Chapter 5

The Shadow Technique

5.1 Event Driven Scheduling and Efficiency

The event driven algorithms presented in Chapters 3 and 4 require a significant amount of scheduling code. The amount of code executed can be reduced significantly by the compiled event-driven techniques presented in Chapter 3, but the total quantity of code explodes with the growing circuit size. The explosion in code size can cause several problems. For extremely large circuits, the code may grow to the point where it is impossible to run the program on any existing system. Even for modestly sized programs, there is a limit to the amount of code that can reside in the instruction cache of a computer. Suppose Algorithm A and Algorithm B execute the same sequence of instructions. Suppose further that Algorithm A fits entirely within the instruction cache, but Algorithm B is too large to fit in the cache. Algorithm A will normally exhibit much better performance than Algorithm B, because Algorithm A can be executed directly from the cache.

The code generated by the threaded code technique is not particularly “cache-friendly.” Because each net and each gate has its own subroutine, code reuse during the simulation of a single input vector is minimal. In other words, the locality of reference is poor. In most cases, there will be too much code for the entire simulator to fit in the cache. Combined with the poor locality of reference, this virtually eliminates any beneficial effect of the cache.

Before discussing methods for handling this problem, it is necessary to examine why such a large amount of code is necessary. xxx illustrates two gates and the code which is generated for each.
The two generated routines shown in Fig. 5-1 are identical except for the variable names. These routines are generated in this way to eliminate the decoding of gate types, and to eliminate the input-net processing loop. One way to reduce the amount of generated code would be convert these two routines into a common subroutine, such as that illustrated in Fig. 5-2.

```
TwoInputAnd(Op1, Op2, Out, NetID)
{
    NewVal := Op1 And Op2
    If NewVal ≠ Out Then
        Queue(NetID, NewVal)
    EndIf
    Address := PopQueue()
    Go To *Address
}
```

**Figure 5-2. A Gate-Simulation Subroutine.**

The subroutine of Fig. 5-2 leaves several questions unanswered. It is not clear what calls the `TwoInputAnd` subroutine. It also appears that the subroutine exits by branching to another address, rather than by executing a proper return. Since the individual routines have been eliminated, it is not clear what the target of the final indirect branch will be. Even if these problems could be resolved, there will be efficiency problems. A subroutine call is seldom as efficient as an indirect branch, particularly when it is necessary to push operands onto the stack. Access to the operands may not be as efficient, since accessing stack values requires the CPU to compute the operand address, rather than using a direct address from the instruction. (Depending on the CPU this may or may not be a problem.)

### 5.2 The Shadow Technique

The shadow technique is a method for creating general-purpose gate simulation routines, without the complication of calling subroutines. The basis of this technique is a data structure called a shadow, which is used to provide a portion of the environment for the gate simulation routines. In a typical program, the stack is used to provide an
environment for a subroutine. The environment contains subroutine parameters and local variables. Access to the environment is through a register called the Stack Pointer. The shadow technique uses a second register, called the Shadow Pointer, to provide additional parameters and local variables to various segments of the program. As xxx illustrates, there is one standard environment for each subroutine, but there may be many shadows per subroutine.

![Diagram of Environments and Shadows](image)

**Figure 5-3. Environments and Shadows.**

Each shadow will be a statically allocated structure containing the parameters and variables necessary to simulate a single gate. To avoid the overhead of gate-type decoding, each shadow will contain a pointer to the simulation subroutine of the gate. Several shadows may point to the same subroutine. Unlike the threaded code techniques which schedule code segments, the shadow technique schedules the shadows themselves. The final step in processing a shadow, is to dequeue the next shadow, load the pointer into the shadow pointer register, and branch to the processing routine pointed to by the new shadow. xxx gives the structure of a shadow, and xxx shows a shadow-based gate processing routine.
In xxx, shadow variables are indicated using a special **HEAVY FONT**. Each gate must have its own shadow. The shadows are created at compile time, and for the most part, will not be altered during run time. In most cases, net values must be accessible from several different shadows, so the shadow must contain a pointer to the value rather than the value itself. Note that when an event occurs on the output of a gate a shadow is placed in the queue. This shadow is a net shadow, which is illustrated in xxx. Net shadows are used to schedule the processing of events. An example of an event processing routine is given in xxx. There must be one event processing routine for each different fanout found in the circuit.
5.3 The Interpretive Algorithm.

Neither the gate-simulation routines nor the event processors depend on the structure of the circuit. This implies that the gate simulation routines could be pre-compiled and loaded from a library, rather than being generated and compiled. Since the number of different routines, even in the worst case, is quite small, this suggests the possibility of using a single standard simulation routine for all circuits, and generating only the data structures. This procedure permits the shadow algorithm to be used interpretively, eliminating the need for compilation. The standard simulation routine has the drawback that not all of the simulation routines will be used during the simulation of a particular circuit, which implies that unexecuted code could be loaded into the cache. This will affect the performance of the algorithm, but apart from this, one would expect the performance of the interpretive algorithm to be quite close to the performance of the compiled algorithm.

In our implementation of the interpretive shadow algorithm, we provided explicit routines for AND gates with from 2 to 10 inputs, and a generic AND routine for gates with more than 10 inputs. We provided similar routines for other gates such as NAND, OR, and NOR. Similarly, explicit net-handling routines were generated for nets with fanouts from 0 through 10, with a generic routine for larger fanouts. Other, reasonably obvious changes were made to convert the algorithm from a compiled algorithm to an interpreted algorithm.

5.4 Summary

5.5 Exercises