Pipelined Asynchronous Circuits

Andrew Matthew Lines

June 1995, revised June 1998

This thesis presents a design style for implementing communicating sequential processes (CSP) as quasi delay insensitive asynchronous circuits, based on the compilation method of [1]. Although hand compilation can always yield optimal circuits to a good designer, a restricted approach is suggested which can easily implement circuits with some slack between inputs and outputs. These circuits are fast and versatile building blocks for highly pipelined designs. The first chapter presents the implementation approach for individual cells. The second chapter investigates the time behavior of complex pipelined circuits, with the goal of adding slack where necessary and adjusting transistor sizes to optimize the overall throughput.

1 Pipelined Cells

1.1 Pipelines

A "pipeline" is a linear sequence of buffers where the output of one buffer connects to the input of the next buffer. "Tokens" are sent into the input end of the pipeline, and flow through each buffer to the output end. The tokens remain in first-in-first-out (FIFO) order. For synchronous pipelines, the tokens usually advance through one stage on each clock cycle. For asynchronous pipelines, there is no global clock to synchronize the movement. Instead, each token moves forward down the pipeline when there is an empty cell in front of it. Otherwise, it stalls. (This has exactly the same behavior as cars on a freeway.) The buffer capacity or "slack" of an asynchronous pipeline is the maximum number of tokens that can packed into it without stalling the input end of the pipeline. The "throughput" is the number of tokens per second which pass a given stage in the pipeline. The "forward latency" is the time it takes a given token to travel the length of the pipeline.

1.2 Buffer Reshuffling

A single rail buffer has the CSP specification $*[L;R]$. Using a passive protocol for L and a lazy active protocol for R, the buffer will have the handshaking expansion (HSE):

$$*[[L];\ L^{\mathrm{a}}\uparrow;\ [\neg L];\ L^{\mathrm{a}}\downarrow;\ [\neg R^{\mathrm{a}}];\ R\uparrow;\ [R^{\mathrm{a}}];\ R\downarrow].$$

The environment will perform $*[[\neg L^{a}]; L\uparrow; [L^{a}]; L\downarrow]$ and $*[[R]; R^{a}\uparrow; [\neg R]; R^{a}\downarrow]$. The wait for $[L]$ is interpreted to be the arrival of an input token, and the transition $R\uparrow$ is the beginning of the output token. An array of these buffers preserves the desired FIFO order and properties of a pipeline.

Direct implementation of this HSE will require a state variable to distinguish the first half from the second half, and has too much sequencing per cycle. Instead, it is usually better to reshuffle the waits and events to reduce the amount of sequencing and the number of state variables. We wish to maximize the throughput and minimize the latency of a pipeline.

The first requirement for a valid reshuffling is that the HSE maintain the handshaking protocols on L and R. That is, the projection on the L channel is $*[[L]; L^{a}\uparrow; [\neg L]; L^{a}\downarrow]$ and the projection on the R channel is $*[[\neg R^{a}]; R\uparrow; [R^{a}]; R\downarrow]$. In addition, the number of completed $L\uparrow$ minus the number of completed $R\uparrow$ (the slack of the buffer) must be at least zero. This conserves the number of tokens in the pipeline. Also, since this is a "buffer" and is supposed to introduce some nonzero slack, the $L^{a}\uparrow$ must not wait for the corresponding $[R^{a}]$, or the reshuffling will have zero slack. This is the "constant response time" requirement.

Although these three requirements are sufficient to guarantee a correct implementation, one more is useful. Soon we will expand the L and R channels to encode data. If we were to move the $R\uparrow$ past the corresponding $L^{a}\uparrow$, that data would need to be saved in internal state variables proportional to the number of bits on R or L. It is better to avoid the additional latching.

Given these requirements, there are only nine valid reshufflings:

$$MSFB \equiv *[[\neg R^{a} \wedge L];\ R\uparrow;\ ([R^{a}];\ R\downarrow),\ (L^{a}\uparrow;\ [\neg L];\ L^{a}\downarrow)]$$
$$PCFB \equiv *[[\neg R^{a} \wedge L];\ R\uparrow;\ L^{a}\uparrow;\ ([R^{a}];\ R\downarrow),\ ([\neg L];\ L^{a}\downarrow)]$$
$$PCHB \equiv *[[\neg R^{a} \wedge L];\ R\uparrow;\ L^{a}\uparrow;\ [R^{a}];\ R\downarrow;\ [\neg L];\ L^{a}\downarrow]$$
$$WCHB \equiv *[[\neg R^{a} \wedge L];\ R\uparrow;\ L^{a}\uparrow;\ [R^{a} \wedge \neg L];\ R\downarrow;\ L^{a}\downarrow]$$
$$B1 \equiv *[[\neg R^{a} \wedge L];\ R\uparrow;\ L^{a}\uparrow;\ [R^{a} \wedge \neg L];\ L^{a}\downarrow;\ R\downarrow]$$
$$B2 \equiv *[[\neg R^{a} \wedge L];\ R\uparrow;\ L^{a}\uparrow;\ [\neg L];\ L^{a}\downarrow;\ [R^{a}];\ R\downarrow]$$
$$B3 \equiv *[[\neg R^{a} \wedge L];\ R\uparrow;\ L^{a}\uparrow;\ [\neg L];\ ([R^{a}];\ R\downarrow),\ L^{a}\downarrow]$$
$$B4 \equiv *[[\neg R^{a} \wedge L];\ R\uparrow;\ L^{a}\uparrow;\ [R^{a}];\ R\downarrow,\ ([\neg L];\ L^{a}\downarrow)]$$
$$B5 \equiv *[[\neg R^{a} \wedge L];\ R\uparrow;\ L^{a}\uparrow;\ [R^{a} \wedge \neg L];\ R\downarrow,\ L^{a}\downarrow]$$

It takes two state variables to implement the *MSFB* reshuffling. The *PCFB*, *B1*, *B2*, *B3*, *B4*, and *B5* reshufflings all require one state variable en (short for enable) with $en\downarrow$ inserted after $L^{a}\uparrow$ and $en\uparrow$ inserted before the end.

Are any of these reshufflings obviously inferior? For this thesis we assume that the goal is fewer transistors and faster operation. By that metric, it can be shown that *B3*, *B4*, and *B5* are always inferior to *PCFB*. They all require the same state variable. They produce only a subset of the trace of *PCFB*, with additional unnecessary waits. These waits add extra transistors and slow the circuit down, compared to *PCFB*.

What about *B1* and *B2*? They are also very similar to *PCFB*, except they have more sequencing. However, that extra sequencing simplifies the production rule for $en\uparrow$ to $\neg R \rightarrow en\uparrow$ instead of $\neg R \wedge \neg L^{a} \rightarrow en\uparrow$, in the case of *PCFB*. It is therefore not possible to say that these will always be inferior to *PCFB*. However, due to the extra sequencing and additional transistors elsewhere, we assume that these reshufflings will seldom, if ever, be better than *PCFB*, and will not consider them further.

The *MSFB* has the least possible sequencing of any of these reshufflings. However, it requires two state variables and has more complicated production rules than *PCFB*. Its only possible advantage in speed is that it allows $R\downarrow$ to happen a little earlier. If one counts transitions, it turns out that the next buffer in the pipeline (if it is reshuffled similarly) will not even raise R^{a} until after $L^{\mathrm{a}}\uparrow$ occurs, so this is really not an advantage at all. Therefore, the *MSFB* will not be considered further.

That leaves only three interesting reshufflings, *WCHB*, *PCHB*, and *PCFB*. The names are derived from characteristics of the circuit implementations. *WC* indicates weak-condition logic, *PC* indicates precharge logic. *HB* indicates a halfbuffer (slack $\frac{1}{2}$), and *FB* indicates a fullbuffer (slack 1). In the halfbuffer reshufflings, only every other stage can have token on its output channel, since a token on that channel blocks the previous stage from producing an output token. In practice, each of these three reshufflings seems to be best for certain applications, so they are all useful. With state variables inserted, the three reshufflings are:

$$PCFB \equiv\ *[[\neg R^{\mathrm{a}} \ \wedge\ L];\ R\uparrow;\ L^{\mathrm{a}}\uparrow;\ en\downarrow;\ ([R^{\mathrm{a}}];\ R\downarrow),\ ([\neg L];\ L^{\mathrm{a}}\downarrow);\ en\uparrow]$$
$$PCHB \equiv\ *[[\neg R^{\mathrm{a}} \ \wedge\ L];\ R\uparrow;\ L^{\mathrm{a}}\uparrow;\ [R^{\mathrm{a}}];\ R\downarrow;\ [\neg L];\ L^{\mathrm{a}}\downarrow]$$
$$WCHB \equiv\ *[[\neg R^{\mathrm{a}} \ \wedge\ L];\ R\uparrow;\ L^{\mathrm{a}}\uparrow;\ [R^{\mathrm{a}} \ \wedge\ \neg L];\ R\downarrow;\ L^{\mathrm{a}}\downarrow]$$

1.3 Logic with Buffering

Suppose we wanted to implement a unit with CSP of the form:

$$P \equiv\ *[A?a,\ B?b,\ \ldots\ ;\ X!f(a,b,\ldots),\ Y!g(a,b,\ldots),\ \ldots]$$

On each cycle, P receives some inputs, then sends out functions computed from these inputs. The channels A,B,X, and Y must encode some data. The usual way to do this is to use sets of 1-of-N rails for each channel. For instance, to send two bits, one could use two 1-of-2 rails with one acknowledge, or one 1-of-4 rails with one acknowledge.

As a notational convention, a rail is identified by the channel name with a superscript for the 1-of-N wire which is active, and a subscript for what group of 1-of-N wires it belongs to (if there is more than one group in the channel). The corresponding acknowledge will be the channel name with a "a" superscript, or an "e" superscript if it is used in the inverted sense.

As in the single rail buffer case, we could implement P by expanding each channel communication into a handshaking expansion. Direct implementation of this HSE requires state variables for the a, b variables and more. It would produce an enormously big and slow circuit, so some reshuffling is obviously desired. How should the HSE be reshuffled? The same set of arguments presented in the last section apply, and we conclude that the *PCFB*, *PCHB*, and *WCHB* reshufflings will be the most useful ones.

The correspondence between the single rail "templates" for *PCFB*, *PCHB*, and *WCHB* and a process like P is as follows. The L and L^{a} will represent all the input data and acknowledges. The R and R^{a} will represent all the output data and acknowledges. $[L]$ indicates a wait for the validity of all inputs, and $[\neg L]$ a wait for the neutrality of all inputs. $[\neg R^{\mathrm{a}}]$ indicates a wait for all the output acknowledges to be false, and $[R^{\mathrm{a}}]$ indicates a wait for all the output acknowledges to be true. $L^{\mathrm{a}}\uparrow$ indicates raising all the input acknowledges in parallel, and $L^{\mathrm{a}}\downarrow$ indicates lowering them. $R\uparrow$ means that all the outputs are set to their valid states in parallel. $R\downarrow$ means that all the outputs are set to their neutral states. When $R\uparrow$ occurs, it is meant that particular rails of the outputs are raised depending on which rails of L are true. This expands $R\uparrow$ into a set of exclusive selection statements executing in parallel.

Unfortunately, this simple translation will introduce more sequencing then necessary. Of the various actions which occur in parallel, like setting all the outputs valid ($R\uparrow$), each action might need to wait for only a portion of the preceding guard ($[\neg R^{\mathrm{a}} \wedge L]$). For instance, raising $X^0\uparrow$ or $X^1\uparrow$ needs to check $[\neg Xa]$ but not $[\neg Ya]$. Similarly, the semicolons between actions ($R\uparrow; L^{\mathrm{a}}\uparrow$) might also over sequence. However, this cannot be easily fixed while still using the HSE language. For instance, in the sequence $X\uparrow, Y\uparrow; A^{\mathrm{a}}\uparrow, B^{\mathrm{a}}\uparrow$, it might be necessary for $A^{\mathrm{a}}\uparrow$ to wait for $[X]$ only (if $Y\uparrow$ did not use the value of A) while $B^{\mathrm{a}}\uparrow$ might need to wait for $[X \wedge Y]$. This case could be written as $X\uparrow, Y\uparrow, ([X]; A^{\mathrm{a}}\uparrow), ([X \wedge Y]; B^{\mathrm{a}}\uparrow)$, but this is starting to get a bit illegible, and if the next actions aren't fully sequenced, it gets even worse. In the limit, the HSE just mirrors the actual production rule set (PRS). To skirt the issue of "weak-sequencing" this thesis will just use HSE which might be a bit over sequenced, with the understanding that the unnecessary sequencing will be optimized out in the compilation to production rules. On-going research includes a more formal examination of when sequencing can be eliminated, including an algorithm which automatically produces weakened PRS from a given HSE. However, this is not yet fully developed, and is beyond the scope of this thesis.

The *PCFB* version of a P with dual rail channels would therefore be:

$$
\begin{array}{l}
*[[\neg X^{\mathrm{a}} \wedge f^0(A,B,..) \longrightarrow X^0\uparrow [\!]\, \neg X^{\mathrm{a}} \wedge f^1(A,B,...) \longrightarrow X^1\uparrow], \\
\quad [\neg Y^{\mathrm{a}} \wedge g^0(A,B,..) \longrightarrow Y^0\uparrow [\!]\, \neg Y^{\mathrm{a}} \wedge g^1(A,B,...) \longrightarrow Y^1\uparrow], ...; \\
\quad A^{\mathrm{a}}\uparrow, B^{\mathrm{a}}\uparrow, ...; \\
\quad en\downarrow; \\
\quad [X^{\mathrm{a}} \longrightarrow X^0\downarrow, X^1\downarrow], [Y^{\mathrm{a}} \longrightarrow Y^0\downarrow, Y^1\downarrow], ..., \\
\quad [\neg A^0 \wedge \neg A^1 \longrightarrow A^{\mathrm{a}}\downarrow], [\neg B^0 \wedge \neg B^1 \longrightarrow B^{\mathrm{a}}\downarrow], ...; \\
\quad en\uparrow \\
\;]
\end{array}
$$

In this HSE, the f^0,f^1,g^0, and g^1 are boolean expressions in the data rails of the input channels. They are derived from the f and g of the CSP and indicate the conditions for raising the various data rails of the output channels. Note that each output channel waits only for its own acknowledge, which is less sequenced than a direct translation of the *PCFB* template would be.

In P it is seen that A^{a} and B^{a} tend to switch at about the same time. They could actually be combined into a single AB^{a} which would wait for the conjunction of the guards on A^{a} and B^{a}. Combining the acknowledges tends to reduce the area of the circuit, but might slow it down. The best decision depends on the circumstances.

1.4 Examples of Logic with Buffering

To put the previous section into practice, several CSP processes with the same form as P will be compiled into pipelined circuits. The simplest CSP buffer that encodes data has a dual rail input L, and a dual rail output R. The CSP is $*[L?x; R!x]$. Three HSE reshufflings for this process are:

$$
\begin{aligned}
&\mathit{WCHB\_BUF} \equiv \\
&\qquad *[[\neg R^{\mathrm{a}} \wedge L^0 \longrightarrow R^0{\uparrow} \,[]\, \neg R^{\mathrm{a}} \wedge L^1 \longrightarrow R^1{\uparrow}]; L^{\mathrm{a}}{\uparrow}; \\
&\qquad\quad [R^{\mathrm{a}} \wedge \neg L^0 \wedge \neg L^1 \longrightarrow R^0{\downarrow}, R^1{\downarrow}]; L^{\mathrm{a}}{\downarrow}]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{PCHB\_BUF} \equiv \\
&\qquad *[[\neg R^{\mathrm{a}} \wedge L^0 \longrightarrow R^0{\uparrow} \,[]\, \neg R^{\mathrm{a}} \wedge L^1 \longrightarrow R^1{\uparrow}]; L^{\mathrm{a}}{\uparrow}; \\
&\qquad\quad [R^{\mathrm{a}} \longrightarrow R^0{\downarrow}, R^1{\downarrow}]; [\neg L^0 \wedge \neg L^1 \longrightarrow L^{\mathrm{a}}{\downarrow}]]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{PCFB\_BUF} \equiv \\
&\qquad *[[\neg R^{\mathrm{a}} \wedge L^0 \longrightarrow R^0{\uparrow} \,[]\, \neg R^{\mathrm{a}} \wedge L^1 \longrightarrow R^1{\uparrow}]; L^{\mathrm{a}}{\uparrow}; en{\downarrow}; \\
&\qquad\quad [R^{\mathrm{a}} \longrightarrow R^0{\downarrow}, R^1{\downarrow}], [\neg L^0 \wedge \neg L^1 \longrightarrow L^{\mathrm{a}}{\downarrow}]; en{\uparrow}]
\end{aligned}
$$

After bubble-reshuffling (which suggests using the inverted acknowledges, L^{e} and R^{e}), the production rules for the *WCHB_BUF* follow, and the circuit diagram is shown in Figure 1.

$$
\begin{array}{lcl}
R^{\mathrm{e}} \wedge L^0 & \rightarrow & \overline{R^0}{\downarrow} \\
R^{\mathrm{e}} \wedge L^1 & \rightarrow & \overline{R^1}{\downarrow} \\
\neg\overline{R^0} & \rightarrow & R^0{\uparrow} \\
\neg\overline{R^1} & \rightarrow & R^1{\uparrow} \\
\neg\overline{R^0} \vee \neg\overline{R^1} & \rightarrow & \overline{L^{\mathrm{e}}}{\uparrow} \\
\overline{L^{\mathrm{e}}} & \rightarrow & L^{\mathrm{e}}{\downarrow}
\end{array}
$$

$$
\begin{array}{lcl}
\neg R^{\mathrm{e}} \wedge \neg L^0 & \rightarrow & \overline{R^0}{\uparrow} \\
\neg R^{\mathrm{e}} \wedge \neg L^1 & \rightarrow & \overline{R^1}{\uparrow} \\
\overline{R^0} & \rightarrow & R^0{\downarrow} \\
\overline{R^1} & \rightarrow & R^1{\downarrow} \\
\overline{R^0} \wedge \overline{R^1} & \rightarrow & \overline{L^{\mathrm{e}}}{\downarrow} \\
\neg\overline{L^{\mathrm{e}}} & \rightarrow & L^{\mathrm{e}}{\uparrow}
\end{array}
$$

The other HSE's can be implemented similarly, but they are both somewhat bigger. For this reshuffling, the validity and neutrality of the output data R implies the validity and neutrality of the input data L. Logic which has this property is called "weak-condition". It means that the L does not need to be checked anywhere else, besides in R. The *WCHB* also gets some of its semicolons implemented for free. The semicolon between $L^{\mathrm{a}}{\uparrow}; [R^{\mathrm{a}} \wedge \neg L]$ is implemented by the environment, as is the implicit semicolon at the end of the loop. So it seems that the *WCHB* has some inherent benefits. However, it turns out that although *WCHB* works well for buffers, the "weak-condition" requirement can cause problems with other circuits.

This *WCHB_BUF* bubble-reshuffling has 2 transitions forward latency and 3 transitions "backward" latency (for the path from the right acknowledge to the left acknowledge). Combining these times for the whole handshake yields $2 + 3 + 2 + 3 = 10$ transitions per cycle.

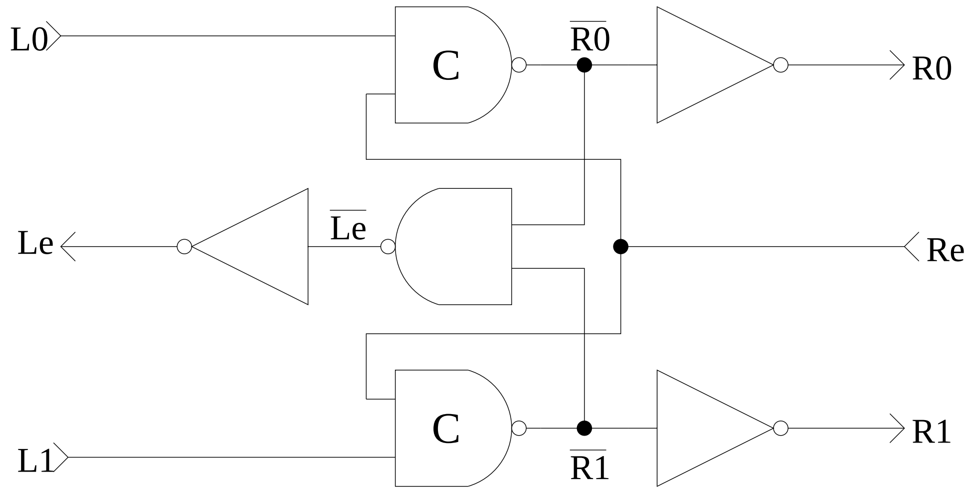

Figure 1: WCHB_BUF

The minimum number of transitions per cycle of any 4-phase buffer is 6. There must be at least 1 transition at each end of the channel, for each phase of the handshake. Due to the inverse monotonicity of CMOS, an oscillator must have an odd number of transitions on each half cycle. Therefore, each half of the handshake requires an odd number of transitions, greater than 2. So there must be at least 3 transitions for each half, or 6 for the whole cycle. Why do we add extra inverters to *WCHB_BUF* to get 10 transitions per cycle? Adding the inverters can actually speed up the throughput, despite the increased transition count, because inverters have high gain. Also, the 6 transition per cycle buffer would invert the senses of the data and acknowledges after every stage, which is highly inconvenient when composing different pipelined cells. As a standard practice, most pipelined logic cells will be done with 2 transitions of forward latency, but more complicated circuits will have 5, 7 or even 9 transitions backward latency, yielding transitions per cycle from 10 to 22 (even numbers only, of course).

Next we consider a fulladder, with the CSP $*[A?a, B?b, C?c; S!XOR(a,b,c), D!MAJ(a,b,c)]$. The A,B,C,S, and D channels are dual rail. The acknowledges for A,B, and C will be combined into a single F^e. We use inverted acknowledges from the start. The three HSE reshufflings are:

$$
\begin{array}{l}
\mathit{WCHB\_FA} \equiv \\
\quad *[[S^e \wedge XOR^0(A,B,C) \longrightarrow S^0\!\uparrow \,[\!]\, S^e \wedge XOR^1(A,B,C) \longrightarrow S^1\!\uparrow], \\
\quad\quad [D^e \wedge MAJ^0(A,B,C) \longrightarrow D^0\!\uparrow \,[\!]\, D^e \wedge MAJ^1(A,B,C) \longrightarrow D^1\!\uparrow]; \\
\quad\quad F^e\!\downarrow; \\
\quad\quad [\neg S^e \wedge \neg A^0 \wedge \neg A^1 \wedge \neg C^0 \longrightarrow S^0\!\downarrow, S^1\!\downarrow], \\
\quad\quad [\neg D^e \wedge \neg B^0 \wedge \neg B^1 \wedge \neg C^1 \longrightarrow D^0\!\downarrow, D^1\!\downarrow]; \\
\quad\quad F^e\!\uparrow \\
\quad]
\end{array}
$$

$PCHB\_FA \equiv$

$$
\begin{array}{l}
*[[S^e \wedge XOR^0(A,B,C) \longrightarrow S^0\!\uparrow [\!] S^e \wedge XOR^1(A,B,C) \longrightarrow S^1\!\uparrow],\\
\quad [D^e \wedge MAJ^0(A,B,C) \longrightarrow D^0\!\uparrow [\!] D^e \wedge MAJ^1(A,B,C) \longrightarrow D^1\!\uparrow];\\
\quad F^e\!\downarrow;\\
\quad [\neg S^e \longrightarrow S^0\!\downarrow, S^1\!\downarrow],\\
\quad [\neg D^e \longrightarrow D^0\!\downarrow, D^1\!\downarrow];\\
\quad [\neg A^0 \wedge \neg A^1 \wedge \neg B^0 \wedge \neg B^1 \wedge \neg C^0 \wedge \neg C^1 \longrightarrow F^e\!\uparrow]\\
\;]
\end{array}
$$

$PCFB\_FA \equiv$

$$
\begin{array}{l}
*[[S^e \wedge XOR^0(A,B,C) \longrightarrow S^0\!\uparrow [\!] S^e \wedge XOR^1(A,B,C) \longrightarrow S^1\!\uparrow],\\
\quad [D^e \wedge MAJ^0(A,B,C) \longrightarrow D^0\!\uparrow [\!] D^e \wedge MAJ^1(A,B,C) \longrightarrow D^1\!\uparrow];\\
\quad F^e\!\downarrow;\\
\quad en\!\downarrow;\\
\quad [\neg S^e \longrightarrow S^0\!\downarrow, S^1\!\downarrow],\\
\quad [\neg D^e \longrightarrow D^0\!\downarrow, D^1\!\downarrow],\\
\quad [\neg A^0 \wedge \neg A^1 \wedge \neg B^0 \wedge \neg B^1 \wedge \neg C^0 \wedge \neg C^1 \longrightarrow F^e\!\uparrow];\\
\quad en\!\uparrow\\
\;]
\end{array}
$$

In the *WCHB_FA*, the validity of the outputs S and D implies the validity of the inputs, because the S must check all of A,B, and C. The test for the neutrality of the inputs is split between $S\!\downarrow$ and $D\!\downarrow$. This works as long as both $S\!\downarrow$ and $D\!\downarrow$ check at least one input's neutrality completely, and if both rails of S and D wait for the same expression. In both *PCHB_FA* and *PCFB_FA*, the expression for the neutrality of the inputs is obviously too large to implement as a single production rule. Instead, the neutrality test must be decomposed into several operators. The usual decomposition is nor gates for each dual rail input, followed by a 3-input c-element. $F^e\!\downarrow$ must now wait for the validity of the inputs just to acknowledge the internal transitions. However, this means the logic for S and D no longer needs to fully check validity of the inputs; it is not required to be weak-condition.

The bubble-reshuffled and decomposed production rules for *WCHB_FA* are:

$$
\begin{array}{lcl}
S^e \wedge XOR^0(A,B,C) & \rightarrow & \overline{S^0}\!\downarrow\\
S^e \wedge XOR^1(A,B,C) & \rightarrow & \overline{S^1}\!\downarrow\\
D^e \wedge MAJ^0(A,B,C) & \rightarrow & \overline{D^0}\!\downarrow\\
D^e \wedge MAJ^1(A,B,C) & \rightarrow & \overline{D^1}\!\downarrow\\
\neg\overline{S^0} & \rightarrow & S^0\!\uparrow\\
\neg\overline{S^1} & \rightarrow & S^1\!\uparrow\\
\neg\overline{D^0} & \rightarrow & D^0\!\uparrow\\
\neg\overline{D^1} & \rightarrow & D^1\!\uparrow\\
(\neg\overline{S^0} \vee \neg\overline{S^1}) \wedge (\neg\overline{D^0} \vee \neg\overline{D^1}) & \rightarrow & \overline{F^e}\!\uparrow\\
\overline{F^e} & \rightarrow & F^e\!\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
\neg S^{\mathrm{e}} \wedge \neg A^0 \wedge \neg A^1 \wedge \neg C^0 & \rightarrow & \overline{S^0}\uparrow \\
\neg S^{\mathrm{e}} \wedge \neg A^0 \wedge \neg A^1 \wedge \neg C^0 & \rightarrow & \overline{S^1}\uparrow \\
\neg D^{\mathrm{e}} \wedge \neg B^0 \wedge \neg B^1 \wedge \neg C^1 & \rightarrow & \overline{D^0}\uparrow \\
\neg D^{\mathrm{e}} \wedge \neg B^0 \wedge \neg B^1 \wedge \neg C^1 & \rightarrow & \overline{D^1}\uparrow \\
\overline{S^0} & \rightarrow & S^0\downarrow \\
\overline{S^1} & \rightarrow & S^1\downarrow \\
\overline{D^0} & \rightarrow & D^0\downarrow \\
\overline{D^1} & \rightarrow & D^1\downarrow \\
\overline{S^0} \wedge \overline{S^1} \wedge \overline{D^0} \wedge \overline{D^1} & \rightarrow & \overline{F^{\mathrm{e}}}\downarrow \\
\neg \overline{F^{\mathrm{e}}} & \rightarrow & F^{\mathrm{e}}\uparrow
\end{array}
$$

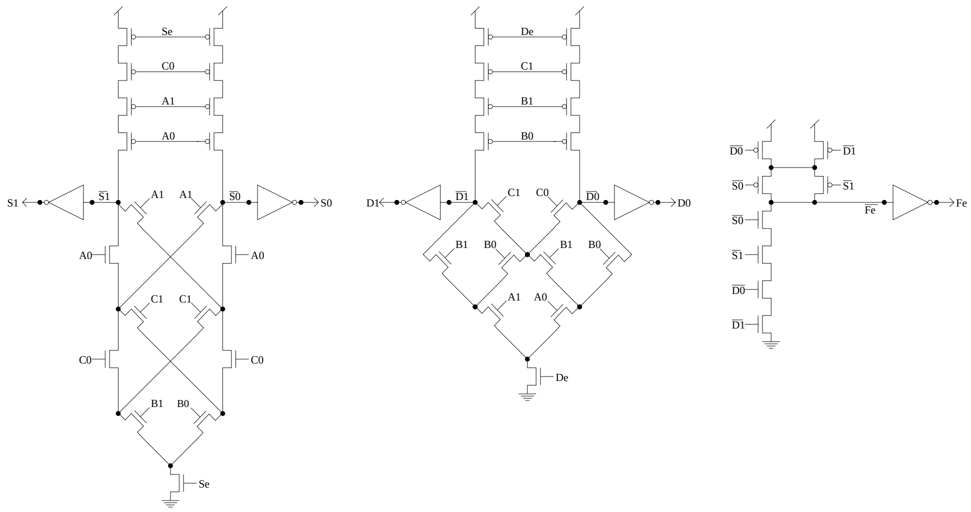

Figure 2: WCHB_FA

The circuit diagram is shown in Figure 2. The pull-up logic for $\overline{S^0}$, $\overline{S^1}$, $\overline{D^0}$, and $\overline{D^1}$ has 4 p-transistors in series, which is quite weak, due to the lower mobility of holes. Other *WCHB* circuits can be even worse. Since all the inputs are checked for neutrality before the outputs reset, a process with three inputs and only one output would end up with 7 p-transistors in series to reset that output. The solution is to use the “precharge-logic” reshufflings, *PCHB_FA* or *PCFB_FA*. These test the neutrality of the inputs in a different place, which is much more easily decomposed into manageable gates, and does not slow the forward latency. The *PCHB_FA* reshuffling has the production rules:

$$
\begin{array}{lcl}
A^0 \vee A^1 & \rightarrow & \overline{A^{\mathrm{v}}}\downarrow \\
B^0 \vee B^1 & \rightarrow & \overline{B^{\mathrm{v}}}\downarrow \\
C^0 \vee C^1 & \rightarrow & \overline{C^{\mathrm{v}}}\downarrow \\
F^{\mathrm{e}} \wedge S^{\mathrm{e}} \wedge XOR^0(A,B,C) & \rightarrow & \overline{S^0}\downarrow \\
F^{\mathrm{e}} \wedge S^{\mathrm{e}} \wedge XOR^1(A,B,C) & \rightarrow & \overline{S^1}\downarrow \\
F^{\mathrm{e}} \wedge D^{\mathrm{e}} \wedge MAJ^0(A,B,C) & \rightarrow & \overline{D^0}\downarrow \\
F^{\mathrm{e}} \wedge D^{\mathrm{e}} \wedge MAJ^1(A,B,C) & \rightarrow & \overline{D^1}\downarrow \\
\neg\overline{S^0} & \rightarrow & S^0\uparrow \\
\neg\overline{S^1} & \rightarrow & S^1\uparrow \\
\neg\overline{D^0} & \rightarrow & D^0\uparrow \\
\neg\overline{D^1} & \rightarrow & D^1\uparrow \\
\neg\overline{A^{\mathrm{v}}} \wedge \neg\overline{B^{\mathrm{v}}} \wedge \neg\overline{C^{\mathrm{v}}} & \rightarrow & ABC^{\mathrm{v}}\uparrow \\
\neg\overline{S^0} \vee \neg\overline{S^1} & \rightarrow & S^{\mathrm{v}}\uparrow \\
\neg\overline{D^0} \vee \neg\overline{D^1} & \rightarrow & D^{\mathrm{v}}\uparrow \\
S^{\mathrm{v}} \wedge D^{\mathrm{v}} \wedge ABC^{\mathrm{v}} & \rightarrow & F^{\mathrm{e}}\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
\neg A^0 \wedge \neg A^1 & \rightarrow & \overline{A^{\mathrm{v}}}\uparrow \\
\neg B^0 \wedge \neg B^1 & \rightarrow & \overline{B^{\mathrm{v}}}\uparrow \\
\neg C^0 \wedge \neg C^1 & \rightarrow & \overline{C^{\mathrm{v}}}\uparrow \\
\neg S^{\mathrm{e}} \wedge \neg F^{\mathrm{e}} & \rightarrow & \overline{S^0}\uparrow \\
\neg S^{\mathrm{e}} \wedge \neg F^{\mathrm{e}} & \rightarrow & \overline{S^1}\uparrow \\
\neg D^{\mathrm{e}} \wedge \neg F^{\mathrm{e}} & \rightarrow & \overline{D^0}\uparrow \\
\neg D^{\mathrm{e}} \wedge \neg F^{\mathrm{e}} & \rightarrow & \overline{D^1}\uparrow \\
\overline{S^0} & \rightarrow & S^0\downarrow \\
\overline{S^1} & \rightarrow & S^1\downarrow \\
\overline{D^0} & \rightarrow & D^0\downarrow \\
\overline{D^1} & \rightarrow & D^1\downarrow \\
\overline{A^{\mathrm{v}}} \wedge \overline{B^{\mathrm{v}}} \wedge \overline{C^{\mathrm{v}}} & \rightarrow & ABC^{\mathrm{v}}\downarrow \\
\overline{S^0} \wedge \overline{S^1} & \rightarrow & S^{\mathrm{v}}\downarrow \\
\overline{D^0} \wedge \overline{D^1} & \rightarrow & D^{\mathrm{v}}\downarrow \\
\neg S^{\mathrm{v}} \wedge \neg D^{\mathrm{v}} \wedge \neg ABC^{\mathrm{v}} & \rightarrow & F^{\mathrm{e}}\uparrow
\end{array}
$$

This circuit can be made faster by adding two inverters to F^{e} and then two more to produce the F^{e} used internally (which is now called en). This circuit is shown in Figure 3. A $PCFB\_FA$ reshuffling would have only slightly different production rules:

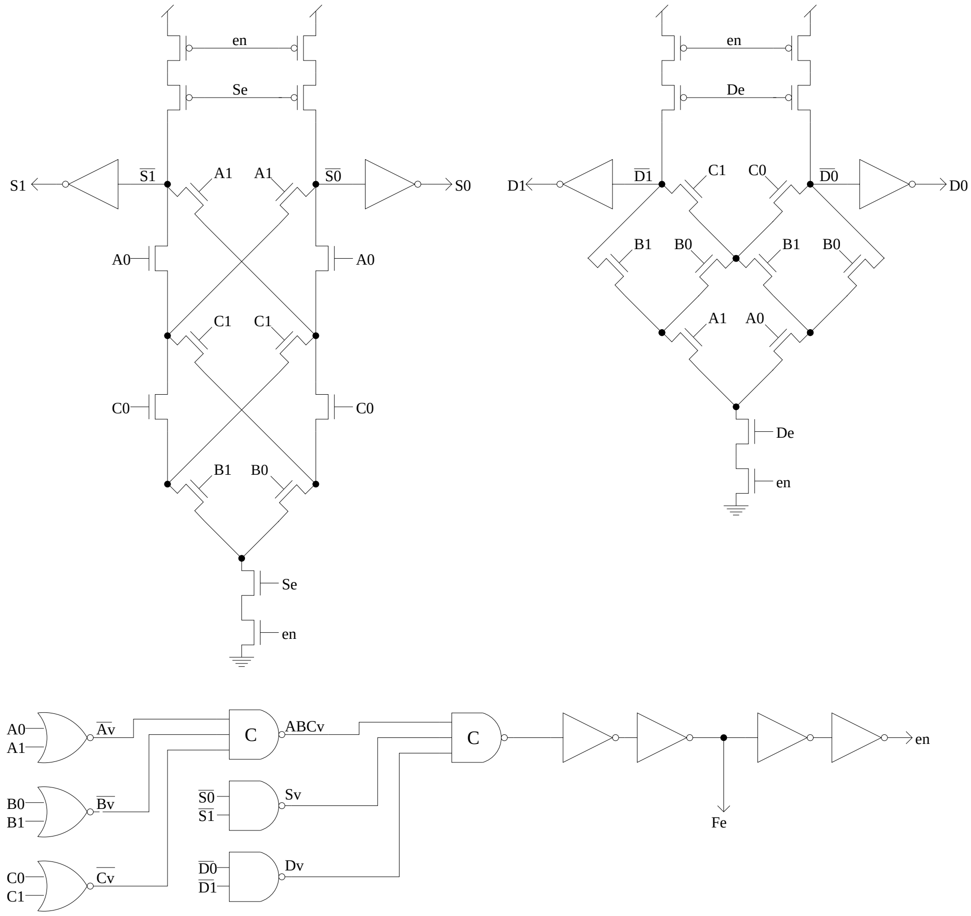

Figure 3: PCHB_FA

$$
\begin{array}{lcl}
A^0 \vee A^1 & \rightarrow & \overline{A^{\mathrm{v}}}\downarrow \\
B^0 \vee B^1 & \rightarrow & \overline{B^{\mathrm{v}}}\downarrow \\
C^0 \vee C^1 & \rightarrow & \overline{C^{\mathrm{v}}}\downarrow \\
en \wedge S^{\mathrm{e}} \wedge XOR^0(A,B,C) & \rightarrow & \overline{S^0}\downarrow \\
en \wedge S^{\mathrm{e}} \wedge XOR^1(A,B,C) & \rightarrow & \overline{S^1}\downarrow \\
en \wedge D^{\mathrm{e}} \wedge MAJ^0(A,B,C) & \rightarrow & \overline{D^0}\downarrow \\
en \wedge D^{\mathrm{e}} \wedge MAJ^1(A,B,C) & \rightarrow & \overline{D^1}\downarrow \\
\neg\overline{S^0} & \rightarrow & S^0\uparrow \\
\neg\overline{S^1} & \rightarrow & S^1\uparrow \\
\neg\overline{D^0} & \rightarrow & D^0\uparrow \\
\neg\overline{D^1} & \rightarrow & D^1\uparrow \\
\neg\overline{A^{\mathrm{v}}} \wedge \neg\overline{B^{\mathrm{v}}} \wedge \neg\overline{C^{\mathrm{v}}} & \rightarrow & ABC^{\mathrm{v}}\uparrow \\
\neg\overline{S^0} \vee \neg\overline{S^1} & \rightarrow & S^{\mathrm{v}}\uparrow \\
\neg\overline{D^0} \vee \neg\overline{D^1} & \rightarrow & D^{\mathrm{v}}\uparrow \\
en \wedge S^{\mathrm{v}} \wedge D^{\mathrm{v}} \wedge ABC^{\mathrm{v}} & \rightarrow & F^{\mathrm{e}}\downarrow \\
S^{\mathrm{v}} \wedge D^{\mathrm{v}} & \rightarrow & \overline{SD^{\mathrm{v}}}\downarrow \\
\neg F^{\mathrm{e}} \wedge \neg\overline{SD^{\mathrm{v}}} & \rightarrow & \overline{en}\uparrow \\
\overline{en} & \rightarrow & en\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
\neg A^0 \wedge \neg A^1 & \rightarrow & \overline{A^{\mathrm{v}}}\uparrow \\
\neg B^0 \wedge \neg B^1 & \rightarrow & \overline{B^{\mathrm{v}}}\uparrow \\
\neg C^0 \wedge \neg C^1 & \rightarrow & \overline{C^{\mathrm{v}}}\uparrow \\
\neg S^{\mathrm{e}} \wedge \neg F^{\mathrm{e}} & \rightarrow & \overline{S^0}\uparrow \\
\neg S^{\mathrm{e}} \wedge \neg F^{\mathrm{e}} & \rightarrow & \overline{S^1}\uparrow \\
\neg D^{\mathrm{e}} \wedge \neg F^{\mathrm{e}} & \rightarrow & \overline{D^0}\uparrow \\
\neg D^{\mathrm{e}} \wedge \neg F^{\mathrm{e}} & \rightarrow & \overline{D^1}\uparrow \\
\overline{S^0} & \rightarrow & S^0\downarrow \\
\overline{S^1} & \rightarrow & S^1\downarrow \\
\overline{D^0} & \rightarrow & D^0\downarrow \\
\overline{D^1} & \rightarrow & D^1\downarrow \\
\overline{A^{\mathrm{v}}} \wedge \overline{B^{\mathrm{v}}} \wedge \overline{C^{\mathrm{v}}} & \rightarrow & ABC^{\mathrm{v}}\downarrow \\
\overline{S^0} \wedge \overline{S^1} & \rightarrow & S^{\mathrm{v}}\downarrow \\
\overline{D^0} \wedge \overline{D^1} & \rightarrow & D^{\mathrm{v}}\downarrow \\
\neg en \wedge \neg ABC^{\mathrm{v}} & \rightarrow & F^{\mathrm{e}}\uparrow \\
\neg S^{\mathrm{v}} \wedge \neg D^{\mathrm{v}} & \rightarrow & \overline{SD^{\mathrm{v}}}\uparrow \\
F^{\mathrm{e}} \wedge \overline{SD^{\mathrm{v}}} & \rightarrow & \overline{en}\downarrow \\
\neg \overline{en} & \rightarrow & en\uparrow
\end{array}
$$

Of the three fulladder reshufflings, which one is best? The *WCHB_FA* has only 10 transitions per cycle, while the *PCHB_FA* has 14 and the *PCFB_FA* has 12 (7 on the setting phase, but 5 on the resetting phase, since the L and R handshakes reset in parallel). Although the *WCHB_FA* has fewer transistors, to make it reasonably fast, the 4 p-transistors in series must be made very large. Despite the lower transition count of the *WCHB_FA*, both *PCHB_FA* and *PCFB_FA* are substantially faster in throughput and latency. *PCFB_FA* is the fastest of all, since it relies heavily on n-transistors and saves 2 transitions on the reset phase. However *PCFB_FA* is a bit larger than *PCHB_FA*, due to the extra state variable en and the extra completion $\overline{SD^{\mathrm{v}}}$. If the speed of the fulladder is not critical, the *PCHB_FA* seems to be the best choice.

In general, the *WCHB* reshuffling tends to be best only for buffers and copies ($[L?x; R!x, S!x]$). The *PCHB* is the workhorse for most applications; it is both small and fast. When exceptional speed is called for, the *PCFB* dominates. It is also especially good at completing 1-of-N codes where N is very large, since the completion can be done by a circuit which looks like a tied-or pulldown, as opposed to many stages of combinational logic. The reshufflings can actually be mixed together, with each channel in the cell using a different one. This is most commonly useful when a cell computes on some inputs using *PCHB*, but also copies some inputs directly to outputs using *WCHB*. In this case, the neutrality detection for the *WCHB* outputs is only one p-gate, which is no worse than an extra en gate.

Another common class of logic circuits use shared control inputs to process multi-bit words. This is not really any different from a fulladder. The control is just another input, which happens to have a large fanout to many output channels. Since the outputs only sparsely depend on the inputs (usually with a bit to bit correspondence), the number of gates in series in the logic doesn't become prohibitive. However, if the number of bits is large (like 32) the completion of all the inputs and outputs will take many stages in a c-element tree, which adds to the cycle time, as does the load on the broadcast of the control data. To make high throughput datapath logic, it is better to break the datapath up into manageable chunks (perhaps 4 or 8 bits) , and send buffered copies of the control tokens to each chunk. This cuts down the cycle time, but doesn't change the high-level meaning, except to introduce extra slack.

1.5 Conditionally producing outputs

Although the cells discussed in the previous section can be shown to be Turing complete (they can be turned into a VonNeumann state machine, with some outputs fed back through buffers to store state), they are clearly inefficient for many applications. A very useful extension is the ability to skip a communication on a channel on a given cycle. This turns out to require only a few minor modifications to the scheme as presented so far.

Suppose the process completes at most one communication per cycle on the outputs, but always receives all its inputs. The CSP would be:

$$
\begin{array}{ll}
\mathit{P1} \equiv & *[A?a, B?b, ...; \\
& \quad [\mathit{do\_x}(a,b,...) \longrightarrow X!f(a,b...) [] \neg \mathit{do\_x}(a,b,...) \longrightarrow \mathit{skip}], \\
& \quad [\mathit{do\_y}(a,b,...) \longrightarrow Y!g(a,b...) [] \neg \mathit{do\_y}(a,b,...) \longrightarrow \mathit{skip}], ... \\
&]
\end{array}
$$

As usual, we can reshuffle this like *WCHB*, *PCHB*, or *PCFB*. The selection statements for the outputs expand into exclusive selections for setting the output rails, plus a new case for producing no output at all on the channel. A dual-rail version of *P1* with a *PCFB* reshuffling is:

$$
\begin{array}{l}
*[[\mathit{do\_x}(A,B,...) \wedge \neg X^{\mathrm{a}} \wedge f^0(A,B,..) \longrightarrow X^0\uparrow \\
\quad [] \mathit{do\_x}(A,B,...) \wedge \neg X^{\mathrm{a}} \wedge f^1(A,B,...) \longrightarrow X^1\uparrow \\
\quad [] \neg \mathit{do\_x}(A,B,...) \longrightarrow \mathit{skip}], \\
\quad [\mathit{do\_y}(A,B,...) \wedge \neg Y^{\mathrm{a}} \wedge g^0(A,B,..) \longrightarrow Y^0\uparrow \\
\quad [] \mathit{do\_y}(A,B,...) \wedge \neg Y^{\mathrm{a}} \wedge g^1(A,B,...) \longrightarrow Y^1\uparrow \\
\quad [] \neg \mathit{do\_y}(A,B,...) \longrightarrow \mathit{skip}], ...; \\
\quad A^{\mathrm{a}}\uparrow, B^{\mathrm{a}}\uparrow, ...; \\
\quad en\downarrow; \\
\quad [X^{\mathrm{a}} \vee \neg X^0 \wedge \neg X^1 \longrightarrow X^0\downarrow, X^1\downarrow], [Y^{\mathrm{a}} \vee \neg Y^0 \wedge \neg Y^1 \longrightarrow Y^0\downarrow, Y^1\downarrow], ..., \\
\quad [\neg A^0 \wedge \neg A^1 \longrightarrow A^{\mathrm{a}}\downarrow], [\neg B^0 \wedge \neg B^1 \longrightarrow B^{\mathrm{a}}\downarrow], ...; \\
\quad en\uparrow \\
\ ]
\end{array}
$$

Note that the resetting of the output channels X and Y must accommodate the cases when those channels weren't used. Since they produced no outputs, they must not wait for the acknowledges. Adding in the $\neg X^0 \wedge \neg X^1$ terms will allow the wait to be completed vacuously. It doesn't actually generate any production rules. This HSE can be compiled into production rules, but there are some tricky details.

An interesting choice arises from the use of the *skip*. A *skip* causes no visible change in state, so the next statements in sequence ($A^{\mathrm{a}}\uparrow, B^{\mathrm{a}}\uparrow, ...$) must actually look directly at the boolean expression for $\neg \mathit{do\_x}(A,B,...)$ and $\neg \mathit{do\_y}(A,B,...)$ in addition to the output rails X^0, X^1, Y^0, Y^1.

The completion condition for setting the outputs would be $en \wedge (X^0 \vee X^1 \vee \neg do\_x(A, B, ...)) \wedge (Y^0 \vee Y^1 \vee \neg do\_y(A, B, ...))$. However, this expression cannot be used directly in the guards for $A^a\uparrow$ and $B^a\uparrow$, since if one fired first, it could destabilize the other. (This would work if A^a and B^a were combined into one acknowledge.)

A better approach is to introduce a new variable to represent the $\neg do\_x$ and $\neg do\_y$ cases. Suppose we replaced the *skip*'s with $no\_x\uparrow$ and $no\_y\uparrow$, respectively, and added $no\_x\downarrow$ to $X^0\downarrow, X^1\downarrow$ and $no\_y\downarrow$ to $Y^0\downarrow, Y^1\downarrow$. Now the production rules are simply produced as if X and Y were 1-of-3 channels instead of 1-of-2, except the extra rail doesn't check the right acknowledge, or, in fact, leave the cell.

Finally, there are many cases were some expression of the outputs is sufficient to produce the output completion expression without reference to the inputs. For instance, if one input is used to decide if a certain output is used, but is also copied to another output, the copied output could be used to check the completion of the optional output. Similarly, if two output channels are used exclusively, such that one or the other will be used each cycle, the completion for both is just the or of each one's completion.

To put this discussion into practice, we will implement a *split*, a fundamental routing process which uses one control input to route a data input to one of two output channels. The simple one-bit CSP is $*[S?s, L?x;\ [\neg s \rightarrow A!x \,[]\, s \rightarrow B!x]]$. The *PCHB* reshuffling is:

$$
\begin{array}{l}
PCHB\_SPLIT \equiv \\
\quad *[[A^e \wedge S^0 \wedge L^0 \longrightarrow A^0\uparrow \,[]\, A^e \wedge S^0 \wedge L^1 \longrightarrow A^1\uparrow \,[]\, S^1 \longrightarrow skip], \\
\quad\quad [B^e \wedge S^1 \wedge L^0 \longrightarrow B^0\uparrow \,[]\, B^e \wedge S^1 \wedge L^1 \longrightarrow B^1\uparrow \,[]\, S^0 \longrightarrow skip]; \\
\quad\quad SL^e\downarrow; \\
\quad\quad [\neg A^e \vee \neg A^0 \wedge \neg A^1 \longrightarrow A^0\downarrow, A^1\downarrow], \\
\quad\quad [\neg B^e \vee \neg B^0 \wedge \neg B^1 \longrightarrow B^0\downarrow, B^1\downarrow]; \\
\quad\quad SL^e\uparrow \\
\quad]
\end{array}
$$

To produce the production rules, we first notice that the first two selection statements are known to be finished when $A^0 \vee A^1 \vee B^0 \vee B^1$. Hence, this will be used as the guard for $SL^e\downarrow$. The bubble-reshuffled production rules are:

$$
\begin{array}{lcl}
S^0 \vee S^1 & \rightarrow & \overline{S^v}\downarrow \\
L^0 \vee L^1 & \rightarrow & \overline{L^v}\downarrow \\
SL^e \wedge A^e \wedge S^0 \wedge L^0 & \rightarrow & \overline{A^0}\downarrow \\
SL^e \wedge A^e \wedge S^0 \wedge L^1 & \rightarrow & \overline{A^1}\downarrow \\
SL^e \wedge A^e \wedge S^1 \wedge L^0 & \rightarrow & \overline{B^0}\downarrow \\
SL^e \wedge A^e \wedge S^1 \wedge L^1 & \rightarrow & \overline{B^1}\downarrow \\
\neg\overline{A^0} & \rightarrow & A^0\uparrow \\
\neg\overline{A^1} & \rightarrow & A^1\uparrow \\
\neg\overline{B^0} & \rightarrow & B^0\uparrow \\
\neg\overline{B^1} & \rightarrow & B^1\uparrow \\
\neg\overline{S^v} \wedge \neg\overline{L^v} & \rightarrow & SL^v\uparrow \\
\neg\overline{A^0} \vee \neg\overline{A^1} \vee \neg\overline{B^0} \vee \neg\overline{B^1} & \rightarrow & AB^v\uparrow \\
AB^v \wedge SL^v & \rightarrow & SL^e\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
\neg S^0 \wedge \neg S^1 & \rightarrow & \overline{S^{\mathrm{v}}}\uparrow \\
\neg L^0 \wedge \neg L^1 & \rightarrow & \overline{L^{\mathrm{v}}}\uparrow \\
\neg SL^{\mathrm{e}} \wedge \neg A^{\mathrm{e}} & \rightarrow & \overline{A^0}\uparrow \\
\neg SL^{\mathrm{e}} \wedge \neg A^{\mathrm{e}} & \rightarrow & \overline{A^1}\uparrow \\
\neg SL^{\mathrm{e}} \wedge \neg B^{\mathrm{e}} & \rightarrow & \overline{B^0}\uparrow \\
\neg SL^{\mathrm{e}} \wedge \neg B^{\mathrm{e}} & \rightarrow & \overline{B^1}\uparrow \\
\overline{A^0} & \rightarrow & A^0\downarrow \\
\overline{A^1} & \rightarrow & A^1\downarrow \\
\overline{B^0} & \rightarrow & B^0\downarrow \\
\overline{B^1} & \rightarrow & B^1\downarrow \\
\overline{S^{\mathrm{v}}} \wedge \overline{L^{\mathrm{v}}} & \rightarrow & SL^{\mathrm{v}}\downarrow \\
\overline{A^0} \wedge \overline{A^1} \wedge \overline{B^0} \wedge \overline{B^1} & \rightarrow & AB^{\mathrm{v}}\downarrow \\
\neg AB^{\mathrm{v}} \wedge \neg SL^{\mathrm{v}} & \rightarrow & SL^{\mathrm{e}}\uparrow
\end{array}
$$

The circuit is shown in Figure 4.

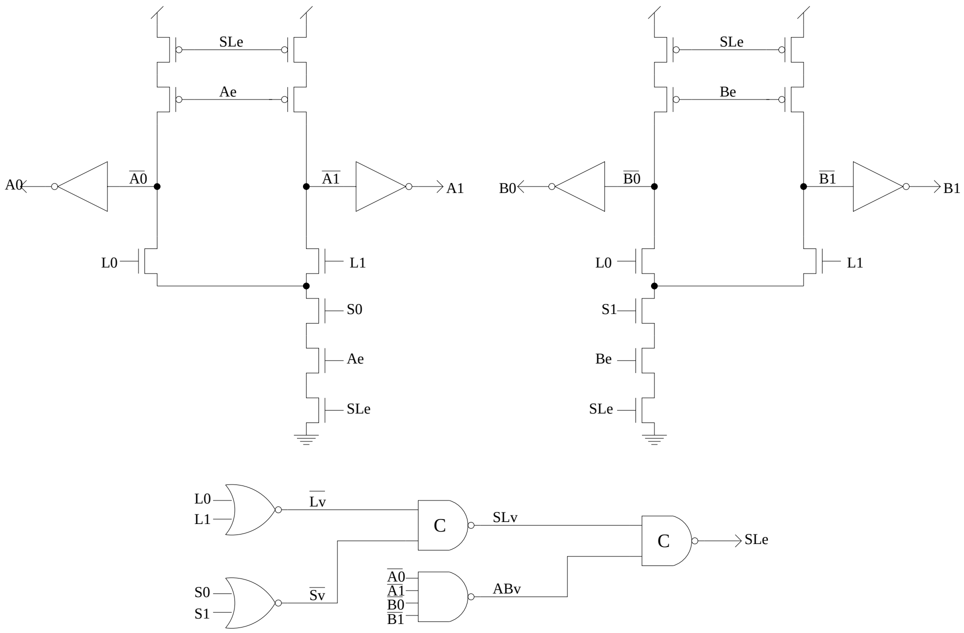

Figure 4: PCHB_SPLIT

1.6 Conditionally reading inputs

It is also highly useful to be able to conditionally read inputs. Normally the condition is read in on a separate unconditional channel, but in general it could be any expression of the rails of the inputs. A CSP template for type of cell this would be:

$$
\begin{aligned}
P2 \equiv\ & *\texttt{[}\texttt{[}do\_a(\overline{A},\overline{B}) \longrightarrow A?a \,\texttt{[]}\, no\_a(\overline{A},\overline{B}) \longrightarrow a := \text{"}unused\text{"}\texttt{]},\\
& \quad \texttt{[}do\_b(\overline{A},\overline{B}) \longrightarrow B?b \,\texttt{[]}\, no\_b(\overline{A},\overline{B}) \longrightarrow b := \text{"}unused\text{"}\texttt{]}, ...;\\
& \quad X!f(a,b...),\ Y!g(a,b...), ...\\
& \texttt{]}
\end{aligned}
$$

The $\overline{A}$ in this context refers to a probe of the value of A, not just its availability. This is not standard in CSP, but is a useful extension which is easily implemented in HSE. Basically, the booleans for *do_a*, *do_b*, *no_a*, and *no_b* may inspect the rails of A and B in order to decide whether to actually receive from the channels. The selection statements will suspend until either *do_a* or *no_a* are true. These expressions are required to be stable; that is, as additional inputs show up, they may not become false as a result.

For the HSE, instead of assigning *"unused"* to an internal variable, the f and g expressions will examine the inputs directly. The results of the *do_a*/*no_a* and *do_b*/*no_b* expressions must be latched into internal variables u and v, so that A and B may be acknowledged in parallel without destabilizing the guards of *do_a* and the like. The *PCFB* version of the HSE is:

$$
\begin{aligned}
& u^0\!\downarrow, u^1\!\downarrow, v^0\!\downarrow, v^1\!\downarrow, ...;\\
& *\texttt{[}\texttt{[}f^0(A,B,...) \longrightarrow X^0\!\uparrow \texttt{[]} f^1(A,B,...) \longrightarrow X^1\!\uparrow\texttt{]},\\
& \quad \texttt{[}g^0(A,B,...) \longrightarrow Y^0\!\uparrow \texttt{[]} g^1(A,B,...) \longrightarrow Y^1\!\uparrow\texttt{]}, ...,\\
& \quad \texttt{[}do\_a(A,B) \longrightarrow u^1\!\uparrow \texttt{[]} no\_a(A,B) \longrightarrow u^0\!\uparrow\texttt{]},\\
& \quad \texttt{[}do\_b(A,B) \longrightarrow v^1\!\uparrow \texttt{[]} no\_b(A,B) \longrightarrow v^0\!\uparrow\texttt{]}, ...;\\
& \quad \texttt{[}u^1 \longrightarrow A^{\mathrm{a}}\!\uparrow \texttt{[]} u^0 \longrightarrow skip\texttt{]},\\
& \quad \texttt{[}v^1 \longrightarrow B^{\mathrm{a}}\!\uparrow \texttt{[]} v^0 \longrightarrow skip\texttt{]}, ...;\\
& \quad en\!\downarrow;\\
& \quad \texttt{[}X^{\mathrm{a}} \longrightarrow X^0\!\downarrow, X^1\!\downarrow\texttt{]}, \texttt{[}Y^{\mathrm{a}} \longrightarrow Y^0\!\downarrow, Y^1\!\downarrow\texttt{]}, ...,\\
& \quad (u^0\!\downarrow, u^1\!\downarrow;\ \ \texttt{[}\neg A^0 \wedge \neg A^1 \vee \neg A^{\mathrm{a}} \longrightarrow A^{\mathrm{a}}\!\downarrow\texttt{]}),\\
& \quad (v^0\!\downarrow, v^1\!\downarrow;\ \ \texttt{[}\neg B^0 \wedge \neg B^1 \vee \neg B^{\mathrm{a}} \longrightarrow B^{\mathrm{a}}\!\downarrow\texttt{]}), ...;\\
& \quad en\!\uparrow\\
& \texttt{]}
\end{aligned}
$$

Similarly to the conditional output HSE, the guards for $A^{\mathrm{a}}\!\downarrow$ and $B^{\mathrm{a}}\!\downarrow$ are weakened to allow the vacuous case. Also, the *skip* again poses a problem, since it makes no change in the state. However, with the u^0 and v^0 variables it is possible to infer the skip and generate the correct guard for en. On the reset phase, the u and v must return to the neutral state. There are several places to put this, but the symmetric placement which sequences them with the $A^{\mathrm{a}}\!\downarrow$ and $B^{\mathrm{a}}\!\downarrow$ simplifies the PRS.

In many cases, this general template can be greatly simplified. For instance, if a set of unconditional inputs completely controls the conditions for reading the others, these can be thought of as the "control" inputs. If raising the acknowledges of the various inputs is sequenced so that the conditional ones precede the control ones, then the variables u and v may be eliminated without causing stability problems. Also in some cases the u and v may be substituted with an expression of the outputs, instead of stored separately.

As a concrete example, we derive the circuit for the *merge* process, which reverses the *split* of the last section by conditionally reading one of two data input channels (A and B) to the single output channel R based on a control input M. The CSP is $*[M?m;\, [\neg m \rightarrow A?x\, []\, m \rightarrow B?x];\, X!x]$. Here we use the simplification of acknowledging the data inputs A and B before the control input M. The $PCHB$ reshuffling is:

$$
\begin{array}{l}
PCHB\_MERGE \equiv \\
\quad *[[X^e \wedge (M^0 \wedge A^0 \vee M^1 \wedge B^0) \longrightarrow X^0\uparrow [] X^e \wedge (M^0 \wedge A^1 \vee M^1 \wedge B^1) \longrightarrow X^1\uparrow], \\
\qquad [M^0 \longrightarrow A^e\downarrow [] M^1 \longrightarrow B^e\downarrow]; \\
\qquad M^e\downarrow; \\
\qquad [\neg X^e \longrightarrow X^0\downarrow, X^1\downarrow], \\
\qquad [\neg A^0 \wedge \neg A^1 \wedge \neg M^0 \vee \neg A^e \longrightarrow A^e\uparrow], \\
\qquad [\neg B^0 \wedge \neg B^1 \wedge \neg M^1 \vee \neg B^e \longrightarrow B^e\uparrow]; \\
\qquad M^e\uparrow \\
\quad]
\end{array}
$$

A subtle simplification used here is to make $A^e\uparrow$ and $B^e\uparrow$ check the corresponding $\neg M^0$ and $\neg M^1$. This reduces the guard condition for $M^e\uparrow$ and makes the reset phase symmetric with the set phase. We do some decomposition to add A^v, B^v, and X^v to do validity and neutrality checks. After bubble-reshuffling, the PRS is:

$$
\begin{array}{lcl}
A^0 \vee A^1 & \rightarrow & \overline{A^v}\downarrow \\
B^0 \vee B^1 & \rightarrow & \overline{B^v}\downarrow \\
\neg\overline{A^v} & \rightarrow & A^v\uparrow \\
\neg\overline{B^v} & \rightarrow & B^v\uparrow \\
M^e \wedge X^e \wedge (M^0 \wedge A^0 \vee M^1 \wedge B^0) & \rightarrow & \overline{X^0}\downarrow \\
M^e \wedge X^e \wedge (M^0 \wedge A^1 \vee M^1 \wedge B^1) & \rightarrow & \overline{X^1}\downarrow \\
\neg\overline{X^0} & \rightarrow & X^0\uparrow \\
\neg\overline{X^1} & \rightarrow & X^1\uparrow \\
\neg\overline{X^0} \vee \neg\overline{X^1} & \rightarrow & X^v\uparrow \\
A^v \wedge M^0 \wedge X^v & \rightarrow & A^e\downarrow \\
B^v \wedge M^1 \wedge X^v & \rightarrow & B^e\downarrow \\
\neg A^e \vee \neg B^e & \rightarrow & \overline{M^e}\uparrow \\
\overline{M^e} & \rightarrow & M^e\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
\neg A^0 \wedge \neg A^1 & \rightarrow & \overline{A^v}\uparrow \\
\neg B^0 \wedge \neg B^1 & \rightarrow & \overline{B^v}\uparrow \\
\overline{A^v} & \rightarrow & A^v\downarrow \\
\overline{B^v} & \rightarrow & B^v\downarrow \\
\neg M^e \wedge \neg X^e & \rightarrow & \overline{X^0}\uparrow \\
\neg M^e \wedge \neg X^e & \rightarrow & \overline{X^1}\uparrow \\
\overline{X^0} & \rightarrow & X^0\downarrow \\
\overline{X^1} & \rightarrow & X^1\downarrow \\
\overline{X^0} \wedge \overline{X^1} & \rightarrow & X^v\downarrow \\
\neg A^v \wedge \neg M^0 \wedge \neg X^v & \rightarrow & A^e\uparrow \\
\neg B^v \wedge \neg M^1 \wedge \neg X^v & \rightarrow & B^e\uparrow \\
A^e \wedge B^e & \rightarrow & \overline{M^e}\downarrow \\
\neg\overline{M^e} & \rightarrow & M^e\uparrow
\end{array}
$$

As usual for *PCHB* reshufflings, most of the work is done in a large network of n transistors. The circuit is shown in Figure 5.

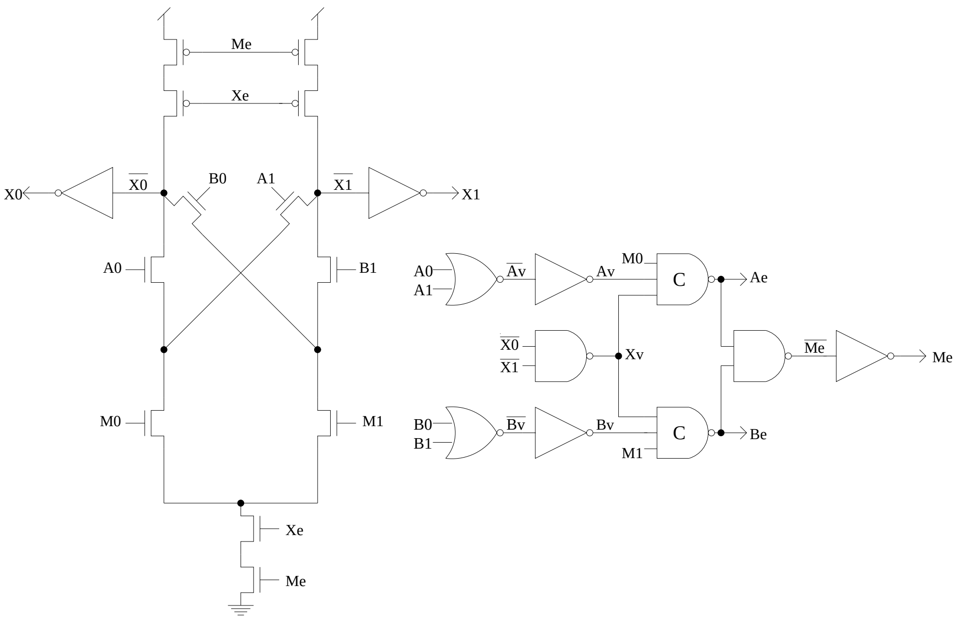

Figure 5: PCHB_MERGE

1.7 Internal state

One final extension to this design style is the ability to store internal state from one cycle to the next. A CSP template for a state holding process with state variable s is:

$$
\begin{aligned}
P3 \equiv\ \ & s := initial\_s; \\
& *[A?a,\ \ B?b,\ \ ...\ ;\ \ X!f(s,a,b,...),\ \ Y!g(s,a,b,...),\ \ ...;\ \ s := h(s,a,b,...)]
\end{aligned}
$$

This can be implemented in a variety of ways. The simplest, which requires no new circuits, is to feed an output of a normal pipelined cell back around to an input, via several buffer stages. One of these feedback buffers is initialized containing a token with the value of the initial state. Enough buffers must be used to avoid deadlock, and even more are needed to maximize the throughput (as discussed in chapter 2). Therefore, this solution can be quite large. For control circuitry (where area is less of an issue) this is often adequate. As an added benefit, the feed forward portion of the state machine can be implemented as several sequential stages of pipelined logic, which correspondingly reduces the number of feedback buffers necessary and allows far more complicated functions.

Aside from using feedback buffers, there are three main approaches to retaining state, of increasing generality and complexity. First, pipelining channels by themselves store state. Usually, these values move forward down the pipeline, passing through each stage only once. However, if a stage uses but does not acknowledge its input, the input value will still be there on the next cycle. Essentially, the token is stopped and sampled many times. In CSP, this can be expressed with the probe of the value of the channel. A conditional input type of circuit is used, which uses an input to produce outputs without acknowledging that input. This technique can be used for certain problems. For example, a loop unroller could take an instruction on the input channel, and produce many copies of it on an output channel based on a control input. Of course, this type of state variable can never be set, only read one or more times from an input.

If the state variable is exclusively set or used in a cycle, a simple modification of the standard pipelined reshufflings will suffice. The state variable, s is assigned to a dual-rail value at the same time the outputs are produced. On the reset phase, it remains stable. Unlike the usual return-to-zero variables, s will only briefly transition through neutrality between valid states. If s doesn't change, it does not go through a neutral state at all. The CSP for this behavior is expressed just like *P3*, except the semicolon before the assignment to s is replaced with a comma. This is made possible by the assumption that s only changes when the outputs X and Y do not depend on it; this avoids any stability problems.

The only tricky thing about deriving the HSE for this is the assignment statement. Basically, the assignment is done by lowering the opposite rail first, then raising the desired rail. This guarantees that the variable passes through neutral when it changes, and also bubble-reshuffles nicely. The completion detection of this assignment is basically equivalent to checking that the value of s corresponds to the inputs to s. So $s := x$ becomes $[x^0 \rightarrow s^1{\downarrow}; s^0{\uparrow} \,[]\, x^1 \rightarrow s^0{\downarrow}; s^1{\uparrow}]; [x^0 \wedge s^0 \vee x^1 \wedge s^1]$. The PCFB version of the HSE for this type of state holding process is:

$$
\begin{array}{l}
*[[\neg X^{\mathrm{a}} \wedge f^0(s, A, B, ...) \longrightarrow X^0{\uparrow} \,\|\, \neg X^{\mathrm{a}} \wedge f^1(s, A, B, ...) \longrightarrow X^1{\uparrow}], \\
\quad [\neg Y^{\mathrm{a}} \wedge g^0(s, A, B, ...) \longrightarrow Y^0{\uparrow} \,\|\, \neg Y^{\mathrm{a}} \wedge g^1(s, A, B, ...) \longrightarrow Y^1{\uparrow}], \\
\quad [h^0(A, B, ...) \longrightarrow s^1{\downarrow};\ s^0{\uparrow} \,\|\, h^1(A, B, ...) \longrightarrow s^0{\downarrow};\ s^1{\uparrow}], ...; \\
\quad A^{\mathrm{a}}{\uparrow}, B^{\mathrm{a}}{\uparrow}, ...; \\
\quad en{\downarrow}; \\
\quad [X^{\mathrm{a}} \longrightarrow X^0{\downarrow}, X^1{\downarrow}], [Y^{\mathrm{a}} \longrightarrow Y^0{\downarrow}, Y^1{\downarrow}], ..., \\
\quad [\neg A^0 \wedge \neg A^1 \longrightarrow A^{\mathrm{a}}{\downarrow}], [\neg B^0 \wedge \neg B^1 \longrightarrow B^{\mathrm{a}}{\downarrow}], ...; \\
\quad en{\uparrow} \\
]
\end{array}
$$

It is often desirable to decompose the completion detection of the state variable into a 4 phase completion variable s^{v} which detects the completion of the assignment on the set phase and is cleared on the reset phase. This makes it easier to have multiple state variables. One thing to note is that the assignment sequence and completion has 3 transitions if it changes state, and therefore often takes more transitions than a typical output channel. However, on the reset phase or if the state is unchanged, this only takes 1 transition. Another caveat is that the state variable shown here works best for only dual rail 1 bit state variables.

As an example of this type of state variable, consider the "register" process $x := 0; *[C?c; [c \rightarrow R!x \,[]\, \neg c \rightarrow L?x]]$. This uses a control channel C to decide whether to read or write the state bit x via the input and output channels L and R. Obviously, the state bit is exclusively used or set on any given cycle. This process also conditionally communicates on L and R, but since that was covered in the last two sections, we include it here. The PCHB version of the HSE is:

$$
\begin{array}{l}
PCHB\_REG \equiv \\
\quad x^0\uparrow, x^1\downarrow; \\
\quad *[[C^1 \wedge R^e \wedge x^0 \longrightarrow R^0\uparrow \\
\quad\quad [] C^1 \wedge R^e \wedge x^1 \longrightarrow R^1\uparrow \\
\quad\quad [] C^0 \wedge L^0 \longrightarrow x^1\downarrow;\ \ x^0\uparrow \\
\quad\quad [] C^0 \wedge L^1 \longrightarrow x^0\downarrow;\ \ x^1\uparrow]; \\
\quad\quad [C^0 \longrightarrow L^e\downarrow [] C^1 \longrightarrow skip]; \\
\quad\quad C^e\downarrow; \\
\quad\quad [\neg R^e \vee \neg R^0 \wedge \neg R^1 \longrightarrow R^0\downarrow, R^1\downarrow]; \\
\quad\quad [\neg L^0 \wedge \neg L^1 \longrightarrow L^e\uparrow]; \\
\quad\quad [\neg C^0 \wedge \neg C^1 \longrightarrow C^e\uparrow] \\
\quad]
\end{array}
$$

The PRS has a few tricky features. Due to the exclusive pattern of the communications the rules for C^e can be simplified. The decomposed and bubble reshuffled PRS follows, and the circuit is shown in Figure 6:

$$
\begin{array}{lcl}
C^e \wedge C^0 \wedge R^e \wedge x^0 & \rightarrow & \overline{R^0}\downarrow \\
C^e \wedge C^0 \wedge R^e \wedge x^1 & \rightarrow & \overline{R^1}\downarrow \\
\neg\overline{R^0} & \rightarrow & R^0\uparrow \\
\neg\overline{R^1} & \rightarrow & R^1\uparrow \\
\neg\overline{R^0} \vee \neg\overline{R^1} & \rightarrow & R^v\uparrow \\
R^v & \rightarrow & \overline{R^v}\downarrow \\
C^e \wedge C^1 \wedge L^0 & \rightarrow & x^1\downarrow \\
C^e \wedge C^1 \wedge L^1 & \rightarrow & x^0\downarrow \\
\neg L^0 \wedge \neg x^0 & \rightarrow & x^1\uparrow \\
\neg L^1 \wedge \neg x^1 & \rightarrow & x^0\uparrow \\
C^e \wedge (x^0 \wedge L^0 \vee x^1 \wedge L^1) & \rightarrow & L^e\downarrow \\
\neg L^e \vee \neg\overline{R^v} & \rightarrow & \overline{C^e}\uparrow \\
\overline{C^e} & \rightarrow & C^e\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
\neg C^e \wedge \neg R^e & \rightarrow & \overline{R^0}\uparrow \\
\neg C^e \wedge \neg R^e & \rightarrow & \overline{R^1}\uparrow \\
\overline{R^0} & \rightarrow & R^0\downarrow \\
\overline{R^1} & \rightarrow & R^1\downarrow \\
\overline{R^0} \wedge \overline{R^1} & \rightarrow & R^v\downarrow \\
\neg R^v & \rightarrow & \overline{R^v}\uparrow \\
\neg C^e \wedge \neg L^0 \wedge \neg L^1 & \rightarrow & L^e\uparrow \\
L^e \wedge \overline{R^v} & \rightarrow & \overline{C^e}\downarrow \\
\neg\overline{C^e} \wedge \neg C^0 \wedge \neg C^1 & \rightarrow & C^e\uparrow
\end{array}
$$

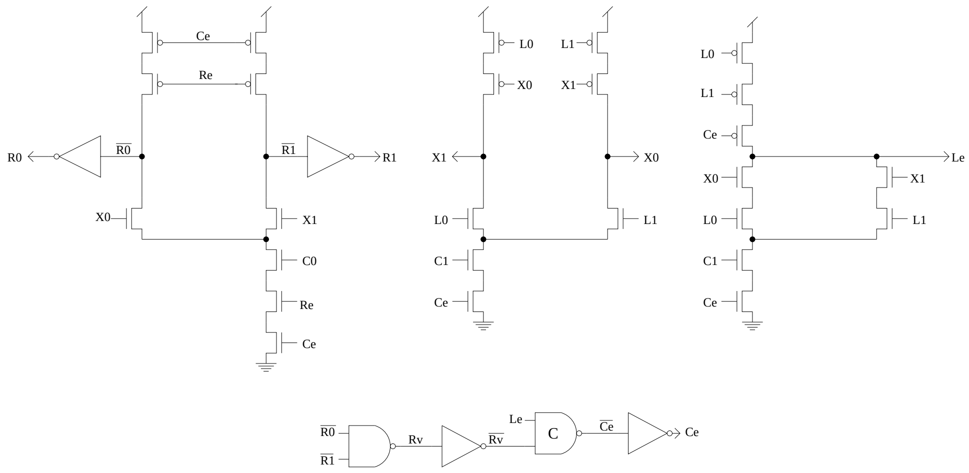

Figure 6: PCHB_REG

Finally, the most general form of state holding cell is one where the state variable can be used and set in any cycle. In order to do this, it is necessary to have separate storage locations for the new state and the old state. This may be done by introducing an extra state variable t which holds the new state until s is used. The CSP for this is:

$$
\begin{aligned}
P4 \equiv\;\; & s := 0; \\
& *[A?a,\;\; B?b,\;\; \ldots\; ;\;\; X!f(s,a,b,\ldots),\;\; Y!g(s,a,b,\ldots),\;\; t := h(s,a,b,\ldots),\;\; \ldots;\;\; s := t]
\end{aligned}
$$

When this is converted into an HSE, there are several choices for where to put the assignment $s := t$. It works best to do this assignment on the reset phase of the channel handshakes. After the assignment $s := t$, t returns to neutral just like a channel. The PCFB version of this type of cell is:

$$
\begin{aligned}
& s := 0; \\
& *[[\neg X^{a} \wedge f^{0}(s,A,B,\ldots) \longrightarrow X^{0}\uparrow [] \neg X^{a} \wedge f^{1}(s,A,B,\ldots) \longrightarrow X^{1}\uparrow], \\
& \quad [\neg Y^{a} \wedge g^{0}(s,A,B,\ldots) \longrightarrow Y^{0}\uparrow [] \neg Y^{a} \wedge g^{1}(s,A,B,\ldots) \longrightarrow Y^{1}\uparrow], \\
& \quad [h^{0}(s,A,B,\ldots) \longrightarrow t^{0}\uparrow [] h^{1}(s,A,B,\ldots) \longrightarrow t^{1}\uparrow], \ldots; \\
& \quad A^{a}\uparrow, B^{a}\uparrow, \ldots; \\
& \quad en\downarrow; \\
& \quad [X^{a} \longrightarrow X^{0}\downarrow, X^{1}\downarrow], [Y^{a} \longrightarrow Y^{0}\downarrow, Y^{1}\downarrow], \ldots, \\
& \quad [t^{0} \longrightarrow s^{1}\downarrow;\;\; s^{0}\uparrow;\;\; t^{0}\downarrow [] t^{1} \longrightarrow s^{0}\downarrow;\;\; s^{1}\uparrow;\;\; t^{1}\downarrow], \ldots, \\
& \quad [\neg A^{0} \wedge \neg A^{1} \longrightarrow A^{a}\downarrow], [\neg B^{0} \wedge \neg B^{1} \longrightarrow B^{a}\downarrow], \ldots; \\
& \quad en\uparrow \\
& \;]
\end{aligned}
$$

The assignment statements may be compiled into production rules as before. Of special interest is the compilation of the sequence $[t^{0} \rightarrow s^{1}\downarrow; s^{0}\uparrow; t^{0}\downarrow [] t^{1} \rightarrow s^{0}\downarrow; s^{1}\uparrow; t^{1}\downarrow]$. Due to correlations of the data, this compiles into the simple (bubble-reshuffled) production rules:

$$
\begin{array}{lcl}
\neg en \wedge \neg \overline{t^0} & \rightarrow & s^0\uparrow \\
\neg en \wedge \neg \overline{t^1} & \rightarrow & s^1\uparrow \\
s^0 \wedge \overline{t^1} & \rightarrow & s^1\downarrow \\
s^1 \wedge \overline{t^0} & \rightarrow & s^0\downarrow \\
\neg en \wedge \neg s^1 & \rightarrow & \overline{t^0}\uparrow \\
\neg en \wedge \neg s^0 & \rightarrow & \overline{t^1}\uparrow
\end{array}
$$

The s^0 and s^1 should also be reset to the correct initial value. The completion of this sequence is just the normal check for $\overline{t^0} \wedge \overline{t^1}$. If the state variable doesn't change, this sequence takes only 1 transition, since the first 4 rules are vacuous. If the state changes, it takes 3 transitions. This is 2 transitions longer than the reset of a normal output channel, so this should be considered to optimize the low level production rule decomposition. This type of structure only works well if s and t are dual-rail, although several dual-rail state variables can be used in parallel to encode more states.

Of the various types of state-holding cells, the more restricted versions generally have simpler and faster implementations, and should therefore be used if possible. For the most general case, either a pair of state variables should be used, or if area is not an issue, a feedback loop of buffers.

1.8 Conclusions

This chapter has presented a guide to designing asynchronous cells which combine buffering and computation functions. Three main types of handshaking reshufflings have proved superior for different circumstances. The weak-condition half-buffer variety works well for buffers and copies without logic. The precharge-logic half-buffering is the simplest good way to implement most logic cells. The precharge-logic full-buffering has an advantage in speed and is good at decoupling the handshakes of neighboring units. It should be used when necessary to improve the throughput. In addition, extensions to these cells which allow for conditionally receiving inputs or conditionally sending outputs were explained. Finally, various approaches to storing internal state in the cells were presented.

How far can these techniques go? An entire digital filter was designed using only WCHB and PCHB cells [2]. Even a complete asynchronous microprocessor (the MiniMIPS) uses basically these types of pipelined cells [3]. The MiniMIPS busses are all varieties of fullbuffers, the datapath logic and control logic are usually PCHB, and much of the control distribution and buffering is WCHB. Even cells as unusual as the caches were essentially implemented as one giant PCHB cell which has an exclusively used/assigned state variable structure plus a few low level transistor tricks for the SRAM cells themselves. The register locking is a little trickier, but all its input and output handshakes follow the standard PCFB approach. Basically, these techniques can account for almost all of the design of any type of asynchronous circuit, and strongly influence all design decisions. Other possibilities may be explored as the design constraints require, but these techniques form a default option for any asynchronous circuit implementation.

The prior state of the art was to use un-pipelined weak condition logic. Extra buffers or registers would be added between blocks of logic to add some pipelining. This approach was smaller, but much slower. The extra buffers also increased the forward latency. Essentially, in the limit of using more and more buffers, they should eventually be merged into the logic and all cells should be "maximally" pipelined. That is, any discrete state of logic gets its own pipelining, so that no more slack could be added without just throwing in excess buffers. In practice, the cost of such fine pipelining amounts to a 50% to 100% increase in area over a completely un-pipelined circuit. It reduces the latency (since no separate buffers are added), and, of course, increases the throughput. At this natural limit of pipelining, all handshakes between neighboring cells require a small number of transitions per cycle, typically 14 to 18. The internal cycles usually keep up. This yields a very high peak throughput (comparable to 14 transition per cycle hyper-pipelined synchronous designs like the DEC Alpha) but is more easily composable. However, composing fast pipelined cells in various patterns can yield much lower system throughputs unless special care is taken to match the latencies as well as the throughputs of the units. This is discussed in the next chapter.

2 Pipeline Dynamics, Slack Matching, and Transistor Sizing

Many techniques can be used to optimize the performance of a circuit, measured as latency, throughput, energy, or any other metric. The most significant optimizations can be made at the highest level, in the selection of algorithms and the decomposition of processes. Once the processes and topology of communication are fixed, further improvements can be made by improving the transistor implementations, as described in Chapter 1. Final optimization can use the techniques of slack matching and transistor sizing. Slack matching is the insertion or removal of slack (buffer capacity) along channels. It is most relevant in a highly pipelined design. Transistor sizing is the selection of transistors widths to optimize performance. This chapter investigates the proper use of slack matching, and its influence on transistor sizing. The emphasis is on optimizing throughput.

2.1 Pipeline Dynamics

Removing slack from channels may introduce deadlock, if the slack of a channel is not greater than the number of tokens it must contain. If the processes do not rely on inverted probes of channels or inherent disjunctions (as is normally the case), then adding slack to channels will not cause an error, since the process has no means of distinguishing the slack of a channel [4]. With the deadlock restriction in mind, it is possible to add or remove slack in two ways. One is to alter the slack inherent in the communicating processes, by varying the amount of logic between latches. As suggested in Chapter 1, the overhead of making all processes introduce slack $\frac{1}{2}$ or 1 between inputs and outputs is fairly small, and can substantially improve the throughput. The second way of altering the slack is to add buffers to channels.

In either case, to use slack matching as an optimization tool, we must understand the effect it has on the performance of the system. While the critical path may always be computed and used to check the effects of adding or removing slack, a simplified model of pipeline behavior can make the analysis much easier.

2.1.1 Linear Pipelines

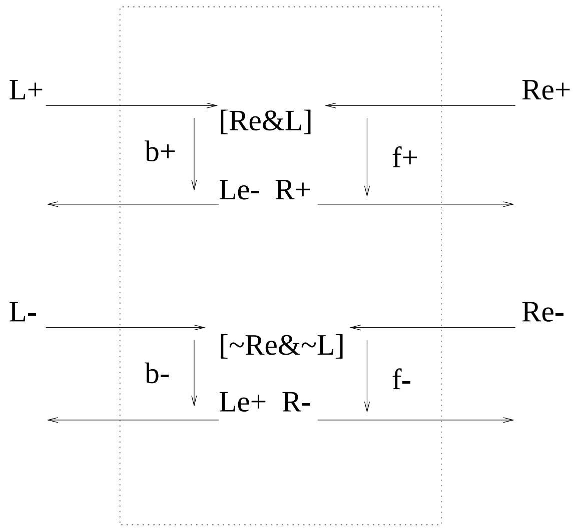

Figure 7: Timing Diagram of Halfbuffer Pipeline

Suppose we build a linear pipeline out of the *WCHB_BUF* halfbuffer of chapter 1. Figure 7 is a timing chart for this type of cell. Time progresses downward, and the pipeline flows from left to right. Only 4 delays are necessary to describe the cells: the forward latencies $f\uparrow$ and $f\downarrow$ and the backward latencies $b\uparrow$ and $b\downarrow$. Starting at the wait $[R^e \wedge L]$ we can traverse the directed graph until it gets back to another $[R^e \wedge L]$ in the same cell after N cycles. The cycle time of this path is the sum of all latencies on it divided by N. There are only three single cycle ($N = 1$) paths. They have cycle times of $f\uparrow + b\uparrow + f\downarrow + b\downarrow$, $f\uparrow + f\uparrow + b\uparrow + b\downarrow$, and $f\downarrow + f\downarrow + b\downarrow + b\uparrow$. There are arbitrarily many multi-cycle paths, but they turn out to be combinations of the three single cycle paths. Therefore, the critical path has cycle time $\tau = 2max(f\uparrow, f\downarrow) + b\uparrow + b\downarrow$.

We wish to analyze the throughput $t = \gamma(x)$ as a function of the number of tokens in the pipeline, x. Both t and x are taken to be averages in steady state operation. Steady state means the throughput measured at the input and output of the pipeline is equal and remains roughly constant over time. Let the pipeline consist of N stages.

First of all we know that if we have no tokens in a pipeline, it will have zero throughput. Also, we can never have more tokens in the pipeline that it has buffer capacity, henceforth called "static slack", s. In the case of the halfbuffers, $s = \frac{N}{2}$. We introduce "holes" to indicate the absence of a token, where the number of holes is $s - x$. When the pipeline is full, none of the tokens has a "hole" to move into, so the throughput is also zero.

If we introduce tokens at widely spaced intervals, the tokens flow forward through an empty pipeline, and exit after the total forward latency, which is $Nmax(f\uparrow, f\downarrow)$. The max is needed here because if the $f\downarrow$ is slower than the $f\uparrow$, the tail of the token lags behind the head, and will slow down the tokens behind it. The cycle time is the forward latency divided by the number of tokens, $\frac{Nmax(f\uparrow, f\downarrow)}{x}$.

Starting from a full pipeline, we can look at the problem from the point of view of holes moving backward. Once again, if the holes are separated widely in time, they will not affect each other and will exit after the total backward latency, which is $\frac{N}{2}(b\uparrow + b\downarrow)$. Here, the cycle time is $\frac{N}{2}\frac{b\uparrow+b\downarrow}{s-x}$. The intersection of the forward and backward constraints is found to be at x equal to the forward latency over the cycle time τ, with a peak throughput of $\frac{1}{\tau}$. The throughput versus tokens graph is a triangle.

The other buffer reshufflings produce similar results, except that they may have internal cycles which are slower than the handshake ones, which can cut off the top of the triangle and make it a trapezoid. The fullbuffer has a static slack, s, of 1 per stage, but otherwise has similar forward latency and peak cycle time, so only the right leg of the triangle is usually much different. Ted Williams has done this analysis in [5] and calls these regions of the trapezoid regions "data limited", "handshake limited" and "bubble limited".

A new term "dynamic slack" (d) is defined to indicate the number of tokens at which the throughput peaks. It is a dimensionless unit (like static slack). In cases where the throughput curve is a trapezoid, this left edge of the flat peak is at the "minimum dynamic slack" ($d_{\min}$) and the right edge is at the "maximum dynamic slack" ($d_{\max}$). Dynamic slack is inversely proportional to Ted Williams' dynamic wavelength, but it a useful concept because it is dimensionless and analogous to static slack.

The equations governing the throughput versus tokens trapezoid of a linear pipeline with peak throughput T, dynamic slack $d_{\min}$ to $d_{\max}$, and static slack s are:

$$\gamma(x) = T\frac{x}{d_{\min}}\, if\; x \le d_{\min}$$

$$\gamma(x) = T\, if\; d_{\min} \le x \le d_{\max}$$

$$\gamma(x) = T\frac{s-x}{s-d_{\max}}\, if\; d_{\max} \le x$$

$$t \le \gamma(x)$$

The number of tokens in the pipeline is equal to the forward latency over the cycle time (and likewise for the holes and the backward latency). This always holds in steady state, since the tokens are evenly spaced in time such that the number in a pipeline will be its latency divided by time separation of tokens (the cycle time). Equivalently, the forward latency is equal to the number of tokens over the throughput. Therefore the forward and backward latency as a function of tokens in the pipeline can be inferred from the throughput versus tokens curve. The forward latency remains constant as the throughput increases linearly with the number of tokens. When (and if) the internal cycle limit takes effect, then the forward latency increases proportionate to the number of tokens, since the throughput remains constant in this region. Finally, when the pipeline is bubble limited, the forward latency is increases as $\frac{x}{s-x}$, which asymptotically approaches infinity as the pipeline fills up. A similar graph is made for backward latency. The equations are:

$$f(x) = \frac{x}{\gamma(x)}$$

$$b(x) = \frac{s - x}{\gamma(x)}$$

The throughput versus tokens characterization $\gamma(x)$ has enough information to predict both steady-state throughput and latency. It takes four parameters to completely describe a throughput trapezoid. The parameters may include any four of static slack, minimum dynamic slack, maximum dynamic slack, peak throughput, forward latency or backward latency. Only static slack can be calculated from a circuit diagram; the others require a timing simulation or estimate. Note that dynamic slack may vary independently of static slack, although it must remain bounded by zero and the static slack. This means that the peak is not necessarily at half the static slack, as common rules of thumb imply.

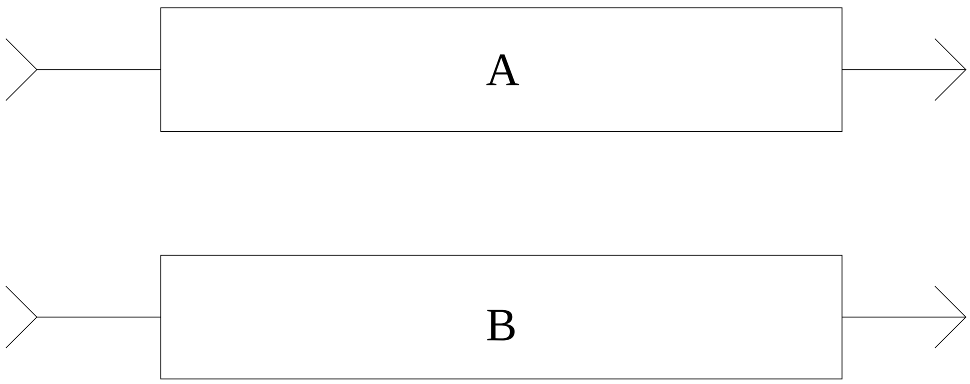

Figure 8: Two Linear Pipelines

Figure 9 displays throughput versus tokens triangles for two pipelines made up of the weak condition logic halfbuffer of Figure 1. These two buffers are shown schematically in Figure 8. The pipelines were designed with different static slack, transistor sizes, dynamic slack, and peak throughput. They have no internal cycles. However, they do appear to have a slightly chopped off peak. This is because the coincident arrival of two inputs into a c-element has a somewhat larger switching delay since the early input can't precharge some of the channel. Data points were taken from HSPICE simulations for a 0.8μm CMOS process and fit with a triangle. The "A" pipeline has static slack 20, dynamic slack 7.64, and peak throughput 458MHz. The "B" pipeline has static slack 25, dynamic slack 12.20, and peak throughput 412MHz.

2.1.2 Composition of Linear Pipelines

Only trivial circuits consist entirely of a homogenous linear pipeline. Many computations combine different types of pipelines that branch and join and feedback in loops. Now that we understand

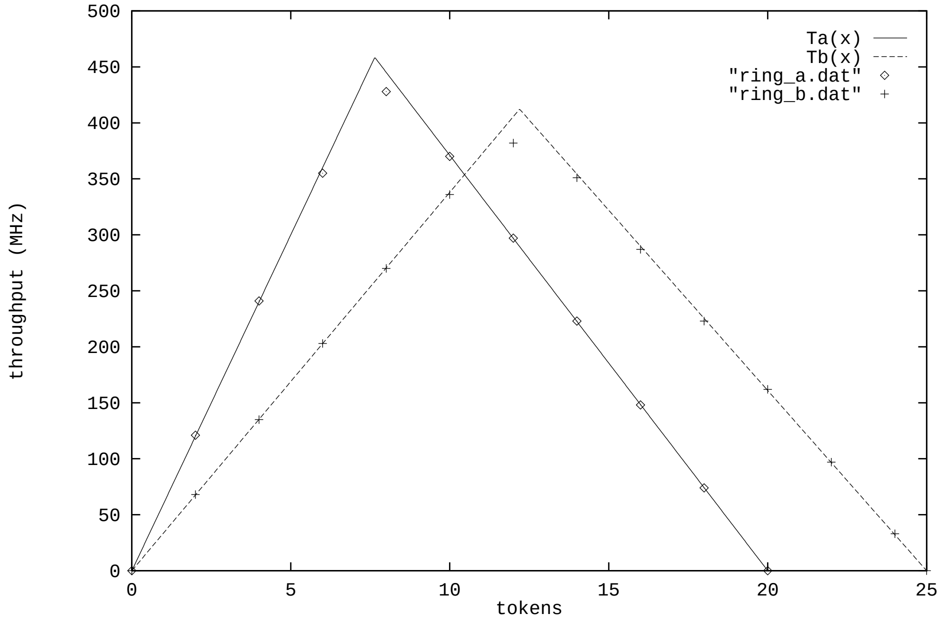

Figure 9: Throughput Versus Tokens for Two Linear Pipelines

the dynamic behavior of a homogenous linear pipeline, we can use that to build up to more complicated systems.

The general approach is to introduce independent variables for each of N homogenous linear pipeline $\{x_0..x_{\mathrm{N}-1}\}$ that indicate the number of tokens in that pipeline in steady state. Each section of linear pipeline must operate within it throughput versus tokens trapezoid:

$$\left\langle \forall i : 0 \leq i < N : t_{\mathrm{i}} \leq \gamma_{\mathrm{i}}(x_{\mathrm{i}}) \right\rangle$$

Connections between the pipelines amount to additional constraints on the system. Many forms of synchronization can be described by the constraint:

$$\left\langle = j : j \in S : \frac{t_{\mathrm{j}}}{w_{\mathrm{j}}} \right\rangle$$

This means that the throughputs of the pipelines of set S are constrained to have equal throughputs, with the possible use of a weighting factor w_{j}. This synchronization can describe the constraint introduced between the inputs and outputs of a buffering logic cell that always receives from all inputs and sends to all outputs (all $w_{\mathrm{j}} = 1$). A series connection is just a synchronization constraint applied to two pipelines with $w = 1$. A strictly interleaved split is modeled as a three way synchronization with $w = 1$ for the input and $w = \frac{1}{2}$ for the two outputs.

Similarly, a strictly interleaved merge is modeled with $w = \frac{1}{2}$ for the two inputs, and $w = 1$ for the output.

Unfortunately, this type of synchronization implies that the throughputs of each of the connected pipelines are in steady state. If the synchronization is more complicated and produces non uniformly spaced tokens, the assumptions all break down. This happens if the token flow is data dependent, or if the unit produces outputs in bursts. For these cases, the simplified "steady state" model of pipeline dynamics does not apply, and a full search for the critical path given a closed environment may be necessary.

Rings can introduce additional constraints on the x_i. To be in steady state, the total number of tokens in the ring must remain constant, so $x_0 + x_1 + ...x_{N-1} = C$ where the x's are for the segments around the ring, and C is the total. C is also know initially, since upon reset the buffers are know to be full or empty. Branching and joining paths also constrain the number of tokens in each branch to be equal, with a constant offset if any tokens were created in the branches on reset. The weights along the paths must also be equal to remain in steady state.

Once all the constraints are known, the legal range of t_i can be solved as a function of the x_i. If the pipelines are all connected to each other in some manner via the synchronization model, then all the throughputs will be multiples of each other, and one pipeline (usually an input or output) can be chosen to have the "system throughput". All of these constraints are piecewise linear and convex, so this is a straight forward linear programming problem to solve for the system throughput as a function of the x_i. Since CMOS circuits transition as early as they can, the steady state behavior will converge to the fastest allowed operating point(s). There may be many degenerate solutions for x_i at the peak throughput, but only the throughput is important.

2.1.3 Examples of Composite Circuits

The effect of various compositions on the two pipelines of Figure 8 is illustrated for several simple but common cases. If a pipeline is connected in a ring, the number of tokens in it remain constant, and the throughput can be predicted directly from its throughput versus tokens curve, as in Figure 9.

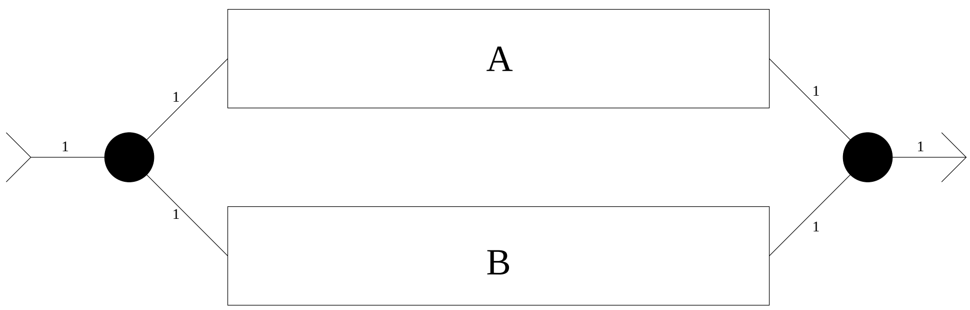

Figure 10: Parallel Composition of Two Pipelines

Figure 11 shows the behavior when the inputs and outputs of the pipelines are synchronized as in Figure 10. Both pipelines must run at the same throughput and must contain the same

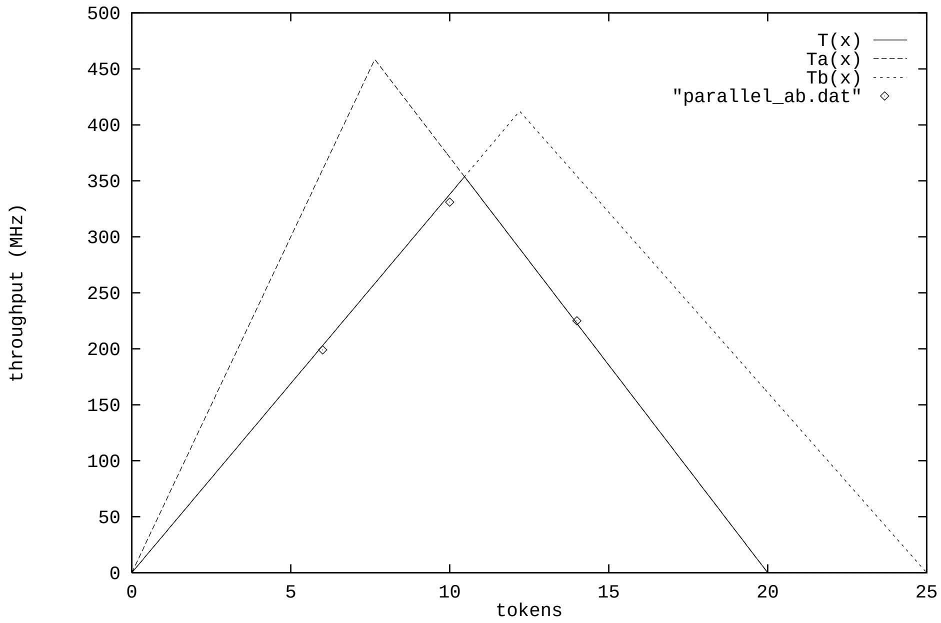

Figure 11: Throughput Versus Tokens for Parallel Composition of Two Pipelines

number of tokens. The combined throughput versus tokens curve is simply the intersection of the original curves (which may no longer be a trapezoid, but is still convex). As expected, the resulting peak is not greater than either component. More surprisingly, the peak is actually smaller than both of the original pipelines. This is of crucial importance in slack matching, since two equally fast, highly optimized pipelines can perform poorly in parallel if they don't also have the same dynamic slack. Note that the static slack is irrelevant if the dynamic slacks match. For the simple but common case of the intersection of two triangles where one is not wholly contained in the other, the resulting peak throughput and dynamic slack are found at the intersection of the right side of the triangle with lesser dynamic slack and the left side of the triangle with greater dynamic slack. Call the dynamic slack, static slack, and peak throughput of pipeline A d_a, s_a, and T_a, and likewise for pipeline B. Let $d_\mathrm{a} \leq d_\mathrm{b}$. The intersection is at:

$$x = \frac{T_\mathrm{a} d_\mathrm{b} s_\mathrm{a}}{T_\mathrm{b} s_\mathrm{a} - T_\mathrm{b} d_\mathrm{a} + T_\mathrm{a} d_\mathrm{b}}$$

$$t = \frac{T_\mathrm{a} T_\mathrm{b} s_\mathrm{a}}{T_\mathrm{b} s_\mathrm{a} - T_\mathrm{b} d_\mathrm{a} + T_\mathrm{a} d_\mathrm{b}}$$

The degradation of throughput due to dynamic slack mismatch between parallel pipelines is fairly gradual. Suppose the peak throughput and static slack of the two pipelines were equal.

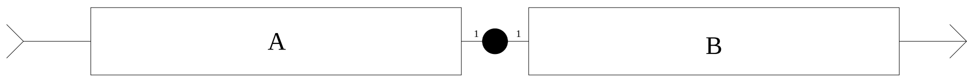

Figure 12: Series Composition of Two Pipelines

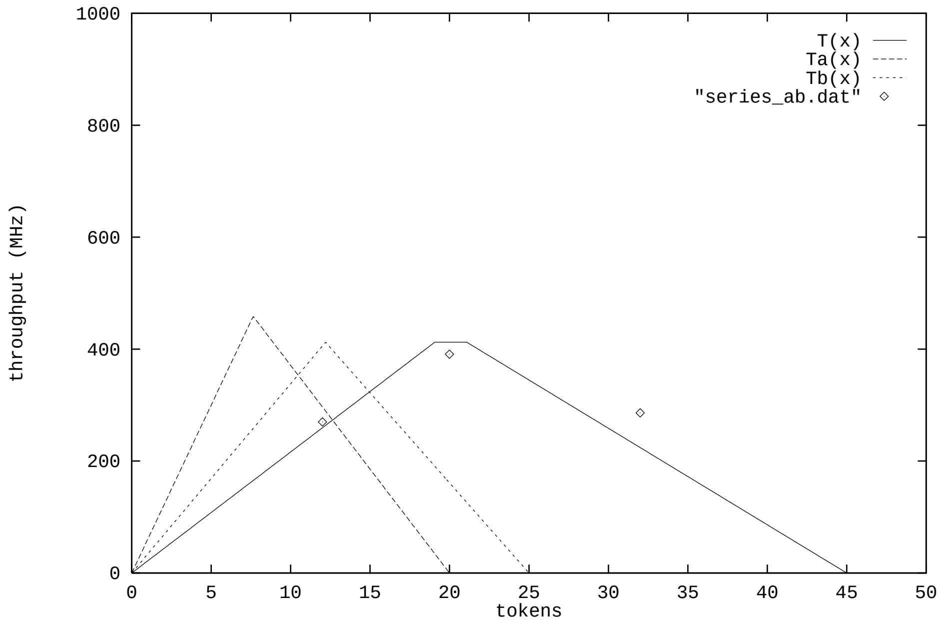

Figure 13: Throughput Versus Tokens for Series Composition of Two Pipelines

The worst possible dynamic slack mismatch would be when $d_{\mathrm{a}} \approx 0$ and $d_{\mathrm{b}} \approx s$. This would yield a throughput of $t = \frac{T}{2}$ with $x = \frac{s}{2}$. Less extreme mismatches run faster.

Figure 13 shows the behavior when the two pipelines are connected in series as in Figure 12. Although there are different numbers of tokens in the two pipelines, it is not important to distinguish how many tokens are in each part. In fact, at most throughputs there will be many degenerate solutions with the same throughput but a different distribution of tokens. Instead, we consider the throughput versus the total number of tokens, allowing the distribution to automatically optimize itself. For each particular throughput, we can see what range of tokens could be in each pipeline, as given by the triangle. To get the resulting range, we add the small ends and the large ends of the ranges. Obviously we can't have a throughput higher than the slower pipeline. This method traces out the trapezoid of Figure 13. For the simple case of two triangles, this leads to the following equations for T, $d_{\min}$, $d_{\max}$:

$$T = min(T_{\mathrm{a}}, T_{\mathrm{b}})$$

$$d_{\min} = \frac{d_{\mathrm{a}}T}{T_{\mathrm{a}}} + \frac{d_{\mathrm{b}}T}{T_{\mathrm{b}}}$$

$$d_{\max} = s_{\mathrm{a}} - \frac{T}{T_{\mathrm{a}}}(s_{\mathrm{a}} - d_{\mathrm{a}}) + s_{\mathrm{b}} - \frac{T}{T_{\mathrm{b}}}(s_{\mathrm{b}} - d_{\mathrm{b}})$$

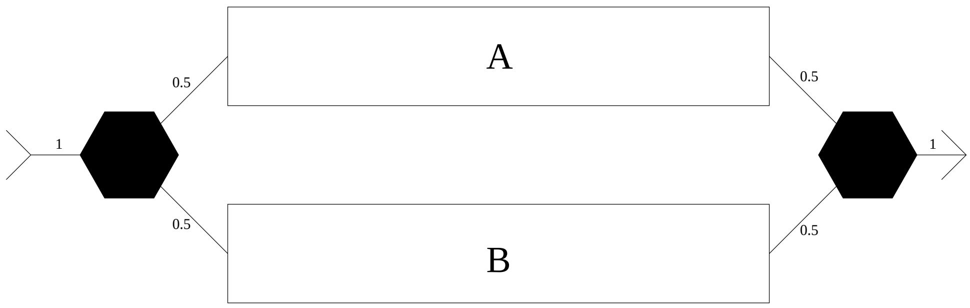

Figure 14: Interleaved Composition of Two Pipelines

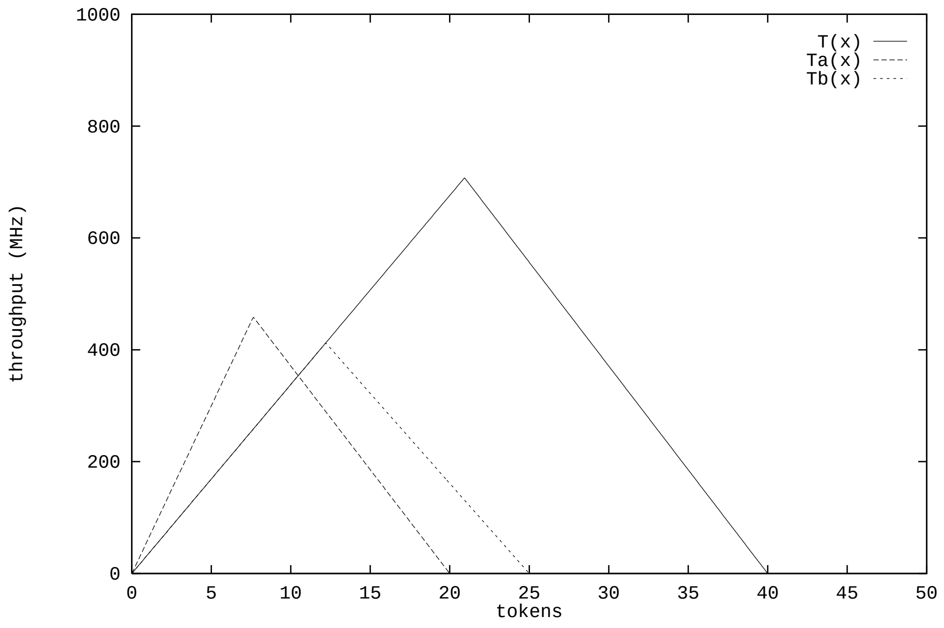

Figure 15: Throughput Versus Tokens for Interleaved Composition of Two Pipelines

Figure 15 shows the behavior of strict interleaving between the two pipelines. The constraint here is almost the same as for parallel pipelines. However, the results are interpreted differently, since the token's in each branch are different. This takes the intersection, but doubles its throughput and number of tokens. In practice, it is impossible to make a perfect interleaver with an arbitrarily high peak throughput. A realistic split and merge can be modeled as a short linear pipeline with reasonable throughput in series with the ideal interleaver.

2.1.4 Tree Buffers

A binary tree buffer can be made by recursively interleaving between linear buffers. Realistic alternating split and merge cells are modeled as slack $\frac{1}{2}$ buffers followed or preceded by an ideal interleaver. Such buffers have long been know to have desirable properties, including a forward latency and energy dissipation that is only logarithmic in the static slack. An understanding of slack matching illustrates several other interesting properties.

To compute the overall throughput versus tokens curve for a binary tree buffer, the easiest approach is to figure out the range of tokens in each segment for any given overall throughput. At each interleaving, the throughput is reduced by half. Adding up all the lower ends of the ranges and all the upper ends will give the overall range of tokens at any given overall throughput.

For an example, assume that the tree is k stages deep, and therefore interleaves between 2^k linear buffers. Each split and merge will be assumed to have an s and d consistent with a normal buffering cell. A split is followed by a perfect interleaving split, and a merge is preceded by a perfect interleaving merge. There are a total of n linear buffers in the center, evenly divided among the 2^k paths. Assume each path has $\frac{ns}{2^k}$ total static slack and $\frac{nd}{2^k}$ total dynamic slack. Assume all peak throughputs are T, and that all the throughput versus tokens curves are triangles. The total static slack is $ns + 2s(2^k - 1)$.

If a pipeline is run at a throughput $t \leq T$, we can calculate the range of tokens it might contain as follows. The triangle is described by the equations:

$$t = T\frac{x}{d}\, if\; x \leq d$$

$$t = T\frac{s-x}{s-d}\, if\; x \geq d$$

Given t, we solve for $x_{\min}$ and $x_{\max}$ as:

$$x_{\min} = \frac{t}{T}d$$

$$x_{\max} = s - \frac{t}{T}(s-d)$$

For our binary tree buffer, we must add up the $x_{\min}$ and $x_{\max}$ for all the linear buffers in the tree, for the particular t each one runs at. After some simplification, this produces the expressions:

$$x_{\min} = \frac{1}{2^k}nd + 2kd$$

$$x_{\max} = ns - \frac{1}{2^k}(ns - nd) + 2s(2^k - 1) - 2k(s - d)$$

Note that the first terms result from the linear buffers in the middle of the tree, while the final terms account for the slack inherent in the splits and merges. The equations can also be separated into overhead and incremental terms:

$$\alpha_{\min} = \frac{1}{2^k}d$$

$$\alpha_{\max} = s - \frac{1}{2^k}(s - d)$$

$$\beta_{\min} = 2kd$$

$$\beta_{\max} = 2s(2^k - 1) - 2k(s - d)$$

$$x_{\min} = \alpha_{\min}n + \beta_{\min}$$

$$x_{\max} = \alpha_{\max}n + \beta_{\max}$$

Suppose $d = \frac{1}{5}$, and $s = \frac{1}{2}$, which are typical values for weak condition halfbuffering cells. Then we get the following table:

k	$\alpha_{\min}$	$\alpha_{\max}$	$\beta_{\min}$	$\beta_{\max}$
0	0.2	0.2	0	0
1	0.1	0.35	0.4	0.4
2	0.05	0.425	0.8	1.8
3	0.025	0.4625	1.2	5.2
4	0.0125	0.48125	1.6	12.6

The throughput versus tokens curve for a $k = 3$, $n = 20$ binary tree buffer is shown in Figure 17. It has static slack 17, minimum dynamic slack 1.7, and maximum dynamic slack 14.45.

As we increase k, $\alpha_{\min}$ rapidly approaches 0, and $\alpha_{\max}$ approaches s. In effect, the overall throughput can be maintained with very near the entire range of tokens, $0 \leq x \leq s$. This can be extremely useful in many ways. In some circuits, a ring contains a variable number of tokens at different times. If the ring was implemented in a linear fashion, it would not always operate at peak throughput. However, with a binary tree buffer in the ring, it might operate at peak throughput for a wide range of tokens.

Yet another significant property of a binary tree fifo is that for some purposes it can be much smaller than an equivalent linear fifo. Suppose a buffer is to operate at peak throughput and contain a large number of tokens. To maintain peak throughput, the number of tokens must

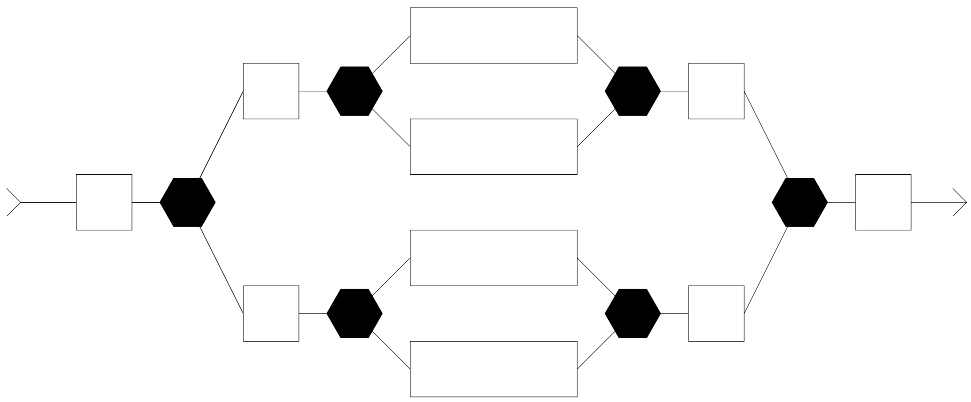

Figure 16: Binary Tree Buffer

be within the range of dynamic slack. For a linear buffer, the dynamic slack might be around $\frac{2}{5}$ of the static slack. For a tree buffer, the maximum dynamic slack can be much closer to the static slack. Since area is proportional to the number of buffers, and hence the static slack, a binary tree buffer that can keep the same number of tokens moving fast might be up to 2.5 times smaller than a linear buffer. In practice, the area overhead of the alternating split and merge must also be considered, but for very large n tree buffers, this overhead is very small. From the table, we see that the $\alpha_{\max}$ (the incremental increase in maximum dynamic slack) of a depth 2 fifo is already 85% of the static slack, so no excessive wiring is necessary to obtain the area benefit of using binary tree fifos. Naturally, if the objective is to have large static slack, at low throughput, the linear buffer will still be denser.

2.2 Slack Matching

Pipeline dynamics should be used to guide the design of a pipelined system. Several designs might be proposed and compared in their expected peak throughputs, but a few guidelines can be used to produce very good circuits without experimentation.

2.2.1 Heuristics for Slack Matching

First, specify the topology of communication and the static slacks of all cells and channels. The static slack must be enough to avoid deadlock, but no more. Assume that all buffering cells will have a characteristic dynamic slack d and peak throughput T. The dynamic slack is more directly relevant than the static slack to fast operation. The goal of slack matching is to add additional dynamic slack so that the whole system will maintain the T of the individual pieces.

It is most convenient to target a dynamic slack per cell which is the inverse of an integer. The choice should reflect the properties of the buffer implementation. Since the dynamic slack is just the forward latency divided by the cycle time, an estimate of these figures can yield a reasonable dynamic slack. For the weak condition halfbuffers with 10 transitions per cycle, d should probably be $\frac{1}{5}$, and for the precharge logic with 14 transitions per cycle, d should be $\frac{1}{7}$.

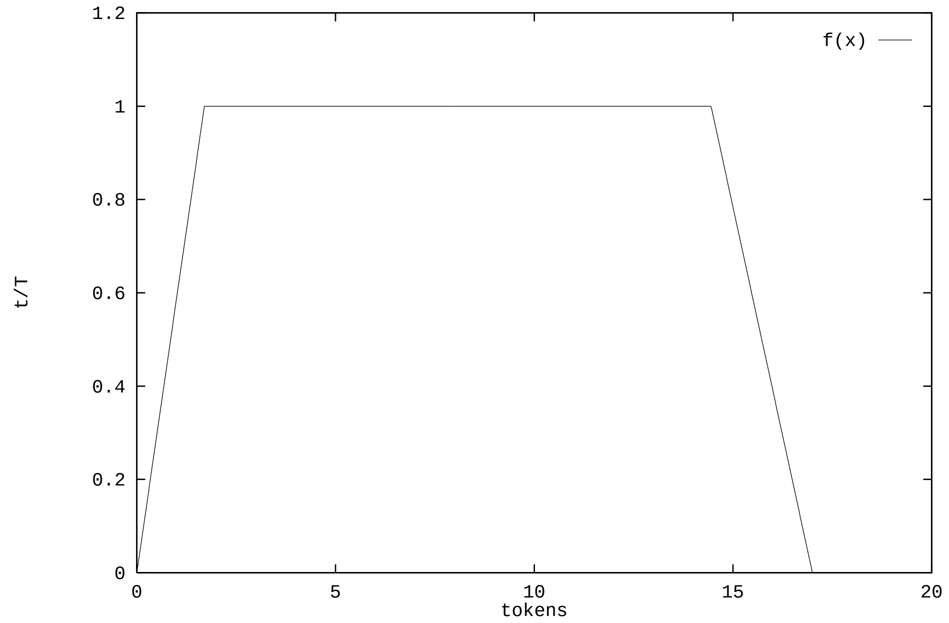

Figure 17: Throughput Versus Tokens for Binary Tree Buffer

It is possible to pick other values, like $\frac{1}{4}$ or $\frac{1}{6}$, but the peak throughput will depend on how closely matched the d is to the buffer implementation. One approach would be to optimize the throughput of a pipeline independent of the latencies, measure its dynamic slack, and pick the nearest inverse integer dynamic slack. The reason for fixing the dynamic slack as well as the peak throughput of all cells is that it becomes much easier to analyze the dynamics of a composition of pipelines.

Identify any rings in the topology, and predict the number of tokens they will contain. From pipeline dynamics, we know a ring will operate at peak throughput when the number of tokens is within the range of dynamic slack. If a ring has less dynamic slack than the number of tokens in it, extra buffers should be added until the dynamic slack equals the number of tokens. If the dynamic slack of the ring is already greater than the number of tokens in it, the circuit should be redesigned. The d was chosen to be the inverse of an integer so that an integral number of buffering cells in a loop will be optimal. For instance, if $d = \frac{1}{5}$, there should be 5 buffers per token in a loop.

If a ring is expected to contain a variable number of tokens, a deep binary tree in the ring can expand the range of dynamic slack such that the ring keeps operating at peak throughput. Note that the pipeline dynamics analysis assumes steady state operation, so it won't work if the number of tokens in the ring changes throughout the computation. However, if the number is only changed rarely, as when loading numbers in to a ring buffer for use over many cycles, then the system will probably have time to reach steady state, and the pipeline dynamics analysis will predict the performance for the most common and time consuming behavior.

Any paths which branch and later join will have reduced throughput unless they have equal dynamic slacks. Since we have assumed all cells have the same d, we can simply count the number of buffering cells along each path. If paths are unequal, extra buffers should be added to the shorter segment. In a feed forward pipeline with arbitrary branching and joining, it is always possible to add buffers until all paths have equal dynamic slack.

Adding extra buffers to slack match maximizes the throughput, but makes the forward latency the worst case always. In an extreme example, an N bit ripple adder operating on simultaneously arriving N bit numbers will have a latency of 1 to N depending on the length of the longest carry chain. The average case is $O(log_2 N)$. The throughput would vary along with the latency. Slack matching the ripple adder adds a triangle of buffers to delay the upper bits so that they arrive after the same dynamic slack as the rippling carry. A similar triangle of buffers is added to realign the output. The throughput now remains constant at the fastest cycle time of a full adder, but the latency is always N. Several better solutions can be obtained through architectural modifications. A fast rippling adder like a carry- select or carry-skip adder can minimize the forward latency while keeping constant, high throughput. Alternately, all numbers could be "bit-skewed" if that decision isn't inconvenient elsewhere.

Anywhere dynamic slack is added for the purposes of slack matching, it is possible to use a binary tree buffer. For the purposes of dynamic slack matching, binary trees can be significantly smaller, so trees of various depths should be considered to see which is smallest. Due to the overhead of interleaving, less deep trees will be better for smaller dynamic slack, but deeper trees will be better for longer dynamic slack. In most cases a linear buffer or depth 1 tree is probably appropriate. An additional benefit of the trees is reduced power dissipation.

For moderate amounts of dynamic slack, fullbuffers are often superior to halfbuffers. If a buffer is designed to run at higher than the target throughput, it can maintain the system throughput with a range of dynamic slack, just like a binary tree buffer. Since the right leg of the fullbuffer triangle extends twice as far as the halfbuffer, its maximum dynamic slack extends farther at lower system throughputs. Suppose we have a halfbuffer and a fullbuffer with $d = \frac{1}{5}$ which peak at $2T$. Evaluating the range at T shows that the halfbuffer has $d_{\max} = 0.35$ but the fullbuffer has $d_{\max} = 0.6$. Therefore, we could use almost half the number of these fullbuffers to achieve the same dynamic slack. If this compensates for the increased size of the fullbuffer implementation, the fullbuffers will be a smaller way to slack match. Fullbuffers were used for most slack matching in the MiniMIPS.

2.2.2 Appropriateness of Slack Matching

Slack matching aims to maximize the throughput of a collection of linear pipelines. If the logic is not buffering, or if early completion units are desired, it is not directly applicable. Also, if the system does not maintain steady state operation, the pipeline dynamics analysis does not apply. Therefore, slack matching is best suited for highly pipelined systolic systems with a regular pattern of computation. This is reasonable for many special purpose hardware chips, such as digital filters, compression, encryption, and other types of signal processing.

However, even in a more complex system such as a CPU, slack matching can be used to improve the design. Individual pieces may not always be used at full throughput, but slack matching can be used to improve their momentary peak throughputs. Reasonably isolated units may actually operate at full throughput for many cycles. The best examples of this are iterative computations done by rings, such as multiplication or division. Slack matching is an essential element to optimizing the performance of such units. Also, units can be partially slack matched so that their peak throughput may not be as high as that of linear pipelines, but is adequate for their use. Finally, the simplified timing model described by pipeline dynamics may be used to perform timing simulations of complex systems be reducing the complexity of representing various subunits.

2.3 Transistor Sizing

Given a topology of communicating processes, it is possible to identify the "critical path", a cyclic sequence of transitions, which limits the cycle time of the system as described in [6]. However, the search for this critical path can be time consuming, and even when it is found, it may not be obvious how to improve the performance of the system. Transistor sizing can speed up transitions on the critical path, at the cost of slowing down transitions not on the critical path. If this is done aggressively, most paths will be tied as the critical path, and no further improvements can be made by transistor sizing.

In the higher level optimization of slack matching, certain assumptions were made about the components of the circuit, namely that they had the same peak throughput T and dynamic slack d. Under these assumptions, the overall throughput of the system was optimized. In this context, the purpose of transistor sizing is to satisfy the assumptions made during slack matching. That is, the forward and backward latencies, and any internal cycles should meet the desired timing constraints. Once these timing constraints are met, another metric could be optimized, such as the energy per cycle or estimated area.

These latencies can be computed by adding up the transition delays along the paths. The transition delays depend on the width of the transistors driving them, and the capacitive load they drive, also determined by other transistor widths. Expressions for the worst forward, backward, and internal delays are computed from the circuit, and the transistor widths are adjusted (by gradient descent) until the constraints are met. If the constraints can't be met, less ambitions timing constraints should be tried. The target latencies should be consistent the the dynamic slack assumed for the cells, or it would invalidate the slack matching.

The delay of a single node transition can be approximated in many ways. The most popular method seems to be the tau model, which uses the effective resistance of the transistors in the channel and the capacitance on the output node to estimate an RC delay for the transition. An alternative is to construct a table which characterizes the peak currents through various networks of transistors, and to assume the delay is proportional to the capacitive load divided by the peak current. This later technique has the advantage of including significant process unidealities, most importantly velocity saturation. Parasitic capacitances (non gate capacitances) are generally smaller and depend greatly on the layout and physical distance between cells, which may not be known initially. Hence it is reasonable to initially size the transistors assuming no parasitic capacitances (or guessing for major sources of capacitance, such as long wires). After doing most of the layout, parasitic capacitances can be extracted from the layout and used to resize the transistors. Most likely one iteration of this procedure will be adequate, since the parasitic wiring capacitances don't change much with changing transistor widths and the change in diffusion capacitance can be easily predicted.

This approach to transistor sizing, using slack matching assumptions to set timing constraints, is quite efficient. The only delays that must be evaluated are local to each buffer cell, and depend only on the capacitive loads of other cells on the outputs. Identical cells in identical environments will give the same delays, and therefore need only be evaluated once. The algorithm will take memory only proportional to the number of unique cells in unique environments in the circuit. Likewise, the gradient descent will involve similar local calculations. For a fairly complex application, like the digital filter, there are only about 40 unique cells to size. However, there are undoubtably hundreds of thousands of potential critical paths. This approach to optimizing transistors in an asynchronous pipelined circuit shares much in common with the usual synchronous approach. Both methods rely only on local constraints. Both constrain the forward latency, but the asynchronous method must also constrains the backward latency and internal cycle time.

2.4 Conclusions

A complex asynchronous circuit may be designed by composing the pipelined cells described in chapter 1, evaluating the pipeline dynamics, and adding extra buffers to increase the system throughput to the same as the local handshake throughputs. Transistor sizing is used at the end to optimize the slowest handshake cycles and perhaps the latencies. Although the slack matching only applies precisely to circuits with regular systolic patterns of communication, the same concepts can be applied to portions of a more irregular system, such as a microprocessor. With the new techniques presented in this thesis, asynchronous quasi-delay-insensitive design has become more competitive with the synchronous alternatives.

References

[1] A. J. Martin, "Compiling Communicating Processes into Delay-Insensitive Circuits," *Distributed Computing,* Vol.1, No. 4, pp. 226-234, 1986.

[2] U.V. Cummings, A.M. Lines, A.J. Martin, "An Asynchronous Pipelined Lattice Structure Filter." *Advanced Research in Asynchronous Circuits and Systems,* IEEE Computer Society Press, 1994.

[3] A.J. Martin, A.M. Lines, et al, "The Design of an Asynchronous MIPS R3000 Microprocessor." *Proceedings of the 17th Conference on Advanced Research in VLSI,* IEEE Computer Society Press, 1997.

[4] R. Manohar, "The Impact of Asynchrony on Computer Architecture" *PhD. Thesis,* Caltech Technical Report, 1998.

[5] T. Williams, "Latency and Throughput Tradeoffs in Self-Timed Asynchronous Pipelines and Rings," *Stanford Tech. Report,* CSL-TR-90-431, May 1990.

[6] S. M. Burns, "Performance Analysis and Optimization of Asynchronous Circuits," Ph.D. Dissertation, Caltech, 1991.