating processes, and is transformed into circuits via an intermediate mapping to production rules. A similar approach (and a similar circuit design language) is used in [1,2], except that specifications are mapped into connections of small components for which standard transistor implementations exist. In [15] circuits are modeled as networks of finite state machines, and their formalism is used to assist in proving the correctness of their compiled circuits. The work of [6,10] is most similar to ours, but their designs have a global clock; ours do not.

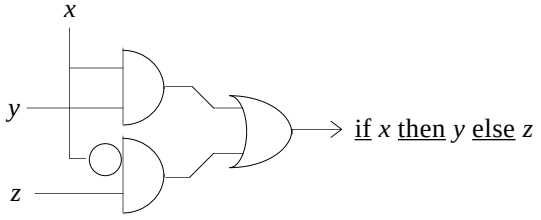
The success of high-level circuit design will probably be judged the same way high-level programming was judged: on whether the circuits produced are competitive with low-level designs, and on whether we are able to design complex circuits more easily and more reliably. The outcome will probably be the same for circuits as for programming.

This paper is intended to be self-contained, showing how circuits are built from the gate level up. Consequently we must ask for the patience of knowledgeable readers whenever we cover familiar ground. Sometimes we cover familiar ground in an unfamiliar way.

2 Diagrams

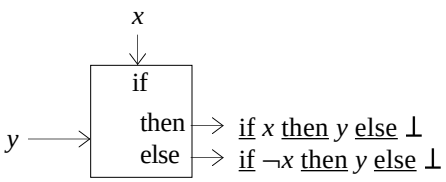
Circuits are often expressed as diagrams constructed from "and", "or", and "not" gates. We can give a diagram to any other operator by the expedient method of placing its symbol in a box. Here is the if then else (also called multiplexer). On the right we show an implementation using negation, conjunction, and disjunction.

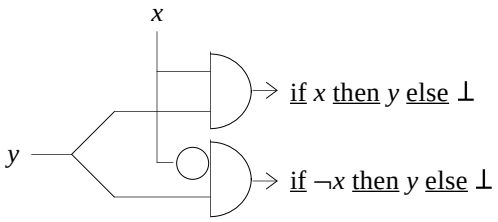




By this pair of diagrams, we do not intend to suggest that if then else is best implemented as shown, but only to show one way it can be implemented, and to show how we obtained our performance estimates by counting gate delays.

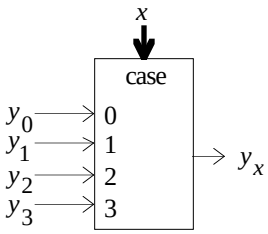
It is also convenient to define the switch (or demultiplexer), and to give it a diagram.

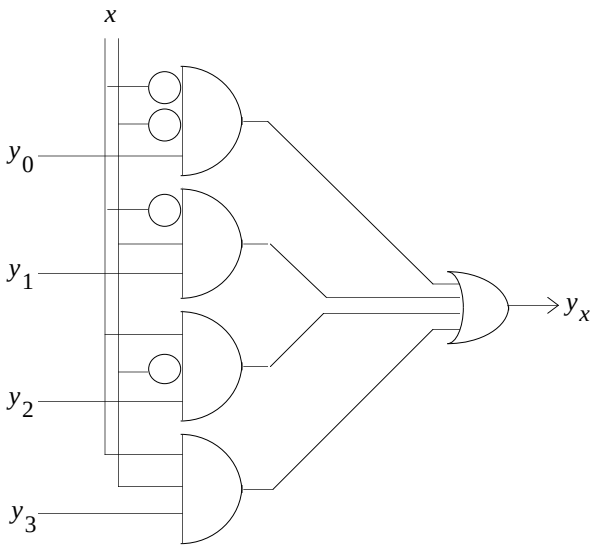


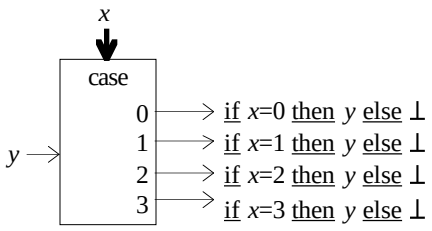


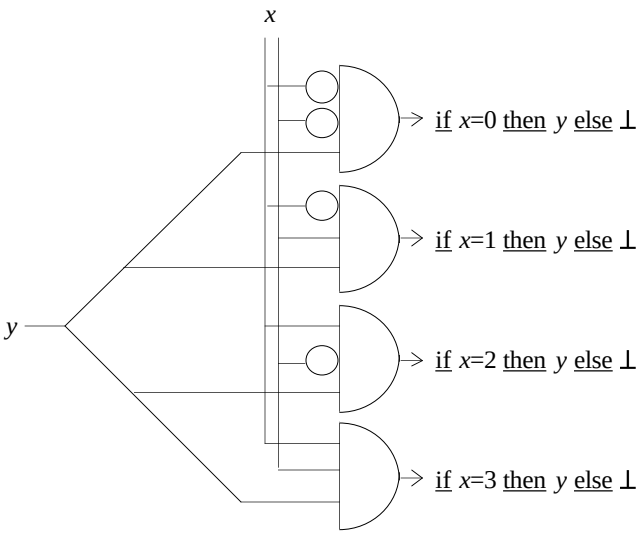
The symbol ⊥ is for "low voltage" or "ground" or "false". We use ⊤ for "high voltage" or "power" or "true".

The if then else and switch also come in a more general form in which the selection is made by an integer rather than a boolean. For a 2-bit integer selection the diagrams are









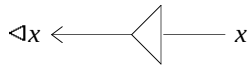
A thin line indicates a single wire, and a thick line indicates any number of wires. Crossing wires are not connected.

Circuit diagrams are helpful when planning the layout of a circuit, but for designing the logic we prefer to use algebraic notation for its ease of manipulation. We give both diagrams and algebraic descriptions, but we do not intend the diagrams to indicate layout.

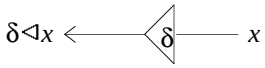
3 Time

Ideally, we might suppose that circuit components act instantly, with no gate delays, and are represented accurately by timeless boolean expressions. Realistically, there are gate delays, and sometimes there are transient signals (glitches) while a circuit settles into a stable state. We must introduce a timing discipline to ensure that we do not require, and are not affected by, a result before it is ready. We can consider time to be continuous or discrete; nothing in this paper will depend on that choice.

To talk about time, we find it convenient to introduce the operator $\lhd$, pronounced "delay" or "previous". It gives the value that its operand had previously, a short time ago. Its diagram looks like this:



Whenever we need to say formally what constraints a delay time must satisfy, we write it to the left of the delay operator, and inside its circuit graphic:



Delay time is dependent on context and technology, it is usually determined by experiment, and can be known only approximately, say with an upper and lower bound. Sometimes we want the delay to be as short as possible; when that is the case, signal propagation time through the wire and surrounding gates is sufficient, and no extra circuitry is required. When more delay is needed, it can be implemented as an even number of negations, or by a suitable choice of layout; these implementations are not subject to glitches, and so do not raise again the problem they are solving. In addition to its logical use, the delay sometimes has the electrical job of reshaping a pulse, both height and width, to compensate for degradation. But that is a level of detail below our concern in this paper.

As a formal requirement, for proof of correctness, we need to define the output of a delay to be initially $\bot$ for the delay time, and thereafter it is the same as the input but delayed. This initial $\bot$ is the only initialization in our circuits; we don't consider initialization circuitry in this paper.

4 Flip-flops

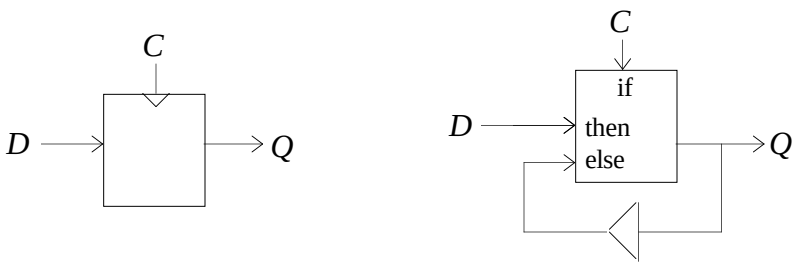
A flip-flop has inputs C (clock) and D (data), and output Q (the letter Q is like the letter O for output, but distinguishable from the digit 0). A flip-flop behaves as follows: If the clock is $\top$, then the output is the data input, otherwise the output remains as it was. This behavior can be formalized directly as follows.

$\underline{\text{if}}\ C{=}\top\ \underline{\text{then}}\ Q{=}D\ \underline{\text{else}}\ Q{=}\lhd Q$

We can simplify this boolean expression in two ways. Equating to $\top$ is always superfluous, just as adding zero or mutiplying by one are superfluous. And we can factor out the "$Q=$", obtaining

$Q\ =\ \underline{\text{if}}\ C\ \underline{\text{then}}\ D\ \underline{\text{else}}\ \lhd Q$

which we might well have written in the first place. We now have a definitional equation, which is the form suitable for automatic circuit fabrication. An equation is "definitional" if one side is the output. Here are the black box and implementation diagrams.



For the circuit to operate correctly, the delay must be small enough that no change to D occurs during the delay preceding the fall of the clock. For the circuit to be as fast as possible, the delay must be as small as possible. There will be a constraint on the minimum delay for electrical reasons, which we do not consider here. With no delay, this circuit is logically equivalent to a standard textbook flip-flop.

We generalize the flip-flop to allow any number of pairs of clock and data inputs. If exactly one of the clock lines is $\top$, then the output is the corresponding data input; if none of the clocks are $\top$, then the output remains as it was. Let the clock lines be C_0, C_1, ... and let the corresponding data lines be D_0, D_1, ... Then the formal specification is

$(\exists_1 i\cdot C_i) \Rightarrow Q=(\exists i\cdot C_i \wedge D_i)$

$(\neg\exists i\cdot C_i) \Rightarrow Q=\triangleleft Q$

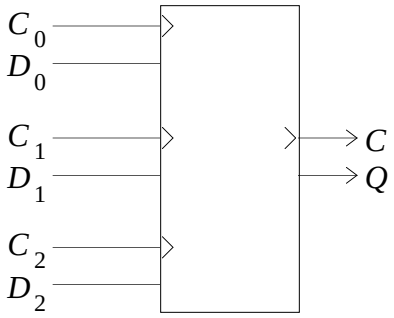
Actually, the formal specification is the conjunction of the two formulas. As is customary in mathematics, we sometimes write a list of boolean expressions when we mean their conjunction. The specification is nondeterministic, or in circuit terminology, it has “don't cares”, when more than one clock input is $\top$ (because we don't intend to make more than one clock input $\top$ at the same time). We can strengthen the specification if we wish (resolving nondeterminism, deciding “don't cares”) because all behavior satisfying a stronger specification will also satisfy the original specification. The circuit designer's job is to find an equivalent or stronger specification in the form of a definitional equation. Here's one:

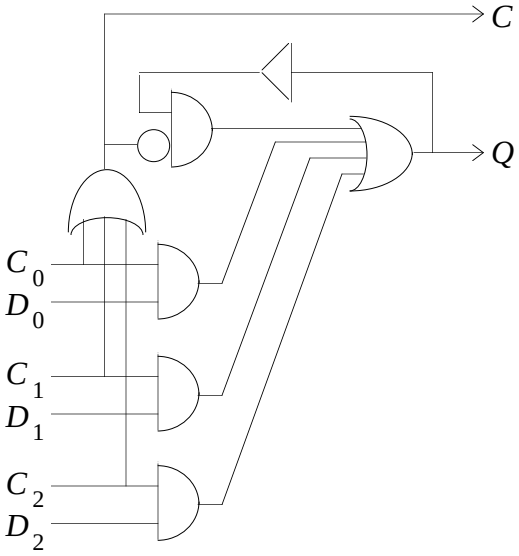
$Q = ((\exists i\cdot C_i \wedge D_i) \vee (\neg\exists i\cdot C_i) \wedge \triangleleft Q)$

We will find it convenient later to provide a clock output C that is $\top$ when any of the clock inputs are $\top$.

$C = \exists i\cdot C_i$

Here are the more general flip-flop diagrams.





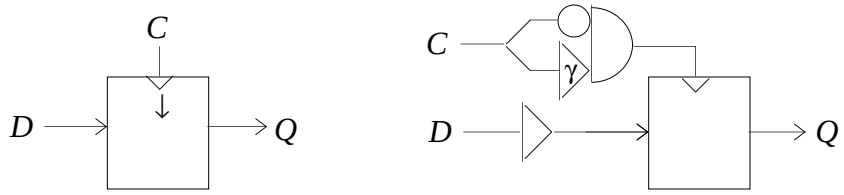
5 Edge-Triggering

The flip-flops we have just described remain sensitive to the data input as long as the clock is $\top$. Sometimes we want a flip-flop that is sensitive to its data input only at the rising or falling edge of the clock input. For example, a falling-edge-triggered flip-flop can be defined as

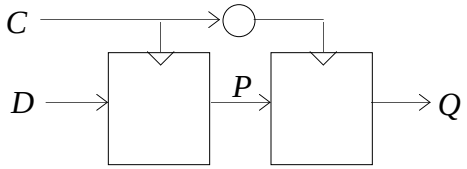
Q = if $\neg C \wedge \gamma\lhd C$ then $\lhd D$ else $\lhd Q$

$\gamma \geq$ (edge time) + (negation delay)

The expression $\neg C \wedge \lhd C$ says that the clock is down but was just previously up, so it is a falling edge. The C-delay γ should be just large enough to allow C to fall and to allow that falling edge to be negated. The D-delay determines what data is latched; for example, we might want the data from before the falling edge, or at its start, or at its end (this delay could be omitted). As always, the Q-delay should be as small as possible. The diagrams (note the down-arrow in the black-box diagram to indicate falling-edge-triggering):



Aside The standard way to obtain an edge-triggered flip-flop is with a master-slave pair of flip-flops. For example, we obtain a falling-edge-triggered flip-flop as follows (since we use a triangle for delay, we use just a circle for negation):



Algebraically, this is

P = if C then D else $\lhd P$

Q = if $\neg C$ then P else $\lhd Q$

We now prove that this master-slave pair is equivalent to our single flip-flop. Apply a delay to the P equation and to the Q equation to obtain

$\lhd P$ = if $\lhd C$ then $\lhd D$ else $\lhd\lhd P$

$\lhd Q$ = if $\neg\lhd C$ then $\lhd P$ else $\lhd\lhd Q$

From the P, Q, and $\lhd P$ equations we find

$\neg C \wedge \lhd C \;\Rightarrow\; P{=}\lhd P \wedge Q{=}P \wedge \lhd P{=}\lhd D$

and so, by transitivity,

(*) $\neg C \wedge \lhd C \;\Rightarrow\; Q{=}\lhd D$

From the P, Q, and $\lhd Q$ equations we find

$\neg C \wedge \neg\lhd C \;\Rightarrow\; P{=}\lhd P \wedge Q{=}P \wedge \lhd Q{=}\lhd P$

and so, by transitivity,

$\neg C \wedge \neg\lhd C \;\Rightarrow\; Q{=}\lhd Q$

Also, from the Q equation alone,

$C \;\Rightarrow\; Q{=}\lhd Q$

Putting these last two implications together we find

$\neg C \wedge \neg\lhd C \vee C \;\Rightarrow\; Q{=}\lhd Q$

The antecedent can be rewritten

(**) $\neg(\neg C \wedge \lhd C) \;\Rightarrow\; Q{=}\lhd Q$

Putting (*) and (**) together we have

$\underline{\text{if}}\ \neg C \wedge \triangleleft C\ \underline{\text{then}}\ Q{=}\triangleleft D\ \underline{\text{else}}\ Q{=}\triangleleft Q$

which can be rewritten

$Q\ =\ \underline{\text{if}}\ \neg C \wedge \triangleleft C\ \underline{\text{then}}\ \triangleleft D\ \underline{\text{else}}\ \triangleleft Q$

So the master-slave combination is logically equivalent, but more complicated. *End of Aside*

Edge-triggering is applicable to more than just flip-flops. For example, to create a switch (demultiplexer) that triggers its output on the rising edge of the y input, and then holds its output for the duration of $\top$ on y (ignoring fluctuations on x),

$t\ =\ \underline{\text{if}}\ x \wedge y \wedge \neg\triangleleft y\ \underline{\text{then}}\ \top\ \underline{\text{else}}\ \underline{\text{if}}\ y\ \underline{\text{then}}\ \triangleleft t\ \underline{\text{else}}\ \bot$

$e\ =\ \underline{\text{if}}\ \neg x \wedge y \wedge \neg\triangleleft y\ \underline{\text{then}}\ \top\ \underline{\text{else}}\ \underline{\text{if}}\ y\ \underline{\text{then}}\ \triangleleft e\ \underline{\text{else}}\ \bot$

These equations can be simplified, and delay times can be added, to get

$t\ =\ ((x \wedge \neg\psi\triangleleft y\ \vee\ \tau\triangleleft t)\ \wedge\ y)$

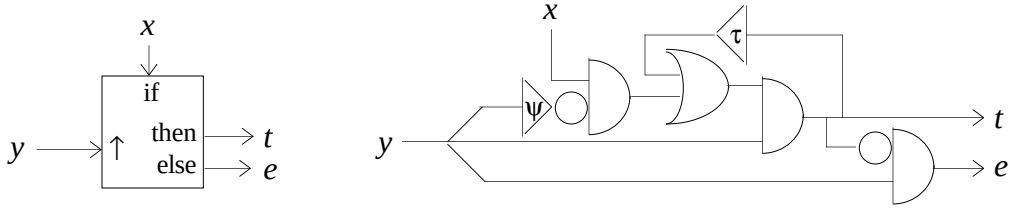
$e\ =\ ((\neg(x \vee \psi\triangleleft y)\ \vee\ \tau\triangleleft e)\ \wedge\ y)$

$\psi >$ (edge time) $\wedge\ \tau < \psi$

By induction over significant instants of time (induction hypothesis: it's true up to some time; induction step: it's still true after the next time that something changes), we can prove

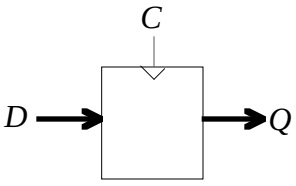
$t\ =\ (\neg e \wedge y)$ $\qquad\qquad$ $e\ =\ (\neg t \wedge y)$

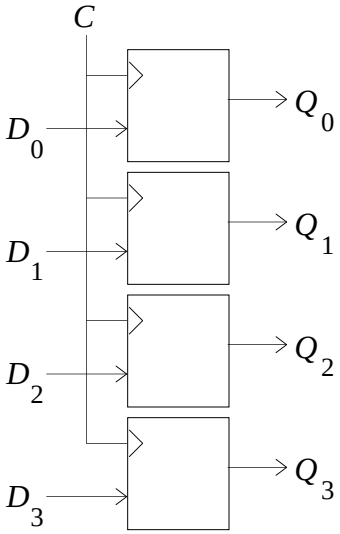
and further simplify our circuit. The diagrams (note the up-arrow in the black-box diagram to indicate rising-edge-triggering):



6 Memory

We can aggregate flip-flops into a larger amount of memory called a “word”, suitable for storing an **int** or **real** value. We use the same diagram as for a flip-flop but with a thick D input and Q output to indicate many data lines.





All bits in the word are written at the same time, and read at the same time. Algebraically we describe the word by

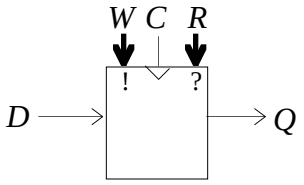
$\forall i \cdot \; Q_i \;=$ <u>if</u> C <u>then</u> D_i <u>else</u> $\lhd Q_i$

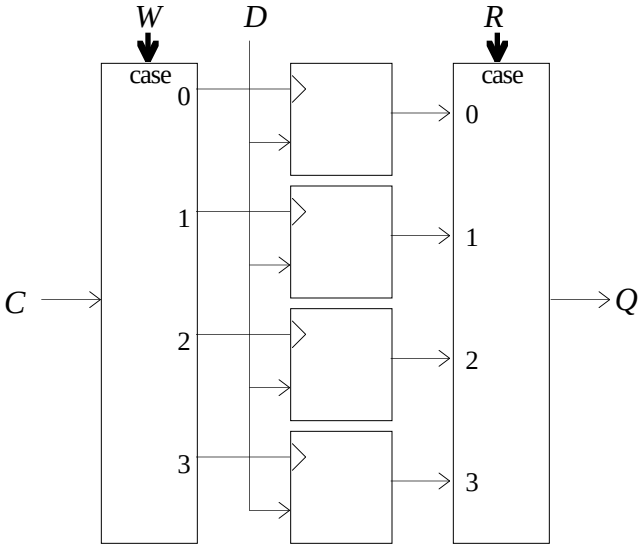
More conveniently, we write

$Q \;=$ <u>if</u> C <u>then</u> D <u>else</u> $\lhd Q$

as before, even when Q and D are several bits each.

A RAM (random access memory) is an independent way to aggregate flip-flops into a larger amount of memory. One bit is written at a time, and one bit is read at a time. A bit to be written is selected by the writing address W, and a bit to be read is selected by the reading address R.





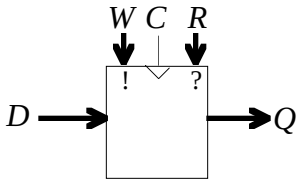
Algebraically,

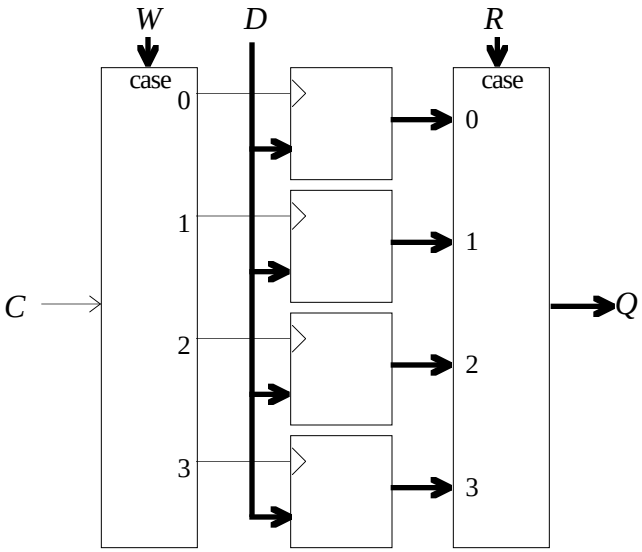
$Q \;=\; M[R]$

$\forall i \cdot \; M[i] \;=$ <u>if</u> $C \wedge i{=}W$ <u>then</u> D <u>else</u> $\lhd M[i]$

where $M[i]$ is bit i in the RAM.

The two ways of aggregating bits can be combined to provide a RAM of words. Here are the diagrams.





Adding an arrow in the diagram indicates that the flip-flops are edge-triggered.

7 Merge

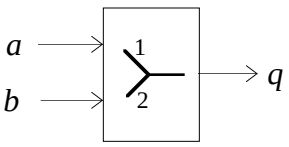
A merge turns two sequences of pulses into a single sequence of pulses. (A pulse is a momentary $\top$). In a sense, any circuit with two inputs and one output is a kind of merge. An or-gate allows pulses on either input to pass through; an and-gate allows only simultaneous pulses to pass through.

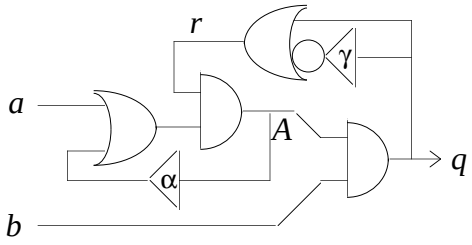
The 1-2-merge has inputs a and b and output q. It outputs a pulse when pulses arrive on a and b in that order, or simultaneously, but not in the other order. To design a 1-2-merge, we introduce an internal wire A with the meaning "a is $\top$ or has been $\top$".

$A = (a \vee \alpha \triangleleft A)$
$q = (A \wedge b)$
$\alpha \le$ (pulse time)

(Recall that $\triangleleft A$ is initially $\bot$.) Unfortunately this is a one-time-only circuit; if ever there is a pulse on a, it will allow all subsequent pulses on b to pass. To obtain a circuit that resets itself on the falling edge of q ready to be used repeatedly, we introduce one more internal wire r that is $\top$ except at the falling edge of q. The circuit becomes

$r = (q \vee \neg\gamma \triangleleft q)$
$A = (r \wedge (a \vee \alpha \triangleleft A))$
$q = (A \wedge b)$
$\alpha \le$ (pulse time) $\wedge$ $\alpha \le \gamma$





Internal wires can be left exposed, as in the above specification of 1-2-merge and the right-hand diagram, or they can be hidden as in the left-hand diagram and the following specification:

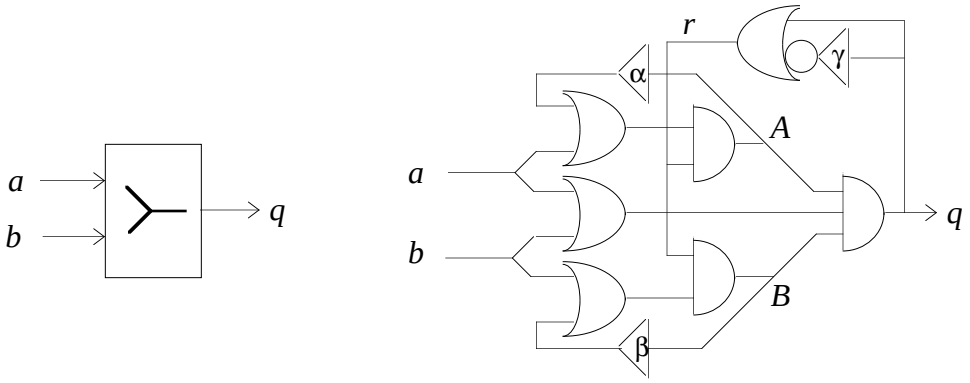
$\exists r, A \cdot \quad r = (q \vee \neg\gamma \triangleleft q)$
$\wedge \quad A = (r \wedge (a \vee \alpha \triangleleft A))$
$\wedge \quad q = (A \wedge b)$

If a pulse on a follows a pulse on b, there must be a delay of at least γ after the end of b before the start of a to avoid truncating the output pulse. No circuit can constrain its inputs; its context of use must constrain its inputs, so a constraint is expressed formally as an antecedent rather than a conjunct. The circuit specification is therefore

$\neg(a \wedge \neg\gamma \triangleleft a \wedge b) \Rightarrow \exists r, A \cdot \; r = (q \vee \neg\gamma \triangleleft q) \wedge A = (r \wedge (a \vee \alpha \triangleleft A)) \wedge q = (A \wedge b)$

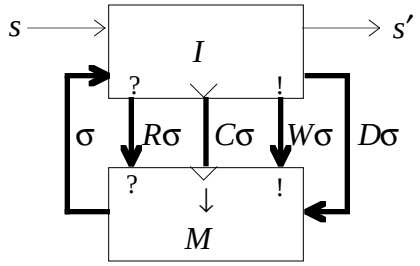
A merge that outputs a pulse when the second of the two input pulses arrives, regardless of their order, and resets itself for reuse, is as follows. The inputs are a and b and the output is q. Internal wire A means "a is $\top$ or has been $\top$"; internal wire B means "b is $\top$ or has been $\top$"; internal wire r is $\top$ except at the falling edge of q. The circuit is

$r = (q \vee \neg\gamma \triangleleft q)$
$A = (r \wedge (a \vee \alpha \triangleleft A))$
$B = (r \wedge (b \vee \beta \triangleleft B))$
$q = (A \wedge B \wedge (a \vee b))$
$\alpha \le$ (pulse time) $\wedge$ $\alpha \le \gamma$ $\wedge$ $\beta \le$ (pulse time) $\wedge$ $\beta \le \gamma$



8 Imperative Circuits

The first of our two translations from programs to circuits produces "imperative" circuits (as in "imperative programming"). An imperative circuit has two components, an imperative control I, and a memory M, connected like this.



The memory consists of a word for each global variable and a RAM for each global array in the program. (We present local variables later. By making variables as local as possible, we minimize the need for the global memory.) Suppose the variables are x and y, and the arrays are A and B. Then there are four clock wires, called Cx, Cy, CA, and CB, and collectively called $C\sigma$. With one clock wire for each variable and each array, the variables and arrays can be independently and asynchronously changed. The data inputs are Dx, Dy, DA, and DB, collectively called $D\sigma$. The address wires are WA, WB, RA, and RB, collectively called $W\sigma$ and $R\sigma$. The memory outputs are x, y, $A[RA]$ and $B[RB]$, collectively called σ, the state of memory. Altogether, memory is

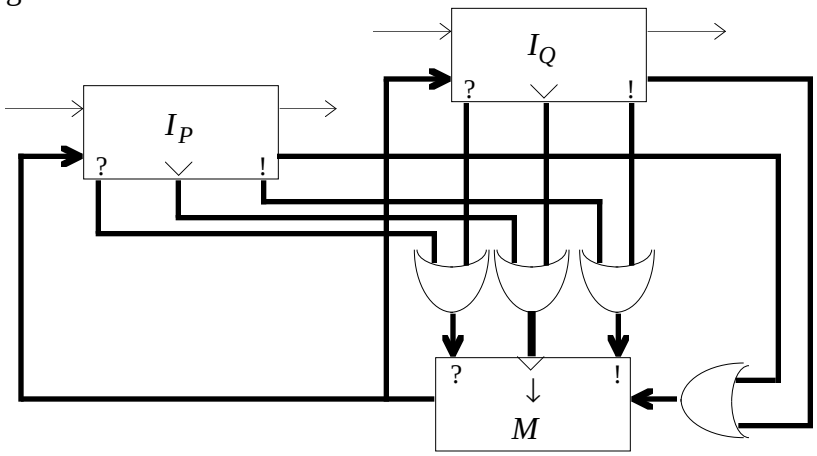
$$\begin{aligned} M \;=\; (\;& x = (\underline{\text{if}}\ \neg Cx \wedge \lhd Cx\ \underline{\text{then}}\ \lhd Dx\ \underline{\text{else}}\ \lhd x) \\ \wedge\;& y = (\underline{\text{if}}\ \neg Cy \wedge \lhd Cy\ \underline{\text{then}}\ \lhd Dy\ \underline{\text{else}}\ \lhd y) \\ \wedge\;& (\forall i\cdot A[i] = \underline{\text{if}}\ \neg CA \wedge \lhd CA \wedge i{=}WA\ \underline{\text{then}}\ \lhd DA\ \underline{\text{else}}\ \lhd A[i]) \\ \wedge\;& (\forall i\cdot B[i] = \underline{\text{if}}\ \neg CB \wedge \lhd CB \wedge i{=}WB\ \underline{\text{then}}\ \lhd DB\ \underline{\text{else}}\ \lhd B[i])\ ) \end{aligned}$$

We mention again that we are depicting logic, not layout; the best place for a bit of memory may be with a part of the control that uses it.

The state is input to the control, along with an initiator wire s. A pulse on s starts the computation. As the computation progresses, the control changes the state of memory, thus providing itself with further input. To change the value of variable x in memory, the control must send a pulse on clock wire Cx and the desired new value on wire Dx. If the computation is finite, then when it is complete, the control indicates termination by a pulse on the completion wire s'. It is the responsibility of the context to ensure that the control is not restarted before it has completed an execution.

A program is sometimes composed of smaller programs. (In other terminology, a statement is sometimes composed of smaller statements; we do not distinguish between

"program" and "statement".) When a program is composed of parts, the control will be composed of the controls for the parts. To make the composition easy, we require of each part that its output Dx be $\perp$ at any instant when it is not changing variable x. Then we can disjoin the Dx wires on their way to memory. Other variables and arrays are similar. Here's the diagram.



Each disjunction is really many disjunctions, one for each bit in its operands.

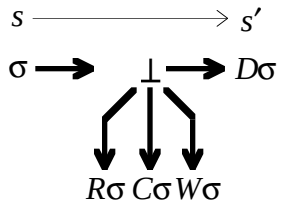
It is not our intention to present a new programming language for circuit design; we advocate using a standard programming language. We now describe the control for a sampling of programming constructs from typical imperative languages.

Construct: empty

We begin with the simplest program: **ok** (sometimes called **skip**). It is the "empty" program, whose execution does nothing, taking no time. Program **ok** yields the control

$s' = s \wedge \neg R\sigma \wedge \neg C\sigma \wedge \neg W\sigma \wedge \neg D\sigma$

Its diagram is



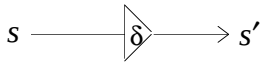
We have shown all its inputs and outputs. But since the σ input is not connected to anything, there is no point in bringing those wires from memory. And since the $R\sigma$, $C\sigma$, $W\sigma$, and $D\sigma$ outputs are $\perp$, there is no point in taking them into a disjunction. So the circuit reduces to nothing, which is appropriate for a circuit that does nothing.

Construct: delay

The next simplest program is **tick**, which also does nothing, but takes time δ to do it.

$s' = \delta \lhd s \wedge \neg R\sigma \wedge \neg C\sigma \wedge \neg W\sigma \wedge \neg D\sigma$

Constraints on δ must be stated with each use of **tick**. Leaving out the nonexistent wires, we have this picture:



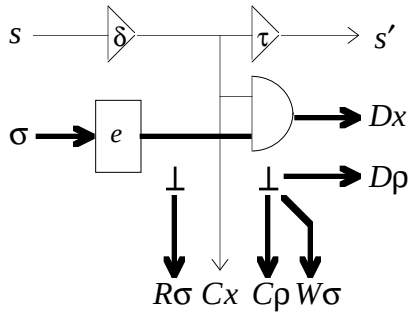
Construct: assignment

A variable assignment program $x := e$ yields the control

$s' = \tau \lhd \delta \lhd s \ \wedge \ Cx = \delta \lhd s \ \wedge \ Dx = (\delta \lhd s \wedge e) \ \wedge \ \neg R\sigma \ \wedge \ \neg C\rho \ \wedge \ \neg W\sigma \ \wedge \ \neg D\rho$

$\delta \geq (\ e$ time$) \ \wedge \ \tau \geq (\ s$ pulse time$) \geq$ (memory latch time)

where ρ is the state of memory except for x. Its diagram is



Box e evaluates the data expression in the assignment. We assume for now that adders and other circuits to perform numerical operations are available; when we have finished presenting high-level circuit design, we will have the means to design the circuits to perform integer and floating-point operations by writing programs that use only boolean variables and arrays with a restricted form of indexing. The input to e is shown as the entire state of memory, but in practice it is just the part of memory that e depends on. Expression e may depend on an array element; if so, the reading address for that array element must be output from the expression circuit, conjoined with s, and routed to memory (instead of $\perp$ as shown in the diagram). There may be references to elements of several arrays, but for now, assume there is at most one array element reference per array in e; later, the **result** expression will provide a way to allow an arbitrary number of array element references. We are also assuming that evaluation of expression e takes a uniform, known amount of time, and the δ delay must exceed that time; later, with the **result** expression we will remove that assumption. Each wire in the output from e is conjoined with s' to produce one of the wires of Dx. The $\perp$ outputs, as usual, are not really there.

An array element assignment program $A[i] := e$ yields the control

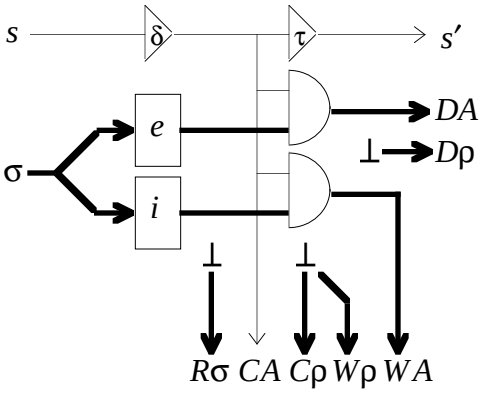
$s' = \tau \lhd \delta \lhd s$

$CA = \delta \lhd s \wedge DA = (\delta \lhd s \wedge e) \wedge WA = (\delta \lhd s \wedge i)$

$\neg R\sigma \wedge \neg C\rho \wedge \neg W\rho \wedge \neg D\rho$

$\delta \geq (\ e$ time$) \ \wedge \ \delta \geq (\ i$ time$) \ \wedge \ \tau \geq (\ s$ pulse time$) \geq$ (memory latch time)

where ρ is the state of memory except for A. Its diagram is

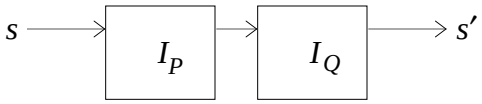


Construct: sequential composition

To implement sequential composition $P;Q$ we suppose that we already have the controls I_P and I_Q for programs P and Q. To avoid name clashes we systematically rename the inputs and outputs of I_P by adding the subscript $_P$, and similarly for I_Q. Then the control for $P;Q$ is

$I_P \wedge I_Q$
$s{=}s_P \wedge s'_P{=}s_Q \wedge s'_Q{=}s'$
$\sigma_P{=}\sigma_Q{=}\sigma$
$R\sigma{=}(R\sigma_P \vee R\sigma_Q) \wedge C\sigma{=}(C\sigma_P \vee C\sigma_Q) \wedge W\sigma{=}(W\sigma_P \vee W\sigma_Q) \wedge D\sigma{=}(D\sigma_P \vee D\sigma_Q)$

Diagrammatically, ignoring the connections between the controls and memory, we have

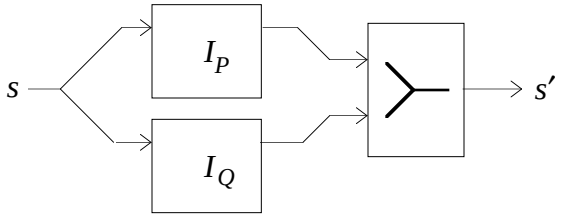


Construct: parallel composition

To implement parallel composition $P \| Q$,we need to start both programs (operands of $\|$ are often called “processes”), and then merge the completion pulses. We suppose that we already have the controls I_P and I_Q for programs P and Q. To avoid name clashes we systematically rename the inputs and outputs of I_P by adding the subscript $_P$, and similarly for I_Q. Then the control for $P \| Q$ is

$I_P \wedge I_Q \wedge merge$
$s{=}s_P{=}s_Q \wedge a{=}s'_P \wedge b{=}s'_Q \wedge s'{=}q$
$\sigma_P{=}\sigma_Q{=}\sigma$
$R\sigma{=}(R\sigma_P \vee R\sigma_Q) \wedge C\sigma{=}(C\sigma_P \vee C\sigma_Q) \wedge W\sigma{=}(W\sigma_P \vee W\sigma_Q) \wedge D\sigma{=}(D\sigma_P \vee D\sigma_Q)$

Diagrammatically, ignoring the connections between the controls and memory, we have



This implementation of parallel composition allows P and Q to access memory simultaneously. For the memory we have described, simultaneous access to different variables or arrays poses no problem. Even for the same variable, simultaneous reads are no problem. But simultaneously reading and writing the same variable, or two simultaneous writes to the same variable, have unpredictable results. We will soon introduce communication channels to allow programs to share information without memory contention.

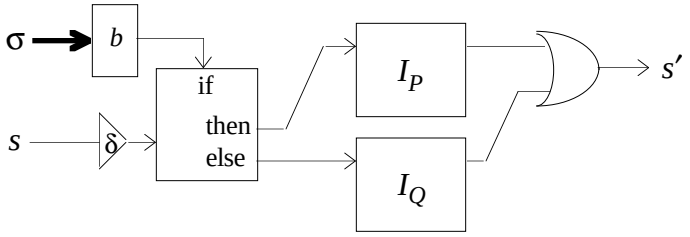
Construct: conditional composition

To implement conditional composition **if** b **then** P **else** Q we suppose that we already have the controls I_P and I_Q for programs P and Q. To avoid name clashes we systematically rename the inputs and outputs of I_P by adding the subscript $_P$, and similarly for I_Q. Then the control for **if** b **then** P **else** Q is

$I_P \wedge I_Q$

$s_P=(\delta\lhd s\wedge b)\ \wedge\ s_Q=(\delta\lhd s\wedge\neg b)\ \wedge\ s'=(s'_P\vee s'_Q)$
$\sigma_P=\sigma_Q=\sigma$
$R\sigma=(R\sigma_P\vee R\sigma_Q)\wedge C\sigma=(C\sigma_P\vee C\sigma_Q)\wedge W\sigma=(W\sigma_P\vee W\sigma_Q)\wedge D\sigma=(D\sigma_P\vee D\sigma_Q)$
$\delta\geq$ (b time)

Diagrammatically, ignoring the connections between the controls and memory, we have



The assumptions about b are the same as those about the expression in an assignment.

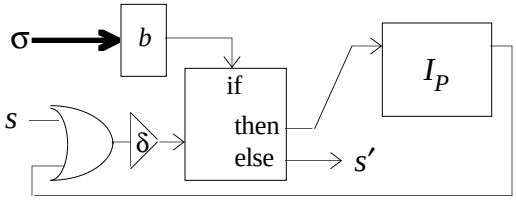
A one-tailed **if** b **then** P is just **if** b **then** P **else ok**. To make a circuit for a **case** program, the **if** circuit is generalized in the obvious way.

Construct: loop

To implement **while** b **do** P we suppose that we already have the control I_P for program P. To avoid name clashes we systematically rename the inputs and outputs of I_P by adding the subscript $_P$. Then the control for **while** b **do** P is

I_P
$s_P=(\delta\lhd(s\vee s'_P)\wedge b)\ \wedge\ s'=(\delta\lhd(s\vee s'_P)\wedge\neg b)$
$\sigma_P=\sigma\wedge R\sigma=R\sigma_P\wedge C\sigma=C\sigma_P\wedge W\sigma=W\sigma_P\wedge D\sigma=D\sigma_P$
$\delta\geq$ (b time)

Diagrammatically, ignoring the connections between I_P and memory, we have

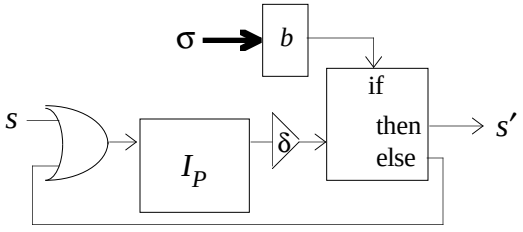


Again, the assumptions about expression b are the same as those about the expression in an assignment.

To implement **repeat** P **until** b we suppose that we already have the control I_P for program P. To avoid name clashes we systematically rename the inputs and outputs of I_P by adding the subscript $_P$. Then the control for **repeat** P **until** b is

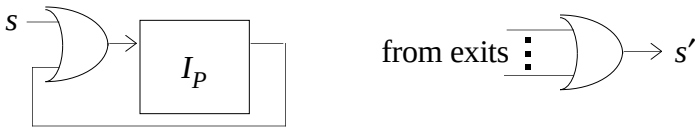
I_P
$s_P=(s\vee\delta\lhd s'_P\wedge\neg b)\ \wedge\ s'=(\delta\lhd s'_P\wedge b)$
$\sigma_P=\sigma\wedge R\sigma=R\sigma_P\wedge C\sigma=C\sigma_P\wedge W\sigma=W\sigma_P\wedge D\sigma=D\sigma_P$
$\delta\geq$ (b time)

Diagrammatically, ignoring the connections between I_P and memory, we have



Again, the assumptions about expression b are the same as those about the expression in an assignment.

Some programming languages include a loop with intermediate exits. Unlike the previous constructs, **loop** P and **exit** cannot be implemented in isolation, but must be implemented together. Ignoring the connections between I_P and memory, the control for **loop** P is

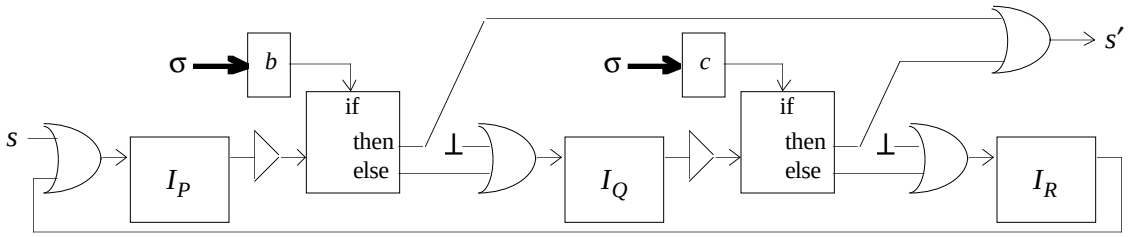


and that for an **exit** is

The **exit** wire to s' is shown only so that the circuit has the right inputs and outputs, but in practice it is unnecessary. To see how this works, consider the following example.

loop (*P*; **if** *b* **then exit**; *Q*; **if** *c* **then exit**; *R*)

Its circuit is



Each **exit** consists of a wire leading from a "then" to the final disjunction, and a ⊥ leading into a disjunction. These ⊥ inputs and the disjunctions they lead into can be eliminated.

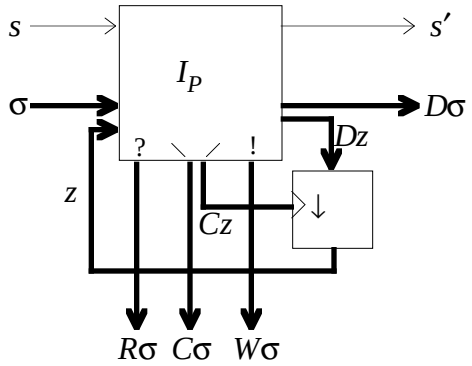
Now that we have **loop** and **exit** , we could have defined **while** and **repeat** as special cases.

Construct: local variable

To declare local variable z of type T with scope P we write **var** z: $T\cdot P$. It simply adds another word of memory, which is used only within P . Formally, its control is

$\exists z, Cz, Dz\cdot I_P$

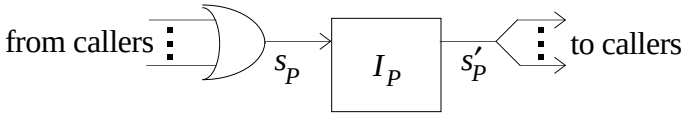
where I_P is the control for P . Local declaration helps to locate the words of memory near the control circuitry that uses them. The diagram:



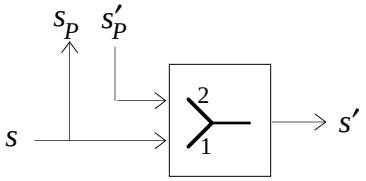
To declare local array A of size s and type T with scope P we write **var** $A[s]$: $T\cdot P$. The size must be a compile-time constant. It simply adds another RAM, which is used only within P. There is another way to implement array declarations that is preferable in some circumstances. We can treat the declaration of array $A[3]$ as syntactic sugar for the declaration of three variables $A0$, $A1$, $A2$. We treat the data expression $A[i]$ as sugar for case i of $A0 \mid A1 \mid A2$, and the assignment $A[i]:= e$ as sugar for **case** i **of** $A0:= e \mid A1:= e \mid A2:= e$. This implementation allows parallel access and update of array elements.

Construct: procedure

In many programming languages, a procedure is a unit of program that can be named, so that it can be called from several places, it is a scope for local declarations, and it can have parameters. These three aspects of procedures are separable; we have already dealt with local scope, we will come to parameters in a moment, and now we consider calls and returns. We suppose that we already have the control I_P for procedure P. This circuit is started from any of the calls, and indicates its completion to all calling points.



The calling points each become



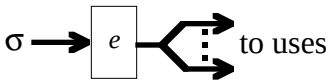
It is a programmer's responsibility (using communications to be described later) to make sure that calls from parallel programs are mutually exclusive, so that the procedure is not restarted before it completes an execution. Our implementation does not work for recursive calls in general, which are significantly harder (actually, the calls are easy but the returns are hard), but it does work for tail-recursive calls.

A parameter declaration can be treated exactly as though it were introducing a local variable instead of a parameter. Whenever a procedure P with parameter x is supplied an argument a, the resulting program $P\,a$ can be treated as though it were $(x:= a\,;\ P)$, except that x has been taken out of scope.

Construct: function

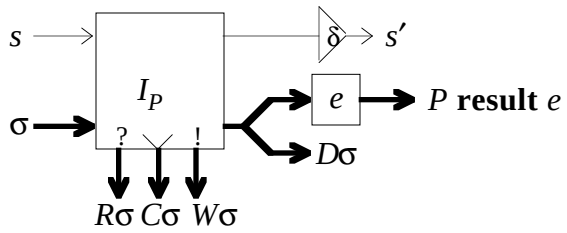
A function, in many languages, is even more of a mixture than a procedure. Its separable features are: the ability to name a data expression so that it can be used in different places; the ability to nest programs (statements) within a data expression; local scope; parameters. The last two aspects have been dealt with, and we now consider the first two.

To associate a name with a data expression e, just put the circuit to evaluate e somewhere. Its input comes from memory, and its output goes to all uses of the name. The diagram:



Data expressions occur in various forms of program, such as assignment and **if**. We have

been assuming that their evaluation time is predictable at compile-time, but to be general, we allow circuits for data expressions to have a control line (s input and s' output). The data expression P **result** e requires execution of program P in order to create the correct state for evaluation of e. Its circuit inserts the appropriate delay in the control line. The delay may depend on the initial state, varying from one evaluation to another; it is not a worst-case delay. Its diagram is



where I_P is the control for program P and $\delta \geq$ (e time) . If P changes only local variables, so that there are no side-effects, then the outputs $C\sigma$, $W\sigma$, $D\sigma$ to memory are unnecessary. Expression e should be evaluated in the local scope, so the input to e should include local variables as necessary. A **result** expression is often used as the body of a function. Another use is to help us out of an earlier difficulty: we were not allowed to have references to different elements of the same array within one basic data expression. But a compiler can transform an expression like $A[i]+A[j]$ into

(**var** t: **int**· t:= $A[i]$ **result** $t+A[j]$)

and so we now lift the earlier restriction.

Construct: communication

To declare local channel c of type T with scope P we write **chan** c: T· P . For one writing program and one reading program it is defined as follows.

(**chan** c: T· P) = (**var** c: T· **var** $\surd c$: **bool**· $\surd c$:= $\perp$; P)

It introduces two variables, called the buffer and the probe. The buffer c (same name as the channel) holds the value being communicated, and the probe $\surd c$ (pronounced "check c") tells whether there is an unread message in the buffer. We define output of expression e and input to variable x on this channel as follows.

$c!\ e$ = (**while** $\surd c$ **do tick**; c:= e; $\surd c$:= $\top$)

$c?\ x$ = (**while** $\neg\surd c$ **do tick**; x:= c; $\surd c$:= $\perp$)

Since we have already implemented all constructs on the right sides of these definitions, we therefore have implementations of channel declaration, input, and output. But there are two points that need attention. The **tick** delay must be longer than the control pulse (the pulse on s) so the control pulse is not lost. And the **while** must use an edge-triggered switch so the control pulse will not be truncated, split, or otherwise damaged by a change in $\surd c$ due to a parallel program. Although the buffer may also be shared by parallel programs that both read and write it, the discipline of use imposed by input and output ensures noninterference.

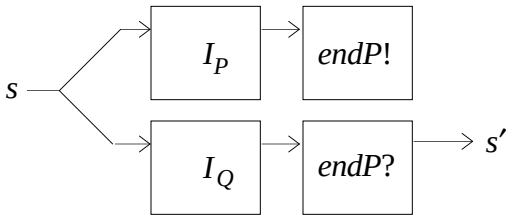
It may also be useful to introduce signals, which are messages without content. To declare local signal s with scope P we write **sig** s· P . For one sending program and one receiving program it is defined as follows.

(**sig** s· P) = (**var** $\surd s$: **bool**· $\surd s$:= $\perp$; P)

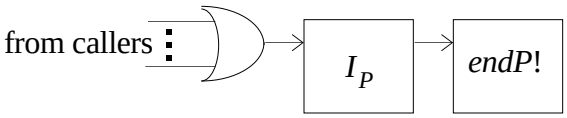
It introduces only the probe. We define sending and receiving this signal as follows.

$s!$ = (**while** $\surd s$ **do tick**; $\surd s := \top$)
$s?$ = (**while** $\neg\surd s$ **do tick**; $\surd s := \bot$)

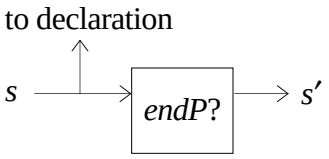
As before, the **tick** delay must be longer than the control pulse, and the **while** must use an edge-triggered switch. As examples of their use, we offer a second implementation of parallel programs. For each parallel composition $P \| Q$ we introduce a signal *endP* and the circuit is



We can also use a signal to reimplement procedures. For each procedure P introduce signal *endP* and the circuit is

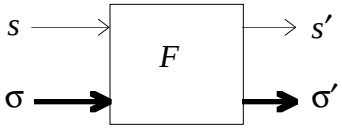


A call to P can be implemented as follows.



9 Functional Circuits

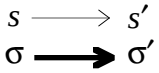
The second of our two translations from programs to circuits produces "functional" circuits (as in "functional programming"). Functional circuits are not composed of a control and a memory; instead, each functional circuit computes its output σ' from its input σ without the benefit of a separate memory (although some constructs will require internal memory). There is still a start signal s to initiate the computation and a stop signal s' to indicate completion. Here's the diagram.



The data input σ includes all variables and array elements. To use the circuit we must provide the desired data input σ and a pulse (momentary $\top$) on the start wire s, and we must hold σ constant ever after (until the circuit is restarted). The functional circuit F must provide a correct data output σ' and a pulse on the stop wire s', and it must hold σ' constant ever after (until the circuit is restarted).

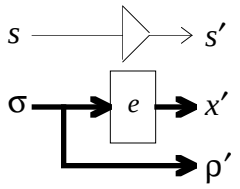
Construct: empty

For program **ok** the functional circuit is trivial, as it should be.



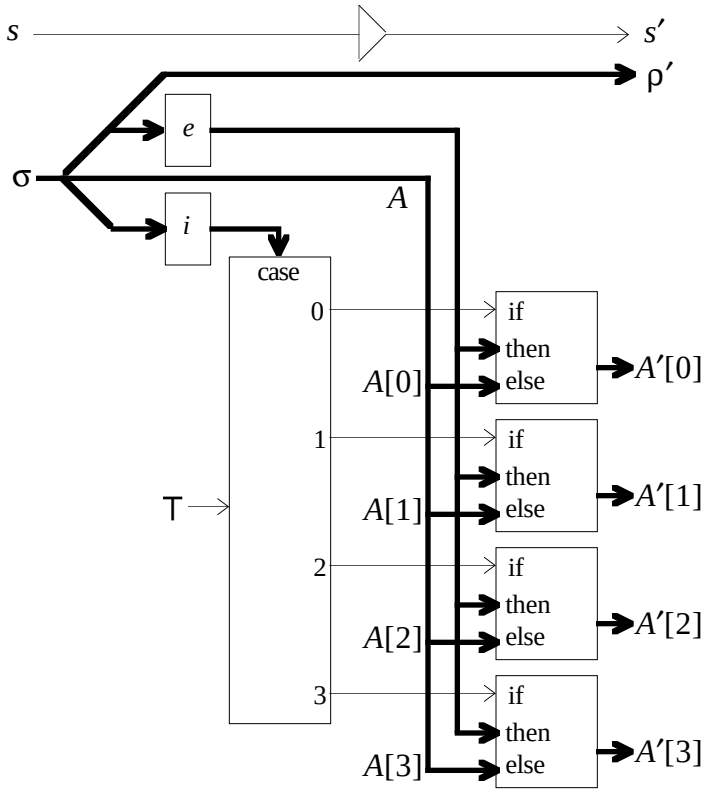
Construct: assignment

A variable assignment $x := e$ looks like this:



where ρ' is all variables and arrays other than x'.

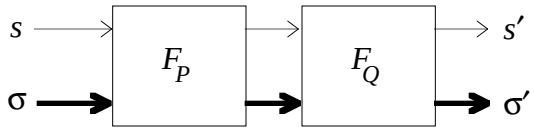
Array element assignment $A[i] := e$ looks like this:



where ρ' is all variables and arrays other than A'.

Construct: sequential composition

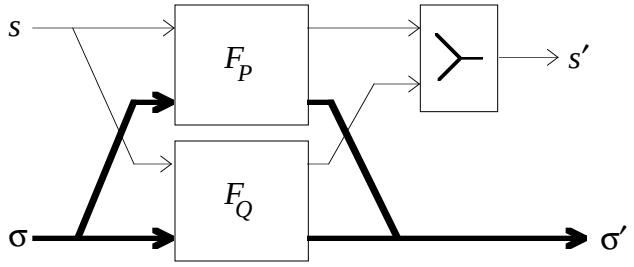
Sequential execution is easy.



Construct: parallel composition

For parallel execution, we make the simplifying assumption that the parallel programs do not communicate via shared memory, but only via the communication constructs provided. The variables and array elements changed by one program are disjoint from those changed by a parallel program, and the changes made by one program are not seen by a parallel

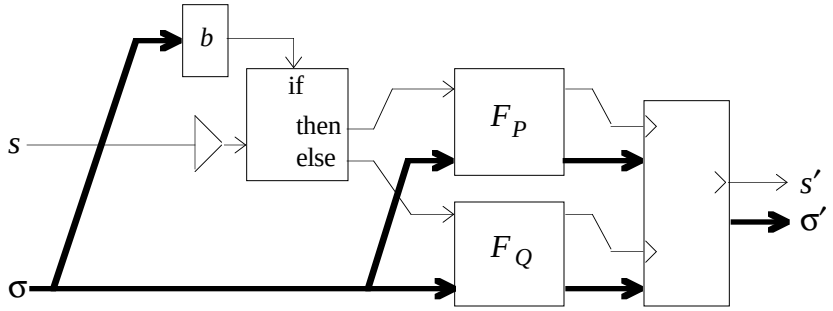
program.



The entire input can go to both programs, but each program produces only part of the output. These outputs together form the entire output.

Construct: conditional composition

Here is the functional circuit for **if** *b* **then** *P* **else** *Q* .

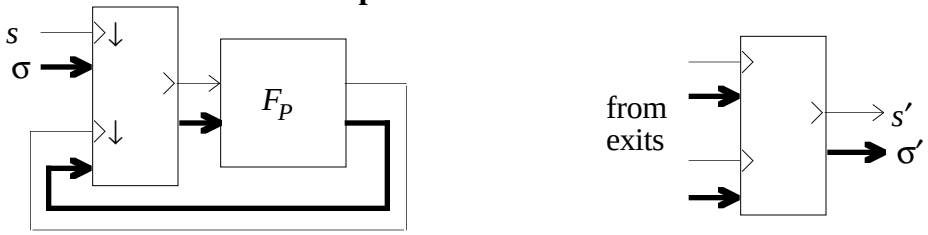


The data output could, alternatively, be selected using the *b* output without memory.

The **if** circuit is easily generalized to a **case** circuit.

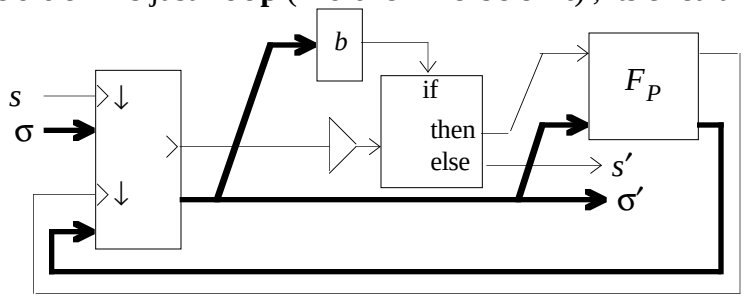
Construct: loop

Here is the functional circuit for **loop** *P* with **exit**s in it.



If there is only a single exit, the exit memory is unnecessary.

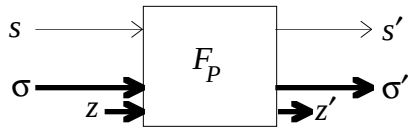
Since **while** *b* **do** *P* is just **loop** (**if** *b* **then** *P* **else exit**) , its circuit is



Similarly **repeat** *P* **until** *b* is just **loop** (*P*; **if** *b* **then exit else ok**).

Construct: local variable

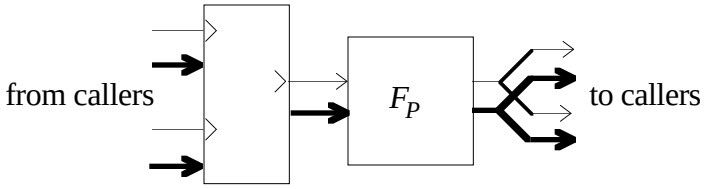
The functional circuit for local variable declaration **var** z: $T\cdot P$ is particularly easy.



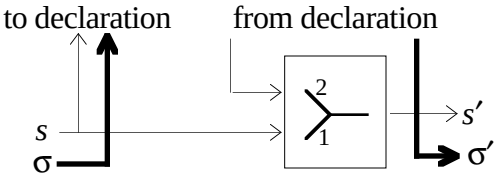
Into the functional circuit for P we must feed data lines for z, with any desired initial value. The diagram shows the final value z' coming from F_P, but it is not wanted so its wires are not needed. A local array declaration is just like a local variable declaration; there is no extra circuitry needed here for access to elements.

Construct: procedure

Program declaration and calling work the same way in functional circuits as in imperative circuits, except that the data input must also come from the calling point, and the data output must be delivered back to the calling point. Like the imperative version, our functional implementation of procedures does not work for general recursion. Because the functional parallelism is disjoint, procedures cannot be called from parallel programs. For a procedure declaration we have the circuit

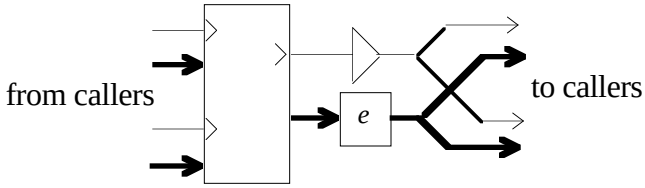


and for each call we have the circuit



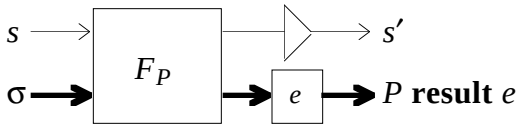
Construct: function

Function declaration is similar to procedure declaration.



Function call and return are identical to procedure call and return. The programmer must ensure that calls from parallel programs are mutually exclusive.

The data expression P **result** e is almost identical for functional and imperative circuits.



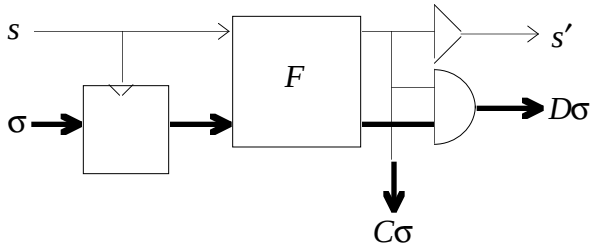
The communication constructs are not given functional implementations for lack of truly shared memory in the parallel composition. For these constructs, we use hybrid circuits, described next.

10 Hybrid Circuits

In general, functional circuits are faster than imperative circuits, but imperative circuits occupy less space. Each approach has merit. We can obtain almost the speed of a functional circuit with almost the compactness of an imperative circuit by combining the two kinds within one hybrid circuit. For example, we might make most of a circuit imperative, but make inner loops functional.

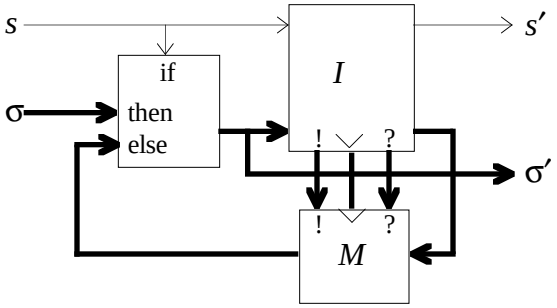
Hybrid circuits also allow us to use our imperative implementation of communication within a larger functional circuit.

To place a functional circuit within an imperative one, we must make the functional circuit look imperative. Ignoring arrays for simplicity of presentation, here's what we do.



The s' output also causes memory to change state. As usual, only some of the wires to and from memory are needed. The local memory is needed only if there are parallel programs.

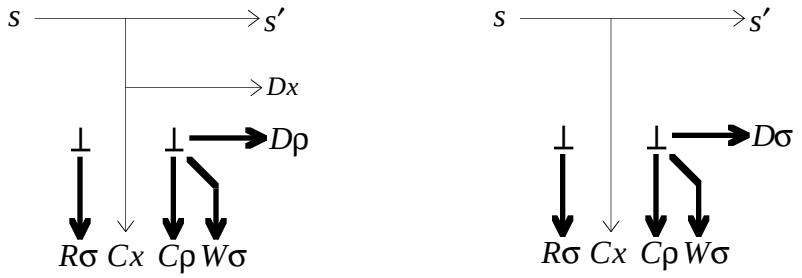
To place an imperative circuit within a functional one, we must make the imperative circuit look functional. Here's what we do.



In effect, we make the memory (as much of it as as necessary) local to the circuit.

11 Optimization

We have presented the best imperative and functional circuits that we have been able to devise, in general, for each construct in isolation. But in special cases, or in combination, we can sometimes do better. We may find a conjunction or disjunction with $\top$ or $\bot$, which can then be eliminated. For example, the imperative circuits for $x := \top$ and $x := \bot$ are



Infinite programs in parallel do not need a *merge* circuit to detect their completion. There are other special cases that can be optimized.

12 Performance

Our two measures of performance are circuit size and execution time. We measured size as the number of conjunctions, disjunctions, negations, and delays. Other gates were expressed in terms of these gates; one bit of memory has size 4 (we did not count the delay). The size measure might be improved by choosing different primitives. We measured time as the number of sequential gate delays during execution. We measured circuits that were produced without any optimizations; the results might be better if we were to optimize. In any case, the performance numbers are very approximate and indicate only that our circuits are competitive.

We measured our imperative circuits, our functional circuits, the circuits of Martin [3,7], the circuits of Weber et al. [15], and the circuits of Philips [1,2]. To obtain our results, we hand-compiled the circuits of Martin; for details see [11].

Our six test programs were chosen to use as many language features as possible. They are:

Parity: computes and verifies a parity bit for 3 bits of data
Parallel: uses the parallelism and interprogram communication features
Arbiter: arbitration between two parallel programs
Counter: a 4-bit binary countdown timer and test-and-set function
Triple: a program to compute three times its input
Ring: mutually-exclusive execution of 7 parallel programs in a ring using Peterson's algorithm

A detailed description of the test programs and a similar analysis can be found in [11].

The results are listed in the following table.

TIME	*Parity*	*Parallel*	*Arbiter*	*Counter*	*Triple*	*Ring*
imperative	16	31	165	1091	127	2321
functional	16	25	113	799	90	3792
Martin	18	23	209	1243	148	2044

SIZE	*Parity*	*Parallel*	*Arbiter*	*Counter*	*Triple*	*Ring*
imperative	73	210	52	407	400	427
functional	94	413	81	886	1240	1835
Martin	98	123	76	431	385	357
Weber et al.	259	145	95	1156	657	715
Philips	96	196	79	810	597	940

13 Correctness

To prove that our circuits are correct, we must have a formal semantics for our source programs and circuits. Here is the source semantics.

Let t and t' be the initial and final execution times, the times at which execution starts and ends. If the execution time is infinite, $t'=\infty$. Let the state variables x, y, ... be functions of time. The value of x at time t is $x\,t$. An expression such as $x+y$ is also a function of time; its argument is distributed to its variable operands as follows: $(x+y)t = x\,t + y\,t$. Let

$wait = (t'\geq t \wedge \forall t'': t\leq t''\leq t' \cdot xt''=xt \wedge yt''=yt \wedge \ldots)$

so that *wait* takes an arbitrary time during which the variables are unchanging.

The programming notations are defined as follows.

ok $= (t'=t)$

tick $= (t'=t+\delta \wedge wait)$

$(x:= e) = (t'=t+\delta+\tau \wedge xt'=e(t+\delta) \wedge wait_{y,z\ldots})$

where $\delta \geq$ (e time) $\wedge$ $\tau \geq$ (memory time) .

$(P;Q) =$ $\exists t'' \cdot$ (substitute t'' for t' in P) $\wedge$ (substitute t'' for t in Q)

$(P_\alpha \parallel Q_\beta) = (P_\alpha \wedge (Q; wait)_\beta \vee (P; wait)_\alpha \wedge Q_\beta)$

(**if** b **then** P **else** Q) $=$ (if $b(t+\delta)$ then P else Q) $=$ $(b(t+\delta) \wedge P \vee \neg b(t+\delta) \wedge Q)$

where $\delta \geq$ (b time) .

(**while** b **do** P) $\Rightarrow$ **if** b **then** (P; **while** b **do** P) **else ok**

($\forall x, x', y, y', \ldots, t, t' \cdot$ $W \Rightarrow$ **if** b **then** (P; W) **else ok**)

$\Rightarrow$ ($\forall x, x', y, y', \ldots, t, t' \cdot$ $W \Rightarrow$ **while** b **do** P)

var z: $T \cdot P = \exists z$: *time*$\rightarrow T \cdot P$

where *time*$\rightarrow T$ is the functions from time values (including ∞) to T values.

Here is a simple example, in variables x and y. In this example we use discrete time and take δ to be 0 and τ to be 1 .

$x:= x+3;\ x:= x+4$

$= (t'=t+1 \wedge xt'=xt+3 \wedge yt'=yt);\ (t'=t+1 \wedge xt'=xt+4 \wedge yt'=yt)$

$= \exists t'' \cdot\ (t''=t+1 \wedge xt''=xt+3 \wedge yt''=yt) \wedge (t'=t''+1 \wedge xt'=xt''+4 \wedge yt'=yt'')$

$= t'=t+2 \wedge x(t+1)=xt+3 \wedge x(t+2)=xt+7 \wedge yt=y(t+1)=y(t+2)$

In the parallel composition, α consists of those variables that appear on the left of assignments within P, and β consists of those variables that appear on the left of assignments within Q; α and β must be disjoint. The use of *wait* is just to make the faster side of the parallel composition wait until the slower side is finished. To illustrate the semantics, here is an example in variables x and y, and discrete time with $\delta=0$ and $\tau=1$. In the left-hand program, only x is assigned, so only x is treated as a state variable. In the right-hand program, only y is assigned, so only y is treated as a state variable.

$(x:= 2;\ x:= x+y;\ x:= x+y) \parallel (y:= 3;\ y:= x+y)$

$= \quad (t' = t+1 \wedge xt'=2;\ t' = t+1 \wedge xt' = xt+yt;\ t' = t+1 \wedge xt' = xt+yt)$

$\quad \wedge (t' = t+1 \wedge yt'=3;\ t'=t+1 \wedge yt' = xt+yt;\ t'\geq t \wedge \forall t'': t\leq t''\leq t' \cdot yt''=yt)$

$\quad \vee \quad (t' = t+1 \wedge xt'=2;\ t' = t+1 \wedge xt' = xt+yt;\ t' = t+1 \wedge xt' = xt+yt;$

$\qquad t'\geq t \wedge \forall t'': t\leq t''\leq t' \cdot xt''=xt)$

$\quad \wedge (t' = t+1 \wedge yt'=3;\ t'=t+1 \wedge yt' = xt+yt)$

$= \quad t' = t{+}3 \ \wedge\ x(t{+}1){=}2 \ \wedge\ x(t{+}2) = x(t{+}1){+}y(t{+}1) \ \wedge\ x(t{+}3) = x(t{+}2){+}y(t{+}2)$
$\quad \wedge\ t' \geq t{+}2 \ \wedge\ y(t{+}1){=}3 \ \wedge\ y(t{+}2) = x(t{+}1){+}y(t{+}1) \ \wedge\ \forall t'' .\ t{+}2 \leq t'' \leq t' \cdot\ yt'' = y(t{+}2))$
$\quad \vee\ \ t' \geq t{+}3 \ \wedge$ (other conjuncts)
$\quad \wedge\ t' = t{+}2 \ \wedge$ (other conjuncts)
$= \ t' = t{+}3 \ \wedge\ x(t{+}1){=}2 \ \wedge\ y(t{+}1){=}3 \ \wedge\ x(t{+}2){=}5 \ \wedge\ y(t{+}2){=}5 \ \wedge\ x(t{+}3){=}10 \ \wedge\ y(t{+}3){=}5$

The example has the appearance of lock-step parallelism, as though there were a global clock, only because, for the sake of simplicity, we used discrete time with constants $\delta{=}0$ and $\tau{=}1$ for all assignments.

The first formula concerning the **while** loop says that it refines its first unrolling. Stated differently, **while** *b* **do** *P* is a pre-fixed-point of $W \ \Rightarrow$ **if** *b* **then** (*P*; *W*) **else ok**. The second formula says that it is as weak as any pre-fixed-point, so it is the weakest pre-fixed-point.

The other programming constructs (channel declaration, input, output, signal declaration, sending, receiving, parameter declaration, argumentation) are defined in terms of the ones we have already defined, so we do not need to give them a separate semantics. And that completes the source semantics.

The imperative circuit semantics was given with each circuit. For example, the control for **ok** was

$s' = s \ \wedge\ \neg R\sigma \ \wedge\ \neg C\sigma \ \wedge\ \neg W\sigma \ \wedge\ \neg D\sigma$

and the control for **while** *b* **do** *P* was

I_P
$s_P = (\delta \lhd (s \vee s'_P) \wedge b) \ \wedge\ s' = (\delta \lhd (s \vee s'_P) \wedge \neg b)$
$\sigma_P = \sigma \wedge R\sigma = R\sigma_P \wedge C\sigma = C\sigma_P \wedge W\sigma = W\sigma_P \wedge D\sigma = D\sigma_P$
$\delta \geq$ (*b* time)

where I_P is the control for P.

Before we can prove correctness, we need one more idea, adapted from [9]. Roughly speaking, a circuit is "busy" if it has been started and has not yet stopped. Formally, define B as

$B \ = \ ((s \vee \delta \lhd B) \wedge \neg s')$
$\delta \ \leq$ (pulse time)

The delay here must be shorter than the pulse length used on the control lines (s and s'). If time is discrete and $\delta{=}1$, then for any A

$(\lhd A)\ 0 \ = \ \bot$
$(\lhd A)\ (t{+}1) \ = \ At$

and so for busy B

$B0 \ = \ \bot$
$B(t{+}1) \ = \ ((s(t{+}1) \ \vee Bt) \wedge \neg s'(t{+}1))$

To prove that a circuit is correct, we must prove

$I_P \wedge M \wedge st \wedge (\forall t'' \cdot\ Bt'' \wedge \lhd Bt'' \Rightarrow \neg st'') \wedge t' = (min\ t'' \cdot\ t'' \geq t \cdot \wedge\ s't'') \ \Rightarrow\ P$

Suppose we have the control I_P (for program P), and we have the memory M , and we put a pulse on the start wire s at time t , and we don't try to restart the circuit while it's busy, and we give the name t' to the first time at or after t when s' becomes $\top$; then we expect the circuit to satisfy the semantics of program P . We do not have to prove correct each circuit that we design; instead, we prove that our circuit generation scheme is correct. The proof is long, and we omit it, stating only two lemmas that are useful steps on the way to the proof.

(a) $\quad I \wedge \neg \lhd B \wedge \neg s \ \Rightarrow\ \neg s'$

which says that a circuit does not spontaneously generate s'.
(b) $I \wedge \neg B \Rightarrow \neg R\sigma \wedge \neg C\sigma \wedge \neg W\sigma \wedge \neg D\sigma$
which says that if a circuit is not busy, its $R\sigma$, $C\sigma$, $W\sigma$, and $D\sigma$ outputs are all $\perp$.

14 Synchronous and Asynchronous Circuits

There are two ways to control the timing in circuits. One is by using delays calculated, or experimentally determined, to be long enough to ensure that all data values have settled properly. The other way, called "delay-insensitive", is to use handshaking signals that allow a data transfer to occur just when both sender and receiver are ready. These solutions can be applied locally, or globally, or at any level in between. The word "synchronous" is usually used to describe a global delay, or clock; the word "asynchronous" is sometimes used to describe local handshaking.

The circuits resulting from the methods we have presented use local delays. But as a special case, it is possible to write a program in the form of a single loop, whose body is a parallel composition of assignments.

loop ($x := e_x \parallel y := e_y \parallel z := e_z \parallel \ldots$)

This program structure forces a single, common delay for all state changes; that delay is in effect a global clock. We can thus program a synchronous circuit when we want one. When designing a circuit, there is little point in aiming for the synchronous structure, and equally little point in aiming to avoid it. One chooses a program structure that is appropriate for the task, and one gets a circuit that accomplishes that task.

In principle, local delays should be faster than a single global delay. That is because a global delay must be the maximum of all the local delays. In a synchronous circuit, each state change takes as long as the slowest state change requires. In principle, local delays should also be faster than local handshaking. That is because the handshaking takes time. A local delay is just long enough for the data to be ready, not long enough for the data to be ready and to indicate its readiness.

15 Conclusions

Circuit design can be done more effectively by describing the function that a circuit is intended to perform than by describing a circuit that is intended to perform that function. A programming language is more convenient for that purpose than a gate-level language. It seems quite obvious that complex circuits can be designed this way more easily and reliably than by low-level gate descriptions. And the resulting circuits seem, from a preliminary investigation, to show the promise of competing successfully with hand-crafted circuits. They should be smaller and faster than synchronous circuits due to the absence of a global clock. They should also be smaller and faster than delay-insensitive circuits due to the absence of handshaking. These gains come at a price: the language implementer must provide local delays. We do not suppose it is easy to provide local delays, but this price is paid only once; circuit designers who use the high-level language do not need to be concerned with them.

We have compiled a sampling of programming constructs that are representative of many high-level languages. Some obviously desirable constructs, such as modules, are missing only because they do not present any circuit generation problems (modules restrict the use of identifiers). For programs that we compiled and simulated, and for the text of

the simulators, see [11].

We have shown two ways to implement ordinary programs with logic gates. The logic gates can, of course, be implemented with electronic transistors, resistors, and diodes. We could therefore bypass the logic gates, implementing the programs directly with transistors, resistors, and diodes. Doing so makes more optimizations and more efficient circuits possible. Ultimately, perhaps logic gates will have no remaining role in circuit design.

Acknowledgments We received useful feedback from IFIP WG2.3, and a lot of help from Jan van de Snepscheut. The paper was improved, both in content and presentation, in response to helpful criticisms from very competent and thorough anonymous referees.

References

1. C.H.vanBerkel, J.Kessels, M.Roncken, R.W.J.J.Saeijs, F.Schalij: "The VLSI programming language Tangram and its translation into handshake circuits". In *Proceedings of the European Design Automation Conference*, 1991.
2. C.H.vanBerkel: *Handshake circuits – an asynchronous architecture for VLSI programming*. Cambridge University Press, 1993.
3. S.M.Burns, A.J.Martin: "Performance analysis and optimization of asynchronous circuits". In *Proc. of the 1991 UC Santa Cruz Conf. on VLSI*, MIT Press, 1991.
4. C.DelgadoKloos: *Semantics of Digital Circuits*, LNCS 285, Springer, 1987.
5. E.C.R.Hehner: "Abstractions of Time". In *A Classical Mind: Essays in Honour of C.A.R.Hoare*, A.W. Roscoe (ed.), Prentice-Hall, 1994.
6. W.Luk, D.Ferguson, I.Page: "Structured Hardware Compilation of Parallel Programs". In *More Field-Programmable Gate Arrays*, W.Moore and W.Luk (eds.), Abingdon EE&CS Books, 1994.
7. A.J.Martin: "Programming in VLSI: from communicating processes to delay-insensitive circuits". In *Developments in Concurrency and Communication*, C.A.R.Hoare (ed.), Addison-Wesley, University of Texas at Austin Year of Programming Series, 1990.
8. S.Mazor, P.Langstraat: *A Guide to VHDL*, Kluwer, 1992
9. T.S.Norvell: *A Predicative Theory of Machine Languages and its Application to Compiler Correctness*. PhD thesis, University of Toronto, 1994
10. I.Page, W.Luk: "Compiling occam into field-programmable gate arrays". In *Field-Programmable Gate Arrays*, W.Moore and W.Luk (eds.), p. 271-283, Abingdon EE&CS Books, 1991.
11. R.F.Paige: *Correctness and Performance Analysis of Imperative and Functional Circuits*. MSc thesis, University of Toronto, 1994.
    www.cs.yorku.ca/~paige/Writing/MSc.dvi
12. M.Rem: *Partially Ordered Computations with Applications to VLSI Design*, Technical Report MR83/3, Eindhoven University of Technology, 1982
13. J.L.A.van de Snepscheut: *Trace Theory and VLSI Design*, LNCS 200, Springer, 1985
14. D.E.Thomas, P.Moorby: *The Verilog Hardware Description Language*, Kluwer, 1991
15. S.Weber, B.Bloom, G.Brown: "Compiling Joy into silicon". In *Advanced Research in VLSI and Parallel Systems*, T. Knight and J. Savage (eds.), MIT Press, 1992.