Local Stall Propagation

David Harris
Harvey Mudd College

September 29, 2000

Abstract

Microprocessors commonly encounter data hazards that must be resolved by stalling the pipeline until all inputs become available. The stall is broadcast globally to previous stages so no tokens are lost. As propagation delays of cross-chip wires exceed 1 cycle, it is no longer possible to stop the processor "on a dime." Local Stall Propagation is a systematic approach to solving the problem. In such a system, each pipeline register contains storage for two tokens. On a stall, the pipeline retains its primary token, but accepts a secondary token from the previous stage that has not yet received the stall. An entire cycle is available to communicate the stall to the previous stage. Flip-flops, transparent latches, or pulsed latches may be modified to hold two tokens per stage with minimal additional hardware or latency and while maintaining scan capability. Valid bits provide a natural mechanism for automatic power reduction.

I. Introduction

Microprocessors commonly encounter data hazards. Some data hazards can be resolved by forwarding, but many, such as use after load, require stalling the pipeline until data becomes available. Stalling a pipeline involves stopping not only the pipe stage experiencing the data dependency, but previous stages as well. Broadcasting the stall across the die is time consuming and is becoming a very difficult speed path in modern processors. In Section II, we will see that scaling trends will make simplistic global stall implementations impossible in future large high-performance processors. A number of alternatives have been used recently, but these approaches suffer from latency, hardware cost, and lack of scalability. Section III introduces an approach called local stall propagation. Special "stall registers" are used that can hold two tokens. In normal operation, both tokens are identical. When a stall occurs, the second token is preserved unchanged, as must occur in a stall, but the first token is accepted from the previous stage. Therefore, the previous stage does not have to stall immediately. A stall signal is locally propagated backward through the pipeline one stage per cycle, eliminating the tight timing constraint on global stalls. Section IV addresses low-overhead stall register designs for systems using flip-flops, pulsed latches, and transparent latches, including the support of clock enabling and scan. Section V presents simulation results validating the method.

II. Global Stalls

In-order microprocessor pipelines stall on unresolved data dependencies [1]. For example, a machine language program may load data from memory into register 1 on one instruction, then add registers 1 and 2 on the next instruction. If the load takes multiple cycles, perhaps because of a 2-cycle data cache latency or a data cache miss, the add cannot begin on the cycle immediately following the load. Instead, the pipeline must stall until the data becomes available. A stall involves holding the current inputs to the pipeline stage until the missing data arrives. In an ordinary pipeline, this requires that all the previous stages also hold their data so that tokens don't get lost. Stalls are commonly caused by instruction or data cache misses, TLB misses, and by data dependencies on instructions with long latencies, such as loads, floating point operations, or transfers between multimedia and integer execution units.

A pipeline stage that generates a stall must broadcast the stall to all the previous stages. Each stage must only enable its input registers if none of the subsequent stages generate stalls. For example, Figure 1 shows an abstract pipeline. Each of the three stages consists of a clocked register followed by a block of combinational logic. The combinational logic may generate a stall signal (s1-s3). On a stall, the input register should remain unchanged and the previous stages also must stall to avoid losing data. The load enable signals (l1-l3) are deasserted during the stall.

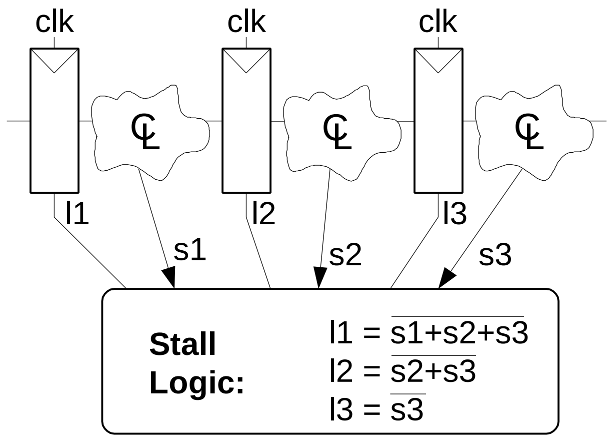

Figure 1: Pipeline with global stalls

Cycle	Stage 1	Stage 2	Stage 3	Notes
1	A	*	*	
2	B	A	*	
3	C	B	A	
4	D	C	B	s3 asserted
5	D	C	B	
6	D	C	B	
7	D	C	B	s3 released
8	E	D	C	
9	F	E	D	
10	G	F	E	
11	H	G	F	
12	I	H	G	s2 asserted
13	I	H	*	s2 released
14	J	I	H	
15	K	J	I	

Table 1: Movement of data through pipeline with global stalls

Table 1 shows the movement of data through the pipeline as stalls are asserted and released. The letters represent tokens moving through the pipeline. When a stall occurs in a given stage, the token is held in that stage and the previous stages stall too. Bubbles (*) consisting of invalid data move into subsequent stages.

The critical path for a stall typically involves some logic to determine that a stall must occur, followed by flight along a long line, a NOR function to merge stalls, and buffering to drive the register enables. As gate delays shrink relative to wire delays, the flight time is becoming a greater problem. Figure 2 shows the distance an optimally repeated signal may travel in a CPU clock cycle for various technology generations [2]. It is becoming impossible to drive a stall to all parts of the chip in a single cycle, even using wide upper-level metal lines and repeaters. In other words, it is becoming impossible to “stop the processor on a dime.”

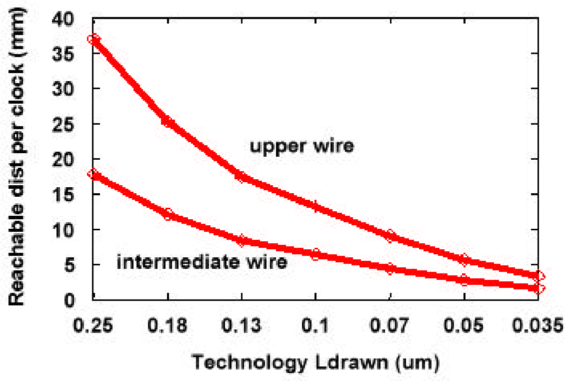

Figure 2: Reachable distance per clock

Microprocessor designers have dealt with this problem in a number of ad-hoc ways. Many exploit the specific structure of the pipeline. For example, an out-of-order processor inherently provides a dispatch queue that decouples the front-end instruction fetch logic from the back-end execution logic. Therefore stalls only impact part of the pipeline. Some machines use a replay mechanism in which extra registers are provided to hold original values in case a stall occurred. In ordinary operation, the extra registers are ignored. When a stall occurs that cannot be broadcast in time, incorrect data is written into the normal pipeline registers. When the stall arrives, the incorrect data is dropped and the pipeline state is reconstructed from the extra registers. This is expensive in terms of chip area and adds a multiplexer delay to the critical path. Asynchronous pipelines use a request/acknowledge handshake to pass data. This can remove the need for global stalls, but asynchronous designs raise their own difficulties.

The Alpha 21264 combines several approaches for stall control in the instruction queues. In the integer queue, a replay or "bypass path" is provided in the event of late stalls, but this is costly in area and delay: 320 bits are enqueued per instruction each cycle. In the floating-point queue, a simpler estimator was used to determine queue fullness. The estimator provides the stall earlier and eliminates the need for the bypass path, but pessimistically forces stalls in some cases when the queue is not yet full, increasing the total number of stalls by 20%. Clearly stall signals are a challenge for high-speed designs and a mechanism of stalling using only short wires is beneficial.

III. Local Stall Propagation

We now consider a more scalable approach to stalls called local stall propagation. Local stall propagation requires the capability of storing two tokens rather than just one at each pipeline stage. The primary token contains the data being processed. Ordinarily the secondary token is identical to the primary token. When we stall the pipeline, we retain the primary token. However, there may not be enough time to stall the predecessor stage. Therefore, we allocate storage for a secondary token that is delivered by the predecessor. This gives us an entire extra cycle to pass the stall information to the predecessor stage by lowering an acknowledge signal. When we release the stall, the secondary tokens that have accumulated may gradually continue down the pipeline. Table 2 shows the primary and secondary tokens in such a locally-stalled system.

Figure 3 shows a pipeline using local stalls. Observe how stalls no longer must be broadcast across the entire pipeline. Instead, they are propagated serially between pipeline stages on acknowledge (a) lines, with a full cycle available for to pass between each stage. Acknowledge goes low when a stage can no longer accept new data.

Two load enable signals, ls and lp, are required by each register to control the primary and secondary storage. Table 3 describes their operation, relating the primary and secondary tokens (P, S) and the data input D. Recall that the primary token P is the output of the register.

Cycle	Stage 1		Stage 2		Stage 3		Notes
	S	P	S	P	S	P	
1	A	A	*	*	*	*	
2	B	B	A	A	*	*	
3	C	C	B	B	A	A	
4	D	D	C	C	B	B	s3 asserted
5	E	E	D	D	C	B	
6	F	F	E	D	C	B	
7	G	F	E	D	C	B	s3 released
8	G	F	E	D	C	C	
9	G	F	E	E	D	D	
10	G	G	F	F	E	E	
11	H	H	G	G	F	F	
12	I	I	H	H	G	G	s2 asserted
13	J	J	I	H	*	*	s2 released
14	K	J	I	I	H	H	
15	K	K	J	J	I	I	

Table 2: Movement of data through pipeline with local stalls

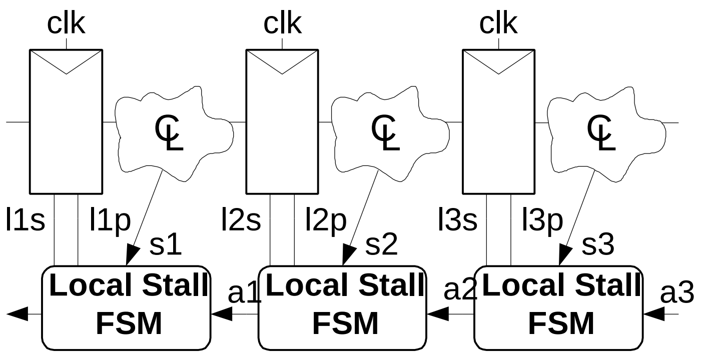

Figure 3: Pipeline with local stalls

ls	lp	Operation
0	0	no change
0	1	P=S, S unchanged
1	0	S=D, P unchanged
1	1	P=D, S=D

Table 3: Operation of latch enables

A simple locally-stalled register can be constructed from two flip-flops, as shown in Figure 4b. A more efficient implementation requires only three latches, rather than the four usually contained in two flip-flops, as shown in Figure 4c. In Section IV, we will develop a register using only two latches.

The local stall logic is a two-state finite state machine shown in Figure 5. The transitions are based on an internal signal *x*, which is true when the current stage generates a stall or the successor propagates a stall by deasserting the ao acknowledge. It generates load enables ls and lp and an acknowledge ai indicating the predecessor may safely send data. From the state transition diagram, we derive the next state and output logic.

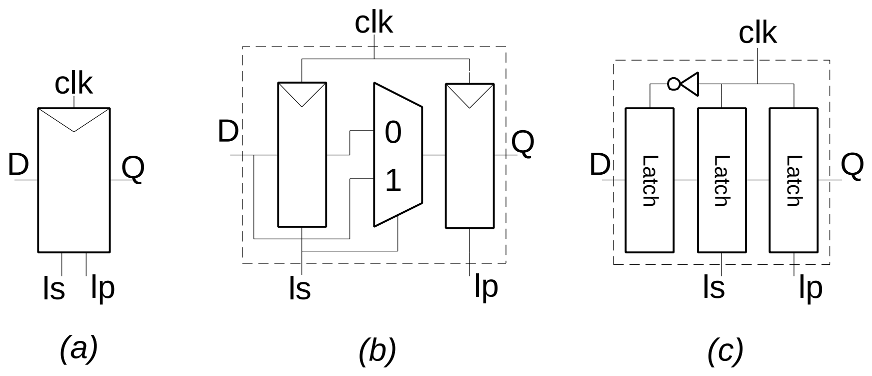

Figure 4: Locally stalled register (*a*) implemented with two flip-flops (*b*) or three latches (*c*).

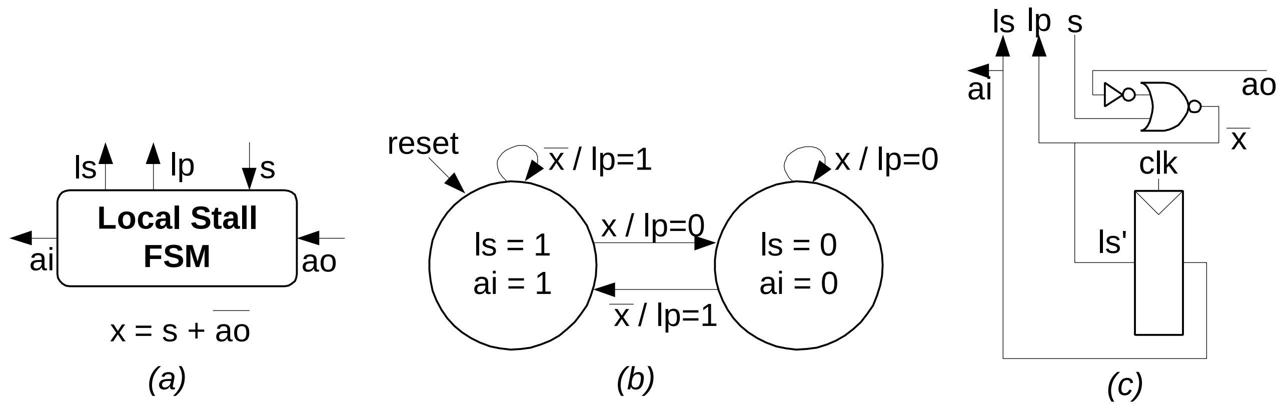

Figure 5: Local stall FSM: (a) interface, (b) state transition diagram, and (c) circuit implementation

When a pipeline stage stalls, it must deassert an associated valid bit to indicate that the output data is not valid so that the data will not be written to register files or otherwise contaminate the machine. When the valid bit is low, it must be passed along the pipeline but the data need not propagate because it is invalid anyway. Therefore, the valid bit can be used as an additional enable on data registers to eliminate unnecessary switching and automatically reduce the power consumption of units where no useful computation is occurring.

In this locally stalled system, a full cycle is available to compute the stall and disable the primary half of the register. Another cycle is available for the deasserted acknowledge to be driven to the previous pipeline stage. This is much better than having only one cycle to compute the stall and drive it to all previous pipeline stages, as required for simple global stalls. Sometimes the extra time is not all necessary. For example, we may have enough time in the second cycle to drive the deasserted acknowledge to two previous pipeline stages. In such a case, we can use a hybrid approach in which we alternate locally-stalled registers and ordinary registers to reduce the overhead, as illustrated in Figure 6.

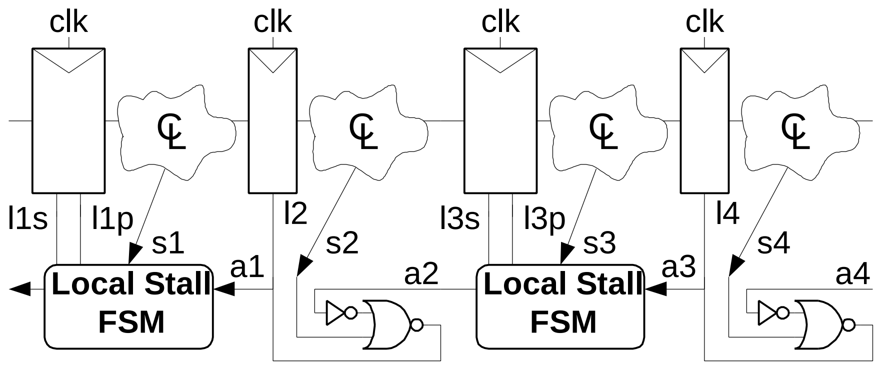

Figure 6: Hybrid approach broadcasting acknowledges across two stages

In this example, a low acknowledge a2 from the second FSM is broadcast to disable both l2 and l1p. Similarly, a stall generated at s2 must disable l2 and l1p. A stall at s1 only immediately disables l1p.

The hybrid design presents exactly the same data and acknowledge interface as does an ordinary locally-stalled pipeline. Therefore, the designer can mix and match broadcasting stalls to one, two, or even more predecessor stages as timing dictates.

IV. Stall Register Design

Figure 4 showed methods of constructing locally stalled registers using four or three latches with a latency penalty of one multiplexer or latch delay. This section develops an improved flip-flop using only two latches. It then extends the design for systems using two-phase transparent latching or pulsed latches [4] rather than flip-flops. Locally-stalled registers hold twice the state and therefore might be expected to have twice the overhead in a scan chain. A better approach requires only a single additional access transistor and a modification to the local stall FSM.

A. Flip-Flops

The locally-stalled register designs of Figure 4 are straightforward but are more expensive in area and latency than ordinary master-slave flip-flops. An improved design separately controls the enables to the master and slave latches in a master-slave flip-flop so the latches behave as the secondary and primary state storage, respectively. Figure 7 shows the datapath and local stall FSM for such a design. The FSM exactly matches that given in Figure 5, but lp is latched to ensure it does not glitch while clk is high. Now the datapath incurs zero overhead in latency. No additional latches are necessary in the datapath, though the latch area may be larger because both latches must accept enable inputs. The small cost of the local stall FSM is amortized across the large number of registers in the datapath.

B. Transparent Latches

Systems using two-phase transparent latches may use a similar approach to provide local stalls with no area or delay overhead in the datapath, as shown in Figure 8. Each stage of logic is divided into two half-cycles separated by transparent latches. The heavy dashed lines indicate the pipeline stage boundary; the second latch in the previous pipeline stage is also shown. The first latch of each stage holds the primary token. The second latch in the previous stage holds the secondary token. Note that the stall may be generated in either the first or second half-cycle of the pipeline stage. It is shown coming from the first half. If the stall comes from the second half of the cycle and might be critically late, the clkb latch in the FSM may be moved to latch ao rather than x to reduce the latency from stall generation to deasserting lp.

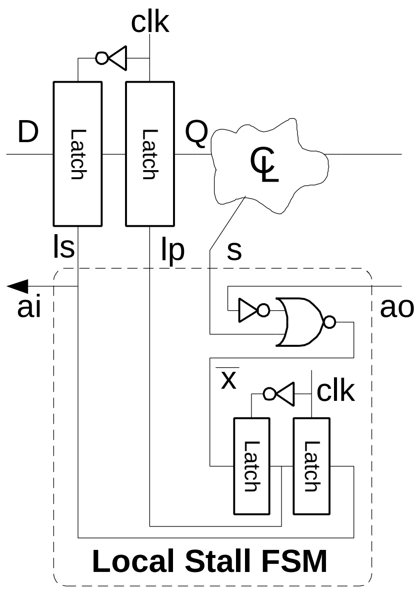

Figure 7: Locally-stalled flip-flop using two latches, along with corresponding local stall FSM

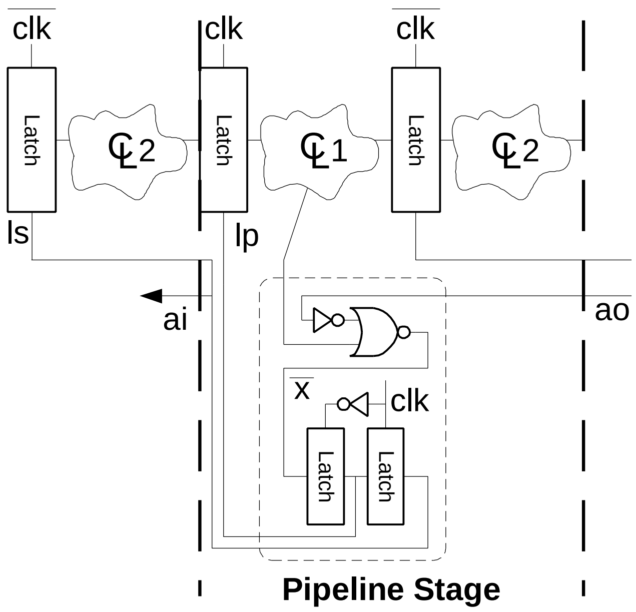

Figure 8: Locally-stalled transparent latch system, along with corresponding local stall FSM

C. Pulsed Latches

In globally-stalled pipelines, pulsed latches are attractive because they require only one latch delay per cycle [4]. Unfortunately, this offers no place to store the secondary token in locally-stalled pipelines. Therefore, locally-stalled pipelines using pulsed latches require a second pulsed latch, as shown in Figure 9. This increases the area and adds one latch delay of latency to the critical path. The system uses an ordinary local stall FSM like that of Figure 5.

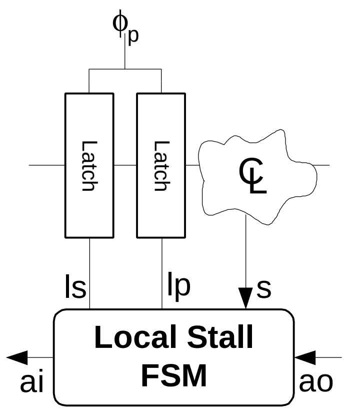

Figure 9: Locally-stalled transparent latch system, along with corresponding local stall FSM

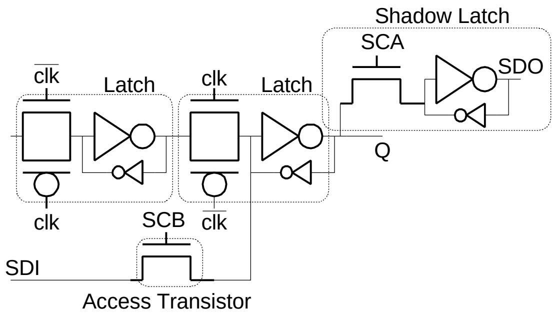

Figure 10: Ordinary scannable flip-flop

D. Scan

An ordinary scannable flip-flop is shown in Figure 10. The global clock is stopped low with the first latch transparent and the second latch opaque. The small inverters represent weak feedback devices. A third "shadow" latch and an access transistor are added for scan. Scan clocks SCA and SCB are alternately pulsed to walk data through the scan chain from SDO of one flop to SDI of the next. When scan is complete, the global clock restarts.

When we construct locally-stalled flip-flop like that of Figure 7, both latches may be opaque and contain tokens if the machine is stalled. We must be able to scan both tokens out and scan new data in to both latches to get complete scan coverage. Moreover, as the scan chain modifies the values of data elsewhere in the pipeline, we must not allow the first latch to become transparent and lose its token. To solve this problem, we can add control to the local stall FSM forcing ls low during scan so the first latch is opaque. Finally, we require an extra access transistor to move data through both latches. It is controlled by a third scan clock SCC, as shown in Figure 11. The scan procedure is otherwise unchanged, so locally stalled flip-flops can be mixed with ordinary flip-flops in the scan chain.

An alternative approach is to build flip-flops with three latches as shown in Figure 4c (plus the shadow latch), avoiding the need for the special latch disabling. This comes at the cost of the extra latch.

V. Simulation Results

A linear 4-stage pipeline using locally-stalled flip-flops was modeled in Verilog and simulated to verify the logic. The simulation waveforms are shown in Figure 12.

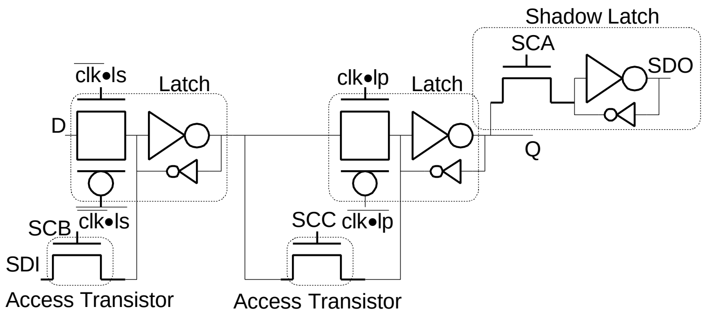

Figure 11: Locally-stalled scannable flip-flop

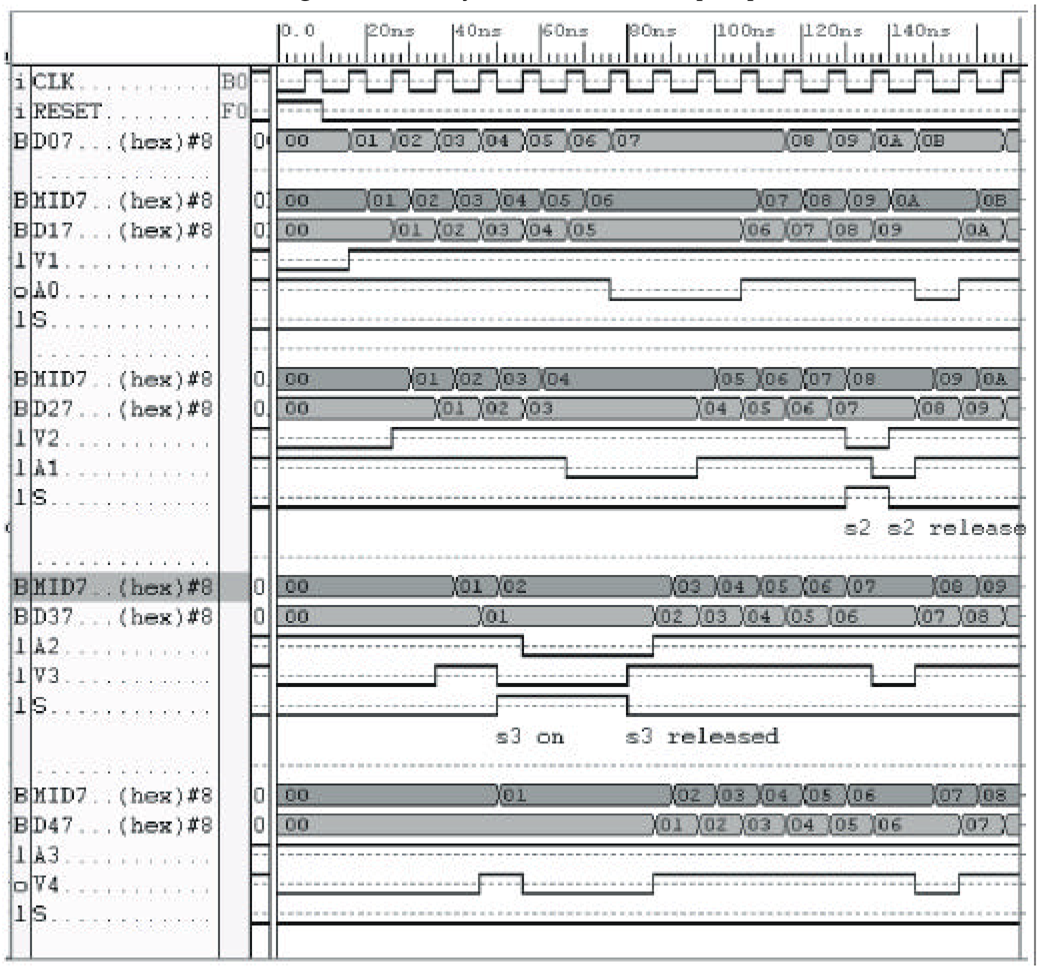

Figure 12: Pipeline simulation results

Stalls are applied as with the example from Table 2. Tokens 0, 1, 2, … correspond to A, B, C, …, respectively. The primary token D, secondary token Mid, valid signal V, acknowledge A, and stall S of each of the four stages are shown. A stall is asserted when token B (01) reaches stage 3 (D37) and is held for three cycles. The output of the stage is immediately marked invalid and the acknowledges fall and work their way back along the pipeline. When the stall is released, the output becomes valid again and the acknowledges are reasserted. Similarly, another stall is generated when token H (07) reaches stage 2 (D27) and is held for a single cycle.

The Verilog module describing a stage is shown in Figure 13. The module illustrates the use of the valid bit to disable the data flops to reduce power.

```
module stage(clk, reset, di, do, vi, vo, ai, ao);
      input       clk, reset;
      input [7:0] di;
      output[7:0] do;
      input       vi, ao;
      output      vo, ai;

      wire  [7:0] dm;
      wire        vm, s, ls, lp;

      flop8 dataflop(clk, reset, (ls & vi), (lp & vi), di, dm);
      flop validflop(clk, reset, vi, vm);
      stagelogic stagelogic(clk, reset, dm, vm, lp, do, vo, s);
      stallfsm stallfsm(clk, reset, ai, ao, s, ls, lp);
endmodule
```

Figure 13: Verilog `stage` module with data and valid bits

VI. Conclusion

In conclusion, we have seen that stalls no longer can be broadcast globally in a single cycle. Local stall propagation offers a systematic approach to solving the timing problems of stalls. In such a system, each pipeline register contains storage for two tokens. On a stall, the pipeline retains its primary token, but accepts a secondary token from the previous stage that has not yet received the stall. An entire cycle is available to communicate the stall to the previous stage. Systems using flip-flops, transparent latches, or pulsed latches may be modified to hold two tokens per stage with minimal additional hardware or latency and while maintaining the capability to scan all the state of the machine.

Acknowledgments

This work was funded by a Beckman New Faculty grant from Harvey Mudd College. Ivan Sutherland and Ian Jones provided valuable suggestions from their asynchronous point of view. David Diaz assisted with simulation.

References

[1] D. Patterson and J. Hennessy, *Computer Organization & Design, 2nd Edition.* San Francisco, CA: Morgan Kaufmann, 1998, p. 489-495.

[2] R. Ho, K. Mai, H. Kapadia, and M. Horowitz, "Interconnect Scaling Implications for CAD," *Intl. Conf. CAD*, 1999, pp. 425-429.

[3] T. Fischer and D. Leibholz, "Design Tradeoffs in Stall-Control Circuits for 600-MHz Instruction Queues," Proc. Intl. Solid-State Circuits Conf., Feb 1998, pp. 232-233.

[4] D. Harris, *Skew-Tolerant Circuit Design.* San Francisco, CA: Morgan Kaufmann, 2001.