Design of a multi-thread ROM-controller

van Woerkom, H.H.A.

Award date:
1995

Eindhoven University of Technology
Department of Electrical Engineering
Design Automation Section

Design of a Multi-Thread ROM–controller

Master Thesis
by H. v. Woerkom

May 1994 – December 1994
By order of Prof. Dr. -Ing. J.A.G. Jess
Supervised by Ir. H.A. Hilderink

Abstract

Designs generated by high level synthesis tools are becoming increasingly complex and have to execute ever faster. Therefore more and more parallelism has to be used. Present day controllers are not well–suited to drive a parallel design and they require quite a lot of overhead.

The Multi–thread ROM–controller is designed specifically for parallel designs. It is based on a conventional ROM–based controller, but the address decoders have been replaced by latches, so each ROM–row is driven by a separate latch. This way, multiple ROM–rows can be driven at the same time, allowing the controller to have multiple states active at the same time. A further advantage of the Multi–thread ROM–controller is that the state control logic (logic used to advance to the next state or states) can be extremely simple.

This thesis presents a complete design of a Multi–thread ROM–controller. The state control logic will be described in VHDL, while a layout of the ROM–core (including latches) will be presented.

Results indicate that the Multi–thread ROM–controller is an efficient controller, even for single thread designs. It is smaller and almost as fast as a conventional ROM–based controller and in a design with 50 states and 30 outputs, the Multi–thread ROM–controller is about the same size as a random logic controller. In designs with fewer states or less outputs the random logic controller will be smaller, while in a design with more states or more outputs, the Multi–thread ROM–controller will be smaller. The Multi–thread ROM–controller does handle multi–thread designs well, but as other multi–thread controllers are not readily available or perform worse than single thread controllers, no comparisons are made.

Contents

Chapter 1

Introduction

In high level synthesis an architecture is designed that can implement a certain behavioral description. Such an architecture contains a number of logic modules (e.g. registers, adders, multiplexers, multipliers) and interconnect. The modules are controlled by a controller, which is usually specified as a finite state machine.

The algorithms that are synthesized are becoming increasingly complex and they have to be executed in ever fewer cycles. To comply with these timing constraints multiple threads of the algorithm have to be executed in parallel (a thread is a self–contained part of an algorithm). The controller has to be adapted to accommodate these multiple threads.

The conventional solutions to accommodate multiple threads are to translate these multiple thread algorithm into a single thread algorithm, to use a set of communicating controllers, where each one controls a single thread, or to use distributed controllers. The first solution has the problem that the translation step can cause a state explosion, which can cause the controller to become extravagantly large. The second solution causes a lot of overhead in the communication between the controllers. The third solution has not yet been properly studied, but it leads to a large number of (very) small controllers, which may use up less space than one central controller.

Controllers are usually implemented as either a random logic block or a PLA– or ROM–based controller. A random logic block is ideal for small state machines. The logic can be highly optimized and the result is a fast and small controller. For larger state machines a random logic block is usually larger than a PLA/ROM–based controller (in [Gerb92] it is stated that a ROM–based controller becomes more efficient than a random logic controller if the state machine has 100 or more states).

These problems with the conventional implementations arise from the fact that they are essentially single–thread solutions, that is, only one state can be active at any time. A requirement for a multiple thread controller is that more states can be active at the same time. This way all threads can have an active state simultaneously, so each state in the state machine can be projected onto one state in the multiple thread controller, which circumvents the problem of state explosion.

The Multi–thread ROM–controller is a controller that can accommodate multiple threads.

It was devised by H.A. Hilderink (see [Hild93]). It is essentially a ROM–based controller, but the address decoders have been replaced by latches, so each ROM–row is driven by a separate latch. This way, multiple ROM–rows can be driven at the same time. A complete design of a Multi–Thread ROM–controller had not been made yet. This thesis presents such a design and compares it to other possible implementations of controllers.

In chapter 2 the description of the finite state machine is discussed. Chapter 3 deals with the designs on which the Multi–Thread ROM–controller is based. The projection of the finite state machine onto the Multi–Thread ROM–controller is discussed in chapter 4. Chapter 5 deals with the logic that drives the ROM–core and chapter 6 deals with the implementation of ROM–core. Chapter 7 gives results and compares the design to other controller implementations, while Chapter 8 presents conclusions and recommendations.

Chapter 2

Controller specification

A controller is an autonomous module that drives the control signals for the modules in a chip design. The Multi–thread ROM–controller is a controller specifically designed to accommodate multiple threads. Internally it functions as a finite state machine that can handle multiple threads. Branches in the state machine are determined by test–signals, which, together with the reset–, start– and clock–signals, are the only input signals.

The finite state machine for the Multi–thread ROM controller is defined as a 5–tuple (S, C, δ, s, f) in which:

S is the set of states.

C is a set of conditionals for use in state transitions. There must be a special conditional TRUE to represent unconditional state transitions.

δ is the set of triples $(S \times C \times S)$ of state transitions. If $(x, \sigma, y) \in \delta$ then the next state from x will be y if σ is true.

s is the starting state.

f is the final state.

The operator $\rightarrow$ is used to indicate that a state can be reached from another state by one state transition, that is $x \rightarrow y \Rightarrow (x, \sigma, y) \in \delta$. The operator $\overset{*}{\rightarrow}$ is used to indicate that a state can be reached from another state by any number of state transitions (this includes zero transitions, so $x \overset{*}{\rightarrow} x$ is always true). For a finite state machine the following conditions must hold:

- $\forall_{x \in S}\, s \overset{*}{\rightarrow} x$
- $\forall_{x \in S}\, x \overset{*}{\rightarrow} f$

This ensures that all states in the finite state machine can actually be used.

The finite state machine can be represented as a token flow graph (see [Eijn91]). The set of nodes is the set of states S and the set of edges is the set of state transitions δ. In a token flow graph any number of states can have a token. The set T_i contains all

nodes that have a token at time i (this is similar to the current state at time i). The set of successors of set T_i, T_{i+1} is calculated as follows:

$$T_{i+1} = \{y \in S \mid x \in T_i \wedge (x, \sigma, y) \in \delta \wedge \sigma \; is \; true\}$$

To simplify the design of the Multi–thread ROM controller, some restrictions are placed on the finite state machine. There must be a regular path between the starting state and the final state, that is, any control structure must be fully contained in some other control structure (a control structure being, for example, an if–then–else construction). It is not allowed to break out of or in to a control structure.

To formalize this notion, paths will be introduced. A path is a finite state machine that is regular. A path may contain other paths, that is, the contained path has fewer state transitions and may have fewer states. This is denoted by $\sqsubset$ ($P \sqsubset A$ means A contains P). If $P \sqsubset A$ then:

- $S_P \subseteq S_A \wedge C_P \subseteq C_A$
- $\delta_P \subset \delta_A$
- P must comply with the rules for starting and final states.

Note that a path does not contain itself, so $A \not\sqsubset A$. In the following text A will denote the finite state machine itself.

Control structures (if–then–else constructions and concurrent sequences) can be recognized by a branch in a path and a corresponding merge further down the path. Every branch must have its own path, because that branch itself must also be regular. The following condition ensures that every branch has a path of its own, that is, there is only one state transition from the starting state and that transition corresponds with the branch. It also ensures that there is a different path that starts and ends at the same states that the branch starts and ends ($\exists!$ means "there is exactly one"):

$$\forall_{(x,\sigma,y)\in\delta_A \wedge (x,\tau,z)\in\delta_A} \exists_{P \sqsubset A} (s_P = x \wedge \exists!(x, \phi, w) \in \delta_P \wedge \exists_{Q \not\sqsubset P} s_P = s_Q \wedge f_P = f_Q)$$

Because every control structure has its own path, the conditions that you cannot break out of or in to a control structure can be expressed as the condition that a path can only be reached through its starting state and exited through its final state. For this all states outside the path are replaced by one source state and one sink state. The source and sink states may have no state transitions to or from the "inside" of a path (any state except the starting and final state). For each path P a new state machine Q is defined with the extra source and sink states:

- $S_Q = S_P \cup \{source, sink\}$
- $C_Q = C_P$
- $\delta_Q = \delta_P \cup \{(source, \sigma, x) | x \in S_P \wedge y \notin S_P \wedge (y, \sigma, x) \in A\} \cup$
  $\{(x, \sigma, sink) | x \in S_P \wedge y \notin S_P \wedge (x, \sigma, y) \in A\}$

- $s_Q = source$
- $f_Q = sink$

The state transitions of Q from $source$ may only go to s_P or f_P and the transitions to $sink$ may only come from f_P or s_P:

$$\neg\exists_{(source,\sigma,x)\in\delta_Q} x \neq s_P \wedge x \neq f_P$$

$$\neg\exists_{(x,\sigma,sink)\in\delta_Q} x \neq s_P \wedge x \neq f_P$$

Figure 2.1 gives some examples of valid finite state machines. Figure 2.2 gives some examples of invalid finite state machines. Consider the path with states S_3, S_4, S_6 in the left example. The state transition $\{S_2$,TRUE, $S_4\}$ breaks into that path. The right example is similar. Here the state transition $\{S_5$,TRUE, $S_2\}$ breaks into the path with states S_1, S_2, S_4 and S_6.

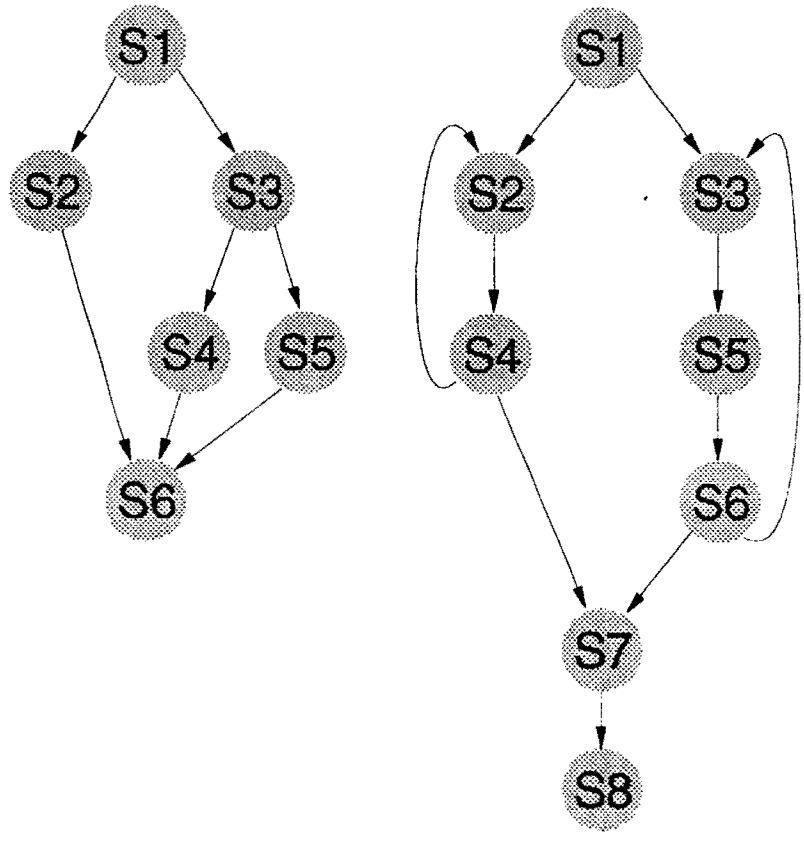

Figure 2.1: Valid finite state machines. Conditions on the edges have no impact on the validity of the state machines, so they have been left out.

The Multi–thread ROM controller can also be used to implement single thread finite state machines (state machines that contain absolutely no parallelism). In this case the restrictions for the finite state machine are lifted, because some behavioural descriptions cannot be expressed as a regular single thread finite state machine.

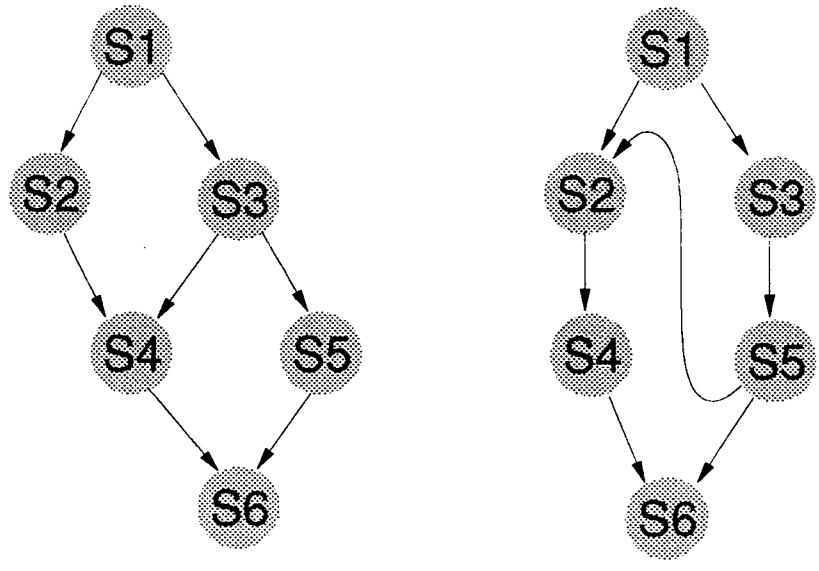

Figure 2.2: Invalid finite state machines. Conditions on the edges have no impact on the validity of the state machines, so they have been left out.

Chapter 3

Basic structure of the Multi-thread ROM-controller

The Multi-thread ROM-controller is based on conventional ROM-based controllers. By applying some changes to the structure of a ROM-based controller the single threaded nature of the controller can be changed into a multi threaded nature.

Conventional ROM-based controllers consist of a ROM-matrix, address decoders and some logic to determine the next state (see figure 3.1).

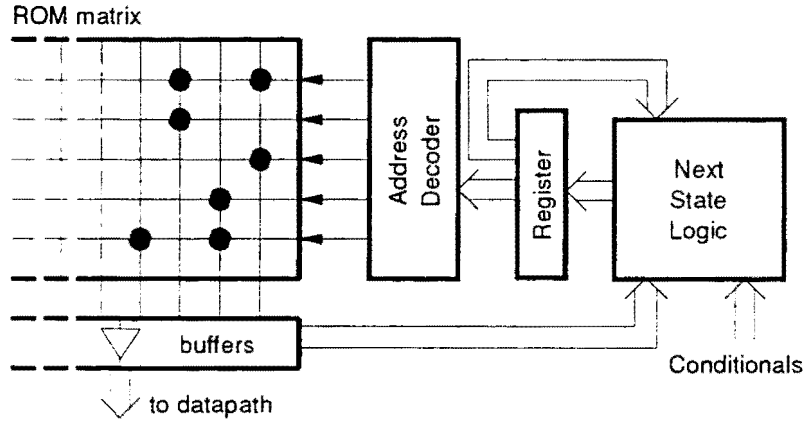

Figure 3.1: A conventional ROM-based controller

Each ROM-row represents a state. The current state is held in a register (often called instruction-pointer), which is $\log_2$(#states) wide. This register drives the address decoders, so exactly one ROM-row is activated. This rules out multiple active states at the same time.

By replacing the address decoders with latches and addressing each ROM-row with a separate latch (see figure 3.2), more states can be made active at the same time. Now the current state is effectively held in a register that is #states wide. The extra space needed

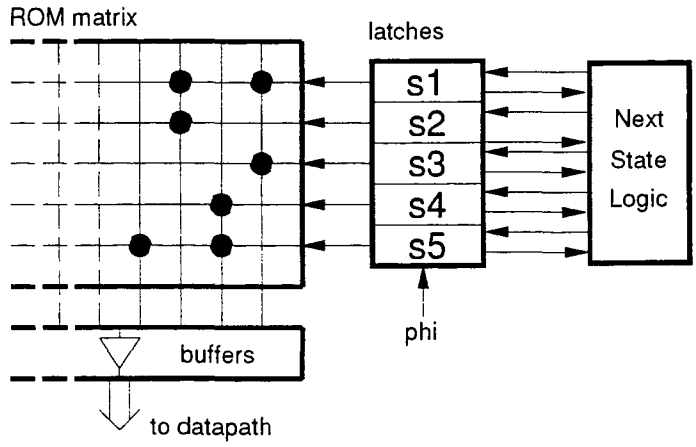

Figure 3.2: Basic Multi Thread ROM–controller structure

for this is offset by the disappearance of the address decoders and the simplification of the logic to determine the next state (see chapter 4). Also, since more than one state can be active at the same time, this structure is not subject to state–explosion, caused by the flattening of a multiple thread state machine. This results in a smaller number of states, so a smaller number of ROM–rows is needed.

Chapter 4

The state control logic

In chapter 2 it was seen that the finite state machine can be seen as a token flow graph. Each state that has a token is active. A state passes its token on to other state(s) at the beginning of each clock cycle. Control structures can be realised by controlling the flow of the tokens.

A state in the Multi–thread controller is active if its corresponding latch has a "one" clocked in. Each clock cycle a latch clocks in its input value, so the input of each latch must be a "zero" if no token arrives and a "one" if a token arrives.

A simple sequence of states can be achieved by chaining the latches (see figure 4.1). If a state is active, it passes a "one" to the following state, so that state will become active in the next cycle. Inactive states pass a "zero", so the following state will not be active.

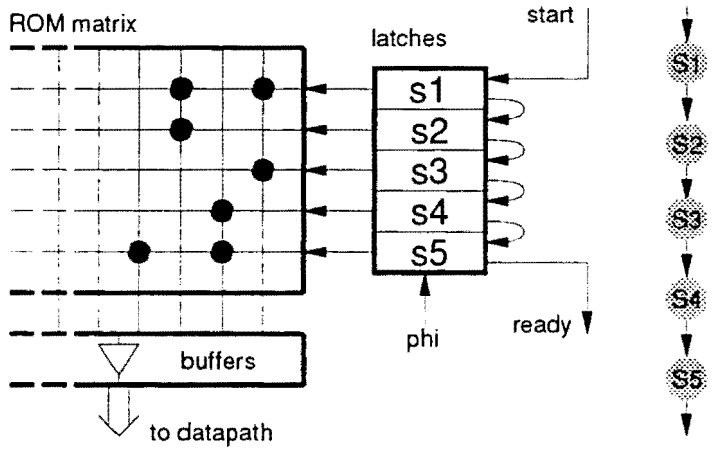

Figure 4.1: Chaining the latches for a sequence of states

In an if–then–else construction the token is passed to the left or to the right branch, depending on some conditional. The token must be ANDed with the conditional for one branch and with the inverse of the conditional for the other branch. This way the token passes to one of the branches. At the end of the if–then–else construction both branches

come together. Now the output of both latches must be ORed, so a token from either of the branches will pass. An example of an if–then–else construction is given in figure 4.2. The token passes from *S1* to *t1* (if *test* is true) or to *e1* (if *test* is false). The token flows through either *t1* and *t2* or through *e1*, *e2* and *e3* and arrives finally at *Sn*.

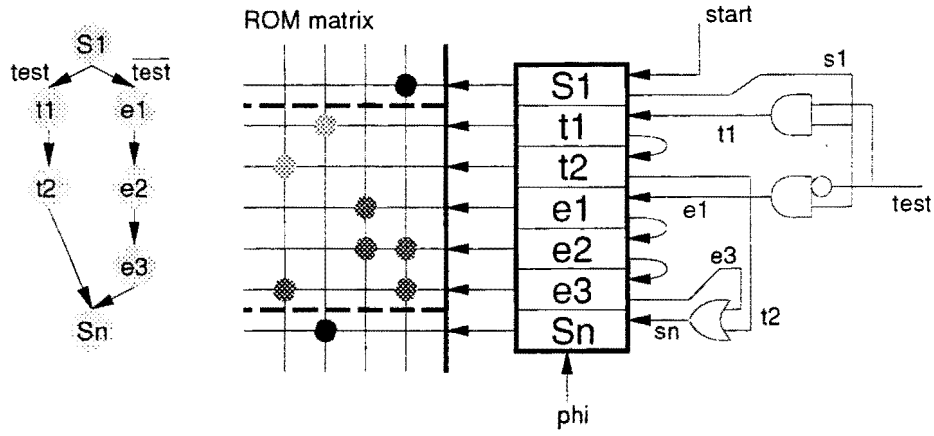

Figure 4.2: An if–then–else construction.

A loop construction is a special case of an if–then–else construction. An example is given in figure 4.3. The token passes from *S1* to *L0* (the OR gate passes the token). The token then flows through *L1* and *L2* to *L3*. After *L3* it will be passed to *L0* if *test* is true (then it passes the AND gate and the OR gate), or it will be passed to *Sn* if *test* is false.

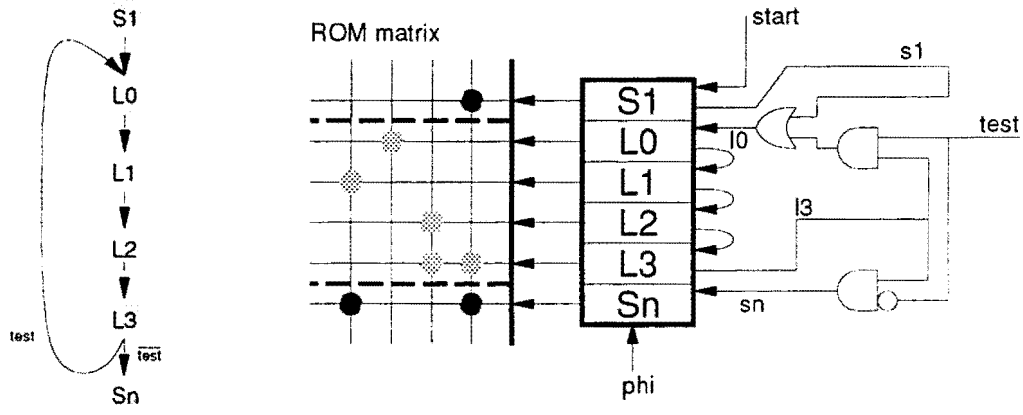

Figure 4.3: A loop construction.

Concurrent sequences can be started by feeding the output of a latch to two (or even more) latches. The first state following the concurrent sequence may only execute if all concurrent sequences are finished. If the slowest of the sequences can be determined, the sequences can be implicitly synchronized (see figure 4.4, left side). In this case the

tokens from the faster sequences are ignored and only the token of the slowest sequence is passed along. If the length of the sequences is not known in advance (this is the case if some sequences contain loop constructions), some extra logic is needed to wait for all sequences to end (see figure 4.4, right side). The wait logic will only execute *Sn* if tokens from both *a2* and *b3* have been received.

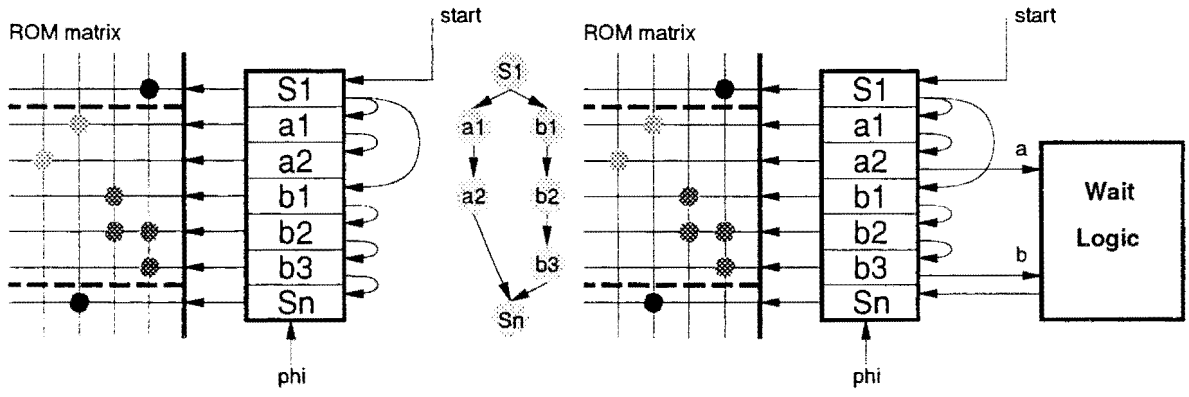

Figure 4.4: Concurrent sequences. To the left, implicit synchronization, to the right, explicit synchronization.

The logic in figure 4.5 can be used to explicitly synchronize two sequences. If a token is received along *a2*, then *X* will be activated. *Y* will be activated when a token passes along *b3*. If *X* and *Y* are active, a token will be passed to *Sn*. This causes *Sn* to be active the next cycle, which deactivates *X* and *Y*.

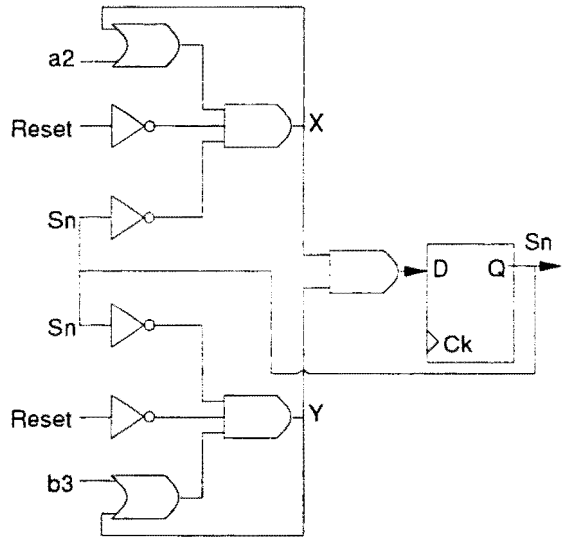

Figure 4.5: Logic for explicit synchronization.

A state can be part of several constructions. In figure 4.6 state X is the ending of an if–

else–then construction, a concurrent sequence and the beginning of a loop. It is important to note that the consequences of a construction affect the input of a state. The output of a state is used, but not directly affected by any construction.

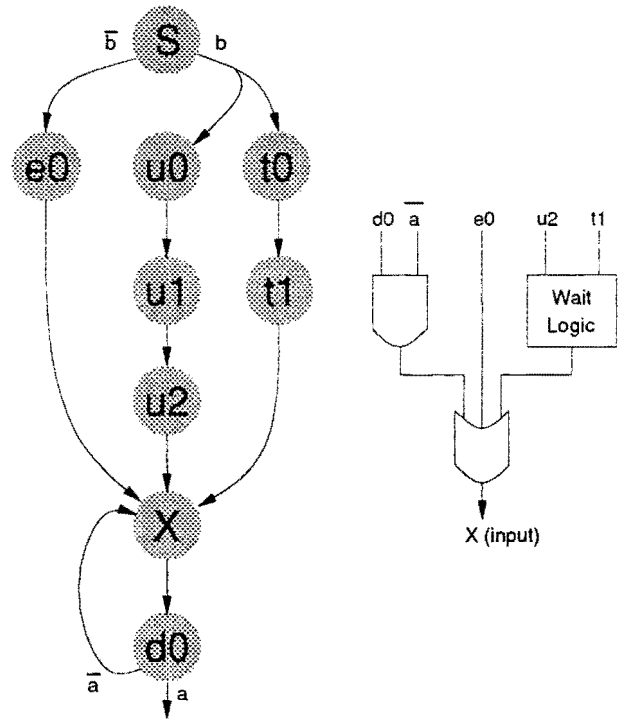

Figure 4.6: State X is part of many constructions.

Chapter 5

The state control logic generator

The state control logic is generated by a state control logic generator. This generator generates structural VHDL-descriptions (see [Ins88]). The input of the generator is a tabular description of a symbolic finite state machine (see appendix C).

5.1 State transition graph

The control flow of the state machine is represented by a token flow graph. Each node in the graph can have many edges to other nodes. These edges are kept in a "link tree" for each node. The link tree is a binary tree that groups edges to the nodes they belong to (there are forward link trees for outgoing edges and backward link trees for incoming edges). The link tree is used to create an easily travesable hierarchy of the edges.

As was noted in chapter 4, logic has to be generated for the input of each state. Because of this it is important to know the possible previous states and the conditions of the transition. The input file, however, gives information about the next state.

The input file contains state transitions which can be used to incrementally create a forward state transition graph. This graph gives complete information about the next state for each possible state. A backward state transition graph can also be generated from the input file, but this graph does not contain complete information about the previous state(s), because it is not immediately clear what kind of structure a node belongs to (see section 5.2). An extra parsing step to complete this information is discussed in section 5.3.

5.2 Link trees

The edges of each node are kept in a link tree. There are two link trees per node, one link tree for the forward state transition graph and one link tree for the backward state transition graph.

A forward link tree has four possible kinds of links:

- an *empty* link, an empty square in figures.
- a *direct* link (linked to the next state), a square with the target state in it.
- an *if* link (contains a conditional and two references to other links, one for the true–condition, one for the false–condition), an I node.
- a *split* link (contains two references to other links), a S node.

The link tree is a binary tree where each link contains at most two branches. Conditional structures or two concurrent sequences can be expressed by an *if* or *split* link. More complex constructions, such as a multi–way branch or more concurrent sequences can be expressed by nested links. Figure 5.1 shows an example of a link tree. This example also shows that the hierarchy of a link tree can be very important.

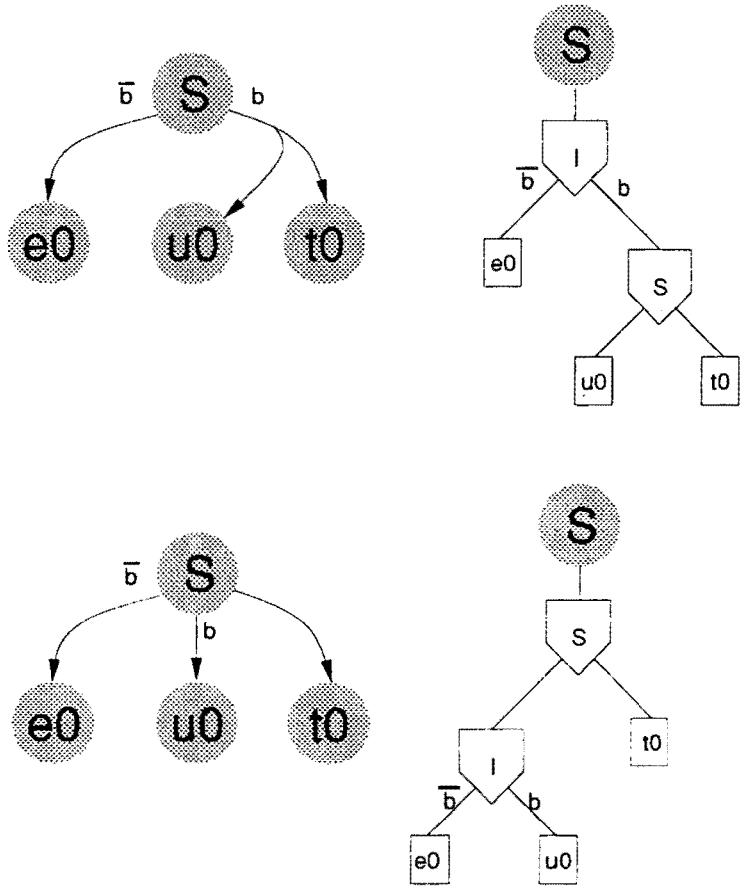

Figure 5.1: Two examples of link trees.

A backward link tree is similar to a forward link tree, but it has five possible kinds of links:

- an *empty* link, an empty square in figures.

- a *direct* link (linked to a previous state), a square with the target state in it.
- a *from–if* link (linked to a previous state and containing a conditional), a square with the target state in it.
- an *end–if* link (contains two references to other links), an E node.
- a *join* link (contains two references to other links), a J node.

From–if links are similar to *direct* links, but indicate that the state can only be reached if some conditions are met (the previous state has one or more if–conditions), whereas a *direct* link is an unconditional link. Note that an *end–if* link does not have a conditional. The conditional is used at the beginning of the if construction, while the *end–if* link signifies the end of the construction.

Link trees are used to model edges in the state transition graph. A *direct* forward link indicates an outgoing edge, while a *direct* or *from–if* backward link indicates an incoming edge. Because of this, each *direct* forward link can be coupled to exactly one unique *direct* or *from–if* backward link. The other kinds of edges are used for the hierarchy of the edges, but do not directly represent an edge.

5.3 backward tree construction

A single step parser cannot generate a backward link tree, because it cannot determine if the convergence of two branches should be indicated by an *end–if* link of by a *join* link. The type of link is determined by the kind of construction that it is part of and this information can only be derived from the complete state transition graph. Instead, the parser creates a backward link tree without hierarchy (a balanced tree) and uses *gather* links (a sixth kind of link, similar to *end–if* and *join* links), G nodes in figures. Later on, the *gather* links will be replaced by *end–if* or *join* links and the hierarchy of the tree will be changed if necessary.

In chapter 2 paths were introduced to define the structure of the state transition graph. Now the notion of paths can be extended: a path has not only a starting state and a final state (both can be shared with other paths), but also a starting link and a final link. The starting link is a link in the forward link tree of the starting state, the final link is a link in the backward link tree of the final state. A link can only serve as starting or final link for one path. An example of some paths is given in figure 5.2 (note that all paths have starting state S and final state F, only the starting and final links differ).

Before the algorithm to complete the backward link tree can execute, all back–branches (edges to an already visited state, mainly used in loop constructions) must be removed. This is done with a depth–first search (see [Corm90], page 477–483). The edges are removed for the duration of the backward tree construction.

Every state has a forward link tree. Every node in that tree divides the paths that start at that state in two distinct groups (the left branch and the right branch). There may be valid paths that contain paths from both branches, but these paths are of no consequence for the backward tree completion (and, incidentally, cannot have a starting

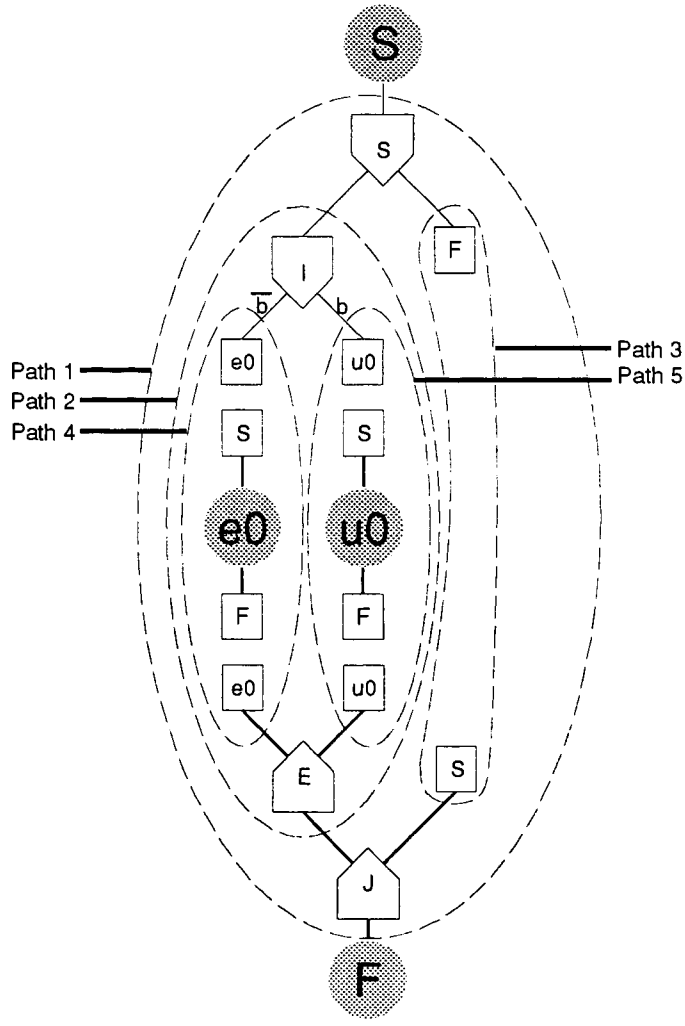

Figure 5.2: Paths and link trees.

link). Both groups terminate at the same state (this follows from the regularity of the state machine) and should be joined by a *join* or *end–if* link.

An algorithm to construct the backward link tree is straightforward. It only has to walk down a path. Whenever it encounters a branch it has to process both branches (by recursion). Both branches must merge at the same state and there the correct final link can be constructed. Some support functions are defined for the algorithm:

type(x). Returns the type of link x.

find–coupled–link(x). Returns the backward link that is coupled to forward link x.

state(x). Returns the state that x belongs to

backward–root(y) and forward–root(y). Return the root of the backward/forward link tree of state y.

join-links(*x1*,*x2*,*t*). Joins two branches and returns the joining link (this function will be detailed shortly).

```
1     TRAVEL–PATH( start–link )
2         if type( start–link ) = empty
3             return "dead end"
4         else if type( start–link ) = direct
5             target–link ← find–coupled–link( start–link )
6             if target–link = backward–root( state( target–link ) )
7                 start–link ← forward–root( state( target–link ) )
8                 goto line 2
9             else
10                return target–link
11        else
12            target1 ← travelpath( left( start–link ) )
13            target2 ← travelpath( right( start–link ) )
14            target–link ← join–links( target1, target2, type( start–link ) )
15            goto line 6
```

In lines 5-8, the algorithm walks down the path (it will continue to do so as long as no branches or merges are encountered). If it encounters a merging of paths (the target–link is not the root of the tree, but a leaf of a *join* or *end–if* link), it returns to the calling function in line 10 (a merging signifies the end of a path).

In lines 12-15 branches are handled. First the function is recursively called for each branch. Then both branches are joined by a *join* or *end–if* link (depending on type(*start–link*)) in line 14. After the branches have been merged, the resulting link is processed just like a normal *target–link*. Note that both recursive calls should return *target–links* from the same final state. If this is not the case, the finite state machine is not regular and an error message is returned.

At first the backward link tree is a balanced tree with *gather* links (which can be renamed to a *join* or *end–if* link). The *direct* and *from–if* links are distributed randomly amongst the leaves of the tree. The backward link tree will always have a *gather* link subtree (any path from root to leaf encounters a number of *gather* links, followed by one or more other types of links). This is maintained by the fact that the arguments *target1* and *target2* of join–links will never be *gather* links (the return value *target–link* in function travel–path can never be a *gather* link, because it becomes either a *direct* or from–if link in line 5, or a *end–if* or *join* link in line 14), but the parents of *target1* and *target2* will be *gather* links (all parents are *gather* links in the beginning and as soon as they are renamed, they are used as a *target–links* themselves in line 14).

The function join–links has to gather the two links under the same *gather* link and rename that link to the appropriate type. To support join–links, three functions are defined:

depth(*x*). Returns the depth of link *x* in the link tree.

parent(x). Returns the parent link of link x.

swap(n,$x1$,$x2$). If $x1$ is a child of n, swap the other child of n with $x2$. If $x2$ is a child of n, swap the other child of n with $x1$.

```
1    JOIN–LINKS( target1, target2, type )
2        if depth( target1 ) < depth( target2 )
3            switch–link ← parent( target2 )
4        else
5            switch–link ← parent( target1 )
6        swap( switch–link, target1, target2 )
7        type( switch–link ) ← type
8        return switch–link
```

In lines 2-5 the parent of the link that has the greatest depth is selected (and becomes *switch–link*). This ensures that *switch–link* has only leaves (and no *gather* links) as children. Line 6 switches the "foreign" child of *switch–link* with the other link. After this the type of switch–link can be changed in line 7.

Figure 5.3 gives an example of the construction of a backward tree. It has the same structure as the state transition graph of figure 5.2, but the backward link tree of state F is now a balanced tree with *gather* links. On the left side is the situation where half the algorithm has completed, on the right side is the situation where the algorithm has completed.

At first, the algorithm encounters a *split* link. This link becomes *start–link(1)* (the number between parenthesis is the level of the call to the algorithm). The algorithm recursively calls itself for both branches and encounters an *if* link on the left side and a *direct* link to state F on the right side. These links become *start–link(2a,b)*. On the left side, two more *start–links* are found and the algorithm recurses again. Now the third level calls walk through states $e0$ and $u0$ and both encounter a merging of paths at state F. Both calls assign their final link to *target–link* (3a and 3b respectively) and return to the second level call. The second level call calls join–links to merge both branches (figure 5.3a depicts the situation at this point). The function join–links exchanges the link to $e0$ with the link to S (because $u0$ has a greater depth than $e0$). Now the *gather* link is renamed to an *end–if* link and it becomes *target–link(2a)*. *Target–link(2b)* is the link at the receiving end of *start–link(2b)*. The second level calls return their *target–links* and the first level call can rename *target–link(1)* to a *join* link. Now the same backward link tree as in figure 5.2 is formed (note that both *end–if* and *join* links are symmetrical).

5.4 control logic generation

The control logic will be compiled into a structural VHDL description. The ROM–core is available as a separate component (see chapter 6). The logic function of all inputs of the latches can easily be derived from the backward link tree. All *direct* links are replaced by the output of the previous state. All *from–if* links are replaced by the output

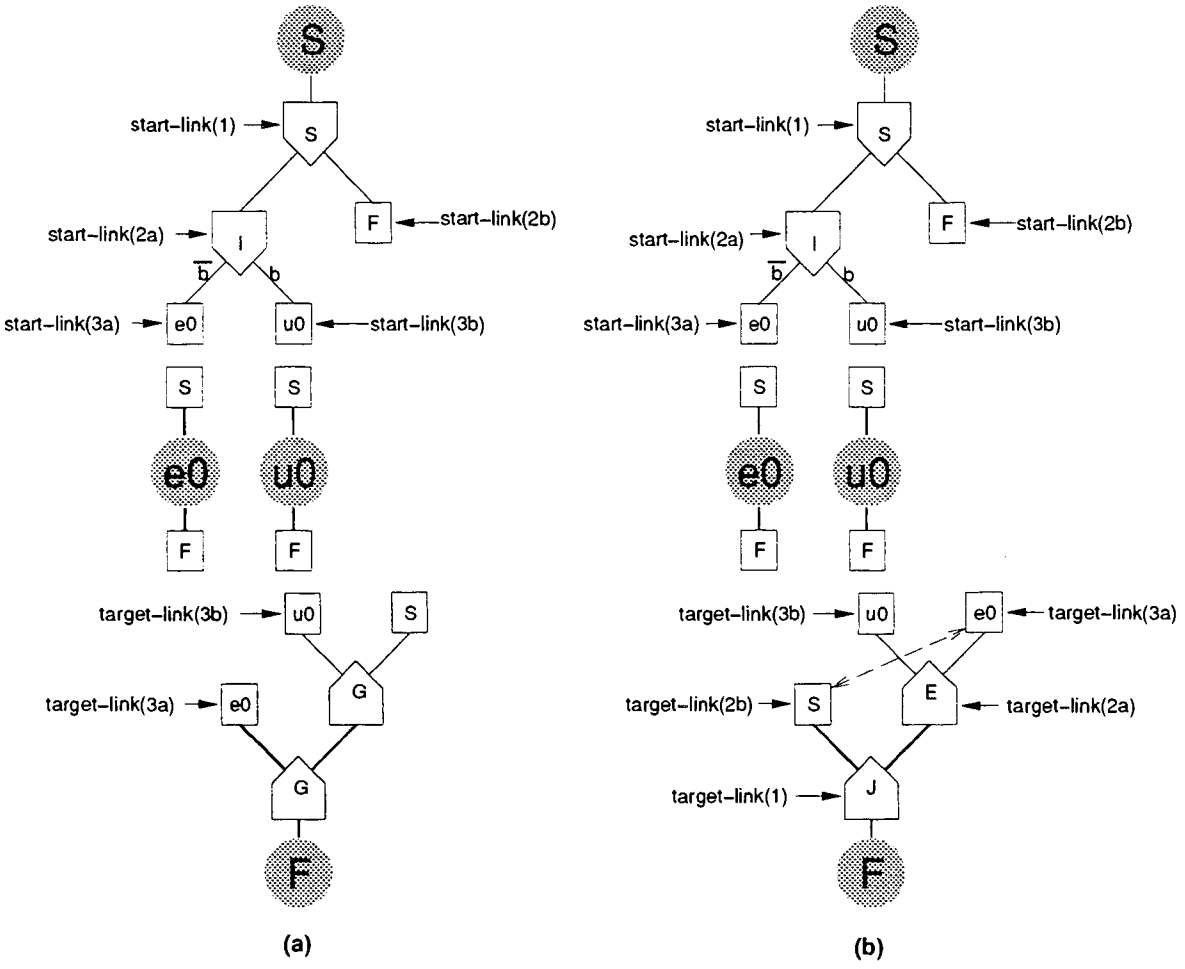

Figure 5.3: Phases in the construction of a backward tree. (a) halfway the algorithm. (b) is after completion of the algorithm.

of the previous state ANDed by the conditional. *End-if* links are replaced by an OR of both reference links. *Join* links are replaced by the output of a wait logic block. This component, called JOIN, is availiable as a VHDL component. This component is driven by the logic functions of both reference links of the *join* link.

Appendix B gives an example of a VHDL–description generated for a small (eight states) finite state machine. It does contain some optimizations which are described in chapter 7.4.

Chapter 6

The ROM–core

The ROM–core serves two functions: it contains the latches that hold the current state(s) and it computes the output vector that belongs to the current state(s). The ROM–core is a layout that can be generated by the ROM–core generator.

The layout is designed and tested in the COMPASS design environment. It is designed in 1.2 micron CMOS technology (λ is 0.6μm).

The ROM–core contains the following five parts (see figure 6.1):

1 the ROM-matrix

2 the latches

3 the output-buffers

4 the precharger

5 the clock–regulation

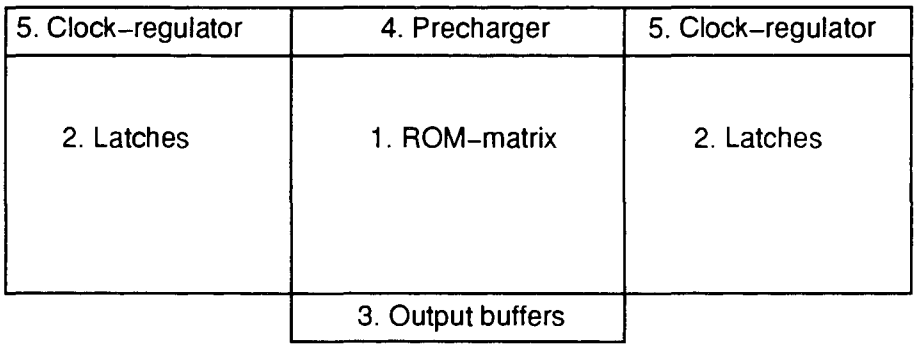

Figure 6.1: General layout of the ROM–core.

Although the ROM–core is a CMOS circuit, the ROM–matrix itself consist only of NMOS transistors. This requires a load device, but CMOS technology does not have suitable

load devices apart from complementary PMOS transistors, which are not suitable for a matrix (see [Dill88], page 410–422). Other load devices do not yield optimal results and use a lot of power. A dynamic circuit solves these problems. It is easy to implement and only uses power when an output switches. The disadvantages are that the outputs are not available during an entire clock cycle and that there is a minimum clock speed.

During the precharge phase the output lines are charged. This charge is held by parasitic capacitances. During the output phase some output lines are discharged and the output becomes valid.

The latches are stacked on both sides of the ROM–matrix, so the latches can be twice as high as a ROM–matrix row (the left and right latches service alternating ROM–matrix rows).

6.1 the ROM–matrix

The ROM–matrix is a (partially filled) array of NMOS-transistors. The VSS–powerlines and the output lines run vertically in metal. The input lines run horizontally in poly. Four transistors share a contact from VSS to the N-diffusion (see figure A.1, appendix A). No space is reserved for bulk contacts, unless there are two zero–entries in the matrix, in which case there is space for a bulk contact (see figure A.2, appendix A).

During the precharge phase all output lines are charged to VDD. At that time all input lines are low. As soon as the precharge phase is over, one or more input-lines will go high. Wherever these lines cross N–diffusion a path from VSS to the output–line will form, so the output–line will discharge. The other output–lines will keep their charge.

If two active input–lines cause the same output–line to discharge, the discharging will only speed up. This means that the ROM itself does not require states that, when simultaneously active, have non-overlapping outputs, although usually the resources driven by the output–lines do.

A 2 by 2 transistor matrix is 19 λ wide and 25 λ high.

6.2 the latches

The latches are of the Master–Slave D-flipflop variety. Two designs were made: a small dynamic flipflop and a static flipflop. The dynamic flipflop is a lot smaller than the static flipflop, but it requires two non–overlapping clocks.

Both flipflops have two outputs: the standard Q-output and a version of the Q-output that is forced low during the precharge phase. The former output can be used for the state control logic and the latter is used to drive the poly–lines of the ROM–matrix. Both flipflops have a reset-input, that causes the output to be low. Although this does not cause the flipflop to reset internally, the control logic guarantees that all flipflops will clock in a zero–value during the next clock cycle.

The dynamic flipflop (see also [West85], page 204) consists of two transmission gates, two invertors and two NOR-gates (see figure 6.2). The flipflop uses two non-overlapping clocks

(clk1 and clk2). During clk1 the left transmission gate passes the D-input to the leftmost invertor. While clk1 is low the invertor keeps its value because of parasitic capacitances. As soon as clk2 becomes high, the value is passed on to the leftmost NOR-gate (that acts as an invertor, as long as the reset-input is not triggered). The other invertor and NOR-gate pass the value to the poly-lines.

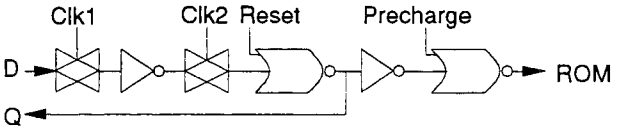

Figure 6.2: Schematic of the dynamic latch.

It is important that the two clocks do not overlap. If they do, both transmission-gates pass their values simultaneously. The effects can range from skipping a state (because the output of the latch is passed directly into the next latch) to a complete randomization of the active states. SPICE-simulations show that some overlap is allowed (because of the slow reaction-times of the circuits), but using an inverted version of clk1 as clk2 does not work.

The layout of the dynamic flipflop is in figure A.3, appendix A. The layout is designed so that multiple flipflops can be stacked vertically. The flipflops share power-lines, so stacked versions must be mirrored horizontally. Control signals run vertically in metal2. The Q-output must be routed back to the beginning of the circuit and in order to do so has to make many detours. This is unavoidable, because the design has to be small and all layers are used. The design is 29 λ high, which is 4 λ higher than two rows of the ROM-matrix, so the rows are padded to account for the difference in height. The last output gate has wide transistors, to drive the long poly-lines in the ROM-matrix.

The static flipflop consists of four NOR-gates for the master, four NOR-gates for the slave and a NOR-gate to stop the output during the precharge. The schematic is given in figure 6.3 (the design of a static Master-Slave D-flipflop is standard and can be found in [Burg88]).

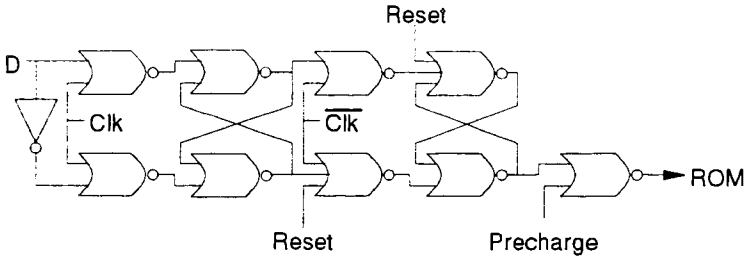

Figure 6.3: Schematic of the static latch.

The layout for this flipflop is given in figure A.4, appendix A. The power-lines run horizontally in metal2 (and are shared by the neighbouring flipflops if they are stacked). The

metal–layer runs mostly vertically, while the poly– and metal2–layers run horizontally. Because of this the metal2 layer can be used to transport signals over the entire length of the flipflop. The design is 50 λ high, so two flipflops side by side must service two ROM–matrix–rows (on the other side of the ROM–matrix two more flipflops service two ROM–matrix–rows, so four ROM–matrix–rows, with a total height of 50 λ, are serviced by four flipflops). The metal2–layer can transport the D–input and Q–output from the innermost flipflop to the outside and the output–line from the outermost flipflop to the ROM–matrix.

The dynamic flipflop is 211 λ wide and the static flipflop is 274 λ wide. This means that two dynamic flipflops are about as wide as 22 ROM–columns and four static flipflops are about as wide as 58 ROM–columns. Two static flipflops are 160% wider than one dynamic flipflop, while two dynamic flipflops are 16% higher than one static flipflop. The extra width of the design with static flipflops will only be offset by its advantage in height in extremely large and impractical designs.

6.3 the output–buffers

The output–buffers are simple invertors. Because of the invertors the presence of a transistor in the ROM–matrix causes a ONE–output, while the absence of a transistor causes a ZERO–output. Because most states have less ONE–outputs than ZERO–outputs this reduces the number of transistors needed in the ROM–matrix (in a normal design the total output word of a controller has less than 50% ONE–outputs, so each state in a parallel configuration has significantly less than 50% ONE–outputs).

The layout of three buffers is given in figure A.5, appendix A. The buffers are packed in groups of three because one buffer is broader than one ROM–matrix column, but three buffers packed together are as broad as three ROM–matrix columns. The output–buffers are wide transistors (the W/L ratio of the PMOS transistor is 12 and the W/L ratio of the NMOS transistor is 9), because they may drive a heavy load.

6.4 the precharger

The precharger is an array of PMOS transistors. The layout of the precharger is given in figure A.6, appendix A. The poly–line that controls the precharger comes from the clock–regulation.

6.5 the clock–regulation

The clock–signals have to drive a lot of transistors. Because of this the clock signal is buffered (and inverted) on arrival. This is done by several invertors in parallel. A clock cycle starts at the down flank of the clock. Similarly, the dual clock design starts at the down flank of clk1 and the two clocks may not be simultaneously *low* (this is the result of the inversion of the clock on entry).

One of the invertors feeds the invertors that are used to derive a non–inverted clock. The others drive signal–lines. If possible, two invertors in parallel drive the same signal–line. All invertors are made of wide transistors. The drive capacity of the invertors is necessary because each signal–line has up to 0.45pF capacitance (a normal gate has 0.01–0.02pF capacitance)

In figure A.7, appendix A, the layout of the clock–regulation is given. There is one version for the dynamic flipflop and a version for the static flipflop.

6.6 total size

Because the matrix–transistors are grouped in 2 by 2 groups, both the number of rows and the number of columns must be a multiple of two. The size of the design can be calculated as follows (R is the number of rows, C is the number of columns):

$$width = x \times W_{flipflop} + C \times W_{transistors} + W_{overhead}$$

$$height = R \times H_{transistors} + H_{clock} + H_{buffer}$$

in which:

- x is the number of flipflops that are used horizontally, 2 for the dynamic flipflops and 4 for the static flipflops.
- $W_{flipflop}$ is the width of the flipflop, 211 λ for the dynamic flipflop and 274 λ for the static flipflop.
- $W_{transistors}$ is the (average) width of the transistors, 10.5 λ.
- $W_{overhead}$ is some extra space needed for interconnect, 52 λ.
- $H_{transistors}$ is the (average) height of the transistors, 12.5 λ for the design with dynamic flipflops and 14.5 λ for the design with static flipflops.
- H_{clock} is the height of the clock–regulation, 71 λ (the precharger is just 39 λ high).
- H_{buffer} is the height of the output–buffer, 189 λ.

The total size of the design with dynamic flipflops is:

$$width = 474 + 10.5 \times C$$

$$heigth = 260 + 14.5 \times R$$

The total size of the design with static flipflops is:

$$width = 1148 + 10.5 \times C$$

$$heigth = 260 + 12.5 \times R$$

In chapter 7.4 extra buffers will be introduced for extra speed in large designs. Those buffers will require an extra 272 λ in height per buffer.

Chapter 7

Results

7.1 Speed

One of the design goals of the Multi–thread ROM–controller was an operating speed of 25Mhz. In 40ns the controller should be able to advance to the next state(s) and supply the outputs. The entire controller could not be simulated by a circuit simulator, because no circuit extractor for the VHDL controller specification was available. The ROM core, however, could be simulated, so various controller configurations were simulated by the Compass Spice circuit simulator.

Preliminary tests showed that the optimum size of the ROM core would be around 100×52 (100 outputs and 52 states). First the speed of the ROM core itself was tested at various sizes (50×28, 100×28, 50×52 and 100×52). After that the performance of the 100×52 ROM core was tested, while there was an output load of 0.4pF (relatively light load), 1pF (medium load) and 3pF (cross–chip wires). The last simulation was of a large core (200×100) with a light load (0.4pF). A ROM core of this size with a heavier load would be too slow, so extra output buffers (a light load) should be used.

The Spice results are plotted in appendix D. Both the ROM core version with static flipflops and the ROM core version with dynamic flipflops have been calculated. In the figures the clock, the output of a flipflop and the output of a ROM column are plotted. The clock is the inverse of the clock that is supplied to the ROM core (the cycle starts at the up flank). The start of each clock cycle is printed on the X-axis.

In the first cycle, the circuit is reset by a reset pulse. The input of the flipflop is high until halfway between cycle 3 and 4, so the output of the flipflop will rise in the first half of cycle 2. The output of the flipflop stays high in cycle 3 and falls in cycle 4. 10ns before cycle 5 the input of the flipflop rises again, so the output of the flipflop rises in cycle 5. The ROM–output rises in the second half of each cycle in which the output of the flipflop is high (cycle 2, 3 and 5). It falls at the beginning of each new cycle (the precharge phase). The inputs are plotted in figure D.1.

In tables 7.1 and 7.2 the rise and fall times of the outputs are given. The rise and fall times of the outputs of the flipflop are given from the start of each cycle and the rise and fall times of the ROM–outputs are given from halfway each cycle. In figure 7.1 the

unloaded designs and the designs loaded with .4pF are compared to each other. The time on the Y-axis is the flipflop rise time added to the ROM–output rise time. In figure 7.2 the same is done for the loaded 100×52 designs.

Table 7.1: Rise and fall times of the ROM core with static flipflops. All times are from the 50% level of the clock input to the 70% level of the output for rise times and the 30% level for fall times.

size	load	flipflop		ROM	
	[pF]	rise [ns]	fall [ns]	rise [ns]	fall [ns]
50×28	0	13	8	10	6
100×28	0	13	8	10	8
50×52	0	14	8	12	8
100×52	0	14	8	12	12
100×52	0.4	14	8	16	12
100×52	1	14	8	18	16
100×52	3	14	8	21	29
200×100	0.4	14	10	22	16

Table 7.2: Rise and fall times of the ROM core with dynamic flipflops. All times are from the 50% level of the clock input to the 70% level of the output for rise times and the 30% level for fall times.

size	load	flipflop		ROM	
	[pF]	rise [ns]	fall [ns]	rise [ns]	fall [ns]
50×28	0	6	6	6	6
100×28	0	6	6	8	8
50×52	0	6	6	8	8
100×52	0	6	6	10	10
100×52	0.4	6	6	12	12
100×52	1	6	6	14	16
100×52	3	6	6	17	30
200×100	0.4	6	8	18	18

The tables show that the design with dynamic latches is the fastest design. Even the 100×52 design with heavy load and the 200×100 design meet the design criterium. All designs have a relatively slow ROM–output fall time. This fall time is not really important, since the ROM–output starts falling at the beginning of a clock cycle and is read at the beginning of the next clock cycle, so it has twice as much time to fall as it has to rise.

One important aspect of both designs is that the rise time of the output of the flipflop is faster than the rise time of the ROM–output. By devoting more time to the ROM–output,

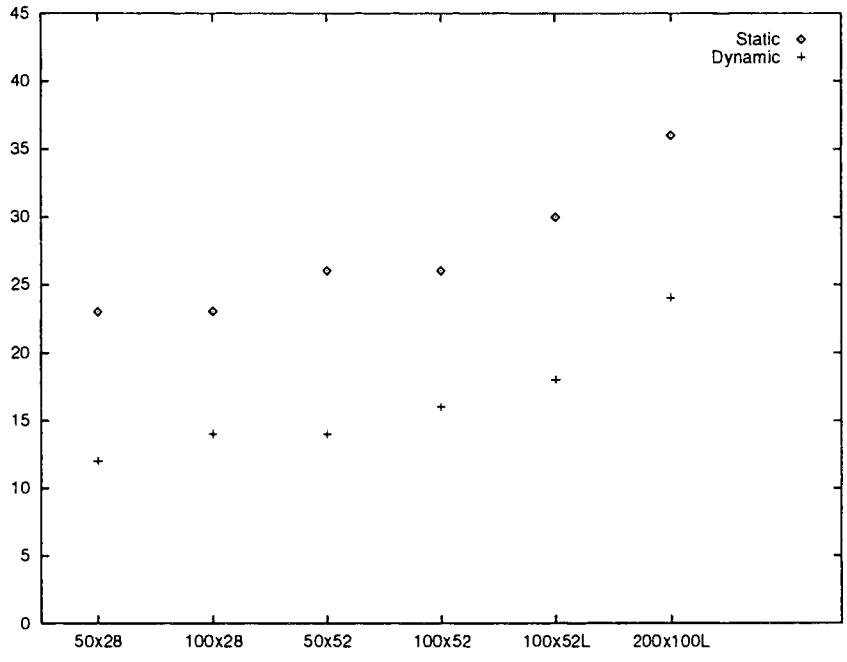

Figure 7.1: Speed comparison between the unloaded designs and the designs loaded with 0.4pF.

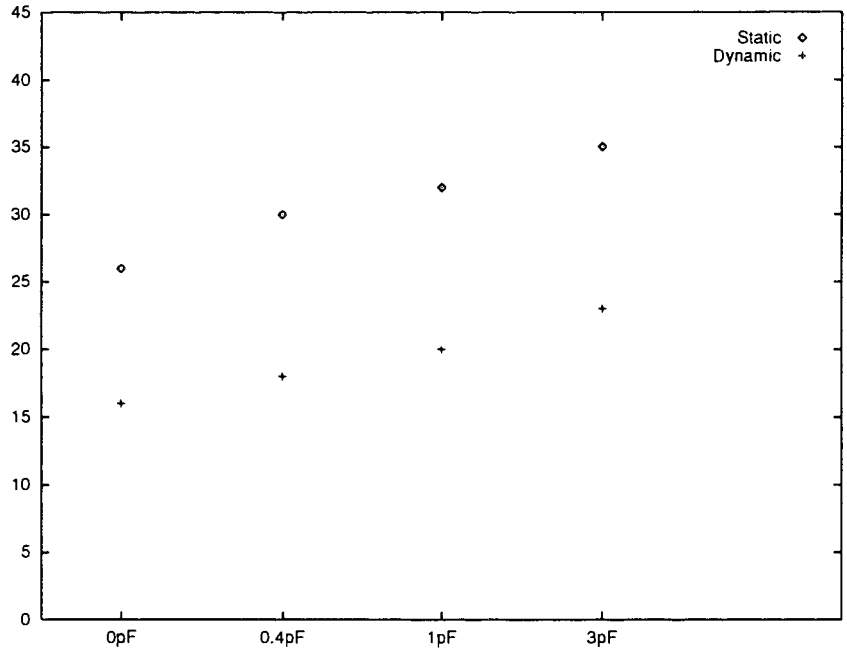

Figure 7.2: Speed comparison between the various 100×52 designs.

a faster design can be realised. This way the two configurations with static flipflops that do not meet the design criterium (100×52 with 3pF and 200×100 with 0.4pF) can be made to work.

Datasheets for the logic gates used by the Compass VHDL–compiler show that any single basic logic function (AND, OR) can be calculated in 2ns. The wait logic for explicitly synchronized sequences takes less than 6ns. There are 21ns between the time that the output of the flipflop becomes available (in the worst case 14ns after the beginning of the clock cycle) and the time that the input of the flipflop must be valid (5ns before the beginning of the next clock cycle, tested with additional Spice–runs). Even three serial wait logic blocks can produce a result in this time (three serial wait logic blocks are the result of a 8-way concurrent sequence). Usually the state control logic does not become any more complicated than one or two wait logic blocks and two basic logic functions.

As a comparison, a standard Compass ROM delivers its output in 10 to 20ns (depending on the size of the ROM), including setup times. The Multi–thread ROM is a bit slower (12 to 36ns, depending on the design and output load), but this includes state transitions. If the state transition for a standard ROM–controller takes 10ns, a standard ROM–controller can accommodate designs at 33Mhz.

The minimum clock speed at which the Multi–thread ROM–controller can function is determined by the dynamic parts, the output lines and the dynamic flipflops. The output lines have a capacitance of 0.03pF (2 rows) to 0.20pF (100 rows). This leads to an estimation (see [Dill88], page 567) of the minimum clock speed of 120Hz. The storage nodes in the dynamic flipflop have a capacitance of 0.05pF, which leads to an estimation of the minimum clock speed of 40Hz.

7.2 Size

The ROM core is basically a standard ROM core with flipflops instead of address decoders. In table 7.3 the size of a Multi–thread ROM core is compared to a standard Compass ROM core. Because Compass ROMs have been designed to have many rows and few columns, the comparison is made on basis of the number of ROM–cells (the smallest design with the given number of cells).

The table shows that the design with static flipflops is up to 25% larger than a standard ROM core. A design with dynamic flipflops is up to 45% smaller, but at moderate sizes it is about as large as a standard ROM core. Only at extreme sizes is it larger than a standard ROM core.

Table 7.4 gives an impression of the overall size of the Multi–thread ROM–controller. For several examples the sizes of the resulting controller are given. The sizes are given for the ROM core, the state control logic and the total design. For the random logic implementation, the total design size is an approximation, because the state control register is not synthesized.

The following examples are used:

Example 1 is a moderately sized multi–thread finite state machine without loops (see

Table 7.3: ROM core areas and aspect ratios. The ratio column after the Multi–thread ROM core size compares it to the Compass ROM core size.

#cells	Compass		static			dynamic		
	Area [Mλ^2]	Asp. [W/H]	Area [Mλ^2]	Asp. [W/H]	Ratio	Area [Mλ^2]	Asp. [W/H]	Ratio
256	.56	1.0	.55	3.90	98%	.31	1.95	55%
512	.62	.98	.68	3.15	110%	.40	1.52	65%
1024	.76	.87	.82	3.79	108%	.55	2.11	72%
2048	1.02	.86	1.15	2.72	113%	.81	1.45	79%
4096	1.33	.91	1.54	3.65	115%	1.26	2.27	95%
8192	1.98	.74	2.41	2.33	122%	2.04	1.40	103%
16384	2.93	.95	3.66	3.54	125%	3.52	2.41	119%

figure E.1). It has 28 states and 15 outputs.

Example 2 is the flat version of example 1 (because there are no loops, no state explosion occurs) and is given in figure E.2. This example has 30 states and 15 outputs.

Example 3 is a small finite state machine with parallel loops (see figure E.3). It has 23 states and 16 outputs.

Example 4 is a large multi–thread finite state machine with 127 states and 137 outputs. The state machine is given in figure E.4.

Example 5 is the same state machine as example 4, but is contains only 70 outputs. This gives an impression of the overhead caused by the partition of a controller in two separate controllers (see chapter 7.4).

Example 6 is the state machine of example 2 with two extra 10–state sequences inserted, resulting in a state machine with 50 states and 11 outputs. This example can be used to compare the Multi–thread ROM–controller to a random logic implementation.

Example 7 is the state machine of example 6 with 33 outputs.

The table shows that an implementation of example 7 (50 states, 33 outputs) with a Multi–thread ROM–controller with dynamic flipflops is smaller than a random logic implementation. In [Gerb92] it is stated that a ROM–controller of about 100 states will be as small as a random logic implementation. One of the clear advantages of the Multi–thread ROM–controller is the amount of non–ROM area (state control logic and interconnect). In [Gerb92] that area is an average 93% of the ROM–area, while for the Multi–thread ROM–controller that area is an average 65% and when considering only large controllers containing "simple sequences" (example 5 and 6), that area is only 45%. This shows that the Multi–thread ROM–controller is also a good alternative for standard ROM–controllers.

Table 7.4: Overall sizes of the ROM controller. The sizes are given for a random logic design (if possible), a static flipflop Multi–thread ROM–controller and a dynamic flipflop Multi–thread ROM–controller. For example 5 and 6 the sizes have been adjusted to negate unnecessary overhead that the chipcompiler generated.

name	design	ROM core [Mλ^2]	state logic [Mλ^2]	overall	
				Area [Mλ^2]	Aspect [W/H]
example 1	random	-	.36	.43	.88
example 1	static	0.83	.06	1.37	1.54
example 1	dynamic	.42	.09	.78	1.05
example 2	static	.79	.05	1.28	1.59
example 2	dynamic	.43	.07	.74	1.06
example 3	static	.73	.09	1.29	1.64
example 3	dynamic	.39	.11	.88	1.48
example 4	static	1.93 (3×)	.09	7.70	.91
example 4	dynamic	1.58 (3×)	.10	6.70	.59
example 5	static	1.53 (3×)	.11	6.25	.73
example 5	dynamic	1.09 (3×)	.10	4.86	.44
example 6	random	-	.64	.73	1.08
example 6	static	1.07	.06	1.70	1.26
example 6	dynamic	.56	.11	1.02	.81
example 7	random	-	1.49	1.61	1.18
example 7	static	1.24	.06	1.92	1.42
example 7	dynamic	.74	.09	1.24	.98

7.3 Power consumption

The last performance criterium is the power consumption. The power consumption of both Multi–thread ROM cores is given in tables 7.5 and 7.6. There is one column for normal "single thread" usage (only one state is active) and one column for heavy usage (five states are active). For normal usage, 25% of the outputs are active (a high percentage for a single state) and for heavy usage all outputs are active. Therefore the power consumption can be considered an upper bound estimation for the given designs.

The tables show that the dynamic flipflop designs use less power if they are unloaded, but they use more power if they must drive a load. This is caused by the fact that the dynamic design is somewhat faster than the static design. Because of this it uses more power when the outputs need to be driven. When the outputs are not loaded, little power is drawn and the advantage of the dynamic design (less transistors) shows.

A standard Compass ROM–cell uses about 8mW per output, if the outputs are loaded with 3pF. It is designed for 16 outputs average, so it uses about 130mW. The Multi–thread ROM core uses less power when it is loaded with 3pF.

Table 7.5: Power usage of the ROM core with static flipflops. The light usage column indicates power usage when one state is active and 25% of the outputs is active. The heavy usage column indicates power usage when five states and all outputs are active.

size [#rows×# cols]	load [pF]	light usage [mW]	heavy usage [mW]
50×28	0	6.69	10.35
100×28	0	8.47	15.79
50×52	0	11.13	16.30
100×52	0	14.40	20.57
100×52	0.4	17.11	37.76
100×52	1	21.32	56.09
100×52	3	33.99	106.27
200×100	0.4	34.35	50.00

7.4 Further improvements

Since a lot of state machines use more than 100 states and some may use more than 200 outputs, the Multi–thread ROM–controller may have to be divided into several smaller parts. One solution is to combine the outputs of several separate parts with OR–gates, but this presents a wiring problem, resulting in an excessive use of area (about 6 times the area of the ROM core in a 128–output design).

It is easier to insert a buffer in the output lines. This buffer separates the ROM core above and below the buffer. The buffer can be effected by using the output of a ROM core as an input for the output line of the second ROM core. The NMOS transistor that processes the output of the first ROM core can be put just after the output buffers of first ROM core. The drain of that transistor can be directly connected to the output line of the second ROM core (resulting in the layout of figure A.9). Because the NMOS transistor is wide, it can easily drive the output line. Spice simulations show that the extra delay incurred by the buffer is about 1ns.

To reduce the number of outputs, one has to split the ROM core in parts with half the number of outputs each. Each part uses the same state control logic, but the total number of latches has doubled. The total amount of wiring will increase, but not by much (few extra wires are needed). Because of the latches and some overhead, the new design will be about 60% larger with static flipflops and about 45% larger with dynamic flipflops.

Another improvement is that the chaining of latches to the next state is taken care of in the ROM–core, eliminating any overhead the chipcompiler could add. Only routing to multiple latches, to the state control logic and to different parts of the ROM–core is left to the chipcompiler.

Table 7.6: Power consumption of the ROM core with dynamic flipflops. The light usage column indicates power usage when one state is active and 25% of the outputs is active. The heavy usage column indicates power usage when five states and all outputs are active.

size [#rows×#cols]	load [pF]	light usage [mW]	heavy usage [mW]
50×28	0	5.59	9.15
100×28	0	8.28	14.87
50×52	0	9.47	13.84
100×52	0	14.90	21.42
100×52	0.4	21.97	45.05
100×52	1	26.75	67.98
100×52	3	39.23	116.83
200×100	0.4	44.08	58.97

Chapter 8

Conclusions

The Multi–thread ROM–controller offers an elegant solution for the implementation of multi–thread algorithms. It circumvents the problem of state–explosion that single–thread controller implementations have and it doesn't require the overhead that communicating single–thread controllers require.

Dynamic flipflops with transmission gates and two non-overlapping clocks are best suited as latches in the Multi–thread ROM–controller. They deliver the best performance (in size and speed) at the cost of little extra power and some extra logic to generate the two clocks.

The Multi–thread ROM–controller also offers small solutions for single–thread algorithms. At about 50 states and 33 outputs, the Multi–thread ROM–controller becomes as efficient as a random logic controller, while conventional ROM–controllers reach the same efficiency as random logic controllers at about 100 states. The small size comes from an efficient ROM–core coupled with very little state control logic (most state transitions require just a simple connection to the next state).

One of the problems of the Multi–thread ROM–controller is that its capacity is somewhat limited. If the ROM–core contains over 100 states, it becomes increasingly slow. Fortunately this problem can be overcome (at the cost of a 1ns delay and some extra area) by partitioning the controller and using the output of the first partition as an input for the second partition. The number of outputs is also limited (200 outputs is a maximum). This problem can be overcome by implementing two parallel controllers, each of which computes a different half of the outputs (if they are fed the same inputs, both controllers should run identically).

Bibliography

[Burg88] P. BURGER. *Digital design, a practical course.* John Wiley & sons, Inc, 1988.

[Corm90] T.H. CORMEN, C.E. LEISERSON, AND R.L. RIVEST. *Introduction to algorithms.* MIT Press, 1990.

[Dill88] T.E. DILLINGER. *VLSI engineering.* Prentice Hall, 1988.

[Eijn91] J.T.J. VAN EIJNDHOVEN, G.G. DE JONG, AND L. STOK. The ASCIS Data Flow Graph: Semantics and Textual Format. Eut report 91-e-251, Eindhoven University of Technology, June 1991.

[Gerb92] L. GERBAUX AND G. SAUCIER. Automatic synthesis of large Moore sequencers. *Integration, the VLSI journal*, 13:259–281, September 1992.

[Hild93] H.A. HILDERINK AND J.A.G. JESS. ROM–based Multi Thread Controller. In *IFIP Workshop on Logic and Architecture Synthesis*, pages 231–241, Grenoble, December 1993. Institute National Polytechnique de Grenoble.

[Ins88] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard VHDL Language Reference Manual*, ieee std. 1076-1987 edition, 1988.

[West85] N. WESTE AND K. ESHRAGHIAN. *Principles of CMOS VLSI design, a systems perspecive.* VLSI systems. Addison-Wesley, 1985.

Appendix A

Layouts

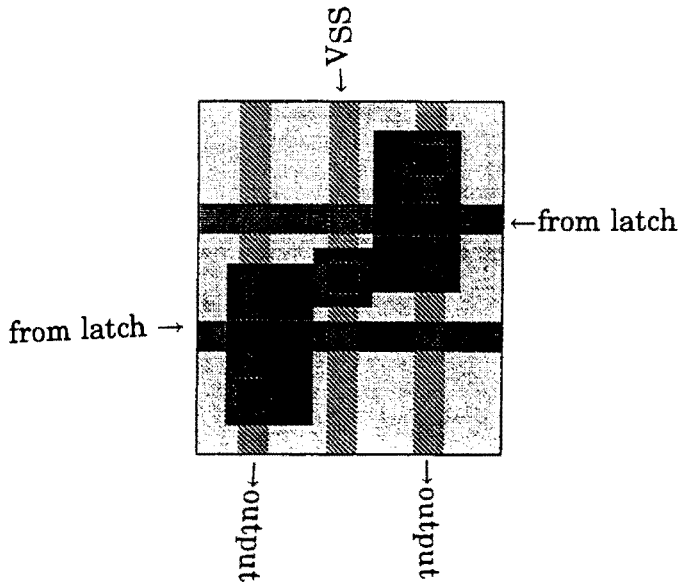

Figure A.1: Layout of the ROM–matrix transistors.

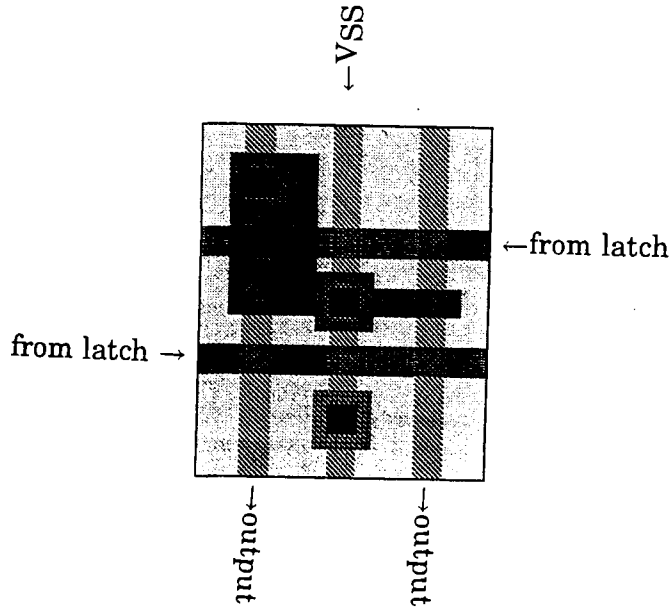

Figure A.2: A bulk contact.

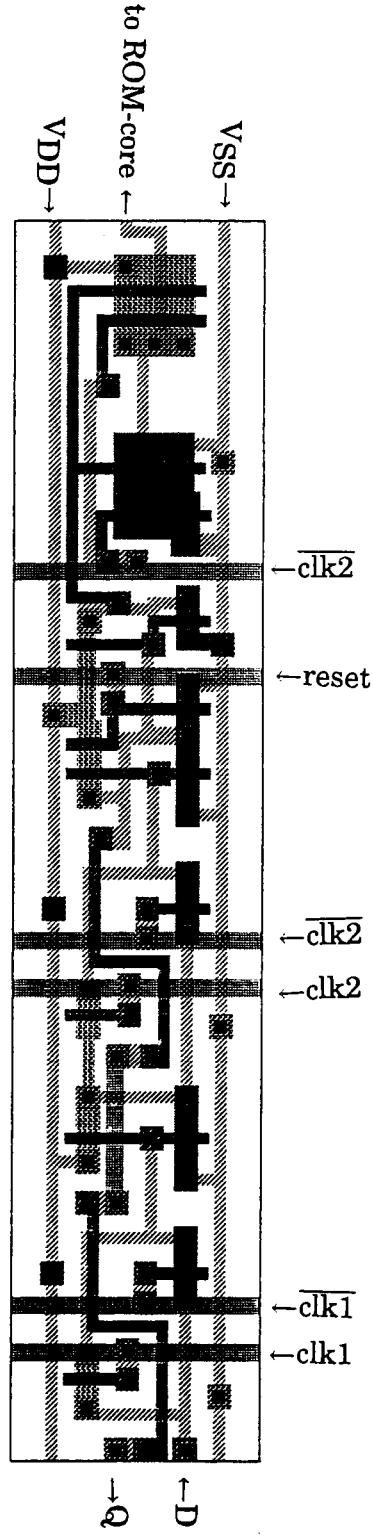

Figure A.3: Layout of the dynamic flipflop.

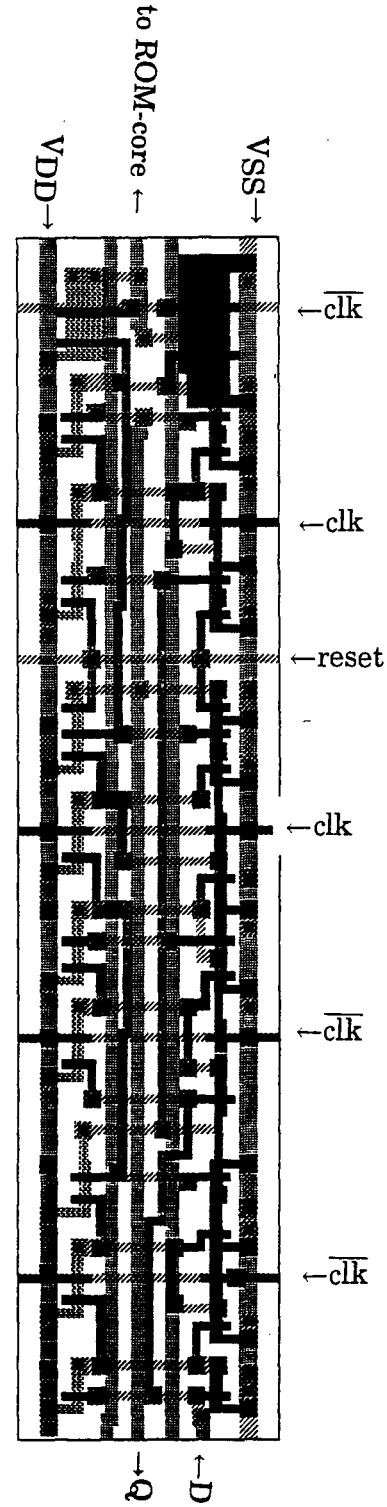

Figure A.4: Layout of the static flipflop.

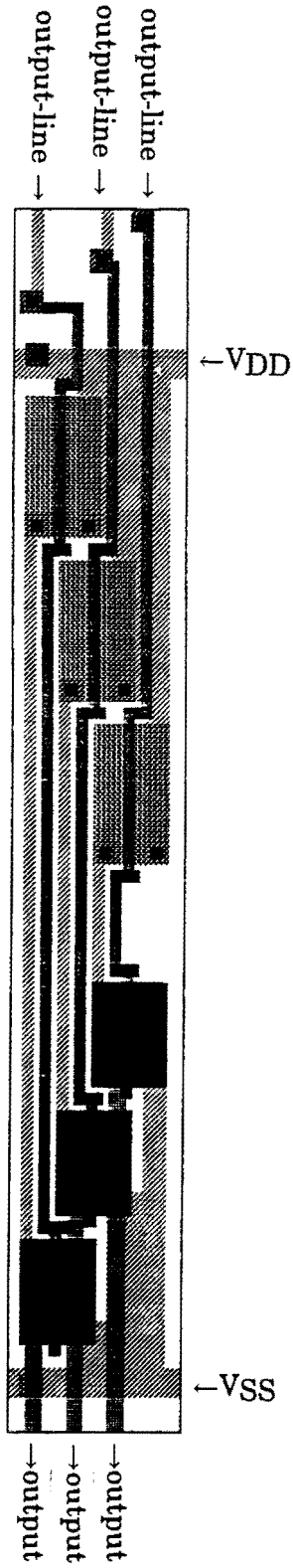

Figure A.5: Layout of the buffers.

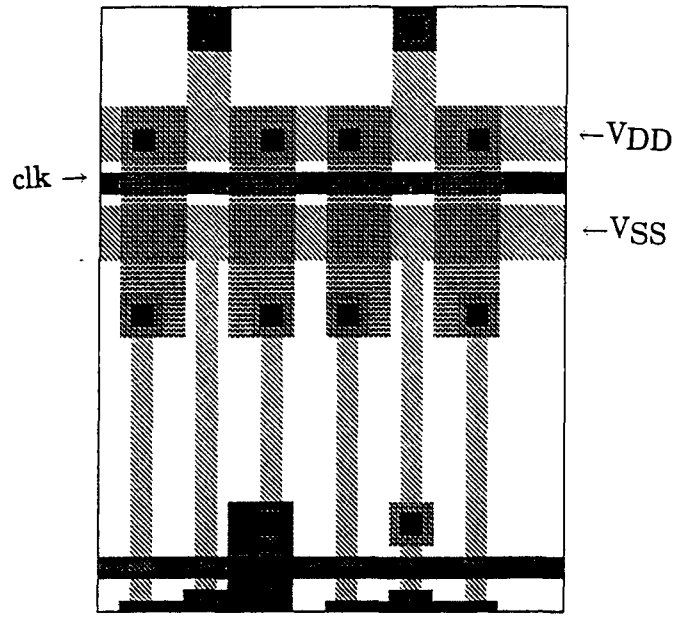

Figure A.6: Layout of the precharger.

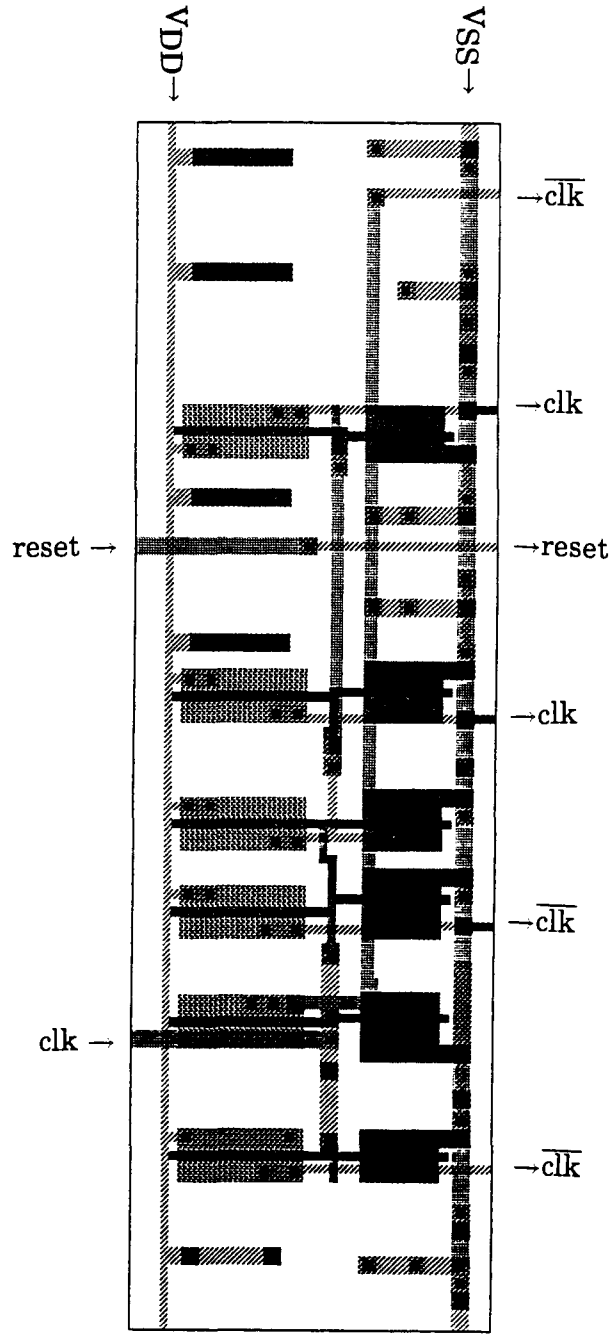

Figure A.7: Layout of the clock–regulation for static flipflops.

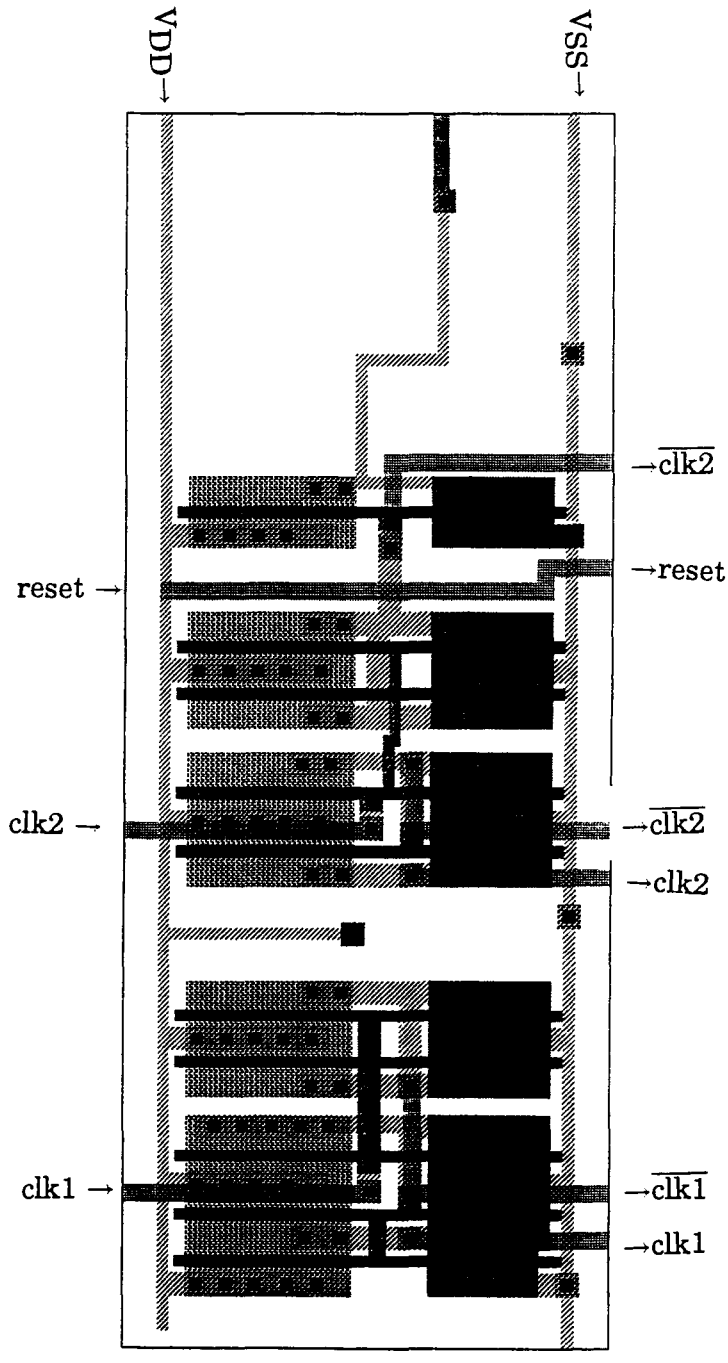

Figure A.8: Layout of the clock–regulation for dynamic flipflops.

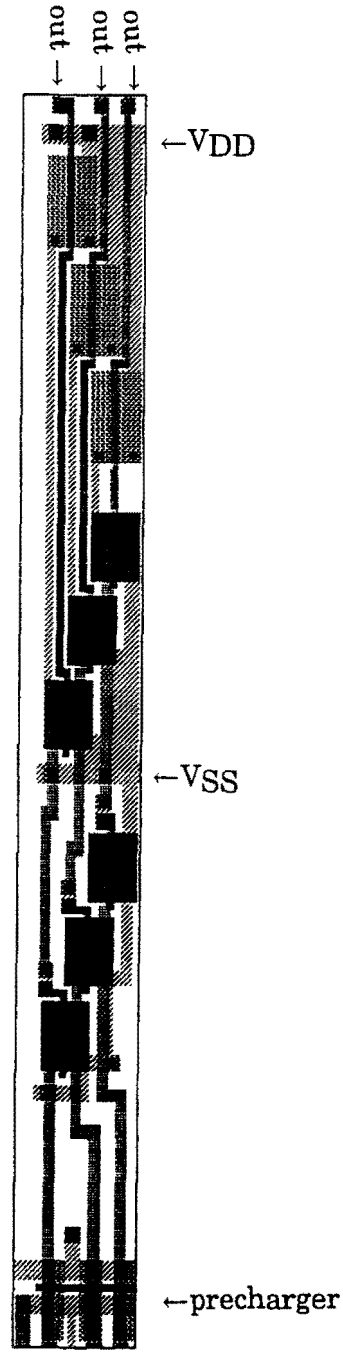

Figure A.9: Layout of the buffer between two ROM cores.

Appendix B

VHDL–description example

Figure B.1 gives an example of a multi–threaded finite state machine that executes two loops in parallel (or one loop if a is true).

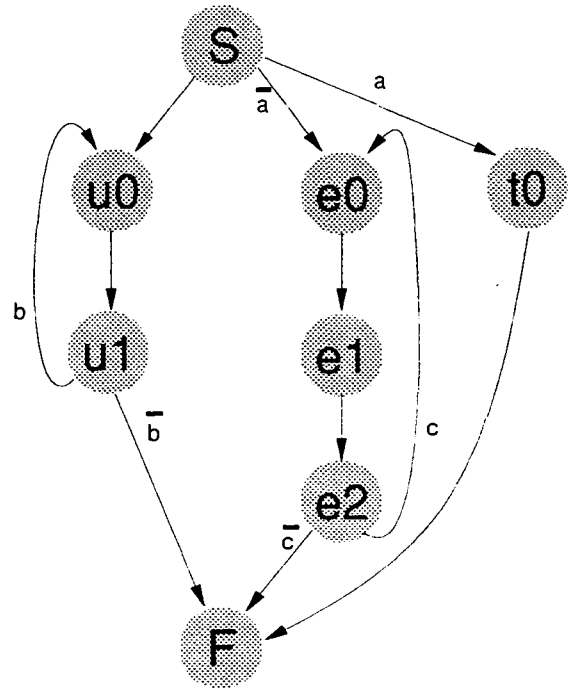

Figure B.1: Example multi–threaded finite state machine.

The following VHDL–description will be generated for this example (at some points extra comments have been added).

```
library ieee;
use ieee.std_logic_1164.all;
library compass_lib;
```

```
use compass_lib.compass.all;

-- The join entity is for explicit synchronisation of concurrent
-- sequences
entity join is
  port (st : in bit; in1 : in bit; in2 : in bit; reset : in bit;
        clk : in bit; outp : out bit);
end join;

architecture join of join is
signal t11, t12, t13, t14, t21, t22, t23, t24 : bit;
-- These components are part of the COMPASS design library and
-- signify an invertor, two NOR-gates and an AND-gate.
component in01d2
  port (i : in bit; zn : out bit);
end component;
component nr02d1
  port (a1 : in bit; a2 : in bit; zn : out bit);
end component;
component nr03d2
  port (a1 : in bit; a2 : in bit; a3 : in bit; zn : out bit);
end component;
component an02d2
  port (a1 : in bit; a2 : in bit; z : out bit);
end component;
begin
  u11 : in01d2 port map (in1, t11);
  u12 : nr02d1 port map (clk, t11, t12);
  u13 : nr02d1 port map (t14, t12, t13);
  u14 : nr03d2 port map (t13, st, reset, t14);
  u21 : in01d2 port map (in2, t21);
  u22 : nr02d1 port map (clk, t21, t22);
  u23 : nr02d1 port map (t24, t22, t23);
  u24 : nr03d2 port map (t23, st, reset, t24);
  u30 : an02d2 port map (t14, t24, outp);
end join;

library ieee;
use ieee.std_logic_1164.all;
library compass_lib;
use compass_lib.compass.all;

-- define the ROM-controller with four outputs. Reset resets all
-- latches, start starts the state machine. The dynamic latches
-- are used, so two clocks (clk and nclk) are required
entity rom is
  port (nclk : in bit; clk : in bit; reset : in bit; a : in bit;
```

```
        b : in bit; c : in bit; start : in bit; o0 : out bit;
        o1 : out bit; o2 : out bit; o3 : out bit);
end rom;

architecture rom of rom is
-- inputs of the latches
signal d0, d3, d4, d6, d7 : bit;
-- signals that carry the outputs of the latches
signal f2, f3, f5, f6, f7 : bit;
-- extra signals for the wait logic
signal e0, e1, e2 : bit;
signal VDD : BIT := '1';
signal VSS : BIT := '0';

-- declare the ROM-core (an external layout)
component rom1
  port (d0 : in bit; d3 : in bit; d4 : in bit; d6 : in bit;
        d7 : in bit; clk1 : in bit; clk2 : in bit; nclk1 : in bit;
        nclk2 : in bit; reset1 : in bit; reset2 : in bit;
        vd : in bit; vs : in bit; q2 : out bit; q3 : out bit;
        q5 : out bit; q6 : out bit; q7 : out bit; o0 : out bit;
        o1 : out bit; o2 : out bit; o3 : out bit);
end component;

component join
  port (st : in bit; in1 : in bit; in2 : in bit; reset : in bit;
        clk : in bit; outp : out bit);
end component;

begin
  d3 <= (start or '0');                    -- state s
  d4 <= ((f3 and a) or (f5 and b));        -- state e0
  d0 <= ((f3 and not a) or (f2 and c));    -- state t0
  -- internal d5 <= f4;                    -- state e1
  -- internal assignments are handled inside the rom1 component
  e1 <= f7;
  e2 <= ((f2 and not c) or (f5 and not b));
  d6 <= e0;   -- state f
  -- internal d1 <= f0;                    -- state t1
  -- internal d2 <= f1;                    -- state t2
  d7 <= f3;                                -- state u0
  u1 : rom1 port map ( d0, d3, d4, d6, d7, clk, clk, nclk, nclk,
                       reset, reset, VDD, VSS, f2, f3, f5, f6, f7,
                       o0, o1, o2, o3 );
  u2 : join port map (f6, e1, e2, reset, clk, e0);
end rom;
```

Appendix C

Description of the finite state machine input file

The description of the finite state machine is a text file. The same format was previously used to describe single thread state machines. For single thread state machine state transitions with the same condition may not occur. For multiple thread state machines this restriction is lifted, but a number of single thread tools will not work with multiple thread input files.

Lines starting with a dot indicate a command. There are five commands:

.inputvars <name>+ Defines the names of the input signals.

.outputvars <name>+ Defines the names of the output signals.

.mv Contains information about the number of input variables, the number of output variables and the number of symbolic states. It has the following parameters:

<total_variables> This is <input_variables> + 3

<input_variables> The number of input signals of the finite state machine

<output_variables> The number of output signals of the finite state machine

<old_states> The number of symbolic old states preceded by a "-" sign

<new_states> The number of symbolic new states preceded by a "-" sign

.p <#transitions> Specifies the number of state transitions in the state transition table.

.end Signifies the end of the file

The other lines define the state transition table. Each line of the state transition table is composed of four entries:

<input_values> In this string the values of the input signals are given for the state transition described by the current line. For each signal (in the order indicated by the ".inputvars" statement) the value is given. The value can be "1" "0" or '-' (don't care).

<old_state> This is the name of the old state for the state transition described by the current line.

<new_state> This is the name of the new state for the state transition described by the current line.

<output_values> In this string the values of the output signals are given for the state transition described by the current line. For each signal (in the order indicated by the ".outputvars" statement) the value is given. The value can be "1" "0" or '-' (don't care).

The state machine in appendix B can be described by the following input file:

```
.inputvars a b c
.outputvars o0 o1 o2 o3
.mv 0 3 -8 -8 4
.p 11
--- s u0  0000
0-- s e0  0000
1-- s t0  0000
--- u0 u1 0000
-0- u1 f  0000
-1- u1 u0 0000
--- e0 e1 0000
--- e1 e2 0000
--0 e2 f  0000
--1 e2 e0 0000
--- t0 f  0000
.end
```

Appendix D

Spice results

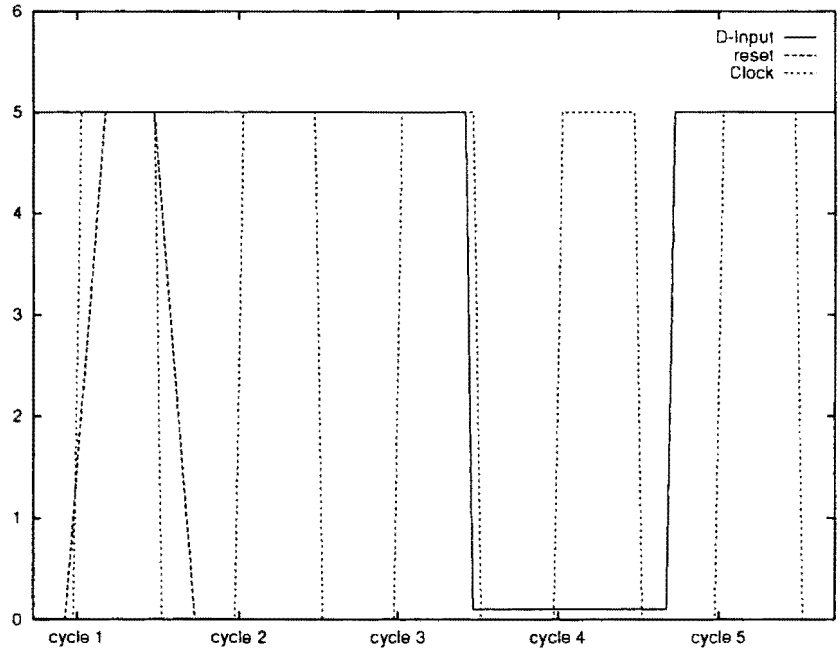

Figure D.1: Input signals for SPICE simulations.

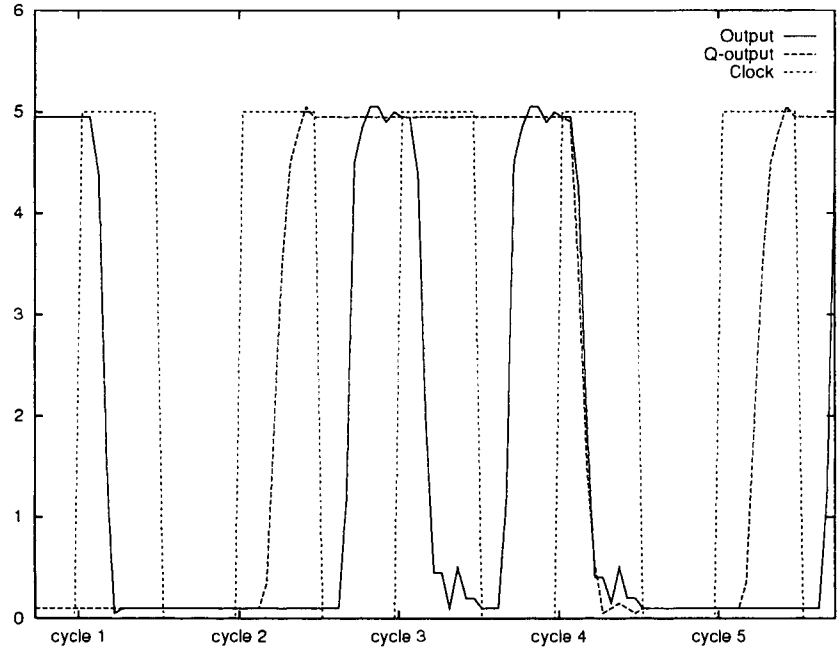

Figure D.2: SPICE results for a 50x26 ROM with static flipflips.

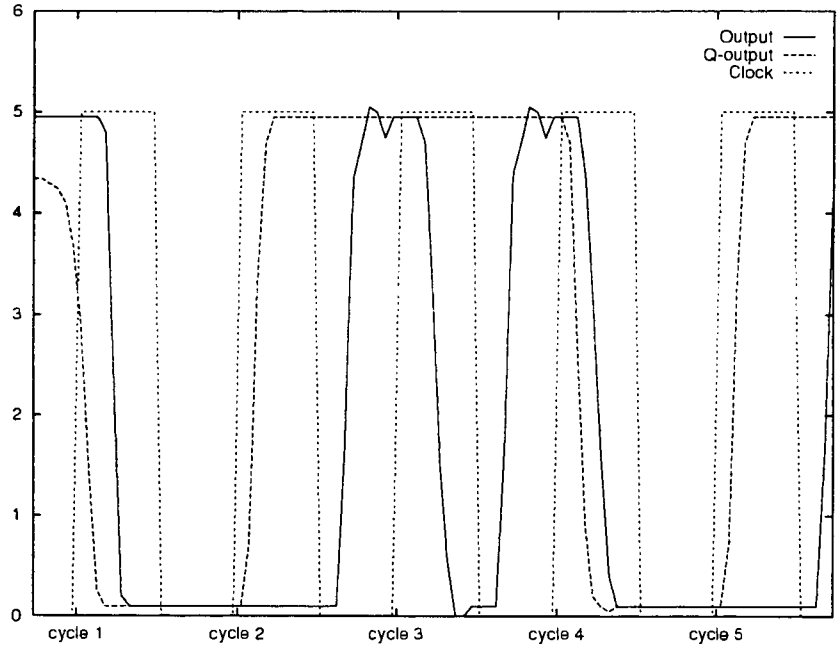

Figure D.3: SPICE results for a 50x26 ROM with dynamic flipflips.

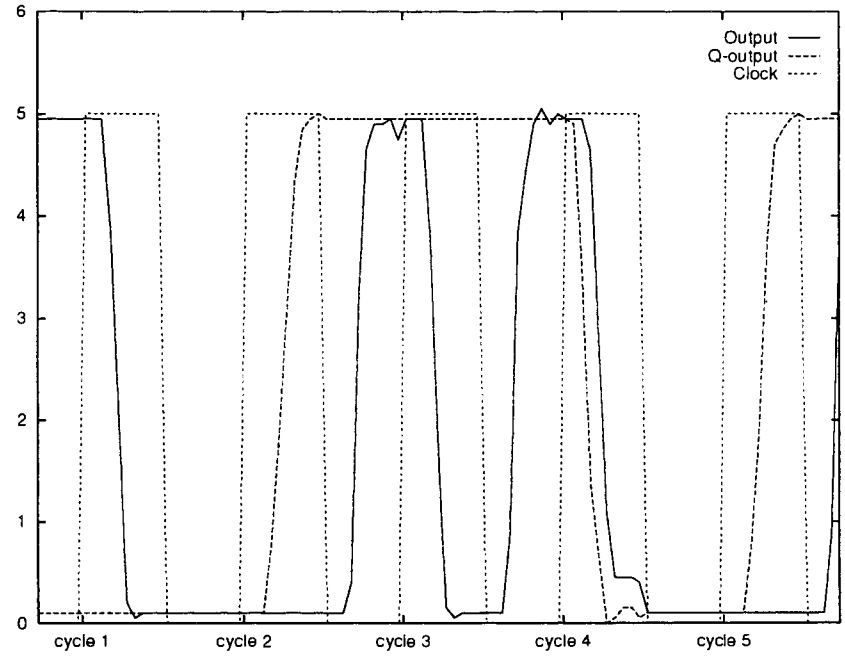

Figure D.4: SPICE results for a 50x50 ROM with static flipflips.

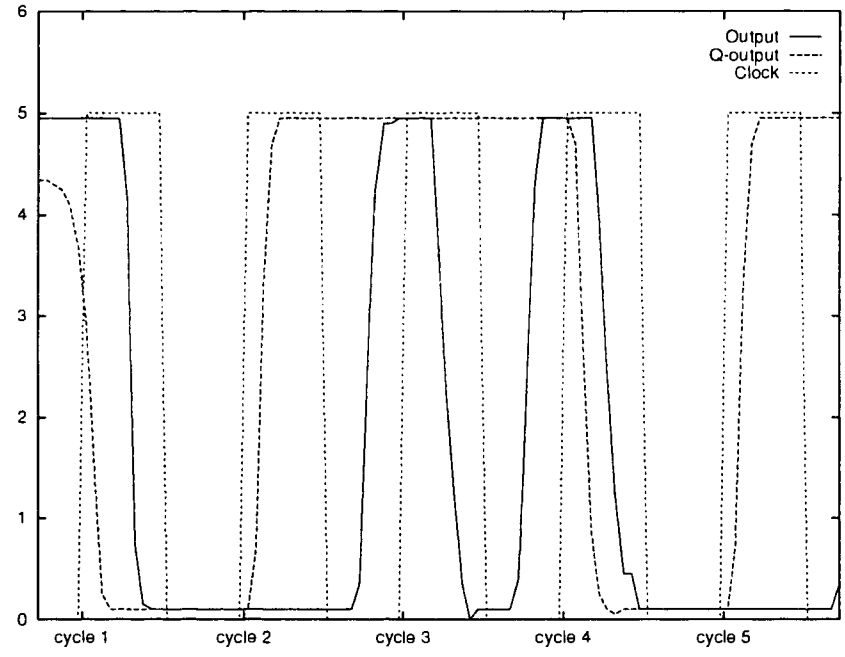

Figure D.5: SPICE results for a 50x50 ROM with dynamic flipflips.

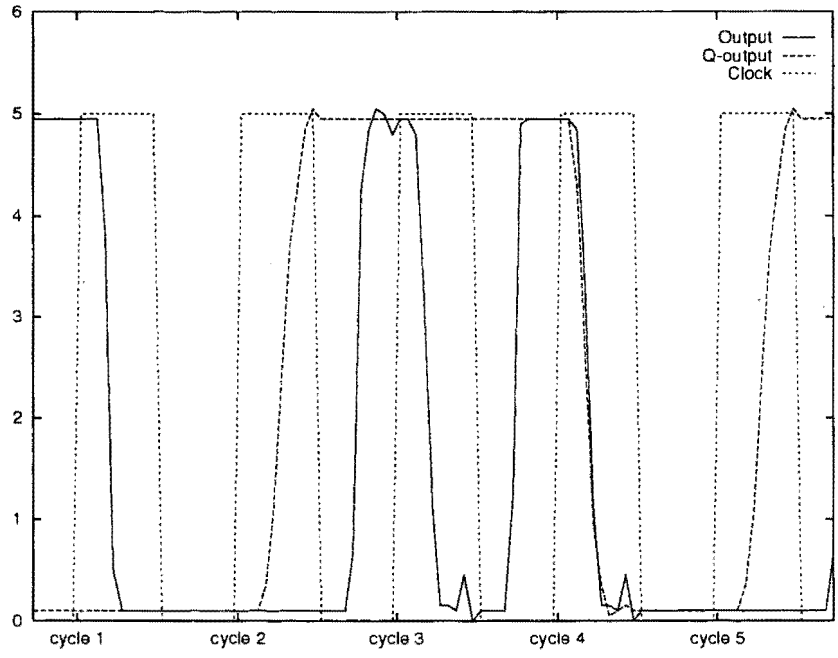

Figure D.6: SPICE results for a 100x26 ROM with static flipflips.

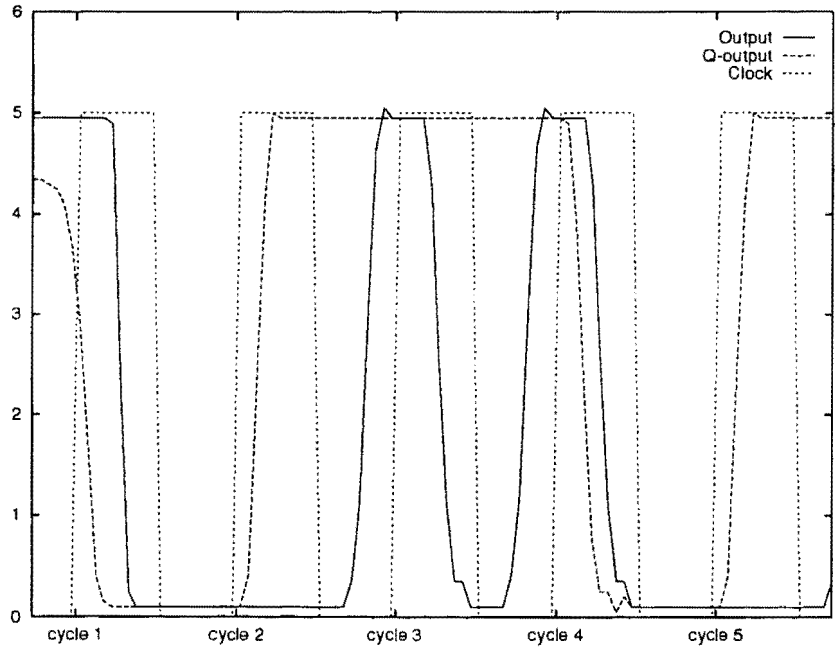

Figure D.7: SPICE results for a 100x26 ROM with dynamic flipflips.

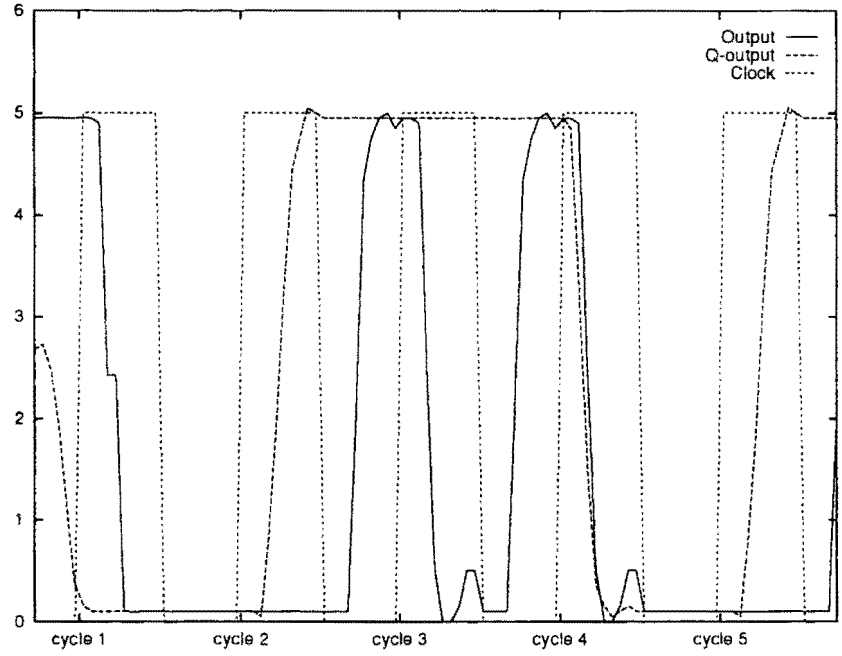

Figure D.8: SPICE results for a 100x50 ROM with static flipflips.

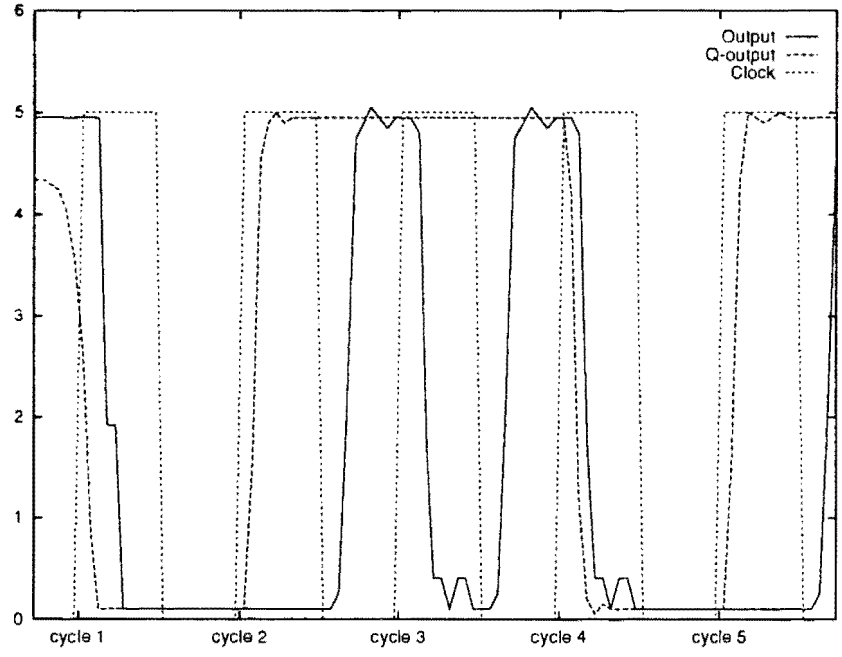

Figure D.9: SPICE results for a 100x50 ROM with dynamic flipflips.

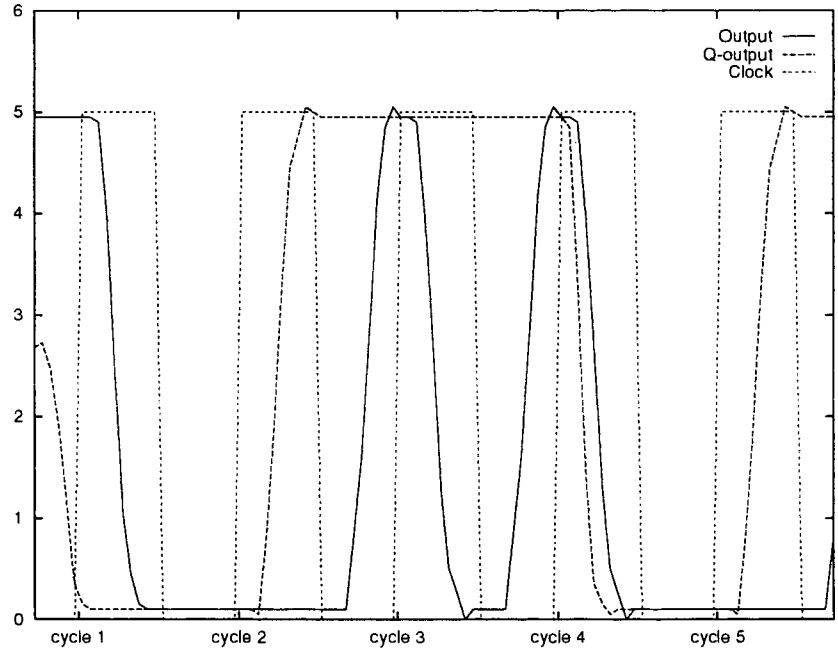

Figure D.10: SPICE results for a 100x50 ROM with static flipflips. All outputs drive a capacitance of 400pF.

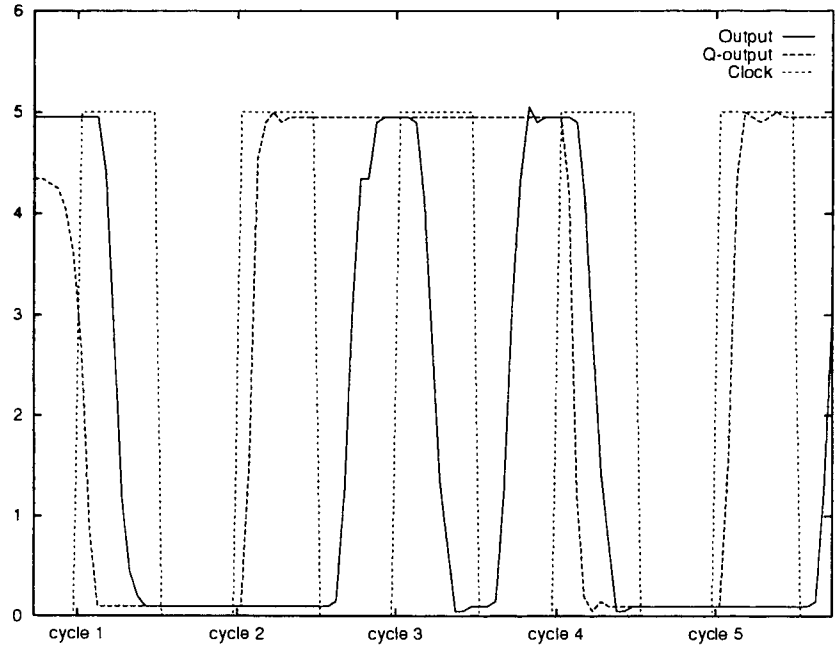

Figure D.11: SPICE results for a 100x50 ROM with dynamic flipflips. All outputs drive a capacitance of 400pF.

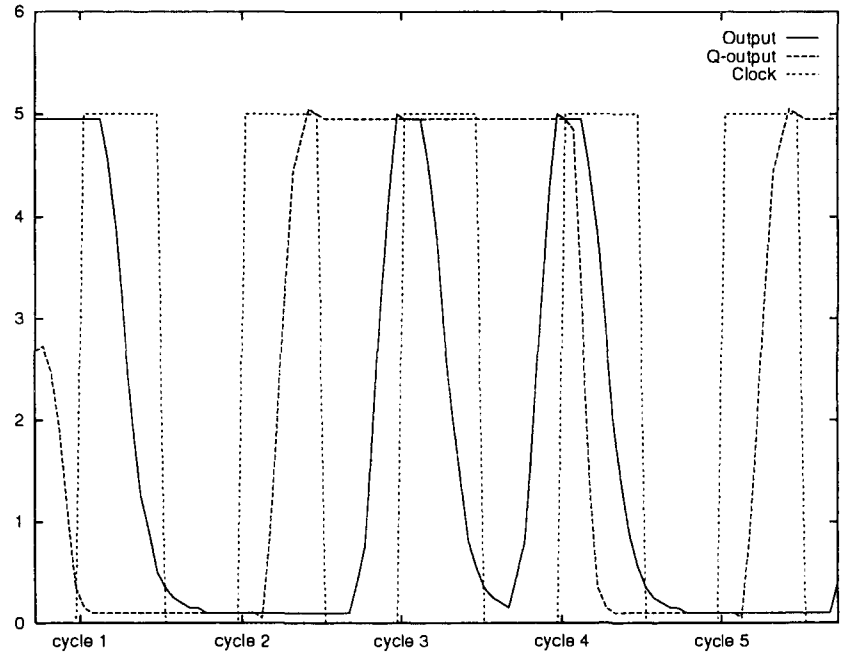

Figure D.12: SPICE results for a 100x50 ROM with static flipflips. All outputs drive a capacitance of 1nF.

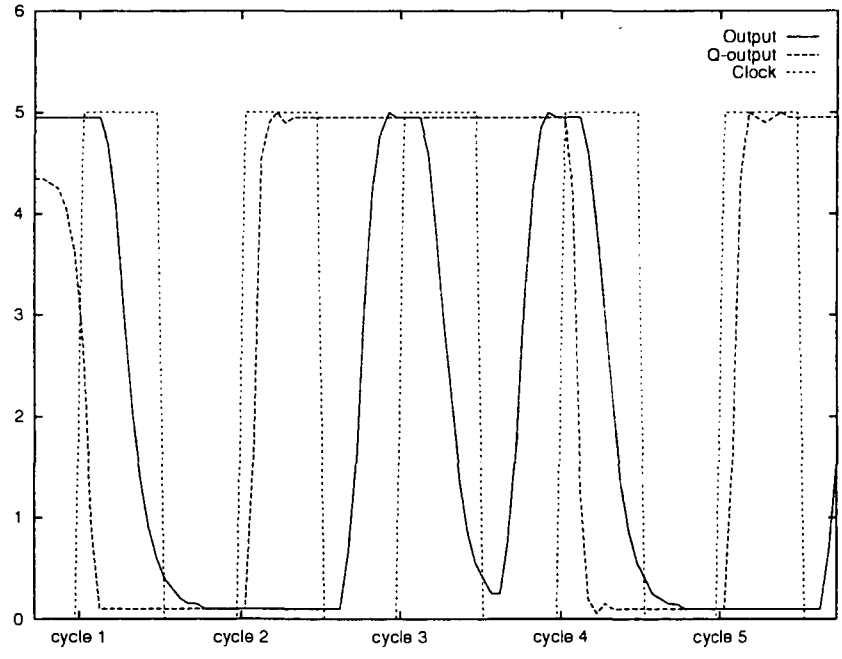

Figure D.13: SPICE results for a 100x50 ROM with dynamic flipflips. All outputs drive a capacitance of 1nF.

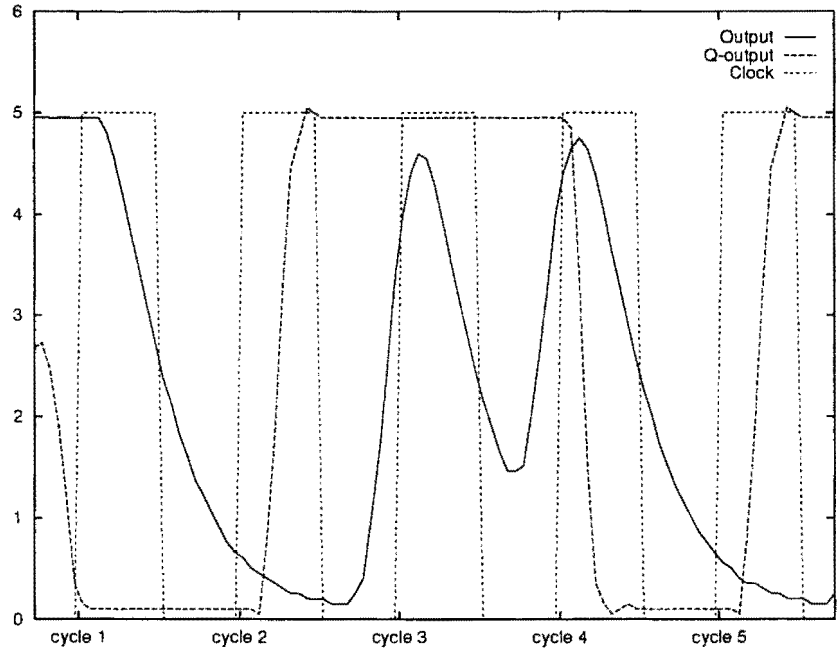

Figure D.14: SPICE results for a 100x50 ROM with static flipflips. All outputs drive a capacitance of 3nF.

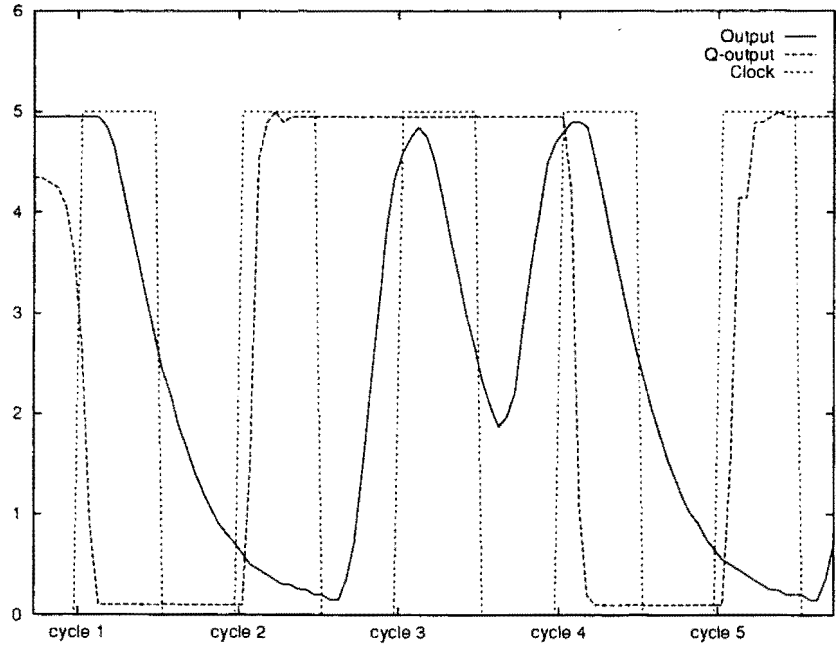

Figure D.15: SPICE results for a 100x50 ROM with dynamic flipflips. All outputs drive a capacitance of 3nF.

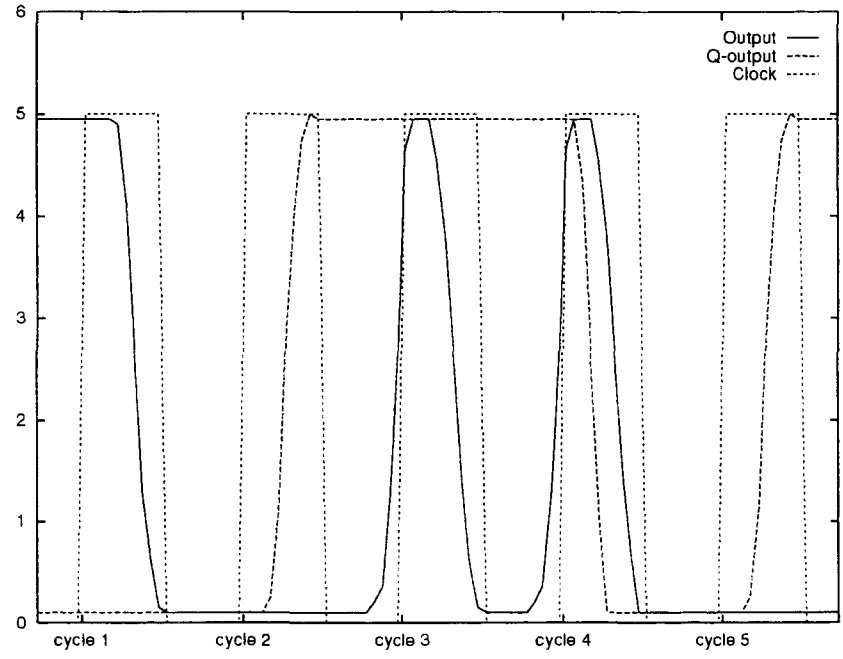

Figure D.16: SPICE results for a 200x100 ROM with static flipflips. All outputs drive a capacitance of 400pF.

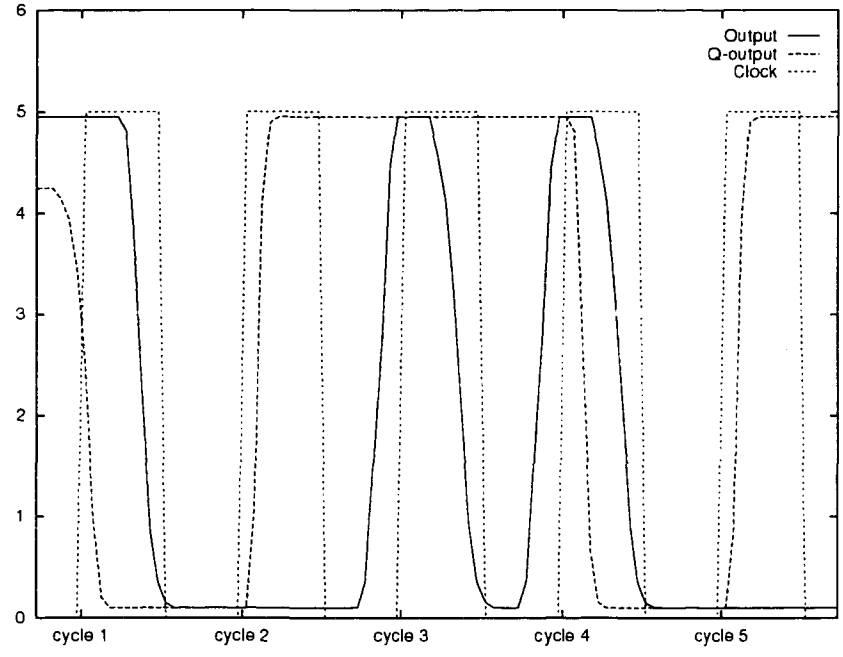

Figure D.17: SPICE results for a 200x100 ROM with dynamic flipflips. All outputs drive a capacitance of 400pF.

Appendix E

Finite state machines for size comparisons

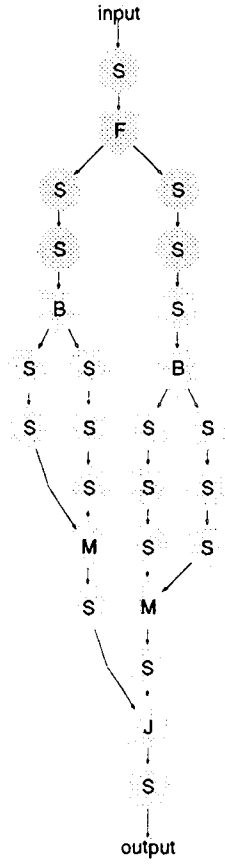

Figure E.1: Moderately sized multi–thread state machine.

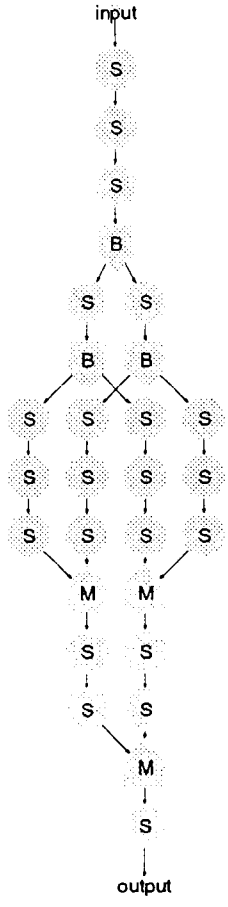

Figure E.2: Moderately sized flat state machine.

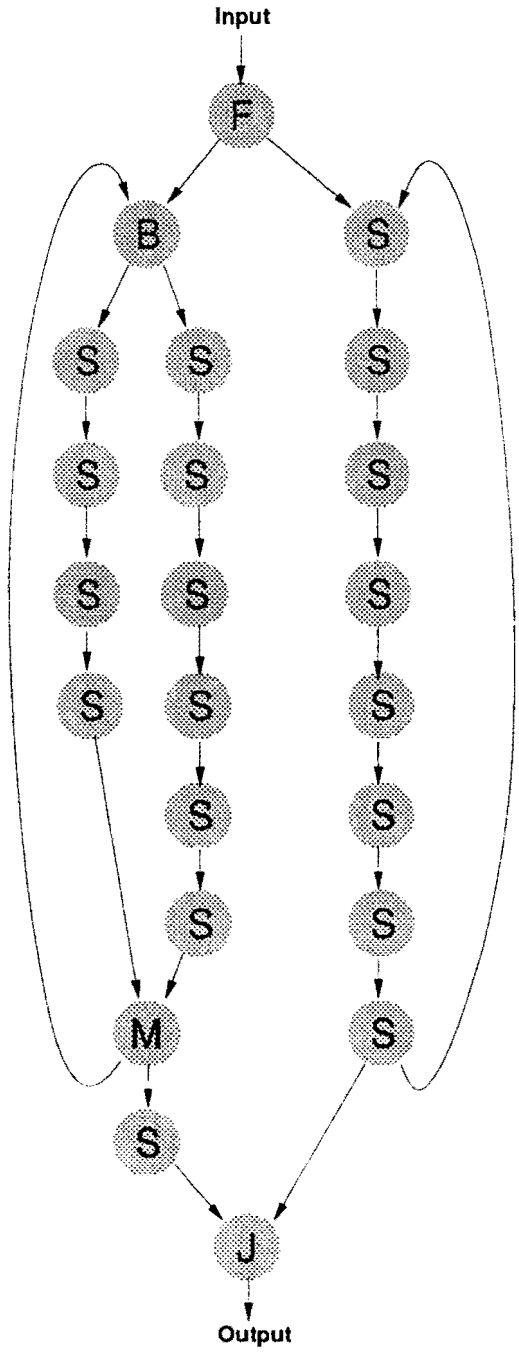

Figure E.3: Multi–thread state machine with loops.

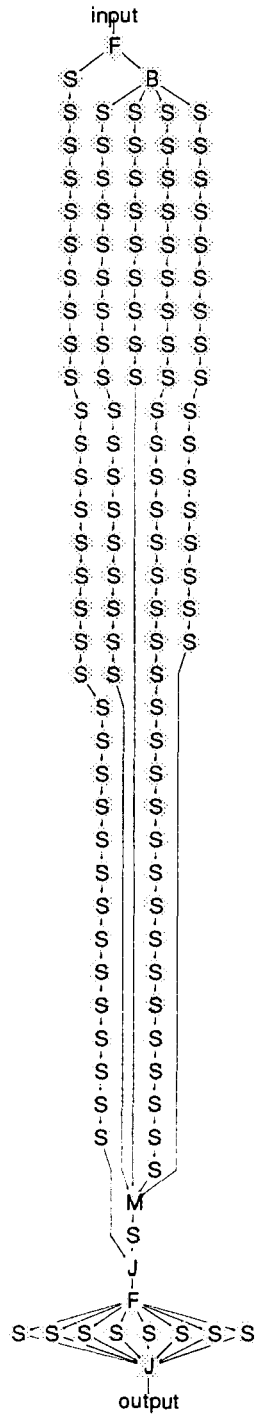

Figure E.4: Large multi–thread state machine.