Chapter 5

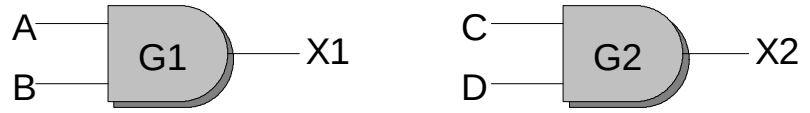
The Shadow Technique

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. Figure 5-1 illustrates two gates and the code which is generated for each.



G1: *NewX1* **:=** *A* **And** *B*
If *NewX1* ≠ *X1* **Then**
Queue(*NetX1***)**
EndIf
Address **:= PopQueue( )**
**Go To ***Address*

G2: *NewX2* **:=** *C* **And** *D*
If *NewX2* ≠ *X2* **Then**
Queue(*NetX2***)**
EndIf
Address **:= PopQueue( )**
**Go To ***Address*

Figure 5-1. Gate Simulation Routines.

The two generated routines shown in Figure 5-2 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 Figure 5-1.

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 Figure 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 Figure 5-3 illustrates, there is one standard environment for each subroutine, but there may be many shadows per subroutine.

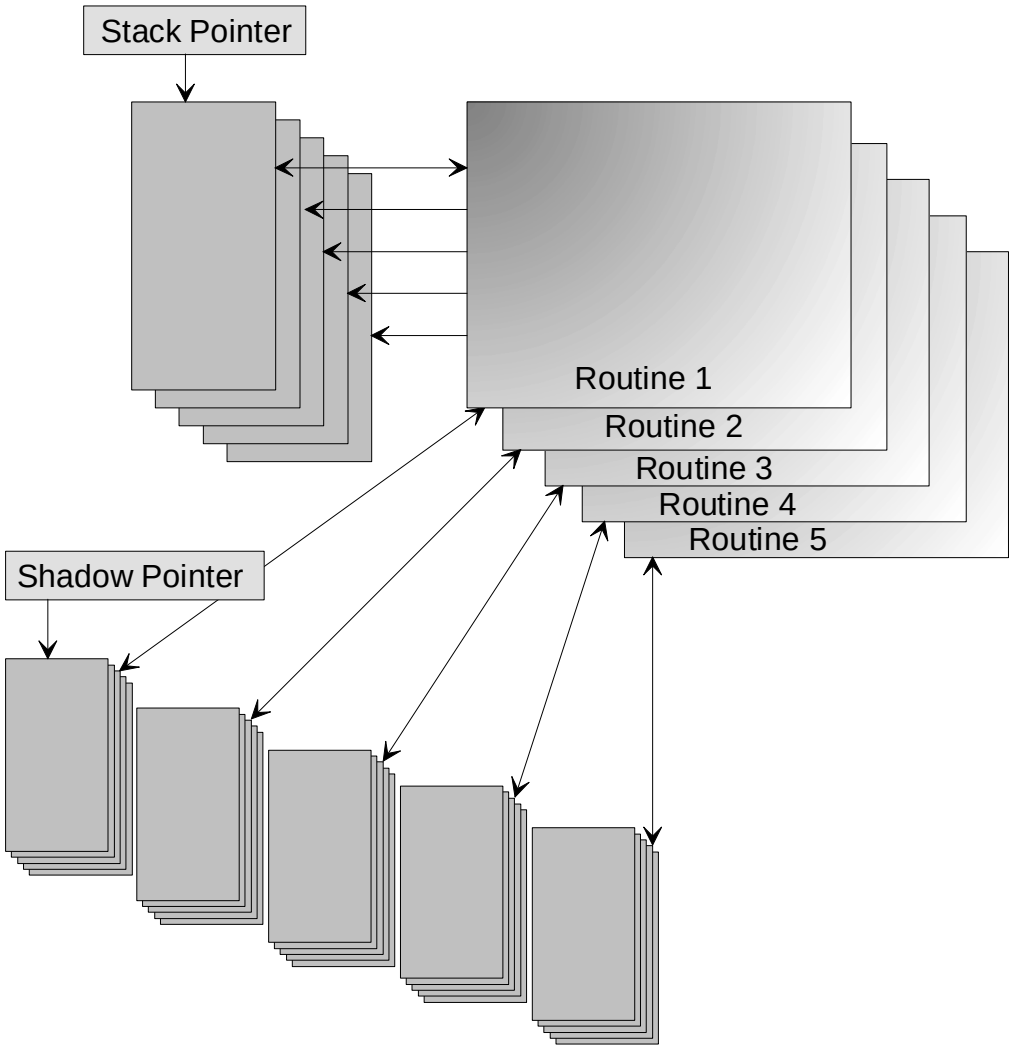


Figure 5-3. Environments and Shadows.

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. Figure 5-4 gives the structure of a shadow, and Algorithm 5-1 shows a shadow-based gate processing routine.

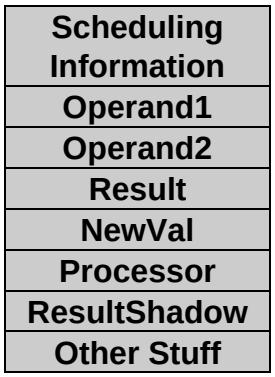


Figure 5-4. The Structure of a Gate Shadow.

```
TwoInputAnd:
    *NewVal := *Operand1 And *OpERAND2
    If *NewVal ≠ *Result Then
        Queue(ResultShadow)
    EndIf
    ShadowPointer := PopQueue( )
    Go To *Processor
```

Algorithm 5-1. Gate Processor for the Shadow Technique.

In Algorithm 5-1, 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 Figure 5-5. Net shadows are used to schedule the processing of events. An example of an event processing routine is given in Algorithm 5-2. There must be one event processing routine for each different fanout found in the circuit.

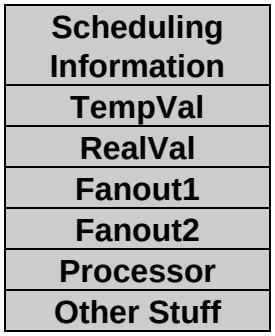


Figure 5-5. The Structure of a Net Shadow.

```
FanoutOfTwo:
    *RealVal := *TempVal
    Queue(Fanout1)
    Queue(Fanout2)
    ShadowPointer := PopQueue( )
    Go To *Processor
```

Algorithm 5-2. Net Processor for the Shadow Technique.

5.3 The Interpretive Algorithm.

Neither the gate-simulation routines nor the event processors depend on the structure of the circuit. Because of this, there is no reason to generate these routines at compile time. It is possible to write a single set of simulation routines which will be used for all circuits, and place these routines in a standard function library. Once this is done, The compiler need only generate the data structures. All run-time code can be included from the standard library during the compilation process.

It is possible to carry this idea one step further. Since the number of different routines, even in the worst case, is quite small, it is possible to place these simulation routines in the compiler, without incurring a severe penalty. If the compiler then places the data structures in temporary storage, rather than compiling them with the library routines, the shadow algorithm can be run interpretively with virtually no performance penalty over the compiled method.

The interpretive approach will incur a few performance penalties over the compiled approach. Because the interpretive approach uses a standard simulation routine, it is possible that some simulation code will be unnecessary. Depending on the structure of the simulation routine and the underlying computer architecture, it still may be necessary to load some of this code into the instruction cache. If instructions are loaded into the cache in multi-word blocks, it is possible for a block containing useful code to also contain unexecuted instructions. Because the cache is used less efficiently, it is possible for more cache-swapping to occur, leading to poorer performance.

Memory usage in the interpretive approach is also somewhat less efficient. The interpretive algorithm has two distinct phases, the compilation phase in which data structures are generated, and the simulation phase in which the circuit is simulated. Each phase requires different segments of code to be loaded into memory. Most of the code used by one phase will be unnecessary for the other phase. At the very least, there will be a performance penalty when the code for the compilation phase is swapped with the code for the simulation phase. In the worst case, the unused portions of code could occupy memory that would best be used for other purposes.

If the number of gate-inputs is unbounded, it is a practical impossibility to provide simulation routines for each different type of gate that could be encountered during simulation. In the experimental implementation of the interpretive shadow algorithm, this problem was solved by providing explicit routines for AND, NAND, OR, and NOR gates with from 2 to 10 inputs, and a generic routines for gates with more than 10 inputs. Explicit net-handling routines were generated for nets with fanouts from 0 through 10, and

a generic routine was provided for nets with larger fanouts. The use of these generic routines can also lead to a performance penalty, if there are gates in the circuit with large numbers of inputs, or nets with large fanouts.

5.4 Summary

5.5 Exercises