Chapter 7

Design and test of high-speed asynchronous circuits

Marly Roncken[1] and Ivan Sutherland[1]

This chapter explores the design and test of high-speed complementary metal oxide semiconductor (CMOS) self-timed circuits. Section 7.1 describes how the properties of CMOS technology itself limit how fast a self-timed circuit can run. Section 7.2 presents our Link and Joint model, a unified point of view of self-timed circuits that allows reasoning about them independently of circuit families and handshake protocols. The model separates communication and storage, done in *Links*, from computation and flow control, done in *Joints*. The model also separates *actions* from *states*. Special *go* signals enable or disable Joint actions on an individual basis. The individual *go* signals make it possible to initialize, start, and stop self-timed operations reliably, which is crucial for design as well as for at-speed test, debug, and characterization. Section 7.3 examines design and test aspects of the Weaver, a self-timed nonblocking 8×8 crossbar switch designed using the Link and Joint model. We report measured test results from a working Weaver chip in 40 nm CMOS with speeds up to 6 Giga data items per second. With 72 bit wide data items, this amounts to 3.5 Tera bits per second for the full crossbar.

7.1 How fast can a self-timed circuit run?

How fast can a self-timed circuit run? What are its fundamental speed limits? What design considerations are important for digital circuits intended to operate at close to maximum speed? How does the potential speed of self-timed circuits compare with the speed of externally clocked circuits?

Lacking an external clock to drive their actions, all self-timed circuits must act on their own. In place of an external clock, all self-timed circuits and systems use the oscillations of logic rings to drive their actions.

Just as the tick of an external clock can be used as a unit of time, this chapter uses the term *gate delay* as if it were a unit of time. Of course, delay varies from

[1]Asynchronous Research Center, Maseeh College of Engineering and Computer Science, Portland State University, Portland, OR, USA

one logic gate to another, but high-speed circuits tend to size their transistors so that their logic gates have similar delay, giving the notion of gate delay a rational basis.

A gate delay is also a partly topological notion of the time it takes a signal to pass through an inverting logic gate, however long that might be. Except for pass gates, all individual CMOS logic gates invert logic signals, and so gate delays also count logic inversions.

7.1.1 Logic gate delays

A good model for CMOS logic gates operating at full power supply voltage associates delay with output transition time as illustrated in Figure 7.1.

When the input of the logic gate reaches a switching threshold the output of the gate begins an approximately linear change in voltage. The rate at which the output voltage changes depends on the *drive strength*, or simply *strength*, of the gate and how much *load* the gate drives. This makes it possible to formulate the notion of gate delay as the ratio of load driven to strength:

$$delay = load\ driven/drive\ strength.$$

Figure 7.1 illustrates the input and output voltage transitions of a logic gate driving respectively a light, medium, and heavy load. It uses as gate delay how long the gate output takes to reach the switching threshold of the next gate. Let us assume for now that the output voltage ramp starts at a power or ground supply rail. As Figure 7.1 illustrates, the actual delay of a logic gate of given drive strength depends on the load it drives and the switching threshold of the next logic gate. Assuming switching thresholds are about midway between the power and ground supply rails, the delay of each logic gate is approximately half the time its output would take to swing from one rail to the other. Transitions in signal voltage are the

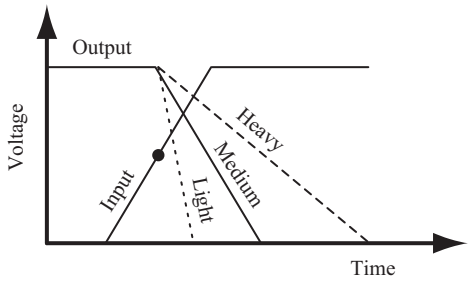

Figure 7.1 CMOS logic gate delay model. The outputs of CMOS logic gates change relatively slowly. The picture shows three possible output transitions for a logic gate driving a light, medium, or heavy load. The output begins to fall when the input voltage exceeds a logic threshold, marked here with a dot at approximately half of the supply voltage. Because the next logic gate acts only when its input reaches its threshold, the delay associated with each gate is roughly half as long as the full output transition time of the gate

major source of delay and energy consumption in CMOS logic. In the usual operating range, the voltage transitions are approximately linear ramps.

If, however, the output ramp starts at a voltage other than a power or ground supply rail, the delay of the gate also depends on that starting voltage. The delay may be very small if the output ramp starts at nearly the switching voltage of the next gate. Moreover, different starting voltages cause different delays. A ring of logic gates with widely varying delay may still oscillate but the output signals of its slower gates may fail to swing rail to rail and may thus create a steady state behavior that is very different from the ring's starting cycles. Reliable operation is best achieved by making sure that the output transitions of all gates start and end near the voltage of a power or ground supply rail.

7.1.2 Rings of logic gates

Ring oscillators internal to a self-timed circuit drive the circuit's actions. It is well known that a CMOS ring oscillator must have an odd number of at least three inverting logic gates. It is less well known that for reliable oscillation the logic gates in a small ring must have very nearly matching delays. One can design self-timed circuits that operate at maximum possible speed, that is, at the speed of a three-gate ring oscillator, like 4-2 GasP [1]. However, such high speed requires very careful design. Practical self-timed circuits, like the 6-4 GasP circuits used in the Weaver and discussed in more detail in Section 7.3, tend to run no faster than the speed of a five-gate ring oscillator. Nevertheless, the Weaver circuits still offer an impressive throughput of one data element every ten fanout-of-four gate transitions, which is about twice the typical speed of a clocked system.

In this section, we examine and compare three-gate rings to five-gate rings. Let us start with a ring oscillator with three logic gates, like the one in Figure 7.2. If all three gates have equal delay, all three signals—*a*, *b*, and *c*—will reach full swing, as illustrated in the upper timing diagram of Figure 7.2. If, however, gate *c* is twice as slow as the others, its output will barely reach full swing, as illustrated in the lower timing diagram. For all three signals of a three-gate ring oscillator to reach full swing, the delays in the three gates must match within about a factor of two. Any greater delay mismatch risks uncertain switching delay and erratic behavior.

Figure 7.3 shows a ring oscillator with five logic gates. If all five gates have equal delay then all signals will reach full swing, as illustrated in the upper timing diagram of Figure 7.3. In the middle timing diagram, signal *e* suffers twice the delay of signals *a*, *b*, *c*, and *d*, and yet *e* still dwells at the power and ground rails. The lower timing diagram shows that *e* has to be about four times slower to barely reach full swing. In comparison, rings with five logic gates offer the following three benefits over maximum-possible-speed rings with three logic gates.

1. **Greater flexibility in accommodating gate delay mismatches:** Rings of five logic gates better tolerate variation in the delays of their individual parts be it from variations in wire capacitance or in manufacturing or otherwise. Rather than matching the logic gate delays within a factor of two as required by a three-gate ring, a ring of five logic gates requires that the delays match within

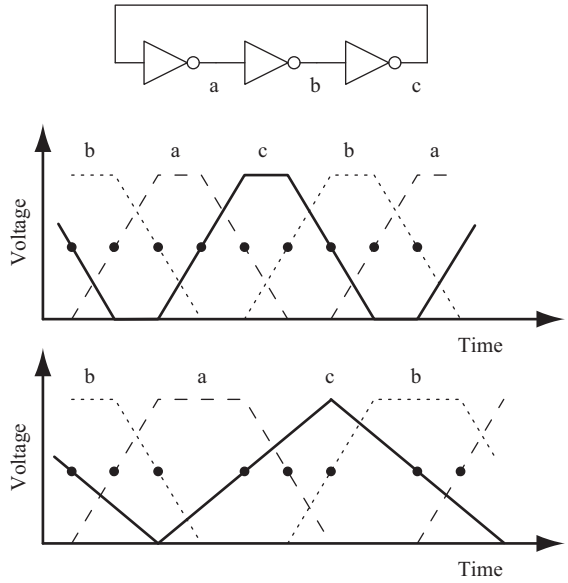

Figure 7.2 Delay in a three-inverter ring. A ring with three inverting gates of similar delay produces full-swing signals, as illustrated in the upper timing diagram. In the lower timing diagram, signal c suffers twice the delay of a and b. Were signal c driven any slower, it would fail to reach full swing. For all three signals to make full transitions, the delays of the gates must match within about a factor of two

about a factor of four. A factor of two requires careful design. A factor of four is easy to achieve.

2. **Greater robustness in signaling:** The signals produced by rings of five logic gates dwell at the power and ground supply rails longer than those of rings of three logic gates. In contrast, the signals in three-gate rings curve and sometimes even form sharp points near the rails as they change direction from rising to falling. The dwell feature in rings of five gates separates cleanly the successive rise and fall ramps of each signal.
3. **Greater flexibility in accommodating logic computations:** The topology of five-gate rings provides more logic gates to do logic. Rings with three gates often fail to have enough stages to invert particular signals and must instead resort to duplicating the entire ring in true and complement form.

Their better tolerance for delay mismatch and greater logic flexibility make ring oscillators with five logic gates not only more robust than those with three gates but also easier to design. In general, the more gates a logic loop has the greater disparity it permits between its slow and fast gates.

As mentioned at the start of Section 7.1, self-timed circuits and systems use the oscillations of logic rings to drive their actions. Going around the ring twice creates

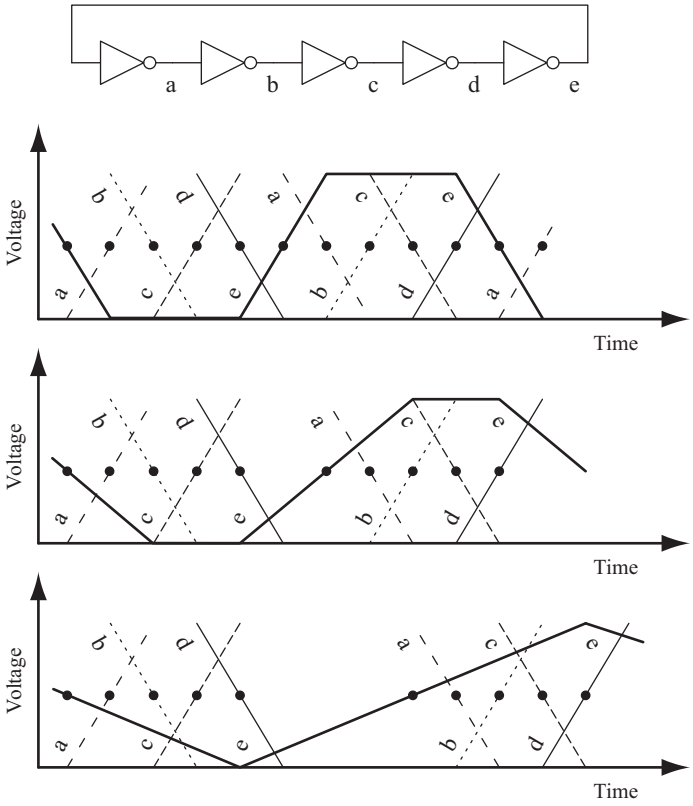

Figure 7.3 Delay in a five-inverter ring. A ring with five inverting gates of similar delay produces full-swing signals, as illustrated in the upper timing diagram. In the middle timing diagram, signal e suffers twice the delay of the other signals and yet still dwells at the power and ground rails. For all five signals to make full transitions, the delays of the gates must match within about a factor of four, as illustrated by the lower timing diagram

[Note: For clarity, the diagrams in Figure 7.3 omit the power-rail and ground-rail voltage levels for signals a through d and show only their rising and falling voltage transitions.]

a high-to-low-to-high or low-to-high-to-low pulse on each signal. The rings in Figures 7.2 and 7.3 generate approximately symmetric pulses with more or less equal high-to-low-to-high and low-to-high-to-low pulse widths. It is also possible to use the signal changes on a logic ring to generate asymmetric pulses, as in Figure 7.4.

When given wider than three gate delay pulse inputs, the two pulse generators in Figure 7.4 create asymmetric pulses with three gate delay high-to-low-to-high pulse widths in (a) and three gate delay low-to-high-to-low pulse widths in (b). One

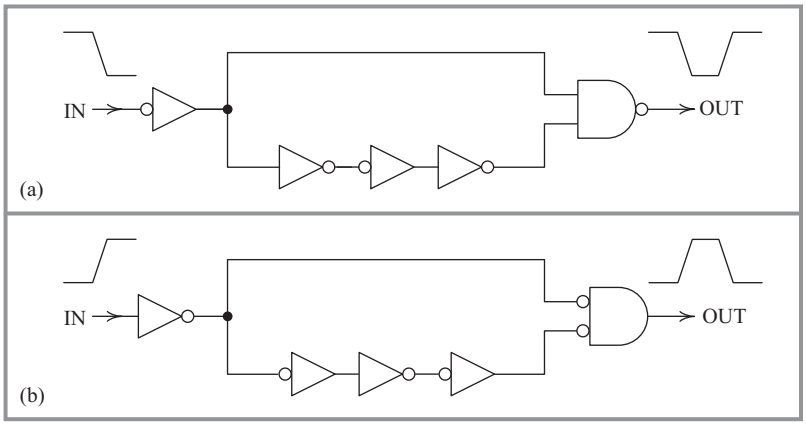

Figure 7.4 Asymmetric pulse generators. A falling transition on IN in (a) makes the output, OUT, of the NAND gate fall and then rise in rapid succession by generating a three gate delay high-to-low-to-high pulse, as suggested by the waveforms shown above signals IN and OUT. Likewise, a rising transition on IN in (b) creates a low-to-high-to-low pulse of three gate delays at the output, OUT, of the input-inverted AND gate—also known as a NOR gate. For full-swing transitions and a three gate delay output pulse, the gates must be sized appropriately and may require artificial load inverters not shown here. All input signals on IN must be high for at least three gate delays and low for at least three gate delays

[Note: Figure 7.4 draws inverting gates with an inversion symbol "○" either at the gate's output or at the gate's input to indicate whether the key transition they drive is a falling or rising one. For instance, the key transition for OUT in (a) is a falling transition, because it starts the pulse on OUT. Therefore, we draw the NAND gate that drives it with the inversion symbol at its output. Likewise, the key transition for OUT in (b) is a rising transition, and so we draw the gate that drives it with the inversion symbol at its inputs. This notation is a matter of taste, but where it matters it enables us to emphasize the meaning of the circuit.]

can style pulse generators similar to those in Figure 7.4 to generate wider pulses of a more or less fixed width.

Locally generated pulses—either symmetric or asymmetric—can be used as "local clock" signals to drive local latches or flipflops or other types of storage elements to update the local state information changed during each action. Local clock pulses that drive many storage elements may need amplification to obtain enough drive strength. Section 7.1.3 discusses how one can amplify pulse signals.

7.1.3 Amplifying pulse signals

To drive a large load from a relatively weak source one can use a series of inverters with exponentially increasing drive strengths, as shown in Figure 7.5.

To support the *strength*, *load*, and *step-up* numbers indicated in the context of Figure 7.5 and the subsequent Figure 7.6, let us define the terms associated with these numbers. All definitions are relative to a unit inverter, which is the smallest inverter allowed in a particular circuit family or manufacturing process. The unit inverter presents a load of 1 at its input and has a drive strength of 1.

- *Drive strength or strength* for short indicates the ability of a logic gate to drive load. Its strength is how many times stronger the logic gate is than a unit inverter in driving load at its output. Typically, one makes the transistors in a logic gate wider or—equivalently—puts them in parallel to increase the drive strength of the gate.
- *Load presented or input load* is how much input charge a logic gate takes to turn its transistors on or off relative to the charge required to switch a unit inverter. In other words, the load that a logic gate presents to its input is how much more difficult it is to turn its transistors on or off than it is to turn on or off the transistors of a unit inverter. Wider transistors are more difficult to drive.
- *Step-up* is the ratio of load driven to strength:

  $$\textit{step-up} = \textit{load driven}/\textit{drive strength}.$$

  Note that we used the same formula for the delay of a logic gate—see page 114. Successive gates that use the same step-up, or delay, are fastest overall.* We chose the strengths in Figures 7.5 and 7.6 to give each gate an equal step-up of four.
- *Amplification or gain* is the ratio of load driven, at the output of a gate or series of gates, to load presented at the input of a gate or series of gates. A logic gate can drive multiple other gates. The input loads presented by the other gates add up to the total load driven by the logic gate. Figures 7.5 and 7.6(a) and (b) show the load presented at the input, *IN*, of each two-stage pulse amplifier and the total load that can be driven at the output, *OUT*, to give each stage a step-up of four. The corresponding amplifications from *IN* to *OUT* are 16, 72, and 60, respectively.

Figure 7.5 shows two series inverters that form a two-stage amplifier with a uniform step-up of four per stage. Together the two stages provide an amplification of 16—from presenting a load of 1 at *IN* to driving a load of 16 at *OUT*.

However, given that an inverter's rise time almost always differs from its fall time, each stage will either retard or advance its output transition relative to its input transition. Each stage in the pulse amplifier of Figure 7.5 has two opportunities to change its output pulse width relative to its input pulse width—once by retarding or advancing the rising transition of the pulse and once again by retarding or advancing the falling transition of the pulse. The accumulated change will inevitably either lengthen or shorten a pulse from the amplifier's input, *IN*, to its output, *OUT*. This is particularly problematic when amplifying short pulses from *IN* to *OUT*.

*An alternative but equivalent guideline for achieving fastest delay follows in Section 7.1.4.

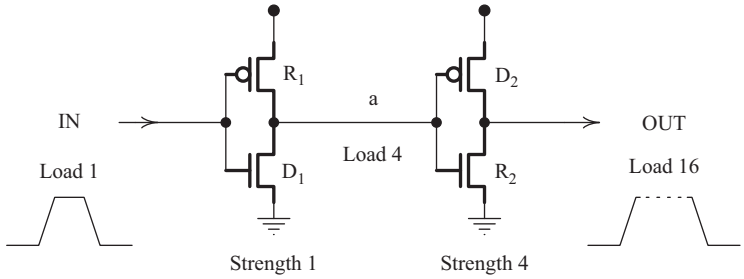

Figure 7.5 Pulse amplifier. A simple two-stage amplifier consisting of two inverters in series can amplify a pulse. The strengths shown give each inverter a step-up of four. The inverter pair provides a 16-fold gain. The broken line in the pulse waveform at signal OUT indicates that the pulse width at OUT can vary and thus be wider or narrower than the pulse width at IN. The variation is due to accumulated differences in rise and fall times at each inverter stage, and makes this simple solution less suitable for amplifying short pulses

If you must amplify a short pulse it is best to do so by avoiding accumulation of delay changes from intermediate stages. To do so, use feedback at the output to control the output pulse width directly, as in Figure 7.6.

The two circuits in Figure 7.6 illustrate a technique called *post-charge logic* described in an expired patent [2] from the late Bob Proebsting that:

> "[· · ·] permits propagation of a pulse through an arbitrary number of stages with the pulse width remaining essentially unchanged."

To understand how each circuit of Figure 7.6 works, first consider only the bold transistors labeled D_1 and D_2. These transistors match the corresponding transistors in Figure 7.5—they turn on the pulse signal at each stage. In the circuits of both Figures 7.5 and 7.6, when the input signal, *IN*, rises, transistor D_1 drives signal *a* low, turning on transistor D_2 to drive the output signal, *OUT*, high. Each stage has more drive strength than its predecessor, as indicated by the strength numbers for each stage in each circuit.

But unlike the two inverter stages in Figure 7.5, the two stages in Figure 7.6(a) and (b) avoid wasting input charge to control the reset transistors, R_1 and R_2. Instead, the *post-charge logic* of Figure 6(a) and (b) drives R_1 and R_2 from the amplified output signal. For each circuit in Figure 7.6, when *OUT* rises, its inverted and amplified signal, *c*, falls to raise its inverted and amplified signal, *d*, turning on transistor R_2 to reset *OUT* to low. Meanwhile, the falling signal *c* turns on transistor R_1 to reset *a* to high, and—for (b)—also turns off transistor D_{1A} to avoid fighting *IN* pulses that might be somewhat wider than three gate delays.

In addition to stabilizing the pulse width, the *post-charge logic* technique offers a much higher amplification. The load numbers for *IN* and *OUT* in Figure 7.6 show an amplification of 72 for (a) and 60 for (b) versus only 16 for Figure 7.5.

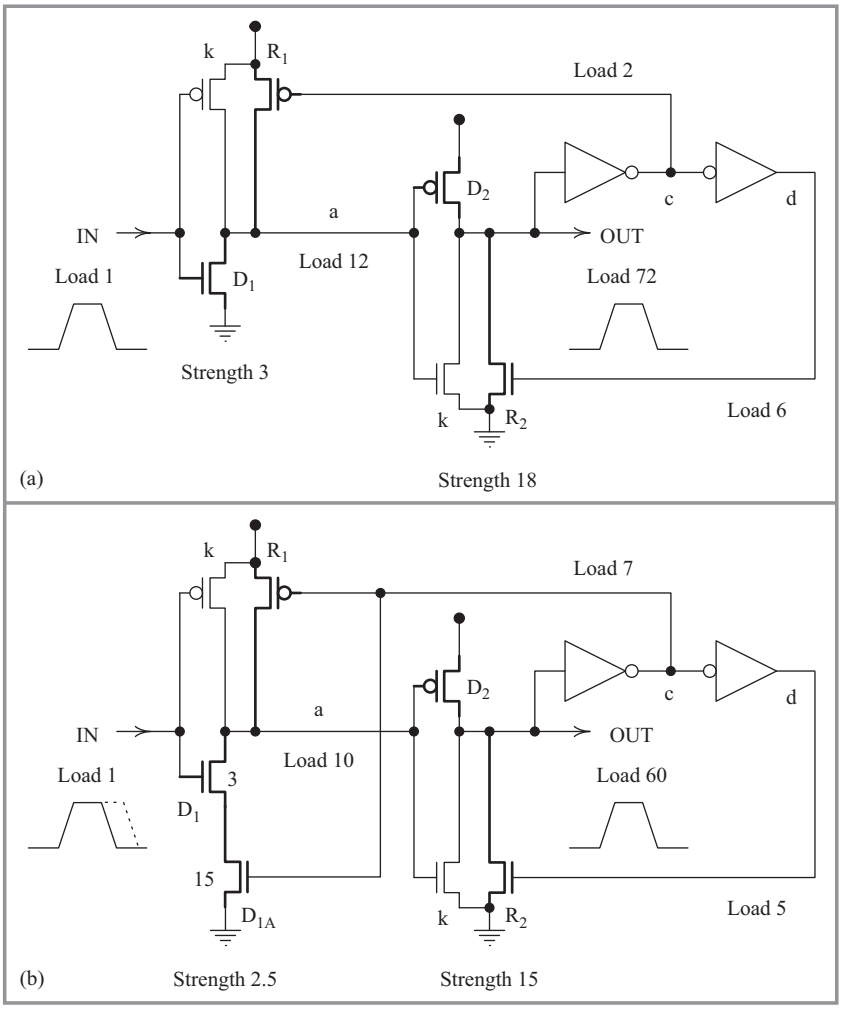

Figure 7.6 Pulse amplifiers with higher gain. More amplification is available by postponing the drive of reset transistors, R_1 and R_2. Two feedback loops from the amplified output signal, OUT, drive R_1 and R_2. Because none of the charge on signals IN and a *is spent on transistor R_1 and R_2 respectively, transistors D_1 and D_2 can be stronger than in Figure 7.5. High gain is extremely useful for driving signals with large fan-out, like clock trees or the "local clock" signals that drive the storage elements in many an asynchronous design. Moreover, the two feedback loops maintain the pulse duration, which is particularly important when amplifying short pulses. Both (a) and (b) output fixed three gate delay wide pulses, but (b) will accept somewhat wider input pulses*

In other words, for the same input loading, Proebsting's two-stage *post-charge logic* can drive about four times as much load as two ordinary inverters, and it responds just as fast.

A rationale for the strength and load numbers in Figures 7.5 and 7.6 follows in the itemized calculation narrative below. The calculations assume that a P-type transistor is about two times harder to drive than an N-type transistor with the same output load. These assumptions are valid for the 40 nm CMOS Weaver chip in Section 7.3.

- An N-type transistor and a P-type transistor each of strength 1 make a unit inverter of strength 1, like the first stage inverter in Figure 7.5. Assuming that P-type transistors are twice as hard to turn on or off as N-type transistors, two thirds of the input load of the first stage inverter in Figure 7.5 is due to R_1. When *IN* goes high this two third serves to turn off R_1, leaving one third to turn on D_1 to pull signal *a* down. Using a step-up of four per amplification stage, transistor D_1 with its drive strength of 1 allows the falling signal *a* to drive a load of 4.
- Instead of turning R_1 off at the last possible time, one could turn R_1 off ahead of time, as do the amplifiers in Figure 7.6(a) and (b). Thus when *IN* goes high, the first stage inverter in the two amplifiers in Figure 7.6 serves fully to turn on D_1 to pull signal *a* down. In other words, these first stage inverters can devote their available input load of 1 for *IN* to drive N-type transistor D_1, that is, to drive $\frac{1}{3} \times x$, with x the strength of D_1. This makes it possible to resize D_1 from a strength 1 in Figure 7.5 to a strength 3 in Figure 7.6(a) and (b). Using a step-up of four per stage, transistor D_1 with its drive strength of 3 allows the falling signal *a* to drive a load of 12 in Figure 7.6(a). The strength-3 transistor D_1 in series with a strength-15 transistor D_{1A} in Figure 7.6(b) yields a combined drive strength of $1/(1/3 + 1/15)$, or 2.5, for the first-stage amplification. Using a step-up of four, the strength-2.5 series pair allows the falling *a* to drive a load of 10 in (b).
- Similarly, with R_2 in the second stage inverter in Figure 7.6(a) and (b) turned off ahead of time, the available load on *a* can be devoted entirely to turn on D_2 to pull *OUT* up. In other words, the second stage inverter in Figure 7.6(a) can devote 12 to drive the load of P-type transistor D_2, that is, to drive $\frac{2}{3} \times y$, with y the strength of D_2. This makes it possible to resize D_2 from a strength 4 in Figure 7.5 to a strength 18 in Figure 7.6(a). Using a step-up of four per stage, the strength-18 transistor D_2 allows the rising *OUT* signal to drive a load of 72 in Figure 7.6(a). Likewise, P-type transistor D_2 in Figure 7.6(b) can be resized to a strength of 15, allowing the falling *OUT* signal to drive a load of 60 in (b).
- These calculations can be extended to choose the strengths of the series inverters on *OUT* that drive signals *c* and *d* to reset the output of each amplification stage in Figure 7.6(a) and (b). Their strengths can be small—between 1 and 2—and hardly impact the remaining drive load on *OUT*. By fine-tuning the drive strengths of these feedback inverters one can fine-tune the pulse width on *OUT*.

The circuit in Figure 7.6(a) assumes similar input and output pulse widths. The circuit in Figure 7.6(b) accommodates wider input pulses. Between pulses, weak keepers—marked with the letter *k*—maintain the charge and corresponding logic high or low voltage level on the output signal of each stage in Figure 7.6(a) and (b).

Figures 7.5 and 7.6 show circuits of only two stages, amplifying a low-to-high-to-low pulse of three gate delays. Similar circuits amplifying high-to-low-to-high pulses or circuits with more stages and wider pulse widths are also possible. All assume that their signals have enough time to reach full swing—see Figures 7.2 and 7.3 for a reminder on full swing rail to rail signal transitions.

7.1.4 The theory of logical effort, or how to make fast circuits

To do its logical function, a NAND gate must have more transistors than an inverter. To drive as much load as a same-strength inverter, a NAND gate not only uses more transistors but also has transistors connected in series that are extra strong. Because it has more transistors, some of them extra strong, a NAND gate has more input load. It is harder to turn the transistors of a NAND gate on or off than it is to turn on or off the transistors of an inverter of the same strength.

The Theory of *Logical Effort* [3] quantifies the "cost" or *logical effort* of doing logic as how much worse input load a logic gate presents than would an inverter of equal strength. In other words:

$$\textit{logical effort} = \textit{input load}/\textit{drive strength}.^{\dagger}$$

We use 1 for the logical effort of an inverter. More complex logic gates tend to have logical effort larger than 1. Usually, the more complex the logic, the larger its logical effort. For example, assuming that P-type transistors are twice as hard to turn on or off as N-type transistors, the logical effort of a NAND gate, NOR gate, multiplexer, and XOR gate are $\frac{4}{3}$, $\frac{5}{3}$, 2, and 4, respectively.

In some situations one can resize the transistor strengths of a complex logic gate to reduce its logical effort for targeted output transitions. For example, recent work by Swetha Mettala Gilla *et al*. on sizing mutual exclusion elements resizes two frequently used and referenced arbiter designs, giving each a logical effort less than 1 for uncontested grants [4]. The resized designs provide least uncontested grant delay, making the common case—uncontested arbitration—fast.

Similarly, by moving the reset load from the input signal to an amplified output signal, each Proebsting amplifier in Figure 7.6 reduces the logical effort of its first stage inverter. With an input load of $\frac{1}{3} \times 3$ or 1 *each* and drive strengths of 3 for (a) and 2.5 for (b) the first stage inverters in Figure 7.6 have a logical effort of $\frac{1}{3}$ for (a) and $\frac{1}{2.5}$ for (b). Likewise, with input loads of 12 for (a) and 10 for (b) and with drive

[†]Rather than measuring the *logical effort* of a logic gate as (1) how much more load the gate presents at its input than would an inverter of equal drive strength—as we do here—feel free to measure instead (2) how much weaker its strength is if the gate is allowed to present only the same input load as an inverter, or—alternatively—(3) how many times longer than an inverter it takes the gate to drive a copy of itself. These three views are mathematically the same [3].

strengths of 18 and 15 respectively, the second stage inverters of the two Proebsting amplifiers have a logical effort of $\frac{2}{3}$ *each*. By reducing the logical effort of each stage, the two Proebsting amplifiers achieve outstanding amplification.

We use logical effort to design fast CMOS circuits, with a guiding principle: *The fastest logic equalizes the product of logical effort and amplification in all stages.* This guiding principle translates any cost increase in doing logic, compared to an inverter, into a corresponding decrease in amplification.[‡] For more background on the use of logical effort to improve circuit performance, see references [3,5,6].

7.1.5 Summary and conclusion of Section 7.1

Ring oscillators set the pace of self-timed circuits. Although rings of three logic gates are possible, we prefer to use slower but easier to design and more robust rings of five or a larger odd number of logic gates. Rings of five logic gates oscillate at ten gate delays per cycle. Globally clocked networks running as fast as that are unlikely due to the difficulty of amplifying short pulses. Section 7.3 in this chapter describes the Weaver—a self-timed on-chip network manufactured in a 40 nm CMOS technology that operates at the speed of a five-gate ring oscillator. We designed this high-speed self-timed network and the various circuit components in it in accordance with the logical effort guideline described in Section 7.1.4.

The circuit components in the Weaver are partitioned into *Links*, which store and transport local data and state information, and *Joints*, which compute on the data and state information in their Links and control the flow and distribution of the locally computed results and state updates. The information exchange from each individual Link to a Joint and back to the Link forms a ring oscillator. In the Weaver, each Link-Joint pair forms a five-gate ring oscillator. Each such ring oscillator generates a five gate delay low-to-high-to-low pulse signal that is amplified and then used to capture results and state updates computed by the Joint and stored by the Link. The Weaver uses simple pulse generation and amplification techniques.

This section broaches the topic of simple as well as advanced pulse management techniques because different designs require different techniques. The Weaver could use simple techniques because (1) all its ring oscillators operate at the same speed, (2) its data items are only 72 bits wide, and (3) the routing logic in each of its Joints has sufficiently low logical effort to leave adequate amplification to "locally clock" each Link.

Many of the design and test aspects for the Weaver are built in from the bottom up—starting at the level of individual Links and Joints. We therefore added an intermediate section, Section 7.2, on Links and Joints.

[‡]In Section 7.1.3 on page 119, we gave an alternative but mathematically identical guideline for achieving fastest delay by equalizing the step-up in all stages. The two guiding principles are identical because the logical effort guideline—equalizing the product *(logical effort* $\times$ *amplification)* in all stages—is, according to the definitions of logical effort and amplification on pages 123 and 119, respectively, the same as equalizing *((input load/drive strength)* $\times$ *(load driven/input load))*, that is, *(load driven/drive strength)*, which—according to the definition of step-up on page 119—amounts to equalizing *step-up* in all stages.

7.2 The Link and Joint model

As do many asynchronous or self-timed circuit designers, we too started out with a favorite set of handshake components that we compiled to and whose function and timing we validated. The moment we started working with two self-timed circuit families, Click and GasP, each with its own handshake protocol, the compilation process got out of hand. Both families had fixed initialization circuitry built into each component to set the initial states of their handshake signals. Differences in initialization, handshake signaling, and static timing required Click and GasP specific code duplication at various levels in the compiler, and made the resulting compiler more complex and less useful [7]. Each additional initialization version of otherwise identical components multiplied our design and validation efforts. While modeling and validating timing constraints for flow control components in Click we noticed that the components used the same handshaking parts for each handshake interface and we were incidentally validating these parts over and over again. Modeling them for each component was useless as well as harmful: we spent significant amounts of time on managing the complexity of the models [8,9]. What went wrong?

Section 7.2.1 analyzes what went wrong and motivates the various steps that we took to make things right [10]. We illustrate these steps on Click and GasP circuits. Figure 7.7 shows the handshake protocols for Click and GasP, and Figure 7.8 shows what the circuits looked like when we started working with them [1,7,11].

7.2.1 Communication versus computation

The fact that validation of different flow control components in the same family results in repeated validation of the same communication circuits suggests that we combine too much in one component. As reference for what is in a component, see Figure 7.8. So, let us separate communication from flow control. We will combine the communication circuits for the same handshake signals, including their initialization circuitry, into a separate component, called *Link*. We will keep the remaining circuits in the original component and call the remainder a *Joint*. By placing the Click and Gasp handshake communication circuits in their own Links, we move their differences from the interface to the internals of each Link. As a result, the interface between Links and Joints can no longer distinguish Click from GasP, and the Joints become the same for Click and for GasP. Thus, by separating communication from flow control, we (1) reduce the complexity as well as the amount of validation required, and (2) make way for a single compilation strategy to Links and Joints that works for both families.

We see Joints as the places where Links meet to exchange information. This information can be dataless to serve as a mere synchronization means, or it can involve data from several Links that the Joint computes on and for which it distributes results to other Links. Thus, in addition to managing flow control, Joints compute. Computations work on data. Who stores the data—the Link or the Joint? In our old design approach, with the communication and computation circuitry

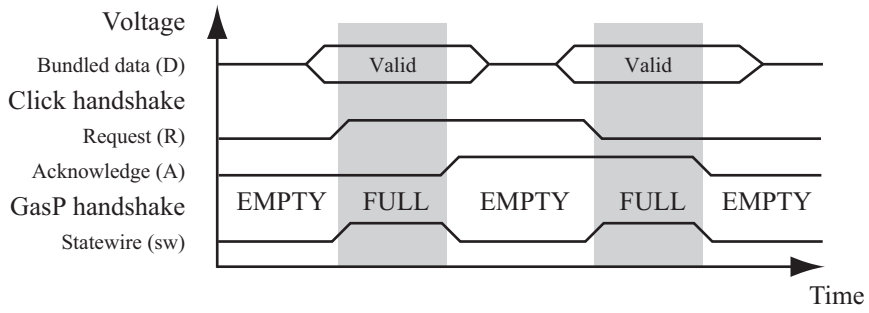

*Figure 7.7 Handshakes protocols for Click and GasP. Self-timed circuits in Click and GasP use bundled-data two-phase handshakes. Click has two handshake signals, request (R) and acknowledge (A). GasP has one, called statewire (sw). These signals tell the receiver when the bundle of data wires sent along them carry valid data. Data are valid in Click when R and A differ, and in GasP when sw is high. In the reverse direction, the handshake signals tell the sender when there is space for new data. In the Link and Joint model we view a handshake protocol as a way to encode the presence of data and space, and we focus on **what** rather than **how** it encodes. So, rather than using R, A, and sw, we use FULL and EMPTY, and we fill a communication Link to make it FULL, and drain it to make it EMPTY. In terms of FULL, EMPTY, fill, and drain, Click and GasP protocols are identical*

residing in the same component, the component stored the data, and the handshake signals were just wires that transferred the data values. In the new design approach with Links and Joints, we made the decision to make the Link both transfer and store the data. As a result, Links and Joints not only separate *communication and storage* from *computation and flow control* but also separate *states* from *actions*.

Figure 7.9 shows what Click and GasP circuits look like in terms of Links and Joints [10]. Except for MrGO, an AND-like gate to be introduced in Section 7.2.2, the two circuits are identical to those in Figure 7.8 for the old design approach—we merely moved the interface! Each Joint in Figure 7.9 acts when its input Link is FULL **and** its output Link is EMPTY. When it acts, it copies the data from input to output Link and it drains the input Link and fills the output Link. The Links respond by changing their FULL or EMPTY states, thus invalidating the conditions for the Joint's action which causes the action to complete its copy, drain, and fill operations—see also Figure 7.11.[§]

[§]Figure 7.11 uses stick figures for Joints and rectangles for Links to illustrate how the Joint acts, how the Links respond, and how the Link responses invalidate the conditions of the Joint's action. The *go* signal used in Figure 7.11 is normally high. It plays a role in initialization and test, as explained in Section 7.2.2.

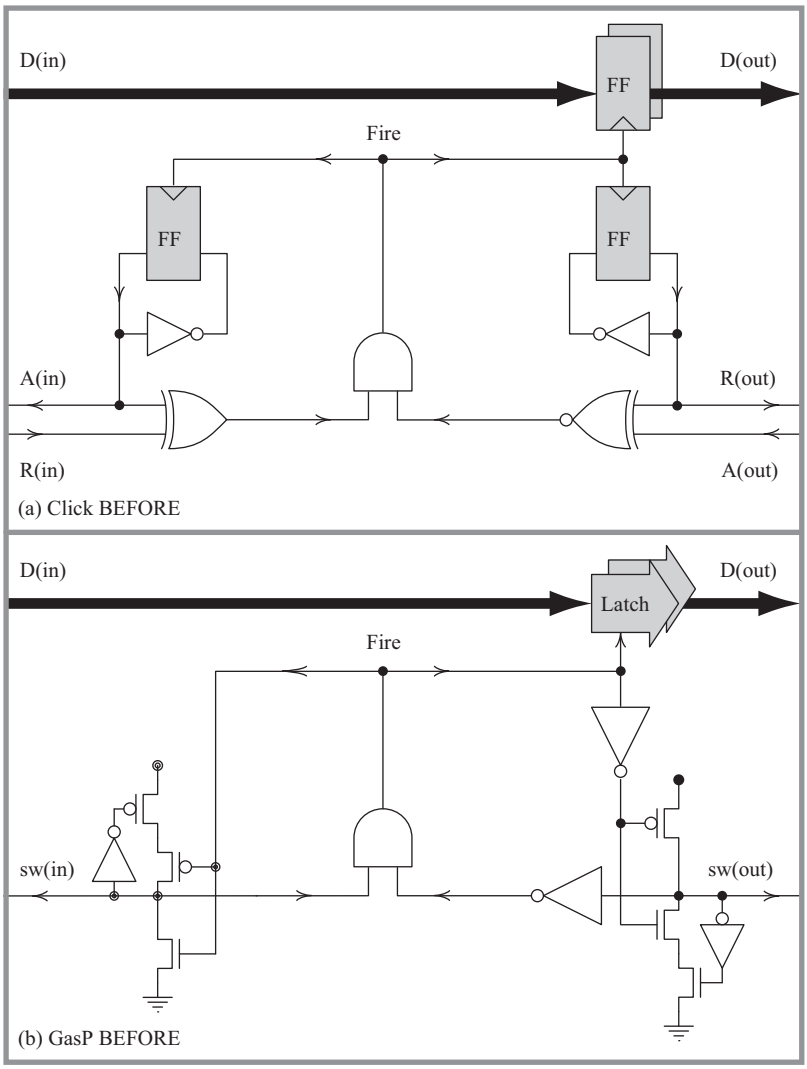

Figure 7.8 A Click and GasP component before the Link and Joint model. Simple circuits in Click (a) and GasP (b) omitting initialization and amplification. Each acts when (1) its input is FULL, that is, R(in) $\neq$ A(in) in Click and sw(in) is high in GasP, and (2) its output is EMPTY, that is, R(out) = A(out) in Click and sw(out) is low in GasP. When it acts, it copies and stores D(in) on D(out), makes in EMPTY and out FULL

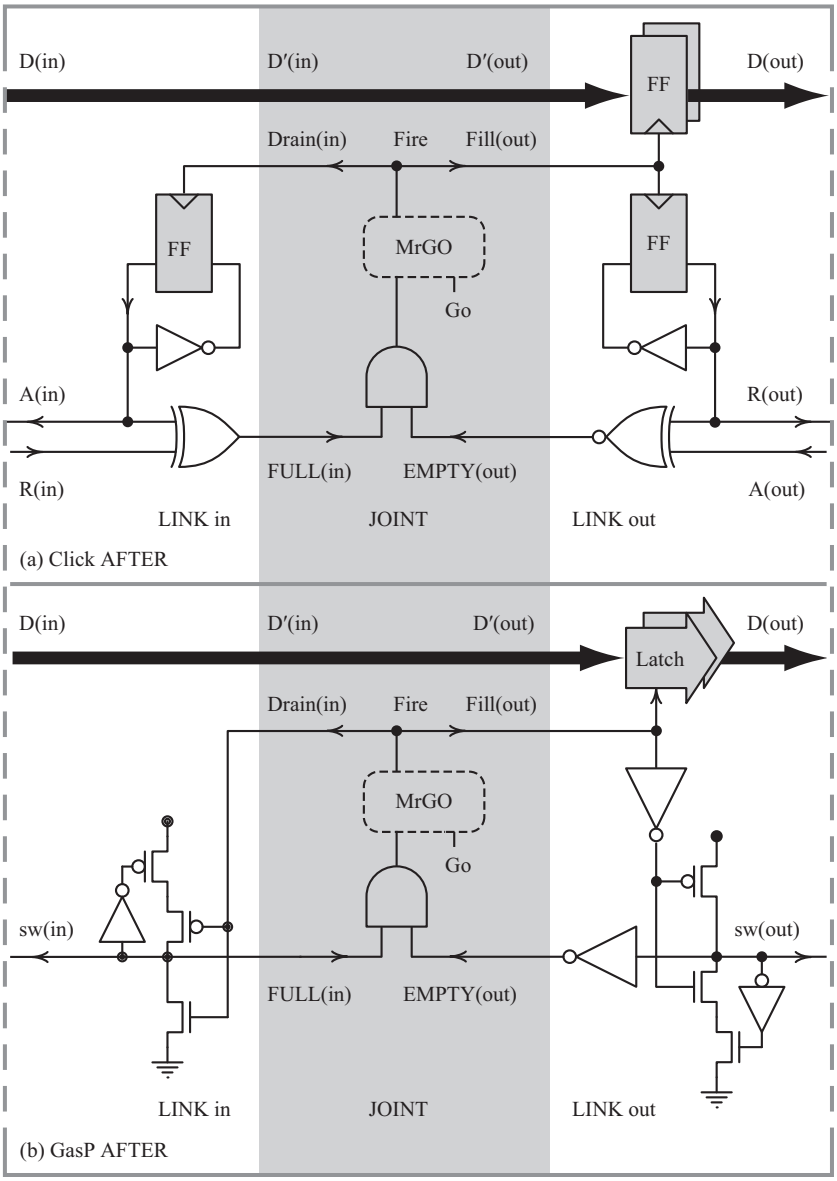

Figure 7.9 Click and GasP as Links and Joints. We moved the interface from the handshake signals to what they encode, FULL, EMPTY, fill, drain. We stored the data in the Link together with the FULL or EMPTY state. The pictures show only half of each Link. Each complete Link looks like the two pictured half Links put together. The Joint's AND function includes MrGO, an (arbitrated) AND gate—see Figure 7.10

We describe the behavior of Links and Joints using guarded commands [12,13]. Links and Joints communicate via the FULL, EMPTY, fill, drain, and data signals at their interface. Because the interface signals remain available until the Link changes its FULL or EMPTY state, the communication protocol uses shared variables rather than message passing. As a result, "probes," that check whether or not a Link can communicate [14] and that require special primitives in a message-passing model, are just guards on the Link's FULL or EMPTY state.

7.2.2 Initialization and test

Differentiating states from actions turns out to be key in enabling initialization for design *and* test. For design, fixed initialization circuitry would suffice. But for test, arbitrary circuit initializations may be required, especially if we wish to accommodate unanticipated debug scenarios. So, why bother following the old design approach by building in fixed initialization circuitry that will be used only once, when the circuit starts up? Why not use existing test methods also to initialize the circuit at start-up? There may even be an additional advantage in terms of security if instead of having the circuit initialize itself automatically, initialization takes a separate step—one that is hard to accomplish successfully by accident and would take substantial time to accomplish through trial and error. This section explains how test access to individual actions and states has helped us not only to initialize the Weaver design discussed in Section 7.3 but also to test and debug the Weaver—at speed.

7.2.2.1 Action control: go and MrGO

It is good practice never to let both the design and the test environment initialize the same state at the same time. Letting them initialize different states at the same time may be fine and perhaps even desirable because the self-timed design propagates states faster than the test environment. During initialization, we disable Joint actions that use Link states set by the test environment. Remember that the Links store the states, and the Joints take the actions. To disable an action, we add an extra condition, called *go*, which we control via the test environment. Each Joint in Figure 7.9 now acts when its input Link is FULL **and** its output Link is EMPTY **and** *go* is high. The original condition "input Link FULL **and** output Link EMPTY" no longer suffices: a low *go* signal disables the action—see Figures 7.10 and 7.11.

Thus, by making the *go* signal of every Joint action low, the test environment can disable all circuit actions and safely initialize FULL or EMPTY Link states and data stored in Links anywhere in the design. For the Weaver, we shift the initial values into the chip serially and bit by bit, using a chain of shift registers also known as a *scan chain* [15]. Each shift register is associated with a particular state signal in the design. When all values are shifted into position, the test interface writes them in parallel into the associated Link states and data bits—see Figure 7.12.

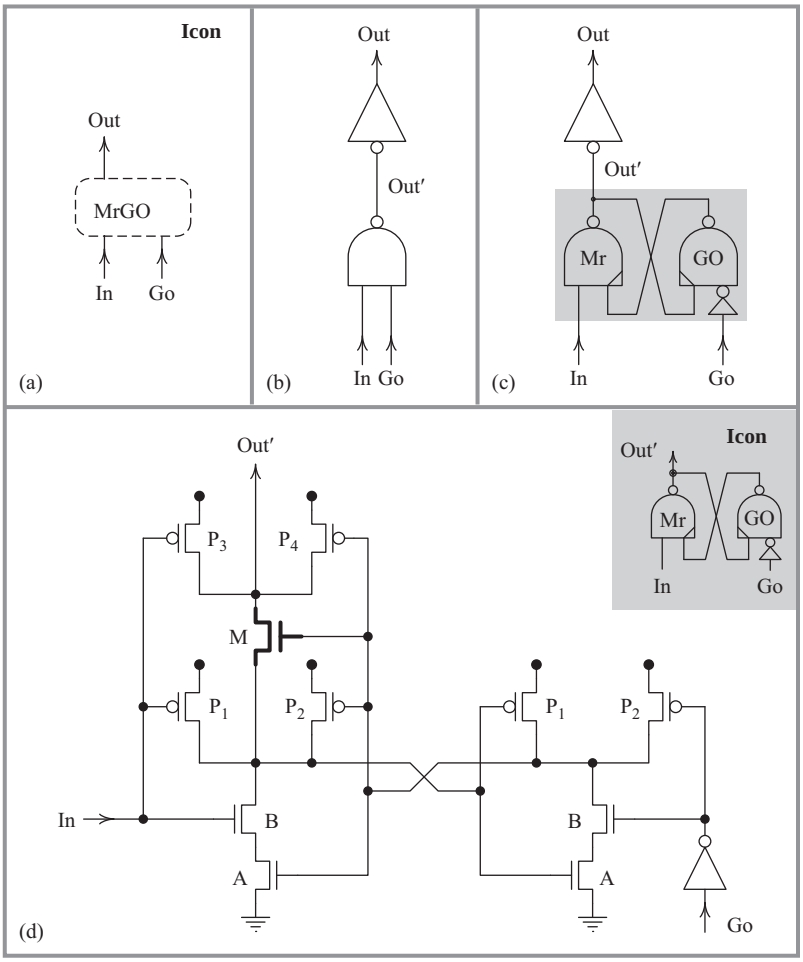

Figure 7.10 *go and MrGO. We use go signals to enable or disable actions. We combine them with the other action conditions through either a simple AND gate (b) or an arbitrated AND gate (c). The arbitrated AND gate, called MrGO and pronounced "Mister GO," is implemented as in (c) and (d). We use the name MrGO with the icon in (a) when the decision which version to use is still open. MrGO arbitrates between a high in signal, to continue the action, and a low go signal, to stop the action. The arbiter's bold transistor in (d) delays active-low grant signal out′ by conducting only after metastability ends. We size the transistors for the common uncontested case to reduce the logical effort from in to out′ [4]. The various pull-up transistors keep out′ from floating. The circuit normally operates with a high go signal*

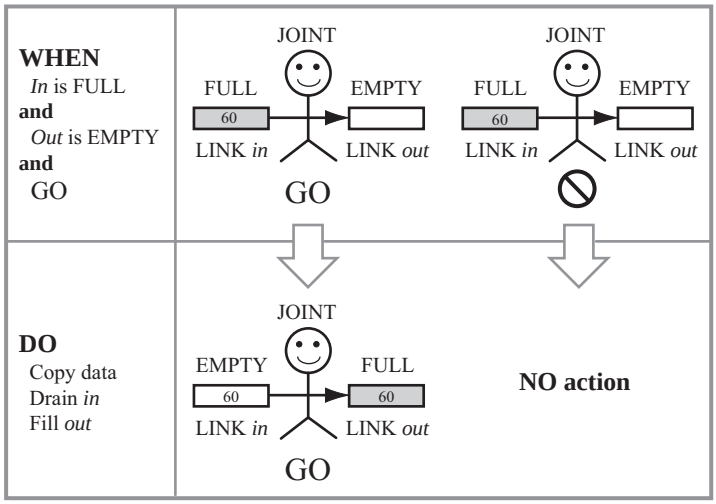

Figure 7.11 *Joints act under go control on Link states. This picture illustrates the action of a simple Joint, like the one in Figure 7.9(a) or (b), and the responses of its two Links. The stick figure represents the Joint. The rectangles represent the Links. FULL Links are colored gray, EMPTY Links lack color. The Joint acts only when (1) its input Link is FULL and (2) its output Link is EMPTY and (3) it has permission to act, that is, its go signal is high. When it acts, it copies the data, with value 60, from its input Link to its output Link, drains its input Link, and fills its output Link. The input Link responds by declaring itself EMPTY. The output Link responds by storing the data and declaring itself FULL. Their responses disable and complete the Joint's action, and may enable actions in neighboring Joints. Note that the data in the input Link are unaffected by the drain operation. The data with value 60 will remain in the input Link until a follow-up fill operation or a test write operation changes them*

For design initialization, the next step is to make every *go* signal of every Joint action high and thereby start the circuit.

Test methods often use a single *go* signal, typically called *test mode*, to start or stop the circuit. This may work fine when circuit actions are synchronized under global control, but is far from ideal when they are asynchronous, widely distributed, and local. As a thought experiment, try to follow a burst of data items through part of a self-timed design—at speed. Any global control to "walk the burst" will conflict with the "at speed" nature of the experiment. The only candidate qualified to run this experiment at speed is the circuit itself, running self-timed. So, if we could enable all Joints within that part and disable the Joints outside it, then a burst of data would run through that part at speed—and after it has run its course, we could

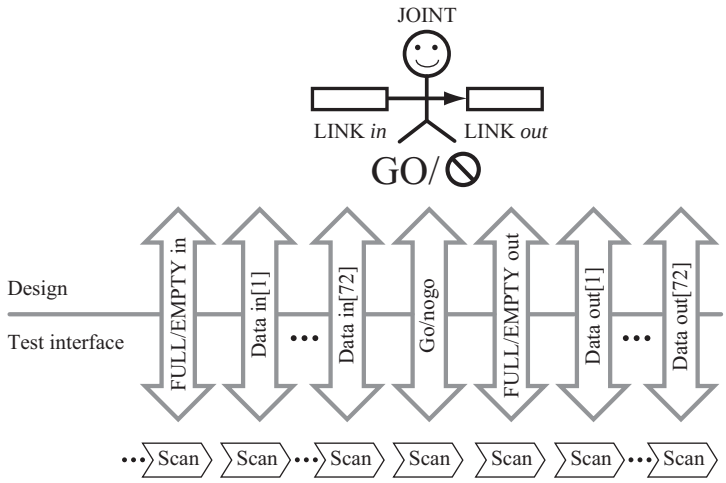

Figure 7.12 Link and Joint scan test interface. A series of shift registers, called a scan chain, shifts bits in and out of the circuit serially. The scan chain (bottom) can shift while the design (top) operates. A scan chain reads or writes the design's data bits, FULL or EMPTY states, and go signals in parallel. To avoid interference when setting up states or actions, we use separate scan chains for states and go signals

read and scan out the states it left in its wake. Guess what. We can do exactly that by giving each Joint action its own *go* signal.

In the Weaver, we use many separate *go* signals, one per Joint. In principle, there are as many *go* signals as actions. To control that many *go* signals, we shift them in using a scan chain—hence the *go/nogo* test interface and scan shift register in Figure 7.12. Note that Figure 7.12 can read as well as write *go* signals, Link states and data bits, and shift their values in as well as out through the scan chain. To avoid interference between controlling actions and reading or writing states, the Weaver has separate mutually exclusive scan chains for *go* signals, for FULL or EMPTY Link states, and for data bits.

Figures 7.13 and 7.14 show examples of the "thought experiment" discussed earlier. The examples run a burst of data items at speed through a part of the design that has a counter attached to it. They test whether the counter can keep up and count the correct number of data items passing by. We did similar test experiments on the Weaver chip, whose counters are located in the NE corner of the floorplan—see Figure 7.16. The test environment can read the Weaver counters, but only reset them to zero. This was a mistake, acceptable in a mere test chip with a transparent circuit-versus-performance relationship, but unacceptable otherwise. Counters are sufficiently important for characterization purposes to justify full scan support for both reading and writing.

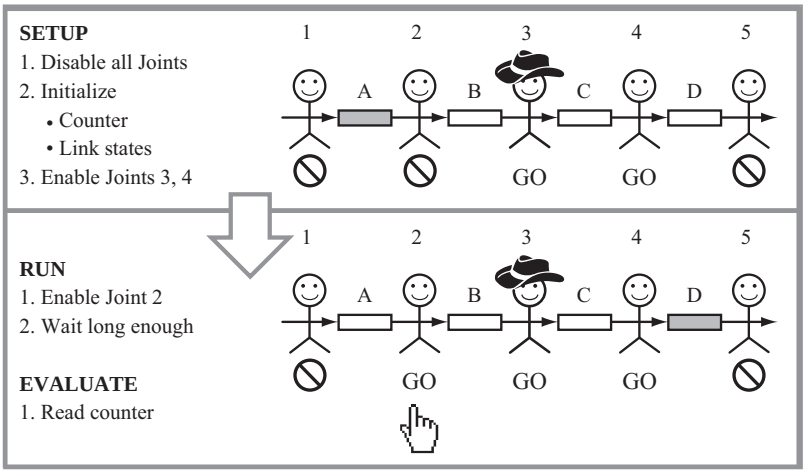

Figure 7.13 Counting one data item at speed. The counter is attached to Joint 3 with the cowboy hat. The test setup leaves several adjacent Joints enabled to permit data to pass through them at speed. Disabled Joints upstream and downstream of the test part prevent entry of other inputs and escape of results. Enabling "gate keeper" Joint 2 releases the test data to flow through the test part at speed and update the counter

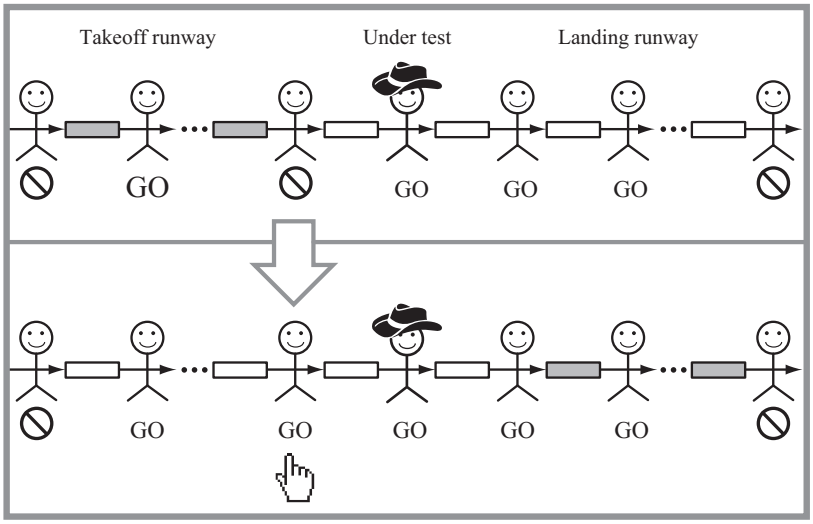

Figure 7.14 Counting a burst of data items at speed. With longer takeoff and landing runways one can run more data past the counter at speed

The Links and Joints in Figures 7.13 and 7.14 are similar to those in Figure 7.9 and operate as illustrated in Figure 7.11. Note that the test segments are bounded upstream and downstream by disabled Joints. Most test interface tasks, including circuit initialization and manufacturing testing against, say, stuck-at faults, work on bounded segments. Some though, notably tasks related to performance characterization, require the circuit to run freely. It is for these tasks and our ability to stop them cleanly, without corrupting circuit states, that the *go* signal has its own arbiter. The arbitrated *go* signal, shown in Figure 7.10(c) and (d), is called *MrGO* and pronounced "Mister GO" and arbitrates between continuing or stopping an action.

The arbitrated version of MrGO in Figure 7.10 must "crown" the AND function for the Joint action, that is, all FULL or EMPTY state signals used in either the guard or the command of the action must be AND-ed *before* we AND the *go* signal. If the guard contains data bits, then these may be AND-ed either before or after MrGO—the FULL state signals of their Links already cover them. The Weaver adds the data bits after MrGO, as can be seen from Figures 7.21 and 7.23(a) and (b). Giving MrGO a "crown" position guarantees a nonblocking arbitration that allows the *go* signal eventually to grab the arbiter, either the first time or the next.[¶] If not the first time, because the arbiter favors a high *in* signal over a low *go* signal and thus continues the action, then the action will release the arbiter one cycle later—without blocking. Because the cycle time for new actions is longer than the arbiter's uncontested grant delay, the arbiter will grant *go* next and thus disable further action until released by a high *go* signal. Any MrGO position other than a "crown" position may allow a temporary state to grab MrGO and may inject actions that interfere with the initialization or test run whenever this state changes.

To obtain the throughput and power measurements reported in Section 7.3.5 for the Weaver chip, we let the self-timed circuit run freely for, say, 10 seconds and then read out the counters. The corresponding test setup resembles Figures 7.13 and 7.14: first disable all Joints, then EMPTY all Link states, then enable all but two Joints—the "gate keeper" before the reloader Link, and the reloader Joint after the reloader Link. Reloaders are the only stages in the Weaver where we can scan data bits in and out. They are in the SE corner of the floorplan—see Figure 7.16. We repeatedly scan a data pattern with FULL Link state into the reloader Link, and temporarily enable the reloader Joint to copy the data forward. The data will queue up behind the "gate keeper." When all scan inputs are delivered, we reset the counters, then enable "gate keeper" and reloader, let the circuit run for 10 seconds, disable the "gate keeper"—using MrGO—let operations peter out, and then read the counters.

[¶]We ignore the fact that the arbiter itself can take arbitrarily long to decide which of two contesting inputs to grant. In practice, arbitration time is less relevant for initialization and test. We can avoid arbitration in general for design initialization. We can avoid MrGO arbitration for tests on bounded segments as in Figures 7.13 and 7.14. We can filter out rare arbitration delays by running performance tests multiple times.

7.2.3 Summary and conclusion of Section 7.2

By distinguishing communication, done in Links, from computation, done in Joints, we obtain a simple interface that unifies both Click and GasP as well as many other self-timed circuit families and the compilation and verification tools around them. By also distinguishing states, stored in Links, from actions, performed by Joints, we obtain a simple model of computation that works for computer scientists and electrical engineers alike—a working relation that we intend to explore further.

Traditional scan test access to individual states combined with *go* and MrGO control of individual actions allows clean initialization as well as at-speed test, debug, and characterization. In addition to playing a key role in test, MrGO can also be used to gradually synchronize a self-timed design to a clock domain [16].

Section 7.3 details the Weaver's Links and Joints and their scan connections. The details include amplification needs and cycle times as discussed in Section 7.1.

7.3 The Weaver, an 8 × 8 crossbar experiment

The Weaver is a self-timed 8 × 8 crossbar switch built in 40 nm CMOS by TSMC. The Weaver's crossbar steers individual data items from any of eight input channels to any of eight output channels. Local arbitration throughout the crossbar resolves internal contention without blocking—on a first-come-first-served basis that is fair to the loser—so that only input and output channel capacity limit throughput. Without contention, a data item can pass through the crossbar in less than one nanosecond. Without contention and at nominal 1.0 volt power supply, each channel can pass about 6 Giga data items per second through the crossbar at less than half a watt. Data items are 72 bits wide, giving the crossbar's eight channels a maximum combined throughput of nearly 3.5 Tera bits per second. In the absence of traffic, only leakage consumes energy. The Weaver runs without a clock.

The crossbar on the Weaver chip occupies an area of about a tenth of a square millimeter in a triangle 433 micrometers by 391 micrometers in size. The triangle contains a triangular array of 56 switches, one for each possible channel to channel connection. The Weaver chip places the switches in close proximity and provides recirculating first-in-first-out (FIFO) rings to connect the crossbar outputs back to the crossbar inputs for extended high speed testing. For a network-on-chip application, one would distribute switches like those in the Weaver geographically to form the on- and off-ramps of a freeway-like data network.

The following sections discuss the design and test features of the Weaver from a logical, electrical, and layout floorplanning point of view. Section 7.3.1 discusses the architecture of the Weaver. Section 7.3.2 shows the key circuits used in the Weaver. For consistency, the drawings in both sections show the architecture and circuits from the Weaver's floorplanning point of view. Sections 7.3.3 and 7.3.4 discuss test logistics and Section 7.3.5 reports performance measurements from the Weaver chip. The final Section 7.3.6 concludes this chapter by summarizing where the Weaver's logical, electrical, and layout views differ—and why.

7.3.1 Weaver architecture and floorplan

Figure 7.15 shows a schematic diagram of the Weaver. Eight self-timed channels, each with 48 Links, form FIFO rings that recirculate data from the output of the 8×8 *crossbar switch*, also called *crossbar*, back to its input. Two additional channels bypass the crossbar to provide a performance contrast. The two bypass channels flank the recirculation channels. The ten rings are labeled in the floorplan of Figure 7.16. One bypass channel, Ring 0, has 48 Links, but the other, Ring 9, has only 40 Links.

In the floorplan long rectangles represent Links and black dots represent Joints. Arrows connecting the dots and rectangles indicate the direction that data flow. The floorplan arranges the Joints in rows numbered 1 to 20 from bottom to top and in columns with letters *A* to *Z* from left to right. The medium- and dark-gray triangle in the North West (NW) identifies the crossbar switch itself. Data enter the crossbar from the South at row 12, going North, and leave the crossbar at column *K*, going East. The diagonal NW edge of the crossbar (*Double-barrel Ricochet*) folds its datapaths so they turn their data from a Northbound to an Eastbound direction.

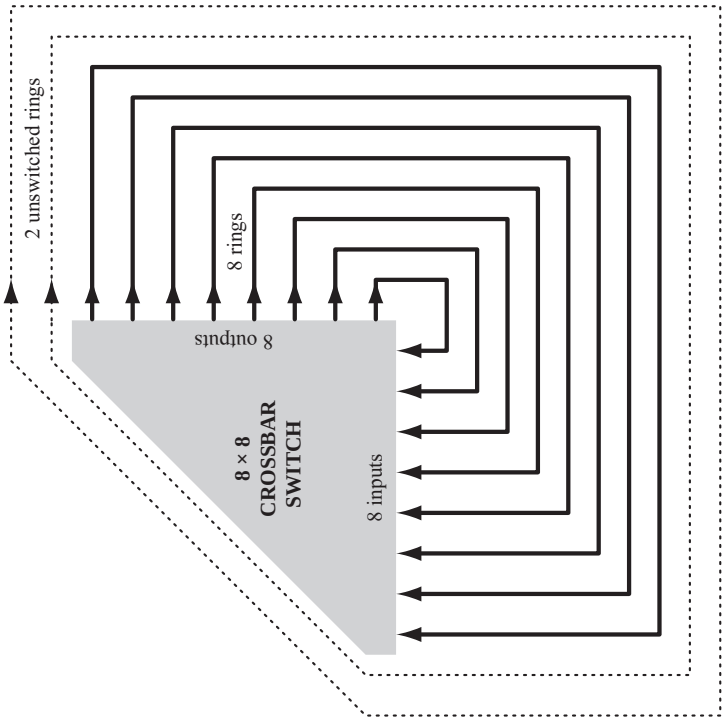

Figure 7.15 Weaver diagram, rotated to match the orientation of Figure 7.16. The crossbar switch is in the gray triangle. Eight rings, 1–8, recirculate data from its output back to its input. Two extra rings, 0 and 9, bypass the crossbar switch to measure relative performance

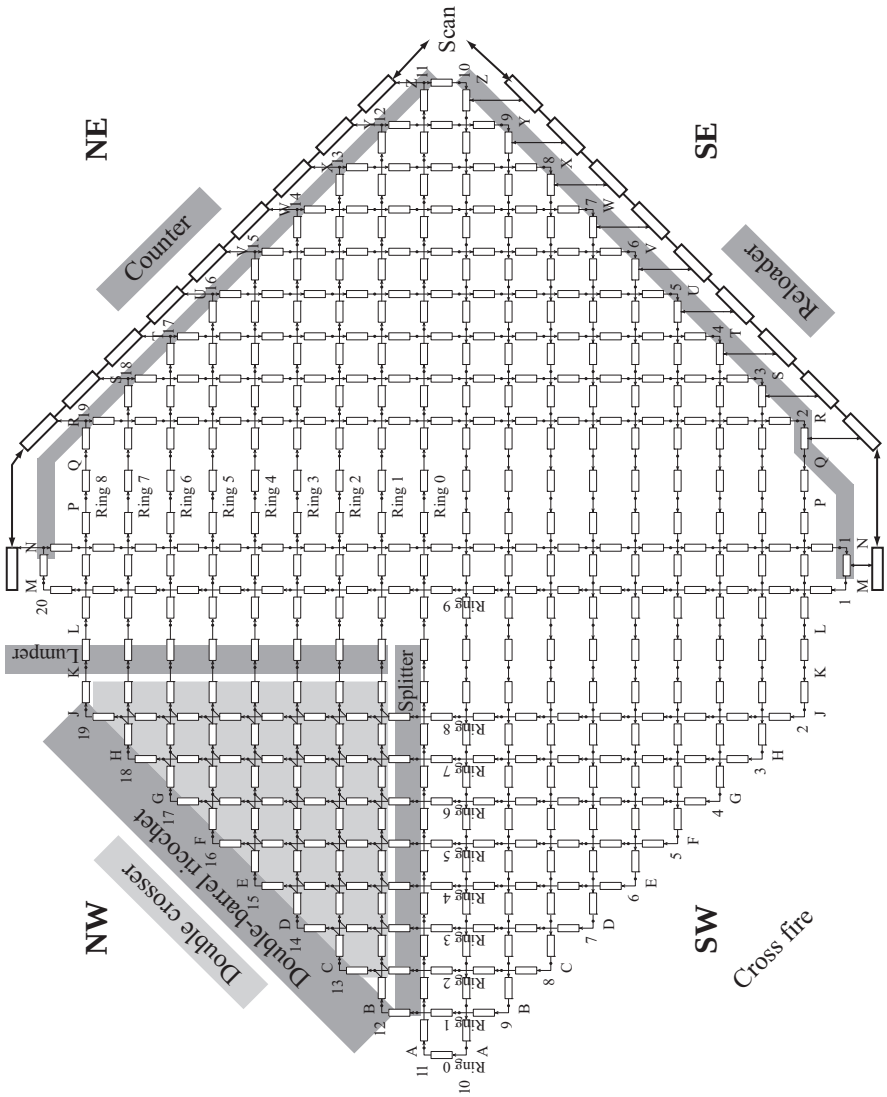

Figure 7.16 *Weaver floorplan, rotated 90 degrees. The Weaver includes ten rings, of which eight recirculate data from the output of the crossbar (NW corner) back to its input. Each of the ten rings has a reloader stage (SE edge) to read and write data from and to one of its Links, and a counter (NE edge) to tally the actions taken by one of its Joints. Scan chains (SE and NE) control the reading and writing of data and the tallying and resetting of ring counters. Other scan chains, omitted here, control the FULL or EMPTY state of each Link and enable or disable the go control signals of each Joint*

Each of the ten rings forms a rectangle through which data circulate clockwise. Ring 0 is wide, stretching between columns *A* and *Z*, and not very tall, going from row 10 to row 11. Ring 1 is a little narrower, stretching between columns *B* and *Y*, and a little taller, going from row 9 to row 12. Successive rings are narrower and taller until Ring 9 is very narrow and very tall, occupying columns *M* and *N* and rows 1 to 20. The rings fold on 45-degree diagonals at the edges of the floorplan—like a ribbon cable. The North East (NE), South East (SE), and South West (SW) ring sections collectively named *Cross Fire* identify simple FIFO pipeline circuits that form most of each ring. Note that even though horizontal and vertical pipelines in the *Cross Fire* sections may cross each other they are entirely independent.

Each ring includes a counter stage to measure throughput. The counters lie along the North East (NE) diagonal edge of Figure 7.16. Each ring also includes a reloader stage to insert, overwrite, or read data values. The reloaders lie along the South East (SE) diagonal edge of Figure 7.16. The Weaver is so named because data items can weave complex paths through its eight middle channels. Many of the part names of the Weaver, such as *Double-barrel Ricochet* and *Cross Fire*, adopt gunslinger lingo from Western movies.

7.3.1.1 Crossbar switch

Instead of a rectangular structure with $N \times N$ switches the Weaver's crossbar switch has a triangular structure with $N \times (N - 1)$ switches and N repeaters. The triangular structure of the crossbar minimizes its wire length and simplifies layout of the rings that recirculate data through the crossbar at high speed.

Figure 7.17(a) illustrates the structure of the Weaver's crossbar. Data items entering from the South on any of eight input channels exit to the East on any of eight output channels. Note that Figure 7.17 shows fewer channel connections. Because the datapath folds diagonally like a ribbon cable, each of the eight input channels crosses every other input channel in the crossbar exactly once. The arrows inside the box at each crossing suggest how the switches at each crossing allow data items to change from one channel to another. The repeaters that fold the datapath at the diagonal edge of the crossbar are *Double-barrel Ricochet* modules described later.

A complete 8×8 crossbar must have 8×7 or 56 individual switches, called *Crossers*. The Weaver arranges these in pairs, called *Double Crossers*, one pair at each of the 28 crossings. Figure 7.17(b) and (c) illustrates one *Double Crosser* while Figure 7.17(d) shows how the *Double Crosser* appears in the floorplan of Figure 7.16. One of its switches serves its North output, and the other one serves its East output. Each *Double Crosser* accepts data items from the South or West and delivers them to the North or East. Conflict in the crossbar happens only if two data items try to leave a *Double Crosser* concurrently by the same exit. Each switch, or *Crosser*, has a mutual exclusion element, or arbiter, with metastability protection [4,17] to resolve exit conflicts on a first-come-first-served basis that is fair to the loser. Although metastability may delay the passage of a data item, the self-timed nature of the Weaver renders such delay harmless. Metastability delays tend to be so rare and short that we expect them to be unnoticeable.

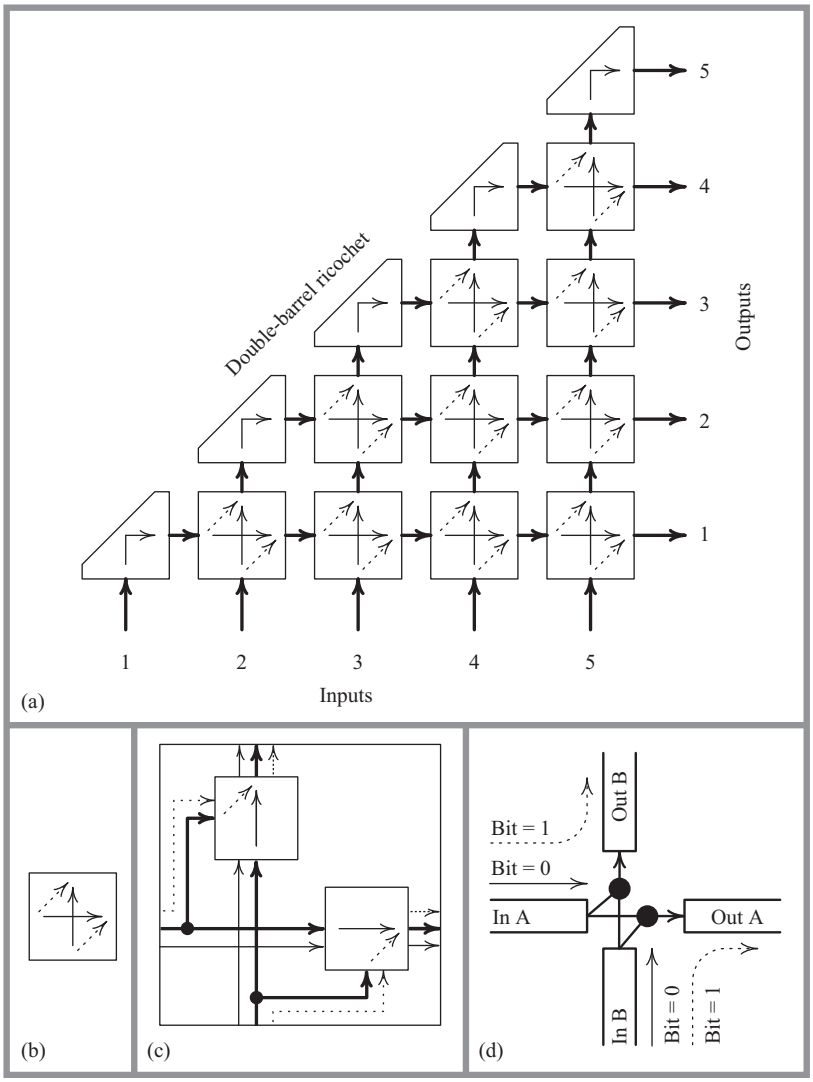

Figure 7.17 *Weaver crossbar switch. The 8 × 8 crossbar switch is a triangular structure with 28 Double Crossers, of which some appear in (a). Each Double Crosser (b) contains two Crossers (c) that arbitrate between competing data items that go in the same direction. The direction is encoded in the data, in a Double Crosser specific steering bit. Data go straight when this bit is 0, and turn otherwise. Link-Joint version (d) of the Double Crosser as used in the floorplan of Figure 7.16 connects four "rectangle" Links and two "black dot" Joints*

7.3.1.2 Steering bits

Steering bits in each data item control how the data item passes through the Weaver. Because each data item carries its own steering bits, each data item weaves its own individual path through the crossbar. To simplify decoding, the Weaver assigns an individual steering bit for each of the 28 *Double Crossers* in the crossbar switch. In other words, 28 of the 72 data bits in each data item may function as steering bits. The remaining 44 data bits are entirely free of assigned meaning in any data item.

Each steering bit applies to a singular *Double Crosser*. Regardless of how it enters a *Double Crosser*, a data item with a 0 in the steering bit position for that *Double Crosser* goes straight through it—West to East or South to North—staying in the same channel. A data item with a 1 in that steering bit position turns either from West to North or from South to East, changing to the other channel.

So, at each *Double Crosser* a data item either remains in its channel or changes to the other channel. Thus, per channel, a data item requires seven steering bits, one for each of the other channels to which it might change. The remaining $72-7$ or 65 bits can carry arbitrary data, though, of course, some of those 65 bit positions may be used as steering bits after the data item switches to another channel.

This choice of rules for steering simplifies testing by forcing every data item to follow a closed path. For instance, data items with 0 in all steering bits keep circulating in their initial ring. Data items with a single 1 in the steering bit position for an intersection of two rings circulate alternately around those intersecting rings. Data items with multiple 1's in steering bit positions weave through several rings in succession, following a closed path that may be less intuitive at first sight. A different application might use other steering rules by decoding a destination address.

7.3.1.3 Test infrastructure: scan, counters, and reloaders

Control of Weaver experiments is entirely through an industry standard low-speed JTAG test interface and scan chains [7,15]. The scan chains serve three purposes.

- First, the scan chains can read or clear the throughput values in each of the ten 54-bit counters, one per ring. These counters appear at the North East (NE) edge of Figure 7.16, and in more detail in Figure 7.18.
- Second, the scan chains can write a data value into or read a data value from a data item in each of the ten reloaders, one for each ring. The reloaders are located at the South East (SE) edge of Figure 7.16.
- Third, because the Weaver implements *Naturalized Communication and Testing* as described in [10] and Section 7.2, the scan chains can stop the flow of data, sense the FULL or EMPTY state of every Link, initialize or reinitialize the FULL or EMPTY state of every Link, and restart the self-timed flow.

In addition to the low-speed JTAG-controlled scan chains, the Weaver has two dedicated medium-speed output pins that deliver reduced-frequency real-time signals from the counters. These reduced-frequency outputs switch at 2^{-20} or approximately one millionth of the throughput rate of the rings. Two series of

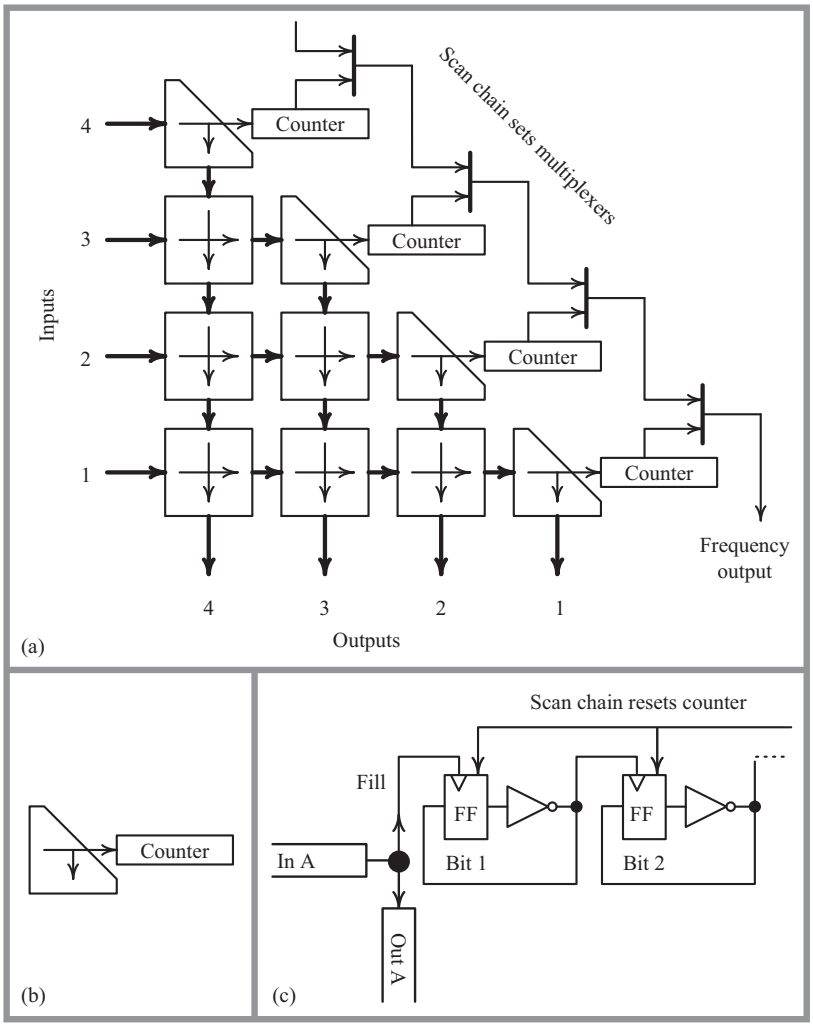

Figure 7.18 Counters. Each of the ten rings in the Weaver has a counter of which some appear in (a). Bit 19 of each counter provides a frequency output for the ring's monitored fill signal (b) divided by 2^{20} or about a million to view in real time on an oscilloscope. The scan chain selects which output is delivered off chip. The counters are implemented as ripple counters (c) that store each bit in a flipflop that uses its inverted output as its input. Each flipflop operates at the rising edge of its clock input. When a bit changes from 1 to 0, its flipflop clocks the flipflop of the next significant bit. The fill signal of the Joint connection to the ring clocks the flipflop of the least significant bit

multiplexers like those in Figure 7.18(a) allow the scan chain to select two rings for frequency output. The frequency outputs permit real-time observation and comparison of throughput.

7.3.2 Weaver circuits

Motivated by the five- versus three-gate ring comparison presented in Section 7.1.2, the Weaver uses the 6-4 GasP circuit family [10], whose circuits run at the speed of a five-gate ring oscillator, rather than the original 4-2 GasP circuits [1] that run at the speed of a three-gate ring oscillator. The use of GasP is a matter of convenience, given that we have both the expertise and design libraries for GasP, and a matter of target speed, given that designs tend to be faster in GasP than, for instance, in Click.

This section shows the Link and Joint organizations and the 6-4 GasP circuit implementations for various Weaver parts. First, Section 7.3.2.1 describes a simple FIFO circuit that copies, stores, and transports 72 bit wide data. Last, Section 7.3.2.3 describes more advanced parts from the crossbar switch that provide data-driven flow control. Section 7.3.2.2, in the middle, focuses on critical paths and on circuit solutions to manage such paths.

7.3.2.1 First-in-first-out (FIFO) circuits

Figure 7.19 shows the 6-4 GasP implementation for the simple FIFO pipeline circuit in the ring sections collectively called *Cross Fire*—see floorplan of Figure 7.16. The gate-level behaviors of the Joint and the two Links in Figure 7.19 are the same as for the Joint and GasP Links in Figure 7.9 in Section 7.2. The implementation in Figure 7.19 is more detailed in that it specifies the inverting—and amplifying—gates so we know exactly how many inversions there are in each self-resetting loop.

Gates *A*, *B*, *C*, *D*, and *Y* form a self-resetting loop with five inverting gates, as do *B*, *C*, *D*, *E*, and *X*. The two loops share three gates—*B*, *C*, and *D*—to form the Joint's AND function, *FULL(in)* **and** *EMPTY(out)* **and** *go*, and to limit delay variations between the loops. Inverters *D*, *E*, *F*, and *G* amplify the output signal of the AND function to drive the large loads presented by the 72 latches in Link *out* and by the driver-and-keeper gates *X* and *Y* in Links *out* and *in*. Figure 7.19 shows only one end of each Link, the end nearest Joint *fifo*. The far end of Link *in* looks like the near end of Link *out* shown in Figure 7.19—and vice versa, the far end of Link *out* looks like the near end of Link *in* shown in Figure 7.19.

Each time Link *in* is FULL and Link *out* is EMPTY and *go* is high, the AND function is asserted, just like in the Joint of Figure 7.9, Section 7.2, and the two loops generate a low-to-high-to-low *fire* pulse of five gate delays to "locally clock" the latches—that is, render them temporarily transparent—so they store the data copied by Joint *fifo* from Link *in* to Link *out*.

Going around each loop, the *fire* pulse also drives *X* and *Y* for five gate delays. For the Links in the Weaver, a five gate delay drive pulse is long enough to drive the state change at the output of *X* or *Y* from rail to rail across the entire Link length. The state change is sensed at the other end of the Link within a gate delay, just like

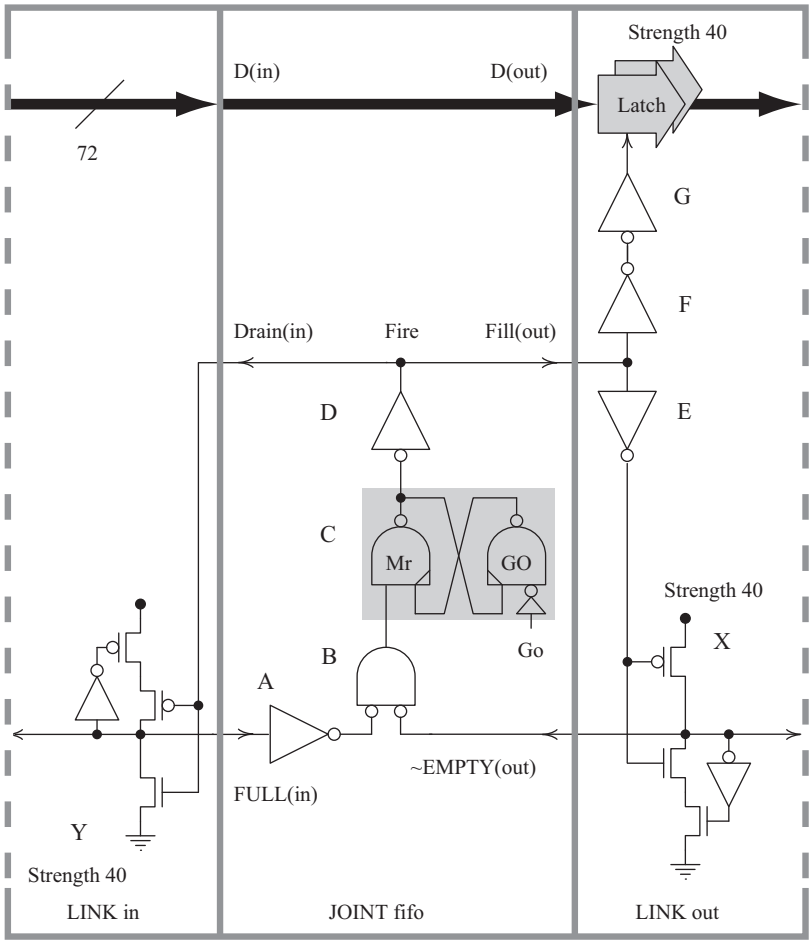

Figure 7.19 FIFO Joint and its two near-end Link connections in 6-4 GasP. This FIFO circuit has the same functional behavior as the Joint with GasP Links in Figure 7.9(b), Section 7.2. With its self-resetting loops, A-B-C-D-Y and B-C-D-E-X, it runs at the speed of a five-gate ring oscillator. We show only one end of each Link, the end nearest Joint fifo. Gray-colored gates are icons—a circuit for MrGO was given earlier in Figure 7.10, the latch follows in Figure 7.20. With the exception of latches all gates invert. To match the inversion count around each loop we use ~EMPTY(out) rather than EMPTY(out) between Joint fifo and Link out. We sized the gates to give each latch and driver-and-keepers X and Y a strength of 40 to help them drive their changes to the other end of their Link within a gate delay

any other gate output change in the FIFO circuit is sensed by subsequent gates within a gate delay. The new FULL or EMPTY Link state is stored by the driver-and-keeper gate at the other end of the Link. To reduce the logical effort of changing a Link's FULL or EMPTY state a driver-and-keeper gate in GasP either drives high and keeps low or drives low and keeps high. This not only creates fast state transitions, it also enables the use of stronger keepers that are more robust to noise.

Note that the asserted AND function is de-asserted after five gate delays, when Link *in* is no longer FULL and Link *out* is no longer EMPTY. The new Link states enable AND functions in neighboring Joints, which in turn fill Link *in* with new data and drain Link *out* and thereby re-assert the AND function in Link *fifo*, and so on.

Fill and drain pulses on the same Link alternate, because one is generated only when the Link is EMPTY and the other only when the Link is FULL. At maximum speed the self-resetting loops run at a cycle time of ten gate delays, barely separating the alternating five gate delay fill and drain pulses on the same Link. Pulse separation is enhanced by the nature of gate *B*—the AND-like gate at the core of each pulse—which turns each pulse on using series transistors and off using parallel transistors. Because parallel transistors are faster than series transistors, each fill or drain pulse turns off before the other turns on. As a result, the driver-and-keeper gate that changes the Link state from EMPTY to FULL or from FULL to EMPTY turns off before the driver-and-keeper gate at the other end of the Link can turn on and change the state back. Separation between fill and drain pulses on the same Link can be enhanced further by threshold shifts in amplifiers *D* and *E* that follow gate *B*.

Because fill and drain pulses on the same Link alternate, latches rather than flipflops can safely hold the data. The pulses play the role of any clock signal that might otherwise have been provided. Because the pulses happen only when needed, "clock gating" is automatic.

The name "6-4 GasP" indicates that (1) the Links are implemented with GasP circuits, that is, with complementary driver-and-keeper pairs and bidirectional state wires, and (2) it takes six gate delays to propagate a FULL state forward from Link *in* through Joint *fifo* to Link *out*—via gates *A*, *B*, *C*, *D*, *E*, and *X*—and (3) it takes four gate delays to propagate an EMPTY state in reverse direction from Link *out* through Joint *fifo* to Link *in*—via gates *B*, *C*, *D*, and *Y*. Together, the forward delay and the reverse delay yield a cycle time of ten gate delays. Note that this cycle time is consistent with the self-resetting of the two five gate delay loops, *A-B-C-D-Y* and *B-C-D-E-X*. We made the forward delay longer than the reverse delay, because the propagation of a FULL state goes together with the propagation of data and thus affects both a Link state and the data stored in the latches of the Link, while the propagation of an EMPTY state affects the Link state only.

7.3.2.2 Critical path: latches to data kiting to double-barrel Links

One might expect that the critical paths in the Weaver are in the crossbar switch where high speed, wide data, and data-driven flow control come together. In this

section we explain why the combination of five-gate ring oscillators and 72 latches per Link lead to *data kiting*, why *data kiting* combined with data-driven flow control necessitates *advance decoding* of steering bits, and why *advance decoding* motivates the use of *double-barrel Links* in the crossbar switch.

We designed the circuits in the Weaver using the Theory of Logical Effort [3] discussed in Section 7.1.4. In particular, logical effort helps us determine the amplification required to drive the larger circuit loads. For each gate that drives a large load, we want (1) subsequent gates to sense changes in the gate's output signal within a gate delay, and (2) the gate output signal to change from rail to rail over its entire length within five gate delays. The larger loads in Figure 7.19 are (1) the many latches in Link *out*, which present a large load to gate *G*, and (2) the data and state wires between the two ends of each Link, whose lengths are likely to exceed those of other wires in Figure 7.19 and which present a large load to each latch, *X* or *Y*. Below, we outline how the FIFO circuit in Figure 7.19 supports these larger loads.

- **Provide amplification for *G*:** Each Link has 72 latches to store 72 bits of data, one bit per latch. The series of inverters *D*, *F*, and *G* in Figure 7.19 amplify the output signal of the AND function, *FULL(in)* **and** *EMPTY(out)* **and** *go*, to enable *G* to drive the large load presented by the 72 latches in Link *out*.
- **Limit the load on *G*:** To make the load on gate *G* as small as possible, the latch design—shown in Figure 7.20(a)—uses two complementary and tiny pass gates that are controlled by input *c*, which is the output of *G* in the context of Figure 7.19. One pass gate is transparent when *c* is high and is used to capture new data from *Din* to *Dout*. The other pass gate is transparent when *c* is low and is used to store the captured data and maintain the value on *Dout*. Each pass gate has an N-type and a P-type transistor and a tiny inverter—omitted from Figure 7.20(a)—to invert *c* locally so it can drive both types of transistors.
- **When needed, provide higher-gain amplification for *G*:** The current design with series inverters *D*, *F*, *G* provides enough amplification for *G* to drive the tiny pass gates in each of the 72 latches. Had the FIFO circuit used substantially more latches, say twice as many, it might have "clocked" these with a higher-gain pulse amplifier based on the *post-charge logic* design by Proebsting [2]—for instance by replacing series inverters *F* and *G* with a version of Figure 7.6(b), Section 7.1.
- **Provide amplification for *X* and *Y*:** Just like series inverters *D*, *F*, *G* build up sufficient amplification for *G* to drive 72 latches, so do inverting gates *D*, *E*, *X* and *D*, *Y* provide enough amplification for *X* and *Y* to drive a FULL or EMPTY state change on Link *out* and Link *in*. We gave *X* and *Y* each a drive strength of 40, as indicated in Figure 7.19. We were able to do so partly because *X* and *Y* use small keepers.

  X uses a P-type transistor to drive Link *out* from EMPTY to FULL. Because its keeper is small, driving *X* boils down to driving its P-type transistor, which is two thirds the effort of driving an inverter. The reduced effort makes it

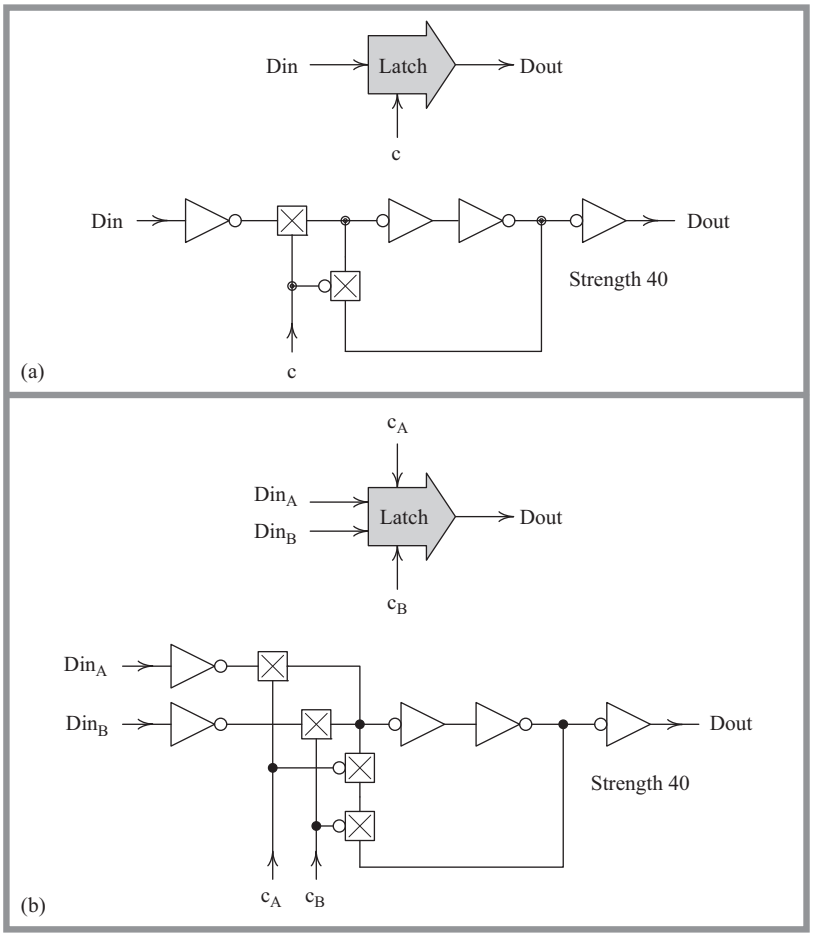

Figure 7.20 *Latch circuits to store and drive one bit of data in a Link. (a) A single-input single-output latch icon (top) and circuit (bottom) uses two complementary and tiny pass gates—the crossed squares—that are controlled by input c. Each pass gate has an N-type and a P-type transistor and a tiny inverter—omitted here—to invert c locally so it can drive both types of transistors. The picture indicates if the pass gate is transparent when c is high or c-inverted is high. With c high, the latch captures data from Din to Dout. With c low, the latch stores the captured data and maintains the value on Dout. The multiplexed latch version in (b) uses two control inputs, c_A and c_B, of which at most one is high at any time. With c_A high, data go from Din_A to Dout. With c_B high, data go from Din_B to Dout. When both c_A and c_B are low, the latch stores and maintains the captured data*

possible to give *X* a drive strength of 40 using three steps of amplification—*D*, *E*, *X*—the first of which, *D*, is shared to amplify *G* and *Y* as well. *Y* uses an N-type transistor to drive Link *in* from FULL to EMPTY. Because the keeper in *Y* is small, driving *Y* boils down to driving its N-type transistor, which takes only one third the effort of driving an inverter. The reduced effort makes it possible to give *Y* a drive strength of 40 with just two steps of amplification—*D* and *Y*.

- **Provide amplification for each latch:** Like *X* and *Y*, each latch has drive strength 40. The design in Figure 7.20(a) achieves this drive strength by amplifying the data signals captured by the pass gates, using a series of three inverting gates directly following the pass gates.

Note that the amplification for each latch, to give it a drive strength of 40, is done inside the latch design. Note too that this amplification comes in addition to the amplification for *G*, to give *G* sufficient strength to drive each of the 72 local latch control signals. Had the data been narrower, say 1 or a few bits, a latch design with two instead of four gate inversions and "clocked" by *fill(out)* instead of the output of gate *G* might have sufficed. In that case, the new data values would have been available at the other end of Link *out* at same time as the FULL state indicator. With 72 bits, however, the new data values will be available three to four gate delays later. In other words, to drive 72 bit wide data at high speed the Weaver must *kite the data*.

Below follows a step by step explanation for why this is the case and how this leads from *data kiting* to *advance decoding* and *double-barrel Links*.

- **Data kiting:** Two gate delays after the start of the *fire* pulse the 72 latches in Figure 7.19 are "clocked" by gate *G*. Likewise, two gate delays after the start of the *fire* pulse driver-and-keeper gate *X* drives the state of Link *out* from EMPTY to FULL.

  In the Weaver, we can assume that the input data for the latch circuits in Figure 7.20 arrive at the pass gates before "clock" control signal, *c*, goes high. Thus, the delay through each latch in Figure 7.20 is determined by the delay from *c* (rising) to *Dout*. With tiny pass gates, the latch delay is closer to three than to four gate delays. As a result, the new data values captured by the latches become available at the other end of Link *out* three to four gate delays after the FULL state. The data are kited—they are tardy by three to four gate delays.

Ring FIFOs in the Weaver can deal with three to four gate delays data kiting. In Figure 7.19, *D(in)* data that arrive three to four gate delays after *FULL(in)* arrives at Joint *fifo* will be at the latches in Link *out* two to three gate delays before *FULL(in)* will have propagated through the six gates *A*, *B*, *C*, *D*, *F*, *G* to "clock" the latches. By the time the "clock" rises, the data will be ready at the pass gates in the latches, having arrived at least one to two gate delays earlier.

If designed with care, *data kiting* can work equally well for Weaver parts with data-driven flow control. Take for instance the *Splitter* circuit in Figure 7.21. Its incoming data contain a steering bit, *D(in[s])*, that must be available in both true and complement forms before the circuit can decide which outgoing Link state to

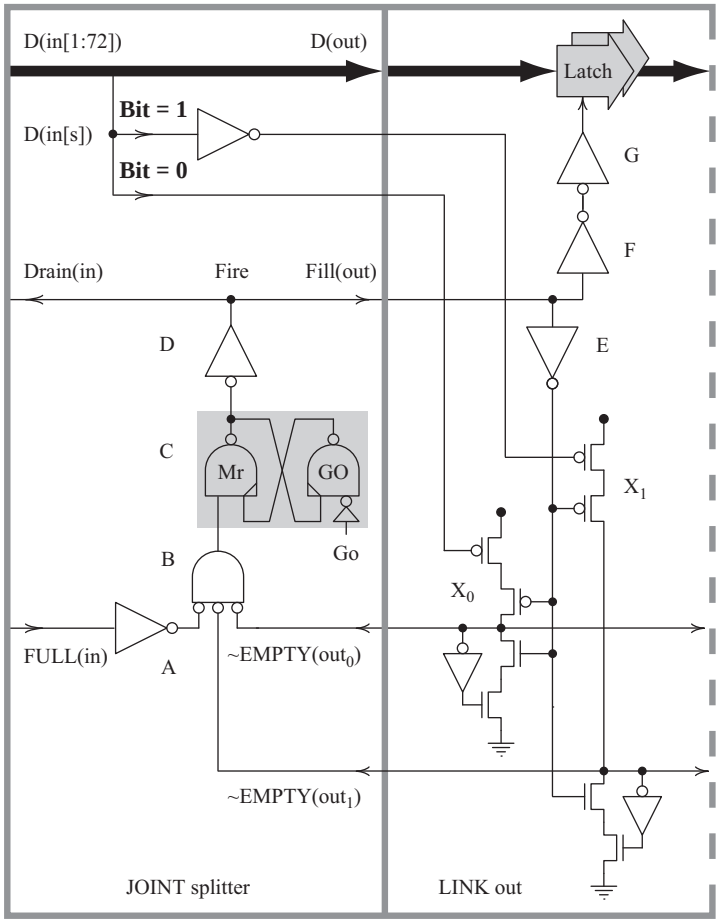

Figure 7.21 Splitter Joint and near-end output Link connection in 6-4 GasP. Gray-colored gates are icons—a circuit for MrGO was given in Figure 7.10, latch circuits can be found in Figure 7.20. Splitters start the advance decoding of steering bits for the crossbar switch by converting a steering bit from bundled data to double barrel form

change—out_0 or out_1. True and complement can be generated within a gate delay. The circuit decides which Link state to change as late as possible by using the true and complement steering signals as an extra selection input to driver-and-keeper gates X_0 and X_1. Thus, *D(in[1:72])* data that arrive three to four gate delays after *FULL(in)* will be at X_0 and X_1 one to two gate delays before *FULL(in)* will have propagated through the five gates *A*, *B*, *C*, *D*, *E* to drive the selected X_0 or X_1.

All circuits in the Weaver have a margin of at least one to two gate delays from the arrival of their data signals to the arrival of their control signals, be it for latching the data or for data-driven flow control. We can get extra delay margins from the wires in the Weaver's layout by allowing different wires to have different widths and different spacings. In particular, data wires in the Weaver use twice the width and twice the spacing used for control wires.

To understand how width and spacing affect the "speed" of a wire, let us consider the real shape of an integrated circuit wire. Wires are relatively thick layers of metal, sandwiched between layers of insulation. Most wires are more tall than they are wide, much as a fence is more tall than it is wide. Wires stand up the full thickness of each layer, like the walls of a room inside a multistory building. Because wires are more tall than wide, most of the capacitance between wires is to adjacent wires on the same layer rather than to wires in layers above or below.

Making a wire twice as wide halves its electrical resistance. Less resistance allows information to get through the wire faster. Doubling the wire width doubles the wire's relatively small capacitance to wires in layers above or below, but preserves the much larger capacitance to adjacent wires in its own layer. Doubling the space between a wire and adjacent wires on the same layer almost halves the capacitive load of that wire. Less capacitance speeds up the transistors that drive the wire.

By doubling both width and spacing, the Weaver's data wires gain an almost four-fold advantage in speed over the Weaver's control wires.

- **From data kiting to advance decoding of steering bits:** The *Splitter* decodes the steering bits one stage in advance, just before the data enter their first *Double Crosser* in the crossbar switch—see Figure 7.16. In the *Double Crosser*, the FULL Link state that accompanies the data is hardwired to the *Crosser* that steers the data in the intended direction—see Figure 7.17.

  The *Crosser* arbitrates between data items that go in the same direction by arbitrating between their FULL Link states—without looking at the data. This is possible because the *Splitter* decoded the direction in advance by making the associated Link state FULL. As a result, each *Crosser* can support arbitration as well as decode its *Double Crosser* specific steering bit one stage in advance.

  The combination of *five*-gate ring oscillators, data kiting, and arbitration would have been impossible without decoding the steering bits in advance.
- **From advance decoding to double-barrel Links:** Note that the *Splitter* circuit in Figure 7.21 as well as the *Crosser* circuit spread over Figure 7.23(a) and (b) take the one-hot Link states that decode the circuit's steering bit and pair these into a single Link with two Link states. Because at most one of these two

Link states can be FULL at any given time, just one set of 72 latches will suffice to store the data sent along each FULL Link state. We call the resulting Link a *double-barrel Link*.

The layout of the Weaver *shields* the narrow FULL or EMPTY state wires in each Link with an adjacent grounded wire on each side, to protect them from capacitance coupled noise. Ordinary Links with only one state wire use a three-wire control bundle: *ground-state-ground*. Double-barrel Links use a five-wire control bundle: *ground-state$_0$-ground-state$_1$-ground*. In each Link, 36 wider data wires with their wider spacing flank each side of the control bundle. Double-barrel Links are only slightly wider than ordinary Links.

One might view a double-barrel Link as a peephole optimization of the more typical implementation with two separate ordinary Links that can share data because they take the data from the same crossbar source to the same crossbar destination and they operate in mutual exclusion. We prefer to view a double-barrel Link as just a Link with typed interfaces, where the type information conveys the one-hot encoding properties of the two Link states. Adding type information to a Link and Joint interface makes it possible to fine-tune the interface and the tasks on each side of it as well as to graduate the delay sensitivity of the information exchange.

The stages in the crossbar switch communicate through double-barrel Links. Section 7.3.2.3 describes three representative circuits related to the crossbar.

7.3.2.3 Crossbar circuits: Splitter, Double-barrel Ricochet, Crosser

Double-barrel Links appear only inside the 8 × 8 crossbar switch and at its inputs and outputs. Data enter the crossbar from the South and leave toward the East. Just South of the crossbar, a dark gray area in Figure 7.16 holds eight *Splitter* stages that fill the double-barrel Links for the first row of *Double Crossers*. The dark gray area at the North West (NW) boundary holds eight *Double-barrel Ricochet* stages that act as FIFO stages for double-barrel Links. They repeat and fold double-barrel Links, directing data from the South heading North to make an Eastbound turn instead. Each *Double Crosser* stage in the light gray area in Figure 7.16 has two double-barrel input Links, coming from the South and the West, and two double-barrel output Links, going North and East, respectively. A *Double Crosser* also has two *Crossers*—one for Northbound data and the other for Eastbound data. As illustrated in Figure 7.17(c) and (d), the two *Crossers* share data from the double-barrel input Links. Each *Crosser* arbitrates between data items that go into the direction it controls. Just East of the crossbar, another dark gray area in Figure 7.16 holds eight *Lumper* stages that drain the double-barrel Links for the last column of *Double Crossers* and pass their data to the ordinary Links and FIFO rings, for recirculation.

Figures 7.21–7.23 show the 6-4 GasP circuit implementations that the Weaver uses for the *Splitter*, *Double-barrel Ricochet*, and *Crosser*. To emphasize how similar these implementations are to each other and to a simple 6-4 GasP FIFO with ordinary Links, all three Figures borrow the alphabetic gate identifier scheme of Figure 7.19. Below follow brief explanations of the circuit implementations in

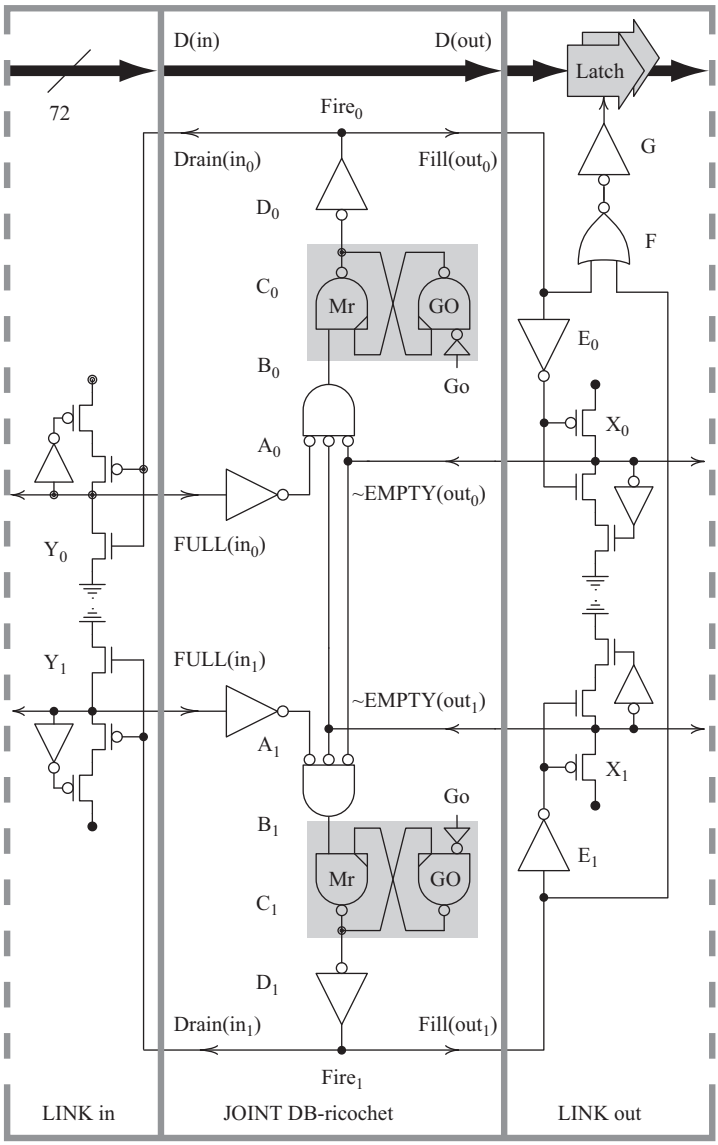

Figure 7.22 Double-barrel Ricochet Joint and near-end Link connections. An advanced FIFO circuit for double-barrel Links

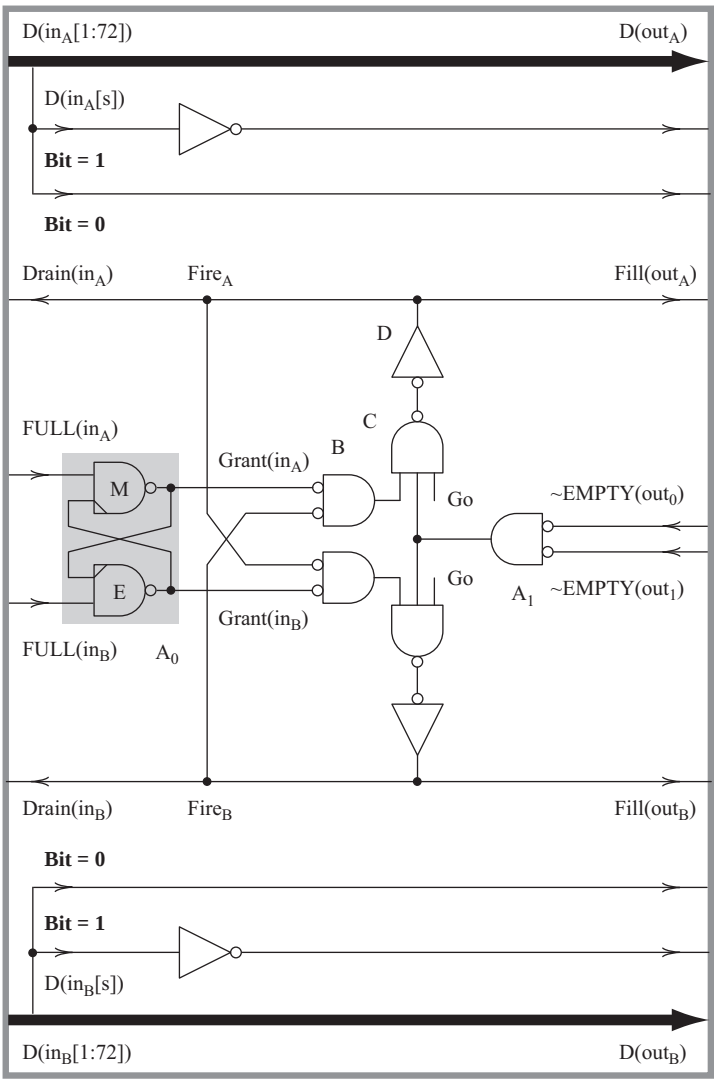

Figure 7.23 (a) Joint crosser and its input and output interfaces. Gray-colored gates are icons—the mutual exclusion (ME) circuit resembles MrGO in Figure 7.10(c) and (d) but exports both its outputs

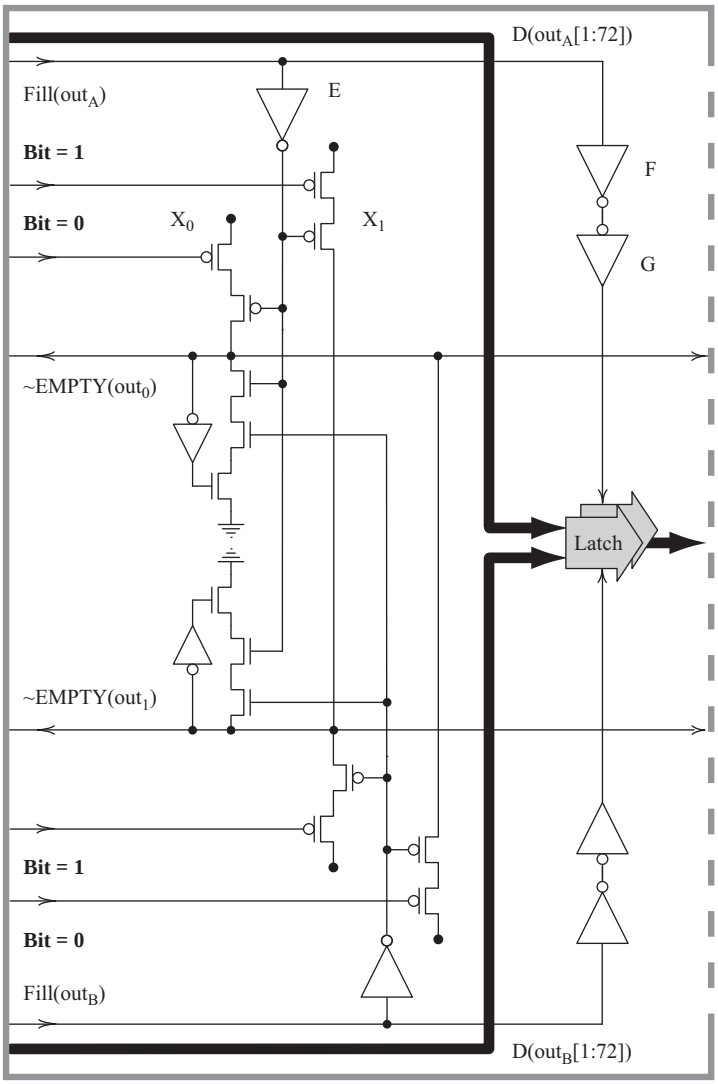

Figure 7.23 (b) Double-barrel output Link connection to Joint crosser. Gray-colored gates are icons—for latch circuits, see Figure 7.20

Figures 7.21–7.23. We focus primarily on new circuit aspects that each subsequent Figure brings in.

- **Splitter:** Figure 7.21 gives a 6-4 GasP implementation of the *Splitter* showing its Joint *splitter* and omitting its input Link *in* and half of its output Link *out*. The Joint receives 72 bundled data input bits labeled *D(in[1:72])*. One of these 72 bits, *Din[s]*, acts as a *Splitter* specific steering bit to select which double-barrel output state to fill—out_0 or out_1. The Joint copies all 72 input bits *D(in[1:72])* to *D(out)*, including steering bit *D(in[s])*. Thus, *Din[s]* remains available in bundled data form to repeat its steering task on a subsequent pass through the *Splitter*. As explained in Section 7.3.2.2, the kited steering bits are used as late as possible, in X_0 and X_1, to compensate for their kiting. Similar to Figure 7.19, inverters *D*, *E*, *F*, *G* amplify the Joint's AND function so it can drive the large loads presented by the 72 latches in Link *out* and by the driver-and-keeper gates X_0, X_1, and Y in Links *out* and *in*. Joint *splitter* has the following AND function: *FULL(in)* **and** *EMPTY*(out_0) **and** *EMPTY*(out_1) **and** *go*.
- **Double-barrel Ricochet:** Figure 7.22 gives a 6-4 GasP implementation of the *Double-barrel Ricochet*, showing its Joint *DB-ricochet* with half of its input and output Links, *in* and *out*. A *Double-barrel Ricochet* is an advanced FIFO circuit for double-barrel Links. It has two mutually exclusive AND functions. When it fires its first one, *FULL*(in_0) **and** *EMPTY*(out_0) **and** *EMPTY*(out_1) **and** *go*, inverters D_0, E_0, *F*, *G* provide the amplification to drive the latches in Link *out* to copy the data from *D(in)* to *D(out)* and to drive X_0 and Y_0 to fill out_0 and drain in_0. When it fires its second one, *FULL*(in_1) **and** *EMPTY*(out_0) **and** *EMPTY*(out_1) **and** *go*, the inverters D_1, E_1, *F*, *G* provide the amplification to drive the latches and copy the data and to drive X_1 and Y_1 to fill out_1 and drain in_1.

  Note that Joint *DB-ricochet* has two MrGO circuits, one per AND function, but that we tied their *go* input signals together, resulting in one *go* signal that enables or disables both AND functions.
- **Crosser:** Figure 7.23(a) and (b) gives a 6-4 GasP implementation of the *Crosser*, showing its Joint, *crosser*, and double-barrel output Link, *out*. The input interface of the Joint suggests two ordinary incoming Links, in_A and in_B, with 72 bits of data including a steering bit. Links in_A and in_B subset the signals of the double-barrel Links that enter the *Double Crosser* and relate to Link *out*—see Figure 7.17. The *Crossers* handle contention in the crossbar switch. At the heart of Joint *crosser* in Figure 7.23(a) is an *arbiter* or *mutual exclusion (ME)* circuit, gate A_0, patterned after the 1980 design by Charles Seitz [17], and sized to minimize its delay for the common uncontested case [4]. This mutual exclusion circuit grants on a first-come-first-served basis, and waits for metastability to end before it lowers the selected grant signal. Besides handling contention, each *Crosser* also decodes its *Double Crosser* specific steering bit one stage in advance of need, by producing double-barrel outputs—just like the *Splitter* in Figure 7.21. The new aspects in the *Crosser* are:

(a) Joint *crosser* in Figure 7.23(a) has two mutually exclusive AND functions: one for granting $FULL(in_A)$ and the other for granting $FULL(in_B)$. When it fires the first, $\sim grant(in_A)$ **and** $\sim fire_B$ **and** $EMPTY(out_0)$ **and** $EMPTY(out_1)$ **and** *go*, inverters *D*, *E*, *F*, *G* spread over Figure 7.23(a) and (b) provide the amplification to drive the latches in Link *out* to copy $D(in_A)$, and drain in_A, and fill either out_0 or out_1 depending on whether steering bit $D(in_A[s])$ is zero or one. The other AND function results in a similar action between in_B and *out*. Note that $\sim fire_B$ is an input to the AND function that generates $fire_A$. Likewise, $\sim fire_A$ is an input to the AND function that generates $fire_B$. Such cross-coupling prevents one AND function from overtaking the other in case of back-to-back grants [13]. Cross-coupling ensures that the signals that drive the data and state changes over the Links have adequate pulse widths—five gate delays wide.

(b) Note that the AND functions in Joint *crosser* both have a *go* input signal, but that both signals come unarbitrated, that is, without a corresponding MrGO circuit. We found the task of adding two MrGO arbiters to an already arbitrated circuit with a tight layout and with tight five gate delay loops and high amplification needs simply too daunting. One *go* input signal serves both AND functions.

7.3.3 Test logistics

The Weaver's rings, including the parts that go through the crossbar switch, can each transfer up to about 6 Giga data items per second (GDI/s). The supporting throughput measurements follow in Figure 7.30, Section 7.3.5. With 72 bit wide data items, this amounts to 3.5 Tera bits per second. Yet, we use a low-speed test interface consisting of only five wires to test the functionality of the Weaver and to debug and characterize its high-speed operations. Moreover, we use this low-speed "test interface" to initialize and start the Weaver. The photo in Figure 7.24 shows a Weaver chip in its ceramic package, mounted on its test board.

In addition to the five low-speed test signals, the Weaver has two dedicated medium-speed outputs that deliver one-millionth reduced ring frequency outputs. These two medium-speed outputs follow the switching frequency of bit 19 in two of the ten 54 bit long ring counters in the Weaver—see Section 7.3.1.3 and Figure 7.18. The two black coaxial cables in Figure 7.24 carry the reduced ring frequency outputs to an oscilloscope for real-time observation.

The ring counters are 54 bits long to accommodate long test experiments. When counting 6 Giga items per second, a 54 bit counter will overflow about every 30 days.

The counters can be reset to zero at the beginning of a test experiment and read out at the end. They are read out over the white flat ribbon cable visible midway the right edge of Figure 7.24. The flat ribbon cable carries low-speed signals between the chip and a computer. The computer contains the test program with instructions for controlling the low-speed test stimuli and observing the low-speed test

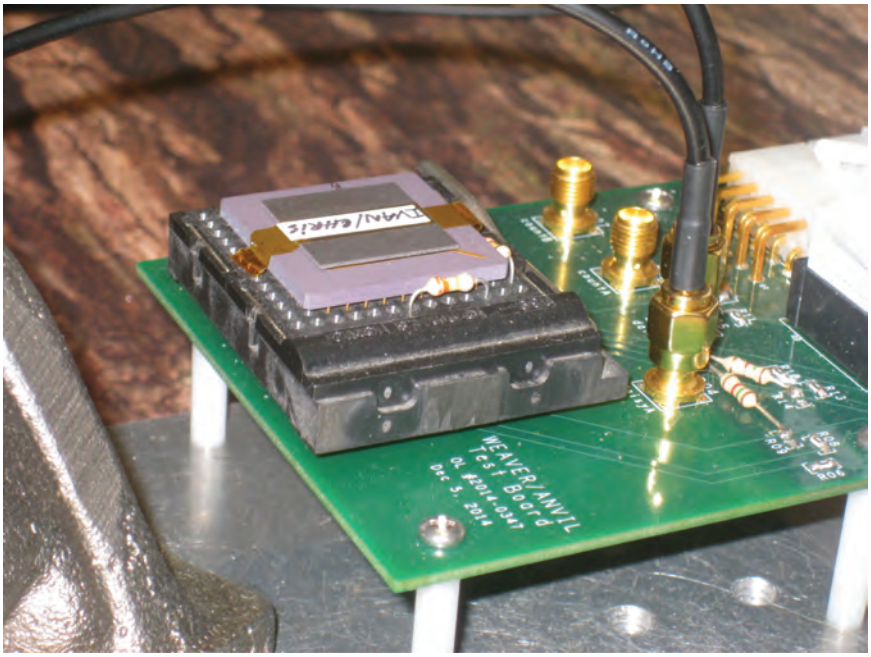

Figure 7.24 *Photo of a packaged Weaver chip on its test board. The chip contains two experiments, one of which is the Weaver. The other, called Anvil and designed by Chris Cowan, is a case study in radiation hardening—not further discussed here. The two black coaxial cables connected near the middle of the board carry two one-millionth reduced ring frequency outputs to an oscilloscope for real-time observation—see Section 7.3.1.3. L-shape connectors at the top-right corner of the photo bring in power and ground. A white flat ribbon cable visible midway the right edge of the photo carries five low-speed test signals to and from the chip and a computer. The computer contains the test program with instructions for controlling the low-speed test stimuli and observing the low-speed test responses. Board and final chip layout are by the late Jon Lexau of Sun Labs*

responses. The chip contains a low-speed JTAG test interface with five test pins, an on-chip test access port, and on-chip scan chains. This low-overhead low-speed test interface is an industry standard for testing manufactured chip designs and printed circuit boards. It was codified by the Joint Test Action Group (JTAG) and the Institute of Electrical and Electronics Engineers (IEEE) in IEEE Standard 1149.1-1990, entitled Standard Test Access Port and Boundary-Scan Architecture [15].

The JTAG test interface in the Weaver runs at 500 kHz. The ten ring counters hold 54 bits each. We use a scan chain to read out all 540 counter bits at once. We then shift the scan bits one by one over the JTAG test interface, which takes on the

order of a millisecond. We use a similar approach for reading and writing approximately 500 *go* control signals, one per Joint, approximately 500 Link states, and 720 data bits, 72 for each of the ten ring reloaders.

The JTAG test interface is synchronous and clocked. It has one output signal, *test data out*, and four input signals, *test clock*, *test data in*, *test mode select*, and an optional *test reset* signal. These signals are used to set up and select test operations, to read and write Weaver states, and to enable and disable Weaver actions. Details about setting up test operations can be found in IEEE Standard 1149.1-1990 [15]. Here, we show the Weaver specific parts of the test interface [7]—the scan chains and the transfer circuits to and from the scan chains and the Links and Joints in the Weaver—that read and write (Link) states and enable and disable (Joint) actions.

7.3.3.1 Scan chains and connections to Weaver Links and Joints

The scan chains in the Weaver consist of shift registers connected in series. Each shift register has two small latches that are also connected in series, as illustrated in Figure 7.27(c). The circuit designs for the two small latches follow in Figure 7.25. With two latches, the shift register can store one bit safely. This bit can be shifted in or out serially. To shift bits in or out, the two latches in the shift register are clocked alternately, using two scan clocks, c_1 and c_2. Instead of shifting a bit in through the scan chain, the shift register can read a bit from the Weaver and store it into its second latch, using a special scan signal called *read*. In addition to shifting a bit out through the scan chain, the shift register can write the bit that it stores in its second latch into the Weaver, using a special scan signal called *write*.

Each shift register comes with a bundle of eight scan signals, including shift register specific scan input and output signals, *sin* and *sout*. The other signals in the scan bundle travel the entire length of the scan chain, with regular amplification. These include the two scan clocks, c_1 and c_2, and the scan *read* and *write* signals for interaction with the Weaver. Other scan signals that travel the entire length of the scan chain—c_1*Return*, c_2*Return*, *sReturn*—are the far-out scan clock signals and the scan output signal of the last scan shift register. These travel in reverse direction through the scan chain, back to the first scan shift register and its JTAG test interface.

The far-out scan clock signals that return to the JTAG test interface are important for generating nonoverlapping clocks for shifting data in and out of the scan chain. The clock generator in Figure 7.26 combines the low-frequency JTAG *test clock* with c_1*Return* and c_2*Return* to generate low-frequency scan clocks c_1 and c_2 that are never high at the same time. If the two clocks are never high at the same time, the two small latches clocked by them in Figure 7.27(c) are never transparent at the same time, and neither are any other subsequent latches in the scan chain. The nonoverlapping clocks make the scan chain shift bits properly.

After a bit has arrived in the second latch of the shift register, we can write it into the Weaver signal associated with this shift register. Or we can read the value of the Weaver signal into the second latch and shift it out for inspection. Some writes, specifically those associated with *go* signals, enable or disable circuit actions and can even start or stop them. Other writes merely change circuit states.

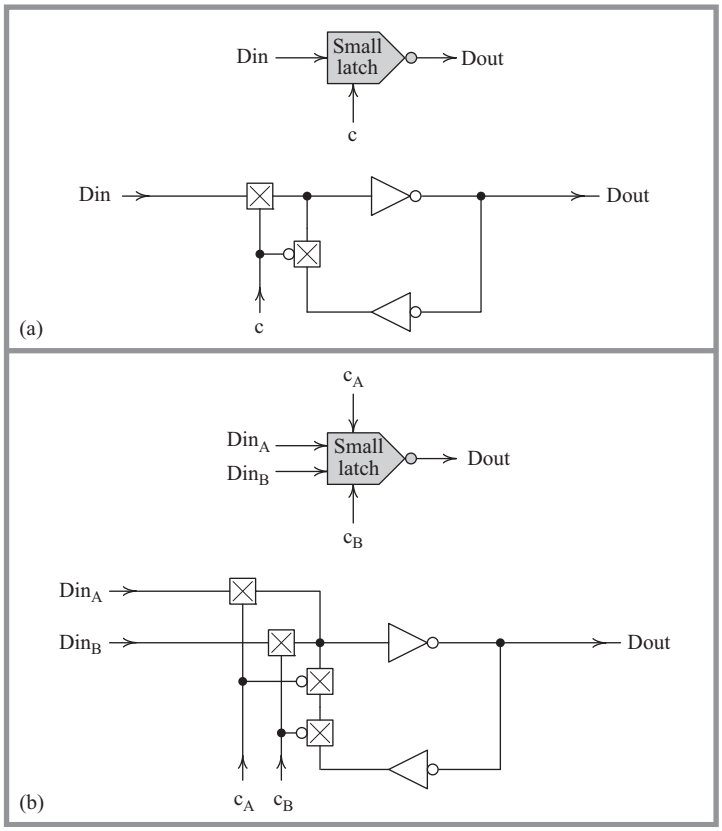

Figure 7.25 Scan latch circuits. Because of their low, 500 kHz, clock frequencies the scan latches can be half the size of the data latches in Figure 7.20

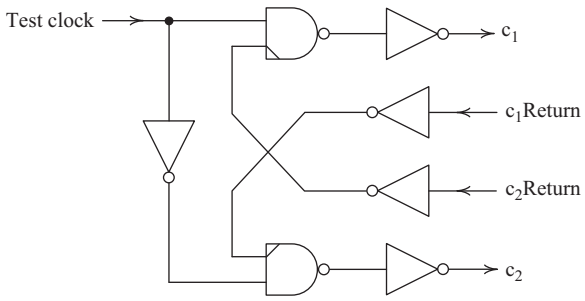

Figure 7.26 Scan clock generation. We use the low-frequency JTAG test clock to generate two low-frequency scan clocks that are never high at the same time. Each scan clock, c_1 or c_2, goes high only after the longest branch of the other scan clock, c_2Return or c_1Return, has gone low

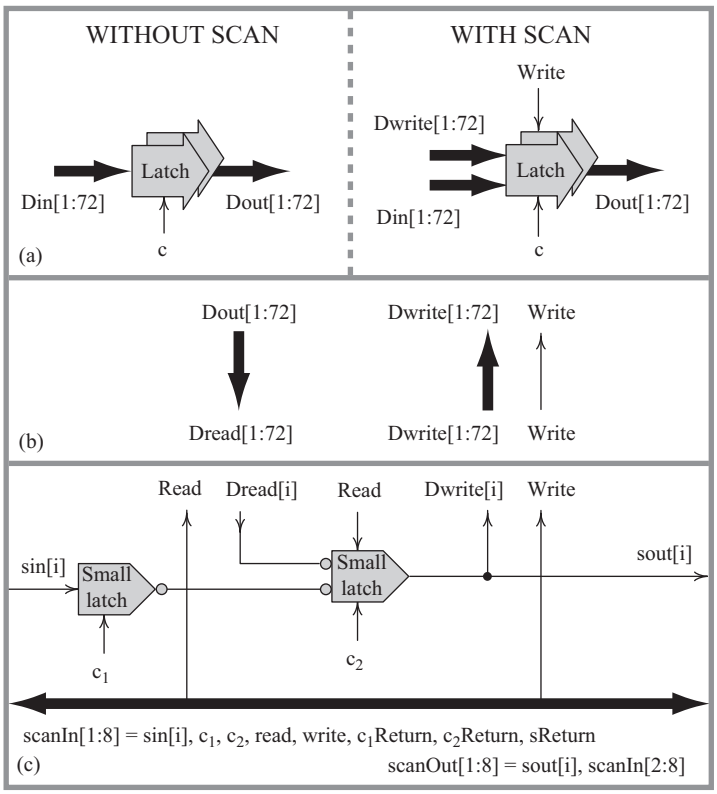

Figure 7.27 Scan connections for reading and writing Weaver data. Weaver data can be read for inspection and written for initialization or test at a reloader stage located in the SE corner of the Weaver floorplan—see Figure 7.16. A reloader stage is just a FIFO circuit, as in Figure 7.19, but one with scan access to the data latches in its output Link. Each data latch is associated with a specific scan shift register (c) which can read the bit stored in the data latch, Dout[i] alias Dread[i], or write its own bit, Dwrite[i], into the data latch (b). To allow the shift register to overwrite latch content, each data latch is replaced by its multiplexed version (a)—see Figure 7.20. Thanks to their small latches, the 72 serially connected scan shift registers and their bundles of scan signals occupy a footprint similar to the FIFO circuit. Reloader FIFO and scan fit in one Weaver layout module

The Weaver uses different circuits to transfer bits to and from the scan chains and its data latches, FULL or EMPTY Link states, and *go* signals. Figures 7.27–7.29 illustrate the differences. The three Figures show similar shift registers (c) but different transfer circuits (b) and different Link and Joint circuit modifications after scan insertion (a). In particular, the Weaver writes and stores the data bits that it receives from the scan chain in its own data latches—see Figure 7.27. Likewise, the Weaver reuses its own driver-and-keeper gates to write and store the FULL (1) or EMPTY (0) bit that it receives from the scan chain. It turns the keepers off while it writes the Link state, and stops driving the Link state when the *write* signal is low—see Figure 7.28. In contrast, the Weaver writes and stores each *go* signal that it receives from the scan chain in a separate small latch—see Figure 7.29. Note that all three Figures enable their shift register to read back what it wrote. To reduce wire capacitance and switching power, we gate the read connections from the high frequency Link state signals to the shift registers when *read* is low.

The Weaver has several scan chains. One scan chain follows the NE edge of the Weaver, reads the ring counters, initializes them to zero, and sets their multiplexers for monitoring frequency outputs in real time—see Figure 7.18. A second and similar scan chain, based on Figure 7.27, follows the SE edge and reads and writes the data latches in each reloader stage. A third one, based on Figure 7.28, reads and writes the Link states of each Link in the Weaver. A fourth, based on Figure 7.29, reads and writes the *go* signals of each Joint. The scan chains for FULL or EMPTY and *go* signals visit each module in the Weaver, and do so in boustrophedonic order—turning like oxen in ploughing a field. Their first shift registers start at the JTAG test interface in the corner where NE and SE edges meet. Their last shift registers end at the opposite corner where NW and SW edges meet. Their scan shift registers and the corresponding transfer circuits to each Link and Joint are combined with the layout modules of the Links and Joints. The Weaver's JTAG test interface operates the scan chains in mutual exclusion. Note that the shift registers in Figures 7.27–7.29 *read* Weaver bits in parallel, *write* Weaver bits in parallel, but *shift* bits serially in and out the scan chain. Note too that the scan chain can shift while the circuit operates, without mutual interference.

7.3.4 How low-speed scan chains test high-speed performance

Asynchronous or self-timed circuits operate as fast as they can—when they can. Externally lock-stepping their operations to, for instance, the JTAG *test clock* would take the "self" out of their timing and run them synchronously and no longer at speed. Once this realization sinks in, it becomes obvious that we need merely identify the borders to where the circuit can run, and allow it to run "flat out" inside these borders. Circuit actions stop at the border. The *go* signals in our circuits give us the necessary control to enable actions within borders and disable actions at borders.

We can test the high-speed Weaver operations at speed because we enabled the JTAG test interface to control actions and states separately. The Weaver's test interface recognizes and controls the individual *go* signals in each Joint's action

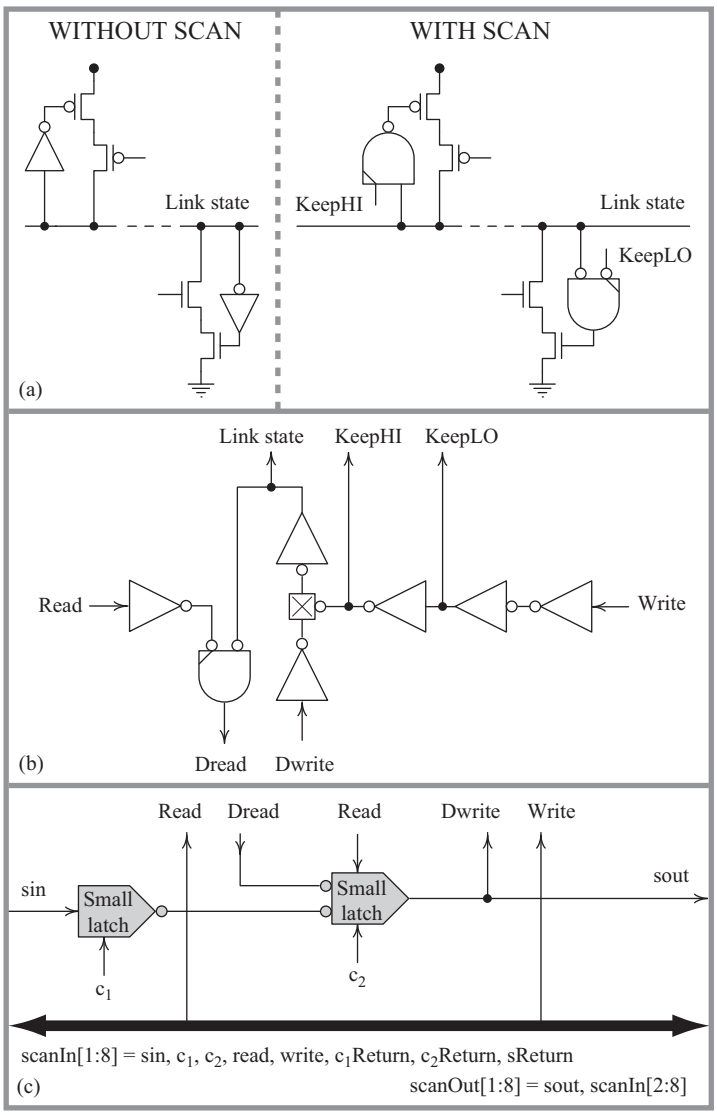

Figure 7.28 Scan connections for reading and writing Link states

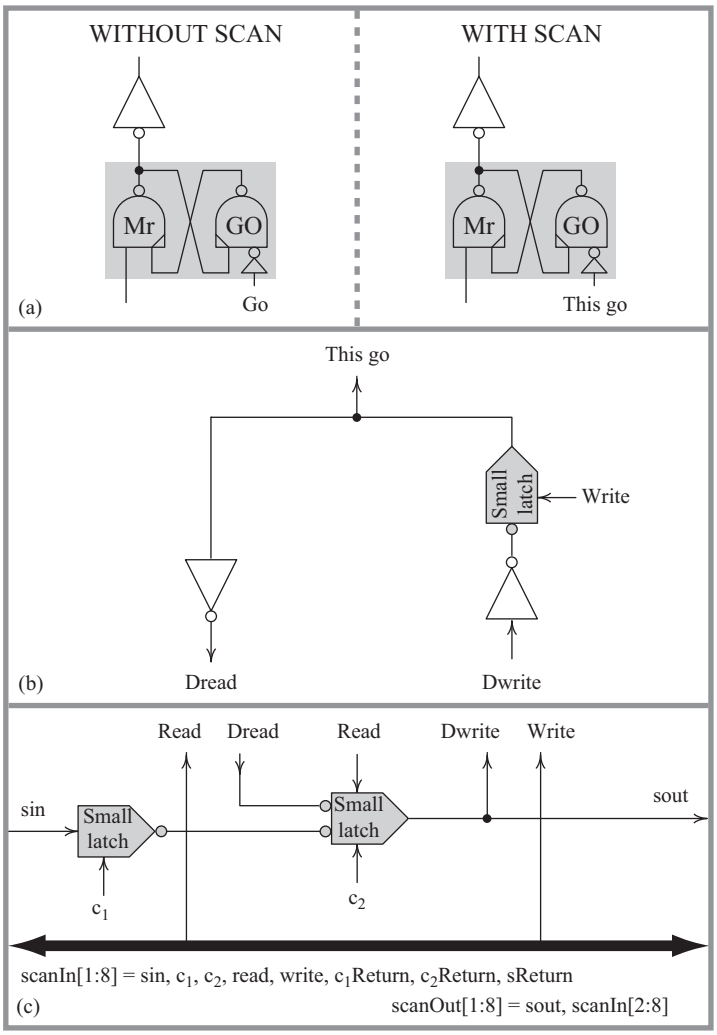

Figure 7.29 Scan connections for reading and writing go signals. Each go signal receives a separate small latch for scan access. The latch isolates scan shift operations from circuit operations, which can be done in parallel without affecting each other's data or control flow

and it recognizes and controls the individual FULL or EMPTY and data signals that are stored in each Link. In Section 7.2, we explained how distinguishing actions from states and controlling them separately accommodate initialization, structural testing, and at-speed testing of parts or the entire design.

In the Weaver, a burst of data items will run *at speed* from one end to the other end of an empty ring segment, with the two ends marked by disabled *go* signals. Any preparation work for running this burst—by (1) first disabling the *go* signals, so we can initialize the Links in the takeoff, under test, and landing parts of the ring, as in Figure 7.14, by (2) then enabling the *go* signals to enable all parts to run freely except for the two ends and the "gate keeper" to the part under test, and by (3) finally enabling the "gate keeper" to release the burst and let it run freely through the ring segment—any of that can be done at low speed, using the JTAG test interface. The at-speed performance follows from letting the circuit run freely from end to end. Similar low-speed preparations let us run data items through an endless ring and stop their circulation by disabling the "gate keeper" in real time, as described at the end of Section 7.2.2. We do this to measure performance—throughput, power, and energy. For throughput, we scan out the ring counters and relate their values to the run time. For power, we use a current probe to measure the average current that the Weaver draws while running, and relate its value to the supply voltage used while running. The energy consumption can be calculated from throughput and power. Section 7.3.5 presents our throughput, power, and energy measurements from the Weaver chip.

7.3.5 Performance measurements

Figures 7.30–7.33 show four collections of canopy graphs with throughput and power measurements from the Weaver chip, measured for various traffic and supply voltage levels. Each graph plots the measured information as a function of ring *occupancy*—the number of FULL Links or valid data items [18,19]. Sections 7.3.5.1–7.3.5.4 analyze the canopy graphs, showing throughputs of about 6 Giga data items per second, and energy dissipation around 3 picojoules to forward one data item one stage.

7.3.5.1 Throughput versus occupancy at nominal power supply

The canopy graphs in Figure 7.30 plot the throughput for four of the ten FIFO rings at nominal power supply voltage. Rings 0 and 9 bypass the crossbar switch and thus omit switching elements. Ring 1 has the highest maximum throughput and Ring 8 has the lowest maximum throughput of all eight rings that go through the crossbar switch. The throughput reflects the count reached in each ring counter stage in the NE corner of the Weaver floorplan—see Figure 7.16. We normalized the count to average one second of run time. Each graph plots the throughput as a function of ring occupancy, that is, of the number of valid data items in the ring.

An empty ring has zero throughput just as an empty freeway carries no traffic. Likewise a completely full ring has zero throughput just as a congested freeway stalls traffic. Therefore, at its left and right ends a canopy graph shows zero throughput. The linear rise in throughput with occupancy at the left of each canopy

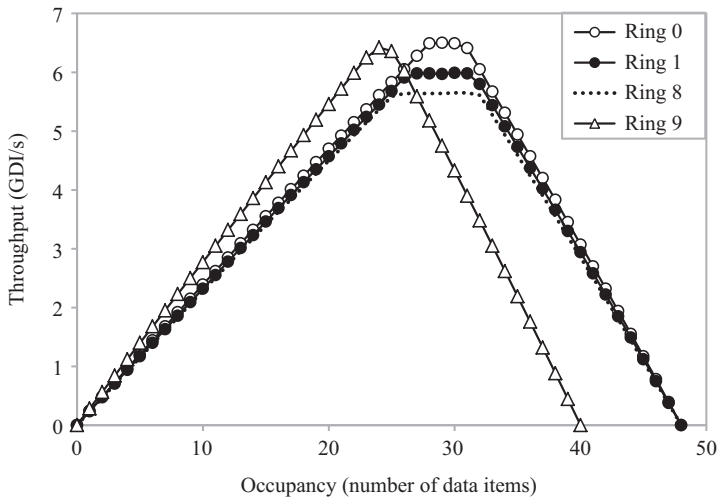

Figure 7.30 *Canopy graphs for throughput versus occupancy. Each canopy graph plots the frequency measured at nominal supply voltage as a function of ring occupancy. Ring 9 has 40 stages, all others have 48 stages. Maximum throughput is at 60% occupancy, and around 6 GDI/s. Weaver's layout accounts completely for the differences in maximum throughput shown by the graphs*

graph is easy to understand. One data item circulates with a period set by the forward latency around the ring, and—just like a few racecars on a circular track—so do any small number of data items. Because data items cannot overtake each other, throughput increases with the number of circulating data items—as long as congestion is avoided. The right side of the canopy graph shows the impact of congestion. As congestion decreases, more spaces become available for forwarding data items and there is a corresponding linear increase in throughput.

Somewhere between a completely full and a completely empty ring there is an occupancy with maximum throughput. The canopy graph for Ring 9 shows its maximum at 6.4 Giga data items per second (GDI/s) at 60% occupancy, that is, with 24 valid data items in its 40 stages. The 6-4 GasP circuits in the Weaver transport space faster than data: the forward latency of each 6-4 GasP circuit in the Weaver is about 100 picoseconds, and the reverse latency is only about 66 picoseconds. The choice to transport space faster than data is inspired by the relative ease of transporting space. It is easier to declare a Link EMPTY, when transporting space, than it is to declare a Link FULL *and* drive the latches *and* capture an arriving data item, when transporting data. At 60% occupancy the net velocity of spaces and data items match, resulting in maximum throughput.

The Weaver's layout accounts completely for the differences in the shapes of its canopy graphs. These differences can be explained by examining the basic layout of the *Cross Fire* sections outside the crossbar switch and by examining the

layout of the *Double Crossers* inside the crossbar switch. We start our explanation by examining the layout of the NE, SE, and SW *Cross Fire* sections in Figure 7.16.

- **Layout modules:** In the *Cross Fire* layout, we pair independent FIFO stages that cross each other at a North-South and East-West ring crossing. Each stage has a FIFO Joint and its two near-end Link connections as shown in Figure 7.19. Crossing stages are paired to form one layout module. Each layout module is approximately square. Layout modules abut. The two FIFO stages fit side by side in the module, each taking a slice that spans the full module height and half the module width. Per FIFO stage or slice, the control circuits—gates *A* to *G*, *X*, *Y* in Figure 7.19—occupy a center row flanked above and below by two groups of 36 latches each that belong to the outgoing Link. The scan circuits for *go* signals and Link state signals occupy the bottom row, below the latches, and part of the center row.
- **North-South module connections are longer:** Per FIFO, the Link state signals extend horizontally in East-West direction, almost all the way across the center row. A Link state signal that connects two FIFO stages in East-West direction must jump horizontally from one layout module's center row to an adjacent module's center row, a slice distance away. The jump requires about half a module width of extra wire. A Link state signal that connects FIFO stages in North-South direction must jump vertically from center row to center row, which requires a full module length of extra wire—twice that of an East-West connection.
- **North-South module connections are slower:** Longer wires are harder to drive than shorter wires. Moreover, the Weaver's throughput depends in part on the control speed of its Links, that is, of its state signals and their driver-and-keeper gates. For design modularity and simplicity, the Weaver uses the same drive strength of 40 for each Link driver-and-keeper gate, independent—within reason—of the wire length of the state signal it drives. Because North-South module connections have longer Link state signals than East-West module connections, using equally strong Link driver-and-keeper gates makes the North-South module connections slower.

The slowness of North-South connections compared to East-West connections is even more pronounced for the *Double Crosser* layout modules in the crossbar switch. Their footprint and organization are similar to those for the *Cross Fire* layout modules: approximately square, with two *Crossers* and their near-end Link connections positioned side by side, with control circuits occupying a center row and groups with 36 latches each above and below. Because the control complexity is higher and the number of gate connections per Link state is higher, *Double Crosser* layout modules have longer Link state signals in and between them than *Cross Fire* layout modules. *Double Crosser* module connections are slower than *Cross Fire* module connections in any direction, and slowest in the direction North-South.

All other circuits, barring the reloader stages, are organized as narrow half-width layout modules, with just one slice instead of two slices side by side. North-

South connections for these are about as slow as for *Cross Fire* layout modules. The reloader stages in the SE corner of Figure 7.16 take an entire layout module each. The FIFO ring stage takes one slice. The other slice contains the scan circuits to read and write the 72 data bits in the FIFO ring stage. North-South and East-West connections to a reloader stage are similar to those for a *Cross Fire* layout module.

We now have enough information to understand the differences between the canopy graphs in Figure 7.30. Consider first the canopy graphs for Ring 0 and Ring 9 that avoid the crossbar switch. The maximum throughputs of Ring 0 and Ring 9 are about the same because both are limited by their slower North-South Links. Next consider the canopy graphs for the switched rings, Ring 1 and Ring 8. Ring 1 passes across the bottom of the *Double Crosser* triangle of the crossbar switch, and is therefore slower than Ring 0 and Ring 9. Ring 1 avoids North-South connections between *Double Crossers* and is therefore faster than Ring 8. The canopy graph for Ring 8 is about the same as for Ring 2 to Ring 7—omitted from Figure 7.30 for this reason—because each is limited by its slower *Double Crosser* North-South Links.

7.3.5.2 Throughput for various power supply voltages

Speed scales with power supply voltage. Figure 7.31 shows canopy graphs that plot the relative throughput of Ring 4 as a function of ring occupancy at different supply voltages. Throughput numbers are scaled relative to the maximum throughput of Ring 4 at nominal supply. The throughput of Ring 4 at nominal supply is about the same as that of Ring 8 in Figure 7.30. The spacing of the flat canopy tops at different voltages indicates a nearly linear relationship between throughput and power supply. The Weaver operates flawlessly between 0.6 and 1.0 volt. In this operating region, throughput is very nearly proportional to the excess of power supply voltage over threshold voltage. Any excess beyond that required just barely to overcome the transistor threshold voltage of about half a volt can be used to charge the wires.

The medium-speed real-time outputs connected to the counters, shown in Figure 7.18, give a vivid demonstration of speed as a function of power supply voltage. Lacking a global clock, it is unnecessary to adjust a clock frequency when changing the supply voltage. Turning the knob to adjust power supply voltage makes the self-timed Weaver automatically speed up or slow down because each part proceeds as fast as the available power supply voltage permits. Turning the power supply voltage knob stretches or shrinks the square wave seen on an oscilloscope attached to Weaver's real-time counter outputs.

7.3.5.3 Power for various power supply voltages

Figure 7.32 shows canopy graphs that plot the relative power of Ring 4 as a function of ring occupancy for five different power supply levels, using a worst-case data pattern.[‖] The graphs are normalized in proportion to the maximum power at

[‖]The data pattern used in Figure 7.32 is a checkerboard pattern, with alternating bit values for each data item that flip in opposite direction for subsequent data items. For more details, see Section 7.3.5.4.

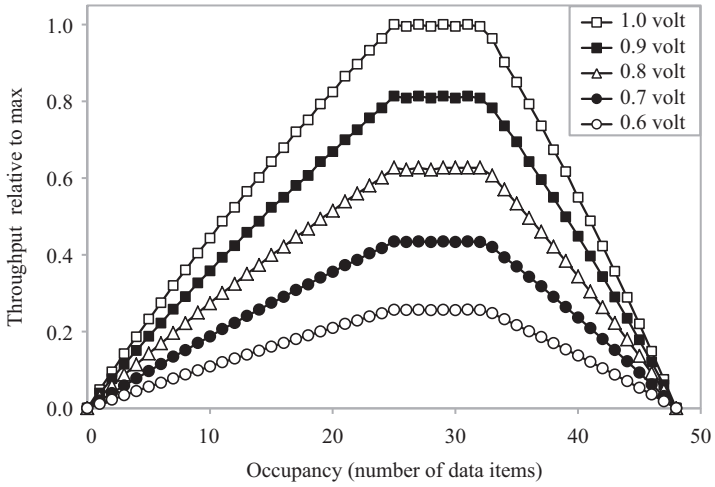

Figure 7.31 Canopy graphs showing throughput at various supply voltages. The graphs plot the relative throughput of Ring 4 as a function of ring occupancy at five different power supply levels, and indicate that throughput is very nearly proportional to (supply voltage − 0.5 volt)

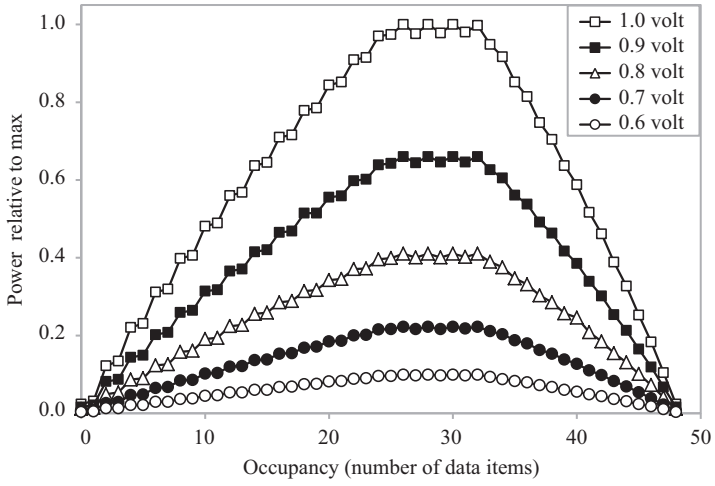

Figure 7.32 Canopy graphs showing power at various supply voltages. The graphs plot the relative power of Ring 4 as a function of ring occupancy at five different power supply levels, using data patterns like 101010 followed by 010101. The measured power is very nearly proportional to (supply voltage − 0.5 volt) ×(supply voltage)^2

the highest voltage. We measured the power as the product of the current drawn and the power supply voltage.

Upon examination one can see that the measured power is very nearly proportional to $(\textit{supply voltage} - 0.5\textit{volt}) \times (\textit{supply voltage})^2$. Figure 7.31 in Section 7.3.5.2 already noted $(\textit{supply voltage} - 0.5\textit{volt})$ as proportional to the throughput. Thus, the first term, $(\textit{supply voltage} - 0.5\textit{volt})$, relates to how many data items per second pass a given point, for example, the counter stage. The second term, $\textit{(supply voltage)}^2$, relates to the energy for forwarding a data item by one stage, which involves charging or discharging the capacitance of the data wires. Power is energy per second, and so the units work out correctly. The graphs in Figure 7.32 serve mostly as a sanity check. The more compelling power measurements follow in Figure 7.33, Section 7.3.5.4.

7.3.5.4 Power for various data patterns

Figure 7.33 shows canopy graphs that plot the active power of Ring 4 as a function of ring occupancy and different patterns of data measured at nominal power supply.

Power is lowest when all data items are identical, because for identical values the data wires need never change. Power is highest for a checkerboard pattern in which data wires adjacent in the layout switch in opposite directions as each data item passes. Power numbers drop slightly if instead of a checkerboard pattern the Weaver alternates all-zeros and all-ones, because of a reduction in side capacitance for adjacent data wires that carry the same value. The intermediate graph in Figure 7.33 is for random data and shows random local variation from sample to sample.

In both the checkerboard graph and alternating all-zeros and all-ones graph, the power numbers ripple between even and odd occupancy, up to about 60% occupancy. For N data items circulating, there are either N or $N - 1$ changes in value depending on whether N is even or odd. Adding one more data item to an existing even set of data items maintains the number of data changes, but adding one more data item to an existing odd set of data items introduces another data change with a corresponding increase in active power—barring congestion.

The power numbers in the graphs for the checkerboard and alternating data patterns differ by only about 5%. The data wires in the Weaver are all double width at double spacing to reduce their capacitive load—see Section 7.3.2.2. The graphs confirm that the side capacitance between data wires contributes relatively little load.

The minimum power measured for circulating a constant data pattern is about one quarter of the maximum power reported in Figure 7.33, for the same occupancy. This minimum reflects the power required to "locally clock" the 72 latches in each occupied stage, making them repeatedly transparent and opaque—even though their data inputs and outputs remain unchanged.

The power measured for circulating a random data pattern is more than half the power measured for circulating a checkerboard pattern. Comparing the two after subtracting the fixed power overhead for "local clocking" provided by the constant data pattern gives $(\textit{random} - \textit{allzero})/(\textit{checkerboard} - \textit{allzero})$, which is about 0.54 over a wide range of occupancies. This is consistent with the statistical model that a random data bit changes about half the time.

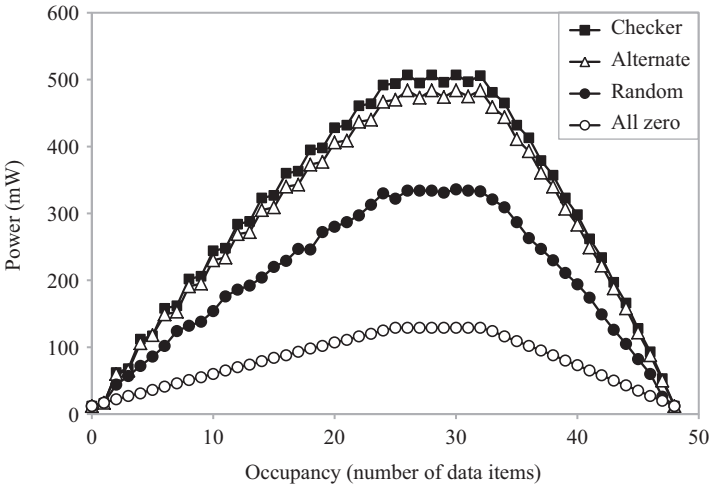

Figure 7.33 *Canopy graphs showing power for various data patterns. The graphs plot the power of Ring 4 at nominal power supply voltage as a function of ring occupancy and four different data patterns. Power depends on how many latches change as data items travel through the Weaver. With constant data (All zero) the latched data remain unchanged, resulting in lowest power. Checkerboard patterns like 101010 followed by 010101 (Checker) cause adjacent data wires to change with every passing data item, resulting in highest power. Patterns with all-zeros alternating with all-ones (Alternating) take almost as much power. Random data (Random) give average power. By combining these power measurements with the throughput measurements in Figure 7.30, one can estimate that the energy to forward one data item one stage is at most 3 picojoules—for details, see end of Section 7.3.5.4*

The canopy graphs in Figure 7.33 show clearly that the Weaver's power is determined by how many data bits change when data circulate through the rings and the crossbar switch. The maximum power of 500 milliwatts is for circulating 60% $\times$ 48 or around 29 data items in a checkerboard pattern through 48 stages of Ring 4, including eight stages in the crossbar switch. Any stage in the Weaver has the same number of latches driving about the same length of data wires. Each stage therefore has the same power when circulating the same pattern at the same speed. So, if we were to circulate a worst-case data pattern at maximum speed through each switching ring, the 8×8 crossbar would run at $(64/48) \times 500$ or about 667 milliwatts.

All switching rings have a throughput up to between 5.5 and 6 GDI/s. Worst-case, a checkerboard pattern with 29 data items running through a switching ring would run at 5.5 GDI/s and 500 milliwatts. If we call x the energy required to forward one data item one stage, then worst-case $x = 500/(5.5 \times 29)$ or 3 picojoules.

7.3.6 Summary and conclusion of Section 7.3

The Weaver implements a simple logical function: an 8 × 8 nonblocking crossbar switch with recirculating channels connecting its eight outputs back to its inputs. Its simplicity allowed us to push its limits. Wide datapaths of 72 bits stretch the Weaver's layout. A short cycle time based on five-gate ring oscillators and a complex flow control with steering bits and arbitration stretch the Weaver's electrical design. The Weaver's high throughput of 6 Giga data items per second per channel—nearly 3.5 Tera bits per second for the full crossbar—is outstanding.

For initialization and at-speed test and debug, the Weaver has separate *go* control in each and every Joint and FULL or EMPTY state access in each and every Link. Its functional simplicity made it possible to read and write all its data through a single reloader stage per channel. Testing the Weaver was a delight—an experience we intend to cultivate further through the Link and Joint model of computation.

The Weaver's logical design separates communication and states in Links from computation and actions in Joints. The Weaver's electrical design maintains this separation. Although all its circuits use the 6-4 GasP self-timed circuit family, they might equally well have used Click.

The Weaver uses different kinds of Links. For instance, the crossbar combines the steering bits and the fill signals in the driver-and-keeper gates of its output Links—see Figure 7.21. Doing so compensates for kiting delay in the data, as explained in Section 7.3.2.2. A novel feature of the Weaver is its use of double-barrel Links. A double-barrel Link bundles data bits with two state signals that carry steering information in one-hot form. The end of Section 7.3.2.2 considers whether to view a double-barrel Link as a peephole optimization of two mutually exclusive ordinary Links or as a Link with a typed interface that carries data in a different form. Adding type information permits fine-tuning of Link-Joint interfaces.

The Weaver's layout modules package a Joint with the near ends of its Links, cutting the Links where they can be stretched. The layout modules conceal the Link-Joint interface and expose the handshake interface—an unfortunate side effect. All too often, designers let layout guide the way they design. The Link and Joint model guides the Weaver's design. And that has made all the difference.

References

[1] Ivan Sutherland and Scott Fairbanks. GasP: a minimal FIFO control. In *International Symposium on Asynchronous Circuits and Systems*, pages 46–53, 2001.

[2] Robert J. Proebsting. Speed enhancement technique for CMOS circuits. US patent US 5,343,090, assigned to National Semiconductor Corporation, 1994.

[3] Ivan Sutherland, Bob Sproull, and David Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann, 1999.

[4] Swetha Mettala Gilla, Marly Roncken, Ivan Sutherland, and Xiaoyu Song. Mutual exclusion sizing for Hoi Polloi. *IEEE Transactions on Circuits and Systems II—Express Briefs*, 66(6):1038–1042, 2018.

[5] Jo Ebergen, Jonathan Gainsley, and Paul Cunningham. Transistor sizing: how to control the speed and energy consumption of a circuit. In *International Symposium on Asynchronous Circuits and Systems*, pages 51–61, 2004.

[6] Ivan Sutherland and Jon Lexau. Designing fast asynchronous circuits. In *International Symposium on Asynchronous Circuits and Systems*, pages 184–193, 2001.

[7] Swetha Mettala Gilla. Silicon compilation and test for dataflow implementations in GasP and Click. PhD thesis, Electrical and Computer Engineering, Portland State University, 2018.

[8] Hoon Park. Formal modeling and verification of delay-insensitive circuits. PhD thesis, Electrical and Computer Engineering, Portland State University, 2015.

[9] Hoon Park, Anping He, Marly Roncken, Xiaoyu Song, and Ivan Sutherland. Modular timing constraints for delay-insensitive systems. *Journal of Computer Science and Technology*, 31(1):77–106, 2016.

[10] Marly Roncken, Swetha Mettala Gilla, Hoon Park, Navaneeth Jamadagni, Chris Cowan, and Ivan Sutherland. Naturalized communication and testing. In *International Symposium on Asynchronous Circuits and Systems*, pages 77–84, 2015.

[11] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click elements: an implementation style for data-driven compilation. In *International Symposium on Asynchronous Circuits and Systems*, pages 3–14, 2010.

[12] Edsger Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[13] Marly Roncken, Ivan Sutherland, Chris Chen, *et al*. How to think about self-timed systems. In *Asilomar Conference on Signals, Systems, and Computers*, pages 1597–1604, 2017.

[14] Alain Martin. The probe: an addition to communication primitives. *Information Processing Letters*, 20:125–130, 1985.

[15] IEEE-SA Standards Board. IEEE Standard Test Access Port and Boundary-Scan Architecture, IEEE Std 1149.1-2001 (Revision of IEEE Std 1149.1-1990), 2001.

[16] Sandra Jackson and Rajit Manohar. Gradual synchronization. In *International Symposium on Asynchronous Circuits and Systems*, pages 29–36, 2016.

[17] Charles Seitz. Chapter 7: System timing. In Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, pages 218–262. Addison-Wesley, 1980.

[18] Gennette Gill and Montek Singh. Automated microarchitectural exploration for achieving throughput targets in pipelined asynchronous systems. In *International Symposium on Asynchronous Circuits and Systems*, pages 117–127, 2010.

[19] Ted E. Williams and Mark A. Horowitz. A zero-overhead self-timed 160-ns 54-b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, 1991.