# Compiling Joy Into Silicon: a Formally Verified Compiler for Delay-Insensitive Circuits


Sam Weber*
*Cornell University*
`samuel@cs.cornell.edu`

Bard Bloom†
*IBM Watson Research Laboratory*
`bard@watson.ibm.com`

Geoffrey Brown‡
*Cornell University*
`gbrown@ee.cornell.edu`


January 26, 1996


**Abstract**

Manually designing delay-insensitive electronic circuits has proven to be difficult in practice. As an alternative, we designed and implemented a compiler that automatically produces such circuits. The source language for the compiler is a language called "Joy", which is a simple but complete parallel language with a syntax similar to that of many procedural languages. The compiler's output is a netlist suitable for input into standard place-and-route tools.

In this paper, we present the highlights of the compilation algorithm, and the proof of correctness for it. This is among the first formally verified algorithms for compiling a general language into circuits.


## 1 Introduction

Most electronic circuits are built assuming that all components have access to synchronized clock signals. For example, an AND-gate in a synchronous circuit is used under the assumption that its inputs will change at most once per clock period (all delay considerations can be absorbed into the global assumption that the clock period is "long enough".) A delay-insensitive AND-gate cannot assume that its inputs will arrive at the same moment, and will wait however long is necessary for both of its inputs to arrive. Delay-insensitive circuits


*Supported by NSF grants CCR-9003441 and CCR-9058180.

†Supported by NSF grant CCR-9003441.

‡Supported by a NSF grant CCR-9058180

have a number of practical advantages. Unfortunately they have proven to be difficult to design manually because one has to ensure that no race conditions occur.

In order to overcome this problem, we designed and built a compiler that automatically generates correct delay-insensitive circuits. The input to the compiler is a program written in a special-purpose language called Joy.

There were a number of factors we considered in producing the compiler. Most importantly, we needed to be able to prove the correctness of the circuits that were produced. To be precise, any circuit generated by the compiler must not only be delay-insensitive, but also perform the same computation as the Joy program that was input. In addition, it was important that Joy programs be easily understandable. Since the original problem was that delay-insensitive circuits are difficult to design manually, it would be self-defeating if Joy programs were difficult to understand or required too much knowledge of circuit design.

For these reasons, Joy was designed with a syntax similar to many procedural programming languages. It is a pared-down but complete concurrent programming language, with basic conditional, looping, and parallel control constructs, Boolean variables, and synchronization along communication channels.

In this paper, we present our scheme for compiling Joy programs to circuits, and present the highlights of the proof that this scheme is correct. We also briefly discuss the implementation. We expect our methods to extend to a variety of other programming constructs: procedure and function calls, data-bearing channels, and so forth. In addition, the proof techiques are general, and should apply to a wide variety of similar problems.

## 1.1 Overview of Circuits

Our circuits are quite simple. A circuit consists of a set of primitive *components* connected by *wires* attached at *input* and *output* ports. A wire connects a single input port to a single output port. The unconnected ports of the constituent components of a circuit are the inputs and outputs of the circuit and connect to the outside environment. The components of a circuit communicate by initiating events on their output ports or by absorbing events on their input ports. One possible physical interpretation of an event is a voltage transition. For physical reasons, circuits cannot refuse input — even if that input may cause them to fail. Furthermore, input and output events on a wire must strictly alternate to prevent possible electrical interference. A wire connecting an output port to an input port is *active* if an event has been initiated on the output port but not yet absorbed on the input port.

We give a compilation scheme to an intermediate form. The modules of this intermediate form are easily implemented using low level asynchronous building blocks. [4, 22] Our set of primitive components is quite small, and most of them are straightforward to implement. Indeed, most components turn out to be patterns of wires. For related work on implementing delay-insensitive circuits see [2, 5, 17, 20, 6, 9]. The resulting circuits are amenable to a simple low-level mathematical model of wires and impulses, which we use to verify the compilation scheme. The full correctness proof is too large to be presented in full. We describe its mathematical foundations (the theory of bisimulation), and sketch the main lemmas describing how programs and circuits execute. More details are found in the appendix.

As the translation scheme is syntax-directed, we found it straightforward to implement with `lex` and `yacc`. Our Joy compiler produces netlists of basic components. The output of the compiler is suitable for input to any of the many place-and-route tools available for silicon compilation, given a VLSI cell-library of the basic components. The development of such a cell-library is fairly standard; see [2, 5, 7, 13]. As the implementation is routine compiler technology, we concentrate on the description of the compilation scheme and correctness proof.

# 2 Joy

In this section we present Joy. In the following BNF-style syntax description, $c$ is drawn from a set of channel identifiers *CID*, $\mathcal{C}$ is a set of channel identifiers, and $v$ is drawn from a set of variable identifiers *VID*.

$$
\begin{array}{lrcl}
 & c & \in & CID \\
 & v & \in & VID \\
 & & & \\
(expr) & E & ::= & v \mid \neg E \mid E_1 \wedge E_2 \mid E_1 \vee E_2 \mid \mathrm{tt} \mid \mathrm{ff} \\
(process) & P & ::= & \texttt{skip} \mid v := E \mid c! \mid P_1; P_2 \\
 & & \mid & \texttt{if}\, G\, \texttt{fi} \mid \texttt{do}\, G\, \texttt{od} \mid P_1 \parallel_{\mathcal{C}} P_2 \\
(guard) & G & ::= & E \rightarrow P \mid c? \rightarrow P \mid E \,\&\, c? \rightarrow P \mid G_1 \,[\!]\, G_2
\end{array}
$$

Informally, the process terms stand for the following processes:

**`skip`:** is a process which simply terminates.

$v := expr$**:** is a process which assigns *expr* to $v$.

$c!$ is a process which is ready to output along channel $c$. Send commands are blocking; that is, they will wait for the signal to be received before allowing the sender to continue. (Receives are done as part of guarded commands, and thus may be non-blocking.)

$P_1; P_2$**:** is the sequential composition of $P_1$ and $P_2$: do $P_1$, then do $P_2$.

**`if`** $G$ **`fi`:** attempts to execute the guarded command set $G$ until it succeeds. A guarded command set consists of one or more guarded commands separated by $[\!]$ and is executed by executing one enabled command from the set. If no command is enabled, execution fails, and the circuit will try $G$ again. A guarded command can be of the form $E \rightarrow P$, $c? \rightarrow P$, or $E\&c? \rightarrow P$. In the first case the command is enabled if $E$ is true. In the second case the command is enabled if some process is prepared to output on channel $c$, and, in the third case the command is enabled if $E$ is true and some process is prepared to output on $c$. The command is executed by inputting from channel $c$ (for the second and third cases) and then executing $P$.

**`do`** $G$ **`od`:** repeatedly executes the guarded command set $G$ until execution fails.

$P_1 \parallel_{\mathcal{C}} P_2$**:** is the concurrent composition of $P_1$ and $P_2$, communicating along the channels in $\mathcal{C}$.

Legal Joy programs must satisfy some further syntactic restrictions. If $P_1 \|_{\mathcal{C}} P_2$ appears in a Joy program, then

1. The input channels of $P_1$ and $P_2$ (except for those appearing in $\mathcal{C}$) must be disjoint.

2. Similarly, the output channels not appearing in $\mathcal{C}$ must be disjoint.

3. Finally, no variable may appear in both $P_1$ and $P_2$.

These restrictions are not essential — for example, it is straightforward to allow parallel reading of variables — but we impose them for simplicity.

In particular, the set of input channels used by two concurrently executing processes must be disjoint; similarly for output channels. Furthermore, concurrently active processes may not share any variables; all communication must be done by communication commands. A shared variable may, of course, be realized by a shared process.

# 3 Compiling Joy Programs

Our compiler does a simple syntax directed translation into primitive circuit components. Each Joy language contruct has a corresponding (usually trivial) circuit component, to make the actual compilation process easy.

For example, the circuit corresponding to the program

$$\mathrm{x} := \mathrm{tt};$$
$$\mathrm{x} := \mathrm{ff}$$

is shown in Figure 1. The purpose of the "WCALL" component will be explained later.

Notice that the circuit is not a tree, despite the fact that Joy programs are structured. This is because every reference to a variable or communication channel has to connect somehow to the component which holds the state of that entity. This fact is an obstacle in our proof, because it prevents us from doing structural induction.

We construct our components out of building blocks, and specify their behaviour using finite state automata. The implementations of our components can then be verified with respect to models of these building blocks using an automated tool such as that developed by Dill [8]. The next level of the proof, verifying the building blocks, is likely to be extremely difficult as any implementation of these is dependent upon analog transistor effects (e.g. transistor threshold voltages and gate capacitance).

In order to ensure that these components are used in a manner which ensures the requirements on input and output events are met, we use a simple communication convention between components. Wires are always used in pairs or triples. This convention is illustrated in Figure 2.

In the two-wire interface, every transition on the request wire `req` must be explicitly followed by a transition on the acknowledgement wire `ack`. There are two types of three wire interfaces. The read interface allows a binary value to be returned with an acknowledgment by allowing a transition on the request wire to be followed by a transition on either of the acknowledgement wires `ack0, ack1`. The write interface allows a value to be sent by

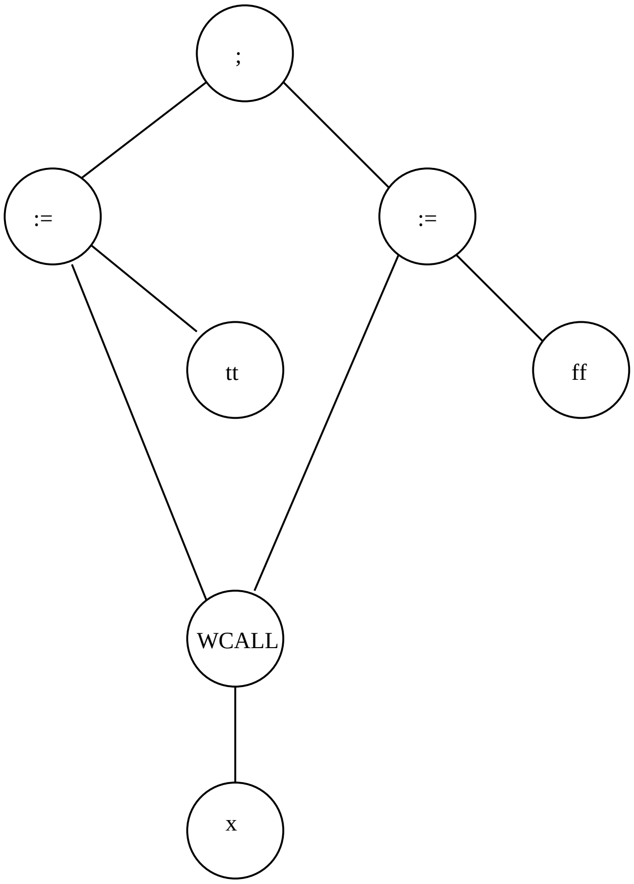


Figure 1: Example Circuit

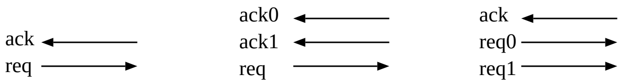


Figure 2: Communication Convention

allowing a transition on either of the request wires `req0, req1` followed by a transition of the `ack` wire. These are all that our compiler requires.

The idea of rigidly following this communication convention was developed by the Philips group [20] who call the resulting components "handshake circuits."

The sequence component, which implements the Joy semi-colon operator, is typical. We will describe it in detail as an illustrative example. A typical use of it is in the previous example, Figure 1. It first executes its left child, and when that has finished executes its right child.

In Figure 3 we present its wiring diagram and its finite-state automata specification. In the wiring diagram, input ports are pictured as arrows pointing into the component. Output ports are pictured as arrows pointing out. The automata has its initial state labeled `Init`. All legal state transitions are indicated by arcs in the specification, labeled by the input (e.g. `req`) or output event (overbarred, e.g. $\overline{\texttt{ack1}}$) associated with the state transition.

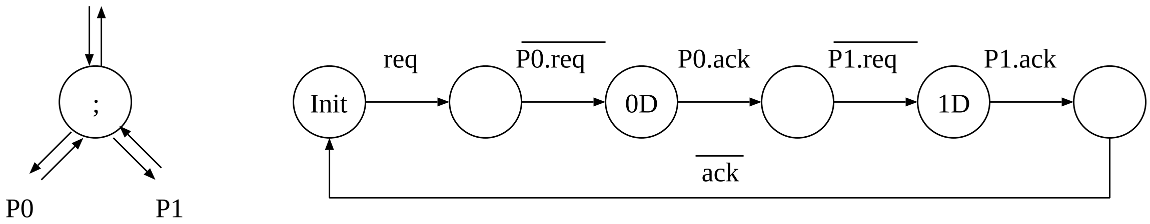


Figure 3: Sequential Composition

The binary operator $\wedge$ is a typical boolean expression component. This component is a short-circuiting **and**; it first evaluates the left expression and then, if necessary, the right expression. Figure 4 contains its wiring diagram and specification.

The component which implements boolean variables has two three-wire interfaces: a read interface for reading the variable, and a write interface for writing.

In practice more than one connection to each of these interfaces may be necessary, as in our previous example. However, our rules for circuits prohibit multiple connections to a port. We solve this problem by introducing a set of components for sharing a single interface – the call *CALL*, read-call *RCALL*, and write-call *WCALL* components.

These call components act like multiplexers. They have two input ports, and can receive signals on either of them. These signals are routed to the output port. When the component receives its acknowledgement from its output port, the signal is returned to whichever input port issued the request. Naturally, requests must be mutually exclusive, and the proof of correctness must demonstrate this. The three types of call components differ only in whether they have two or three wire interfaces.

Every call component contains state, since it has to remember which input report issued a request. Unlike the state of variable components, which corresponds to Joy's variable state,

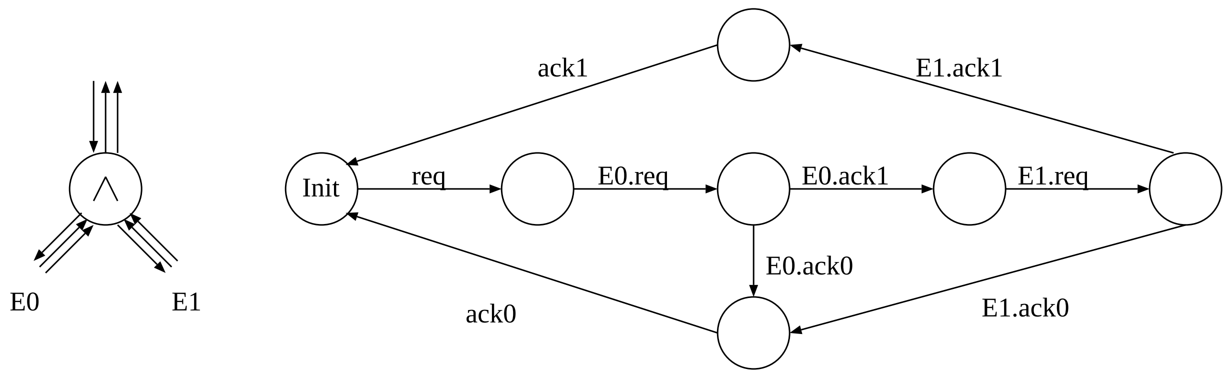


Figure 4: And component

the state of call components has no Joy equivalant. Thus, in our proof we must synthesize the call-box state from the Joy execution, and show that this state is always correct.

Communication channels are implemented with a two wire interface for the sender and a three-wire interface for the receiver. The sender requests communication on its request wire and waits for completion of the communication on its acknowledge wire. The receiver may test the sender's readiness to communicate by initiating an event on its request wire. The receiver receives a "true" acknowledgement if communication is completed; otherwise it receives a "false" acknowledgement.

One peculiarity of our specification is that the sender's request may be ignored. This is necessary for the specification to be delay-insensitive – a delay-insensitive circuit which has its inputs buffered (hence delaying the "arrival" of an input event) must be indistinguishable from the unbuffered circuit. The fact that our channel component is delay-insensitive has an effect on our operational semantics for Joy. In particular, in evaluating a receive guard, the guard may fail even if there is a sender active on the same channel.

# 4 Proof of Correctness

We verify our compilation scheme by a formal proof of correctness. We give a fairly abstract operational semantics of Joy, and a fairly concrete operational semantics of the circuits that the compilation scheme produces. We then show that the two are equivalent. More precisely, we show that there is a precise correspondence between the possible actions of a program and the possible gate-firings of the circuit; as discussed below, we establish a *bisimulation* (one of the strongest forms of process equivalence, and hence a strong guarantee of correctness) between the two.

The semantics for Joy may appear unrealistically optimistic. For example, expressions are evaluated instantaneously and atomically. This is not true in the compiled circuits; a complex expression requires many events to evaluate. This disparity is intentional. The semantics of Joy are designed to give a reasonable and useful model for programming. The compiler is charged with ensuring that the simple semantics of Joy correspond with what actually happens when the circuit is run – and the main theorem of this paper is that the two disparate kinds of semantics agree, and hence that the circuit behaves as it ought to.

## 4.1 Semantics of Joy

We sketch a structured operational semantics (SOS) for Joy, along the lines of [15]. This lets us conceptually execute Joy programs without compiling them; our main theorem will be that they execute the same before and after compilation.

The state of an executing Joy program consists of a *control state*, telling which parts of the program are executing — the program counter(s) — and a *variable state* telling what values variables have. It is convenient (and standard in the SOS methodology) to represent the control state of the circuit as a Joy program $P$, which is the code which remains to be executed. For example, if the original program is `x:=tt; y:=ff`, after executing the first command $P$ will be roughly `y:=ff`. We extend the language slightly, to include a "command" **done** which represents a finished program. The variable state is simply a function $\eta$ from variable names to their (Boolean) values. So, we represent the state of the circuit as a pairs, written $P \mathbin{\dagger} \eta$.

The ultimate purpose of a program is to perform *actions*: receives `c`? and sends `c`! on channels `c`, and a special action "-" for internal computation."-" is $\tau$ in [12], and does not appear explicitly in most models of CSP. Processes will produce "-" actions when the results of their action are not visible outside the circuit; for example, **skip** statements and internal communications simply produce "-" actions.

The main operational semantic relation is $P \mathbin{\dagger} \eta \xrightarrow{\alpha} P' \mathbin{\dagger} \eta'$, which is intended to mean that a process in state $P \mathbin{\dagger} \eta$ can perform the action $\alpha$ and thereafter behave like $P' \mathbin{\dagger} \eta'$. For example, the rather silly program $(\mathtt{x} := \mathrm{tt}\,;\, \mathtt{x} := \mathrm{ff})$ started in state $\{x = \mathrm{ff}\}$ would evolve as follows:

$$
\begin{array}{c}
((\mathtt{x} := \mathrm{tt}\,;\, \mathtt{x} := \mathrm{ff}) \mathbin{\dagger} \{x = \mathrm{ff}\}) \\
\xrightarrow{-} \\
((\mathbf{done}\,;\, \mathtt{x} := \mathrm{ff}) \mathbin{\dagger} \{x = \mathrm{tt}\}) \\
\xrightarrow{-} \\
(\mathtt{x} := \mathrm{ff} \mathbin{\dagger} \{x = \mathrm{tt}\}) \\
\xrightarrow{-} \\
(\mathbf{done} \mathbin{\dagger} \{x = \mathrm{ff}\})
\end{array}
$$

There are many cases in which we know that the variable state does not change over a transition, and one place where we need to know that a transition does not induce a state change. We thus have an auxiliary judgement $P \xrightarrow{\alpha} P'$, and the rule

$$
\frac{P \xrightarrow{\alpha} P'}{P \mathbin{\dagger} \eta \xrightarrow{\alpha} P' \mathbin{\dagger} \eta}
$$

In the operational semantics, the program is gradually consumed as it executes. This is a standard technical convenience for the operational semantics – it allows us to avoid explicit program counters, or more accurately to use the program term itself as the program counters. It is not intended to suggest that pieces of the circuit will flake off as it executes!

Processes without subprocesses are given behavior by axioms. For example, **skip** quietly evolves to a finished process in any state, giving the rule:

$$\textbf{skip} \stackrel{-}{\to} \textbf{done} \tag{1}$$

Assignments behave in the obvious way:

$$(x := E) \dagger \eta \stackrel{-}{\to} \textbf{done} \dagger \eta' \tag{2}$$

where $\eta'$ is the same as $\eta$ except that $\eta'(x)$ is the value of $E$ in state $\eta$.

Sends, of course, send messages on their channel and then halt:

$$\mathsf{c}! \stackrel{\mathsf{c}!}{\to} \textbf{done} \tag{3}$$

Most composite processes are given behavior based on the behavior of their components. For example, a sequential composition executes the first process:

$$\frac{P_1 \dagger \eta \stackrel{\alpha}{\to} P_1' \dagger \eta'}{(P_1 \,;\, P_2) \dagger \eta \stackrel{\alpha}{\to} (P_1' \,;\, P_2) \dagger \eta'} \tag{4}$$

and when the first process finishes, the second one starts:

$$\textbf{done} \,;\, P_2 \stackrel{-}{\to} P_2 \tag{5}$$

Parallel composition is more complex. Either component may execute, or the two may coordinate on a send/receive pair. Note that in the communication rule we use transitions $P \stackrel{\alpha}{\to} P'$ which cannot involve variable state change; we therefore do not need to worry about the problem of merging two differently-changed variable states.

$$\frac{P_1 \dagger \eta \stackrel{\alpha}{\to} P_1' \dagger \eta', \alpha \notin \mathcal{C}}{\begin{array}{c}(P_1 \parallel_{\mathcal{C}} P_2) \dagger \eta \stackrel{\alpha}{\to} (P_1' \parallel_{\mathcal{C}} P_2) \dagger \eta' \\ (P_2 \parallel_{\mathcal{C}} P_1) \dagger \eta \stackrel{\alpha}{\to} (P_2 \parallel_{\mathcal{C}} P_1') \dagger \eta'\end{array}} \qquad \frac{P_1 \stackrel{\mathsf{c}?}{\to} P_1', \quad P_2 \stackrel{\mathsf{c}!}{\to} P_2', \quad \mathsf{c} \in \mathcal{C}}{\begin{array}{c}P_1 \parallel_{\mathcal{C}} P_2 \stackrel{-}{\to} P_1' \parallel_{\mathcal{C}} P_2' \\ P_2 \parallel_{\mathcal{C}} P_1 \stackrel{-}{\to} P_2' \parallel_{\mathcal{C}} P_1'\end{array}} \tag{6}$$

A parallel composition finishes when both branches do:

$$(\textbf{done} \parallel_{\mathcal{C}} \textbf{done}) \stackrel{-}{\to} \textbf{done}. \tag{7}$$

**If** and **do** statements require the evaluation of guarded commands. The semantics include a predicate $GP$ **fails**. Basic guarded commands $(E\&\alpha \to P)$ evaluate $E$. If it is false, the guarded command fails. If it is true, then (nondeterministically) either the communication happens and $P$ is executed or the guarded command fails. This rather peculiar behavior is required by delay-insensitivity as described in section 3.

$$\frac{\eta(E) \Rightarrow \mathtt{tt}}{\begin{array}{c}(E\&\alpha \to P) \dagger \eta \stackrel{\alpha}{\to} P \\ (E\&\alpha \to P)\ \textbf{fails}\end{array}} \qquad \frac{\eta(E) \Rightarrow \mathtt{ff}}{(E\&\alpha \to P)\ \textbf{fails}} \tag{8}$$

Executing a list of guarded commands $G_1 \| G_2 \| \cdots \| G_n$ tries $G_1$, then $G_2$, and so on; if one succeeds, then the list succeeds; if all fail, then the list fails.

$$\frac{GP_1 \mathbin{+\!\!\!\!;} \eta \xrightarrow{\alpha} P}{(GP_1 \| GP_2) \mathbin{+\!\!\!\!;} \eta \xrightarrow{\alpha} P} \tag{9}$$

$$\frac{\begin{array}{c} GP_1 \mathbin{+\!\!\!\!;} \eta \ \textbf{fails} \\ GP_2 \mathbin{+\!\!\!\!;} \eta \xrightarrow{-} P \end{array}}{(GP_1 \| GP_2) \mathbin{+\!\!\!\!;} \eta \xrightarrow{-} P} \qquad \frac{\begin{array}{c} GP_1 \mathbin{+\!\!\!\!;} \eta \ \textbf{fails} \\ GP_2 \mathbin{+\!\!\!\!;} \eta \ \textbf{fails} \end{array}}{(GP_1 \| GP_2) \mathbin{+\!\!\!\!;} \eta \ \textbf{fails}} \tag{10}$$

Finally, **if** and **do** do the natural things:

$$\frac{GP \mathbin{+\!\!\!\!;} \eta \xrightarrow{\alpha} P}{\textbf{if}\ GP\ \textbf{fi} \mathbin{+\!\!\!\!;} \eta \xrightarrow{\alpha} P \mathbin{+\!\!\!\!;} \eta} \tag{11}$$

There is no rule for the case where $GP$ **fails**; when this happens, the **if** blocks and cannot fire.

A **do**-loop finishes when its body is finished, and otherwise executes its body and then runs itself again.

$$\frac{GP \mathbin{+\!\!\!\!;} \eta \ \textbf{fails}}{\textbf{do}\ GP\ \textbf{od} \xrightarrow{-} \textbf{done} \mathbin{+\!\!\!\!;} \eta} \qquad \frac{GP \mathbin{+\!\!\!\!;} \eta \xrightarrow{\alpha} P}{\textbf{do}\ GP\ \textbf{od} \xrightarrow{\alpha} (P \,;\, \textbf{do}\ GP\ \textbf{od}) \mathbin{+\!\!\!\!;} \eta} \tag{12}$$

The rules for evaluating expressions are fairly straightforward. The basic notion here is that expression $E$ in variable state $\eta$ yields value $b$; $\eta(E) \Rightarrow b$, where $b$ is either $\mathrm{tt}$ or $\mathrm{ff}$. The rules are:

$$\begin{array}{c} \eta(\mathrm{tt}) \Rightarrow \mathrm{tt} \\ \eta(\mathrm{ff}) \Rightarrow \mathrm{ff} \end{array} \tag{13}$$

$$\frac{\eta(v) = b}{\eta(v) \Rightarrow b} \tag{14}$$

$$\frac{\eta(E_1) \Rightarrow b_1 \text{ and } \eta(E_2) \Rightarrow b_2}{\begin{array}{c} \eta(E_1 \wedge E_2) \Rightarrow b_1 \wedge b_2 \\ \eta(E_1 \vee E_2) \Rightarrow b_1 \vee b_2 \\ \eta(\neg E_1) \Rightarrow \neg b_1 \end{array}} \tag{15}$$

## 4.2 Semantics of Circuits

We model delay-insensitive circuits as directed graphs. Processing components are nodes; wires are edges. Each node has a set of *port names*, generally including distinguished *request* and *acknowledge* ports. Each wire has a name at each end, *e.g.*, a wire from $l_1$'s $\overline{r_1}$ port to $l_2$'s $r_2$ would be called $l_1.\overline{r_1}$ and $l_2.r2$. The state of the circuit has two components: a set of wires $W$ which currently have signals on them, and a function $\kappa$ giving the states of state-bearing components.

We define the operational semantics of circuits in a quite obvious way. To determine the behavior of $W \dagger \kappa$, we look for some patterns of active wires on the inputs to a processing component. If one of the patterns is found, then the processing component can fire, making other wires active. For example, we have a sequencing component named $l$ with six ports, $\{r, r1, a1, r2, a2, a\}$, corresponding to the ";" connective in Joy programs. If its request wire ($l.r$, leading to port $r$) is active, then $l$ can fire and send a signal on its $r1$ port: this is the pair of transitions from Init to 0D in Figure 3. It then waits for an impulse on $a1$, which is the acknowledgement that the first subterm has finished. It then sends an impulse on $r2$, awaits a signal on $a2$, and finally sends its acknowledgement on $a$. These behaviors correspond to the transition graphs given in Section 3.

Communication within the circuit is straightforward. Communication with the outside environment is somewhat more confusing; we must keep track of what the circuit could do if the environment would cooperate, assuming that the environment obeys the basic conventions of communication.

The formalization of the circuit as a graph is straightforward but tedious.

### 4.3 Bisimulation

Concurrent entities are often described in terms of *labeled transition systems* (lts's): tuples $L = \left\langle \Sigma, A, \dot{\rightarrow}, \sigma_0 \right\rangle$ where $\Sigma$ is a set of states, $A$ a set of actions, $\dot{\rightarrow}$ is a family of transition relations $\xrightarrow{\alpha} \subseteq \Sigma \times \Sigma$ for each $\alpha \in A$, and $\sigma_0 \in \Sigma$ is the initial state. In our case, Joy programs and circuits (plus states) determine lts's, with $\Sigma$ given by the set of $(P \dagger \eta)$'s or $(W \dagger \kappa)$'s.

Two labeled transition systems $L$ and $L'$ are *strongly bisimilar* iff there is a relation $\sim$ between their states such that their initial states are related and if $\sigma \sim \tau$, then any action which one of the two can take, the other can also take and the resulting states are still related. That is, if $\sigma \xrightarrow{\alpha} \sigma'$, then there is a $\tau'$ such that $\tau \xrightarrow{\alpha} \tau'$ and $\sigma' \sim \tau'$; and vice versa. This implies that (but is much stronger than) $\sigma$ and $\tau$ have the same set of traces. Strong bisimulation is too detailed a notion for most purposes, as it insists that the processes have the same silent moves as well as visible ones. There is a variant called *weak bisimulation*, which allows processes to take internal steps (- moves); see [12].

Bisimulation, strong or weak, is an extremely strong form of correctness. It has been shown [14, 11, 3] that bisimilar processes are indistinguishable in any reasonable setting, and in particular in Joy. Indeed, bisimulation is too strong to be a requirement for a specification of correctness, as in most settings there are non-bisimilar processes which are still indistinguishable. However, we are able to achieve bisimulation, and hence all weaker forms of correctness.

### 4.4 Highlights of the equivalence proof

We are now able to state the equivalence between Joy programs and their compilations. Let $P_0$ be any Joy program, and $\eta_0$ an initial state of its variables. Let $C$ be the circuit into which $P_0$ is compiled, $W_0$ be its initial set of active wires (an impulse on the external request line and nothing else active), and $\kappa_0$ the initial state of variables corresponding to $\eta_0$. Then the lts's with initial state $P_0 \dagger \eta_0$ and $W_0 \dagger \kappa_0$ are bisimilar.

The major technical difficulty in the proof of correctness is that Joy programs and circuits execute in quite different ways. The operational semantics of Joy were designed to be understandable, to demonstrate that Joy is a reasonable programming language. We have tried to minimize the amount of unnecessary information kept by Joy programs. Consequently, as a Joy program executes, it loses information about the original program.

Conversely, circuits are physical entities, and our model of them is intended to be reasonably realistic. In particular, the wires and gates of a circuit continue to exist throughout and after the circuit's execution. This disparity between the two languages makes proving a correspondence directly rather challenging.

For example, consider the Joy program **skip** ; **skip**. This does nothing in three transitions:

$$\mathbf{skip} \mathbin{;} \mathbf{skip} \xrightarrow{-} \mathbf{done} \mathbin{;} \mathbf{skip} \xrightarrow{-} \mathbf{skip} \xrightarrow{-} \mathbf{done} \tag{16}$$

However, as we have defined our circuits, the circuit does nothing in eight transitions:

1. The request line of **skip** ; **skip**
2. The request line of the first **skip**.
3. The first **skip** itself.
4. The acknowledge line of the first **skip**.
5. The request line of the second **skip**.
6. The second **skip** itself.
7. The acknowledge line of the second **skip**.
8. The acknowledge line of **skip** ; **skip**

These transitions are illustrated in Figure 5.

### 4.4.1 Joy Prime

To connect the two disparate models, we introduce an intermediate language Joy Prime which has aspects of both. Joy Prime programs resemble Joy programs augmented with additional labels and markers which keep track of some information that circuits retain and Joy programs lose. For example, the Joy Prime execution of the program corresponding to

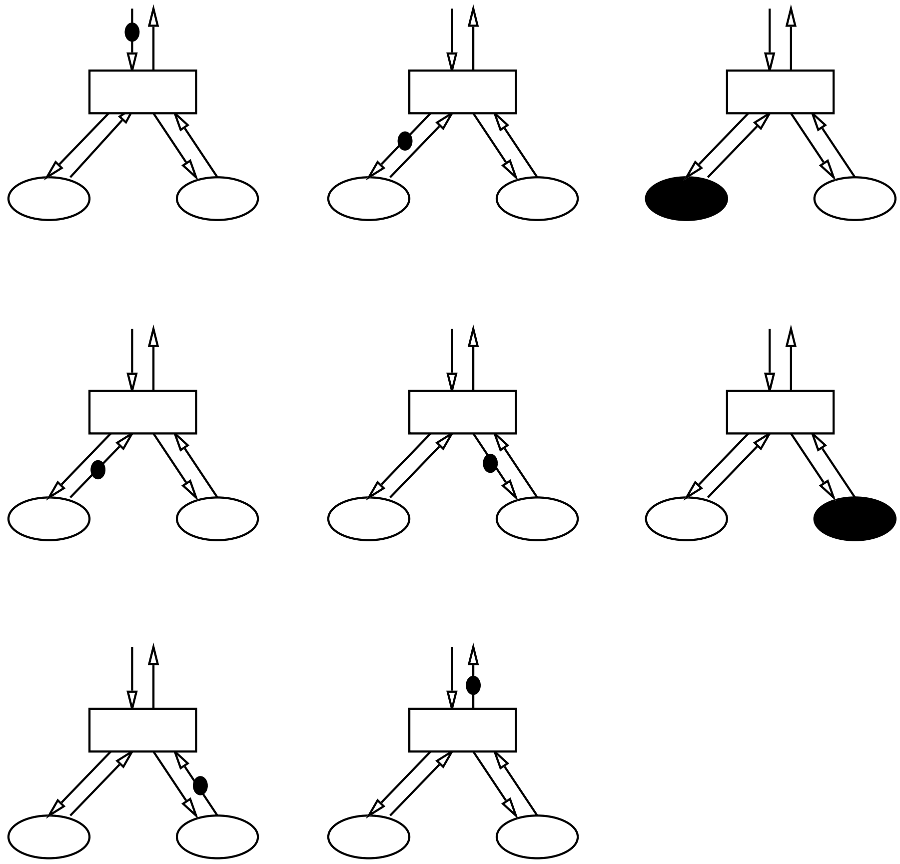

Figure 5: **skip** ; **skip** circuit execution

**skip** ; **skip** is:

$$
\begin{array}{ll}
1. & \left(\mathtt{skip}^{v2\mathtt{skip}}; \mathtt{skip}^{v3\mathtt{skip}}\right)^{v1;} \\
 & \quad\downarrow - \\
2. & \left(\mathtt{skip}^{v2\mathtt{skip}}; \mathtt{skip}^{v3\mathtt{skip}}\right)^{1;} \\
 & \quad\downarrow - \\
3. & \left(\mathtt{skip}^{2\mathtt{skip}}; \mathtt{skip}^{v3\mathtt{skip}}\right)^{1;} \\
 & \quad\downarrow - \\
4. & \left(\mathtt{done}^{2\mathtt{skip}}; \mathtt{skip}^{v3\mathtt{skip}}\right)^{1;} \\
 & \quad\downarrow - \\
5. & \left(\mathtt{skip}^{v3\mathtt{skip}}\right)^{1;} \\
 & \quad\downarrow - \\
6. & \left(\mathtt{skip}^{3\mathtt{skip}}\right)^{1;} \\
 & \quad\downarrow - \\
7. & \left(\mathtt{done}^{3\mathtt{skip}}\right)^{1;} \\
 & \quad\downarrow - \\
8. & (\mathtt{done})^{1;}
\end{array}
$$

where the numbers in the first column correspond to the circuit transitions.

The similarity with Joy makes it straightforward to relate Joy and Joy Prime programs. However, Joy Prime programs execute in the same way as the corresponding circuits, making it possible (though not easy) to prove that the two are equivalent. We illustrate this construction for sequencing.

Each instance of each operation in Joy Prime is given a label, consisting of a unique identifier $l$ and the operator symbol; for $P_1 ; P_2$, it will have the form "$l;$". This label will remain with the statement as long as it is active. We will have to identify *virgin programs*; that is, statements which have not been executed. We use a marker $v$. The Joy Prime statement corresponding to $P_1; P_2$ will be of the form $(P_1'; P_2')^{vl;}$ This corresponds to being in state Init of Figure 3.

To start executing this, we use the Joy Prime rule

$$P^{vl*} \xrightarrow{-} P^{l*}$$

where $*$ is any operator; in our case, this gives us the transition $(P_1 ; P_2)^{vl;} \xrightarrow{-} (P_1 ; P_2)^{l;}$. This corresponds to a component processing a signal on its request line; in this case, to the transitions from Init to 0D in the figure.

The Joy Prime rule for executing a sequence is identical to the Joy rule (4), except that it includes a label (which does not change):

$$\frac{P_1 \dagger \eta \xrightarrow{\alpha} P_1' \dagger \eta'}{(P_1 ; P_2)^{l;} \dagger \eta \xrightarrow{\alpha} (P_1' ; P_2)^{l;} \dagger \eta'}$$

When $P_1$ finishes executing, we use the rule corresponding to the Joy rule (5) and to the 0D transitions between states 0D and 1D on page 6: when the first child finishes, the ";"

component starts the second.

$$(\mathbf{done}\,;P_2)^{l;} \xrightarrow{-} (P_2)^{l;} \tag{17}$$

Notice that the label of $P_2$ says that $P_2$ is part of a sequencing statement, even though the sequencing operator is gone. We use this to remember the ; module. $P_2$ will execute ordinarily, but the "$l;$" label will remain:

$$\frac{P_2 \xrightarrow{\alpha} P_2'}{(P_2)^{l*} \xrightarrow{\alpha} (P_2')^{l*}}$$

When $P_2$ finishes, it will be in the state $\left(\mathbf{done}^{l'*}\right)^{l;}$ where "$l'*$" is the label of the main operator of $P_2$. The Joy Prime rule

$$\left(\mathbf{done}^{l'*}\right)^{l\star} \xrightarrow{-} \mathbf{done}^{l\star} \tag{18}$$

applies. This corresponds to the 1D to Init transitions on page 6: after the second child acknowledges that it is finished, the ";" component acknowledges as well.

### 4.4.2 Toward the Proof of Equivalence

Once we have developed Joy Prime, it is possible to prove the main result: that the compilation scheme is correct. Specifically,

**Theorem 4.1** *Let $P$ be a Joy program, $P'$ the corresponding Joy Prime program, and $C$ the corresponding circuit. Then:*

1. *$P'$ behaves like $P$, except that it takes more silent moves. Formally $P$ and $P'$ are weakly bisimilar.*

2. *$P'$ behaves just like $C$. Formally, $P'$ and $C$ are strongly bisimilar.*

*As bisimulation is an equivalence relation, $P$ and $C$ are thus (weakly) bisimilar.*

The first part of Prop. 4.1 is straightforward, as the rules for $P'$ are essentially the same as those of $P$ with a good deal of extra baggage added. The second part of Prop. 4.1 is proved by a detailed analysis of the behavior of programs and circuits, finishing with the precise correspondence demanded for strong bisimulation.

The proof of the second part starts with the *label sanity lemma*, which states that the structure of a program is not scrambled as it executes: e.g., if a term labelled $l$ is a component of one labeled $l'$, then if $l$ is still present after executing, then $l'$ is as well and $l'$ is still a component of $l$.

The compilation scheme is a function $\mathsf{compile}(V)$ from programs to circuits. We convert $V$ to a Joy Prime program $V'$ (by adding labels). Consider any possible state $P' \ddagger \eta'$ which could arise in the execution of $V'$ in some initial state $\eta_0'$. We define *wires*$(P' \ddagger \eta')$, a state of $\mathsf{compile}(V)$, which we hope corresponds to the execution of $V' \ddagger \eta_0'$ which gave $P' \ddagger \eta'$. So, the proof reduces to (1) defining *wires*$(\cdot)$, and (2) showing that it indeed maps executions to executions.

Defining *wires*$(\cdot)$ requires some auxiliary definitions. The hard part of the definition is to figure out which wires should have signals on them. However, wires which have signals correspond to *active* commands in the program, which, by no coincidence, are those which either are just ready to take a step, or have just finished taking one.

### 4.4.3 Active Components

Thus, we define $act(P)$, the *active components* of $P$, by induction. Indeed, for many cases, $act(P) = \{P\}$ — the part of the program which takes a step is $P$ itself. We call such $P$'s *immediately active*. For example, c! and **done** ; $P_2$ are immediately active; the entire term is involved in the state transition.

The inductive cases involve composite statements. For example, when one of $P_1$ and $P_2$ is not **done**, then any activity in $P_1 \parallel_C P_2$ must take place within $P_1$ and $P_2$.

$$act(P_1 \parallel_C P_2) = act(P_1) \cup act(P_2) \text{ if } P_1 \neq \mathbf{done} \vee P_2 \neq \mathbf{done} \tag{19}$$

If both are done, the Joy Prime equivalent of (7) applies, so $P$ is immediately active, and we have

$$act(P) = \{P\} \text{ if } P = \left(\mathbf{done}^{l_1 * _1} \parallel_C \mathbf{done}^{l_2 * _2}\right)^{l_3 * _3}. \tag{20}$$

We next prove two lemmas which show that the definition of $act(P)$ is correct; they are also the main tools we use to prove bisimulation. Together, they show that the subterms we have identified as being active are in fact the ones where activity occurs. The Activity Lemma states that, if a whole program can execute, then some active component enables it to do so; the Converse Activity Lemma states that if an active component could execute in isolation, then it can still do so in the whole program. Each lemma has a second case, in which two components act simultaneously by communicating along a channel.

**Lemma 4.2 (Activity Lemma)** *If $P \fatsemi \eta \xrightarrow{\alpha} P' \fatsemi \eta'$, then either:*

1. *There is some term $Q \in act(P)$ such that $Q \fatsemi \eta \xrightarrow{\alpha} Q' \fatsemi \eta'$ and $P'$ is $P$ with $Q$ replaced by $Q'$.*[1]

2. *There are two terms $Q_1, Q_2 \in act(P)$ which communicate in the step giving $P'$.*

   - $Q_i \fatsemi \eta \xrightarrow{\beta_i} Q'_i \fatsemi \eta$ *for* $i = 1, 2$;
   - $\beta_1$ *is a send along some channel* $c$ *and* $\beta_2$ *is a receive from the same channel;*
   - $Q_1$ *and* $Q_2$ *occur in parallel in* $P$;
   - $P'$ *is* $P$ *with* $Q_1$, $Q_2$ *replaced by* $Q'_1$, $Q'_2$ *respectively;*

[1] In general in algebra, substituting terms for terms is somewhat dangerous; when one turns c! into **done** in the program c!; c!, one might intend the result to be either **done**; c!, c!; **done**, or **done**; **done**. This is not a problem in Joy Prime. Each subprogram of a Joy Prime program has a unique label, and so substitution of terms for terms is meaningful.

**Lemma 4.3 (Converse Activity Lemma)** *Suppose that $Q \in act(P)$, and $Q \dagger \eta \xrightarrow{\alpha} Q' \dagger \eta'$, where $\alpha$ is not a communication along a channel hidden by $P$. Let $P'$ be $P$ with $Q$ replaced by $Q'$. Then $P \dagger \eta \xrightarrow{\alpha} P' \dagger \eta'$.*

*Similarly, suppose $Q_1, Q_2 \in act(P)$, and $Q_1 \dagger \eta \xrightarrow{\beta_i} Q'_i \dagger \eta$ where $\beta_1$ and $\beta_2$ are send and receive along some internal channel* c *of $P$. Let $P'$ be $P$ with $Q_1$,$Q_2$ replaced by $Q'_1$, $Q'_2$. Then $P \dagger \eta \xrightarrow{-} P' \dagger \eta$.*

### 4.4.4 Activated Wires

For any immediately active term $R$, we define the set of *wires the term makes active*, *activates*$(R)$. This is one of the longer definitions of the proof, as there are three dozen kinds of wires in the circuit, and somehow each of them must be made active; hence the definition has three dozen cases. A typical case is *activates*$(Q^{vl*}) = \{l*.r\}$; that is, when a program is ready to be executed but its execution has not started, its request line is active.

Finally, we are in a position to define *wires*$(P)$ for all $P$: the wires that should correspond to the circuit being in state $P$ are simply the wires corresponding to each immediately active subterm:

$$wires(P) = \bigcup_{R \in act(P)} activates(R) \tag{21}$$

It is also necessary to calculate the expected states of call-boxes, which can be done by a similar but simpler syntactic analysis of terms. From this definition, we can also show the *Communication Sanity Lemma*, stating that only one process is trying to use any call-box at any given time.

### 4.4.5 Proof of Bisimulation

We are now ready to apply the bisimulation methodology. We define $P \dagger \eta \sim W \dagger \kappa$ if $W = wires(P')$ and $\eta$ and $\kappa$ give the same values to all state-bearing components. We must show that $\sim$ is a bisimulation relation; that is,

1. If $P \dagger \eta \xrightarrow{\alpha} P' \dagger \eta'$, then $W \dagger \kappa \xrightarrow{\alpha} W' \dagger \kappa'$ where $P' \dagger \eta' \sim W' \dagger \kappa'$, and
2. if $W \dagger \kappa \xrightarrow{\alpha} W' \dagger \kappa'$, then $P \dagger \eta \xrightarrow{\alpha} P' \dagger \eta'$ where $P' \dagger \eta' \sim W' \dagger \kappa'$

The labels and markers of Joy Prime were painstakingly chosen to give this exact correspondence between circuit and Joy Prime execution.

We need one Joy Prime rule for each input to each kind of component: some fifty rules. This may seem excessive for as simple a language as Joy. However, it is probably impossible to avoid. There are about fifty kinds of wires ands components. A proof that a circuit is correct which does not at some point validate the behavior of each wire and component is too abstract at best, and unlikely to be fully satisfactory. Most of the subtlety of the proof of Theorem 4.1 is in the design of Joy Prime (which corresponds roughly to an induction hypothesis), and in careful choice of supporting lemmas such as the activity and converse activity lemmas. After much development of this mathematical machinery, the body of the proof consists of some eighty straightforward cases. This establishes Theorem 4.1.

# 5 Related Work

Previous efforts at verifying compilers for VLSI have been restricted to weaker languages than Joy. An early work by Milne verified the correctness of a compiler for NOR expressions [10]. A more recent work described a compiler for translating path expressions into self-timed circuits [19]. The target architecture in [19] was a synchronizer consisting of an arbiter circuit and a sequencer for each of the component path expressions. The resultant circuits are not free from timing assumptions; however, the construction for path recognizers (a component part of each sequencer) is very close in spirit to the algorithm used by our compiler to generate delay-insensitive circuits.

A related approach was taken by Straunstrup and Greenstreet. [18] They present a method for describing the behavior of a circuit in terms of a set of transition rules and define a set of "implementation conditions" which a circuit must meet in order to be delay-insensitive. It is the designer's obligation to prove, for each circuit, that these conditions are met. Our compiler guarantees that all circuits which it generates are delay-insensitive.

Martin's work, like that of Straunstrup and Greenstreet, provides a methodology for verifying delay-insensitive circuits with respect to an idealized model of hardware behavior [9]. Although Martin has developed a compiler from a process algebra to delay-insensitive circuits, he has not published a proof that his compilation algorithm is correct [6].

Van Berkel's thesis [21] verifies a quite similar compilation technique. His language Tangram is somewhat weaker than Joy. In particular, it does not seem to allow selective communication: a Tangram process cannot wait for a signal from either of two other processes.

Furthermore, the theorem that he gives is somewhat weaker in several ways. We show weak bisimulation, one of the more detailed notions of process equivalence; van Berkel shows only trace equivalence, which is one of the least detailed notions. Essentially, we show that our circuits and programs behave identically, to the point of making the same decisions at the same times; van Berkel's merely produce the same output in response to the same input. This difference is insignificant in the current setting, though it might be significant in a more powerful language; van Berkel's proof will not show his circuits correct with respect to the restriction and parallel composition operations of CCS and ACP.

The van Berkel meanings of programs are given as trace sets, which is a reasonable high-level semantics. The meaning of a circuit is the composition of the meanings of its primitive components as trace sets. Though this description is evidently correct, it is fairly high-level. The primitive components are quite reasonably described as simple trace structures. However, the parallel composition operation is far from trivial, requiring many pages of mathematics to describe. While we do not dispute that van Berkel's definition accurately models parallel composition, we prefer to use a simpler definition inspired by the physical behavior of the circuit: a model of signals moving around on a graph of components.

The weaker theorem we regard as a minor matter, as (1) van Berkel's theorem is strong enough to show that Tangram programs behave correctly, and (2) we suspect that the Tangram compiler is actually correct up to bisimulation, for essentially the same reason that ours is; indeed, it should be routine to adapt our proof to cover Tangram. The difference in language is more significant, as selective communication is essential in many communication circuits.

Various researchers have given vaguely similar proofs of various systems; *e.g.*, [1] show a operating system kernel for a theoretical stack machine correct using a sequence of operational semantics and simulation relations.

Smith et al [16] give a silicon compilation scheme from a language C-CSP. Their ground rules are somewhat different from ours; *e.g.*, they assume gates may ignore spikes, which ours do not. They describe their compilation scheme as a sequence of translations, describing the intermediate stages and compiled code as (very simple) code in their high level language. They verify each step of translation individually. Their techniques are independent of ours, as their proof is organized orthogonally.

# 6 Conclusion

We have presented Joy, a simple yet complete programming language for circuits. We have described the implementation of Joy programs in silicon, and stated and sketched the proof that the compilation scheme is correct.

Many extensions are desirable before this will be widely useful. Joy is a fairly sparse language. There are a number of obvious extensions: variables of types other than Boolean, procedures and functions, value-bearing communication channels, modules, and so forth. We have compilation techniques for many of these extensions, and expect our correctness proof to extend straightforwardly to cover them.

# Appendix: Joy Prime, and the Formal Compiler

# A Formalizing the Compiler

We restrict our discussion to a subset of the language by eliding important features such as expressions involving variables and assignments to variables; however, we present out formalism in a manner which is consistent with a proof of the full language. Thus, our formalism includes features for modelling variables, while the proofs as presented make little use of these features.

In order to verify the compiler it was necessary to formally define both the compilation algorithm and the semantics of the target circuits.

## A.1 Well-Formed Programs

As stated before, we place certain restrictions on the use of channels and variables in Joy programs. For convenience, we make channel names unique throughout the program – that is, the process $\left(\mathsf{c}! \parallel_{\{\mathsf{c}\}} P_1\right) ; \left(\mathsf{c}! \parallel_{\{\mathsf{c}\}} P_2\right)$ is illegal, because the channel $\{\mathsf{c}\}$ is being used in two different components. Note that the similar program $\left(\mathsf{c1}! \parallel_{\{\mathsf{c1}\}} P_1\right) ; \left(\mathsf{c2}! \parallel_{\{\mathsf{c2}\}} P_2\right)$ is legal, and indeed any program violating this requirement can be converted to one which satisfies it by suitable renaming of channels. Also, we require that only one branch of each parallel computation uses a given channel in a given direction; that is, a component cannot both send and receive along $\mathsf{c}$.

Therefore we define the sets $\mathsf{In}(P)$, $\mathsf{Out}(P)$, and $\mathsf{Hid}(P)$ for each process and guarded process $P$. Also, $\mathsf{Chan}(P) = \mathsf{In}(P) \cup \mathsf{Out}(P) \cup \mathsf{Hid}(P)$. The intention is that $\mathsf{In}(P)$ will be the set of channels that are used for input only in $P$, $\mathsf{Out}(P)$ is the set of channels that are used for output, $\mathsf{Hid}(P)$ is the set of channels that are "hidden" by $P$, *i.e.* cannot be used outside of $P$, and that $\mathsf{Chan}(P)$ is the set of all channels used by $P$.

These sets are defined on the syntax of the language in a straightforward manner. From these definitions we then proved obvious sanity conditions; *e.g.*$\mathsf{In}(P)$, $\mathsf{Out}(P)$, and $\mathsf{Hid}(P)$ are disjoint. A similar set of conditions were proved for the variables of a legal Joy program.

## A.2 Semantics of Circuits

We can consider a circuit as a graph, with the gates being nodes of the graph, and the wires being edges connecting them.

The state of a circuit has the form $W \dagger \langle \eta, \xi \rangle$, where:

$W$ is a set of wires of the graph, representing those currently active.

$\eta$ is a function mapping variables to values, giving the states of variables. (In the proofs as presented, little use is made of this value.)

$\xi$ is a function telling the states of the call-box components: $\mathsf{c}!, \mathsf{c}?, x\uparrow, x\downarrow$ and $x$, for every channel $c$ and every variable $x$. It maps each such node to either "-" (indicating that no call is pending), or a label $l$ of a gate connected to the call-box (indicating that gate $l$ has made the most recent call to the component).

Notice that we could simply have defined the state of a circuit to be a set of wires and a mapping from all its state-bearing components to their values, but that would have blurred some useful distinctions. We will be defining a relation between Joy Prime programs and circuits, and we want the values of the variables in the Joy Prime program to be identical to the values of the circuit components representing those variables. Therefore, we separate the state of the state-bearing components into $\eta$ and $\xi$, so that $\eta$ will be used both for circuit and Joy Prime states, while the function $\xi$ will have to be computed from the Joy Prime state.

Wires are named in the format *gate-name.port-name*, where "gate name" is the name of the circuit component, and "port name" is the name of one of the connections to that type of component. For example, the name of the request wire connecting to a component labeled $l$ which happens to be a sequence component would be $l;.r$, because the gate would be named "$l;$".

One side-effect of this convention is that each wire has two names – one name for each of the two gates it is connected to. Although this has the potential of being confusing, in practice this naming system proves to be less inconvenient than the alternatives, and some lemmas we prove later on guarantee that the two names for a wire will always be consistent. The ports for a representative sample of gates are shown in Figures 6 and 7.

We define the transition relation $W \dagger \langle \eta, \xi \rangle \overset{\alpha}{\rightsquigarrow} W' \dagger \langle \eta', \xi' \rangle$, by simply listing all possible transitions. Figure 8 describes the behaviour of some of the simple gates, *i.e.* the gates which do not affect $\eta$ or $\xi$, and which do not undertake any visible actions. Figure 9 describes the gates involved in communication. A similar table for variables was constructed, but is omitted.

The interpretation of the tables is:

$W \dagger \langle \eta, \xi \rangle \overset{\alpha}{\rightsquigarrow} W' \dagger \langle \eta', \xi' \rangle$ if

- **Before** $\subseteq W$
- $l$ is the label of a component of the type given in the column labeled **Box**
- $W' = (W -$ **Before**$) \cup$ **After**
- $\alpha$ is given in the **Action** column of the table,
- $\eta'$ and $\xi'$ are equal to $\eta$ and $\xi$ except on any value stated in the **Effect** column. A new mapping $\eta$ reflects a change in the value of a variable, and a change in $\eta$ reflects a change in the internal state of a callbox, recording which input wire a request came from.

The structure of the circuit transition relation guarantees the following useful fact:

**Lemma A.1 Wire Monotonicity Lemma**
*If*

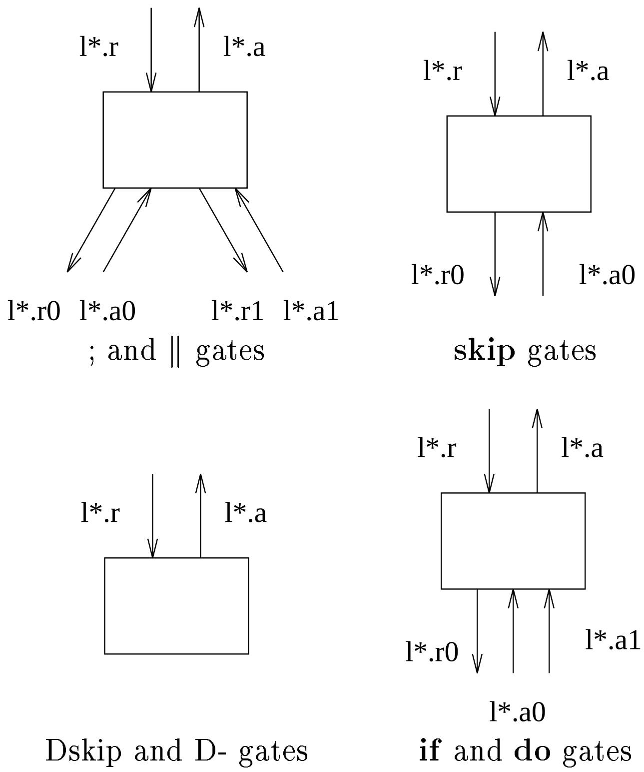


Figure 6: Diagrams of Gates

- $W \ddagger \langle \eta, \tau \rangle \overset{\bar{}}{\rightsquigarrow} W' \ddagger \langle \eta', \tau' \rangle$
- *$W$ and $W''$ are disjoint*
- *The domain of $\eta''$ is disjoint from that of $\eta$ and $\eta'$*
- *The domain of $\tau''$ is disjoint from that of $\tau$ and $\tau'$*

*then*

$$W \cup W'' \ddagger \Big\langle (\eta \ddagger \eta''), (\tau \ddagger \tau'') \Big\rangle \overset{\alpha}{\rightsquigarrow} W' \cup W'' \ddagger \Big\langle (\eta' \ddagger \eta''), (\tau' \ddagger \tau'') \Big\rangle \tag{22}$$

The above notation for wire names is convenient for discussing circuits, but turns out to be somewhat awkward for proving things. Therefore, we introduce alternate names for the wires connected to callboxes. Since this is a purely syntactic change, it doesn't affect the circuit operation at all, but it does allow Definition B.8 to be made with only 38 cases, instead of the considerably more it would otherwise have.

**Notation A.2** *For all call-box gates ($\mathsf{c}!, \mathsf{c}?, x\uparrow, x\downarrow, x$), we will give alternate names for the wires associated with them as follows:*

- *wire $*.rl$ will be called $l*.r$, $* \in \{\mathsf{c}!, \mathsf{c}?, x\uparrow, x\downarrow, x\}$*

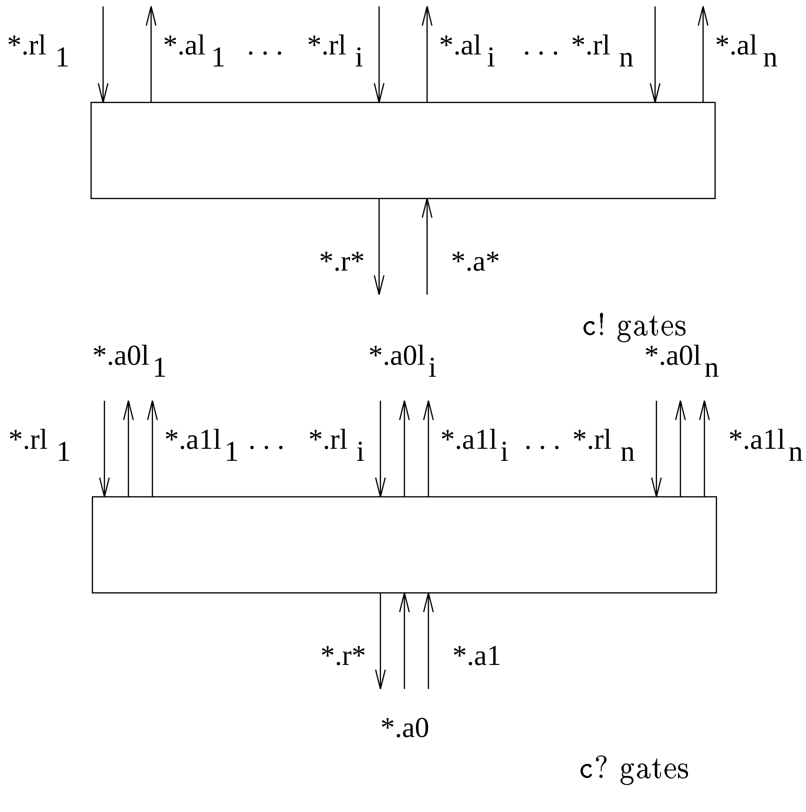


Figure 7: Diagrams of communication and variable gates

| **Box** | **Before** | **Action** | **After** | **Effect** |
|---|---|---|---|---|
| | $\{l;.r\}$ | - | $\{l;.r0\}$ | |
| ; | $\{l;.a0\}$ | - | $\{l;.r1\}$ | |
| | $\{l;.a1\}$ | - | $\{l;.a\}$ | |
| **skip** | $\{l\mathbf{skip}.r\}$ | - | $\{l\mathbf{skip}.r0\}$ | |
| | $\{l\mathbf{skip}.a0\}$ | - | $\{l\mathbf{skip}.a\}$ | |
| Dskip | $\{l\mathrm{Dskip}.r\}$ | - | $\{l\mathrm{Dskip}.a\}$ | |
| | $\{l\mathbf{do}.r\}$ | - | $\{l\mathbf{do}.r0\}$ | |
| **do** | $\{l\mathbf{do}.a0\}$ | - | $\{l\mathbf{do}.a\}$ | |
| | $\{l\mathbf{do}.a1\}$ | - | $\{l\mathbf{do}.r\}$ | |
| | $\{l\&.r\}$ | - | $\{l\&.r0\}$ | |
| | $\{l\&.af0\}$ | - | $\{l\&.a0\}$ | |
| & | $\{l\&.af1\}$ | - | $\{l\&.r1\}$ | |
| | $\{l\&.as0\}$ | - | $\{l\&.a0\}$ | |
| | $\{l\&.as1\}$ | - | $\{l\&.a1\}$ | |
| | $\{l \rightarrow .r\}$ | - | $\{l \rightarrow .r0\}$ | |
| $\rightarrow$ | $\{l \rightarrow .af0\}$ | - | $\{l \rightarrow .a0\}$ | |
| | $\{l \rightarrow .af1\}$ | - | $\{l \rightarrow .r1\}$ | |
| | $\{l \rightarrow .as\}$ | - | $\{l \rightarrow .a1\}$ | |
| | $\{l\|.r\}$ | - | $\{l\|.r0\}$ | |
| | $\{l\|.af0\}$ | - | $\{l\|.r1\}$ | |
| $\|$ | $\{l\|.af1\}$ | - | $\{l\|.a1\}$ | |
| | $\{l\|.as0\}$ | - | $\{l\|.a0\}$ | |
| | $\{l\|.as1\}$ | - | $\{l\|.a1\}$ | |
| tt | $\{l\mathrm{tt}.r\}$ | - | $\{l\mathrm{tt}.a1\}$ | |

Figure 8: Actions of Simple Gates

| **Box** | **Before** | **Action** | **After** | **Effect** |
|---|---|---|---|---|
| c! | $\{\mathsf{c!}.rl\}$ | - | $\{\mathsf{c!.r!}\}$ | $\xi(\mathsf{c!}) := l$ |
| | $\{\mathsf{c!.a!}\}$ | - | $\{\mathsf{c!}.a(\xi(\mathsf{c!}))\}$ | $\xi(\mathsf{c!}) :=$ - |
| | $\{\mathsf{c?}.rl\}$ | - | $\{\mathsf{c?.r?}\}$ | $\xi(\mathsf{c?}) := l$ |
| c? | $\{\mathsf{c?.a0?}\}$ | - | $\{\mathsf{c?}.a0(\xi(\mathsf{c?}))\}$ | $\xi(\mathsf{c?}) :=$ - |
| | $\{\mathsf{c?.a1?}\}$ | - | $\{\mathsf{c?}.a1(\xi(\mathsf{c?}))\}$ | $\xi(\mathsf{c?}) :=$ - |
| $c\star$ | $\{c\star.\mathsf{r!}, c\star.\mathsf{r?}\}$ | - | $\{c\star.\mathsf{a!}, c\star.\mathsf{a1?}\}$ | |
| c?,c! in $S$ | $\{c\star.\mathsf{r?}\}$ | - | $\{c\star.\mathsf{a0?}\}$ | |
| $c\star$ | $\{c\star.\mathsf{r!}\}$ | c! | $\{c\star.\mathsf{a!}\}$ | |
| c? not in $S$ | | | | |
| $c\star$ | $\{c\star.\mathsf{r?}\}$ | c? | $\{c\star.\mathsf{a1?}\}$ | |
| c! not in $S$ | $\{c\star.\mathsf{r?}\}$ | - | $\{c\star.\mathsf{a0?}\}$ | |
| - | $\{l\text{-}.r\}$ | - | $\{l\text{-}.r0\}$ | |
| | $\{l\text{-}.af1\}$ | - | $\{l\text{-}.a1\}$ | |
| D- | $\{l\mathrm{D}\text{-}.r\}$ | - | $\{l\mathrm{D}\text{-}.a\}$ | |

Figure 9: Actions of Communication Gates

- *wire* $*.al$ *will be called* $l*.a$, $* \in \{\mathsf{c!}, x\uparrow, x\downarrow\}$
- *wire* $*.a0l$ *will be called* $l*.a0$, $* \in \{\mathsf{c?}, x\}$ *and*
- *wire* $*.a1l$ *will be called* $l*.a1$, $* \in \{\mathsf{c?}, x\}$

*These names do not conflict with any other existing names.*

With this notation, any component which is labelled $l$ and has type $*$ will be activated by a signal along a wire called $l*.r$, and will return an acknowledgement on wire $l*.a$. Also, any expression will be activated using wire $l*.r$, and return its result on wire $l*.a0$ or $l*.a1$.

## A.3 The Compiler

We now formalize the compilation. Consider a Joy program $J$. The circuit corresponding to $J$, *circuit*($J$), is a directed multigraph corresponding to the parse tree of $J$ with extra nodes for communication and variables. In the following, we will assume that call-boxes can have arbitrarily many inputs. This is not true of real circuit components, however, as is commonly done, two or more call-boxes can be connected together to form one call-box with more inputs. Thus, this assumption is reasonable. The graph is defined as follows:

- Give each subterm of $J$ a unique label $l$.
- For each variable $x$ in $J$, there will be a gate $\nu_x$, and call-boxes $x\uparrow, x\downarrow$ and $x$.
- For each communication channel $c$ there will be a gate $c\star$. If $c \in \mathsf{In}(P) \cup \mathsf{Hid}(P)$, there will be a c? call-box. If $c \in \mathsf{Out}(P) \cup \mathsf{Hid}(P)$, there will be a c! call-box.

- Every subterm of $J$ which is not a communication (*i.e.*, not $\mathtt{c}!, \mathtt{c}?$ or $x\downarrow$), there is a gate $l*$, where $l$ is the label of that subterm, and $*$ is its operation.
- For each $l$**skip** gate in the circuit, there is an accompanying $l$Dskip gate. Also, for every $l$- gate there is an $l$D- gate.
- Connect the wires for every **skip**, Dskip, - and D- gate as follows:
  - Every $l$**skip**.$r0$ wire is connected to the corresponding $l$Dskip.$r$ wire.
  - Every $l$Dskip.$a$ wire is connected to the corresponding $l$**skip**.$a0$ wire.
  - Every $l$-.$r0$ wire is connected to the corresponding $l$D-.$r$ wire.
  - Every $l$D-.$a$ wire is connected to the corresponding $l$-.$ad$ wire.
- The c! and c? gates will be connected to the $c\star$ gates as follows:
  - c!.r! is connected to $c\star$.r!
  - c!.a! is connected to $c\star$.a!
  - c?.r? is connected to $c\star$.r?
  - c?.a0? is connected to $c\star$.a0?
  - c?.a1? is connected to $c\star$.a1?
- The other gates are connected according to their function, in the obvious way.

For example, if $L = L_0\,;L_1$, then we have a sequence gate, called $l;$. $l;$ is connected to its parent and children by appropriate interfaces; *e.g.*, if $L = L_0\,;L_1$ is labelled $l$, $L_i$ is labelled $l_i$, $L$'s parent is labelled $l_p$ and the operations of $L_0$ and $L_1$ are $*_0$ and $*_1$ respectively, then we have the following edges. Note that each edge has two names, one from each node.

1. Request wires from $l;$ to $l_i*_i$. These wires are called $l;.r0$ and $l;.r1$ as seen from $l;$, and $l_0.r$ and $l_1.r$ as seen from $\nu_{l_i}$.
2. Acknowledge wires from $l_i*_i$ to $l;$, called $l;.a0$ and $l_0*_0.a$, and $l;.a1$ and $l_1*_1.a$.
3. A request wire $l;.r$ from $l_p*_p$ to $l;$, where $*_p$ is the operation of $L$'s parent. The $l_p*_p$-name of this wire depends on the structure of $L$'s parent; e.g. if $L$ is the second component of a sequence, the $l_p$-name is $l_p*_p.r1$.
4. An acknowledge wire $l;.a$ from $l;$ to $l_p*_p$. As with $l.r$, the $l_p*_p$-name depends on $L$'s parent.

See Figure 10 for a diagram of this.

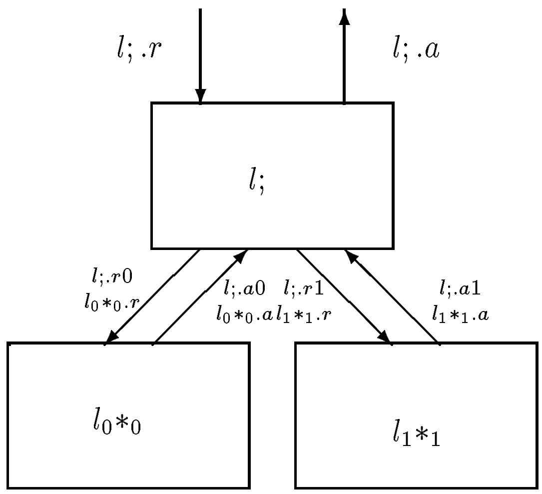


Figure 10: The construction of the circuit for a sequence operation.

# B Joy Prime

We now define the intermediate language "Joy Prime". As discussed in Section 4.4.1, Joy Prime is intended to be a variant of Joy which precisely matches circuit execution. Joy Prime programs are similar to Joy programs, with vast amounts of extra information added.

Every subterm of a Joy Prime program has a *label* associated with it. A label consists of three parts, as follows:

- An optional symbol "$v$". The presence of this symbol will serve to indicate that the term (and, consequently, all its subterms), has not begun to execute.
- A unique identifier. In our translation from Joy Prime to circuits, there is exactly one circuit component corresponding to every program term. The identifier in the label of the Joy Prime term will be the same identifier as in the name of the corresponding circuit component. In our grammar and rules we will usually use $l, l', l_1$ and so on to represent these identifiers.
- The symbol of an operation. This will always be the symbol of the circuit component corresponding to the identifier in the label. In the following we will use $*, *_1, *_2$ and so on as variables ranging over operation symbols.

The last two components of a label (the identifier and the operation symbol) will not actually have any effect upon the computation of a Joy Prime program. Their only purpose is to make easier the association between Joy Prime programs and circuits. It would be possible to remove them entirely from Joy Prime, however doing so would make it more difficult to compute the circuit component corresponding to a particular term. Also, these components are used in various places in the proof: for example, subcircuits which return values have to be distinguished from those which don't in (67) and (68).

| | |
|---|---|
| VP | Virgin Programs: program terms which have not yet started to execute |
| VGP | Virgin Guarded Programs (appearing in the bodies of virgin **if** and **do** commands). |
| VR | Virgin Read Requests, which appear as the communication possibilities in guards: - to do no communication, or c? to look for input on channel c. Guards have the form $E\&R$. |
| AP | Active Programs: programs which have been executed, partially or completely. **done** is the only completely executed program. |
| AGP | Active Guarded Programs: partially executed guarded command bodies. |
| AE | Active Expressions. |

Figure 11: Joy Prime syntactic classes

First we define the syntax of Joy Prime. A summary of the syntactic classes we use may be found in Figure 11. The $v$ marker distinguishes virgin from active programs.

Virgin programs, then, are marked everywhere with $v$'s. However, active programs generally contain $v$'s as well: for example, in "$P_1; P_2$", $P_2$ has not yet started executing; control is still in $P_1$, or at latest in the transition from $P_1$ to $P_2$. We choose to embody these restrictions directly in the definition of Joy Prime programs, giving a BNF grammar which enforces them. For example, $AP$ is the syntactic class of active programs; we have a production

$$AP \rightarrow (AP\,;VP)^{l;}$$

embodying the fact that $P_2$ has not started executing. This design choice complicates the language, but simplifies the proofs. (Another way of viewing this trick is that we conceal the statement of a theorem in the grammar of Joy Prime, and the proof of it in the fact that the consequents of the SOS rules for Joy Prime are always Joy Prime terms.)

In addition to the language constructs used in Joy, the syntax of Joy Prime has a number of other constructs which allow us to more concretely represent the actual execution of a program. Some of these symbols are

**done(c!):** This indicates that a c! command has communicated with its corresponding c? statement, but has not yet completed execution.

**pdo**$_{(V)}$ $A$ **odp:** In the same way as is done for the if command, **pdo**$_{(V)}$ $A$ **odp** records which guards have yet to evaluate and the complete list of guards.

**fail:** This indicates that a c? or a guard has failed.

**done(c?),fail(c?):** These indicate that a c? operation has communicated with a c! statement, and has not yet finished execution. The former symbol indicates that the communication was successful, and the latter that it was not.

**done(tt),done(ff):** These indicate that the reading of a variable has been done, but that evaluation has not stopped yet. The value of the variable read is tt or ff, respectively.

**Definition B.1**

*(Abbreviated)*

$$
\begin{array}{rcllcl}
VP & ::== & \mathbf{skip}^{vl\mathbf{skip}} & | & \mathsf{c}!^{vl\mathsf{c}!} \\
 & | & (VP\,;VP)^{vl;} & | & \mathbf{do}\,VGP\,\mathbf{od}^{vl\mathbf{do}} \\
VGP & ::== & (VGP \| VGP)^{vl\|} & & \\
 & | & ((VE \& VR)^{vl\&} \rightarrow VP)^{vl\rightarrow} & & \\
VR & ::== & \text{-}^{vl\text{-}} & | & \mathsf{c}?^{vl\mathsf{c}?} \\
VE & ::== & (VE \wedge VE)^{vl\wedge} & | & \mathsf{tt}^{vl\mathsf{tt}} \\
AP & ::== & (AP)^{l;} & | & \mathbf{done}(\mathsf{c}!)^{l\mathsf{c}!} \\
 & | & \mathsf{c}!^{l\mathsf{c}!} & | & (AP \parallel_{\mathcal{C}} AP)^{l\|} \\
 & | & (AP\,;VP)^{l;} & | & \mathbf{done}^{l*} \\
 & | & VP & | & \mathbf{skip}^{l\mathbf{skip}} \\
 & | & \mathbf{do}\,VGP\,\mathbf{od}^{l\mathbf{do}} & | & \mathbf{pdo}_{(VGP)}\,AGP\,\mathbf{odp}^{l\mathbf{do}} \\
AGP & ::== & AGP^{l\|} & | & (AGP \| VGP)^{l\|} \\
 & | & VGP & | & \mathbf{done}^{l\|} \\
 & | & \mathbf{fail}^{l\|} & | & \mathbf{fail}^{l\rightarrow} \\
 & | & AP^{l\rightarrow} & | & \mathbf{done}^{l\rightarrow} \\
AR & ::== & \text{-}^{l\text{-}} & | & \mathbf{done}^{l\text{-}} \\
 & | & \mathsf{c}?^{l\mathsf{c}?} & | & \mathbf{done}(\mathsf{c}?)^{l\mathsf{c}?} \\
 & | & \mathbf{fail}(\mathsf{c}?)^{l\mathsf{c}?} & | & \mathbf{done}^{l\mathsf{c}?} \\
 & | & \mathbf{fail}^{l\mathsf{c}?} & | & VR \\
AE & ::== & \mathbf{done}(\mathsf{tt})^{lx} & | & \mathsf{tt}^{l*} \\
 & | & (AE \wedge VE)^{l\wedge} & | & AE^{l\wedge} \\
 & | & VE & & 
\end{array}
$$

It is straightforward to translate a Joy program, $J$, to a Joy Prime program $Prime(J)$; add distinct virgin labels to every subterm of the program.

**Definition B.2** *A Joy Prime program $P$ is* I/O correct *iff each subterm (including the whole program) meets the necessary requirements for* $\mathsf{In}(P)$, $\mathsf{Out}(P)$, *and* $\mathsf{Hid}(P)$, *with the definitions extended for Joy Prime constructs which are not in Joy in the obvious manner. A Joy Prime program $P$ is* variable correct *in the same manner. A Joy Prime program $P$ is a* well-formed program *iff it is both I/O and variable correct.*

## B.1 Operational Semantics of Joy Prime

Joy Prime programs execute according to the following rules. Notice that, except for the four label rules, the labels on terms are simply preserved. We have the following operational relations:

$P \dagger \eta \xrightarrow{-} P' \dagger \eta'$**:** Process $P$ in variable state $\eta$ can perform an $\alpha$-action and thereafter behave like process $P'$ in variable state $\eta'$.

$P \fatsemi \eta \xrightarrow{\alpha} P' \fatsemi \eta$: where $\alpha$ can be -, c?, or c!. This means that process $P$ in variable state $\eta$ can perform the action $\alpha$ and thereafter behave like $P'$ without changing any variables.

$P \xrightarrow{\alpha} P'$: In any variable state, $P$ can perform $\alpha$ and thereafter behave like $P'$.

Note that, as in Joy, no single transition can involve both communication and variable update. As both types of update are allowed in most contexts, we use the following notation:

**Notation B.3** *The expression*

$$P \fatsemi \eta \xrightarrow{\alpha} P' \fatsemi \eta'$$

*is an abbreviation for*

- $P \fatsemi \eta \xrightarrow{-} P' \fatsemi \eta'$*, or*
- $P \fatsemi \eta \xrightarrow{\alpha} P' \fatsemi \eta'$ *and* $\eta' = \eta$*, or*
- $P \xrightarrow{\alpha} P'$*, and* $\eta' = \eta$*, for any* $\eta$

Notice that we do not have the Joy semantic relations for "$E$ evaluates to $v$ in state $\eta$" and "$GP$ fails." These relations are too abstract for Joy Prime, as they collapse many steps of computation into a single one, and cannot be computed in one step on a circuit. They can be derived from the Joy Prime semantics:

1. $E$ evaluates to $v$ in state $\eta$ iff $E \fatsemi \eta \xrightarrow{-}^{n} v^{l*} \fatsemi \eta'$ for some $l$, $n$ and $*$.
2. $GP \fatsemi \eta$ fails iff $GP \fatsemi \eta \xrightarrow{-}^{n} \mathbf{fail}^{l*} \fatsemi \eta$ for some $l$, $n$ and $*$.

The operational semantics of Joy Prime is given by the following rules:

**Label rules:** Executing a virgin Joy Prime program corresponds to putting a signal on its request line. The following transition models the processing of that signal:

$$Q^{vl*} \xrightarrow{-} Q^{l*} \tag{23}$$

For the most part, during program execution, the label (and the fact that there is a surrounding larger circuit) makes no difference:

$$\frac{P \fatsemi \eta \xrightarrow{\alpha} P_1 \fatsemi \eta'}{P^{l*} \fatsemi \eta \xrightarrow{\alpha} P_1^{l*} \fatsemi \eta'} \tag{24}$$

When a circuit has finished executing, it places a signal on its acknowledge line, or (in the case of circuits returning values) on either its acknowledge-success or acknowledge-failure line. This will correspond to a $\mathbf{done}^{l*}$ or $\mathbf{fail}^{l*}$ term. All circuit components which simply forward acknowledgements from a single subcomponent will use the the following rules:

$$\mathbf{done}^{(l_1*_1)(l_2*_2)} \xrightarrow{-} \mathbf{done}^{l_2*_2} \tag{25}$$

$$\mathbf{fail}^{(l_1*_1)(l_2*_2)} \xrightarrow{-} \mathbf{fail}^{l_2*_2} \tag{26}$$

**Command Rules:** Skips simply execute and finish. Communication requests are sent to components which coordinate communication, requiring two transitions in circuits; we thus give $\mathtt{c!}$ two transitions in Joy Prime.

$$
\begin{aligned}
\mathbf{skip}^{l\mathbf{skip}} &\xrightarrow{-} \mathbf{done}^{l\mathbf{skip}} && (27)\\
\mathtt{c!}^{l\mathtt{c!}} &\xrightarrow{\mathtt{c!}} \mathbf{done}(\mathtt{c!})^{l\mathtt{c!}} && (28)\\
\mathbf{done}(\mathtt{c!})^{l\mathtt{c!}} &\xrightarrow{-} \mathbf{done}^{l\mathtt{c!}} && (29)
\end{aligned}
$$

A parallel composition finishes when both components finish. It is otherwise identical to the Joy parallel composition, with the addition of labels.

$$
(\mathbf{done}^{l_1 *_1} \parallel_C \mathbf{done}^{l_2 *_2})^{l\parallel} \xrightarrow{-} \mathbf{done}^{l\parallel} \tag{30}
$$

$$
\frac{P_1 \dagger \eta \xrightarrow{-} P_1' \dagger \eta'}{\begin{array}{c}(P_1 \parallel_C P_2)^{l\parallel} \dagger \eta \xrightarrow{-} (P_1' \parallel_C P_2)^{l\parallel} \dagger \eta' \\ (P_2 \parallel_C P_1)^{l\parallel} \dagger \eta \xrightarrow{-} (P_2' \parallel_C P_1)^{l\parallel} \dagger \eta'\end{array}} \tag{31}
$$

$$
\frac{P_1 \xrightarrow{\alpha} P_1', \alpha \notin C}{\begin{array}{c}(P_1 \parallel_C P_2)^{l\parallel} \xrightarrow{\alpha} (P_1' \parallel_C P_2)^{l\parallel} \\ (P_2 \parallel_C P_1)^{l\parallel} \xrightarrow{\alpha} (P_2' \parallel_C P_1)^{l\parallel}\end{array}} \tag{32}
$$

We allow communicating processes to take steps simultaneously. Here, $\beta$ is either $\mathtt{c?}$ or $\mathtt{c!}$, and $\hat{\beta}$ is the other one of those.

$$
\frac{P_1 \xrightarrow{\beta} P_1', P_2 \xrightarrow{\hat{\beta}} P_2'}{(P_1 \parallel_C P_2)^{l\parallel} \xrightarrow{-} (P_1' \parallel_C P_2')^{l\parallel}} \tag{33}
$$

Sequencing is as in Joy:

$$
\frac{P_1 \dagger \eta \xrightarrow{\alpha} P_1' \dagger \eta'}{(P_1 \,;\, P_2)^{l;} \dagger \eta \xrightarrow{\alpha} (P_1' \,;\, P_2)^{l;} \dagger \eta'} \tag{34}
$$

**Do rules:**

$$
\frac{GP \dagger \eta \xrightarrow{\alpha} GP' \dagger \eta'}{\begin{array}{c}(\mathbf{pdo}_{(G)}\, GP\ \mathbf{odp})^{l\mathbf{do}} \dagger \eta \xrightarrow{\alpha} (\mathbf{pdo}_{(G)}\, GP'\ \mathbf{odp})^{l\mathbf{do}} \dagger \eta' \\ (\mathbf{do}\ GP\ \mathbf{od})^{l\mathbf{do}} \dagger \eta \xrightarrow{\alpha} (\mathbf{pdo}_{(GP)}\, GP'\ \mathbf{odp})^{l\mathbf{do}} \dagger \eta'\end{array}} \tag{35}
$$

$GP$ will reduce to either **done** (for successful termination: both discovering that a guard is true, and executing the corresponding command), or **fail** (for failure, indicating that the guard must be tried again).

$$
\begin{aligned}
(\mathbf{pdo}_{(G)}\, \mathbf{done}^{L*}\ \mathbf{odp})^{l\mathbf{do}} &\xrightarrow{-} (\mathbf{pdo}_{(G)}\, G\ \mathbf{odp})^{l\mathbf{do}} && (36)\\
(\mathbf{pdo}_{(G)}\, \mathbf{fail}^{L*}\ \mathbf{odp})^{l\mathbf{do}} &\xrightarrow{-} \mathbf{done}^{l\mathbf{do}} && (37)
\end{aligned}
$$

**Guard rules:** Multiple guards are evaluated left to right.

$$\frac{GP_1 \ddagger \eta \xrightarrow{\alpha} GP_1' \ddagger \eta'}{(GP_1 \| GP_2)^{l\|} \ddagger \eta \xrightarrow{\alpha} (GP_1' \| GP_2)^{l\|} \ddagger \eta'} \tag{38}$$

If the left guard succeeds, then the right guard is not evaluated at all; but if it fails, the right guard must be evaluated.

$$(\mathbf{done}^{L*} \| GP_2)^{l\|} \quad \xrightarrow{-} \quad \mathbf{done}^{l\|} \tag{39}$$

$$(\mathbf{fail}^{L*} \| GP_2)^{l\|} \quad \xrightarrow{-} \quad GP_2^{l\|} \tag{40}$$

When evaluating a single guard $R \rightarrow P$, $R$ is attempted. As with sends and assignments, receives pass through an intermediate state required for the circuit.

$$\frac{R \xrightarrow{\alpha} R'}{(R \rightarrow P)^{l\rightarrow} \xrightarrow{\alpha} (R' \rightarrow P)^{l\rightarrow}} \tag{41}$$

$$(\mathbf{done}^{l\&} \rightarrow P)^{l'\rightarrow} \quad \xrightarrow{-} \quad P^{l'\rightarrow} \tag{42}$$

$$(\mathbf{fail}^{l\&} \rightarrow P)^{l'\rightarrow} \quad \xrightarrow{-} \quad \mathbf{fail}^{l'\rightarrow} \tag{43}$$

**Communication read rules:** Internal moves, "-", can be used as reads in guards; they always succeed. As explained in Section 3, reads always have the option of failing.

$$\text{-}^{l\text{-}} \quad \xrightarrow{-} \quad \mathbf{done}^{l\text{-}} \tag{44}$$

$$\mathtt{c?}^{l\mathtt{c?}} \quad \xrightarrow{-} \quad \mathbf{fail}(\mathtt{c?})^{l\mathtt{c?}} \tag{45}$$

$$\mathtt{c?}^{l\mathtt{c?}} \quad \xrightarrow{\mathtt{c?}} \quad \mathbf{done}(\mathtt{c?})^{l\mathtt{c?}} \tag{46}$$

$$\mathbf{fail}(\mathtt{c?})^{l\mathtt{c?}} \quad \xrightarrow{-} \quad \mathbf{fail}^{l\mathtt{c?}} \tag{47}$$

$$\mathbf{done}(\mathtt{c?})^{l\mathtt{c?}} \quad \xrightarrow{-} \quad \mathbf{done}^{l\mathtt{c?}} \tag{48}$$

**Expression Rules:** Expressions are evaluated in a fairly conventional fashion. As usual, variable references require an intermediate state.

$$x^{lx} \ddagger \eta \quad \xrightarrow{-} \quad \mathbf{done}(\eta(x))^{lx} \ddagger \eta \tag{49}$$

$$\mathbf{done}(\mathtt{tt})^{lx} \quad \xrightarrow{-} \quad \mathtt{tt}^{lx} \tag{50}$$

The remaining Boolean operations simply do the obvious things:

$$\frac{E_1 \ddagger \eta \xrightarrow{-} E_1'}{(E_1 \wedge E_2)^{l\wedge} \ddagger \eta \xrightarrow{-} (E_1' \wedge E_2)^{l\wedge}} \tag{51}$$

$$(\mathtt{tt}^{L*} \wedge E_2)^{l\wedge} \quad \xrightarrow{-} \quad E_2^{l\wedge} \tag{52}$$

As described earlier, we wish that the labels on Joy Prime terms indicate both which circuit component computes that term, and which operation the component performs. To make sure that our labels do, indeed, indicate what we want them to, we prove the following Label Sanity Lemma. In English, this lemma states that

1. As a Joy Prime program executes, all the components will maintain their relationships with each other. For instance, the left subterm of a sequence operator will always be the left subterm of that term, as long as it is still there.

2. The operation symbol in the label of a term remains constant. For instance, a term which is a sequence operator initially will always be labelled one, as long as it is present at all. For example,

$$\left(\mathbf{done}^{l_1\mathbf{skip}}\,;\mathbf{skip}^{l_2\mathbf{skip}}\right)^{l;} \xrightarrow{-} \left(\mathbf{skip}^{l_2\mathbf{skip}}\right)^{l;}$$

   There is no term labelled $l_1$. There is still a term labelled $l;$, because the ; operation hasn't yet terminated.

3. An operation will never be labeled by a label which contains the wrong operation symbol. For example, no term which is a sequence operation will ever be labelled as a parallel operation.

4. If a term is marked as being unexecuted (virgin), then all of its subterms are virgin as well.

**Lemma B.4 Label Sanity**

*Let $S$ be a virgin Joy Prime program, and let $S \xrightarrow{\vec{\alpha}} P$. Let $T_S$ and $T_P$ be the parse trees of $S$ and $P$ respectively, where a parse tree is defined so as not to include the subscript of a partial if. Then,*

1. *Considering only labels, ignoring virginity, $T_P$ is a subtree of $T_S$*

2. *The label on a node of $T_P$ has the same operation as that on the corresponding node of $T_S$*

3. *The term underneath a given label always has a operation which corresponds to the operation symbol in the label in $T_P$.*

4. *If a label is not virgin in $T_P$, all labels above it are not.*

## B.2 Active Subterms

In order to prove bisimulation between Joy Prime and circuits, we will have to show that for every action that a Joy Prime program can do, a circuit can do a corresponding action, and vice-versa. To make it feasible to analyze every single action a Joy Prime program can make, we will characterize exactly when a program can take each action, and exactly what the results of the action are.

Luckily, there is a great deal of regularity about how Joy Prime programs execute. For instance, consider Joy Prime program "$(P_1 \,;\, P_2)^{l;}$". If $P_1$ is a finished subprogram, then the program will take one transition to $P_2^{l;}$. Otherwise, the actions that the program can do are exactly those actions that $P_1$ alone can do, and none of these actions will affect any part of the program outside $P_1$.

We define for every program $P$ a set of subterms $act(P)$, the *active subterms* of $P$. Informally, whatever action a term can do is the direct result of one or more of its active subterms taking the corresponding action. The idea is that just as any action a circuit might perform is the result of one or more signals on some of its wires, any action of a program is the result of one or more of its subterms being "ready" to execute. In fact, there will be an exact correspondence between active subterms of a program, and signals on wires in the program's circuit.

**Definition B.5** *act(P) is the following set of subterms of P:*

*(Abbreviated)*

$$
\begin{array}{rcll}
act((P_1 \parallel_C P_2)^{l\parallel}) & = & act(P_1) \cup act(P_2) & (53) \\
 & & \quad \textit{if } P_1 \neq \mathbf{done}^{L*} \vee P_2 \neq \mathbf{done}^{L*} & (54) \\
act((P_1 \,;\, P_2)^{l;}) & = & act(P_1) \quad \textit{if } P_1 \neq \mathbf{done}^{L*} & (55) \\
act(P^{l;}) & = & act(P) \quad \textit{if } P \neq \mathbf{done}^{L*} & (56) \\
act(\mathbf{pdo}_{(Q)}\ P\ \mathbf{odp}^{l\mathbf{do}}) & = & act(P) \quad \textit{if } P \neq \mathbf{done}^{L*}, \mathbf{fail}^{L*} & (57) \\
act((P_1 \Vert P_2)^{l\Vert}) & = & act(P_1) \quad \textit{if } P_1 \neq \mathbf{done}^{L*} & (58) \\
act(P_1^{l\Vert}) & = & act(P_1) \quad \textit{if } P, \neq \mathbf{done}^{L*}, \mathbf{fail}^{L*} & (59) \\
act((C \rightarrow P)^{l\rightarrow}) & = & act(C) \quad \textit{if } C \neq \mathbf{done}^{L*}, \mathbf{fail}^{L*} & (60) \\
act(P_1^{l\rightarrow}) & = & act(P) \quad \textit{if } P \neq \mathbf{done}^{L*}, \mathbf{fail}^{L*} & (61) \\
act((E \& V)^{l\&}) & = & act(E) \quad \textit{if } E \neq \mathtt{tt}^{L*} & (62) \\
act((E_1 \wedge E_2)^{l\wedge}) & = & act(E) \quad \textit{if } E \neq \mathtt{tt}^{L*} & (63) \\
act(E^{l\wedge}) & = & act(E) \quad \textit{if } E \neq \mathtt{tt}^{L*} & (64) \\
act(P) & = & \{P\} \quad \textit{otherwise} & (65)
\end{array}
$$

The Activity Lemma is one of the central lemmas in the proof of bisimulation. It says that the transitions of a Joy Prime program are completely determined by the transitions of its active subcomponents. If a program, $P$, takes an action, then there must be an active subcomponent which can make the same action (or, in the case of communication, two subcomponents which can do the communication). Furthermore, the state resulting from $P$ taking making its transition is simply that state resulting from substituting for every active component that took a transition, the new state of that subcomponent.

**Lemma B.6 Activity Lemma**

*If $P \ddagger \eta \xrightarrow{\alpha} P' \ddagger \eta'$, then either*

1. $\exists Q \in act(P)$ *such that* $Q \ddagger \eta \xrightarrow{\alpha} Q' \ddagger \eta', P' = P[Q := Q']$

2. $\alpha =$ -$, \eta = \eta'$ *and* $\exists Q_1, Q_2 \in act(P)$ *such that* $Q_1 \xrightarrow{\beta} Q'_1, Q_2 \xrightarrow{\hat{\beta}} Q'_2, \beta \in \mathsf{Hid}(P), P' = P[Q_1, Q_2 := Q'_1, Q'_2]$

**Proof:** The activity lemma is proved by induction on the proof that $P \dagger \eta \xrightarrow{\alpha} P' \dagger \eta'$; all the cases are trivial.

As a typical base case, consider $P = Q^{vl*}$. Only one rule applies to virgin terms, *viz.*(23). Hence $P' = P_1^{l*}$ and $\eta' = \eta$. By definition, $act(P) = \{P\}$. Choosing $Q = P$ satisfies the lemma.

As the hardest inductive case, consider $P = (P_1 \parallel_C P_2)^{l\parallel}$, and suppose that $P \dagger \eta \xrightarrow{-} P' \dagger \eta'$ via (33), with $P_1 \xrightarrow{\beta_1} P'_1$ and $P_2 \xrightarrow{\beta_2} P'_2$, where $\beta_1 = \hat{\beta}_2$.

Clearly, from the rules, we have $\alpha =$ - and $\eta = \eta'$. Note that, as $P_i \xrightarrow{\cdot}$, neither $P_i$ is **done**; hence $act(P) = act(P_1) \cup act(P_2)$. By induction, there are processes $Q_i \in act(P_i)$ such that $Q_i \xrightarrow{\beta_i} Q'_i$ and $P'_i = P_i[Q_i := Q'_i]$; note that the inductive cases cannot be case 2, as $\beta_i \neq$ -. As $Q_i \in act(P_i) \subseteq act(P)$, we have $Q_i \in act(P)$ as desired. It is straightforward from the non-duplication of labels that $P' = P[Q_1 := Q'_i, Q_2 := Q'_2]$. The fact that $\beta \in \mathsf{Hid}(P)$ follows because of basic sanity properties (which are proven formally in the complete proof). $\triangle\!\!\!\!+$

The Converse Activity Lemma says that, if the active subcomponents of a program can do something, then the entire program can do the same thing. More precisely, if an active subcomponent, $Q$, of a program $P$ can take a transition to $Q'$, then $P$ can take the same action and end up in the state where $Q'$ is textually substituted for $Q$.

**Lemma B.7 Converse Activity Lemma**
*If* $Q \in act(P), Q \dagger \eta \xrightarrow{\alpha} Q' \dagger \eta', \alpha \notin chan(P)$, *and let* $P' = P[Q := Q']$. *Then* $P \dagger \eta \xrightarrow{\alpha} P' \dagger \eta'$.

*Further, if* $\exists Q_1, Q_2 \in act(P)$ *such that* $Q_1 \xrightarrow{\beta} Q'_1$ *and* $Q_2 \xrightarrow{\hat{\beta}} Q'_2$, *then* $P \xrightarrow{-} P' = P[Q_1, Q_2 := Q'_1, Q'_2]$.

## B.3 Wires corresponding to a subterm

Now, for every active subterm of a program, we will define a set of wires that correspond to the subterm. It is our intention that, if $A$ is an active subterm of a program $P$, then *activates*$(A)$, the wires that the term $A$ makes active, will be the set of wires all of which will have signals on them in the corresponding circuit execution.

**Definition B.8**

*(Abbreviated)*

$$
\begin{aligned}
activates(P^{vl*}) &= \{l*.r\} && (66)\\
activates(\mathbf{done}^{l*}), * \notin \{\&, \rightarrow, \|, \text{-}, \mathsf{c}?\} &= \{l*.a\} && (67)\\
activates(\mathbf{done}^{l*}), * \in \{\&, \rightarrow, \|, \text{-}, \mathsf{c}?\} &= \{l*.a1\} && (68)\\
activates(\mathbf{fail}^{l*}) &= \{l*.a0\} && (69)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{activates}(\mathbf{skip}^{l\mathbf{skip}}) &= \{lDskip.r\} && (70)\\
\mathit{activates}(\mathtt{c!}^{l\mathtt{c!}}) &= \{c\star.\mathtt{r!}\} && (71)\\
\mathit{activates}(\mathbf{done}(\mathtt{c!})^{l\mathtt{c!}}) &= \{c\star.\mathtt{a!}\} && (72)\\
\mathit{activates}(\mathbf{do}\ G\ \mathbf{od}^{l\mathbf{do}}) &= \{l\mathbf{do}.r1\} && (73)\\
\mathit{activates}(\mathbf{pdo}_{(Q)}\ \mathbf{done}^{L*}\ \mathbf{odp}^{l\mathbf{do}}) &= \{L*.a1\} && (74)\\
\mathit{activates}(\mathbf{pdo}_{(Q)}\ \mathbf{fail}^{L*}\ \mathbf{odp}^{l\mathbf{do}}) &= \{L*.a0\} && (75)\\
\mathit{activates}(\text{-}^{l\text{-}}) &= \{lD\text{-}.r\} && (76)\\
\mathit{activates}(\mathtt{c?}^{l\mathtt{c?}}) &= \{c\star.\mathtt{r?}\} && (77)\\
\mathit{activates}(\mathbf{done}(\mathtt{c?})^{l\mathtt{c?}}) &= \{c\star.\mathtt{a1?}\} && (78)\\
\mathit{activates}(\mathbf{fail}(\mathtt{c?})^{l\mathtt{c?}}) &= \{c\star.\mathtt{a0?}\} && (79)\\
\mathit{activates}(x^{lx}) &= \{\nu_x.rx\} && (80)\\
\mathit{activates}(\mathbf{done}(\mathtt{tt})^{lx}) &= \{\nu_x.a1\} && (81)\\
\mathit{activates}(\mathtt{tt}^{l*}) &= \{l*.a1\} && (82)\\
\mathit{activates}((\mathbf{done}^{l_1*_1} \parallel_{\mathcal{C}} \mathbf{done}^{l_2*_2})^{l\parallel}) &= \{l_1*_1.a, l_2*_2.a\} && (83)\\
\mathit{activates}((\mathbf{done}^{l_1*}\ ;\ P_2)^{l;}) &= \{l_1*.a\} && (84)\\
\mathit{activates}(\mathbf{done}^{(l_1*_1)(l_2*_2)}) &= \{l_1*_1.a\} && (85)\\
\mathit{activates}((E_1 \wedge E_2)^{l\wedge}) &= \mathit{activates}(E_1), E_1 \notin \{\mathtt{tt}, \mathtt{ff}\} && (86)\\
\mathit{activates}((\mathbf{done}^{L*} \| P_2)^{l\|}) &= \{L*.a1\} && (87)\\
\mathit{activates}((\mathbf{fail}^{L*} \| P_2)^{l\|}) &= \{L*.a0\} && (88)\\
\mathit{activates}((\mathbf{done}^{L*} \rightarrow P)^{l\rightarrow}) &= \{L*.a1\} && (89)\\
\mathit{activates}((\mathbf{fail}^{L*} \rightarrow P)^{l\rightarrow}) &= \{L*.a0\} && (90)\\
\mathit{activates}((\mathtt{tt}^{L*} \wedge E_2)^{l\wedge}) &= \{L*.a1\} && (91)
\end{aligned}
$$

We define $\mathit{wires}(P)$ to be the union of all the wires that the active subterms of $P$ make active. We will show that this is exactly the set of wires that have signals on them when the circuit is in the state corresponding to $P$.

**Definition B.9**

$$
\mathit{wires}(P) = \bigcup_{A \in \mathit{act}(P)} \mathit{activates}(A)
$$

Part of a circuit's state is the values of its call-boxes. Therefore, we have to define a relationship between the state of a Joy Prime process and the values of the call-boxes in the corresponding circuit. To do this, we first define two relations, *inside*() and *atdoor*(). For any operation $\gamma$ which involves a call-box (a communication or a variable setting), any Joy Prime program $P$, and a label $l$, $\mathit{inside}(\gamma, P, l)$ will be true if the execution of a $\gamma$ operation with the label $l$ has begun and not finished execution. $\mathit{atdoor}(\gamma, P, l)$ is true if the operation has not yet started execution, but it is possible that the next action of the program is to begin executing that operation.

**Definition B.10** *For any call-box operation (*i.e.$c?, c!$*), $\gamma$, and any Joy Prime program, $P$,*

$$inside(\gamma, P, l) \iff \exists R \text{ s.t. } R^{l\gamma} \in act(P), R \neq \textbf{done} \tag{92}$$

$$atdoor(\gamma, P, l) \iff \gamma^{vl\gamma} \in act(P) \tag{93}$$

This definition makes use of the fact that if a term is not labelled as virgin, then it has begun to execute, and that therefore any term of the form $R^{l\gamma}$ has begun to execute the operation $\gamma$.

Now, as it stands, there might be a number of values of $l$ which satisfy $inside(\gamma, P, l)$ or $atdoor(\gamma, P, l)$. Unfortunately, having different values satisfy these predicates would mean that mutual exclusion would be violated, which our circuit components do not allow. Therefore, we have to prove that there is at most one value of $l$ that can satisfy the predicates for a given $\gamma$ and $P$, and thus that when one term is doing or can do a certain call-box operation, then no other term is able to do that operation.

The $inside(\cdot,\cdot,\cdot)$ predicate is used to define the function $return(\cdot)$. This function relates a Joy Prime program to the state of the callboxes in the corresponding circuit. That is, if $P$ is a Joy Prime program, then we claim and show that $return(P)$ is the state of the callboxes in the circuit at that point of execution.

## B.4 Proving Bisimulation

The next two theorems show that steps of computation in Joy Prime can be matched by steps of computation in circuits, and vice versa; together, they establish bisimulation. Theorem B.11, establishes one direction of our bisimulation. It essentially, states that if our Joy Prime program takes one step of computation, then the circuit with the state defined by the *wires*() and *return*() functions can take a corresponding step. The second theorem, B.12 establishes the other direction: if a circuit corresponding to a certain Joy Prime program can take a step of computation, then the Joy Prime program can do that same step.

In the statement of Theorem B.11, we have the hypothesis that $S \dagger \eta_{\text{init}} \xrightarrow{\vec{\gamma}} P \dagger \eta$. This hypothesis is necessary is because the circuit we are considering was constructed from the program $S$, and so we can only relate circuit states to program states which are the result of executing $S$.

**Theorem B.11** *Let $S$ be a well-formed program, and let us consider the circuit defined by $S$. If*

- $S \dagger \eta_{init} \xrightarrow{\vec{\gamma}} P \dagger \eta$ *and*
- $P \dagger \eta \xrightarrow{\alpha} P' \dagger \eta'$,

*then*
$wires(P) \dagger \langle \eta, return(P)\rangle \overset{\alpha}{\rightsquigarrow} wires(P') \dagger \langle \eta', return(P')\rangle$.

**Proof:** We do case analysis, using the Activity Lemma (Lemma B.6); as usual, the cases are routine and we merely present examples.

By the Activity Lemma, $P \dagger \eta \xrightarrow{\alpha} P' \dagger \eta'$ either because one active subterm makes a transition, or because two active subterms communicate.

First, assume that $P \dagger \eta \xrightarrow{\alpha} P' \dagger \eta'$ because of case 1 of the Activity Lemma. Then, $\exists Q \in act(P), Q \dagger \eta \xrightarrow{\alpha} Q' \dagger \eta', P' = P[Q := Q']$.

From the definition of $activates(\cdot)$, and some properties of call-boxes, $activates(Q) \subseteq wires(P)$, and $wires(P') = (wires(P) - activates(Q)) \cup wires(Q')$

By wire monotonicity, it will be sufficient to show that

$$activates(Q) \dagger \langle \eta, return(Q) \rangle \overset{\alpha}{\rightsquigarrow} wires(Q') \dagger \langle \eta', return(Q') \rangle,$$

which follows by a case analysis on the structure of $Q$. We present here a typical case: $Q = (\mathbf{done}^{l_1 *_1} \parallel_C \mathbf{done}^{l_2 *_2})^{l \parallel}$.

In this case, since $Q \dagger \eta \xrightarrow{\alpha} Q' \dagger \eta'$ and only rule 30 applies, $Q'$ must be $\mathbf{done}^{l \parallel}$, $\alpha =$ -, $\eta' = \eta$ and $return(Q) = return(Q')$.

But $activates((\mathbf{done}^{l_1 *_1} \parallel_C \mathbf{done}^{l_2 *_2})^{l \parallel}) = \{l_1 *_1 .a, l_2 *_2 .a\}$. By the Label Sanity Lemma, $l_1 *_1 .a$ and $l_2 *_2 .a$ must be connected to the gate $l \|$ in the circuit defined by $S$, and so, by the definition of the action of that gate, $\{l_1 *_1 .a, l_2 *_2 .a\} \dagger \langle \eta, \xi \rangle \overset{-}{\rightsquigarrow} \{l \parallel .a\} \dagger \langle \eta, \xi \rangle$. But $\{l \parallel .a\} = wires(\mathbf{done}^{l \parallel}) = wires(Q')$, as we were required to show.

Now, let us consider what happens if $P \dagger \eta \xrightarrow{\alpha} P' \dagger \eta'$ by the second case of the Activity Lemma. Then $\exists Q_1, Q_2 \in act(P), Q_1 \xrightarrow{\beta} Q_1', Q_2 \xrightarrow{\hat{\beta}} Q_2', \beta \in \mathsf{Hid}(P), P' = P[Q_1, Q_2 := Q_1', Q_2']$.

By wire monotonicity and the Call-Box Lemma, it will be sufficient to show that

$$activates(Q_1) \cup activates(Q_2) \dagger \langle \eta, \xi \rangle \overset{-}{\rightsquigarrow} activates(Q_1') \cup activates(Q_2') \dagger \langle \eta, \xi \rangle$$

By the definition of $act(\cdot)$, without loss of generality we can assume that $\exists l_1, l_2$ such that $Q_1 = \beta ?^{l_1 \beta ?}$ and $Q_2 = \beta !^{l_2 \beta !}$.

Then $activates(Q_1) \cup activates(Q_2) = \{\beta \star \mathbf{r}?, \beta \star \mathbf{r}!\}, Q_1' = \mathbf{done}(\beta ?)^{l_1 \beta ?}, Q_2' = \mathbf{done}(\beta !)^{l_2 \beta !}$, and $activates(Q_1') \cup activates(Q_2') = \{\beta \star .\mathtt{a1}?, \beta \star \mathsf{a}!\}$

But, by the definition of the action of the gate $\beta \star$, $\{\beta \star \mathbf{r}?, \beta \star \mathbf{r}!\} \dagger \langle \eta, \xi \rangle \overset{-}{\rightsquigarrow} \{\beta \star .\mathtt{a1}?, \beta \star \mathsf{a}!\} \dagger \langle \eta, \xi \rangle$, which is what we needed to show. $\triangle$

Theorem B.12 has a number of hypotheses, which are needed for the same reason that the first hypothesis of Theorem B.11 is needed: we only are concerned about states which can be reached from the initial state. Since we have not tried to define an inverse function from circuit states to Joy Prime states, we require a number of hypotheses. We first require that the circuit state $W \dagger \langle \eta, return(P) \rangle$ be reachable from the initial state. We then require that the Joy Prime state $P \dagger \eta$ be mapped to $W \dagger \langle \eta, return(P) \rangle$, and that $P \dagger \eta$ be also reachable from the initial state. A trivial induction shows that such a $P$ will always exist for any circuit obtained from the initial conditions, so these hypotheses do not restrict the application of the theorem in any way.

**Theorem B.12** *Let $S$ be a well-formed program, and let us consider the circuit defined by $S$. If*

- $wires(S) \dagger \langle \eta_{init}, return(S) \rangle \overset{\vec{\gamma}}{\rightsquigarrow} W \dagger \langle \eta, return(P) \rangle$,

- $W = wires(P)$,
- $S \dagger \eta_{init} \xrightarrow{\vec{\gamma}} P \dagger \eta$ *and*
- $W \dagger \langle \eta, return(P) \rangle \stackrel{\alpha}{\rightsquigarrow} W' \dagger \langle \eta', \tau \rangle$,

*then*

- $\exists P', P \dagger \eta \xrightarrow{\alpha} P' \dagger \eta'$,
- $\tau = return(P')$ *and*
- $W' = wires(P')$.

**Proof:**
The proof is by cases on the transition $\stackrel{\alpha}{\rightsquigarrow}$.

We do one typical case: suppose the transition is a sequence being started: $l;.r \in W$, and

$$W \stackrel{-}{\rightsquigarrow} W' = W - \{l;.r\} \cup \{l;.r1\},$$

where $l;.r1 = l_1{*}.r$ for some $l_1, *$.

By the definition of $activates(\cdot)$, $(P_1 \,;\, P_2)^{vl;} \in act(P)$ for some $P_1$, $P_2$. By label sanity, $P_1 = R^{vl_1*}$. (*i.e.*, the same $l_1$ as connected to the ; gate, and virgin) Therefore,

$$activates(P_1) = \{l_1{*}.r\} \tag{94}$$

$(P_1 \,;\, P_2)^{vl;} \xrightarrow{-} (P_1 \,;\, P_2)^{l;}$ by operational rule (23), and $wires((P_1 \,;\, P_2)^{l;}) = \{l_1{*}.r\}$ by (94).

We apply the converse activity lemma, let $P' = (P_1 \,;\, P_2)^{l;}$. Then, we have $wires(P') = (wires(P) - wires((P_1 \,;\, P_2)^{vl;})) \cup wires((P_1 \,;\, P_2)^{l;}) = W'$, as desired.

♀

This concludes the proof that the compilation algorithm for Joy is correct.

# References


[1] W. Bevier and J. Sogaard-Andersen. Mechanically checked proofs of kernel specifications. pages 70–84, Aalborg, Denmark, July 1991. Springer-Verlag.

[2] H. Bisseling, H. Eemers, M. Kamps, and A. Peeters. Designing delay-insensitive circuits. Technical report, Eindhoven University of Technology, 1990.

[3] B. Bloom. *Ready Simulation, Bisimulation, and the Semantics of CCS-Like Languages.* PhD thesis, Massachusetts Institute of Technology, Aug. 1989.

[4] G. Brown. Towards truly delay insensitive circuit realizations of process algebras. In *Proceedings of the Oxford Workshop on Designing Correct Circuits.* Springer-Verlag, September 1990.

[5] E. Brunvand and R. F. Sproull. Translating concurrent communicating programs into delay-insensitive circuits. Technical Report CMU-CS-89-126, Carnegie-Mellon University, 1989.

[6] S. M. Burns and A. J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In *Advanced Research in VLSI: Proceedings of the 5th MIT Conference*, pages 35–50, 1988.

[7] W. A. Clark and C. E. Molnar. Macromodular computer systems. In B. Waxman and R. Stacey, editors, *Biomedical Research Vol. IV*, pages 45–85. Academic, New York, 1974.

[8] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1989.

[9] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1:226–234, 1986.

[10] G. J. Milne. *The Correctness of a Simple Silicon Compiler.*, pages 1–12. North-Holland, 1983.

[11] R. Milner. A modal characterisation of observable machine-behaviour. In E. Astesiano and C. Böhm, editors, *CAAP '81: Trees in Algebra and Programming, 6th Colloquium*, volume 112 of *Lect. Notes in Computer Sci.*, pages 25–34. Springer-Verlag, 1981.

[12] R. Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, New York, 1989.

[13] C. E. Molnar, T.-P. Fang, and F. U. Rosenberger. Synthesis of delay-insensitive modules. In H. Fuchs, editor, *Chapel Hill Conference on VLSI*, pages 67–86, 1985.

[14] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science*, Lect. Notes in Computer Sci., page 261. Springer-Verlag, 1981.

[15] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, Denmark, 1981.

[16] S. Smitt and A. Zwarico. Correct compilation of specifications to deterministic asynchronous circuits. *Formal Methods in System Design*, 7(3):155–226, Nov 1995.

[17] R. Sproull and I. E. Sutherland. *Asynchronous Systems*. Sutherland, Sproull & Associates, 1986.

[18] J. Staunstrup and M. Greenstreet. Atomicity, programs, and hardware. Technical Report ID-TR 198-46, Technical University of Denmark, 1990.

[19] M. J. F. T. S. Anantharaman, E. M. Clarke and B. Mishra. Compiling path expressions into vlsi circuits. *Distributed Computing*, 1:150–166, 1986.

[20] C. H. K. van Berkel, C. Niessen, M. Rem, and R. W. Saeijs. VLSI programming and silicon compilation; a novel approach from philips research. In *ICCD*, 1988.

[21] K. van Berkel. *Handshake Circuits: an intermediary between communicating processes and VLSI.* PhD thesis, Technische Universiteit Eindhoven, 1992.

[22] S. Weber, B. Bloom, and G. Brown. Compiling Joy to silicon: A verified silicon compilation scheme. In T. Knight and J. Savage, editors, *Proceedings of the Advanced Research in VLSI and Parallel Systems Conference*, pages 79–98. 1992.