program2circuit

ERIC C.R. HEHNER
University of Toronto
CANADA
hehner@cs.utoronto.ca

THEODORE S. NORVELL
Memorial University of Newfoundland
CANADA
theo@engr.mun.ca

Abstract: - We present a new way to implement ordinary programs with logic gates. and a new method of timing within circuits, and a new method of circuit verification. Application-specific 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 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.

Key-Words: - digital circuit design

1 Introduction

The usual alternative to building application-specific circuits is to use a general-purpose processor, and customize it for an application by writing a program. 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 space savings compared to a combination of software and processor.

The VHDL [8] and Verilog [13] languages are presently being used by industry. 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].

There are other high-level circuit design techniques being developed and reported in the literature. Early work includes [11], [12], and [4]. In [3] and [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] and [2], except that specifications are mapped into connections of small components for which standard transistor implementations exist. In [14] 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 works of [6] and [10] are most similar to ours, but their designs have a global clock; ours do not.

2 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 circuit graphic is similarly a triangle. 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.

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. (We use $\bot$ for low voltage, ground, or false, and $\top$ for high voltage, power, or true.)

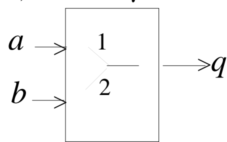
3 Merge

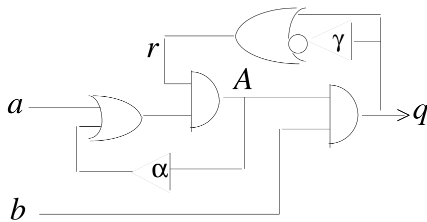
A merge turns two sequences of pulses into a single sequence of pulses. (A pulse is a momentary $\top$). The 1-2-merge has inputs a and b and output q . It outputs a pulse when pulses arrive on a and b in that order, or simultaneously, but not in the other order. To design a 1-2-merge, we introduce an 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)

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 top diagram, or they can be hidden as in the bottom diagram and the following specification:

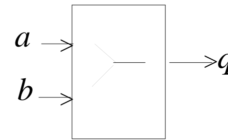
$\exists r, A \cdot \quad r \ = \ (q \vee \neg\gamma \triangleleft q)$
$\quad \wedge \quad A \ = \ (r \wedge (a \vee \alpha \triangleleft A))$
$\quad \wedge \quad q \ = \ (A \wedge b)$

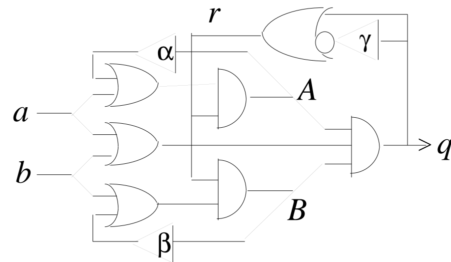
If a pulse on a follows a pulse on b , there must be a delay of at least γ after the end of b before the start of a to avoid truncating the output pulse. No circuit can constrain its inputs; its context of use must constrain its inputs, so a constraint is expressed formally as an antecedent rather than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

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

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

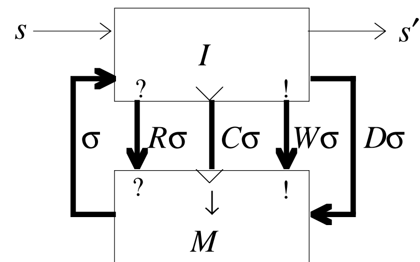
$r \ = \ (q \vee \neg\gamma \triangleleft q)$
$A \ = \ (r \wedge (a \vee \alpha \triangleleft A))$
$B \ = \ (r \wedge (b \vee \beta \triangleleft B))$
$q \ = \ (A \wedge B \wedge \ (a \vee b))$
$\alpha \le$ (pulse time) $\wedge$ $\alpha \le \gamma$
$\beta \le$ (pulse time) $\wedge$ $\beta \le \gamma$





4 Overview

The circuits that result from the translation have two components: a control I , and a memory M , connected as follows.



A thin line indicates one wire; a thick line indicates many wires. 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 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 $D\sigma$. For the arrays, the writing address wires are WA and WB, collectively called $W\sigma$, and the reading address wires are RA and RB, collectively called $R\sigma$. The memory outputs are x, y, $A[RA]$ and $B[RB]$, collectively called σ, the state of memory. Altogether, memory is

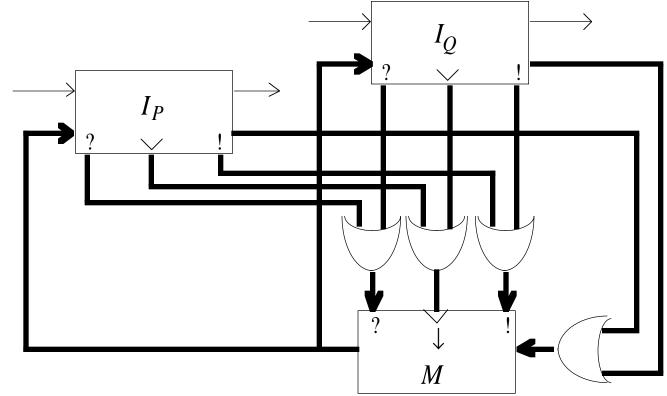
$$\begin{aligned} M \ = \ \ &(\ x = (\underline{\text{if}}\ \neg Cx \wedge \gamma\triangleleft Cx\ \underline{\text{then}}\ \triangleleft Dx\ \underline{\text{else}}\ \triangleleft x) \\ &\wedge\ y = (\underline{\text{if}}\ \neg Cy \wedge \gamma\triangleleft Cy\ \underline{\text{then}}\ \triangleleft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

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

The expression $\neg Cx \wedge \triangleleft Cx$ says that the clock for x is down but was just previously up, so it is a falling edge. The Cx-delay γ should be just large enough to allow Cx to fall and to allow that falling edge to be negated. The Dx-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). The x-delay should be as small as possible. Similarly for the other variables and arrays.

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 $\bot$ 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.



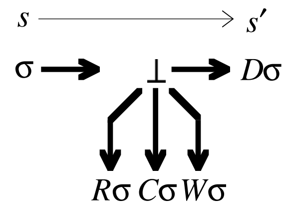
Each disjunction is really many disjunctions, one for each bit in its operands.

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 programming languages.

4.1 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

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



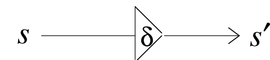
We have shown all its inputs and outputs. But since the σ input is not connected to anything, there is no point in bringing those wires from memory. And since the $R\sigma$, $C\sigma$, $W\sigma$, and $D\sigma$ outputs are $\bot$, there is no point in taking them into a disjunction. So the circuit reduces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

4.2 Construct: delay

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

$$s'{=}\delta\triangleleft s\ \wedge\ \neg R\sigma\ \wedge\ \neg C\sigma\ \wedge\ \neg W\sigma\ \wedge\ \neg D\sigma$$

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

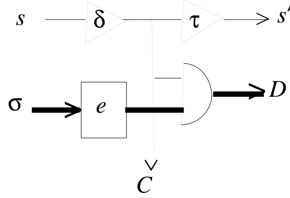


4.2 Construct: assignment

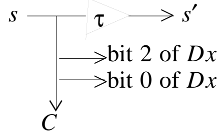
A variable assignment program $x{:=}\ e$ yields the control

$$\begin{aligned} &s'{=}\tau\triangleleft\delta\triangleleft s\ \wedge\ Cx{=}\delta\triangleleft s\ \wedge\ Dx{=}(\delta\triangleleft s \wedge e) \\ &\neg R\sigma\ \wedge\ \neg C\rho\ \wedge\ \neg W\sigma\ \wedge\ \neg D\rho \\ &\delta \ge (\ e\ \text{ time}) \end{aligned}$$

$\tau \geq (\, s$ pulse time$) \geq$ (memory latch time)
where ρ is the state of memory except for x .



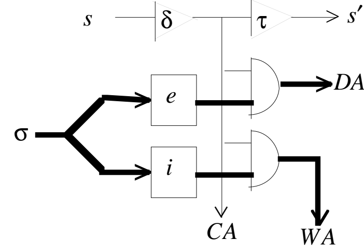
Box e evaluates the data expression in the assignment. We assume for now that adders and other circuits to perform numerical operations are available; when we have finished presenting high-level circuit design, we will have the means to design the circuits to perform integer and floating-point operations by writing programs that use only boolean variables and arrays with a restricted form of indexing. Adders and other arithmetic circuits may be duplicated at each use for maximum speed, or shared among several uses (by means of the function call circuitry which we present later), at the programmer's discretion. 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. When the expression e is a constant, there is a further simplification. For example, the assignment $x{:=}\ 5$ results in the circuit



since the binary representation of 5 , which is ...0000101 , has 1s at bit positions 0 and 2. 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. 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. 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$)$
$\tau \geq (\, s$ pulse time$) \geq$ (memory latch time)
where ρ is the state of memory except for A .

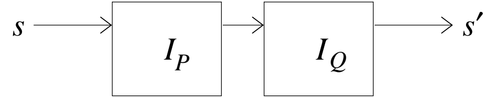


4.3 Construct: sequential composition

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

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

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

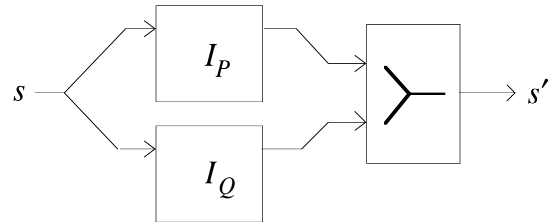


4.4 Construct: parallel composition

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

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

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

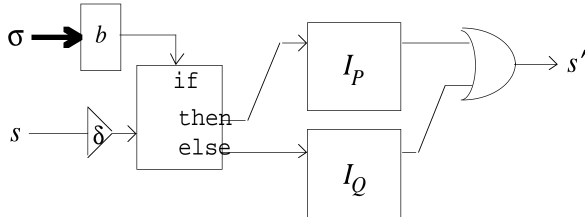
will soon introduce communication channels to allow programs to share information without memory contention.

4.5 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

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

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. The if-then-else box is a one-bit demultiplexer.

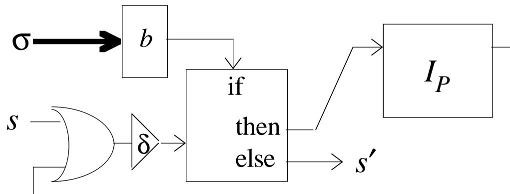
A one-tailed **if** b **then** P is just **if** b **then** P **else ok**. To make a circuit for a **case** program, the **if** circuit is generalized in the obvious way.

4.6 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
$s_P=(\delta \lhd (s \vee s'_P) \wedge b) \wedge s'=(\delta \lhd (s \vee s'_P) \wedge \neg b)$
$\sigma_P=\sigma \wedge R\sigma=R\sigma_P \wedge C\sigma=C\sigma_P$
$W\sigma=W\sigma_P \wedge D\sigma=D\sigma_P$
$\delta \geq (b \text{ time})$

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



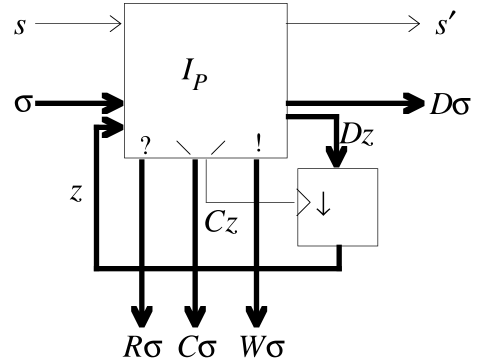
Again, the assumptions about expression b are the same as those about the expression in an assignment.

4.7 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

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

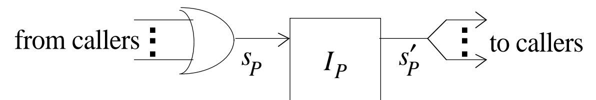
where I_P is the control for P. Local declaration helps to locate the words of memory near the control circuitry that uses them.



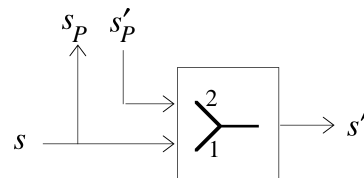
To declare local array A of size s and type T with scope P we write **var** $A[s]$: $T \cdot P$. The size must be a compile-time constant. It simply adds another RAM, which is used only within P. There is another way to implement array declarations that is preferable in some circumstances. We can treat 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.

4.8 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,

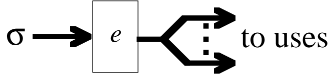
which are significantly harder (actually, the calls are easy but the returns are hard), but it does work for tail-recursive calls.

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.

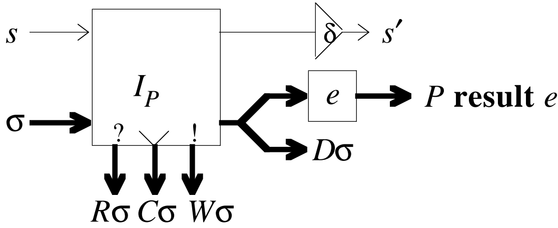
4.9 Construct: function

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.



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.



I_P is the control for program P and $\delta \ge (\ e$ 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.

4.10 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**· $\surd c{:=}\ \bot$; 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**; $x{:=}\ c$; $\surd c{:=}\ \bot$)

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.

5 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
= $(t'{\ge}t\ \wedge\ \forall t''{:}\ t{\le}t''{\le}t'{\cdot}\ \ xt''{=}xt \wedge yt''{=}yt \wedge ...)$

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

The programming notations are defined as follows.

ok = $(t'{=}t)$
tick = $(t'{=}t{+}\delta \wedge wait)$
$(x{:=}\ e)$ = $(t'{=}t{+}\delta{+}\tau \wedge xt'{=}et \wedge wait_{y,z...})$

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

$(P;Q)$ = $\exists t''\cdot$ (substitute t'' for t' in P)
$\wedge$ (substitute t'' for t in Q)
$(P_\alpha\ \|\ Q_\beta)$ = $(P_\alpha\ \wedge\ (Q;\ wait)_\beta$
$\vee\ \ (P;\ wait)_\alpha\ \wedge\ Q_\beta)$
(**if** b **then** P **else** Q) = (if bt then P else Q)
= $(bt \wedge P\ \vee\ \neg bt \wedge Q)$

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

$(\textbf{while } b \textbf{ do } P)$
$\Rightarrow$ $\textbf{if } b \textbf{ then } (P;\ \textbf{while } b \textbf{ do } P) \textbf{ else ok}$
$(\forall x, x', y, y', ..., t, t' \cdot\ W \Rightarrow \textbf{if } b \textbf{ then } (P; W) \textbf{ else ok})$
$\Rightarrow$ $(\forall x, x', y, y', ..., t, t' \cdot\ W \Rightarrow \textbf{while } b \textbf{ do } P)$
$\textbf{var } z: T \cdot P \ = \ \exists z: time \rightarrow T \cdot P$
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.

$(x:= 2;\ x:= x+y;\ x:= x+y) \parallel (y:= 3;\ y:= x+y)$
$= (t' = t+1 \wedge xt'=2;\ t' = t+1 \wedge xt' = xt+yt;$
$t' = t+1 \wedge xt' = xt+yt)$
$\wedge (t' = t+1 \wedge yt'=3;\ t'=t+1 \wedge yt' = xt+yt;$
$t' \geq t \wedge \forall t'': t \leq t'' \leq t' \cdot\ yt''=yt)$
$\vee (t' = t+1 \wedge xt'=2;\ t' = t+1 \wedge xt' = xt+yt;$
$t' = t+1 \wedge xt' = xt+yt;$
$t' \geq t \wedge \forall t'': t \leq t'' \leq t' \cdot xt''=xt)$
$\wedge (t' = t+1 \wedge yt'=3;\ t'=t+1 \wedge yt' = xt+yt)$
$= t' = t+3 \wedge x(t+1)=2 \wedge x(t+2) = x(t+1)+y(t+1)$
$\wedge x(t+3) = x(t+2)+y(t+2)$
$\wedge t' \geq t+2 \wedge y(t+1)=3 \wedge y(t+2) = x(t+1)+y(t+1)$
$\wedge \forall t'': t+2 \leq t'' \leq t' \cdot\ yt''=y(t+2))$
$\vee t' \geq t+3 \wedge$ (other conjuncts)
$\wedge t' = t+2 \wedge$ (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 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
$s_P=(\delta \lhd (s \vee s'_P) \wedge b) \wedge s'=(\delta \lhd (s \vee s'_P) \wedge \neg b)$
$\sigma_P=\sigma \wedge R\sigma=R\sigma_P \wedge C\sigma=C\sigma_P$
$W\sigma=W\sigma_P \wedge D\sigma=D\sigma_P$
$\delta \geq (\ b$ 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.

$I \wedge \neg \lhd B \wedge \neg s \ \Rightarrow \ \neg s'$

which says that a circuit does not spontaneously generate s'.

$I \wedge \neg B \ \Rightarrow \ \neg R\sigma \wedge \neg C\sigma \wedge \neg W\sigma \wedge \neg D\sigma$

which says that if a circuit is not busy, its $R\sigma$, $C\sigma$, $W\sigma$, and $D\sigma$ outputs are all $\bot$.

6 Synchronous and Asynchronous

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. 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.

If we choose to make each assignment into a little procedure, the 1-2-merges at the calling points are an implementation of local handshaking. We can thus program local handshaking when we want it. In principle, local delays should 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.

7 Conclusion

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).

We have implemented 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.

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] M.Rem: *Partially Ordered Computations with Applications to VLSI Design*, Technical Report MR83/3, Eindhoven University of Technology, 1982

[12] J.L.A.van de Snepscheut: *Trace Theory and VLSI Design*, LNCS 200, Springer, 1985.

[13] D.E.Thomas, P.Moorby: *the Verilog Hardware Description Language*, Kluwer, 1991.

[14] 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.