Analysis of transition activity and power dissipation in synchronous logic circuits

Leijten, J.A.J.

Award date:
1993

Master Thesis

Jeroen Leijten

Analysis of Transition Activity and Power Dissipation in Synchronous Logic Circuits

coach at Philips Research Laboratories

Dr. Ir. J.L. van Meerbergen

coach at Eindhoven University of Technology

Prof. Dr. Ing. J.A.G. Jess

Work performed at Philips Research Laboratories, Eindhoven, from 1st May 1993 until 1st December 1993.

Jeroen Leijten

Eindhoven University of Technology
Philips Research Laboratories

Analysis of Transition Activity and Power Dissipation in Synchronous Logic Circuits

Abstract

In this work power dissipation in synchronous circuits is analyzed and divided into three different components: dissipation in the combinational logic, in the flipflops and in the clock lines. Power dissipation in the combinational logic is often the result of glitches (or redundant transitions). Glitches can be almost completely eliminated by simple techniques, such as the introduction of flipflops due to retiming/pipelining and/or by choosing different architectures. Power dissipation in the flipflops and in the clock lines can be minimized by introducing more parallelism.

Contents

List of Figures

List of Tables

1 Introduction

Over the past years much effort has been directed towards increasing the speed and throughput of digital integrated circuits while decreasing the area. On the other hand, less attention has been paid to the power consumed by these circuits.
However, in present day life, the portability of products, using integrated circuits of intense computational nature, is becoming more and more important. This puts a severe constraint on the power that may be consumed by these circuits. Power-efficient design will play a key role in making these portable products feasible. Battery technology is being improved, but it is unlikely that a dramatic solution to the power problem is forth-coming. Furthermore, to prevent malfunction and to guarantee reliability of integrated circuits so called 'hot spots' on the chip must be prevented. Areas of excessive power dissipation also form a technical barrier for the integration of more transistors on a single chip. These considerations reveal the need for lowering the power dissipation in integrated circuits.
There exist several methods to reduce power dissipation in CMOS integrated circuits. These methods lie in the field of technology (device scaling, scaling of supply voltage, choice of threshold voltage), of the basic circuit design style (static versus dynamic CMOS, pass-gate versus conventional CMOS logic styles, synchronous versus asynchronous timing), of the circuit architecture (architectural transformation, pipelining, parallelism, critical path reduction) and of the used algorithms (optimization of the computation complexity, parallelization of algorithms) [1].
In the research described in this report attention has been focused on architectures for CMOS circuits in the context of processing unit design for video applications using the PHIDEO high level synthesis environment. PHIDEO uses synchronous processing units. It is known that so called *glitches* can occur in these units. In this report glitches will be referred to as *redundant* transitions, as opposed to *useful* transitions. Redundant transitions lead to unnecessary power dissipation. An analysis has been made on the problem of this unnecessary dynamic power dissipation in CMOS combinational logic circuits. This was done firstly using transition probability calculations for a ripple carry adder and secondly using switch level simulation to determine transition activity at different circuit nodes in unit delay modelled multiplier circuits. Distinctions were made between useful transitions, necessary to achieve the desired circuit operation, and redundant transitions due to unnecessary glitching of circuit nodes in a single clock period and leading to extra power dissipation. It was found that many redundant transitions can occur in circuits with many reconvergent paths due to delay unbalance.
To include not only the effect of transition activity, but also the effect of nodal capacitance in the analysis, layouts were made and power estimations using circuit level simulation were carried out on the layout extractions. The testcase was a typical processing unit for PHIDEO, known as a *direction detector*. In the simulations a distinction was made between power dissipation in the combinational logic part of a circuit, power dissipation in the clock line and power consumed by pipelining flipflops in the circuit. It appeared that a great deal of reduction in power dissipation can be obtained when the amount of redundant

transitions in logic circuits is reduced by using retiming and pipelining. Furthermore it was found that the power dissipation in the flipflops and clock line can be lowered by making use of a parallel architecture.
Some ideas and recommendations for achieving reduction in power dissipation that followed from this research are given in this report.

2 Power dissipation in CMOS circuits

In circuits using CMOS technology there are three major sources of power dissipation:

- switching currents
- short-circuit currents
- leakage currents

The short-circuit currents and leakage currents can be made negligible by proper circuit design and technology. The remaining problem is formed by the switching currents. This switching component, resulting from the charging and discharging of capacitors, dominates the total power consumption. It is referred to as dynamic power dissipation.
The dynamic power consumption for a CMOS gate with a loading capacitor C_{load} is given by the formula:

$$P_{dyn} = p_t C_{load} V_{dd}^2 f \tag{1}$$

where C_{load} is the loading capacitance,
V_{dd} is the supply voltage,
f is the clock frequency,
p_t is the probability of a power consuming transition.

In this report a signal change from 0 to 1 or from 1 to 0 will be called a *transition.* A *power consuming transition* is defined as a 0 to 1 transition, because only when such a transition occurs power from the supply is consumed. A 1 to 0 transition is nothing more than a discharge of a nodal capacitance, so in this case no power from the supply is dissipated. Because 0 to 1 and 1 to 0 transitions always alternate, the number of transitions in a circuit will be approximately twice as large as the number of power consuming transitions.
The goal of the research is to reduce power dissipation for a given throughput. From equation 1 can be seen that a large reduction in power dissipation can be obtained by lowering the supply voltage V_{dd}. However, this supply voltage is dictated by the used layout technology. It is also clear that the dynamic power dissipation in a circuit is proportional to the number of transitions in the circuit. Therefore a reduction in the total number of transitions can lead to a reduction in the total power dissipation. The power dissipation is of course also linearly dependent on the loading capacitance of switching nodes.

3 Transition activity in a ripple carry adder

To become familiar with the problem of (unnecessary) signal transitions in logic circuits, transition activity in a ripple carry adder was analyzed using probability calculations.

3.1 The carry ripple principle

Figure 1 shows a block diagram of an N-bit ripple carry adder. This type of adder consists of

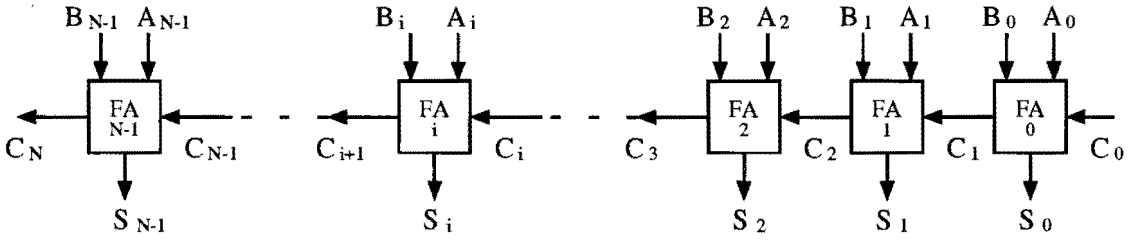


Figure 1: *Block diagram of an N-bit ripple carry adder.*

N cascaded full adders. Such a full adder FA_i computes the boolean sum S_i and carry-out C_{i+1} of its two input bits A_i and B_i and its carry input bit C_i, in the following manner:

$$S_i = A_iB_iC_i + A_i\overline{B}_i\overline{C}_i + \overline{A}_i\overline{B}_iC_i + \overline{A}_iB_i\overline{C}_i = C_i(A_i \oplus \overline{B}_i) + \overline{C}_i(A_i \oplus B_i) \quad (2)$$

$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i = A_iB_i + C_i(A_i + B_i) \quad (3)$$

In the ripple carry adder the carry will "ripple" through the full adder stages in the course of an addition. Of course each full adder has a non-zero delay. Here we assume a unit-delay model for each full adder stage. We further assume that new input bits A_i and B_i of the two input words $A_{N-1}...A_0$ and $B_{N-1}...B_0$ always arrive at equal times for all i.
When the inputs change, full adder FA_0 will introduce a delay in the calculation of S_0 and C_1. Therefore the new value of C_1 will arrive later than the new inputs A_1 and B_1. And thus full adder FA_1 will first compute its sum S_1 and carry-out C_2 using the new inputs A_1 and B_1, but the carry input C_1 of the previous addition. After this C_1 will reach its new value and full adder FA_1 performs another calculation with this new value. This means that S_1 and C_2 can change twice. We say that two transitions can occur in S_1 and C_2 as a result of one input change.
In the same way full adder FA_2 will first compute its sum S_2 and carry C_3 using carry input C_2 of the previous addition. After this it will recalculate its outputs using the first result for C_2 as computed by full adder FA_1 using C_1 of the previous addition. Finally it will calculate the correct output values using the correct value for C_2. So in total three transitions can occur in S_2 and C_3.
Extending this observation to full adder FA_i leads to the conclusion that $i+1$ transitions can occur in S_i and C_{i+1} as a result of a single input change. This means that in the worst case situation N transitions can occur in S_{N-1} and C_N.

3.2 Worst case number of transitions

In this section we will see for which inputs the worst case number of transitions in S_{N-1} and C_N occurs and what the probability of this worst case situation is.

The N full adders in the ripple carry adder together introduce a delay of N units for the carry to ripple from the first stage all the way through to C_N. In the worst situation, C_N makes N transitions. This is only possible if C_N has the values 0, 1, 0, 1, 0, 1,... or 1, 0, 1, 0, 1, 0,... for the delta time moments Δt_0, Δt_1, Δt_2,..., Δt_{N-1} between consecutive input changes. This can only occur if two conditions are met:

1. After the completion of the previous addition the carries of the full adder stages have the values $\underline{C} = \{C_N, C_{N-1}, C_{N-2}, C_{N-3}, ...\} = \{0, 1, 0, 1, ...\}$ or $\{1, 0, 1, 0, ...\}$. We call this the *initial condition* for the worst case number of transitions.

2. The carry must be able to ripple through all the full adder stages. We call this the *ripple condition* for the worst case number of transitions.

This is clear from figure 2 where a single full adder FA_i (with $i = 3$) is depicted with i transitions occuring at its inputs. The i transitions will ripple through the full adder and

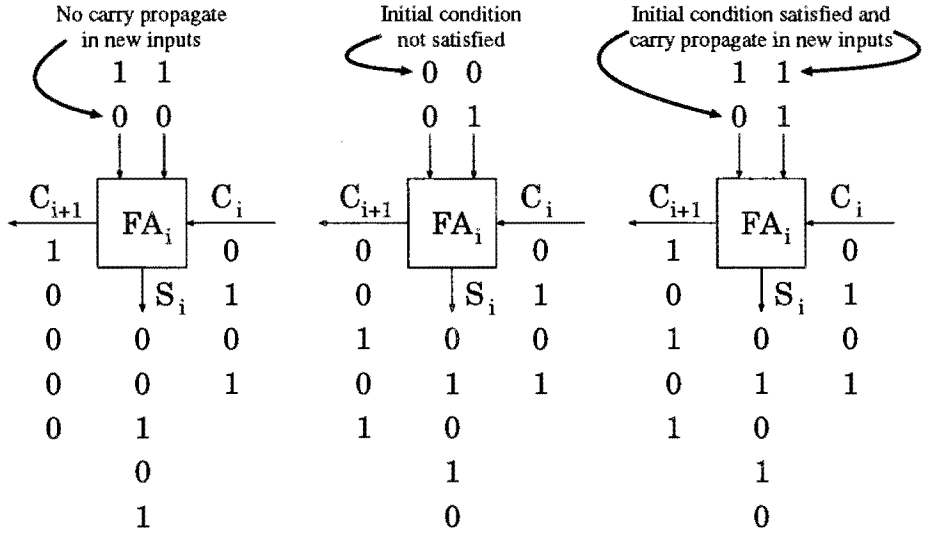


Figure 2: *Initial and ripple conditions for a single full adder.*

cause transitions at the carry output if the new inputs of the full adder constitute a carry propagate condition. If this is not the case a carry kill or carry generate occurs and the carry ripple will be stopped. This confirms the necessary ripple condition.
If as a result of the previous addition the carry output of the full adder was different from the carry input, $i + 1$ transitions will occur in the carry output. If not only i transitions will occur. This confirms the initial condition.

Conditions 1 and 2 can only be satisfied if certain requirements for the previous inputs A'_i and B'_i and the new inputs A_i and B_i are met. Since we assume that C_0 is always equal to zero, these requirements are as follows:

- The result $\underline{C} = \{C_1, C_2, C_3, C_4, ...\} = \{0, 1, 0, 1, ...\}$ of the previous addition can only occur if

  $$(A'_0 B'_0 = 0) \wedge (A'_1 = B'_1 = 1) \wedge (A'_i = B'_i = \overline{A}'_{i-1} = \overline{B}'_{i-1}) \text{ for all i with } 2 \leq i < N \quad (4)$$

  A ripple of the carry through all stages will now occur if

  $$A_0 = B_0 = 1 \wedge A_i \oplus B_i = 1 \text{ for all i with } 1 \leq i < N \quad (5)$$

- The result $\underline{C} = \{C_1, C_2, C_3, C_4, ...\} = \{1, 0, 1, 0, ...\}$ of the previous addition can only occur if

  $$(A'_0 B'_0 = 1) \wedge (A'_1 = B'_1 = 0) \wedge (A'_i = B'_i = \overline{A}'_{i-1} = \overline{B}'_{i-1}) \text{ for all i with } 2 \leq i < N \quad (6)$$

  A ripple of the carry through all stages will now occur if

  $$A_0 B_0 = 0 \wedge A_i \oplus B_i = 1 \text{ for all i with } 1 \leq i < N \quad (7)$$

Figure 3 shows an example of this for N=4. From observations 4 and 6 can be concluded

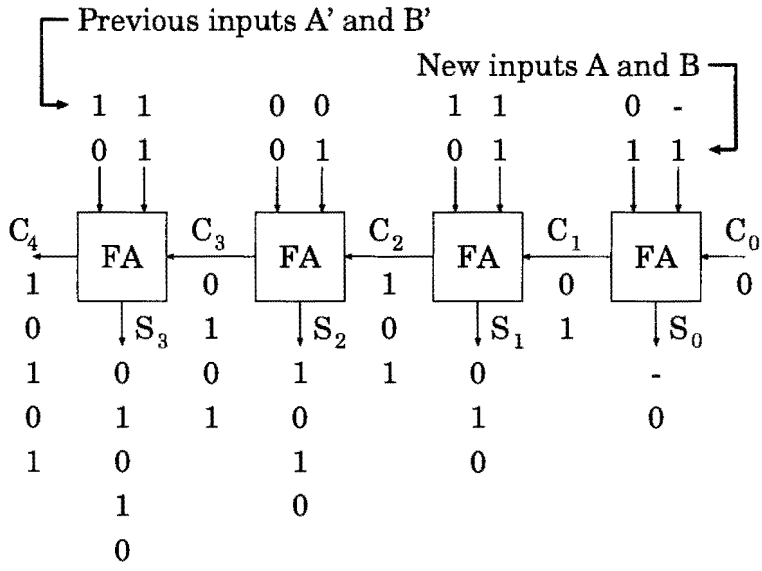


Figure 3: *Worst case number of transitions in a 4 bit ripple carry adder.*

that the initial condition for the worst case number of transitions can only be satisfied by four different input situations, shown in table 1. From observations 5 and 7 one can easily derive the inputs that must follow these initial inputs, so that the ripple condition

Initial condition						*Ripple condition*					
Previous inputs						*New inputs*					
A_0B_0	A_1B_1	A_2B_2	A_3B_3	A_4B_4	etc.	A_0B_0	A_1B_1	A_2B_2	A_3B_3	A_4B_4	etc.
01	11	00	11	00	...	11	01/10	01/10	01/10	01/10	...
10	11	00	11	00	...	11	01/10	01/10	01/10	01/10	...
00	11	00	11	00	...	11	01/10	01/10	01/10	01/10	...
11	00	11	00	11	...	00/01/10	01/10	01/10	01/10	01/10	...

Table 1: *Input combinations resulting in worst case number of transitions*

for the worst case number of transitions is also met. These are also shown in table 1. With input words of N bits there exist 2^{2N} possible combinations of input bits A_i and B_i $(0 \leq i < N)$. So, if random inputs are assumed, each input combination has a probability of $1/2^{2N}$ to appear. Therefore the probability that the initial condition occurs is equal to $4/2^{2N}$. However, for the worst case number of transitions to be generated, the ripple condition must also be satisfied. For the new inputs the probability of $A_0B_0 = 11$ is equal to 1/4. The probability of $A_0B_0 = 00/01/10$ is equal to 3/4. The probability of a single input pair $A_iB_i = 01/10$ with $1 \leq i < N$ is equal to 1/2. In total there exist $N-1$ input pairs A_iB_i for $1 \leq i < N$. With these results the probability that both the initial and the ripple conditions are met becomes equal to:

$$\frac{1}{2^{2N}}\left(\frac{1}{4}\left(\frac{1}{2}\right)^{N-1} + \frac{1}{4}\left(\frac{1}{2}\right)^{N-1} + \frac{1}{4}\left(\frac{1}{2}\right)^{N-1} + \frac{3}{4}\left(\frac{1}{2}\right)^{N-1}\right) = \frac{6}{2^{2N}2^{2N-1}} = 3\left(\frac{1}{8}\right)^{N} \quad (8)$$

So using random inputs of N bits the probability that the number of transitions in S_{N-1} and C_N as a result of a single input change equals N is equal to $3(1/8)^N$. Already for relatively small word sizes N this probability will be almost negligible. In a practical situation more interest lies in the average number of transitions appearing in a circuit. This will be the subject of the next section.

3.3 Average number of transitions

The average number of transitions that occurs at the sum and carry outputs of the different full adder stages in the ripple carry adder as a result of random input changes, can be calculated using probability calculations.
There are two types of events that cause transitions each time a new input is applied:

1. The change in the inputs itself. At the moment of input change all carry bits C_i $(0 \leq i < N)$ of the full adder stages are constant.
2. Transitions in the carry bits of the stages preceding the full adder under examination. When these transitions occur all the inputs A_i and B_i $(0 \leq i < N)$ are constant.

These two events always occur in this order. The first event happens only once each time new input words are applied and can therefore cause only one transition for each input change. The second event can happen at most i times for full adder FA_i as a result of a single input change.

Using equation 3 we can calculate the average number of transitions for the carry bit C_{i+1}. In these calculations we will write the probability of a $0 \rightarrow 1$ transition in a signal X_i as $P(X_i : 0 \rightarrow 1)$ and the probability of a $1 \rightarrow 0$ transition in X_i as $P(X_i : 1 \rightarrow 0)$. We make use of the fact that the input transition probabilities $P(A_i : 0 \rightarrow 0)$, $P(A_i : 0 \rightarrow 1)$, $P(A_i : 1 \rightarrow 0)$, $P(A_i : 1 \rightarrow 1)$, $P(B_i : 0 \rightarrow 0)$, $P(B_i : 0 \rightarrow 1)$, $P(B_i : 1 \rightarrow 0)$ and $P(B_i : 1 \rightarrow 1)$ are all equal to $1/4$ and the signal probabilities $P(A_i = 0)$, $P(A_i = 1)$, $P(B_i = 0)$ and $P(B_i = 1)$ are all equal to $1/2$ for random input signals.

ad 1. C_i is constant.

The transition probabilities for C_{i+1} in this case are as follows:

$$\begin{aligned} P(C_{i+1} : 0 \rightarrow 1) &= P(C_i = 0)P(A_iB_i : 0 \rightarrow 1) + \\ &\quad P(C_i = 1)P(A_i + B_i : 0 \rightarrow 1) \end{aligned} \tag{9}$$

$$\begin{aligned} P(C_{i+1} : 1 \rightarrow 0) &= P(C_i = 0)P(A_iB_i : 1 \rightarrow 0) + \\ &\quad P(C_i = 1)P(A_i + B_i : 1 \rightarrow 0) \end{aligned} \tag{10}$$

The probabilities $P(C_i = 0)$, $P(C_i = 1)$, $P(A_iB_i : 0 \rightarrow 1)$, $P(A_iB_i : 1 \rightarrow 0)$, $P(A_i + B_i : 0 \rightarrow 1)$ and $P(A_i + B_i : 1 \rightarrow 0)$ still have to be determined to find solutions for equations 9 and 10:

$$\begin{aligned} P(C_{i+1} = 0) &= P(C_i = 0)\{P(A_i = 0) + P(A_i = 1)P(B_i = 0)\} + \\ &\quad P(C_i = 1)P(A_i = 0)P(B_i = 0) \\ &= \frac{3}{4}P(C_i = 0) + \frac{1}{4}P(C_i = 1) \end{aligned} \tag{11}$$

$$\begin{aligned} P(C_{i+1} = 1) &= P(C_i = 0)P(A_i = 1)P(B_i = 1) + \\ &\quad P(C_i = 1)\{P(A_i = 1) + P(A_i = 0)P(B_i = 1)\} \\ &= \frac{1}{4}P(C_i = 0) + \frac{3}{4}P(C_i = 1) \end{aligned} \tag{12}$$

C_0 is always equal to zero, so $P(C_0 = 0) = 1$ and $P(C_0 = 1) = 0$. Substituting these values in equations 11 and 12 gives:

$$P(C_1 = 0) = \frac{3}{4} \times 1 + \frac{1}{4} \times 0 = \frac{3}{4} \qquad P(C_1 = 1) = \frac{1}{4} \times 1 + \frac{3}{4} \times 0 = \frac{1}{4}$$

$$P(C_2 = 0) = \frac{3}{4} \times \frac{3}{4} + \frac{1}{4} \times \frac{1}{4} = \frac{5}{8} \qquad P(C_2 = 1) = \frac{1}{4} \times \frac{3}{4} + \frac{3}{4} \times \frac{1}{4} = \frac{3}{8}$$

$$P(C_3=0)=\frac{3}{4}\times\frac{5}{8}+\frac{1}{4}\times\frac{3}{8}=\frac{9}{16} \qquad P(C_3=1)=\frac{1}{4}\times\frac{5}{8}+\frac{3}{4}\times\frac{3}{8}=\frac{7}{16}$$

$$P(C_4=0)=\frac{3}{4}\times\frac{9}{16}+\frac{1}{4}\times\frac{7}{16}=\frac{17}{32} \qquad P(C_4=1)=\frac{1}{4}\times\frac{9}{16}+\frac{3}{4}\times\frac{7}{16}=\frac{15}{32}$$

$$\Downarrow \qquad\qquad \Downarrow$$

$$P(C_i=0)=\frac{2^i+1}{2^{i+1}} \qquad P(C_i=1)=\frac{2^i-1}{2^{i+1}} \tag{13}$$

For large i ($i \to \infty$) probabilities $P(C_i=0)$ and $P(C_i=1)$ will both approximate $1/2$.
The transition probabilities for A_iB_i and A_i+B_i are:

$$\begin{aligned} P(A_iB_i:0\to 1) &= P(A_i:0\to 1)\{P(B_i:0\to 1)+P(B_i:1\to 1)\}+ \\ &\quad P(A_i:1\to 1)P(B_i:0\to 1) \\ &= \frac{1}{4}\left(\frac{1}{4}+\frac{1}{4}\right)+\frac{1}{4}\times\frac{1}{4} \\ &= \frac{3}{16} \end{aligned} \tag{14}$$

$$\begin{aligned} P(A_iB_i:1\to 0) &= P(A_i:1\to 0)\{P(B_i:1\to 0)+P(B_i:1\to 1)\}+ \\ &\quad P(A_i:1\to 1)P(B_i:1\to 0) \\ &= \frac{1}{4}\left(\frac{1}{4}+\frac{1}{4}\right)+\frac{1}{4}\times\frac{1}{4} \\ &= \frac{3}{16} \end{aligned} \tag{15}$$

$$\begin{aligned} P(A_i+B_i:0\to 1) &= P(A_i:0\to 0)P(B_i:0\to 1)+ \\ &\quad P(A_i:0\to 1)\{P(B_i:0\to 0)+P(B_i:0\to 1)\} \\ &= \frac{1}{4}\times\frac{1}{4}+\frac{1}{4}\left(\frac{1}{4}+\frac{1}{4}\right) \\ &= \frac{3}{16} \end{aligned} \tag{16}$$

$$\begin{aligned} P(A_i+B_i:1\to 0) &= P(A_i:0\to 0)P(B_i:1\to 0)+ \\ &\quad P(A_i:1\to 0)\{P(B_i:0\to 0)+P(B_i:1\to 0)\} \\ &= \frac{1}{4}\times\frac{1}{4}+\frac{1}{4}\left(\frac{1}{4}+\frac{1}{4}\right) \\ &= \frac{3}{16} \end{aligned} \tag{17}$$

Substituting the results from equations 13, 14, 15, 16 and 17 in equations 9 and 10 gives:

$$\begin{aligned} P(C_{i+1}:0\rightarrow 1) &= \frac{2^i+1}{2^{i+1}}\times\frac{3}{16}+\frac{2^i-1}{2^{i+1}}\times\frac{3}{16} \\ &= \frac{3}{16}\times\left(\frac{2^i+1}{2^{i+1}}+\frac{2^i-1}{2^{i+1}}\right) \\ &= \frac{3}{16}\times\frac{2\times 2^i}{2^{i+1}} \\ &= \frac{3}{16} \end{aligned} \tag{18}$$

$$\begin{aligned} P(C_{i+1}:1\rightarrow 0) &= \frac{2^i+1}{2^{i+1}}\times\frac{3}{16}+\frac{2^i-1}{2^{i+1}}\times\frac{3}{16} \\ &= \frac{3}{16}\times\left(\frac{2^i+1}{2^{i+1}}+\frac{2^i-1}{2^{i+1}}\right) \\ &= \frac{3}{16}\times\frac{2\times 2^i}{2^{i+1}} \\ &= \frac{3}{16} \end{aligned} \tag{19}$$

So the total probability of a transition in C_i (either $0\rightarrow 1$ or $1\rightarrow 0$) caused by event 1 is equal to $3/8$ for all i with $1\leq i<N$. Therefore the average number of transitions in C_i (with $1\leq i<N$) caused by event 1 is equal to $3/8$ times the number of input changes.

ad 2. A_i and B_i are constant.
C_{i+1} will make a transition if C_i makes a transition and $A_i\oplus B_i=1$. The probability $P(A_i\oplus B_i=1)$ is equal to $1/2$ for random inputs. Therefore, the average number of transitions in C_{i+1} caused by event 2 is equal to half the number of transitions in C_i.

We define the average transition ratio of a signal as the average number of transitions of the signal per input change. For a signal X_i this transition ratio is written as $TR(X_i)$. Summing the effect of the two events and bearing in mind that no transitions can occur in C_0, because this bit is constant, the following average transition ratio for C_{i+1} can be found:

$$\begin{aligned} TR(C_{i+1}) &= \frac{3}{8}+\frac{1}{2}TR(C_i) \\ &= \frac{3}{8}\left(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots\right) \\ &= \frac{3}{8}\sum_{k=0}^{i}\left(\frac{1}{2}\right)^k \\ &= \frac{3}{8}\left(\sum_{k=0}^{\infty}\left(\frac{1}{2}\right)^k-\sum_{k=i+1}^{\infty}\left(\frac{1}{2}\right)^k\right) \end{aligned}$$

$$
\begin{aligned}
&= \frac{3}{8}\left(1-\left(\frac{1}{2}\right)^{i+1}\right)\sum_{k=0}^{\infty}\left(\frac{1}{2}\right)^{k} \\
&= \frac{3}{8}\left(1-\left(\frac{1}{2}\right)^{i+1}\right)\frac{1}{1-\frac{1}{2}} \\
&= \frac{3}{4}-\frac{3}{4}\left(\frac{1}{2}\right)^{i+1}
\end{aligned} \tag{20}
$$

The calculation of the average number of transitions in sum output S_i of full adder FA_i is simple now:

ad 1. C_i is constant.
The probability of a transition in S_i is equal to the probability of a transition in exactly one of the inputs A_i and B_i. This probability is equal to 1/2. Therefore the average number of transitions in S_i caused by event 1 is equal to half the number of input changes.

ad 2. A_i and B_i are constant.
S_i will make a transition if C_i makes a transition. Therefore the average number of transitions in S_i caused by event 2 is equal to the average number of transitions in C_i.

Summing the effect of these two events results in the following average transition ratio for S_i:

$$
\begin{aligned}
TR(S_i) &= \frac{1}{2}+TR(C_i) \\
&= \frac{1}{2}+\frac{3}{4}-\frac{3}{4}\left(\frac{1}{2}\right)^{i} \\
&= \frac{5}{4}-\frac{3}{4}\left(\frac{1}{2}\right)^{i}
\end{aligned} \tag{21}
$$

The average number of signal transitions can be calculated by multiplying the average transition ratio by the total number of input changes applied to the circuit.

3.4 Average number of useful and redundant transitions

It is also possible to calculate the average number of useful transitions that appear at different nodes in the circuit. A useful transition occurs when a signal goes from a stable state (0 or 1) as part of the result of the previous addition to a different stable state (1 respectively 0) as part of the result of the new addition. In other words a useful transition occurs if the signal makes transitions $0 \rightarrow 1 \rightarrow 0 \rightarrow ... \rightarrow 1$ or $1 \rightarrow 0 \rightarrow 1 \rightarrow ... \rightarrow 0$ as a result of a single input change. This implies that if the number of transitions a signal makes as a result of a single input change is an odd number, this signal makes one useful

transition. All extra transitions it may make are not useful, but redundant. It also implies that all transitions a signal makes are redundant, if that number of transitions is even. We will first examine the transition activity in the sum outputs. Note that the probability of a useful transition in S_i is equal to:

P(a useful transition in S_i)
= P(odd number of transitions in S_i)
= P(a transition caused by the input change itself and an even number of transitions caused by the carry ripple)
+ P(no transition caused by the input change itself and an odd number of transitions caused by the carry ripple).

Transitions in the sum output of a full adder FA_i caused by the carry ripple are independent of the inputs A_i and B_i of this full adder. These transitions only depend on transitions in the carry input C_i. Therefore the probability of transitions caused by the input change and the probability of transitions caused by the carry ripple are independent. Making use of this fact and of equation 2 the average number of useful transitions in the sum outputs of the full adder stages can be calculated. In the following calculations the probability of an even respectively odd number of transitions in a signal X_i will be written as P(even X_i) respectively P(odd X_i). Now the probability of a useful transition in S_i is equal to:

$$\begin{aligned}
P(\text{odd } S_i) &= [P(C_i = 0)\{P(A_i \oplus B_i : 0 \rightarrow 1) + P(A_i \oplus B_i : 1 \rightarrow 0)\} \\
&+ P(C_i = 1)\{P(A_i \oplus \overline{B}_i : 0 \rightarrow 1) + P(A_i \oplus \overline{B}_i : 1 \rightarrow 0)\}] \\
&\cdot P(\text{even } C_i) \\
&+ [P(C_i = 0)\{P(A_i \oplus B_i : 0 \rightarrow 0) + P(A_i \oplus B_i : 1 \rightarrow 1)\} \\
&+ P(C_i = 1)\{P(A_i \oplus \overline{B}_i : 0 \rightarrow 0) + P(A_i \oplus \overline{B}_i : 1 \rightarrow 1)\}] \\
&\cdot P(\text{odd } C_i) \\
&= \left[\frac{2^i + 1}{2^{i+1}}\left(\frac{1}{4} + \frac{1}{4}\right) + \frac{2^i - 1}{2^{i+1}}\left(\frac{1}{4} + \frac{1}{4}\right)\right] \cdot P(\text{even } C_i) \\
&+ \left[\frac{2^i + 1}{2^{i+1}}\left(\frac{1}{4} + \frac{1}{4}\right) + \frac{2^i - 1}{2^{i+1}}\left(\frac{1}{4} + \frac{1}{4}\right)\right] \cdot P(\text{odd } C_i) \\
&= \frac{2 \times 2^i}{2^{i+2}} \cdot P(\text{even } C_i) + \frac{2 \times 2^i}{2^{i+2}} \cdot (1 - P(\text{even } C_i)) \\
&= \frac{2^{i+1}}{2^{i+2}} \\
&= \frac{1}{2}
\end{aligned} \tag{22}$$

Or

$$P(\text{useful transition in } S_i) = P(\text{odd } S_i) = \frac{1}{2} \tag{23}$$

For the carry output of the different stages in the ripple carry adder a similar calculation can be performed. Here the probability of a useful transition in C_{i+1} is equal to:

P(a useful transition in C_{i+1})
= P(odd number of transitions in C_{i+1})
= P(a transition caused by the input change itself and
an even number of transitions caused by the carry ripple)
+ P(no transition caused by the input change itself and
an odd number of transitions caused by the carry ripple).

Now the transitions caused by the carry ripple are dependent on the input state, because the carry will only ripple through a full adder stage FA_i and cause transitions in the carry-out, if the new inputs A_i and B_i satisfy $A_i \oplus B_i = 1$. This means that the probability of transitions caused by an input change and the probability of transitions caused by the carry ripple are not independent and as a result of that the probability calculations for C_{i+1} will be somewhat more complicated.
A transition in C_{i+1} resulting from the input change itself will occur if:

- $C_i = 0$ and A_iB_i makes a transition ($0 \rightarrow 1$ or $1 \rightarrow 0$) or
- $C_i = 1$ and $A_i + B_i$ makes a transition ($0 \rightarrow 1$ or $1 \rightarrow 0$).

An even number of transitions in C_{i+1} resulting from the carry ripple will occur if:

- the new inputs satisfy $A_i \oplus B_i = 0$ or
- the new inputs satisfy $A_i \oplus B_i = 1$ and the number of transitions in C_i is even.

No transition in C_{i+1} resulting from the input change itself will occur if:

- $C_i = 0$ and A_iB_i remains constant ($0 \rightarrow 0$ or $1 \rightarrow 1$) or
- $C_i = 1$ and $A_i + B_i$ remains constant ($0 \rightarrow 0$ or $1 \rightarrow 1$).

An odd number of transitions in C_{i+1} resulting from the carry ripple will occur if:

- $A_i \oplus B_i = 1$ and the number of transitions in C_i is odd.

Using these properties and equation 3 the probability of a useful transition in C_{i+1} can be calculated:

$$
\begin{aligned}
&P(\text{odd } C_{i+1}) \\
&= P(C_i = 0)\{P(A_i : 0 \rightarrow 1)P(B_i : 0 \rightarrow 1) + P(A_i : 0 \rightarrow 1)P(B_i : 1 \rightarrow 1) \\
&+ P(A_i : 1 \rightarrow 1)P(B_i : 0 \rightarrow 1) + P(A_i : 1 \rightarrow 0)P(B_i : 1 \rightarrow 0) \\
&+ [P(A_i : 1 \rightarrow 0)P(B_i : 1 \rightarrow 1) + P(A_i : 1 \rightarrow 1)P(B_i : 1 \rightarrow 0)]P(\text{even } C_i)\}
\end{aligned}
$$

$$
\begin{aligned}
&+ \quad P(C_i = 1)\{P(A_i : 0 \to 1)P(B_i : 0 \to 1) + P(A_i : 0 \to 0)P(B_i : 1 \to 0) \\
&+ \quad P(A_i : 1 \to 0)P(B_i : 0 \to 0) + P(A_i : 1 \to 0)P(B_i : 1 \to 0) \\
&+ \quad [P(A_i : 0 \to 0)P(B_i : 0 \to 1) + P(A_i : 0 \to 1)P(B_i : 0 \to 0)]P(\text{even } C_i)\} \\
&+ \quad [P(C_i = 0)\{P(A_i : 0 \to 0)P(B_i : 0 \to 1) + P(A_i : 0 \to 0)P(B_i : 1 \to 1) \\
&+ \quad P(A_i : 0 \to 1)P(B_i : 0 \to 0) + P(A_i : 0 \to 1)P(B_i : 1 \to 0) \\
&+ \quad P(A_i : 1 \to 0)P(B_i : 0 \to 1) + P(A_i : 1 \to 1)P(B_i : 0 \to 0)\} \\
&+ \quad P(C_i = 1)\{P(A_i : 0 \to 0)P(B_i : 1 \to 1) + P(A_i : 0 \to 1)P(B_i : 1 \to 0) \\
&+ \quad P(A_i : 1 \to 0)P(B_i : 0 \to 1) + P(A_i : 1 \to 0)P(B_i : 1 \to 1) \\
&+ \quad P(A_i : 1 \to 1)P(B_i : 0 \to 0) + P(A_i : 1 \to 1)P(B_i : 1 \to 0)] \\
&\cdot \quad P(\text{odd } C_i)\} \\
&= \quad \frac{2^i + 1}{2^{i+1}}\{\frac{1}{4} + \frac{1}{8}P(\text{even } C_i)\} \\
&+ \quad \frac{2^i - 1}{2^{i+1}}\{\frac{1}{4} + \frac{1}{8}P(\text{even } C_i)\} \\
&+ \quad [\frac{2^i + 1}{2^{i+1}}\frac{3}{8} + \frac{2^i - 1}{2^{i+1}}\frac{3}{8}]P(\text{odd } C_i) \\
&= \quad \frac{1}{4} + \frac{1}{8}(1 - P(\text{odd } C_i)) + \frac{3}{8}P(\text{odd } C_i) \\
&= \quad \frac{3}{8} + \frac{1}{4}P(\text{odd } C_i) \qquad (24)
\end{aligned}
$$

Using $C_0 = 0$ and consequently $P(\text{odd } C_0) = 0$ equation 24 can be rewritten as:

$$
\begin{aligned}
P(\text{useful transition in } C_{i+1}) &= P(\text{odd } C_{i+1}) \\
&= \frac{3}{8}\sum_{k=0}^{i}\left(\frac{1}{4}\right)^k \\
&= \frac{3}{8}\left(\sum_{k=0}^{\infty}\left(\frac{1}{4}\right)^k - \sum_{k=i+1}^{\infty}\left(\frac{1}{4}\right)^k\right) \\
&= \frac{3}{8}\left(1 - \left(\frac{1}{4}\right)^{i+1}\right)\sum_{k=0}^{\infty}\left(\frac{1}{4}\right)^k \\
&= \frac{3}{8}\left(1 - \left(\frac{1}{4}\right)^{i+1}\right)\frac{1}{1 - \frac{1}{4}} \\
&= \frac{1}{2} - \frac{1}{2}\left(\frac{1}{4}\right)^{i+1} \qquad (25)
\end{aligned}
$$

For large i this probability will approach 1/2.

The average redundant transition ratio RTR for C_{i+1} and S_i can now be calculated by subtracting the corresponding probabilities of a useful transition (equations 25 respectively 23) from the average transition ratios (equations 20 respectively 21):

$$
RTR(S_i) = TR(S_i) - P(\text{useful transition in } S_i)
$$

$$
\begin{aligned}
&= \frac{5}{4} - \frac{3}{4}\left(\frac{1}{2}\right)^{i} - \frac{1}{2} \\
&= \frac{3}{4} - \frac{3}{4}\left(\frac{1}{2}\right)^{i} \qquad (26)
\end{aligned}
$$

$$
\begin{aligned}
RTR(C_{i+1}) &= TR(C_{i+1}) - P(\text{useful transition in } C_{i+1}) \\
&= \frac{3}{4} - \frac{3}{4}\left(\frac{1}{2}\right)^{i+1} - \left(\frac{1}{2} - \frac{1}{2}\left(\frac{1}{4}\right)^{i+1}\right) \\
&= \frac{1}{4} - \frac{3}{4}\left(\frac{1}{2}\right)^{i+1} + \frac{1}{2}\left(\left(\frac{1}{2}\right)^{i+1}\right)^{2} \\
&= \frac{1}{2}\left(\left(\frac{1}{2}\right)^{i+1} - \frac{1}{2}\right)\left(\left(\frac{1}{2}\right)^{i+1} - 1\right) \qquad (27)
\end{aligned}
$$

The average numbers of useful transitions and of redundant transitions in S_i and C_{i+1} can now be calculated by multiplying the probabilities given in equations 23 respectively 25 and the ratios in equations 26 respectively 27 by the total number of input changes applied to the circuit.

4 Determination of transition activity using switch level simulation

To be able to determine the transition activity in larger and more complex circuits for a large number of different inputs, switch level simulation can be used. It can be viewed as falling between gate level and circuit level simulation. Circuit level simulators are very accurate, but cannot handle circuits of more than a few thousand transistors. Gate level simulators are fast, but cannot accommodate some of the most important properties of MOS circuits such as bidirectionality of transistors and charge sharing.
In switch level simulation a network description consists of interconnections of electrical components such as MOS transistors, resistors, capacitors, etc. However, simplified models of these elements are used in comparison with circuit level simulators, while a sufficient level of accuracy to represent many important characteristics of MOS digital circuits is still maintained. As a result of this simplification the amount of computation is comparable with that of gate level simulators, so that large circuits can be simulated in reasonable time.
In the work presented in this report the switch level simulator SWITCH 4.0 [2, 3] developed at Philips Research Laboratories was used. This simulator has a normal switch level simulation functionality extended with the possibility for estimating the power dissipation in circuits. Using the estimated dynamic part of this power dissipation for each input change, the number of transitions as a result of that input change can be calculated. The following sections describe how SWITCH is used to determine transition activity in circuit nodes.

4.1 Abstract modelling of circuits in SWITCH

We are interested in the dependence of transition activity on circuit architecture. The architecture of a circuit can be expressed as an interconnection of different functional blocks, for example a connection of identical full adder blocks in a ripple carry adder. Here we are not so much interested in what happens inside these functional blocks, but more in the events that take place at the interconnections between the blocks. In other words, we want to know how the interconnection of the blocks (the architecture) influences the transition activity in the circuit. This means we are interested in what happens at the gate level.
A means of using SWITCH as a gate level simulator is the use of functions to specify abstract circuit blocks. These functions can be integrated in the network, together with "real" electrical elements. The output of such a function depends on the inputs and the relationship between output and inputs can be described using boolean expressions. The input terminals of a functional block have infinite input resistance, comparable with the gate of a MOS transistor, and therefore have no influence on the other signals in the network. The output of a function is modelled as being driven by an inverter consisting

of two transistors: the *uppermodel* and the *lowermodel.* An example for this is given in figure 4 where a full adder is described. Please note that, although the output terminal is modelled as an inverter output, the state of the output is determined by the boolean expression specified for the function and not by its inverted state.

The arithmetic circuits that were simulated using SWITCH were all split up in functional

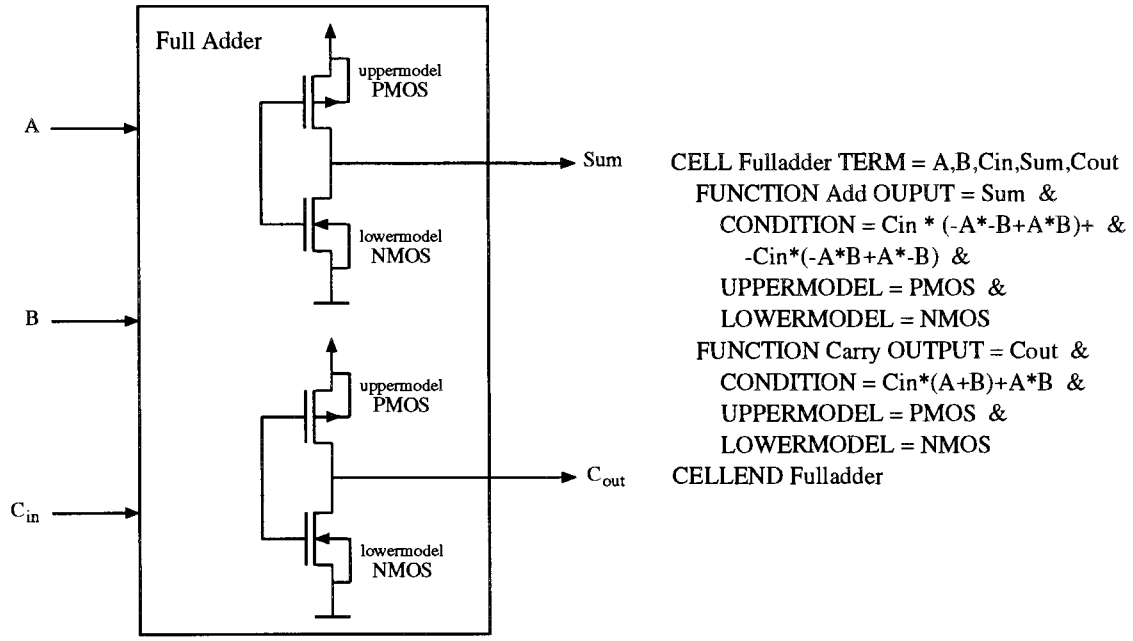


Figure 4: *Example of the use of functions in* SWITCH.

blocks such as single and-gates, full adders and multiplier cells. All these blocks were described using the function model in SWITCH. In other words, no "real" elements, such as transistors, resistors and capacitors were used. Use of these electrical elements was, at this point, not necessary as the simulation was solely focused on the determination of the transition activity of interconnection nodes between unit delay modelled functional blocks.

4.2 Power dissipation estimation in SWITCH

SWITCH 4.0 has the possibility for estimating power dissipation. To do this SWITCH uses two models [4]:

- Dynamic capacitive model.
  For the average dynamic capacitive dissipation the following formula can be derived:

$$P_{dc} = \frac{V_{dd}^2}{T} \sum_{i=1}^{n} C_i N_i \tag{28}$$

  where V_{dd} is the supply voltage, a constant for the circuit,

C_i is the nodal capacitance of node i, a constant for each node in the circuit,
N_i is the number of signal transitions *from low to high* at node i,
n is the number of nodes in the circuit,
T is the time period over which power dissipation is to be computed.

This model has some important aspects [4]. One of these aspects is relevant here. If more than one event occurs at a node at a single time point it is possible that more than one rising event will occur and therefore the power variables will be incremented more than once for that node at that time point. This is correct and is termed as *glitch* power since these events correspond to spikes or glitches in real life.
This is a very useful property of SWITCH and was used in simulations of unit delay modelled circuits to improve simulation speed by increasing the simulation time step and at the same time decreasing the number of simulation steps. This will be further explained in section 4.4.

- Short circuit model.
  The power dissipation estimated using this model is irrelevant for the determination of the transition activity, but for the sake of completeness we will state the formula for static short circuit dissipation, which is as follows:

$$P_{ssc} = V_{dd} I_{sc} \tag{29}$$

where V_{dd} is the supply voltage,
I_{sc} is the short circuit current.

4.3 Using estimated power dissipation to determine transition activity

The formula used for the calculation of the dynamic capacitive power dissipation at a single node can be written as:

$$P_{dyn} = \frac{C_{load} V_{dd}^2}{T} N_{0\to1} \tag{30}$$

Here C_{load} is the load capacitance at the node, V_{dd} is the supply voltage and $N_{0\to1}$ is the number of low to high transitions at the node over the time period T. So, using the dissipation results of the simulations we are able to calculate the number of low to high transitions at a node over the time interval T by rewriting this formula as follows:

$$N_{0\to1} = \frac{P_{dyn} T}{C_{load} V_{dd}^2} \tag{31}$$

However, to be able to determine the exact number of transitions, split up in useful and redundant transitions, we also need to know when $0 \to 1$ and $1 \to 0$ transitions occur. Figure 5 shows the same $0 \to 1$ and $1 \to 0$ transitions occuring at different times. The vertical lines in this figure denote the time moments at which new inputs are applied to the circuit. Equivalently the time between two consecutive vertical lines is equal to one clock

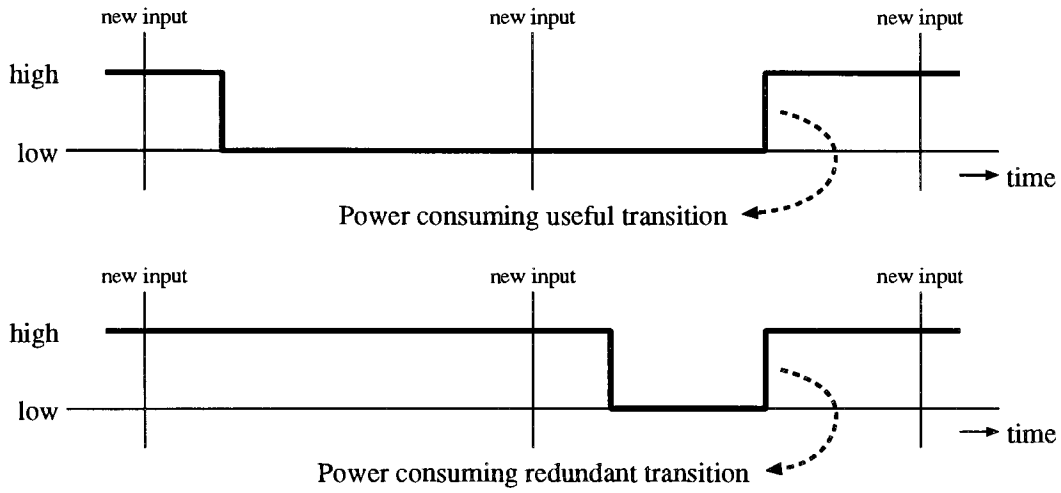


Figure 5: *Power consumption as a result of a useful and of a redundant transition.*

cycle. In both situations depicted in the figure the same amount of power is consumed in the total time period shown and therefore SWITCH will give the same power estimation for both. These estimations are proportional to the number of low to high transitions and it is not clear if these represent useful or redundant power dissipation. So we also need to know the number of high to low transitions. We can then count the total number of low to high and high to low transitions appearing in a clock period as a result of a single input change. By checking if this number is odd or even we are then able to determine the exact number of useful and redundant transitions. The remaining problem is how to make SWITCH also account for high to low transitions.

The modelling of the circuits under test, introduces another problem. As was stated before the simulated circuits are split up in functional blocks and modelled in SWITCH using functions. In a function definition there is no link made with an actual power supply. Furthermore, no electrical elements are used in a function definition. Therefore from these functional descriptions alone it is not possible to compute power dissipation and thereby the number of transitions.

To overcome both problems a special outputbuffer (see figure 6), consisting of two CMOS inverters connected to the same power supply and having equal and known load capacitances, is created for all nodes. Every buffer has a separate power supply. Using this buffer it is possible to determine not only the number of $0 \rightarrow 1$ transitions, but at the same time also the number of $1 \rightarrow 0$ transitions in each clock cycle. This is because SWITCH will now calculate the dynamic capacitive dissipation using the number of $0 \rightarrow 1$ transitions node A makes and the number of $0 \rightarrow 1$ transitions node B makes. Together these numbers equal the number of $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions the function output node under examination makes. In SWITCH power dissipation estimations can now be given for every buffer supply of every circuit node and consequently the number of transitions of every node can be

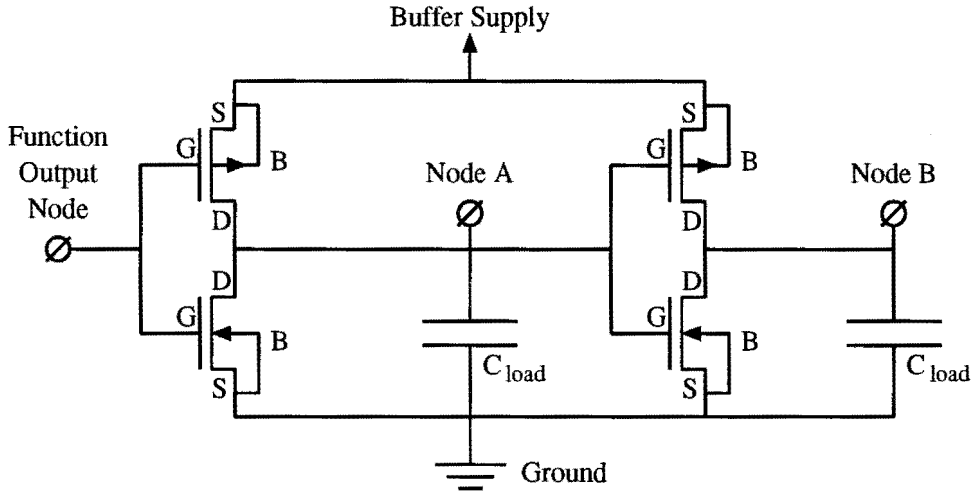


Figure 6: *Outputbuffer used in switch level simulations.*

determined.
With the total number of $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions as a result of an input change we are able to determine exactly how many useful and how many redundant transitions a node makes as a result of that input change. If begin and end state of a series of transitions a node makes as a result of a single input change are equal (for example $0 \rightarrow 1 \rightarrow 0$ or $1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1$), then all these transitions are redundant. If begin and end state of a series of transitions a node makes as a result of a single input change are different (for example $0 \rightarrow 1$ or $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$), then this node makes one useful transition. The extra transitions the node may make as a result of this input change are all redundant.

This leads to the following statements:

- The number of useful transitions as a result of a single input change is equal to 1, if the number of transitions resulting from that input change is odd, and equal to 0, if this number is even.
- The number of redundant transitions as a result of a single input change is equal to the number of transitions resulting from that input change minus one, if this number is odd, and precisely equal to this number of transitions, if it is even.

Because the $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions of a node always alternate, the total number of $0 \rightarrow 1$ transitions, resulting from a sequence of input changes, will differ from the number of $1 \rightarrow 0$ transitions by at most 1. So already for relatively small amounts of transitions these numbers will be approximately the same. This is also true for the number of redundant and the number of useful transitions, because redundant transitions always come in pairs ($0 \rightarrow 1$ followed by $1 \rightarrow 0$ or vice versa) and consequently the useful $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions always alternate. This means that the number of power consuming

useful transitions over a certain time interval can be found by dividing the total number of useful transitions appearing in that interval by a factor of two. The same holds for the number of power consuming redundant transitions which can be found by dividing the total number of redundant transitions by a factor of two.

4.4 The transition activity simulation environment

Figure 7 shows a flow chart of the way in which the transition activity in a circuit can be determined using SWITCH. The oval blocks in this flow chart represent input, output and

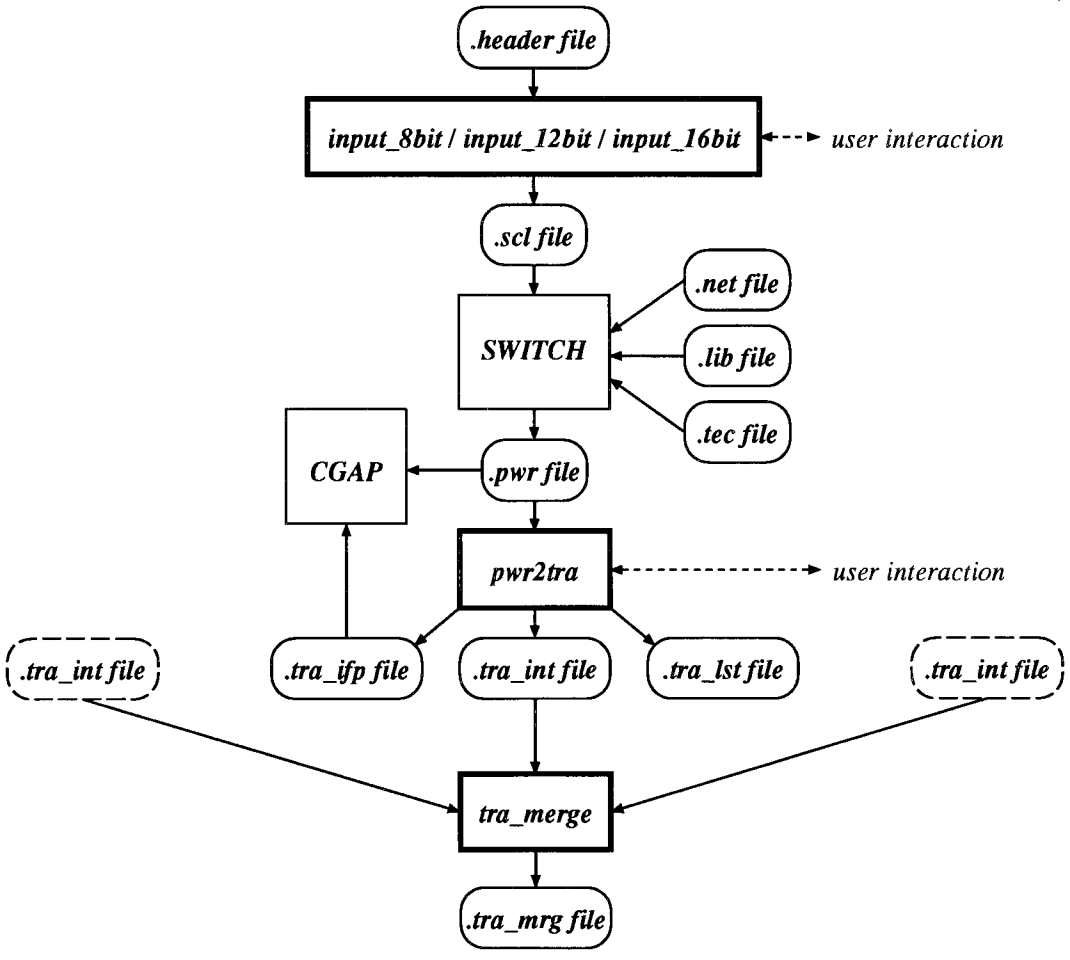


Figure 7: *Flow chart of transition activity simulation.*

control files. The thick lined square blocks are C programs which were especially written for this research project. They can be found in appendices B, C and D. The CGAP block represents a graphics package used to graphically represent the simulation results.
A special header (*.header*) file is used as an input file for the C programs *input_8bit*, *input_12bit* or *input_16bit* that generate 8 bit, 12 bit or 16 bit input stimuli. These stimuli

are appended to the header file resulting in a simulation control (*.scl*) file which is used as a control file for SWITCH. In this control file also the simulation time step must be specified. SWITCH uses a simulation method in which every circuit element internally has a unit delay [2]. So internally every element output is one time unit delayed compared to the input of the same element. This means that when unit delay modelling is used for every element in the circuit there is no need to explicitly define the delay times for these elements. It also means that the simulation time step can be made as large as the time period between consecutive input changes, because unit delay modelling is automatically preserved and all glitches appearing between consecutive simulation time points will be accounted for. Power dissipation is still accounted for all glitches and therefore all transitions appearing between consecutive input changes can still be determined. Choosing the time step as large as possible leads to a decrease in the CPU time needed for the simulations and the storage capacity needed for the result files.
Three other files are used as inputs for the SWITCH simulation package. The netlist (*.net*) file describes the architecture of the circuit. In this netlist file use can be made of user defined models that can be specified in the library (*.lib*) file. The technology (*.tec*) file specifies the transistor models used in the simulations.
The output of the SWITCH simulations is a power (*.pwr*) file, which contains the estimations of the power dissipated at every node connected to an outputbuffer for every simulated time point. This power file can be viewed graphically using the CGAP package. The C program *pwr2tra* can be used to convert this power file to a transition (*.tra*) file. Equation 31 is used in this conversion. The transition list (*.tra_lst*) file contains a list of the total number of useful transitions and the total number of redundant transitions that appeared at every node during the simulation interval. The transition intermediate (*.tra_int*) file contains this same list in another format. It can be used for summing the results from different simulations using the C program *tra_merge*. This is useful when a large circuit with a large number of internal nodes must be simulated for an extensive number of input stimuli. Such a large simulation would lead to memory problems and an enormous *.pwr* file causing storage problems. Using the *tra_merge* program large simulations can be split up in a series of smaller ones. The transition file in IFP format (*.tra_ifp*) contains the number of transitions of every node appearing at every simulation time point. It can also be viewed using the CGAP package.

5 Transition activity simulations

5.1 Simulated arithmetic circuits

The following arithmetic circuits were simulated using SWITCH to determine their transition behaviour:

- a 16 bit ripple carry adder
- an 8x8 array multiplier
- an 8x8 wallace tree multiplier
- a 16x16 array multiplier
- a 16x16 wallace tree multiplier

The ripple carry adder was simulated to verify the simulation method with the probability calculation results given in chapter 3. The structure of a 16 bit ripple carry adder is given in figure 8.

The multiplier circuits were simulated to obtain quantitative results for the influence of

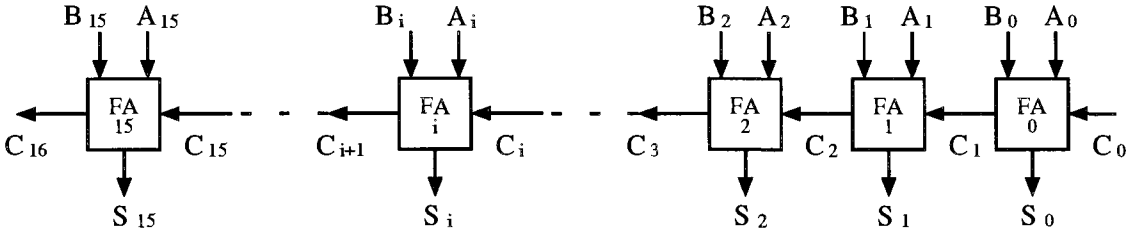


Figure 8: *Basic structure of a 16 bit ripple carry adder.*

unbalanced delay paths on transition activity. Multiplier circuits were used, because of their functional properties, which make it easy to introduce more or less unbalanced delay paths in the architecture, by choosing for example as was done here, between an array architecture and a wallace tree architecture. Furthermore multiplier circuits are simple to construct. To be able to accurately model unbalanced delay paths, no pipelines were used in the multipliers.

The structure of an 8 bit array multiplier is given in figures 9 and 10. The basic structure of an 8 bit wallace tree multiplier is depicted in figure 11. This figure also shows the addition scheme used in this type of multiplier. The exact structure of the 8 bit wallace tree multiplier is given in figure 12. Please note that the multiplier depicted here is only suitable for multiplying positive numbers. However, the extension to negative numbers will lead to similar results. Also note that this a non-optimized version. Full adders with one or two constant zero inputs can be replaced by half adders or simple connections. This will

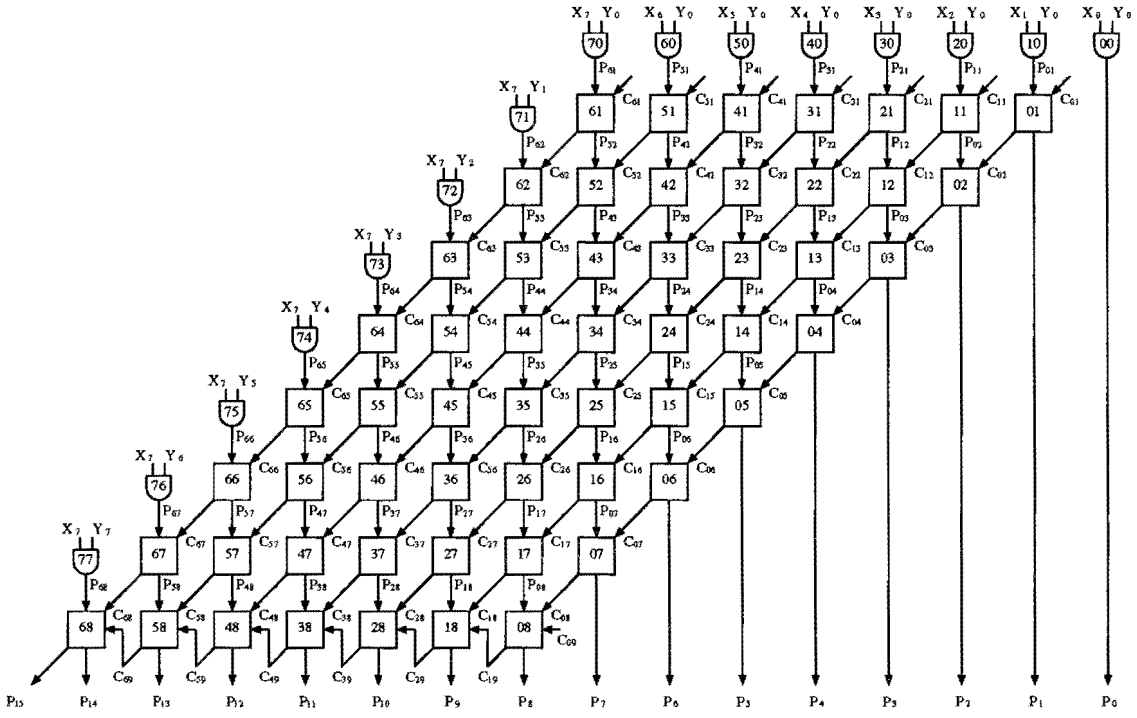


Figure 9: *Basic structure of an 8 bit array multiplier for positive numbers.*

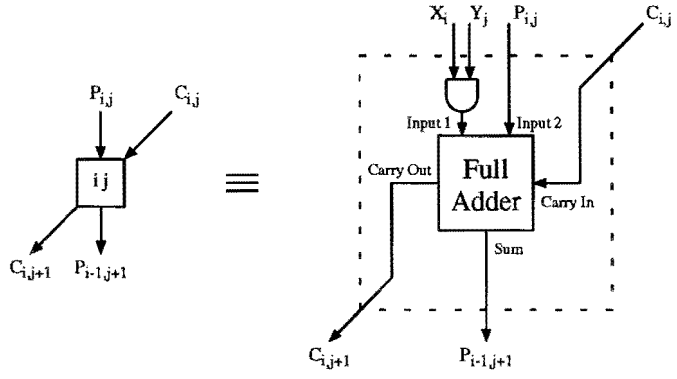


Figure 10: *Structure of an array multiplier cell.*

result in less adder blocks and less interconnections.
The 16 bit array and wallace multipliers have structures similar to their 8 bit variants.

5.2 Simulations using unit delay modelling

As can be seen from the figures presented in the previous section all the arithmetic circuits presented here consist of some type of network made up of full adder blocks. In a first simulation all the full adders and multiplier cells in the circuits were modelled as unit delay

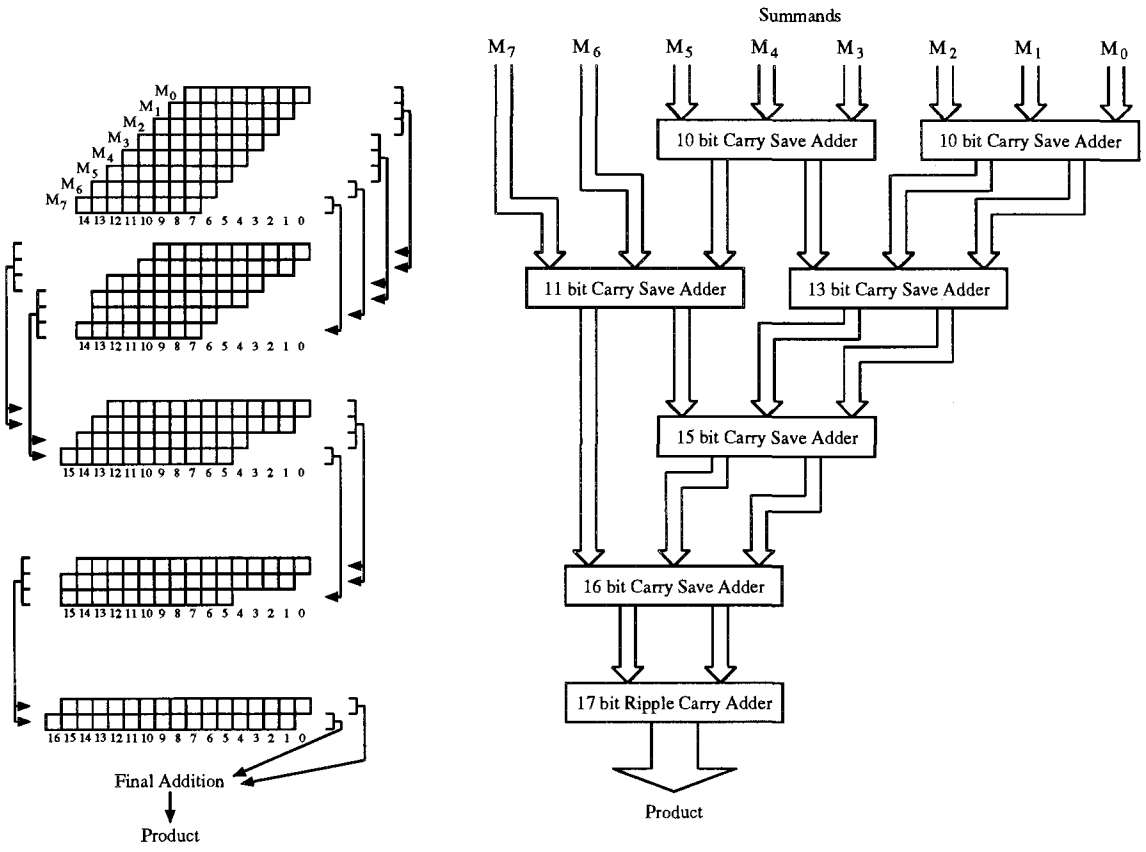


Figure 11: *Addition scheme and basic structure of an 8 bit wallace tree multiplier.*

blocks, having equal delays for both sum, product and carry. The models used in SWITCH are shown in figure 13. The dotted squares in this figure denote the different cell blocks defined in the SWITCH library file. As can be seen from the models, cell blocks can be nested. The full adder cell and multiplier cell shown, each constitute a single element in the circuit.

As was stated in section 4.4 there is no need to explicitly define the delay times for the circuit elements when unit delay modelling is used. The simulation time step can be made as large as the time period between two consecutive input changes, because unit delay modelling is automatically preserved and all glitches appearing between consecutive simulation time points will be accounted for.

In the simulations use was made of random inputs. This type of inputs is a realistic choice. This is because in a practical situation adders and multipliers will often be used in a time multiplexed environment. The original signal statistics will be lost due to this multiplexing and random like signals will arrive at the inputs of the arithmetic circuits.

First a 16 bit ripple carry adder was simulated for a large series of random inputs. This was done to check whether the simulation method is in correspondence with the calculations in sections 3.3 and 3.4. The results of this simulation are shown in table 2 together with the

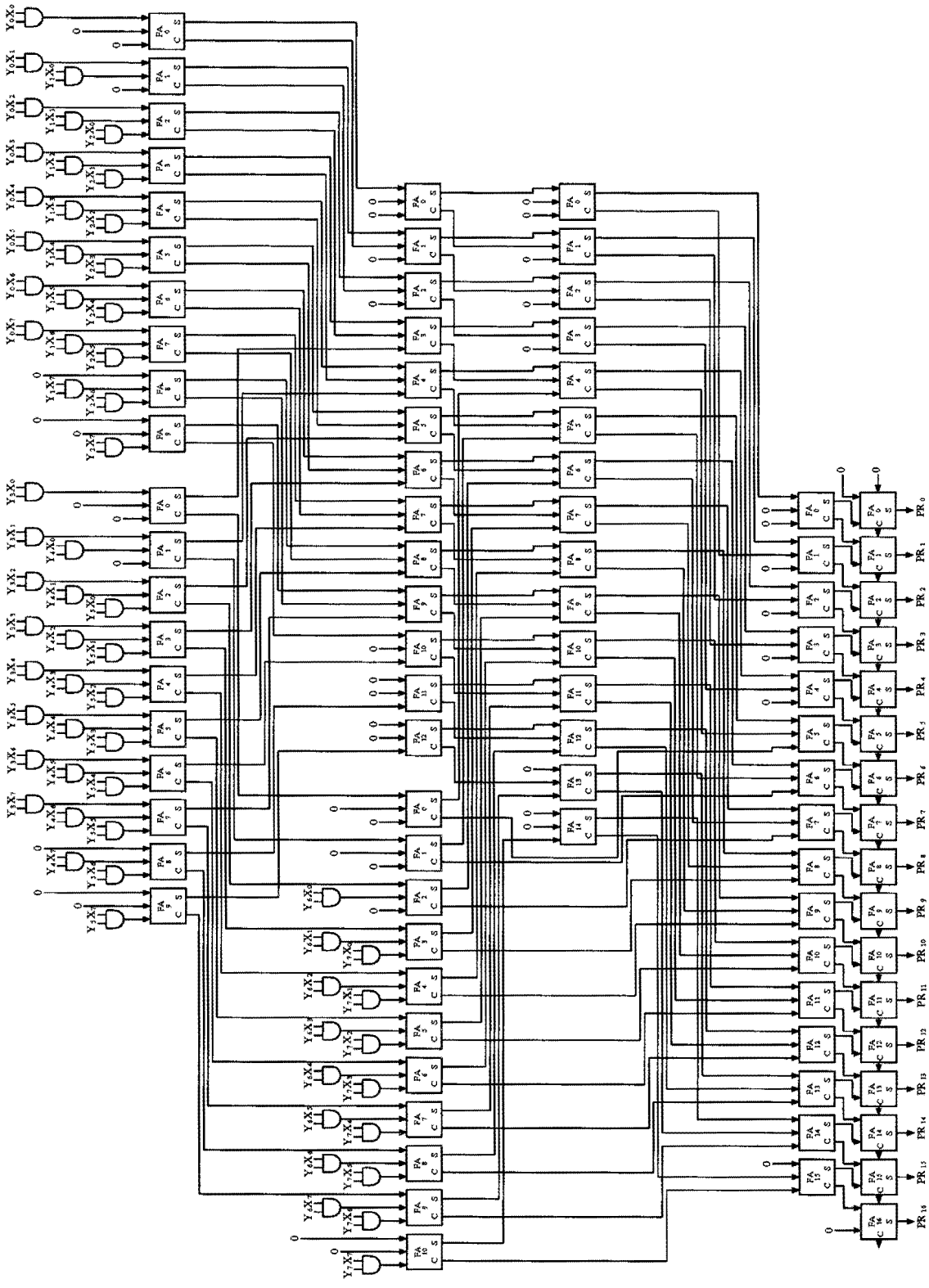


Figure 12: *Structure of an 8 bit wallace tree multiplier.*

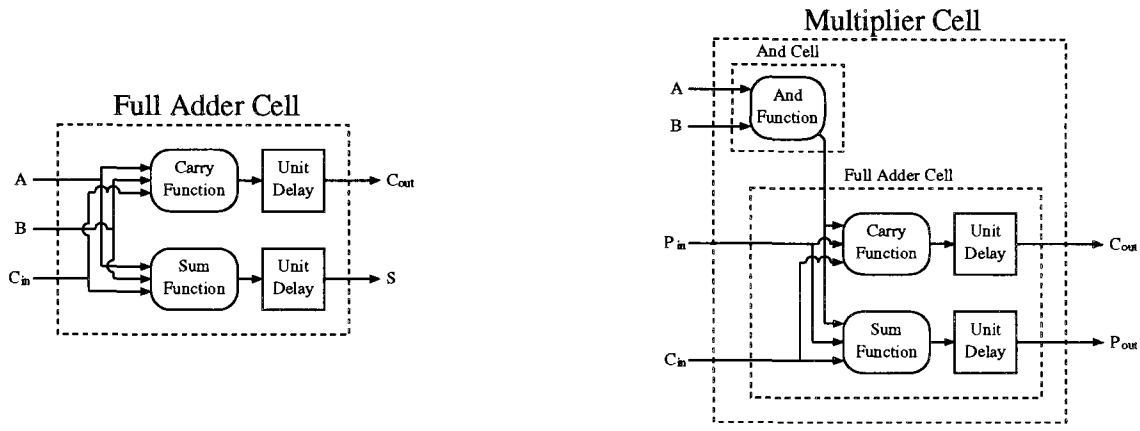


Figure 13: *Balanced delay models of full adder and multiplier cells used in* SWITCH.

calculated number of transitions appearing at the different nodes in the ripple carry adder. In the calculations use was made of equations 21, 23, 26, 20, 25 and 27. As can be seen from the table the differences between the calculated results and the simulated results are small. Therefore it can be concluded that the simulation method is correct.
In appendix A the contents of the *tra_ifp* file for the 16 bit ripple carry adder are graphically displayed. Here the transition activity in sum bit 9 and carry bit 12 of the adder appearing in every clock cycle are displayed for 50 random inputs. From this figure it is clear that the simulation method in effect consists of the counting of transitions in every clock cycle and subdividing these transitions in useful and redundant transitions.

The goal of the next simulations was to arrive at quantitative results for the dependence of transition activity on the used architecture of a typical combinational circuit. For these simulations use was made of two different multiplier architectures. The simulation results for the different multipliers are given in table 3. From this table we see that the wallace tree multiplier, which has far more balanced delay paths, has far less redundant transitions and a far better redundant/useful ratio than an array multiplier. Therefore it is clear that decreasing the number of unbalanced delay paths reduces the number of redundant transitions.
We also see from this table that in a 16x16 array multiplier the redundant/useful ratio *increases* by a factor of more than two, compared to an 8x8 array multiplier. This is due to the large increase in logic depth and delay unbalance in the 16x16 array multiplier. However, in a 16x16 wallace tree multiplier the redundant/useful ratio *decreases* by a factor of almost two, compared to an 8x8 wallace tree multiplier. Figure 14 shows the basic structure of the 16x16 wallace tree multiplier. From this figure it is clear that the logic depth increases only by a small amount compared to the 8x8 wallace tree multiplier. Furthermore the 16x16 wallace tree multiplier appears to have a better delay balance than the 8x8 wallace tree multiplier. Therefore the relative increase in the number of redundant transitions for the 16x16 wallace tree will be smaller than the relative increase in the number of useful transitions, caused by the increased number of gates in the multiplier. This

	Calculated results			*Simulated results*			*Difference (%)*		
node	*total*	*useful*	*redundant*	*total*	*useful*	*redundant*	*total*	*useful*	*redundant*
S_0	2000	2000	0	1948	1948	0	-2,6	-2,6	0,0
S_1	3500	2000	1500	3481	1903	1578	-0,5	-4,9	+5,2
S_2	4250	2000	2250	4190	2024	2166	-1,4	+1,2	-3,7
S_3	4625	2000	2625	4624	1970	2654	0,0	-1,5	+1,1
S_4	4813	2000	2813	4848	2020	2828	+0,7	+1,0	+0,5
S_5	4906	2000	2906	4949	2017	2932	+0,9	+0,9	+0,9
S_6	4953	2000	2953	5038	1952	3086	+1,7	-2,4	+4,5
S_7	4977	2000	2977	5032	2020	3012	+1,1	+1,0	+1,2
S_8	4988	2000	2988	5037	2013	3024	+1,0	+0,7	+1,2
S_9	4994	2000	2994	5034	1994	3040	+0,8	-0,3	+1,5
S_{10}	4997	2000	2997	5024	2000	3024	+0,5	0,0	+0,9
S_{11}	4999	2000	2999	4939	2013	2926	-1,2	+0,7	-2,4
S_{12}	4999	2000	2999	4990	2006	2984	-0,2	+0,3	-0,5
S_{13}	5000	2000	3000	4956	1968	2988	-0,9	-1,6	-0,4
S_{14}	5000	2000	3000	4948	2016	2932	-1,0	+0,8	-2,3
S_{15}	5000	2000	3000	4956	2058	2898	-0,9	+2,9	-3,4
C_1	1500	1500	0	1466	1466	0	-2,3	-2,3	0,0
C_2	2250	1875	375	2209	1793	416	-1,8	+4,3	+10,9
C_3	2625	1969	656	2603	1995	608	-0,8	+1,3	-7,3
C_4	2813	1992	820	2827	1969	858	+0,5	-1,2	+4,6
C_5	2906	1998	908	2941	1973	968	+1,2	-1,3	+6,6
C_6	2953	2000	954	2975	1975	1000	+0,7	-1,3	+4,8
C_7	2977	2000	977	3023	1965	1058	+1,5	-1,8	+8,3
C_8	2988	2000	988	3040	2012	1028	+1,7	+0,6	+4,0
C_9	2994	2000	994	3049	2003	1046	+1,8	+0,2	+5,2
C_{10}	2997	2000	997	3027	1991	1036	+1,0	-0,5	+3,9
C_{11}	2999	2000	999	2963	1975	988	-1,2	-1,3	-1,1
C_{12}	2999	2000	999	2991	2015	976	-0,3	+0,8	-2,3
C_{13}	3000	2000	1000	2982	1956	1026	-0,6	-2,2	+2,6
C_{14}	3000	2000	1000	2978	2012	966	-0,7	+0,6	-3,4
C_{15}	3000	2000	1000	2946	1982	964	-1,8	-0,9	-3,6
C_{16}	3000	2000	1000	2982	2016	966	-0,6	+0,8	-3,4
total	119002	63334	55668	118996	63020	55976	0,0	0,0	+0,1

Table 2: *Calculated and simulated numbers of transitions resulting from 4000 random input changes*

type of circuit	*total amount of transitions*	*useful transitions*	*redundant transitions*	*ratio redundant/useful*
8x8 array multiplier	58858	23418	35440	1.51
8x8 wallace tree multiplier	50824	39608	11216	0.28
16x16 array multiplier	438575	102845	335730	3.26
16x16 wallace tree multiplier	200380	173330	27050	0.16

Table 3: *Results from* SWITCH *simulations using 500 random input changes*

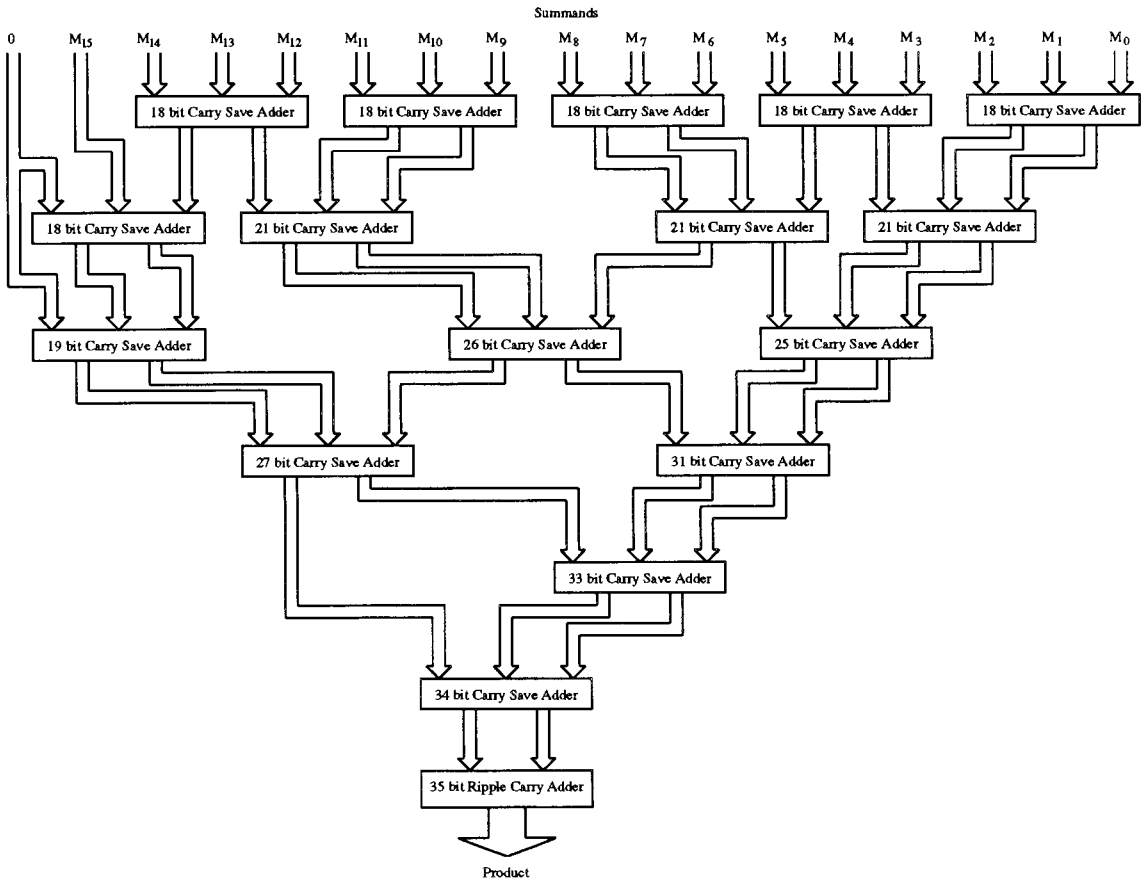


Figure 14: *Basic structure of a 16 bit wallace tree multiplier.*

explains the decrease in the redundant/useful ratio for the 16x16 wallace tree multiplier. There is also a considerable difference in the total number of transitions between the array architecture and the wallace tree architecture. For the 8 bit multipliers this difference is much smaller than for the 16 bit multipliers. Therefore it appears that especially for multipliers with large bitwidths the wallace tree architecture leads to far less transitions than the array architecture.

Note that the number of useful transitions of the wallace tree multiplier is larger than that of the array multiplier. In other words the average number of necessary transitions needed to perform a multiplication is larger for the wallace tree multiplier. This number can be reduced by optimizing the wallace tree multiplier, for example by removing full adders with one or more constant-zero inputs. This was not done here, because the multiplier was only used as a vehicle. The design of a multiplier with minimum power dissipation was not the goal here.
The actual power dissipation of a circuit is dependent on the nodal capacitances in the circuit layout. Therefore at this point no *exact* figure can be given on the amount of power that can be saved when choosing a different architecture. Still, however, table 3 gives a good idea of what can be achieved.

5.3 Simulations using unequal sum and carry delays

A second simulation of the 8x8 array multiplier and the 8x8 wallace tree multiplier was carried out in which the delay path for the sum generation in every full adder was chosen twice as large as the delay path for the carry generation. This leads to a more realistic modelling of a commonly used full adder. Figure 15 shows a graphical representation of the models used in SWITCH. In these non-unit delay simulations the simulation time step

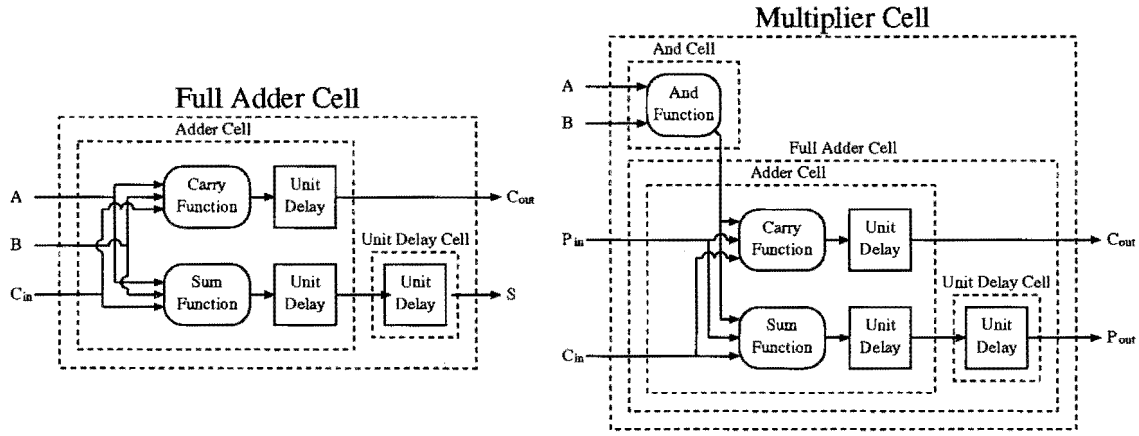


Figure 15: *Unbalanced delay models of full adder and multiplier cells used in* SWITCH.

must be made equal to the smallest delay in the circuit, because if a larger time step is chosen SWITCH will round all transition events (including glitches) appearing in a certain simulation interval to the same simulation time point leading again to an unwanted unit delay simulation.
The expected result of this unbalanced sum and carry delay modelling is that more redundant transitions appear because the unbalance of delay paths is increased, while the number of useful transitions stays the same. The simulation results come up to these expectations as can be seen from table 4.

8x8 array multiplier	*useful transitions*	*redundant transitions*	*ratio redundant/useful*
$\delta_{sum} = \delta_{carry}$	23552	34346	1.46
$\delta_{sum} = 2 \times \delta_{carry}$	23552	47340	2.01
8x8 wallace tree multiplier	*useful transitions*	*redundant transitions*	*ratio redundant/useful*
$\delta_{sum} = \delta_{carry}$	38786	11274	0.29
$\delta_{sum} = 2 \times \delta_{carry}$	38786	24762	0.64

Table 4: *Results from* SWITCH *simulations using 500 random input changes*

5.4 Simulations of a PHIDEO processing unit

Transition activity simulations were also carried out on a processing unit in PHIDEO, known as a *direction detector*. A block diagram of a direction detector is shown in figure 16. The

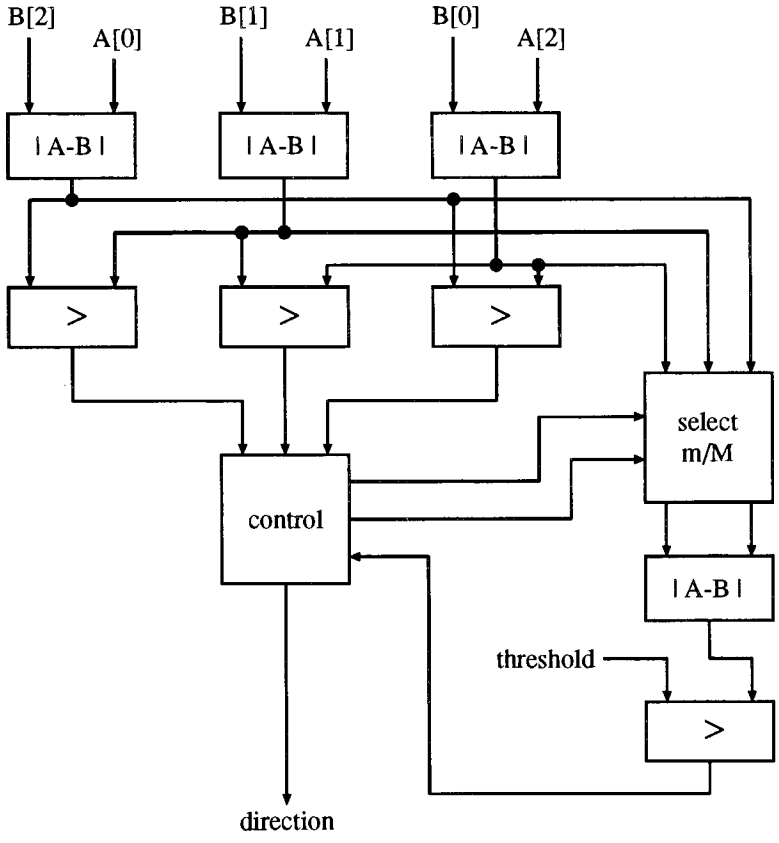


Figure 16: *Block diagram of a direction detector.*

direction detector is used in the implementation of a so called *Progressive Scan Conversion (PSC)* algorithm [5]. The function of this algorithm is to make good estimations of the values of pixels in the 'missing' lines of each field of a 625 line, 2:1 interlaced, 50Hz signal and to combine the input and estimated pixels to produce a 625 line, 1:1 non-interlaced (progressive), 50Hz signal. 25Hz flicker at the horizontal edges of an object is removed by the algorithm. Serration of moving edges and line crawl of fine-line structures are also removed. The method used to estimate the value of a pixel is depicted in figure 17. An interpolation is performed along the direction of an edge, either (a2,b0), (a1,b1)

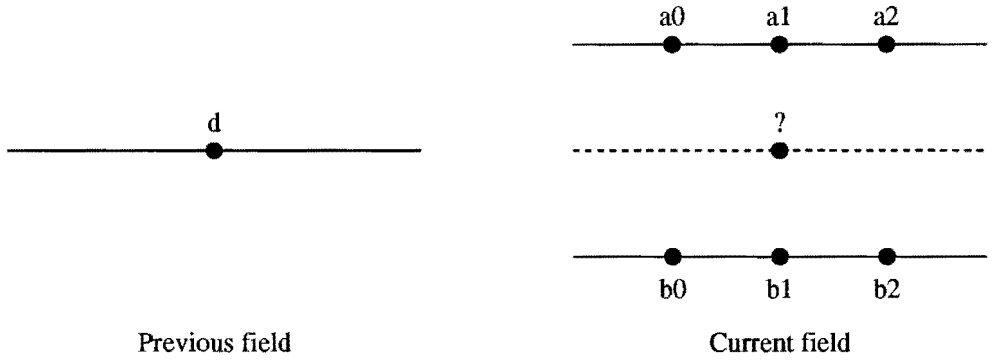


Figure 17: *Progressive Scan Conversion.*

or (a0,b2). The direction of the edge can be estimated by determining the direction of minimum gradient in the current field at the location of the pixel to be estimated. The direction detector is used for this. It calculates the direction of minimum gradient by calculating the three absolute differences of pairs of opposing pixels in the current field and selecting the minimum and maximum of these. If the minimum differs from the maximum by more than a certain threshold then the direction detector outputs the edge with the minimum difference, otherwise a default vertical edge is chosen.

Figure 18 shows the way in which the direction detector is constructed using functional blocks in SWITCH. Note that the two dotted registers do not belong to the actual direction detector, but are used as input registers. Transitions in the outputs of these registers are viewed as the actual input changes of the direction detector. They are not taken into account in the determination of the transition activity of the direction detector, because they do not contribute directly to the actual power dissipation of this direction detector.
The circuit was simulated using 4320 random inputs to obtain the transition activity for the various circuit nodes connecting the different circuit blocks. The simulation results for these nodes are presented in appendix H. For the circuit as a whole the results are as follows:

- Total number of useful transitions: 272842,
- Total number of redundant transitions: 1033970,

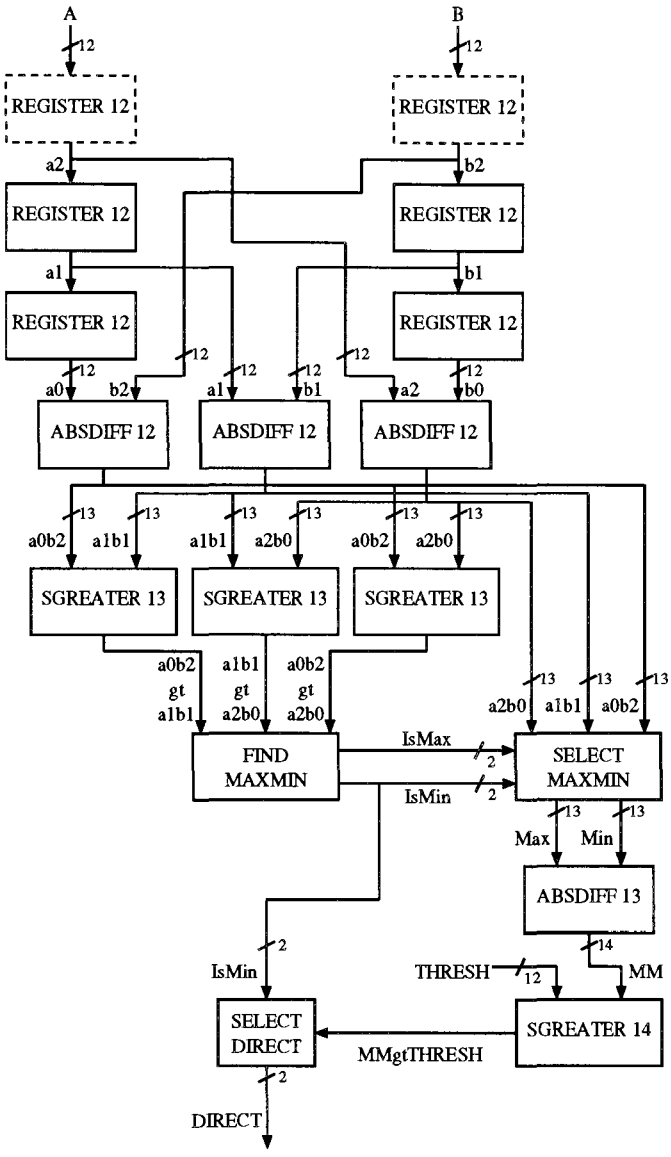


Figure 18: *Direction detector constructed using functional blocks in* SWITCH.

- Ratio $\frac{redundant\ transitions}{useful\ transitions}$: 3.79

So for every useful transition there exist 3.79 redundant transitions. Therefore, in theory, for random inputs transition activity in the combinational logic could be reduced with a factor of 1 + 3.79 = 4.79 if all delay paths were balanced.

To investigate the transition activity in a real life situation the direction detector was also simulated in SWITCH using real video signals. The correlation between consecutive samples of these signals leads to the expectation that overall less transitions will appear in the circuit when these signals are used as inputs instead of random inputs.

An 8 bit 720x288 picture frame was used for this simulation. By using only the even lines of this picture frame, it is handled as if it was part of a 2:1 interlaced field on which the progressive scan algorithm is used. The 8 bit samples were converted to 12 bit input samples for the direction detector by using bit extension on the least significant bit side. The large amount of input samples, equal to $\frac{720 \times 288}{2} = 103680$, cannot be handled by SWITCH as this would lead to memory and disk storage problems. Therefore the simulation was split up in 24 separate simulations using 4320 inputs each. The *tra_merge* program, mentioned in section 4.4 was used to merge the results of these 24 simulations. The results for the internal circuit nodes can be found in appendix I. For the total circuit the results are as follows:

- Total number of useful transitions: 5057396,
- Total number of redundant transitions: 21103924,
- Ratio $\frac{redundant\ transitions}{useful\ transitions}$: 4.17

Dividing these numbers by 24 and comparing them to the random input results shows that the total number of transitions has indeed decreased. However, it is found that a factor of $1 + 4.17 = 5.17$ in transition activity reduction can in theory be achieved if all delay paths are balanced. This figure of 5.17 is close to the 4.79 figure for random inputs. Therefore it appears that simulations using random inputs give a good approximation for real life use of the direction detector. A possible explanation for this is that the correlation of the video signals is already lost at an early stage in the direction detector, namely at the outputs of the *ABSDIFF12* modules. These modules each perform a subtraction of two different video samples and take the absolute value of the result. In this way most of the signal correlation will be lost, resulting in a random like *ABSDIFF12* output signal.

6 Power dissipation estimation on layout extractions

In the previous sections an analysis on power dissipation in combinational logic circuits was carried out by looking at the number of useful and redundant transitions appearing at different nodes in the circuit. In this analysis use was made of functional descriptions of circuit blocks. Therefore no real capacitances could be introduced in the circuits and consequently no real power dissipation could be estimated.
To be able to get accurate power estimation results simulations were carried out on extracted netlists of real circuit layouts. A direction detector, described in section 5.4 was used as the subject in these simulations.

The goal of the simulations on the direction detector consists of three parts:

- Analysis of power dissipation split up in three components:
  1. dissipation in the combinational logic,
  2. dissipation in the (pipelining) flipflops,
  3. dissipation in the clock line.
- Investigation of the influence of retiming,
- Investigation of the influence of architectural choices.

Before the results of the actual simulations on the layout extractions are presented in the next chapters, some information on retiming, layout generation and circuit simulation will be given in the rest of this chapter.

6.1 Retiming

Retiming is a technique for achieving the required throughput of clocked digital circuits whilst maintaining the functional behaviour of these circuits. This is achieved by adding, deleting or repositioning flipflops in a circuit until the desired specification is met. Specifications can be given for area and/or speed optimization. Using retiming it may be possible to increase the clock frequency or reduce the area.
Since the typical throughput of a direction detector is 27MHz, retiming is used. By inserting, moving and/or removing flipflops retiming can change the balance in the delays of different paths in the circuit. Therefore more or less redundant transitions will occur in the combinational logic of the circuit and therefore more or less power will be consumed in this combinational part. This is depicted in figure 19. However retiming a circuit with a large delay unbalance in its signal paths can result in a large amount of extra flipflops inserted in the circuit and consequently in an increase of the loading capacitance of the clock. Extensive retiming or pipelining can therefore lead to a significant increase in the power dissipated in flipflops and clock lines.

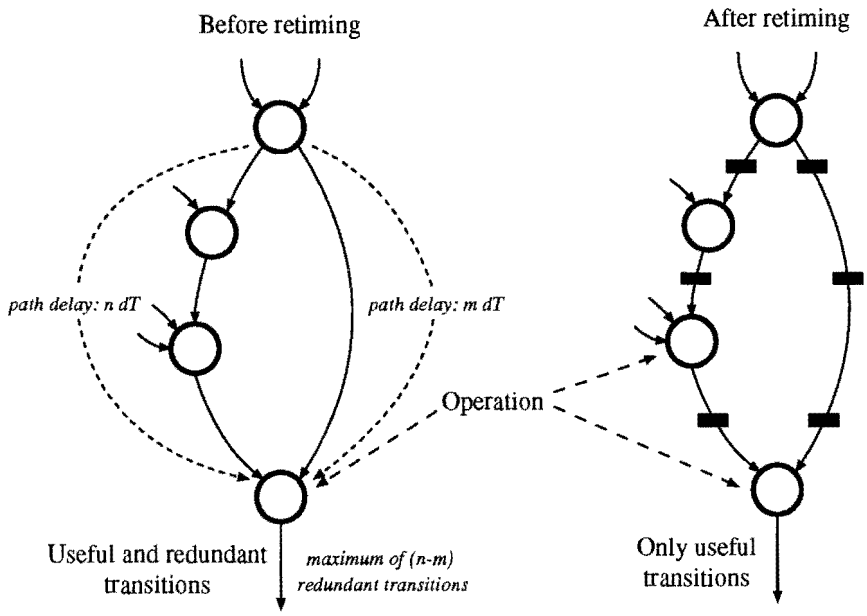


Figure 19: *Retiming eliminates redundant transitions.*

Current retiming techniques are mainly focused on circuit speed and area and less on power dissipation. However, a retiming solution which is optimal as far as area and speed are concerned maybe less of an optimal solution for power dissipation. This can be seen from figure 20 where the left solution with only one inserted register is best for area, while the solution to the right with two registers and consequently more area may be better for power. That is, of course, if the power saved due to the reduction of redundant transitions is larger than the extra power dissipated in the extra flipflop and clock lines. This phenomenon was not thoroughly investigated in this work, but can be of great importance for the improvement of retiming techniques for power minimization in the future.

Retiming usually results in the pipelining of circuit paths. This pipelining causes a certain timing relation between different terminals. Constraints on these timing relations can be given to the retimer to ensure that the proper operation of the circuit is maintained. The timing relation between different terminals of a circuit can be represented in the form of a *time shape* diagram. It defines the skew (in terms of clock cycles) of inputs and outputs relative to some reference point T_{ref} which is called the *start time* of the circuit. Figure 21 shows two possible time shapes. In the left figure the input presented in every clock period is already represented in the output in the same clock period. In the right figure a new input only has influence on an output after two clock periods. Of course many different time shapes are possible.

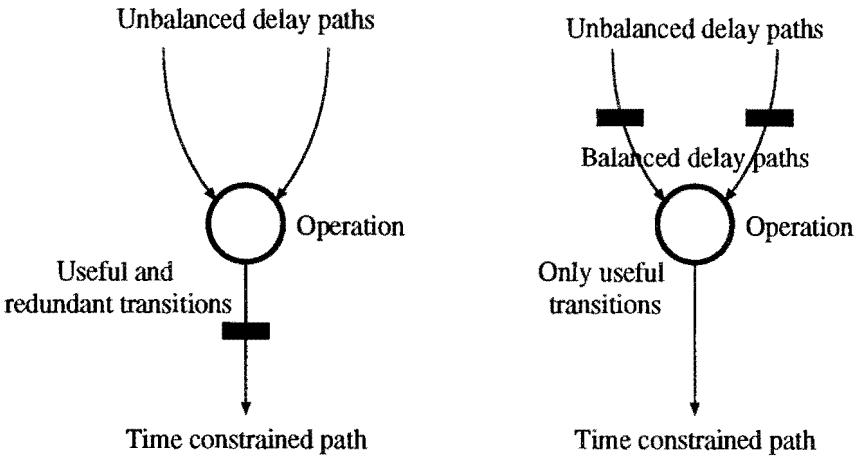


Figure 20: *Two retiming solutions; left figure: optimum area solution; right figure: optimum power solution.*

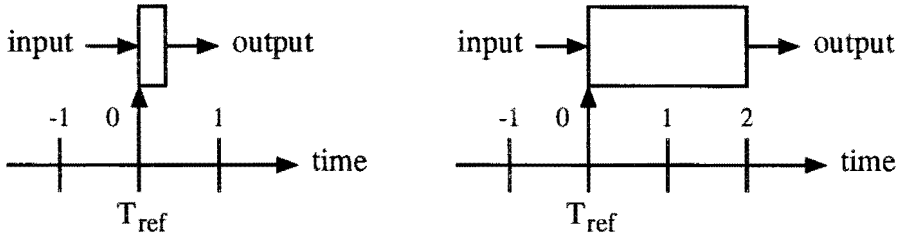


Figure 21: *Time shape diagrams.*

6.2 Layout generation and layout extraction

Circuit layouts can be generated starting from a VHDL (VHSIC Description Language) description of the circuit structure and/or operation. To check the correctness of a VHDL circuit description, use can be made of the VANTAGE VHDL simulation tool. This is depicted in figure 22. To be able to do this a special VHDL file, known as the *testbench* is created, which supplies the proper inputs and checks for the correctness of the outputs of the VHDL circuit description under test. The correct inputs and outputs used in the test of the circuit are generated using a C program describing the functions performed by the circuit under test. The actual testing can then be performed using the VANTAGE simulation package.

When the VHDL description is found to be correct, the circuit layout can be generated (see figure 23). Various operations are performed on the circuit description in VHDL using the shellscript *vhdl2x*. These operations are: inclusion of standard module descriptions from the VHDL library using CPP (C language PreProcessor), parsing of the VHDL file using VSYN (VHDL Synthesis system), optimizing and matching using OMA (Optimizer and MAtcher), hierarchy removal or flattening, redundancy removal or shrinking and re-

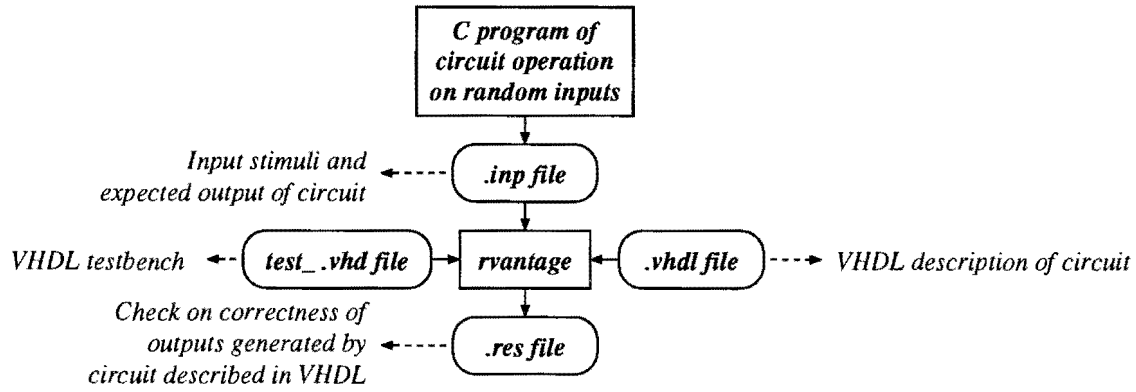


Figure 22: *Check on the correctness of the VHDL description.*

timing using RETLAB. When retiming is performed it is also checked whether gates in the netlist have correct fan out or drive capabilities and corrections are made if necessary. The result of these operations is an ndl-netlist of the desired circuit. Using the *vhdl2x* shell script a scan chain can be created on the chip. This was not done here and therefore the script will not automatically generate a netlist in the so called x-format, but will halt after retiming is performed and an ndl-netlist is issued as output. The actual conversion to an x-netlist must therefore be performed using the shell script *ndl2x* which converts an ndl-netlist generated by the retimer to an x-netlist.
From the x-netlist a standard cell layout is created using the shellscript *x2layout*. This script also generates a *specification* file that can be used to control layout extraction. The specification file contains the coordinates of input and output terminals on the layout and some extraction control parameters.

Layout extraction, performed by the extraction tool LOCAL45 which is called using the shellscript *local*, results in a transistor netlist file including capacitances. This is depicted in figure 24. The netlist file can be generated in various formats, including those of SWITCH and the circuit simulation package PSTAR, depending on the LOCAL45 application file used for the extraction. However, because no LOCAL45 application file to generate SWITCH netlists including capacitances for the more recent layout technologies is available, an extraction in the format of the switch level simulator LSIM, for which these application files do exist, is performed first. The LSIM netlist file is converted to a SWITCH netlist file using the shellscript *lsim2switch*. Also a layout extraction using a capacitance application file for LOCAL45 can be performed to determine in tabular format the total load capacitances of the various input and output terminals of the circuit layout. This can be used, for instance, to determine the total clock load capacitance.

6.3 Verification of the extracted netlist using switch level simulation

To check the correct functional behaviour of the generated circuit layout, switch level simulations can be carried out on the extracted netlists. For this purpose the switch level

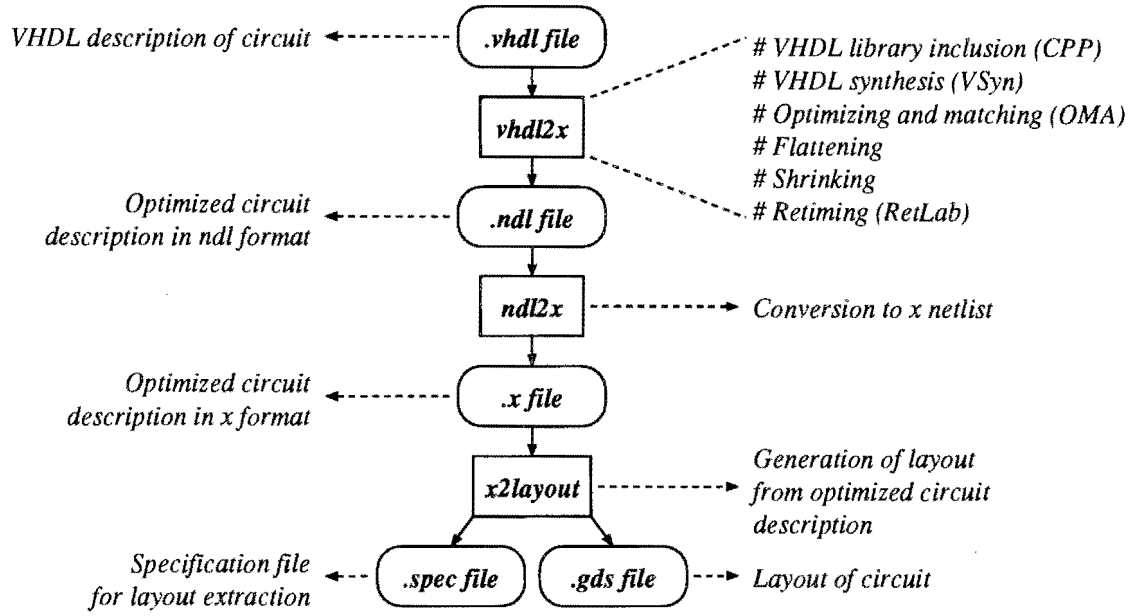


Figure 23: *Layout generation from VHDL description.*

simulator SWITCH can be used. A C program implementing the functional behaviour of the circuit can be used to generate inputs and the corresponding correct outputs. The inputs can be supplied as stimuli for the switch level simulation. The results of the simulation can be compared to the expected outputs generated by the C program.
However, switch level simulation and particularly SWITCH is less suitable to accurately estimate power dissipation from layout extracted netlists. Although the extracted capacitors in the netlist are used in the power estimations to account for the influence of transitions in a node with a certain loading capacitance, they are not used in the determination of nodal delays. In other words, no delay is calculated by SWITCH from the RC-times, present in the circuit due to transistor impedance and capacitances. Thus different delay paths in the circuit are not accurately modelled by SWITCH and transitions due to delay unbalance are not accounted for in the right way. Because no other switch level simulation package with the possibility to accurately model circuit delays from nodal capacitances and also estimate power dissipation was currently available, use was made of circuit level simulation.

6.4 Using PSTAR to estimate power dissipation

The circuit level simulation package PSTAR can be used to accurately estimate power dissipation in circuits with up to about ten thousand transistors. The major disadvantage of using circuit level simulation on rather large circuits is the long simulation time needed. These long simulation times put a bound on the amount of different input stimuli the circuit can be simulated for. When circuits are to be simulated for large amounts of random inputs this is major setback. However, in this research still reliable results could be obtained for

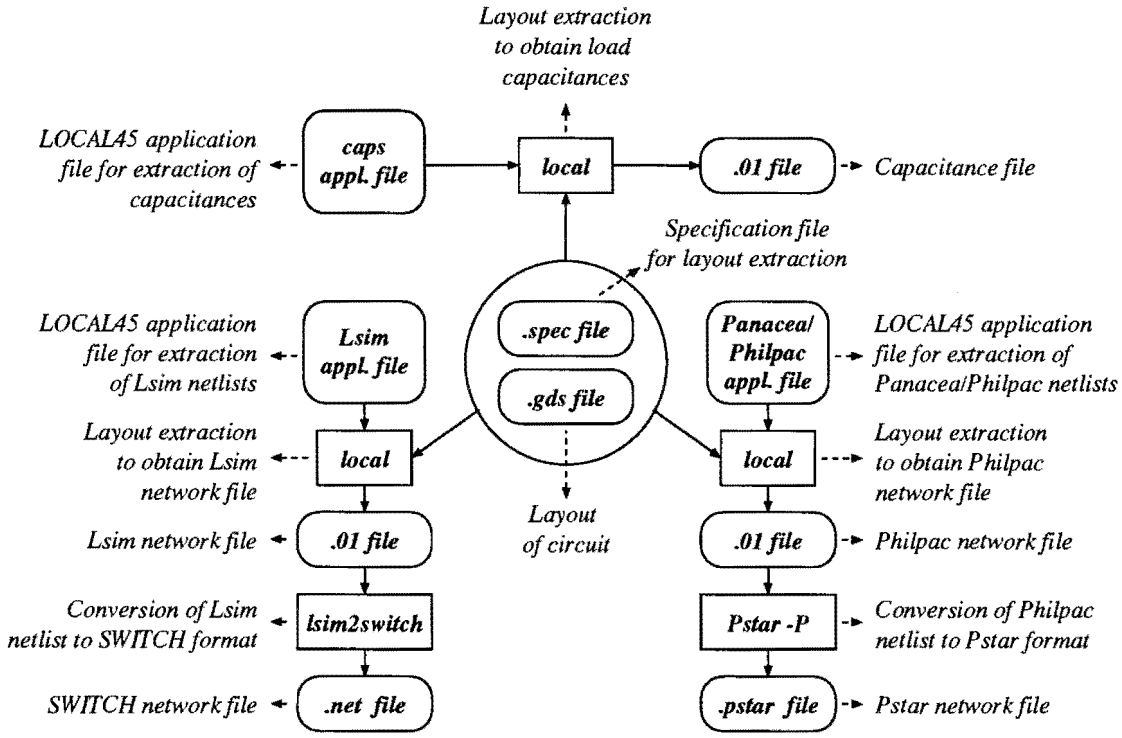


Figure 24: *Layout extraction using* LOCAL45.

relatively few input stimuli.

A major problem of performing PSTAR simulations on large circuit netlists is the possibility of non-convergence of the simulator output. This is mainly caused by the possibility of topological floating nodes in the extracted netlists. These floating nodes, which are only connected to capacitances, appear in the netlist due to the occurrence of unused feed cells in the standard cell layout on which extraction is performed. This phenomenon is depicted in figure 25. The feed cells are normally used to connect adjacent routing channels in a standard cell layout. However, remaining unused they form unconnected metal wire pieces on the chip. These loose metal wires form loose capacitances in the circuit that have no effect on the circuit operation, but in the extracted netlist they are a major convergence barrier for the simulation package, because the voltages on the floating nodes they are connected with are undefined. The simulator keeps on searching for a solution for the voltage on these nodes, finally leading to no convergence, because no converging solution can be found. To overcome these convergence problems all loose capacitors must be removed from the netlist before the simulations are carried out.

Furthermore, when performing a transient analysis on the extracted netlists in PSTAR no implicit DC analysis [7] is performed as this can also lead to convergence problems.

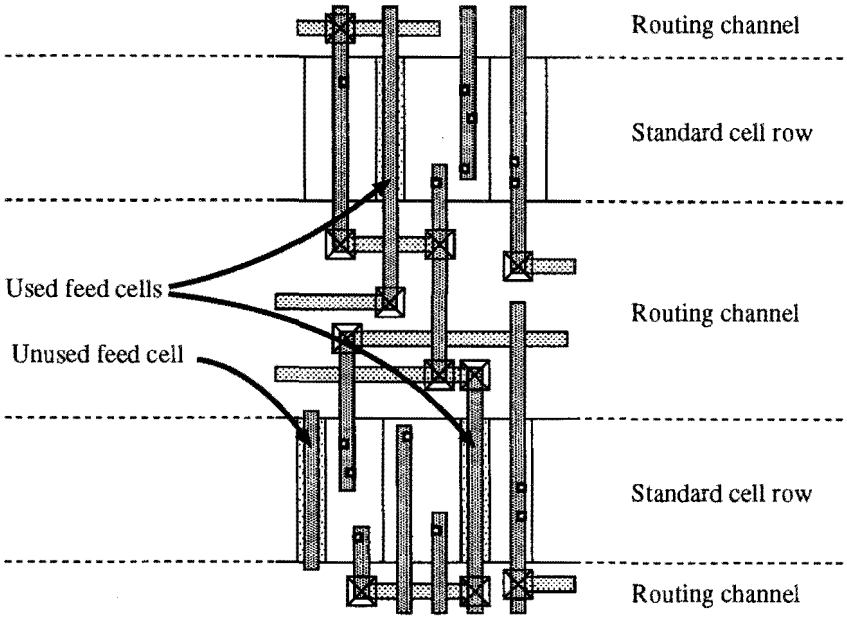


Figure 25: *Unused feed cell in circuit layout.*

7 Power dissipation analysis for the direction detector

7.1 Direction detector layouts

Four different layouts in C150DM technology (a 0.8 micron, 5V process) were generated for the direction detector. These layouts were all generated starting from the same VHDL description shown in appendix E, on which a correctness test was carried out using the VANTAGE simulation package and the VHDL workbench description shown in appendix F. This correctness test was carried out using inputs and reference outputs generated by the direction detector program shown in appendix G, which was written in the C programming language. The layouts were then generated using the shell scripts *vhdl2x*, *ndl2x* and *x2layout* as described in section 6.2.
The differences between the four generated layouts consist of the way in which retiming was performed. One circuit was retimed for a clock period of 200ns, equivalent to a 5MHz clock frequency. This resulted in no insertion of extra pipelining flipflops in the circuit, apart from the already present 48 shift register flipflops. Thus only optimization of gate drive capability and fan out was performed. The time shape diagram of this circuit layout is shown in figure 26.
Two circuits were retimed for a clock period of 36ns, equivalent to about 27.8 MHz clock

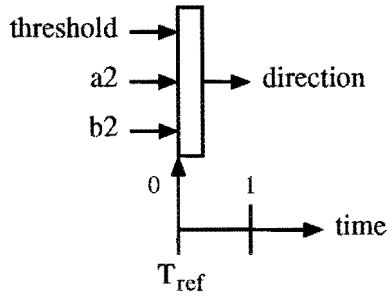


Figure 26: *Time shape diagram of 5MHz retimed direction detector.*

frequency. The difference between these two circuits lies in the time shape of the circuits. One was retimed with a timing constraint demanding equal arrival times of just the inputs a2 and b2, resulting in the insertion of 126 extra flipflops and the time shape diagram of figure 27. The other was retimed with a timing constraint demanding equal arrival times of both the inputs a2, b2 and the threshold input, resulting in the insertion of 170 extra flipflops and the time shape diagram of figure 28.
The fourth circuit was retimed for the fastest possible clock period, which appeared to be 20ns, equivalent to a clock frequency of 50MHz. This resulted in the insertion of 302 extra flipflops in the circuit and the time shape diagram shown in figure 29.
The characteristics of the resulting layouts of these four circuits are summed up in table 5.
Using LOCAL45 extractions were performed on the layouts to obtain transistor netlists in

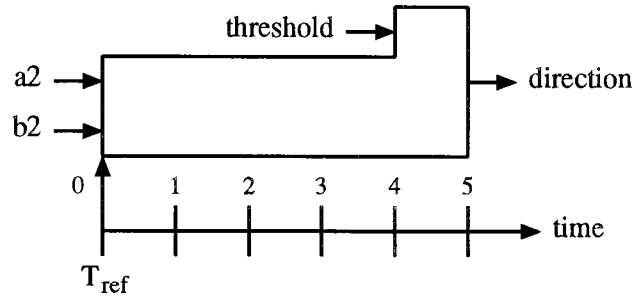


Figure 27: *Time shape diagram of 27.8MHz retimed direction detector with equal timing constraint on inputs a2 and b2.*

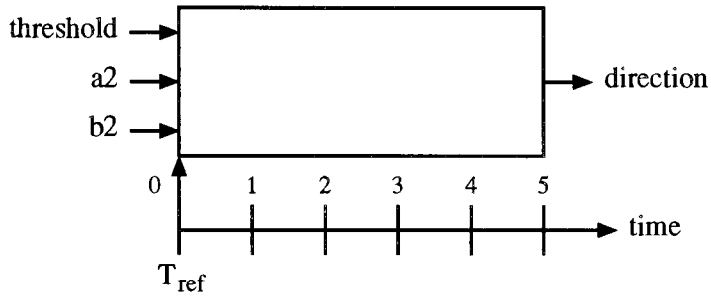


Figure 28: *Time shape diagram of 27.8MHz retimed direction detector with equal timing constraint on inputs a2 and b2 and threshold input.*

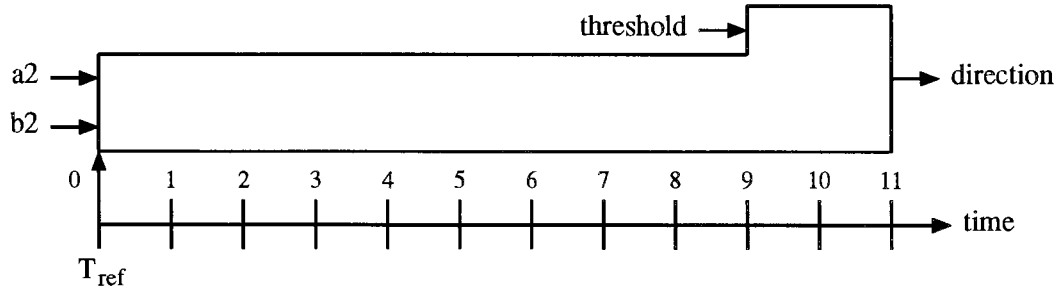


Figure 29: *Time shape diagram of 50MHz retimed direction detector.*

Circuit number	*1*	*2*	*3*	*4*
Retiming period	200 ns	36 ns	36 ns	20 ns
Time shape (skew in clock cycles)	a2 : 0 b2 : 0 threshold : 0 direct : 0	a2 : -5 b2 : -5 threshold : -1 direct : 0	a2 : -5 b2 : -5 threshold : -5 direct : 0	a2 : -11 b2 : -11 threshold : -2 direct : 0
Number of inserted flipflops	0	126	170	302
Chip area	0.73 mm^2	0.99 mm^2	1.00 mm^2	1.23 mm^2
Clock capacitance	3172 fF	10553 fF	12801 fF	19912 fF

Table 5: *Results of the C150DM layout generation of the direction detector for four different retiming specifications.*

SWITCH format as described in section 6.2. The proper operation of the generated circuits could then be verified. Also extractions were performed to obtain a PSTAR netlist of the circuits for the actual power dissipation simulations. As was stated in the previous chapter, the floating nodes had to be removed from these netlists, before PSTAR simulations could be performed.
The power dissipation analysis in the circuits is focused on three disjunct components:

- Supply power dissipated in the combinational logic parts,
- Supply power dissipated in the flipflops,
- Power, coming from the clock driver, dissipated in the clock line.

These components will be further explained in the following sections.

7.2 Analysis of power dissipation in the combinational logic

When retiming is used in layout generation, the occurrence of redundant transitions can be reduced because delay paths will be more balanced due to the insertion, repositioning or removal of flipflops that influence the dataflow in these paths. In general, the more flipflops are inserted by the retimer, the more unbalance in the delay paths will be avoided and consequently the more redundant transitions will be suppressed. Therefore, the amount of redundant transitions compared to the amount of useful transitions is a measure for the reduction in the number of transitions that can be achieved by retiming and consequently an indication for the reduction in power dissipation possible in the combinational logic of a circuit. It was found in section 5.4 that for random inputs transition activity in the

combinational logic of the direction detector can be reduced with a factor of 4.79 if all delay paths are balanced.

7.3 Analysis of power dissipation in the flipflops

To be able to estimate the average power dissipated by the flipflops in the direction detector circuit, a single flipflop of the same type as was used in the (retimed) direction detector, was used as a reference. This master-slave D-flipflop, of standard cell type DNN10TAD, is shown in figure 30. An extraction was performed on the layout of this flipflop cell to

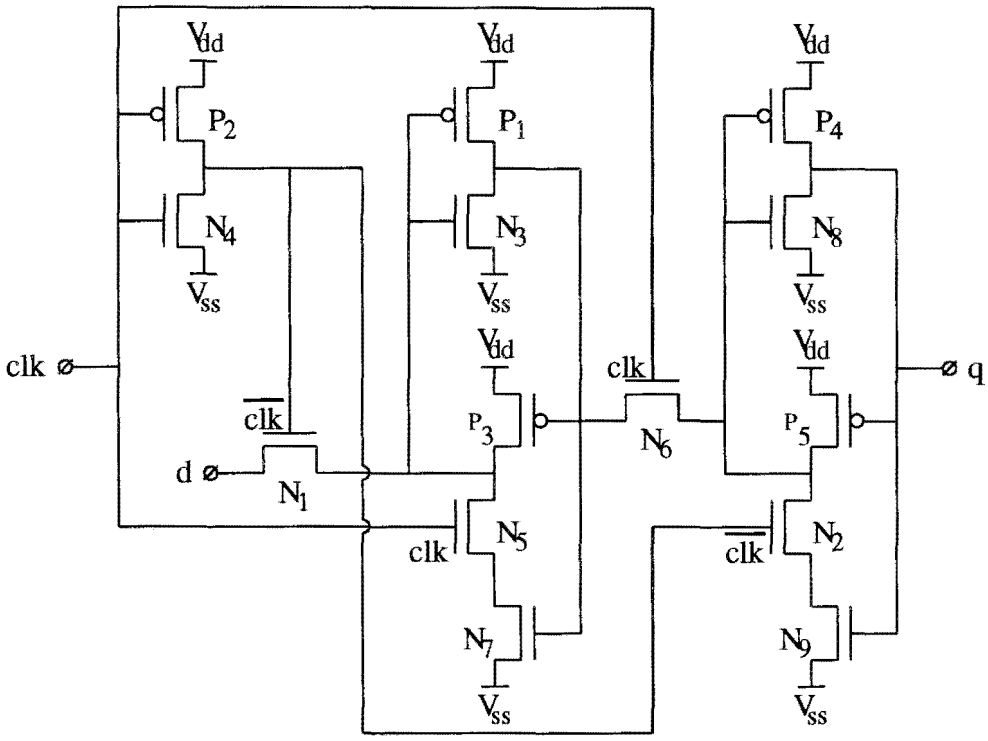


Figure 30: *Standard cell DNN10TAD.*

obtain a PSTAR netlist on which simulations could be carried out.
The flipflop was simulated for 200 clock cycles at a frequency of 5MHz (200 ns clock period). When the flipflop input was held at a constant (zero) value an average supply power dissipation was of $6.7\mu W$ was found. A constantly switching input at 2.5MHz, which is equal to half the clock frequency according to figure 31, resulted in an average supply power dissipation of $31.1\mu W$. It is assumed that on the average a flipflop in the circuit has to clock a constant input for about 50 percent of the time and a switching input for the rest of the time under normal circuit operation for random inputs. In other words, the transition activity for a flipflop is assumed to be 50%. Therefore an average flipflop supply power dissipation of about $\frac{6.7+31.1}{2} = 18.9\mu W$ during normal operation is found. The same result can be found when random inputs are applied to the flipflop. This figure of $18.9\mu W$

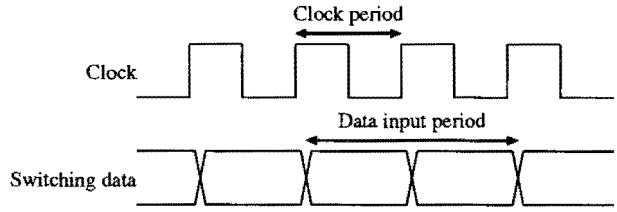


Figure 31: *Constantly switching data input.*

supply power dissipation can now be used to calculate the total amount of supply power dissipated by the flipflops in the direction detector layouts. By multiplying it by the total number of flipflops in the circuits an accurate average flipflop power dissipation figure for random inputs can be determined.

7.4 Analysis of power dissipation in the clock line

To be able to split the power dissipation in three components using PSTAR the simulation method shown in figure 32 can be used. The clock power component is determined sepa-

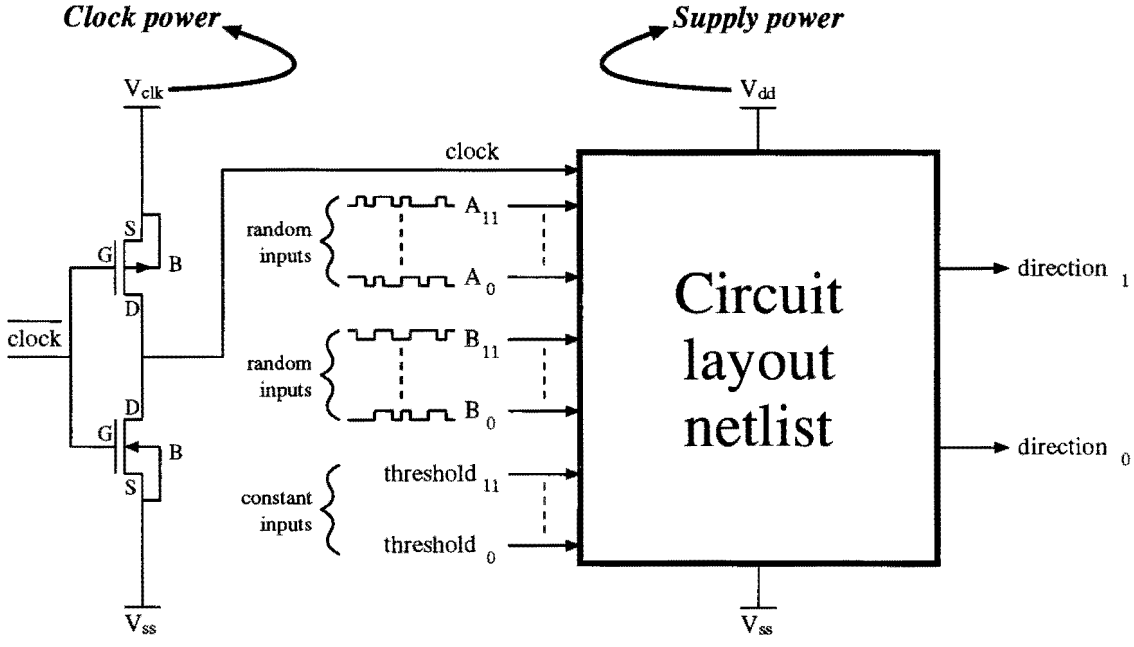


Figure 32: *Graphical representation of netlist simulation method in* PSTAR.

rately from the combinational logic component and the flipflop component. These last two components together constitute the power coming from the supply. The clock input is fed to the netlist via a separately defined clockbuffer with a separate power supply V_{clk}. In this way this power supply is effectively connected to the total load capacitance presented

by the netlist to the clock input. So now the power supplied by this clock buffer power supply V_{clk} is approximately equal to the power which will be dissipated in the clock line. The channel width (W) and channel length (L) of the PMOS and NMOS transistors used in the clockbuffer are chosen such that these transistors can supply enough current to make sure that the rise and fall times of the clock input are approximately equal to $1ns$. Because the necessary amount of current supplied by the clockbuffer is dependent on the loading capacitance of the clock input, the transistor sizes have to be determined for all four layouts separately. In practice not the W and L sizes of the clock buffer transistors are adapted, but rather a special multiplication factor $MULT$ used in the PSTAR models for these transistors. This factor represents the number of the same transistor in parallel and is calculated for each circuit layout.
It was found from the DNN10TAD flipflop simulations that the clock input of this flipflop, which has an effective load capacitance of about $22fF$, can be fed from a clock buffer with transistor sizes of $0.8\mu m$ for both W and L and a multiplication factor $MULT$ equal to 1 if clock rise and fall times of about $1ns$ are desired. The W and L sizes of $0.8\mu m$ were also used in the clock buffers for the direction detector simulations. The transistor multiplication factors in the clock buffers for these simulations were calculated from:

$$MULT = \frac{C_{clock,\ dirdet}}{C_{clock,\ DNN10TAD}} = \frac{C_{clock,\ dirdet}}{22fF} \tag{32}$$

The clock capacitances shown in table 5, obtained from capacitance extractions as described in section 6.2 were used in these calculations, leading to the results shown in table 6.
The dynamic power dissipation in the clock lines can be predicted using the clock capacitances and the formula:

$$P_{clock,\ dynamic} = C_{clock} V_{dd}^2 f_{clock} \tag{33}$$

The results of the calculations using this formula for a clock frequency of 5MHz and a supply voltage of 5V are also shown in table 6.

Circuit number	*1*	*2*	*3*	*4*
Clock capacitance	3172 fF	10553 fF	12801 fF	19912 fF
Transistor multiplication factor	144	480	582	905
Dynamic power dissipation in clock	0.40 mW	1.32 mW	1.60 mW	2.49 mW

Table 6: *Calculated clock buffer transistor multiplication factors and dynamic power dissipation in clock lines for 5MHz clock frequency and 5V supply voltage.*

7.5 PSTAR simulation results for the direction detector

The netlists extracted from the four direction detector layouts were simulated in PSTAR for 20 random input changes, or 20 clock cycles. Of course to obtain very accurate results for random inputs of 12 bit wordlength (which is the input wordlength of the direction detector) many more input changes would be necessary. However, simulating for large amounts of inputs would lead to unacceptably long simulation times (1 clock cycle in the direction detector takes up about 1 hour of CPU time in PSTAR simulations) and is therefore not feasible. Still the average power estimation results which were obtained with only 20 randomly generated inputs appeared to be converging and were found to be accurate enough for the power dissipation analysis. This can be seen from appendices J, K, L and M where the average power dissipation, estimated by PSTAR, is plotted as a function of time for the four different direction detectors.
In the simulations the threshold input of the direction detector was kept at a constant level of 1024 in 12 bit two's complement. This figure was chosen quite arbitrarily. Note that numbers that have to be compared to this threshold in the direction detector range from 0 to 4096. This can easily be seen from the direction detector architecture.
The direction detectors were all simulated using the same clock frequency of 5MHz. The results are shown in table 7.

Circuit number	*1*	*2*	*3*	*4*
Retiming period	200 ns	36 ns	36 ns	20 ns
Number of flipflops	48	174	218	350
Chip area	0.73 mm^2	0.99 mm^2	1.00 mm^2	1.23 mm^2
Clock capacitance	3172 fF	10553 fF	12801 fF	19912 fF
Logic power	21.8 mW	9.7 mW	7.5 mW	6.1 mW
Flipflop power	0.9 mW	3.3 mW	4.1 mW	6.6 mW
Clock power	0.5 mW	1.5 mW	1.8 mW	2.8 mW
Total power	23.2 mW	14.5 mW	13.4 mW	15.5 mW

Table 7: PSTAR *simulation results for the direction detector operated at equivalent 5MHz clock frequency using 20 random inputs.*

The flipflop power results are calculated manually by multiplying the number of flipflops in the circuit by the supply power dissipation of a single flipflop as was explained in section 7.3. The combinational power results are obtained by subtracting the power dissipated in the flipflops from the total supply power dissipation estimated by PSTAR. The clock power dissipation follows directly from the PSTAR output, using the simulation method as described in section 7.4.
The results in table 7 are also plotted in figure 33. We see a decrease in the combinational logic power dissipation as the retiming frequency and consequently the number of pipelin-

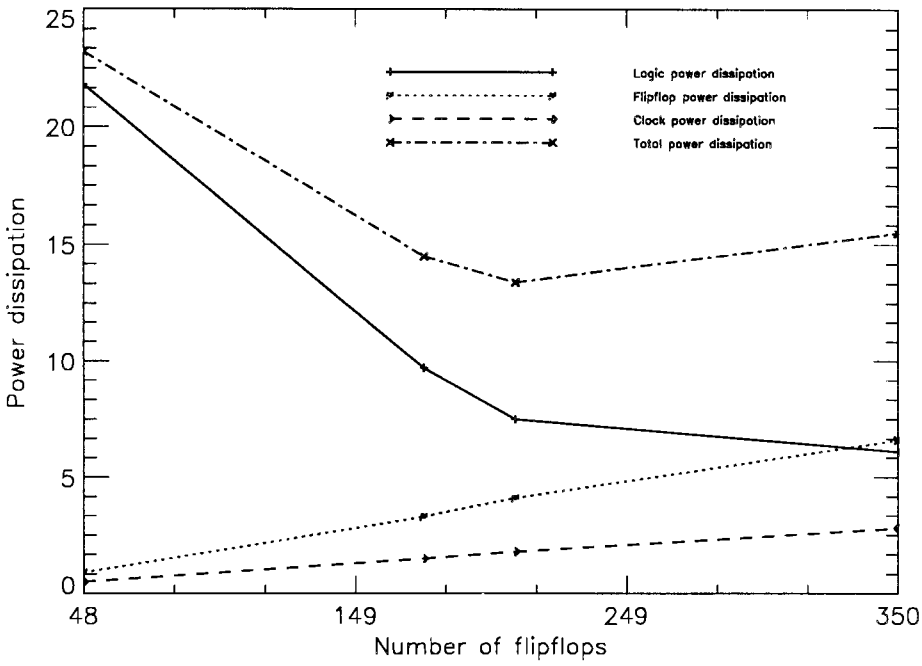


Figure 33: *Power dissipation results as a function of the number of flipflops in the circuit.*

ing flipflops in the circuit increases. Also the difference between the two circuits retimed for a 36ns clock period is significant. The circuit with the larger amount of flipflops and therefore presumably less reconvergent delay paths dissipates less power in its combinational part. A factor of $\frac{21.8}{6.1} \approx 3.57$ difference in power dissipation in the combinational logic is found between circuits 1 and 4. This reduction comes from the dramatic decrease in the number of redundant transitions when a large number of pipelining flipflops is inserted in the circuit. However, a factor of 4.79 as was mentioned in section 7.2 is not achieved, so apparently still some redundant transitions are present in circuit 4.

The increase in flipflop power dissipation follows directly from the increased number of flipflops inserted in the circuits as the retiming frequency is enlarged.

The clock power dissipation is highly dependent on the load capacitance of the clock input. Because extra clock circuitry is necessary when more flipflops are inserted in the circuits as a result of a larger retiming frequency, this capacitance will increase. This explains the fact that the clock power dissipation increases with an increasing number of flipflops. The estimated total clock power is somewhat larger than the ideally calculated results from table 6. An explanation for this can be found in the finite ($1ns$) rise and fall times of the clock signal that is applied to the direction detector, leading to static power dissipation. Another explanation may be the extra capacitances introduced by the transistors in the clock buffer.

The total power dissipation plot in figure 33 exhibits a minimum for a certain retiming frequency or pipelining configuration. In other words, a retiming frequency which is optimal for power can be found. Therefore, retiming a circuit for a higher throughput than necessary, but running it at the original clock frequency may result in lower power dissipation.

8 Reducing power dissipation in the direction detector

In the previous chapter an analysis on power dissipation was performed. In this chapter we will look at some methods to reduce power dissipation. Various methods exist to reduce power dissipation in logic circuits. These lie in the field of for example circuit architecture. Introducing more balanced paths in the architecture is an example for this. The influence of power supply voltage reduction and changes in circuit architecture is investigated in this chapter. A realistic clock frequency of 27.8MHz is chosen for the direction detector.

8.1 Lowering the supply voltage

In the previous chapter an analysis was made on the power dissipation in a direction detector layout in C150DM technology, operating under 5V supply voltage. As was clear from equation 1 the supply voltage is dominant in the dynamic power dissipation. Reducing the supply voltage will result in a large reduction in power dissipation. Therefore, to obtain a realistic low power reference the supply voltage in the direction detector was reduced, while the required throughput was specified on 27MHz. However, the subdivision in the different power dissipation components (combinational logic power, flipflop power, clock power) may be different when a smaller supply voltage is used. The choice of the actions that will be taken to reduce power dissipation is determined by the ratio between the different power dissipation components. Therefore the influence of supply voltage reduction on the three power dissipation components in the direction detector was investigated.
To be able to reduce the supply voltage use can be made of a special low power process. The low power version of C150DM, known as C150LP, uses a 3.3V supply voltage instead of the 5V supply voltage of C150DM. It was not possible to generate a complete new retimed layout in C150LP, because not all necessary C150LP library files were available. Therefore the same PSTAR netlist for the 50MHz retimed direction detector as was used in the C150DM simulation, described in the previous section, was used here. The 50MHz direction detector was used to make sure that the circuit is still fast enough if operated at a 3.3V supply voltage. In the PSTAR input, the various process parameters were of course adapted to their C150LP values. In this way still very accurate results could be obtained, without the need for generating a complete new layout.
Due to the reduced supply voltage and changed process parameters the operating speed of the circuit will be lower than 50MHz. Typically a reduction in throughput in the order of the reduction in supply voltage is expected. At present no exact timing analysis of the low power direction detector could be performed, due to the absence of suitable C150LP library files, but it is expected that it will operate at a maximum frequency of about 30MHz [8].
The simulation of the 50MHz direction detector at 27.8MHz clock frequency and 3.3V supply voltage is taken as a reference for what can be achieved in terms of power dissipation in a low power 27.8MHz direction detector. Because no exact speed figure for this direction detector is available these figures cannot be seen as the exact minimum power solution. If the maximum possible direction detector throughput exceeds 27.8MHz less retiming

flipflops would be sufficient and consequently less flipflop power and less clock power, but possibly more logic power would be dissipated in the circuit.

8.1.1 Power dissipation in a single low voltage flipflop

The DNN10TAD netlist was simulated in PSTAR for 27.8MHz operating frequency and 3.3V supply voltage using C150LP process parameters. The same simulation method for 200 clock cycles as described in section 7.3 was used. For a constant flipflop input an average supply power dissipation of $18.6\mu W$ was found. For switching input the result was an average supply power dissipation of $79.4\mu W$. The resulting average supply power dissipation of the DNN10TAD under normal operation is therefore equal to $\frac{18.6+79.4}{2} = 49.0\mu W$. To obtain a reference for the improvement in power dissipation obtained by lowering the supply voltage the DNN10TAD was also simulated for 27.8MHz clock frequency using the 5V supply and C150DM process parameters. This resulted in a power dissipation of $37.5\mu W$ for the constant zero input case and $172.7\mu W$ for the switching input case. Thus an average power dissipation of $\frac{37.5+172.7}{2} = 105.1\mu W$ per flipflop is found.

8.1.2 Power dissipation in a low voltage direction detector

Implementation 4 of the direction detector with 350 flipflops was simulated for 27.8MHz operating frequency and 3.3V supply voltage using C150LP process parameters, for 20 random inputs. From section 8.1.1 follows that the flipflop power dissipation must be equal to $350 \times 49.0\mu W = 17.2mW$. The PSTAR simulation showed a total supply power dissipation of $32.2mW$. Therefore the power dissipation in the combinational logic must be equal to $32.2mW - 17.2mW = 15.0mW$. Again the clock power dissipation can be calculated using equation 33 resulting in a figure of $19912 \times 10^{-15} \times (3.3)^2 \times 27.8 \times 10^6 = 6.0mW$. The clock power dissipation estimated by PSTAR is equal to $7.0mW$. This is again somewhat higher than the calculated value due to finite clock rise and fall times. The simulation results in comparison with the 5V/27.8MHz case are shown in table 8. From this table can be seen that all three power dissipation components decrease by a factor of more than two when supply power is reduced to 3.3V. This factor is proportional to the expected decrease in power dissipation by a factor of $\frac{5^2}{(3.3)^2} \approx 2.3$ according to equation 1.

8.2 The influence of different comparator architectures on transition activity

The critical path in the direction detector circuit architecture, depicted in figure 18 is formed by the three comparators *SGREATER13* and the *FIND MAXMIN* module. A significant amount of redundant transitions can be prevented if this critical path is made as short and as balanced as possible, leading to less delay unbalance in the circuit. The

Power dissipation (mW)	*C150LP process* $V_{dd} = 3.3V$	*C150DM process* $V_{dd} = 5V$
Combinational logic	15.0	34.0
Flipflops	17.2	36.8
Clock	7.0	15.8
Total	39.2	86.6

Table 8: PSTAR *simulation results for direction detector 4 operated at 27.8MHz for C150DM and C150LP.*

comparators form the main part of this critical path. Therefore a closer look is taken at the architecture of these comparators.

8.2.1 Two comparator architectures

Figure 34 shows the basic structure of a linear 4 bit unsigned comparator. This same basic

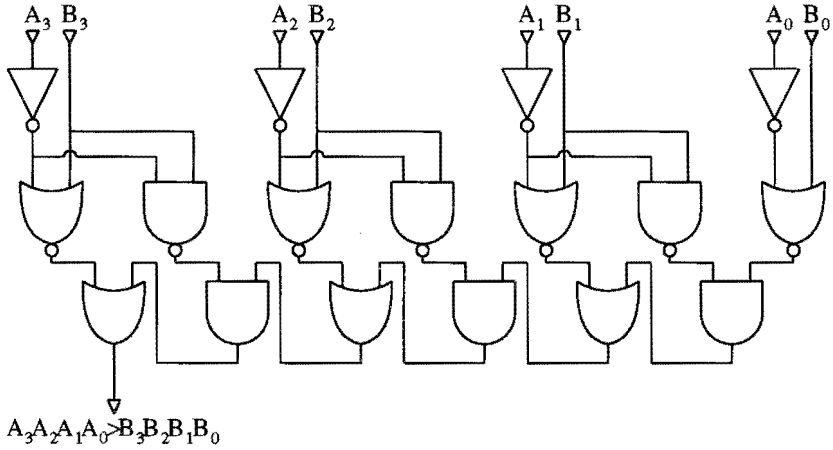


Figure 34: *Linear 4 bit unsigned comparator.*

structure is used in the *SGREATER13* and *SGREATER14* signed comparators. It is clear from this figure that this type of comparator has both a long critical path and a large delay unbalance in its signal paths.
A comparator with a more balanced structure is also possible. Figure 35 shows an example of how two binary unsigned words can be compared using a tree type comparison. Every bit in the first word is compared to its corresponding bit in the second word. The result is a flag which shows whether the bit from the first word is greater, equal or less than the bit from the second word. In the next steps, the result flags are compared two by two leading eventually to a flag which shows if the first word is larger, equal or smaller than the second

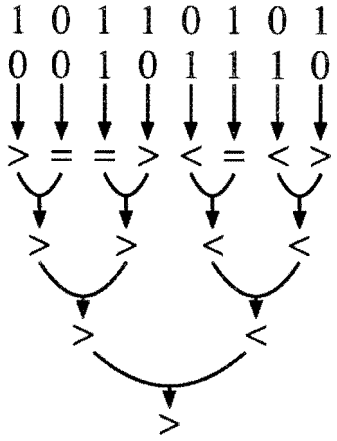


Figure 35: *Tree type comparison method.*

word.

Two different bitwise comparator functions can be distinguished in this comparison. The first row in the comparator tree consists of bitwise comparators comparing single bits of the two input words and deciding if one is greater than, equal to or smaller than the other. The other rows consist of magnitude comparators comparing the result flags from adjacent comparators on a previous row. The truth tables of these two types of comparators are shown in tables 9 and 10. From these truth tables the architectures for the comparators

Inputs		*Outputs*		
A	*B*	>	=	<
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

Table 9: *Truth table for a bit comparator.*

can be derived. In these architectures delay balance is kept in mind, so the different delay paths from inputs to outputs are made as equal as possible. The resulting comparators are shown in figures 36 and 37. A complete tree architecture for a 16 bit unsigned word comparator is shown in figure 38. This 16 bit comparator could be used instead of the *SGREATER13* and *SGREATER14* comparators that were used in the earlier direction detector circuits. The ability for the comparators to compare positive as well as negative numbers is not necessary, because the numbers that have to be compared in the direction detector are coming from absolute difference blocks and are therefore always positive. Thus the unsigned 16 bit word comparator will do, provided that the most significant input bits

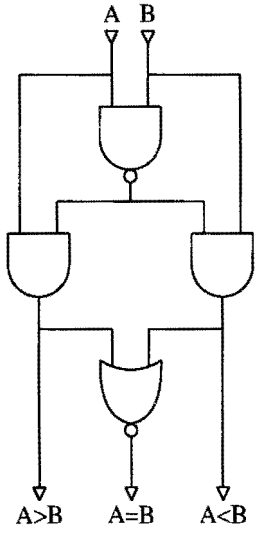


Figure 36: *Bit comparator.*

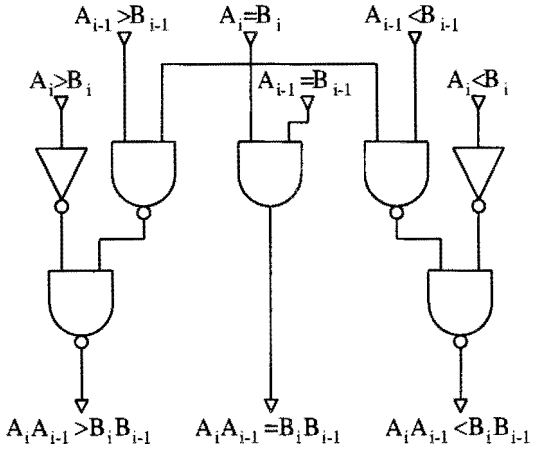


Figure 37: *Magnitude comparator.*

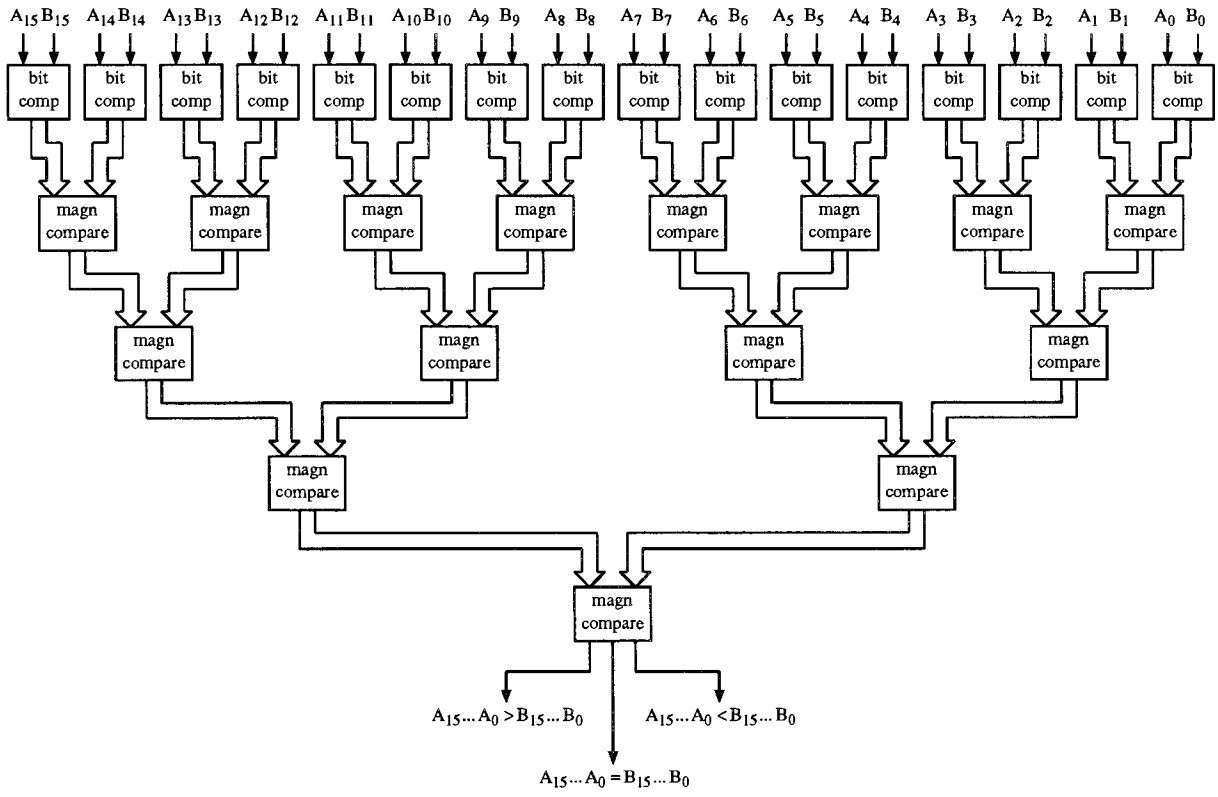


Figure 38: *16 bit tree comparator.*

Inputs						*Outputs*		
$>_i$	$=_i$	$<_i$	$>_{i-1}$	$=_{i-1}$	$<_{i-1}$	$>$	$=$	$<$
0	0	1	0	0	1	0	0	1
0	0	1	0	1	0	0	0	1
0	0	1	1	0	0	0	0	1
0	1	0	0	0	1	0	0	1
0	1	0	0	1	0	0	1	0
0	1	0	1	0	0	1	0	0
1	0	0	0	0	1	1	0	0
1	0	0	0	1	0	1	0	0
1	0	0	1	0	0	1	0	0

Table 10: *Truth table for a magnitude comparator.*

of this comparator are made equal to zero, so that 13 bit and 14 bit comparator functions are obtained.

8.2.2 Power dissipation results using different comparator architectures

A new direction detector layout in C150DM technology was generated and retimed for 50 MHz clock frequency. The 16 bit unsigned tree type comparator from figure 38, instead of the linear *SGREATER13* and *SGREATER14* comparators, was used in this direction detector. The VHDL descriptions of the linear *SGREATER13* comparator and the 16 bit tree type comparator are shown in appendices N and O. The resulting time shape of the direction detector using tree type comparators is shown in figure 39. The time shape of the 50MHz retimed direction detector using linear comparators was already given in figure 29. The extracted netlist of the new layout was simulated in PSTAR at a clock

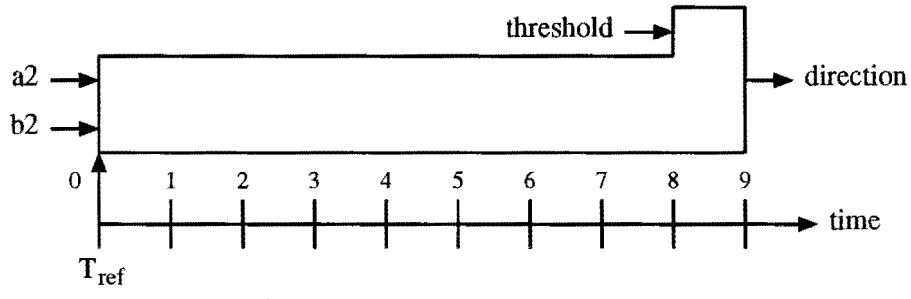


Figure 39: *Time shape diagram of direction detector using tree type comparators.*

rate of 27.8 MHz for 20 random inputs using C150LP low power process parameters. The results of this simulation are shown in table 11. For comparison purposes the results of

the direction detector using linear comparators as presented in sections 7.1 and 8.1.2 are also repeated in this table. From this table some interesting phenomena can be noted.

	Direction detector with linear comparators	*Direction detector with tree type comparators*
Number of flipflops	350	295
Chip area	1.23 mm^2	1.17 mm^2
Clock capacitance	19912 fF	17072 fF
Logic power	15.0 mW	18.4 mW
Flipflop power	17.2 mW	14.4 mW
Clock power	7.0 mW	6.0 mW
Total power	39.2 mW	38.8 mW

Table 11: PSTAR *simulation results for a 50MHz retimed direction detector using different comparator architectures.*

It appears that the direction detector with the tree type comparator has less pipelining flipflops and consequently less clock capacitance and chip area. This is because a tree architecture is fast, so therefore fewer pipelining flipflops have to be inserted. The result of this is that less flipflop power and less clock power is dissipated. However more power is dissipated in the combinational logic compared to the direction detector using linear comparators. One explanation for this fact is that the linear comparators are slow and will therefore be saturated with pipelines when retiming is performed. Consequently only few redundant transitions will be present in these comparators. Therefore the power dissipation in the combinational logic of the direction detector using linear comparators will already be low. Because less pipelines are inserted in the tree comparators, probably more redundant transitions will be present in these comparators and consequently the power dissipation in the combination logic of the direction detector using these comparators will be higher.
However, probably the strongest explanation for the fact that the power dissipation in the combinational logic is larger for the direction detector using tree comparators is as follows. One would expect that redundant transition activity can be reduced if the chosen architecture for the comparators has a more balanced structure and a shorter critical path, like the tree structure. This would be true if equal arrival times for all inputs to the comparators could be assumed. In the direction detector, however, this is not the case. The inputs to the comparators in the direction detector are supplied from *ABSDIFF12* and *ABSDIFF13* module outputs. These modules are comprised of ripple carry subtractor architectures supplied (in the case of *ABSDIFF12*) with at equal times arriving inputs coming from register outputs. The outputs of these ripple carry structures exhibit glitches. This means that in the inputs to the comparator structures also glitches will appear and that stable inputs will not arrive at equal times. Therefore using a more balanced tree

type comparator will have no positive effect, but will even lead to a worsening in the number of transitions, because this type of comparator consists of more gates than a linear comparator. This is depicted in figure 40. It is clear from this observation that it will be

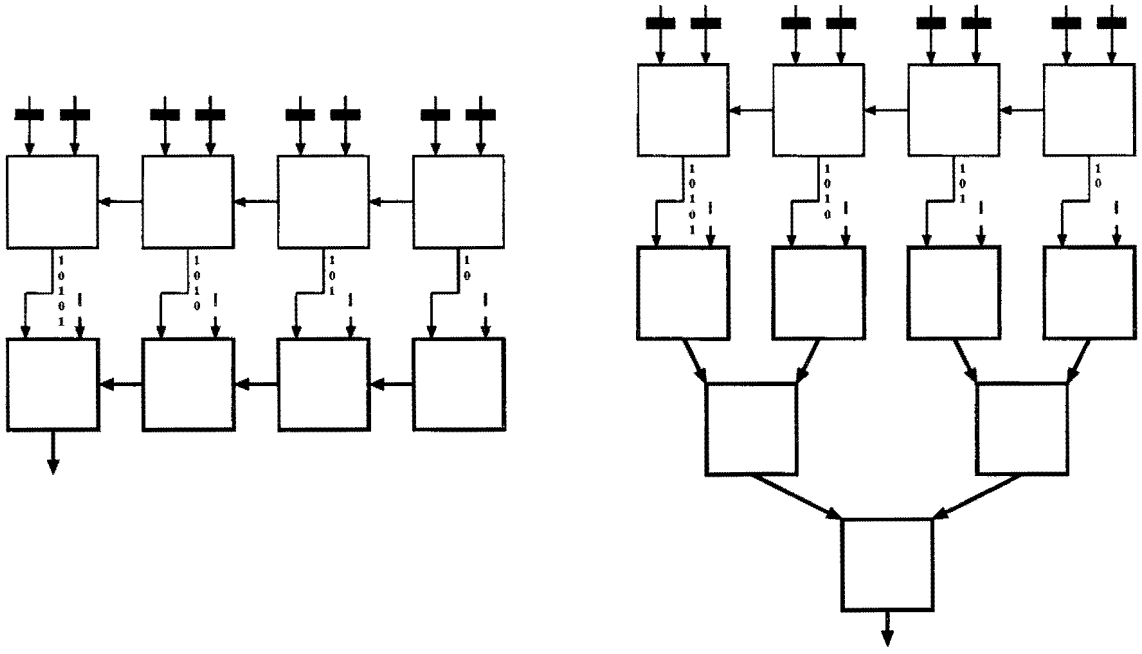


Figure 40: *Linear respectively tree type comparator in combination with a ripple carry structure.*

very difficult to design a low power standard module library, because whether a module architecture has a low power dissipation in practice is also highly dependent on the blocks surrounding the module in a real circuit. It is not enough to look at circuit modules separately and eliminate their delay unbalance. One has to balance the delay paths of the entire circuit as a whole. In the ideal case delay paths between different pipeline stages should be balanced after retiming. In other words, delay balancing and retiming are coupled problems.

In total the direction detector with the tree type comparators dissipates less power than the same circuit with linear comparators. However, the difference of 0.4 mW is negligible. If a lower retiming frequency, resulting in more glitches or redundant transitions, is chosen the results for the tree type comparator case only get worse. This can be seen from table 12 where the simulation results of two direction detector layouts using different comparators and retimed for 25MHz are shown. These circuits were simulated using a 13.9MHz clock frequency, 3.3V supply voltage and C150LP process parameters.

The larger chip area figure for the direction detector with tree type comparators, opposed to the lower number of flipflops proves that the tree type comparators indeed consist of more gates than the linear comparators and that consequently more transitions will occur in these tree type comparators when unequal arriving inputs are applied.

	Direction detector with linear comparators	*Direction detector with tree comparators*
Number of flipflops	161	145
Chip area	0.92 mm^2	0.94 mm^2
Clock capacitance	9589 fF	8554 fF
Logic power	10.1 mW	13.4 mW
Flipflop power	3.9 mW	3.5 mW
Clock power	1.6 mW	1.5 mW
Total power	15.6 mW	18.4 mW

Table 12: PSTAR *simulation results for a 25MHz retimed direction detector using different comparator architectures.*

8.3 Using a parallel architecture

The results of the previous section show that little power can be gained by changing the architecture of the comparator modules. Overall it appears that the power dissipation in the combinational logic cannot be significantly improved by taking architectural measures in this 3.3V design of the direction detector.
Therefore the further improvement in power dissipation must be sought in the flipflop and clock power dissipation. This dissipation can be reduced by making use of a parallel architecture.

8.3.1 Theory behind the parallel architecture

A parallel direction detector architecture is shown in figure 41. In this parallel architecture use is made of two identical direction detectors, each operated at half the original direction detector clock frequency. The inputs $a2$ and $b2$ are alternately supplied to one of the direction detectors via a demultiplexer. The outputs of the two direction detectors are multiplexed to obtain a single *direction* output.

For each direction detector in the parallel architecture the following can be noted:

- Because a direction detector is operated at half the original clock speed, it can be retimed for half the original operating frequency and consequently less flipflops have to be inserted in the circuit. Typically a reduction in pipelining flipflops by a factor comparable to the reduction in circuit speed is expected. So in this example about $N/2$ flipflops are expected, where N is the original number of flipflops in the circuit. This will lead to a reduction in flipflop power dissipation by a factor of about two.

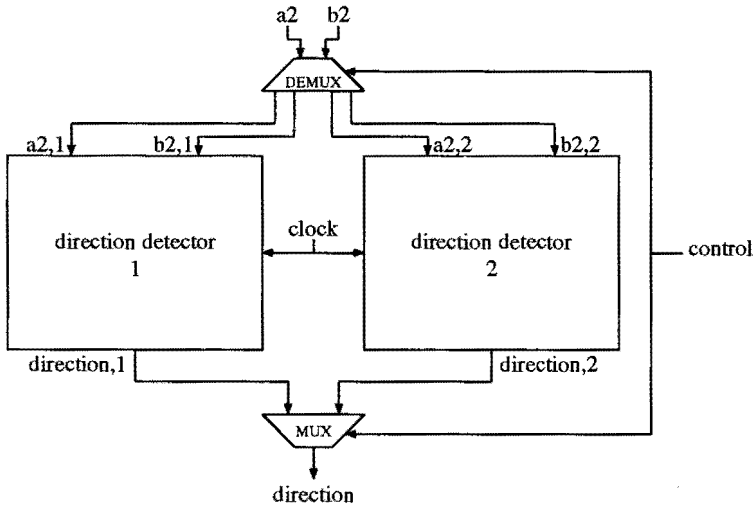


Figure 41: *Parallel direction detector architecture.*

- As a result of the decrease in the amount of flipflops the clock capacitance will also decrease, presumably by the same factor of about two. The consequence of this is a decrease in clock power dissipation by a factor of about two.
- Because less flipflops are inserted in the circuits, the delay unbalance and consequently the amount of redundant transitions will increase. A slight increase in the power dissipation in the combinational logic will occur due to this fact.
- The direction detector is operated at half the original clock rate and the power dissipation in the flipflops, the clock lines and the combinational logic will therefore again decrease by a factor of two.

So for each direction detector, flipflop and clock power dissipation will decrease by a factor of about four and combinational logic power dissipation will decrease by a small factor somewhat larger than one, but smaller than two. This means that in the total parallel direction detector architecture flipflop and clock power dissipation will decrease by a factor of about two and combinational logic power dissipation will slightly increase. Therefore especially when flipflop and clock power dissipation in the original non-parallel architecture are large compared to the power dissipation in the combinational logic much power can be gained by choosing a parallel architecture.

8.3.2 Power dissipation in a single flipflop

To obtain a reference for the flipflop power dissipation a single DNN10TAD flipflop was simulated in PSTAR at a clock frequency of 13.9 MHz (72 ns clock period) for C150LP process parameters and 3.3V supply voltage. This was done for 200 clock cycles and

constant zero, respectively at half the clock rate switching input. The result for the constant zero input case was a supply power dissipation of $7.7\mu W$. For the switching input case the result was a supply power dissipation of $40.3\mu W$. So on average the flipflop dissipates $\frac{7.7+40.3}{2} = 24.0\mu W$ of supply power.

8.3.3 Power dissipation in a parallel direction detector architecture

A new direction detector layout in C150DM technology using linear comparator architectures was generated. This circuit will be named direction detector 5. It was retimed for a clock frequency of 25MHz, equal to half the original retiming frequency of 50MHz. The resulting time shape is shown in figure 42. It was investigated what reduction in power

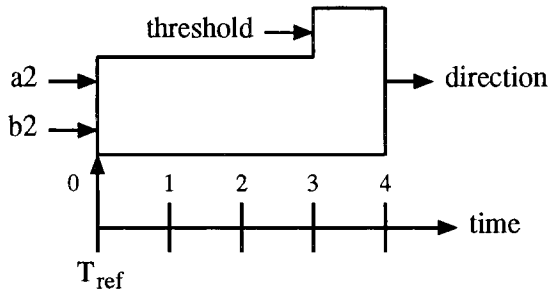


Figure 42: *Time shape diagram of direction detector 5.*

dissipation could be obtained if this direction detector would be used in a parallel architecture.
The extracted netlist of the generated layout was simulated in PSTAR at a clock frequency of 13.9 MHz (half the original clock speed) for 30 random inputs. The results are shown in table 13. The results of table 11 are repeated in the third column of table 13. Comparing the results shows that the flipflop power dissipation has reduced with a factor of $\frac{17.2}{3.9} = 4.4$. Clock power dissipation is lowered with a factor of $\frac{7.0}{1.6} = 4.4$. Power dissipation in the combinational logic decreases by a factor of $\frac{15.0}{10.1} = 1.5$.
So the results of the retiming frequency reduction and clock frequency reduction are as expected.

When the direction detector is used as a module in a parallel architecture an improvement in power dissipation is obtained. Doubling the total power dissipation results from the first column in table 13 (see the second column) and comparing them to the total power dissipation results in the third column gives a reduction of $\frac{39.2}{2\times15.6} = 1.3$ in power dissipation or a 20% improvement. In this power dissipation improvement figure the power dissipation in the extra circuitry necessary in the parallel architecture is not yet accounted for. Therefore these figures might be slightly optimistic.

Of course different low power parallel architectures are possible. Instead of making the complete circuit parallel, it is also possible to generate parallel architectures for parts of

	Direction detector 5	*2× direction detector 5 in parallel architecture (excl. control logic)*	*Direction detector 4*
Number of flipflops	161	323	350
Chip area	0.92 mm^2	1.84 mm^2	1.23 mm^2
Clock capacitance	9589 fF	19178 fF	19912 fF
Logic power	10.1 mW	20.2 mW	15.0 mW
Flipflop power	3.9 mW	7.8 mW	17.2 mW
Clock power	1.6 mW	3.2 mW	7.0 mW
Total power	15.6 mW	31.2 mW	39.2 mW

Table 13: PSTAR *simulation results for a parallel direction detector architecture compared to a non parallel architecture.*

the direction detector. Figure 43 shows a parallel architecture in which the shift registers are extracted from the direction detector before doubling is performed. This means that instead of a demultiplexer and two times four shift registers just six shift registers, clocked at the original clock speed are necessary. Therefore both a reduction in control logic and a reduction in necessary flipflops is obtained, leading to less area and less power dissipation. The way in which video samples are processed when use is made of a parallel architecture is shown in figure 44.

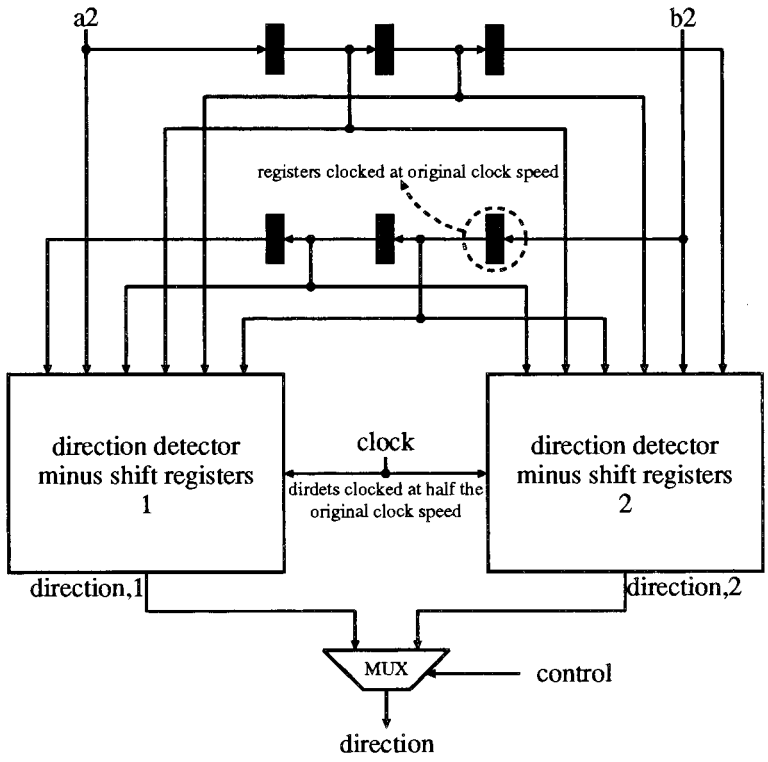


Figure 43: *Alternative parallel direction detector architecture.*

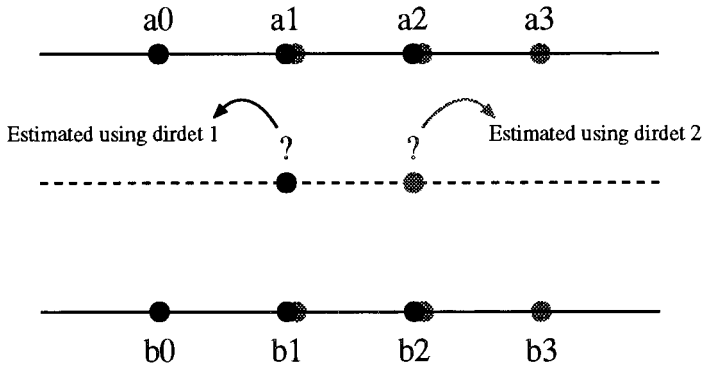


Figure 44: *Progressive Scan using parallel direction detector.*

9 Conclusions

The following main conclusions can be drawn.

The most effective method to reduce power dissipation is to lower the supply voltage. In this way drastic reductions in power dissipation for given throughput constraints are possible. Even at lower voltages high throughputs which are necessary for example for real-time video applications are possible by introducing more parallelism in the architecture. This can be done by using pipelining optimized by retiming and/or by putting several operation units in parallel. The basic idea is to execute the same algorithm or task by splitting it into more and slower subtasks executed on parallel hardware. What must be avoided is hardware sharing, programmable architectures and high clock rates. Functions must be implemented on hardware which operates at a clock rate which is comparable to the rate of the incoming signals or even lower.

In this work power dissipation in synchronous circuits is analyzed and divided into three different components: dissipation in the combinational logic, in the flipflops and in the clock lines. Power dissipation in the combinational logic is often the result of glitches (or redundant transitions). Glitches can be almost completely eliminated by simple techniques already present in PHIDEO, such as the introduction of flipflops due to retiming and pipelining and/or by choosing different architectures. Power dissipation in the flipflops and in the clock lines can be minimized by introducing more parallelism.

These conclusions can be further detailed as follows.

- A mathematical analysis was carried out on the transition probability in a ripple carry adder.
  From this analysis followed that the worst case number of transitions occurs in the MSB full adder FA_{N-1} and is equal to N. The probability on the occurrence of this worst case number of transitions is small, even for small word sizes.
  However, it also can be concluded that the average number of redundant transitions for random input patterns is rather large compared to the average number of useful transitions. Already for small adders the redundant/useful ratio approaches 0.8.

- To analyze the dependence of transition activity on circuit architecture switch level simulation was carried out on two types of multipliers, known as the *array multiplier* and the *wallace tree multiplier*.
  The result of these simulations is that the amount of redundant transitions is highly dependent on the delay unbalance in a chosen architecture and that this amount of redundant transitions can be rather large, especially when the logic depth is large. For the 16x16 array multiplier the redundant/useful ratio was found to be 3.26. For the 16x16 wallace tree multiplier this ratio was equal to 0.16. It appeared that the wallace tree multiplier, which has far more balanced delay paths, exhibits far less redundant transitions than the array multiplier.

So the choice for a different architecture has a great influence on the number of redundant transitions.

- Another simple measure to significantly reduce the number of redundant transitions in combinational logic significantly is the use of pipelining. In this way synchronous circuits can be designed in which power dissipation due to glitches is negligible.
- Circuit level simulations to determine power dissipation were carried out on different layout versions of the *direction detector* circuit which is a typical example of a PHIDEO processing unit. The differences in the circuit layouts are in the way the circuits were retimed. The results of the simulations were divided in three components: power dissipation in the combinational logic, power dissipation in the flipflops and power dissipation in the clock lines.
- It appeared that pipelining for a higher operating frequency resulted in more flipflops in the circuit and therefore less delay unbalance. Thus the power dissipation in the combinational logic decreased with increasing retiming frequency. The power dissipation in the flipflops and in the clock lines increased almost linear with the number of inserted pipelining flipflops in the circuit. However, the amount of power that can be gained by reducing the power dissipation in the combinational logic is larger than the extra power that is lost in the flipflops and the clock lines for a wide range of retiming frequencies. So an optimal retiming frequency for power dissipation can be found.
- When retiming is performed it is very difficult to predict exactly which architecture is optimal for power dissipation. This is due to the fact that the insertion of pipelining flipflops influences the delay balance in a circuit and thereby influences the amount of redundant transitions that can appear in a circuit. An architecture which is optimal at one retiming frequency may be less of an optimal solution when retimed at an other frequency, depending on the resulting positions of the pipelining registers. In designing a low power architecture it would be very helpful if it was possible for the user to have influence on the positioning of pipelining flipflops during retiming or at least to be able to see where in the circuit the retimer has placed those flipflops. When the positions of the pipelining stages are known, it would be ideal if it was possible to resubstitute balanced circuit parts between different pipelining stages, thus creating an almost glitch-free circuit.
- When designing a low power circuit architecture it is important to view the circuit as a whole and make its signal delay paths as balanced as possible. It is not enough to separately optimize the delay balance of different circuit modules. A module that is optimal for power dissipation in one circuit may be non-optimal in another, depending on the surrounding architecture of the module. Therefore it is very difficult to design a low power standard module library on the architectural level.
- Using a parallel architecture consisting of duplicate circuits retimed at half the original retiming frequency and operated at half the original clock speed can result in

a significant reduction in power dissipation as shown by the example of a parallel direction detector.

- A future power analysis tool should not only give the user information on the total power dissipation of the circuit. It should also supply the user with information on which part of this dissipation is useful and which part is not. Furthermore information on redundant and useful transition activity on circuit nodes should be given as well as information on nodal capacitances and signal path delays.

References

[1] R. Brodersen, A. Chandrakasan, S. Sheng.
Low-Power Signal Processing Systems
5th IEEE Workshop on VLSI Signal Processing, California, October 1992.

[2] P.A. Kuppen and B.F. Lynch.
SWITCH Background and Theory, release 3.
Philips Research Laboratories, Eindhoven, Internal document, April 1988.

[3] P.A. Kuppen, B.F. Lynch and G.G. Schrooten.
SWITCH Release 4.0.
Philips Research Laboratories, Eindhoven, Internal document, July 1991.

[4] G.G. Schrooten and M.J. Quinn.
Estimation of power dissipation within SWITCH.
Philips Research Laboratories, Eindhoven, Technical Note 144/91, RWR-555-GS-91022-gs, January 1991.

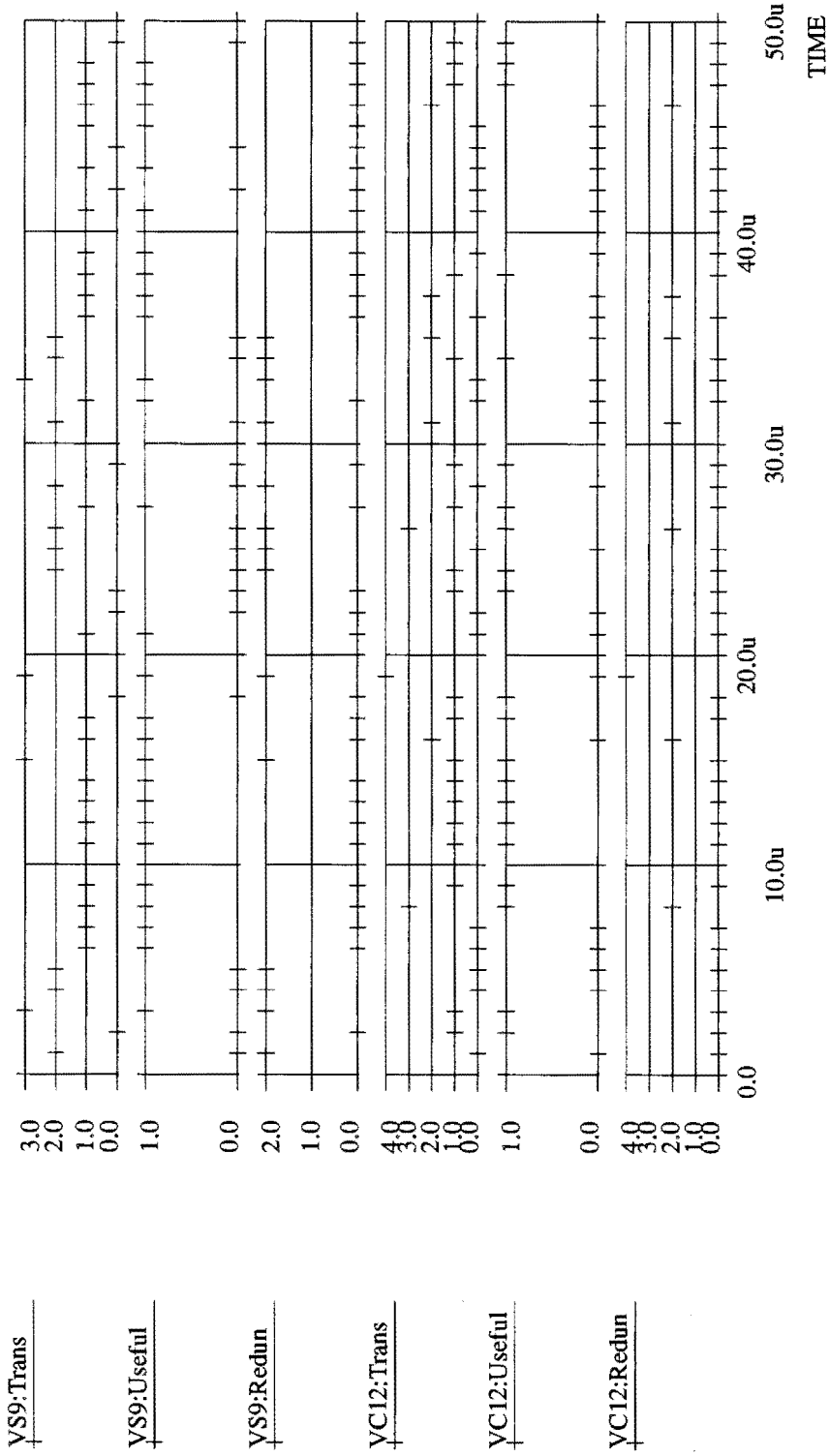
[5] J. van Meerbergen, P. Lippens, B. McSweeney, W. Verhaegh, A. van der Werf, A. van Zanten.
Architectural Strategies for High-Throughput Applications.
Papers on the PHIDEO High-Level Synthesis Project, Philips Research Laboratories, Eindhoven, November 1992.

[6] *Optima v1.1 User Manual.*
Philips Electronic Design & Tools, Hilversum, 1992.

[7] *Pstar User Guide v1.10.*
Philips Electronic Design & Tools, Eindhoven, January 1992.

[8] *Internal communication with Ir. B.J.S. de Loore.*
Philips Research Laboratories, Eindhoven, September 1993.

A Transitions in a 16 bit ripple carry adder



B Source code of input_bit.c

Program to generate a simulation control (.scl) file including 8, 12 or 16 bit (random) input sequences for use in transition activity simulations using SWITCH.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define WORDLENGTH 8 /* User defined */
#define MAX_SAMPLE_NUMBER 10000

int power(x,n) int x,n; /* Raise x to n-th power; n > 0 */
{
   int i,p;

   p = 1;
   for (i = 1; i <= n; ++i) p = p * x;
   return(p);
}

int round(x) float x;   /* Rounding of float to nearest integer */
{
   int y;

   y = (int) x;
   if (x - y >= 0.5) y++;
   return(y);
}

float ramp(x) float x;  /* Ramp function with period 2 pi */
{
   float y;

   y = ((int)(10000 * x) % (int)(10000 * 2 * M_PI))/(10000 * 2 * M_PI);
   return(y);
}

char *integer_to_binary_string(n) int n;
{
   int i;
   char b[WORDLENGTH+1];

   for (i = 0; i < WORDLENGTH ; i++) {
      if ((n & 01) == 0) b[i] = '0';
      else b[i] = '1';
      n >>= 1;
   }
   b[WORDLENGTH] = NULL;
   return(b);
}

main(argc,argv) /* Generate input sequences for SWITCH */

int argc;
char *argv[];

{
   FILE *fopen(),*fpin,*fpout;
   int c;
   int i,j,loop,sample;
   int function_number,number_of_samples,amplitude,time_point,time_step;
   int control_nr_steps,control_time_step,starting_sample;
```

```
float initial_phase,number_of_periods;
char filename1[50],filename2[50];
char terminal_name_prefix[10],terminal2_name_prefix[10];
char *word,word_buffer[MAX_SAMPLE_NUMBER][WORDLENGTH+1];
double dummy_sample;

printf("\nWordlength of generated input sequences is: %d bits\n",WORDLENGTH);

if (argc == 1) {
   printf("\nSpecify name of file containing header for .scl file: ");
   scanf("%s",filename1);
   printf("Specify name of output .scl file: ");
   scanf("%s",filename2);
}
else {
   strcpy(filename1,argv[1]);
   strcpy(filename2,argv[1]);
}
strcat(filename1,".header");
strcat(filename2,".scl");

if ((fpin = fopen(filename1,"r")) == NULL) {
   printf("Error while opening file '%s'\n",filename1);
   return;
}

if ((fpout = fopen(filename2,"w")) == NULL) {
   printf("Error while opening file '%s'\n",filename2);
   return;
}

/* Copy .scl header from input file to output file */
while ((c = getc(fpin)) != EOF) putc(c,fpout);

printf("Specify NR\\STEPS for SWITCH CONTROL statement: ");
scanf("%d",&control_nr_steps);
printf("Specify TIME\\STEP for SWITCH CONTROL statement in nanoseconds: ");
scanf("%d",&control_time_step);
fprintf(fpout,"\nCONTROL NR\\STEPS = %d  TIME\\STEP = %dN\n\n"
        ,control_nr_steps,control_time_step);

for (loop = 1; loop < 3; loop++) {
   printf("\nSpecify input %d terminal name prefix: ",loop);
   scanf("%s",terminal_name_prefix);
   printf("Specify function (1 = sine, 2 = ramp, 3 = random): ");
   scanf("%d",&function_number);
   while (function_number < 1 || function_number > 3) {
      printf("Illegal function number\n");
      printf("Specify function (1 = sine, 2 = ramp, 3 = random): ");
      scanf("%1d",&function_number);
   }
   printf("Specify amplitude (max. resolution = %d (times 0.5 for sine)): "
          ,power(2,WORDLENGTH) - 1);
   scanf("%d",&amplitude);
   if (function_number != 3) {
      printf("Specify initial phase of function (in multiples of pi): ");
      scanf("%f",&initial_phase);
      printf("Specify the amount of function periods to be sampled: ");
      scanf("%f",&number_of_periods);
   }
   printf("Specify number of samples to be taken from function
           (max. %d samples): ",MAX_SAMPLE_NUMBER);
   scanf("%d",&number_of_samples);
   while (number_of_samples > MAX_SAMPLE_NUMBER) {
      printf("Illegal number of samples\n");
```

```
      printf("Specify number of samples to be taken from function
              (max. %d samples): ",MAX_SAMPLE_NUMBER);
      scanf("%d",&number_of_samples);
   }
   printf("Specify time step in nanoseconds: ");
   scanf("%d",&time_step);
   printf("Save random numbers to .scl file starting from sample
          (0 to start from first sample): ");
   scanf("%d",&starting_sample);
   if (starting_sample>0) printf("Ignoring first %d samples...\n",starting_sample);
   for (i=0;i<starting_sample;i++) dummy_sample = drand48();

   printf("Generating and saving %d samples...\n",number_of_samples);
   fprintf(fpout,"\n\nLISTING\n");
   fprintf(fpout,": Input terminal %s[i]\n",terminal_name_prefix);
   fprintf(fpout,": Function number: %d (1 = sine, 2 = ramp, 3 = random)\n"
           ,function_number);
   fprintf(fpout,": Amplitude: %d\n",amplitude);
   if (function_number != 3) {
      fprintf(fpout,": Initial phase: %f * pi\n",initial_phase);
      fprintf(fpout,": Number of periods: %f\n",number_of_periods);
   }
   fprintf(fpout,": Number of samples: %d\n",number_of_samples);
   fprintf(fpout,": Time step: %d nanoseconds\n:\n",time_step);
   fprintf(fpout,"NO\\LISTING\n");

   for (i=0;i<number_of_samples;i++) {
      switch(function_number) {
      case 1:
         sample = round (amplitude * sin(i * number_of_periods * 2 * M_PI /
                         number_of_samples + initial_phase * M_PI));
         break;
    case 2:
      sample = round (amplitude * ramp(i * number_of_periods * 2 * M_PI /
                      number_of_samples + initial_phase * M_PI));
   break;
      case 3:
         sample = round (amplitude * drand48());
         break;
      }

      word = integer_to_binary_string(sample);
      for (j = 0; j <= WORDLENGTH; j++) word_buffer[i][j] = word[j];
   }

   /* Write output to file */
   for (j = 0; j < WORDLENGTH; j++) {
      time_point = 0;
      fprintf(fpout,"INPUT %s%d S\\T = \t",terminal_name_prefix,j);
      for (i = 0; i < number_of_samples-1; i++) {
         fprintf(fpout,"%c,%dN,\t\t&\n\t\t",word_buffer[i][j],time_point);
         time_point += time_step;
      }
      fprintf(fpout,"%c,%dN\n\n",word_buffer[i][j],time_point);
   }
}

/* Close files */
fclose(fpin);
fclose(fpout);
}
```

C Source code of pwr2tra.c

Program to convert a SWITCH power (.pwr) output file to transition activity (.tra_lst, .tra_int, .tra_ifp) files.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define TRUE 1
#define FALSE 0
#define PWR_VAR_STRIP_LEN 11 /* Length = strlen(":AV-MAX-PWR") */

#define C_LOAD 10E-15        /* Loading capacitance in outputbuffer */
#define V_DD 5               /* Supply voltage */
#define MAX_NODE_NUMBER 2000

int power(x,n) int x,n; /* Raise x to n-th power; n > 0*/
{
   int i,p;

   p = 1;
   for (i = 1; i <= n; ++i) p = p * x;
   return(p);
}

int round(x) float x;
{
   int y;

   y = (int) x;
   if (x - y >= 0.5) y++;
   return(y);
}

double string_to_double(value_string) char *value_string;
{
   int i,j,expsign,exponent;
   double value;

   i = 0;
   value = 0;

   while (value_string[i] != '.') {
      value = 10 * value + value_string[i] - '0';
      i++;
   }
   i++; /* Skip '.' */
   j = 1;
   while (value_string[i] != 'D') {
      value += ((double)(value_string[i] - '0')/power(10,j));
      j++;
      i++;
   }
   i++; /* Skip 'D' */

   if (value_string[i] == '+') expsign = 0;
   else if (value_string[i] == '-') expsign = 1;
   i++; /* Skip sign */

   exponent = 10 * (value_string[i] - '0') + (value_string[i+1] - '0');

   if (expsign) value /= (double)power(10,exponent);
```

```
   else value *= power(10,exponent);

   return(value);
}

main(argc,argv) /* Convert .pwr file to .tra file */

int argc;
char *argv[];
{
   FILE   *fopen(),*fpin,*fpout,*fpout2,*fpout3;
   int    i,j,k,c,pwr_var_length,node_name_length,dummy,node_number,counter,input_changes;
   int    d1,d2,d3,d4,d5,d6,d7,d8,d9;
   double power_value[MAX_NODE_NUMBER];
   int    trans_value,useful_value,redun_value;
   int    all_useful_value,all_redun_value;
   float  redunrel_value,all_redunrel_value;
   int    tot_useful_value[MAX_NODE_NUMBER],tot_redun_value[MAX_NODE_NUMBER];
   float  tot_redunrel_value[MAX_NODE_NUMBER];
   float  TIME_STEP,input_time_step;
   int    number_of_steps,fscan_result;
   char   answer[3];
   int    create_ifp;
   char   filename1[50],filename2[50],filename3[50],filename4[50];
   char   variable_name[MAX_NODE_NUMBER][50],value_string[50];
   char   buffer[1000];

   if (argc != 4) {
      printf("\nSpecify name of input .pwr file: ");
      scanf("%s",filename1);
      printf("Specify name of output .tra files: ");
      scanf("%s",filename2);
      printf("Specify 'control TIME\\STEP' of SWITCH simulation in nanoseconds: ");
      scanf("%f",&TIME_STEP);
      printf("Specify 'input change time interval' of SWITCH simulation in nanoseconds: ");
      scanf("%f",&input_time_step);
      printf("Create .tra_ifp file (yes/no)? ");
      scanf("%s",answer);
      if (answer[0] == 'y') create_ifp = TRUE;
      else create_ifp = FALSE;
   }
   else {
      strcpy(filename1,argv[1]);
      strcpy(filename2,argv[1]);
      create_ifp = FALSE;
      TIME_STEP = atof (argv[2]);
      input_time_step = atof (argv[3]);
   }
   strcpy(filename3,filename2);
   strcpy(filename4,filename2);
   strcat(filename1,".pwr");
   strcat(filename2,".tra_ifp");
   strcat(filename3,".tra_lst");
   strcat(filename4,".tra_int");

   if ((fpin = fopen(filename1,"r")) == NULL) {
      printf("Error while opening file '%s'\n",filename1);
      return;
   }
   if (create_ifp) {
      if ((fpout = fopen(filename2,"w")) == NULL) {
         printf("Error while opening file '%s'\n",filename2);
         return;
      }
   }
```

```
if ((fpout2 = fopen(filename3,"w")) == NULL) {
   printf("Error while opening file '%s'\n",filename3);
   return;
}
if ((fpout3 = fopen(filename4,"w")) == NULL) {
   printf("Error while opening file '%s'\n",filename4);
   return;
}

printf("\n* Conversion of .pwr file to .tra_ifp, .tra_lst and .tra_int files\n");
printf("*\n* Default values used in conversion:\n");
printf("*\tVdd = %d V\n",V_DD);
printf("*\tCload = %e F\n",C_LOAD);
printf("*\tMaximum allowed number of nodes = %d\n",MAX_NODE_NUMBER);
printf("* Please note: 'input change time interval' must be a multiple
      of 'control TIME\\STEP'\n");
printf("* Read values: TIME\\STEP = %f ns   input change interval = %f ns\n\n"
      ,TIME_STEP,input_time_step);

number_of_steps = (int)(input_time_step / TIME_STEP);

/* Copy begin (= 8 lines) of .pwr file to buffer */
j = 0;
for (i=0;i<8;i++) {
   while ((c = getc(fpin)) != '\n') {
      buffer[j] = c;
      j++;
   }
   buffer[j]='\n';
   j++;
}
buffer[j] = NULL;


/*******************************
 * Conversion of variable names *
 *******************************/

/* Number of read node names */
node_number = 0;
/* Read power variable name length */
while (fscanf(fpin,"%d",&pwr_var_length) && (pwr_var_length != 0)) {
   /* Read power variable name */
   fscanf(fpin,"%d%s",&dummy,variable_name[node_number]);
   /* Extract node name from power variable name */
   node_name_length = pwr_var_length - PWR_VAR_STRIP_LEN;
   variable_name[node_number][node_name_length] = NULL;

   /* Skip 11 lines (12 end of line characters)                    *
    * The other power variable names of the same node can be ignored */
   for (i=0;i<12;i++) while ((c = getc(fpin)) != '\n');
   /* Increment number of read node variables */
   node_number+=1;
}

/*******************************
 * Create begin of .tra_ifp file *
 *******************************/

if (create_ifp) {
   j = 0;
   /* Copy first 3 lines of .pwr file from buffer */
   for (i=0;i<3;i++) {
      while ((c = buffer[j]) != '\n') {
         putc(c,fpout);
```

```
            j++;
         }
         putc('\n',fpout);
         j++;
      }

      /* Convert 4th line of .pwr file from buffer */
      /* Read the 9 numbers on this line */
      sscanf(buffer+j,"%d%d%d%d%d%d%d%d%d",&d1,&d2,&d3,&d4,&d5,&d6,&d7,&d8,&d9);
      /* Enter new value for number of variables (6th number) in .tra_ifp file */
      d6 = 4*node_number + 1;
      /* Enter new value for number of time steps (7th number) in .tra_ifp file */
      d7 = (int) (float)(d7 + number_of_steps - 1) / (float)number_of_steps;
      fprintf(fpout,"  %7d %7d %7d %7d %7d %7d %7d %7d %7d",d1,d2,d3,d4,d5,d6,d7,d8,d9);


      /* Adapt pointer to new position in buffer */
      while ((c = buffer[j]) != '\n') j++;
      /* Copy next 4 lines including rest of current line to .tra_ifp file */
      for (i=0;i<5;i++) {
         while ((c = buffer[j]) != '\n') {
            putc(c,fpout);
            j++;
         }
         putc('\n',fpout);
         j++;
      }

      /* Enter the new variable names in the .tra_ifp file */
      for (i=0; i < node_number; i++) {
         fprintf(fpout,"%6d %5d  %s:Trans\n",node_name_length+6,0,variable_name[i]);
         fprintf(fpout,"%6d %5d  %s:Useful\n",node_name_length+7,0,variable_name[i]);
         fprintf(fpout,"%6d %5d  %s:Redun\n",node_name_length+6,0,variable_name[i]);
         fprintf(fpout,"%6d %5d  %s:RedunRel\n",node_name_length+9,0,variable_name[i]);
      }

      /* Mark end of variable name list */
      fprintf(fpout,"     0\n");
   }

   /************************
    * Conversion of values *
    ************************/

   /* Initialisation */
   for (i=0; i < node_number; i++) {
      tot_redunrel_value[i] = 0;
      tot_redun_value[i] = 0;
      tot_useful_value[i] = 0;
      power_value[i]=0;
   }
   input_changes = 0;

   /* Copy TIME value */
   fscan_result = fscanf(fpin,"%s",value_string);
   while (fscan_result != EOF) {
      fprintf(fpout," %e",string_to_double(value_string));

      counter = 2;
      /* Calculate the total power over one input time interval */
      for (j=0; j<number_of_steps;j++) {
         for (i=0; i < node_number; i++) {
            /* Skip the following 8 values */
           for (k = 0; k < 8; k++) fscanf(fpin,"%s",value_string);
           /* Read the MAX-DYN value */
```

```
            fscanf(fpin,"%s",value_string);
            /* Calculate the total power in the input time interval */
            power_value[i] += string_to_double(value_string);
            /* Skip the following 3 values */
            for (k = 0; k < 3; k++) fscanf(fpin,"%s",value_string);
         }
         /* Skip the next time value */
         fscan_result = fscanf(fpin,"%s",value_string);
      }
      /* Calculate the converted values */
      for (i=0; i < node_number; i++) {
         trans_value = round(power_value[i] * TIME_STEP * 1E-9 / (C_LOAD * V_DD * V_DD));
         if (trans_value > 0) {
            /* Odd number of transitions -> all minus one redundant */
            if (trans_value % 2 == 1) {
               useful_value = 1;
               tot_useful_value[i] += useful_value;
               redun_value = (trans_value - 1);
               tot_redun_value[i] += redun_value;
            }
            /* Even number of transitions -> all redundant */
            else {
               useful_value = 0;
               redun_value = trans_value;
               tot_redun_value[i] += redun_value;
            }
         }
         else {
            useful_value = 0;
            redun_value = 0;
         }

         if (tot_useful_value[i] != 0) redunrel_value = (float)tot_redun_value[i] /
                                                        (float)tot_useful_value[i];
         else redunrel_value = 0;

         if (create_ifp) {
            fprintf(fpout," %e",(float)trans_value);
            if (((counter++) % 6) == 0) fprintf(fpout,"\n");
            fprintf(fpout," %e",(float)useful_value);
            if (((counter++) % 6) == 0) fprintf(fpout,"\n");
            fprintf(fpout," %e",(float)redun_value);
            if (((counter++) % 6) == 0) fprintf(fpout,"\n");
            fprintf(fpout," %e",redunrel_value);
            if (((counter++) % 6) == 0) fprintf(fpout,"\n");
         }
         /* Clear power value */
         power_value[i] = 0;
      }
      if (create_ifp) fprintf(fpout,"\n");
      input_changes++;
      printf("Step %d\n",input_changes);
   }

   /************************************
    * Create .tra_lst and .tra_int files *
    ************************************/

   fprintf(fpout2,"*\n");
   fprintf(fpout2,"* Results of simulation '%s':\n*\n",filename2);
   fprintf(fpout2,"* USEFUL   = total number of useful transitions\n");
   fprintf(fpout2,"* REDUN    = total number of redundant transitions\n");
   fprintf(fpout2,"* REDUNREL = REDUN/USEFUL\n*\n");
   fprintf(fpout2,"* Total number of input changes: %d\n*\n\n",input_changes);
   fprintf(fpout3,"%d\n%d\n",input_changes,node_number);
```

```
    all_useful_value = 0;
    all_redun_value = 0;
    all_redunrel_value = 0;
    for (i=0; i < node_number; i++) {
       if (tot_useful_value[i] != 0)
          tot_redunrel_value[i] = (float)tot_redun_value[i] / (float)tot_useful_value[i];
       else
          tot_redunrel_value[i] = 0;
       all_useful_value += tot_useful_value[i];
       all_redun_value += tot_redun_value[i];
       fprintf(fpout2,"%s \tUSEFUL: %d \tREDUN: %d \tREDUNREL: %f1\n",
             variable_name[i],tot_useful_value[i],tot_redun_value[i],tot_redunrel_value[i]);
       fprintf(fpout3,"%s\n%d\n%d\n",
        variable_name[i],tot_useful_value[i],tot_redun_value[i]);
    }
    all_redunrel_value = (float)all_redun_value / (float)all_useful_value;
    fprintf(fpout2,"\n*\n* Total number of nodes: %d\n*\n",node_number);
    fprintf(fpout2,"* Total number of useful transitions: %d\n",all_useful_value);
    fprintf(fpout2,"* Total number of redundant transitions: %d\n",all_redun_value);
    fprintf(fpout2,"* Relative number of redundant transitions: %d/%d = %f1\n*\n"
           ,all_redun_value,all_useful_value,all_redunrel_value);
    fprintf(fpout3,"%d\n%d\n",all_useful_value,all_redun_value);
    /* Close files */
    fclose(fpin);
    if (create_ifp) fclose(fpout);
    fclose(fpout2);
    fclose(fpout3);
}
```

D Source code of tra_merge.c

Program to merge intermediate transition activity (.tra_int) files to obtain a single total transition activity (.tra_lst) file.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define MAX_NODE_NUMBER 1000
#define MAX_FILE_NUMBER 5

main(argc,argv) /* Merge .tra_int files */

int argc;
char *argv[];
{
   FILE  *fopen(),*fpin[MAX_FILE_NUMBER+1],*fpout;
   int   i,j,input_changes,useful,redun;
   int   node_number[MAX_FILE_NUMBER+1];
   int   all_useful_value,all_redun_value;
   float all_redunrel_value;
   int   tot_useful_value[MAX_NODE_NUMBER],tot_redun_value[MAX_NODE_NUMBER];
   float tot_redunrel_value[MAX_NODE_NUMBER];
   int   tot_input_changes;
   char  input_filename[MAX_FILE_NUMBER+1][50];
   char  output_filename[50];
   char  variable_name[MAX_NODE_NUMBER][50];

   if (argc == 1) {
      printf("Error: No input file specified\n");
      return;
   }
   if (argc > MAX_FILE_NUMBER+1) {
      printf("Error: Too many input files (maximum of %d allowed)\n",MAX_FILE_NUMBER);
      return;
   }
   for (i = 1; i < argc ; i++) {
      strcpy(input_filename[i],argv[i]);
      strcat(input_filename[i],".tra_int");
      if ((fpin[i] = fopen(input_filename[i],"r")) == NULL) {
         printf("Error while opening file '%s'\n",input_filename[i]);
         return;
      }
   }

   printf("\n*\n* Merging of .tra_int files to .tra_mrg file\n");
   printf("* Default: Maximum allowed number of nodes = %d\n*\n\n",MAX_NODE_NUMBER);

   printf("Specify name of output .tra_mrg file: ");
   scanf("%s",output_filename);
   strcat(output_filename,".tra_mrg");
   if ((fpout = fopen(output_filename,"w")) == NULL) {
      printf("Error while opening file '%s'\n",output_filename);
      return;
   }

   /**************************
    * Calculate merged values *
    **************************/
   tot_input_changes = 0;
   for (i = 1; i < argc ; i++) {
      fscanf(fpin[i],"%d\n%d",&input_changes,&node_number[i]);
```

```
      tot_input_changes += input_changes;
      if (node_number[1] != node_number[1]) {
         printf("Error: Number of nodes in .tra_int files do not match\n");
         return;
      }
   }

   for (j = 0; j < node_number[1]; j++) {
      tot_useful_value[j] = 0;
      tot_redun_value[j] = 0;
      for (i = 1; i < argc ; i++) {
         fscanf(fpin[i],"%s\n%d\n%d",variable_name[j],&useful,&redun);
         tot_useful_value[j] += useful;
         tot_redun_value[j] += redun;
      }
      if (tot_useful_value[j] != 0)
         tot_redunrel_value[j] = (float)tot_redun_value[j] / (float)tot_useful_value[j];
      else
         tot_redunrel_value[j] = 0;
   }
   all_useful_value = 0;
   all_redun_value = 0;
   for (i = 1; i < argc ; i++) {
      fscanf(fpin[i],"%d\n%d",&useful,&redun);
      all_useful_value += useful;
      all_redun_value += redun;
   }
   if (all_useful_value > 0) all_redunrel_value = (float)all_redun_value /
                                                  (float)all_useful_value;
   else all_redunrel_value = 0;

   /*****************************************
    * Create .tra_mrg file in .tra_lst format *
    *****************************************/

   fprintf(fpout,"*\n");
   fprintf(fpout,"* Merged results of simulations: \n");
   for (i = 1; i < argc ; i++) fprintf(fpout,"*\t%s\n",input_filename[i]);
   fprintf(fpout,"*\n* USEFUL   = total number of useful transitions\n");
   fprintf(fpout,"* REDUN    = total number of redundant transitions\n");
   fprintf(fpout,"* REDUNREL = REDUN/USEFUL\n*\n");
   fprintf(fpout,"* Total number of input changes: %d\n*\n\n",tot_input_changes);
   for (i=0; i < node_number[1]; i++) {
      fprintf(fpout,"%s \tUSEFUL: %d \tREDUN: %d \tREDUNREL: %f1\n",variable_name[i]
              ,tot_useful_value[i],tot_redun_value[i],tot_redunrel_value[i]);
   }
   fprintf(fpout,"\n*\n* Total number of nodes: %d\n*\n",node_number[1]);
   fprintf(fpout,"* Total number of useful transitions: %d\n",all_useful_value);
   fprintf(fpout,"* Total number of redundant transitions: %d\n",all_redun_value);
   fprintf(fpout,"* Relative number of redundant transitions: %d/%d = %f1\n*\n"
          ,all_redun_value,all_useful_value,all_redunrel_value);
   /* Close files */
   for (i = 1; i < argc ; i++) fclose(fpin[i]);
   fclose(fpout);
}
```

E VHDL description of direction detector

```
#ifdef SYNTHESIS

#ifndef WORD_ADDERS_12
#define width 12
#include "word_adders.vhdl"
#define WORD_ADDERS 12
#endif

#ifndef WORD_ADDERS_13
#define width 13
#include "word_adders.vhdl"
#define WORD_ADDERS 13
#endif

#ifndef WORD_ADDERS_14
#define width 14
#include "word_adders.vhdl"
#define WORD_ADDERS 14
#endif

#ifndef WORD_REGISTERS_12
#define width 12
#include "word_registers.vhdl"
#define WORD_REGISTERS 12
#endif

#ifndef SSUBTRACT_12
#define l1_width 12
#include "ssubtract.vhdl"
#define SSUBTRACT 12
#endif

#ifndef SSUBTRACT_13
#define l1_width 13
#include "ssubtract.vhdl"
#define SSUBTRACT 13
#endif

#ifndef COMP_MEDIAN_13
#define width 13
#include "comp_median.vhdl"
#define COMP_MEDIAN 13
#endif

#ifndef COMP_MEDIAN_14
#define width 14
#include "comp_median.vhdl"
#define COMP_MEDIAN 14
#endif

#endif

------------------------------------------------------------------
package flagtypes is
   type maxflag is (a2_b0_max, a1_b1_max, a0_b2_max);
   type minflag is (a2_b0_min, a1_b1_min, a0_b2_min);
end;

------------------------------------------------------------------
library synopsys,std,sim_lib,work;
use sim_lib.pack_ssubtract_12.all;
use sim_lib.pack_word_adders_13.all;
```

```
use synopsys.types.all;
use sim_lib.VSynLib.all;
-------------------------------------------------------------
entity abs_diff12 is
   port (a,b     :in  mvl7_vector(11 downto 0);
         absdiff :out mvl7_vector(12 downto 0));
end;

architecture behaviour of abs_diff12 is

   signal a_minus_b: mvl7_vector(12 downto 0);

begin
   subtr_12:ssubtract_12 port map (a,b,a_minus_b);
   abs_13:absval_13 port map (a_minus_b,absdiff);
end behaviour;

-------------------------------------------------------------
library synopsys,std,sim_lib,work;
use sim_lib.pack_ssubtract_13.all;
use sim_lib.pack_word_adders_14.all;
use synopsys.types.all;
use sim_lib.VSynLib.all;
-------------------------------------------------------------
entity abs_diff13 is
   port (a,b     :in  mvl7_vector(12 downto 0);
         absdiff :out mvl7_vector(13 downto 0));
end;

architecture behaviour of abs_diff13 is

   signal a_minus_b: mvl7_vector(13 downto 0);

begin
   subtr_13:ssubtract_13 port map (a,b,a_minus_b);
   abs_14:absval_14 port map (a_minus_b,absdiff);
end behaviour;

-------------------------------------------------------------
library synopsys,std,sim_lib,work;
use synopsys.types.all;
use work.flagtypes.all;
-------------------------------------------------------------
entity find_maxmin is
   port (a2b0_gt_a1b1, a2b0_gt_a0b2, a1b1_gt_a0b2 :in  mvl7;
         maxres                                   :out maxflag;
         minres                                   :out minflag);
end;

architecture behaviour of find_maxmin is
begin
   process(a2b0_gt_a1b1, a2b0_gt_a0b2, a1b1_gt_a0b2)
   begin
      if a2b0_gt_a1b1='0' and a2b0_gt_a0b2='0' and a1b1_gt_a0b2='0' then
         maxres <= a0_b2_max;
         minres <= a2_b0_min;
      elsif a2b0_gt_a1b1='1' and a2b0_gt_a0b2='0' and a1b1_gt_a0b2='0' then
         maxres <= a0_b2_max;
         minres <= a1_b1_min;
      elsif a2b0_gt_a1b1='1' and a2b0_gt_a0b2='1' and a1b1_gt_a0b2='0' then
         maxres <= a2_b0_max;
         minres <= a1_b1_min;
      elsif a2b0_gt_a1b1='0' and a2b0_gt_a0b2='0' and a1b1_gt_a0b2='1' then
         maxres <= a1_b1_max;
         minres <= a2_b0_min;
```

```
      elsif a2b0_gt_a1b1='0' and a2b0_gt_a0b2='1' and a1b1_gt_a0b2='1' then
         maxres <= a1_b1_max;
         minres <= a0_b2_min;
      else
         maxres <= a2_b0_max;
         minres <= a0_b2_min;
      end if;
   end process;
end behaviour;

-------------------------------------------------------------
library synopsys,std,sim_lib,work;
use synopsys.types.all;
use work.flagtypes.all;
-------------------------------------------------------------
entity select_maxmin is
   port (a2_b0, a1_b1, a0_b2 :in  mvl7_vector(12 downto 0);
         is_max              :in  maxflag;
         is_min              :in  minflag;
         max,min             :out mvl7_vector(12 downto 0));
end;

architecture behaviour of select_maxmin is
begin
   process(a2_b0, a1_b1, a0_b2, is_max, is_min)
   begin
      case is_max is
         when a2_b0_max =>
            max <= a2_b0;
         when a1_b1_max =>
            max <= a1_b1;
         when a0_b2_max =>
            max <= a0_b2;
         end case;
      case is_min is
         when a2_b0_min =>
            min <= a2_b0;
         when a1_b1_min =>
            min <= a1_b1;
         when a0_b2_min =>
            min <= a0_b2;
      end case;
   end process;
end behaviour;

-- root entity of direction detector
-------------------------------------------------------------
library synopsys,std,sim_lib,work;
use sim_lib.pack_word_registers_12.all;
use sim_lib.pack_comp_median_13.all;
use sim_lib.pack_comp_median_14.all;
use synopsys.types.all;
use work.flagtypes.all;
-------------------------------------------------------------
entity dirdet is
   port (clk          :in  mvl7;
         a2,b2,thresh :in  mvl7_vector(11 downto 0);
         direct       :out minflag);
end;

architecture struct_behav of dirdet is


   component abs_diff12
      port (a,b     :in  mvl7_vector(11 downto 0);
```

```
            absdiff :out mvl7_vector(12 downto 0));
      end component;

   component abs_diff13
      port (a,b     :in  mvl7_vector(12 downto 0);
            absdiff :out mvl7_vector(13 downto 0));
      end component;

   component find_maxmin
      port (a2b0_gt_a1b1, a2b0_gt_a0b2, a1b1_gt_a0b2 :in  mvl7;
            maxres                                   :out maxflag;
            minres                                   :out minflag);
      end component;

   component select_maxmin
      port (a2_b0, a1_b1, a0_b2 :in  mvl7_vector(12 downto 0);
            is_max              :in  maxflag;
            is_min              :in  minflag;
            max,min             :out mvl7_vector(12 downto 0));
      end component;

   signal a1: mvl7_vector(11 downto 0);
   signal a0: mvl7_vector(11 downto 0);

   signal b1: mvl7_vector(11 downto 0);
   signal b0: mvl7_vector(11 downto 0);

   signal a2_b0: mvl7_vector (12 downto 0);
   signal a1_b1: mvl7_vector (12 downto 0);
   signal a0_b2: mvl7_vector (12 downto 0);

   signal a2b0_gt_a1b1: mvl7;
   signal a2b0_gt_a0b2: mvl7;
   signal a1b1_gt_a0b2: mvl7;

   signal is_max: maxflag;
   signal is_min: minflag;

   signal max: mvl7_vector(12 downto 0);
   signal min: mvl7_vector(12 downto 0);

   signal max_min: mvl7_vector (13 downto 0);

   signal ext_thresh: mvl7_vector (13 downto 0);

   signal max_min_gt_thresh: mvl7;

begin

   ra1:word_regist_12 port map (clk,a2,a1);
   ra0:word_regist_12 port map (clk,a1,a0);

   rb1:word_regist_12 port map (clk,b2,b1);
   rb0:word_regist_12 port map (clk,b1,b0);

   abs_a2_b0:abs_diff12 port map (a2,b0,a2_b0);
   abs_a1_b1:abs_diff12 port map (a1,b1,a1_b1);
   abs_a0_b2:abs_diff12 port map (a0,b2,a0_b2);

   comp_a2b0_a1b1:sgreater_13 port map (a2_b0, a1_b1, a2b0_gt_a1b1);
   comp_a2b0_a0b2:sgreater_13 port map (a2_b0, a0_b2, a2b0_gt_a0b2);
   comp_a1b1_a0b2:sgreater_13 port map (a1_b1, a0_b2, a1b1_gt_a0b2);

   fmm:find_maxmin port map (a2b0_gt_a1b1, a2b0_gt_a0b2, a1b1_gt_a0b2, is_max, is_min);
   smm:select_maxmin port map (a2_b0, a1_b1, a0_b2, is_max, is_min, max, min);
```

```
   amm:abs_diff13 port map (max, min, max_min);

   f1:for i in 0 to 11 generate
      g1:ext_thresh(i) <= thresh(i);
   end generate;
   ext_thresh(12) <= thresh(11);
   ext_thresh(13) <= thresh(11);

   comp_mm_thresh:sgreater_14 port map (max_min, ext_thresh, max_min_gt_thresh);

   process (max_min_gt_thresh,is_min)
      begin
         if max_min_gt_thresh='1' then direct <= is_min;
         else direct <= a1_b1_min;
         end if;
      end process;
end struct_behav;


-------------------------------------------------------------------------------
library synopsys,std,sim_lib,work;
use synopsys.types.all;
use sim_lib.VSynLib.all;
use work.flagtypes.all;
-------------------------------------------------------------------------------
package pack_dirdet is
   component dirdet
      port (clk          :in  mvl7;
            a2,b2,thresh :in  mvl7_vector(11 downto 0);
            direct       :out minflag);
   end component;
end pack_dirdet;
```

F VHDL workbench description for direction detector

```
------------------------------------------------------------
library synopsys,std,sim_lib;
use work.flagtypes.all;
------------------------------------------------------------
package flagfunction is
   function minflagtoi(x: minflag) return INTEGER;
end;

package body flagfunction is
   function minflagtoi(x: minflag) return INTEGER is
      variable result: integer;
      begin
         if x = a2_b0_min then result := 120;
         elsif x = a1_b1_min then result := 111;
         else result := 102;
         end if;
         return result;
      end;
end;

------------------------------------------------------------
library synopsys,std,sim_lib;
use synopsys.types.all;
use synopsys.arithsupp.all;
use std.textio.all;
use sim_lib.my_mvl7textio.all;
use work.pack_dirdet.all;
use work.flagtypes.all;
use work.flagfunction.all;
------------------------------------------------------------
entity test_dirdet is
end test_dirdet;

architecture structural of test_dirdet is

file inpfile: text is in "dirdet.inp";
file resfile: text is out "dirdet.res";


   signal clk: mvl7;
   signal thresh: mvl7_vector(11 downto 0);
   signal direct: minflag;

   signal a2: mvl7_vector(11 downto 0);
   signal a1: mvl7_vector(11 downto 0);
   signal a0: mvl7_vector(11 downto 0);

   signal b2: mvl7_vector(11 downto 0);
   signal b1: mvl7_vector(11 downto 0);
   signal b0: mvl7_vector(11 downto 0);

   signal a2_b0: mvl7_vector (12 downto 0);
   signal a1_b1: mvl7_vector (12 downto 0);
   signal a0_b2: mvl7_vector (12 downto 0);

   signal a2b0_gt_a1b1: mvl7;
   signal a2b0_gt_a0b2: mvl7;
   signal a1b1_gt_a0b2: mvl7;

   signal is_max: maxflag;
   signal is_min: minflag;
```

```
   signal max: mvl7_vector(12 downto 0);
   signal min: mvl7_vector(12 downto 0);

   signal max_min: mvl7_vector (13 downto 0);

   signal ext_thresh: mvl7_vector (13 downto 0);

   signal max_min_gt_thresh: mvl7;

begin

   test:dirdet port map(clk,a2,b2,thresh,direct);

   process
   begin
      loop
         clk <= '1';
         wait for 5 ns;
         clk <= '0';
         wait for 5 ns;
      end loop;
   end process;

   process
      variable inpline, resline: line;
      variable title: string (1 to 72) := "  time  clk        a2
                     b2 threshold direction exp_direction error ";
      variable uline: string (1 to 72) := "-----------------------------
                     ---------------------------------------- ";
      variable ermsg: string(1 to 6) := "   e  ";

      variable s_a2: integer;
      variable s_b2: integer;
      variable s_thresh: integer;

      variable exp_direction: integer;
      variable good, error, flag: boolean := true;

   begin
      write (resline, title);
      writeline (resfile, resline);
      write (resline, uline);
      writeline (resfile, resline);

      loop
         error :=true;
         readline(inpfile, inpline);
         read (inpline, s_a2, good);
         read (inpline, s_b2, good);
         read (inpline, s_thresh, good);
         read (inpline, exp_direction, good);
         wait for 2 ns;

         a2 <= itomvl7v(s_a2,12);
         b2 <= itomvl7v(s_b2,12);
         thresh <= itomvl7v(s_thresh,12);

         assert good report "Error occurred during file read-operation"
                           severity failure;
         wait for 5 ns;

         write (resline, now, right, 6);
         write (resline, clk, right, 5);
         write (resline, smvl7vtoi(a2), right, 10);
         write (resline, smvl7vtoi(b2), right, 10);
```

```
         write (resline, smvl7vtoi(thresh), right, 10);
         write (resline, minflagtoi(direct), right, 10);
         write (resline, exp_direction, right, 14);

         if (exp_direction /= minflagtoi(direct)) then
            error := false;
            flag := false;
            write (resline, ermsg);
         end if;

         writeline (resfile, resline);
         assert error report "Expected output is different from actual output;
                              simulation continues"  severity warning;
         assert error report "Check ""e""-messages in resultsfile !!!"
                              severity warning;
         wait for 3 ns;
      end loop;

      assert flag report "Inconsistencies found between expected output and
                          actual output !!!"  severity warning;

   end process;
end structural;

configuration struct_test_dirdet of test_dirdet is
   for structural
      for all : dirdet use entity work.dirdet(struct_behav);
      end for;
   end for;
end;

configuration behav_test_dirdet of test_dirdet is
   for structural
      for all : dirdet use entity work.dirdet(struct_behav);
      end for;
   end for;
end;
```

G Functional description of direction detector in C

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define THRESHOLD_VALUE 1024
#define NR_OF_STIMULI 100
#define WORDLENGTH 12
#define FILENAME "dirdet.inp"

#define FALSE 0
#define TRUE 1
typedef int boolean;

#define a2_b0_max 220
#define a1_b1_max 211
#define a0_b2_max 202
#define a2_b0_min 120
#define a1_b1_min 111
#define a0_b2_min 102
typedef int maxflag;
typedef int minflag;

int power(x,n) int x,n; /* Raise x to n-th power; n > 0 */
{
   int i,p;

   p = 1;
   for (i = 1; i <= n; ++i) p = p * x;
   return(p);
}

int round(x) float x;
{
   int y;
   y = (int) x;
   if (x - y >= 0.5) y++;
   return(y);
}

int *find_maxmin(a,b,c) boolean a,b,c;
{
   maxflag maxflg;
   minflag minflg;
   int result[2];

   if ((a == FALSE) && (b == FALSE) && (c == FALSE)) {
      maxflg = a0_b2_max;
      minflg = a2_b0_min;
   }
   else if ((a == TRUE) && (b == FALSE) && (c == FALSE)) {
      maxflg = a0_b2_max;
      minflg = a1_b1_min;
   }
   else if ((a == TRUE) && (b == TRUE) && (c == FALSE)) {
      maxflg = a2_b0_max;
      minflg = a1_b1_min;
   }
   else if ((a == FALSE) && (b == FALSE) && (c == TRUE)) {
      maxflg = a1_b1_max;
      minflg = a2_b0_min;
   }
   else if ((a == FALSE) && (b == TRUE) && (c == TRUE)) {
```

```
      maxflg = a1_b1_max;
      minflg = a0_b2_min;
   }
   else if ((a == TRUE) && (b == TRUE) && (c == TRUE)) {
      maxflg = a2_b0_max;
      minflg = a0_b2_min;
   }
   result[1] = maxflg;
   result[0] = minflg;
   return(result);
}

int *select_maxmin(a2_b0,a1_b1,a0_b2,is_max,is_min)
   int a2_b0,a1_b1,a0_b2;
   maxflag is_max;
   minflag is_min;
{
   int max;
   int min;
   int result[2];

   switch(is_max) {
      case a2_b0_max: max = a2_b0;break;
      case a1_b1_max: max = a1_b1;break;
      case a0_b2_max: max = a0_b2;break;
   }
   switch(is_min) {
      case a2_b0_min: min = a2_b0;break;
      case a1_b1_min: min = a1_b1;break;
      case a0_b2_min: min = a0_b2;break;
   }
   result[1] = max;
   result[0] = min;
   return(result);
}

main()  /* Simulation of direction detector */
{
   FILE *fopen(), *fpout;
   int i;
   int a2,a1,a0,b2,b1,b0,threshold;
   int sign,max_inp_value;
   int a2_b0,a1_b1,a0_b2;
   boolean a2b0_gt_a1b1,a2b0_gt_a0b2,a1b1_gt_a0b2;
   int *result_pointer;
   int is_maxmin[2];  /* maxflag, minflag */
   int maxmin[2];     /* integer */
   int max_min;
   boolean max_min_gt_thresh;
   minflag direction;

   if ((fpout = fopen(FILENAME,"w")) == NULL) {
      printf("Error while opening file '%s'\n",FILENAME);
      return;
   }

   max_inp_value = power(2,WORDLENGTH-1)-1; /* 2's complement */

   threshold = THRESHOLD_VALUE;

   /* Initialisation */
   a1 = 0;a2 = 0;
   b1 = 0;b2 = 0;

   for (i = 0; i < NR_OF_STIMULI ;i++) {
```

```
        /* Generate input a[i] */
        a0 = a1;
        a1 = a2;
        if (drand48() > 0.5) sign = -1;
        else sign = 1;
        a2 = sign * round(max_inp_value * drand48());

        /* Generate input b[i] */
        b0 = b1;
        b1 = b2;
        if (drand48() > 0.5) sign = -1;
        else sign = 1;
        b2 = sign * round(max_inp_value * drand48());

        /* Generate random threshold input */
        threshold = round(max_inp_value * drand48());

        /* Calculate the absolute differences */
        a2_b0 = abs(a2-b0);
        a1_b1 = abs(a1-b1);
        a0_b2 = abs(a0-b2);

        a2b0_gt_a1b1 = (a2_b0 > a1_b1);
        a2b0_gt_a0b2 = (a2_b0 > a0_b2);
        a1b1_gt_a0b2 = (a1_b1 > a0_b2);

        result_pointer = find_maxmin(a2b0_gt_a1b1,a2b0_gt_a0b2,a1b1_gt_a0b2);
        is_maxmin[1] = result_pointer[1];
        is_maxmin[0] = result_pointer[0];
        result_pointer = select_maxmin(a2_b0,a1_b1,a0_b2,is_maxmin[1],is_maxmin[0]);
        maxmin[1] = result_pointer[1];
        maxmin[0] = result_pointer[0];

        max_min = abs(maxmin[1]-maxmin[0]);
        max_min_gt_thresh = (max_min > threshold);

        if (max_min_gt_thresh) direction = is_maxmin[0];
        else direction = a1_b1_min;

        /* Output to file */
        fprintf(fpout,"%10d %10d %10d %10d\n",a2,b2,threshold,direction);
    }
    /* Close file */
    fclose(fpout);
}
```

H Transition activity in dirdet for random signals

Results obtained from SWITCH using functional blocks with unit delay modelling and 4320 random inputs. Numbers appended to nodenames refer to bitnumbers.

nodename	*number of useful transitions*	*number of redundant transitions*	*ratio redundant/useful*
VDIRECT0	2090	23020	11.014354l
VDIRECT1	1610	13532	8.404969l
VMMGTTHRESH	1629	27836	17.087784l
VMM0	2176	14928	6.860294l
VMM1	2131	22306	10.467386l
VMM2	2148	24750	11.522346l
VMM3	2143	25868	12.070929l
VMM4	2187	26280	12.016461l
VMM5	2166	26670	12.313020l
VMM6	2141	26904	12.566091l
VMM7	2200	26996	12.270909l
VMM8	2115	26992	12.762175l
VMM9	2147	26822	12.492781l
VMM10	2205	26906	12.202268l
VMM11	1730	26810	15.497109l
VMM12	0	20994	undefined
VMM13	0	21108	undefined
VMIN0	2162	10812	5.000925l
VMIN1	2170	12700	5.852535l
VMIN2	2187	13304	6.083219l
VMIN3	2145	13750	6.410256l
VMIN4	2155	13898	6.449188l
VMIN5	2209	14090	6.378452l
VMIN6	2166	14016	6.470914l
VMIN7	2146	14196	6.615098l
VMIN8	2060	13820	6.708738l
VMIN9	1919	13982	7.286087l
VMIN10	1222	13794	11.288053l
VMIN11	130	10394	79.953850l
VMIN12	0	5626	undefined
VMAX0	2172	10850	4.995396l
VMAX1	2081	13040	6.266218l
VMAX2	2173	13686	6.298205l

nodename	*number of useful transitions*	*number of redundant transitions*	*ratio redundant/useful*
VMAX3	2118	14180	6.6949951
VMAX4	2165	14482	6.6891461
VMAX5	2160	14398	6.6657411
VMAX6	2178	14398	6.6106521
VMAX7	2130	14424	6.7718311
VMAX8	2111	14588	6.9104691
VMAX9	2092	14630	6.9933081
VMAX10	2111	14274	6.7617241
VMAX11	1898	10962	5.7755531
VMAX12	0	7256	undefined
VISMIN0	1933	10598	5.4826691
VISMIN1	1982	9974	5.0322901
VISMAX0	2014	9996	4.9632571
VISMAX1	1996	11844	5.9338681
VA1B1GTA0B2	2266	10668	4.7078551
VA2B0GTA0B2	2212	10352	4.6799281
VA2B0GTA1B1	2247	10578	4.7076101
VA0B2N0	2172	1686	0.7762431
VA0B2N1	2131	4096	1.9221021
VA0B2N2	2137	5328	2.4932151
VA0B2N3	2164	5888	2.7208871
VA0B2N4	2123	6218	2.9288741
VA0B2N5	2172	6458	2.9732961
VA0B2N6	2128	6526	3.0667291
VA0B2N7	2162	6372	2.9472711
VA0B2N8	2127	6444	3.0296191
VA0B2N9	2174	6466	2.9742411
VA0B2N10	2065	6682	3.2358351
VA0B2N11	1694	4280	2.5265641
VA0B2N12	0	6018	undefined
VA1B1N0	2122	1606	0.7568331
VA1B1N1	2172	3896	1.7937381
VA1B1N2	2155	5224	2.4241301
VA1B1N3	2131	5868	2.7536371
VA1B1N4	2138	6230	2.9139381
VA1B1N5	2137	6298	2.9471221

nodename	*number of useful transitions*	*number of redundant transitions*	*ratio redundant/useful*
VA1B1N6	2167	6348	2.929395l
VA1B1N7	2100	6444	3.068571l
VA1B1N8	2169	6444	2.970954l
VA1B1N9	2105	6590	3.130641l
VA1B1N10	2018	6564	3.252725l
VA1B1N11	1592	4274	2.684673l
VA1B1N12	0	5936	undefined
VA2B0N0	2156	1502	0.696660l
VA2B0N1	2200	3838	1.744545l
VA2B0N2	2149	5130	2.387157l
VA2B0N3	2176	5756	2.645221l
VA2B0N4	2140	6174	2.885047l
VA2B0N5	2154	6376	2.960074l
VA2B0N6	2182	6288	2.881760l
VA2B0N7	2113	6502	3.077142l
VA2B0N8	2124	6454	3.038606l
VA2B0N9	2116	6494	3.068998l
VA2B0N10	2009	6624	3.297163l
VA2B0N11	1618	4368	2.699629l
VA2B0N12	0	5998	undefined
VB00	2128	0	0
VB01	2148	0	0
VB02	2205	0	0
VB03	2149	0	0
VB04	2126	0	0
VB05	2142	0	0
VB06	2095	0	0
VB07	2180	0	0
VB08	2214	0	0
VB09	2165	0	0
VB010	2183	0	0
VB011	2133	0	0
VB10	2127	0	0
VB11	2147	0	0
VB12	2205	0	0
VB13	2149	0	0

nodename	*number of useful transitions*	*number of redundant transitions*	*ratio redundant/useful*
VB14	2125	0	0
VB15	2142	0	0
VB16	2094	0	0
VB17	2180	0	0
VB18	2214	0	0
VB19	2165	0	0
VB110	2183	0	0
VB111	2132	0	0
VA00	2176	0	0
VA01	2225	0	0
VA02	2145	0	0
VA03	2196	0	0
VA04	2136	0	0
VA05	2202	0	0
VA06	2109	0	0
VA07	2196	0	0
VA08	2154	0	0
VA09	2181	0	0
VA010	2192	0	0
VA011	2187	0	0
VA10	2175	0	0
VA11	2225	0	0
VA12	2144	0	0
VA13	2196	0	0
VA14	2135	0	0
VA15	2201	0	0
VA16	2108	0	0
VA17	2195	0	0
VA18	2154	0	0
VA19	2181	0	0
VA110	2192	0	0
VA111	2186	0	0

Total number of nodes: 137

Total number of useful transitions: 272842
Total number of redundant transitions: 1033970
Relative number of redundant transitions: 1033970/272842 = 3.7896291

I Transition activity in dirdet for video signals

Results obtained from SWITCH using functional blocks with unit delay modelling and 103680 video signal inputs. Numbers appended to nodenames refer to bitnumbers.

nodename	*number of useful transitions*	*number of redundant transitions*	*ratio redundant/useful*
VDIRECT0	21846	506976	23.2068121
VDIRECT1	14836	265996	17.9290901
VMMGTTHRESH	22107	685206	30.9949801
VMM0	51624	343168	6.6474511
VMM1	47734	522856	10.9535341
VMM2	43854	565786	12.9015831
VMM3	43854	574500	13.1002871
VMM4	43854	587524	13.3972731
VMM5	50948	622006	12.2086441
VMM6	49100	670518	13.6561711
VMM7	44678	688786	15.4166711
VMM8	35791	691130	19.3101621
VMM9	28209	669112	23.7198071
VMM10	22352	651004	29.1250901
VMM11	21524	615246	28.5841851
VMM12	0	431820	undefined
VMM13	0	442048	undefined
VMIN0	51040	223938	4.3875001
VMIN1	35938	218866	6.0901001
VMIN2	35938	230080	6.4021371
VMIN3	35938	231264	6.4350821
VMIN4	35938	235484	6.5525071
VMIN5	48046	266734	5.5516381
VMIN6	42494	279868	6.5860591
VMIN7	31618	278998	8.8240241
VMIN8	19664	255392	12.9877951
VMIN9	11118	224688	20.2093911
VMIN10	7992	192314	24.0633131
VMIN11	8276	124582	15.0534081
VMIN12	0	52278	undefined
VMAX0	51370	236052	4.5951331
VMAX1	35380	243528	6.8832111
VMAX2	35380	252496	7.1366871

nodename	*number of useful transitions*	*number of redundant transitions*	*ratio redundant/useful*
VMAX3	35380	254678	7.1983601
VMAX4	35380	260886	7.3738271
VMAX5	50568	291172	5.7580291
VMAX6	48930	305476	6.2431231
VMAX7	45428	304938	6.7125561
VMAX8	36719	282024	7.6806011
VMAX9	26577	258940	9.7430111
VMAX10	18558	235220	12.6748571
VMAX11	14790	181972	12.3037191
VMAX12	0	152276	undefined
VISMIN0	46643	221006	4.7382461
VISMIN1	40064	197868	4.9387981
VISMAX0	45943	203374	4.4266591
VISMAX1	47502	255266	5.3737951
VA1B1GTA0B2	51767	218534	4.2214921
VA2B0GTA0B2	47431	218316	4.6028121
VA2B0GTA1B1	52849	218062	4.1261331
VA0B2N0	51784	29126	0.5624521
VA0B2N1	38908	93256	2.3968341
VA0B2N2	38908	104778	2.6929681
VA0B2N3	38908	105804	2.7193381
VA0B2N4	38908	108074	2.7776811
VA0B2N5	50795	97516	1.9197951
VA0B2N6	48942	117334	2.3974091
VA0B2N7	44248	129632	2.9296691
VA0B2N8	35624	133580	3.7497191
VA0B2N9	28884	127016	4.3974521
VA0B2N10	23493	117428	4.9984251
VA0B2N11	21402	58130	2.7161011
VA0B2N12	0	87344	undefined
VA1B1N0	51722	28698	0.5548511
VA1B1N1	39141	94212	2.4069901
VA1B1N2	39141	105984	2.7077491
VA1B1N3	39141	107024	2.7343191
VA1B1N4	39141	109288	2.7921621
VA1B1N5	50368	98582	1.9572351

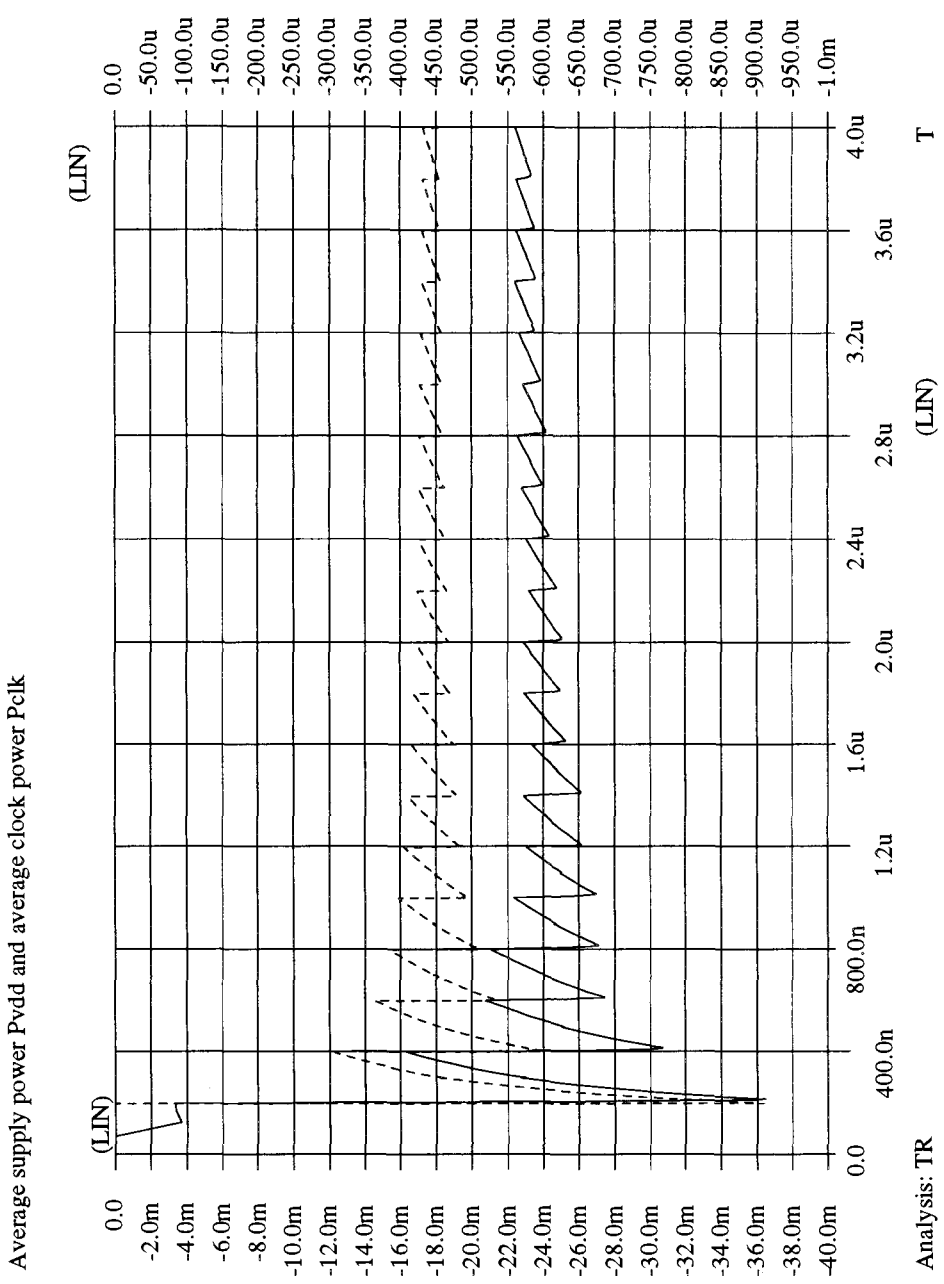
nodename	*number of useful transitions*	*number of redundant transitions*	*ratio redundant/useful*
VA1B1N6	48328	119636	2.475501l
VA1B1N7	43034	131652	3.059255l
VA1B1N8	32348	137142	4.239582l
VA1B1N9	24972	129732	5.195098l
VA1B1N10	20524	118614	5.779283l
VA1B1N11	19512	59352	3.041821l
VA1B1N12	0	86530	undefined
VA2B0N0	51402	30768	0.598576l
VA2B0N1	39400	94418	2.396396l
VA2B0N2	39400	106264	2.697056l
VA2B0N3	39400	107400	2.725888l
VA2B0N4	39400	109728	2.784975l
VA2B0N5	51230	99098	1.934374l
VA2B0N6	48876	119354	2.441976l
VA2B0N7	44771	129890	2.901208l
VA2B0N8	35653	135176	3.791434l
VA2B0N9	28427	129006	4.538150l
VA2B0N10	22982	119822	5.213733l
VA2B0N11	21120	59480	2.816288l
VA2B0N12	0	87804	undefined
VB00	50197	0	0
VB01	50197	0	0
VB02	50197	0	0
VB03	50197	0	0
VB04	50197	0	0
VB05	50175	0	0
VB06	49860	0	0
VB07	46636	0	0
VB08	40442	0	0
VB09	28590	0	0
VB010	21598	0	0
VB011	13630	0	0
VB10	50179	0	0
VB11	50179	0	0
VB12	50179	0	0
VB13	50179	0	0

nodename	*number of useful transitions*	*number of redundant transitions*	*ratio redundant/useful*
VB14	50179	0	0
VB15	50174	0	0
VB16	49860	0	0
VB17	46636	0	0
VB18	40442	0	0
VB19	28566	0	0
VB110	21574	0	0
VB111	13606	0	0
VA00	50554	0	0
VA01	50554	0	0
VA02	50554	0	0
VA03	50554	0	0
VA04	50554	0	0
VA05	50527	0	0
VA06	50182	0	0
VA07	46956	0	0
VA08	40724	0	0
VA09	28784	0	0
VA010	21738	0	0
VA011	13654	0	0
VA10	50541	0	0
VA11	50541	0	0
VA12	50541	0	0
VA13	50541	0	0
VA14	50541	0	0
VA15	50523	0	0
VA16	50179	0	0
VA17	46953	0	0
VA18	40721	0	0
VA19	28760	0	0
VA110	21714	0	0
VA111	13630	0	0

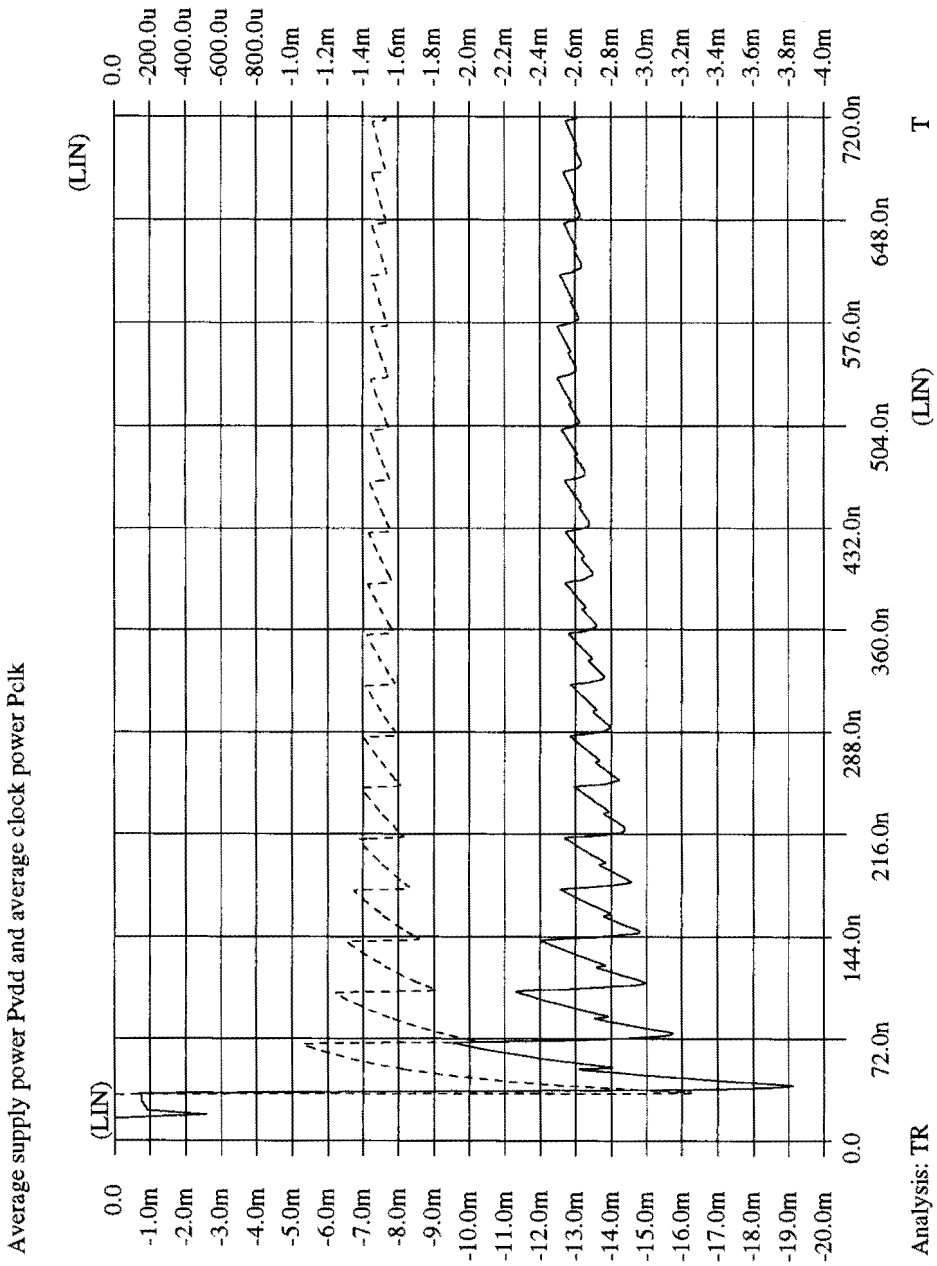
Total number of nodes: 137

Total number of useful transitions: 5057396
Total number of redundant transitions: 21103924
Relative number of redundant transitions: 21103924/5057396 = 4.1728841

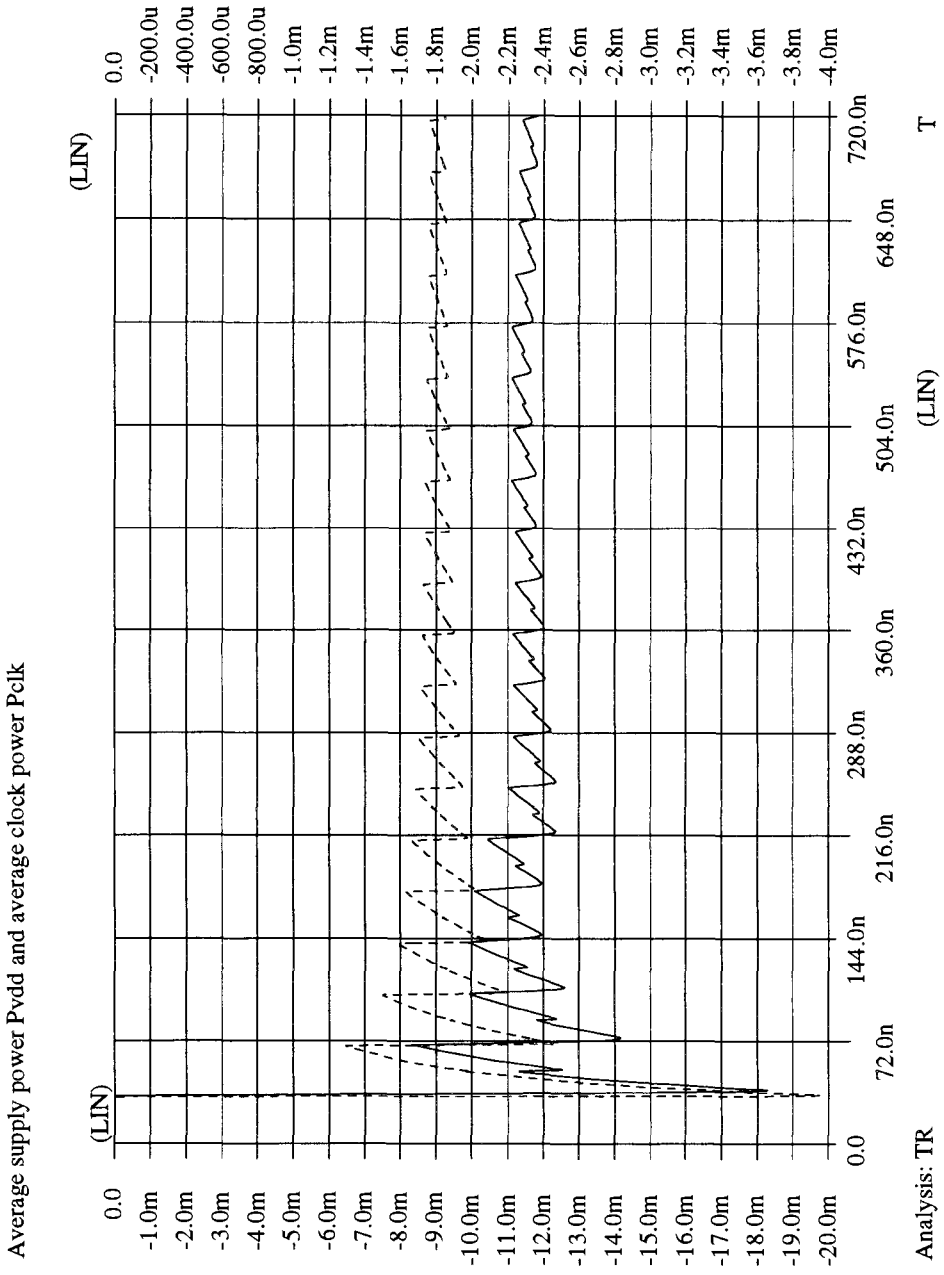
J Average power dissipation in dirdet (5 MHz)



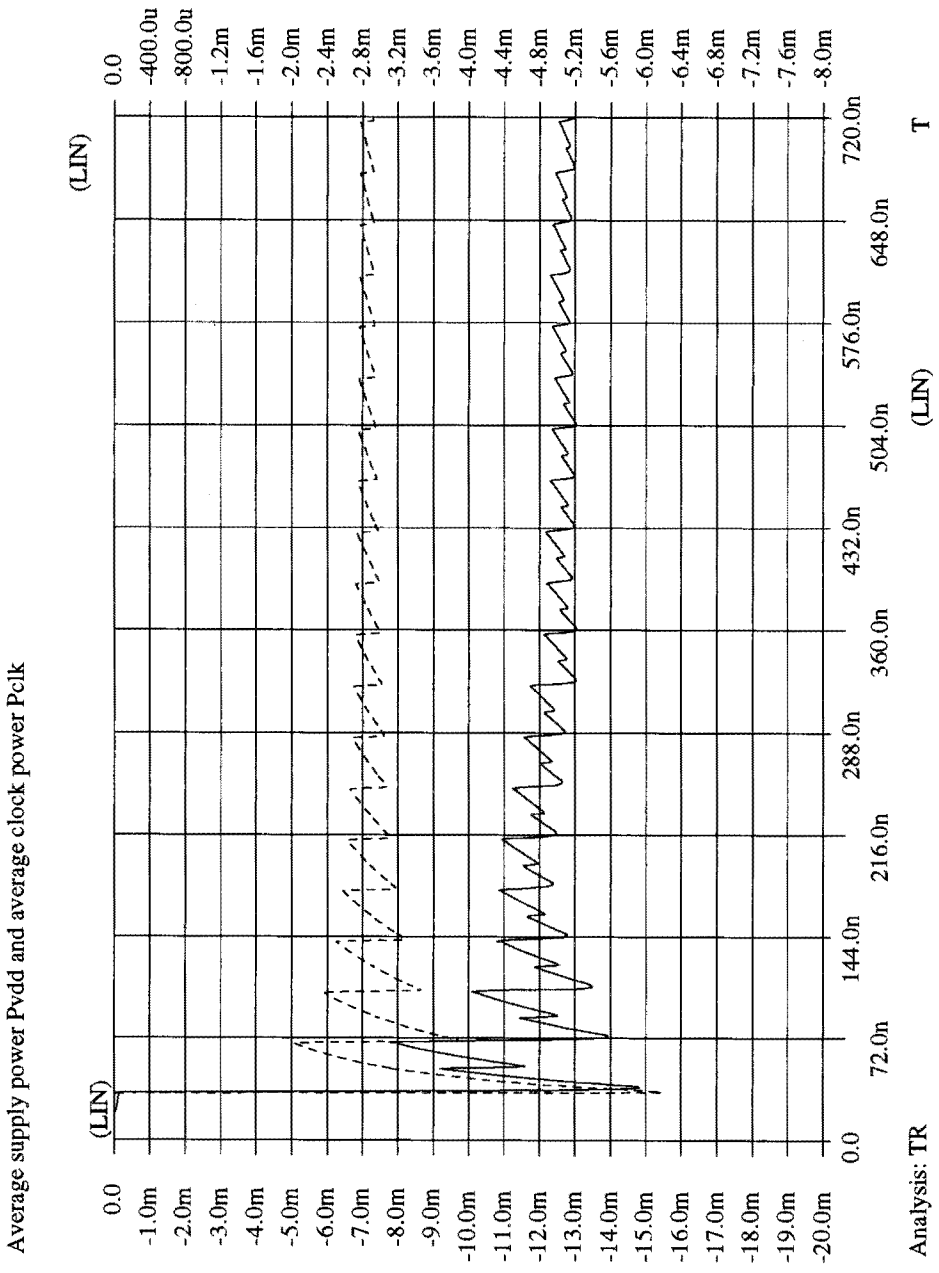
K Average power dissipation in dirdet (27.8 MHz (1))



L Average power dissipation in dirdet (27.8 MHz (2))



M Average power dissipation in dirdet (50 MHz)



N VHDL description of *SGREATER13* comparator

```
library synopsys;
use synopsys.types.all;
-------------------------------------------------
package VSynLib is
  function unknown (a: in mvl7_vector) return boolean;
  function greater (a, b: in mvl7_vector) return boolean;  -- VSyn: primitive GT
end VSynLib;

package body VSynLib is

   function unknown (a: in mvl7_vector) return boolean is
      variable result: boolean;
      variable Xflag: mvl7 := '0';
   begin
      for i in 0 to a'left loop
         case a(i) is
            when '0' => NULL;
            when '1' => NULL;
            when others => Xflag := '1';
         end case;
      end loop;
      if Xflag = '1' then
         result := true;
      else
         result := false;
      end if;
      return result;
   end;

   function greater (a, b: in mvl7_vector) return boolean is
-- only unsigned comparison is implemented
      variable result: boolean;
   begin
      if a > b then
         result := true;
      else
         result := false;
      end if;
      return result;
   end;

end VSynLib;

-------------------------------------------------
-------------------------------------------------
library synopsys,std,sim_lib;
use synopsys.types.all;
use sim_lib.VSynLib.all;
-------------------------------------------------
-- SIGNED GREATER
-------------------------------------------------
-------------------------------------------------
entity sgreater_13 is
   port ( a, b   : in mvl7_vector(13-1 downto 0);
          a_gt_b : out mvl7);
end sgreater_13;

architecture structural of sgreater_13 is
begin
   a_gt_b <= b(13-1) WHEN (a(13-1) XOR b(13-1)) = '1' ELSE
--VSyn: synthesis off
             'X'     WHEN unknown(a) OR unknown(b) ELSE
```

```
--VSyn: synthesis on
             '1'     WHEN greater(a,b) ELSE
             '0';
end structural;

--VSyn: synthesis off

architecture behavioural of sgreater_13 is
begin
   a_gt_b <= b(13-1) WHEN (a(13-1) XOR b(13-1)) = '1' ELSE
             'X'     WHEN unknown(a) OR unknown(b) ELSE
             '1'     WHEN greater(a,b) ELSE
             '0';
end behavioural;

configuration struct_sgreater_13 of sgreater_13 is
   for structural
   end for;
end;

configuration behav_sgreater_13 of sgreater_13 is
   for behavioural
   end for;
end;
--VSyn: synthesis on
```

O VHDL description of 16 bit tree type comparator

```
------------------------------------------------------------
library synopsys,std,sim_lib,work;
use synopsys.types.all;
------------------------------------------------------------
entity bitcompare is
   port (a,b                :in  mvl7;
         greater,equal,less :out mvl7);
end;

architecture behaviour of bitcompare is

   signal nand_ab: mvl7;
   signal and_anab: mvl7;
   signal and_bnab: mvl7;

begin
   process(a,b,nand_ab,and_anab,and_bnab)
   begin
      nand_ab <= not (a and b);
      and_anab <= a and nand_ab;
      and_bnab <= b and nand_ab;
      equal <= not (and_anab or and_bnab);
      greater <= and_anab;
      less <= and_bnab;
   end process;
end;

------------------------------------------------------------
library synopsys,std,sim_lib,work;
use synopsys.types.all;
------------------------------------------------------------
entity magncompare is
   port (greater1,equal1,less1 :in  mvl7;
         greater0,equal0,less0 :in  mvl7;
         greater,equal,less    :out mvl7);
end;

architecture behaviour of magncompare is

   signal nand_g0e1: mvl7;
   signal nand_e1l0: mvl7;

begin
   process(greater1,equal1,less1,greater0,equal0,less0,nand_g0e1,nand_e1l0)
   begin
      nand_g0e1 <= not (greater0 and equal1);
      nand_e1l0 <= not (equal1 and less0);
      greater <= not (not greater1 and nand_g0e1);
      equal <= equal1 and equal0;
      less <= not (not less1 and nand_e1l0);
   end process;
end;

------------------------------------------------------------
library synopsys,std,sim_lib,work;
use synopsys.types.all;
------------------------------------------------------------
entity greater16 is
   port (a,b :in  mvl7_vector(15 downto 0);
         res :out mvl7);
end;
```

```
architecture structure of greater16 is

   component bitcompare
      port (a,b                :in  mvl7;
            greater,equal,less :out mvl7);
   end component;

   component magncompare
      port (greater1,equal1,less1 :in  mvl7;
            greater0,equal0,less0 :in  mvl7;
            greater,equal,less    :out mvl7);
   end component;

   signal a15GRb15: mvl7;
   signal a15EQb15: mvl7;
   signal a15LEb15: mvl7;
   signal a14GRb14: mvl7;
   signal a14EQb14: mvl7;
   signal a14LEb14: mvl7;
   signal a13GRb13: mvl7;
   signal a13EQb13: mvl7;
   signal a13LEb13: mvl7;
   signal a12GRb12: mvl7;
   signal a12EQb12: mvl7;
   signal a12LEb12: mvl7;
   signal a11GRb11: mvl7;
   signal a11EQb11: mvl7;
   signal a11LEb11: mvl7;
   signal a10GRb10: mvl7;
   signal a10EQb10: mvl7;
   signal a10LEb10: mvl7;
   signal a9GRb9: mvl7;
   signal a9EQb9: mvl7;
   signal a9LEb9: mvl7;
   signal a8GRb8: mvl7;
   signal a8EQb8: mvl7;
   signal a8LEb8: mvl7;
   signal a7GRb7: mvl7;
   signal a7EQb7: mvl7;
   signal a7LEb7: mvl7;
   signal a6GRb6: mvl7;
   signal a6EQb6: mvl7;
   signal a6LEb6: mvl7;
   signal a5GRb5: mvl7;
   signal a5EQb5: mvl7;
   signal a5LEb5: mvl7;
   signal a4GRb4: mvl7;
   signal a4EQb4: mvl7;
   signal a4LEb4: mvl7;
   signal a3GRb3: mvl7;
   signal a3EQb3: mvl7;
   signal a3LEb3: mvl7;
   signal a2GRb2: mvl7;
   signal a2EQb2: mvl7;
   signal a2LEb2: mvl7;
   signal a1GRb1: mvl7;
   signal a1EQb1: mvl7;
   signal a1LEb1: mvl7;
   signal a0GRb0: mvl7;
   signal a0EQb0: mvl7;
   signal a0LEb0: mvl7;

   signal gr15_14: mvl7;
   signal eq15_14: mvl7;
   signal le15_14: mvl7;
```

```
    signal gr13_12: mvl7;
    signal eq13_12: mvl7;
    signal le13_12: mvl7;
    signal gr11_10: mvl7;
    signal eq11_10: mvl7;
    signal le11_10: mvl7;
    signal gr9_8: mvl7;
    signal eq9_8: mvl7;
    signal le9_8: mvl7;
    signal gr7_6: mvl7;
    signal eq7_6: mvl7;
    signal le7_6: mvl7;
    signal gr5_4: mvl7;
    signal eq5_4: mvl7;
    signal le5_4: mvl7;
    signal gr3_2: mvl7;
    signal eq3_2: mvl7;
    signal le3_2: mvl7;
    signal gr1_0: mvl7;
    signal eq1_0: mvl7;
    signal le1_0: mvl7;

    signal gr15_12: mvl7;
    signal eq15_12: mvl7;
    signal le15_12: mvl7;
    signal gr11_8: mvl7;
    signal eq11_8: mvl7;
    signal le11_8: mvl7;
    signal gr7_4: mvl7;
    signal eq7_4: mvl7;
    signal le7_4: mvl7;
    signal gr3_0: mvl7;
    signal eq3_0: mvl7;
    signal le3_0: mvl7;

    signal gr15_8: mvl7;
    signal eq15_8: mvl7;
    signal le15_8: mvl7;
    signal gr7_0: mvl7;
    signal eq7_0: mvl7;
    signal le7_0: mvl7;

    signal gr15_0: mvl7;
    signal eq15_0: mvl7;
    signal le15_0: mvl7;

begin
    bc15:bitcompare port map (a(15),b(15),a15GRb15,a15EQb15,a15LEb15);
    bc14:bitcompare port map (a(14),b(14),a14GRb14,a14EQb14,a14LEb14);
    bc13:bitcompare port map (a(13),b(13),a13GRb13,a13EQb13,a13LEb13);
    bc12:bitcompare port map (a(12),b(12),a12GRb12,a12EQb12,a12LEb12);
    bc11:bitcompare port map (a(11),b(11),a11GRb11,a11EQb11,a11LEb11);
    bc10:bitcompare port map (a(10),b(10),a10GRb10,a10EQb10,a10LEb10);
    bc9:bitcompare port map (a(9),b(9),a9GRb9,a9EQb9,a9LEb9);
    bc8:bitcompare port map (a(8),b(8),a8GRb8,a8EQb8,a8LEb8);
    bc7:bitcompare port map (a(7),b(7),a7GRb7,a7EQb7,a7LEb7);
    bc6:bitcompare port map (a(6),b(6),a6GRb6,a6EQb6,a6LEb6);
    bc5:bitcompare port map (a(5),b(5),a5GRb5,a5EQb5,a5LEb5);
    bc4:bitcompare port map (a(4),b(4),a4GRb4,a4EQb4,a4LEb4);
    bc3:bitcompare port map (a(3),b(3),a3GRb3,a3EQb3,a3LEb3);
    bc2:bitcompare port map (a(2),b(2),a2GRb2,a2EQb2,a2LEb2);
    bc1:bitcompare port map (a(1),b(1),a1GRb1,a1EQb1,a1LEb1);
    bc0:bitcompare port map (a(0),b(0),a0GRb0,a0EQb0,a0LEb0);

    mc15_14:magncompare port map (a15GRb15,a15EQb15,a15LEb15,a14GRb14,a14EQb14,a14LEb14,
```

```
                                gr15_14,eq15_14,le15_14);
   mc13_12:magncompare port map (a13GRb13,a13EQb13,a13LEb13,a12GRb12,a12EQb12,a12LEb12,
                                gr13_12,eq13_12,le13_12);
   mc11_10:magncompare port map (a11GRb11,a11EQb11,a11LEb11,a10GRb10,a10EQb10,a10LEb10,
                                gr11_10,eq11_10,le11_10);
   mc9_8:magncompare port map (a9GRb9,a9EQb9,a9LEb9,a8GRb8,a8EQb8,a8LEb8,gr9_8,eq9_8,
                               le9_8);
   mc7_6:magncompare port map (a7GRb7,a7EQb7,a7LEb7,a6GRb6,a6EQb6,a6LEb6,gr7_6,eq7_6,
                               le7_6);
   mc5_4:magncompare port map (a5GRb5,a5EQb5,a5LEb5,a4GRb4,a4EQb4,a4LEb4,gr5_4,eq5_4,
                               le5_4);
   mc3_2:magncompare port map (a3GRb3,a3EQb3,a3LEb3,a2GRb2,a2EQb2,a2LEb2,gr3_2,eq3_2,
                               le3_2);
   mc1_0:magncompare port map (a1GRb1,a1EQb1,a1LEb1,a0GRb0,a0EQb0,a0LEb0,gr1_0,eq1_0,
                               le1_0);

   mc15_12:magncompare port map (gr15_14,eq15_14,le15_14,gr13_12,eq13_12,le13_12,
                                gr15_12,eq15_12,le15_12);
   mc11_8:magncompare port map (gr11_10,eq11_10,le11_10,gr9_8,eq9_8,le9_8,gr11_8,
                                eq11_8,le11_8);
   mc7_4:magncompare port map (gr7_6,eq7_6,le7_6,gr5_4,eq5_4,le5_4,gr7_4,eq7_4,le7_4);
   mc3_0:magncompare port map (gr3_2,eq3_2,le3_2,gr1_0,eq1_0,le1_0,gr3_0,eq3_0,le3_0);

   mc15_8:magncompare port map (gr15_12,eq15_12,le15_12,gr11_8,eq11_8,le11_8,gr15_8,
                                eq15_8,le15_8);
   mc7_0:magncompare port map (gr7_4,eq7_4,le7_4,gr3_0,eq3_0,le3_0,gr7_0,eq7_0,le7_0);

   mc15_0:magncompare port map (gr15_8,eq15_8,le15_8,gr7_0,eq7_0,le7_0,gr15_0,eq15_0,
                                le15_0);

   res <= gr15_0;
end;
```