Register Binding based RTL Power Management for Control-Flow Intensive Designs

Jiong Luo, Lin Zhong, Yunsi Fei and Niraj K. Jha
Dept. of Electrical Engineering
Princeton University, Princeton, NJ 08544

ABSTRACT

One important way to reduce power consumption is to reduce the spurious switching activity in a circuit or circuit component, i.e., activity that is not required by its specified functionality. Given a scheduled behavior and functional unit binding, we show that spurious switching activity can be reduced through proper register binding using retentive multiplexers. Retentive multiplexers can preserve their previous select signal values in the control steps in which the select signals are don’t-cares. A functional unit, in which spurious switching activity is completely eliminated, is called perfectly power managed (PPM). We present a general sufficient condition for register binding to ensure a set of functional units to be PPM. This condition not only applies to data-flow intensive (DFI) behaviors but also to control-flow intensive (CFI) behaviors. It leads to a straightforward power-managed (PM) register binding algorithm, which uses this condition to preserve the previous values in the input registers of a functional unit during the states in which the unit is idle. The proposed algorithm is general and independent of the functional unit binding and scheduling algorithms. Hence, it can be easily incorporated into existing high-level synthesis systems. For the benchmarks we experimented with, an average 42.8% power reduction was achieved by our method at the cost of 7.6% average area overhead, compared to power-optimized register-transfer level (RTL) circuits which did not use PM register binding.

Acknowledgments: This work was supported in part by Alternative System Concepts under an SBIR contract from Army CECOM and in part by DARPA under contract no. DAAB07-00-C-L516.
1. Introduction

A significant amount of work has been done on high-level synthesis for low power, which takes as its input a behavioral description in the form of a data-flow graph (DFG) or control-data flow graph (CDFG) and outputs a power-optimized RTL circuit [1-12]. Most of these techniques target DFI behaviors, only some target CFI behaviors [6], [11]. Power management has been recognized as a very useful technique for reducing power consumption [1], [11-18]. It is well known that register binding not only affects the power consumption in the registers themselves, but may have a significant impact on the power consumption of functional units they feed as well [15], [16]. This is because improperly binding variables in the behavior to registers can result in significant spurious switching activity in the functional units.

One way to eliminate spurious switching activity in functional units is to use transparent latches at the input ports [15]. In the control steps in which the functional unit is idle according to the behavioral specification and its schedule, the latches can prevent propagation of any new value from the registers to functional units. However, the power consumed by the latches themselves reduces the power savings. Also, these latches result in delay overheads that need to be taken into account.

Since register values remain unchanged when their load signal is not enabled, another approach for reducing spurious switching activity is to reconfigure the multiplexer networks and bind variables to registers in such a way that when a functional unit is going to be idle in the next control step, it takes its inputs from the registers it most recently used, and these registers do not load in a new value in this step. A method was proposed in [16] to redesign the control logic to configure existing multiplexer networks to minimize spurious switching activity in the data path for CFI designs. However, it does not address register binding, and is hence unable to completely eliminate spurious switching activity. On the other hand, a register binding method which guarantees a PPM RTL circuit if the multiplexer network is properly designed was presented in [17]. It presents a sufficient condition for obtaining PPM RTL circuits for DFI behaviors. This method was extended in [18] to CFI behaviors based on the concept of variable and path protection. The lifetimes of protected variables are extended selectively along protected paths in the schedule in order to ensure that operand values of a functional unit used by the previous operation mapped to it are preserved at its inputs during subsequent idle cycles. Therefore, the spurious switching activity can be eliminated for functional units along the protected path. Register binding based power management for CFI behaviors differs from that for DFI behaviors in the following three aspects. First, the overlapping of
extended lifetimes of variables needs to be determined by considering the actual runtime execution paths. Second, different protected paths may require the input select signals of multiplexers to be set to different values. This leads to path conflicts in configuring multiplexer networks. Third, a protected variable may be re-assigned a new value along a protected path, which leads to spurious switching activities that cannot be eliminated by register binding alone. The work in [18] proposes to use variable re-naming and re-assignment to address the third aspect. However, for the first two aspects, it is not adequate or efficient. For the first aspect, it ignores path information when performing lifetime analysis of variables statically, and hence two variables may be regarded as interfering with each other for register binding even when their extended lifetimes do not overlap during actual circuit execution. For the second aspect, to resolve path conflicts, it proposes to use extra state memory to remember the previous state, and this state memory is used to configure the multiplexer networks. The number of states to be remembered is determined by the number of conflicting paths. Although extra state memory makes it possible to protect multiple conflicting paths simultaneously, it creates overheads in the controller.

This paper employs register binding based power management as well, with a focus on addressing the limitations of previous work when targeting CFI behaviors. We make the following contributions.

- We present a sufficient condition to ensure that any given set of functional units in an RTL implementation of both DFI and CFI behaviors are PPM. In contrast to the work in [18], this condition is developed based on the analysis of lifetimes of variables in a dynamic manner, by considering actual runtime execution paths. Based on the sufficient condition, we also propose a register binding algorithm for PM circuits.

- We introduce the concept of dynamic retentive multiplexers. Such multiplexers, with minimal extra logic, have the ability to preserve the previous select signal values when the select signals are don’t cares in the controller specification. Therefore, there is no need to remember the state history and add extra logic in the controller to resolve path conflicts while configuring multiplexer networks, as in the approach taken in [18]. As a result, the controller implementation is considerably simplified without sacrificing the ability to protect multiple paths simultaneously.

- To reduce the overheads of registers allocated, a technique called selective power management is proposed. This judiciously applies the sufficient condition to power-hungry functional units only.
The paper is organized as follows. In Section 2, we give an example to motivate our method. In Section 3, we present a sufficient condition for PPM register binding which leads to a register binding algorithm for power management, and discuss selective PM circuits. In Section 4, we present an experimental platform and a technique for incorporating power management checks into existing high-level synthesis tools. We present experimental results in Section 5 and conclude in Section 6.

2. Motivational Example

In the section, we present an example to motivate our approach. Figure 1 shows an example behavior. Figure 2 shows a state transition graph (STG) representing its schedule. A, B, C, D, E and F are six states besides the start and end states. Inside a state, the operations scheduled in it are shown. The variables needed by each operation are shown close to the state in which that operation is scheduled. For example, (+1:a,b) means that operation +1 takes variable a and b as inputs. Similarly, (=1:g,x) means that variable g is assigned the value of variable x.

```
initialize(a, b, c, d, e, h);
x = a+b;
y = c+d;
do {
g = x;
if(x<y) {
x = g+b; f=b;}
else {
f = g+h;
} 
while(e<h)
return f;
```

Figure 1: An example CFI behavior

Suppose the given resource constraints are two adders and a comparator. <1 and <2 are bound to the comparator. +1 and +5 are bound to one of the adders, say adder1, while +2, +3 and +4 are bound to the other one, say adder2. Now let us consider how the variables can share registers.

Suppose we bind variables c and x to the same register, and let d have its own register, as shown in Figure 3. In state A, the two input values for adder2 correspond to c and d, respectively. Since variable x is defined in state A, the register which c is bound to feeds a new value corresponding to variable x in state B in which adder2 is supposed to be idle. Assume in state B, the select signals for multiplexers MUX2 and MUX3 are all 0. Then the
input values for adder2 correspond to \( x \) and \( d \), respectively. This causes spurious switching activity in adder2 in the transition from state \( A \) to state \( B \).

One way to eliminate the possibility of the above spurious switching activity is not to bind variables \( c \) and \( x \) to the same register. Instead, we can bind variables \( c \) and \( f \) to the same register. We can set the select signals of both MUX2 and MUX3 to 0 in state \( B \), such that registers 1 and 3 are still selected to feed input ports 1 and 2 of adder2, respectively, as in state \( A \). Moreover, since variable \( f \) is not defined in state \( A \), register 1 still holds the same value in state \( B \) as in state \( A \). Similarly, register 3 still holds the same value since no other variables are bound to it, except variable \( d \). Therefore, no spurious switching activity occurs in adder2 in state \( B \).
It is worth pointing out that in real designs, spurious switching activity could be much more serious than that in this example if it occurs inside a loop.

3. PM Register Binding

In this section, we first present some rules to guide register binding in order to reduce the spurious switching activity in a set of functional units in the data path. We then discuss the concept and implementation of retentive multiplexers which are used to realize PM circuits. We then address the concept of selective PM circuits.

3.1 Register Binding for PM circuit

The PM property of a functional unit is jointly ensured through controller specification and register binding. In this section, we dwell on register binding only.

As mentioned before, the schedule of a behavior can be represented in the form of an STG. We next provide some definitions required for stating our sufficient condition for a set of functional units to be PPM.

For each variable $v$, $\text{defstates}(v)$ is defined as the set of states in which $v$ is defined, and $\text{usestates}(v)$ is defined as the set of states in which $v$ is used. A variable is live in state $p$ if there is a directed path in the STG from state $p$ to another state that uses the variable, without going through any state in $\text{defstates}(v)$ except state $p$ itself. $\text{livelives}(v)$ is defined as the set of states in which $v$ is live. $\text{livelives}(v)$ can be generated using liveness analysis algorithms from compiler theory [20].

In the example in Section 2, we have

$$
\text{livelives}(a) = \{\text{start}\};
\text{livelives}(b) = \{\text{start},A,B,C,D,E,F\};
\text{livelives}(c) = \text{livelives}(d) = \{\text{start}\};
\text{livelives}(e) = \{\text{start},A,B,C,D,E,F\};
\text{livelives}(f) = \{C,D,E,F\};
\text{livelives}(g) = \{B\};
\text{livelives}(h) = \{\text{start},A,B,E,F\};
\text{livelives}(x) = \text{livelives}(y) = \{A,B,C,D,E,F\}.
$$

For variable $c$, $\text{defstates}(c) = \{\text{start}\}$, $\text{usestates}(c) = \{A\}$; for variable $x$, $\text{defstates}(x) = \{A,C\}$, $\text{usestates}(x) = \{B,E\}$; and for variable $f$, $\text{defstates}(f) = \{C,D\}$, $\text{usestates}(f) = \{\text{end}\}$.

Two variables $a$ and $b$ can share a register if they do not have overlapping lifetimes during circuit execution. This can be determined based on the $\text{defstates}$ and $\text{livelives}$ of variables $a$ and $b$ as follows.
Theorem 1: Two variables \(a\) and \(b\) do not have overlapping lifetimes during circuit execution if \(\text{livestates}(a) \cap \text{defstates}(b) = \emptyset\), and \(\text{livestates}(b) \cap \text{defstates}(a) = \emptyset\).

In the example in Section 2, if we do not consider spurious switching activity, we can bind variables \(c\) and \(x\) to the same register since they satisfy the condition in Theorem 1. However, this condition is not sufficient to avoid spurious switching activity in the circuit. We need to assert some extra conditions to guide register binding, so that during idle states of a functional unit, the input values can be kept the same as those in the previous state. Therefore, no extra switching activity will occur. In order to do this, we introduce the notion of extended set of live states. For a variable \(v\), its extended set of live states with respect to functional unit \(F\) that it feeds is defined and computed recursively by Algorithm 1 shown in Figure 4. In this algorithm, \(\text{son}(p)\) is defined as the set of states which are the direct successors of state \(p\). For each functional unit \(F\), \(\text{idlestates}(F)\) is defined as the set of states in which \(F\) is idle, \(\text{activestates}(F)\) as the set of states in which \(F\) is used, and \(\text{last_idlestates}(F)\) as the subset of states in \(\text{idlestates}(F)\), whose direct successors all belong to \(\text{activestates}(F)\). In the example in Section 2, we have \(\text{idlestates}(adder1) = \{\text{start}, \text{B}, \text{C}, \text{D}, \text{F}, \text{end}\}\), \(\text{idlestates}(adder2) = \{\text{start}, \text{B}, \text{E}, \text{F}, \text{end}\}\), \(\text{last_idlestates}(adder1) = \{\text{start}, \text{C}, \text{D}\}\), and \(\text{last_idlestates}(adder2) = \{\text{start}, \text{B}\}\).

Algorithm 1: Extended_set_of_livestates\((F, \text{livestates}(v))\) {
initialize extlivestates\((v,F)\);
while(I) {
    temp = extlivestates\((v,F)\);
    for each p \in extlivestates\((v,F)\) {
        temp1 = \((\text{idlestates}(F) - \text{last_idlestates}(F)) \cap \text{son}(p)\) \cap temp;
        temp = temp \cup temp1;
    }
    if(extlivestates\((v,F)\) == temp)
        break;
    extlivestates\((v,F)\) = temp;
}
}

Figure 4: Computing extended set of live states of a variable

In Algorithm 1, \(\text{extlivestates}(v,F)\) is initialized with the states in which variable \(v\) is used to feed functional unit \(F\), and whose direct successor states do not all belong to \(\text{activestates}(F)\). Any successors of states in \(\text{extlivestates}(v,F)\) are included recursively into \(\text{extlivestates}(v,F)\), until any state in \(\text{last_idlestates}(F)\) is reached. For the example in Section 2, the extended sets of live states for variables \(c\) and \(d\) with respect to \(adder2\) are \(\text{extlivestates}(c, adder2) = \{A\}\), since in state \(A\), variables \(c\) and \(d\) are used to feed
adder2, and state $B$, the successor of state $A$, belongs to $\text{last_idlestates}(adder2)$. Similarly, $\text{extlivestates}(x, adder1) = \{B,E,F,end\}$.

Theorem 2 below gives a sufficient condition for a set of functional units, $\text{ppm_func}$, to be PPM. First, we present conditions that two variables do not interfere with each other during register binding.

For any two variables $a$ and $b$, assume the set of functional units which $a$ feeds is $\text{func}_a$, and the set of functional units which $b$ feeds is $\text{func}_b$. Let $\text{ppm_func}_a = \text{func}_a \cap \text{ppm_func}$, and $\text{ppm_func}_b = \text{func}_b \cap \text{ppm_func}$. We state that variables $a$ and $b$ do not interfere with each other if the following two conditions are satisfied: (a) $\text{livestates}(a) \cap \text{defstates}(b) = \Phi$ and $\text{livestates}(b) \cap \text{defstates}(a) = \Phi$, and (b) for any $F_a \in \text{ppm_func}_a$, $\text{extlivestates}(a,F_a) \cap \text{defstates}(b) = \Phi$, and for any $F_b \in \text{ppm_func}_b$, $\text{extlivestates}(b,F_b) \cap \text{defstates}(a) = \Phi$. We call these two conditions non-interference conditions. If two variables satisfy the non-interference conditions, we say that they do not interfere with each other.

In the example in Section 2, assume $\text{ppm_func} = \{adder1, addder2\}$. Let us look at variables $c$, $x$, and $f$. Variable $c$ feeds $adder2$, and variable $x$ feeds $adder1$. Hence, $\text{ppm_func}_c = \{adder2\}$, $\text{ppm_func}_x = \{adder1\}$ and $\text{ppm_func}_f = \Phi$.

**Theorem 2**: During register binding, if the following two conditions are satisfied: (1) any two variables are allowed to share the same register only if they do not interfere with each other, and (2) for any variable $a$ and any functional unit $F_a \in \text{ppm_func}_a$, $\text{extlivestates}(a,F_a) \cap \text{defstates}(a) = \Phi$, then no spurious switching activity will occur in the functional units in $\text{ppm_func}$, provided that the multiplexers feeding their input ports are all retentive.

**Proof**: Non-interference condition (a) guarantees that variables $a$ and $b$ do not have overlapping lifetimes during the running of a circuit if they are bound to the same register. Non-interference condition (b) and condition (2) in Theorem 2 guarantee that no extra switching activity occurs in $\text{ppm_func}$. For any functional unit $F \in \text{ppm_func}$, if the multiplexers feeding its ports are retentive, then during circuit execution, in any idle state $p$ of $F$, the same register will be selected at its input port as in the last state $q$. Furthermore, assume in state $q$, the register holds the value of some variable $a$. Then $\text{extlivestates}(a,F)$ should contain state $q$, according to the definitions of extended set of live states. Since any variable which gets defined in state $q$ interferes with variable $a$, such an interfering variable will not be bound to this register. Furthermore, if condition (2) in Theorem 2 is satisfied, no new value of variable $a$ gets defined in state $q$. Hence, no extra switching activity can occur in state $p$. $\square$
In Theorem 2, condition (1) (non-interference conditions) can be satisfied through proper register binding, while condition (2) is determined by the schedule and cannot be controlled by register binding. However, condition (2) can be satisfied through variable re-naming and register re-assignment. If condition (2) is not satisfied, we cannot guarantee the PPM property of a set of functional units. However, we can perform register binding based on condition (1) only, which still significantly reduces the spurious switching activity in the circuit, as shown in Section 5. We call this PM register binding.

According to Theorem 2, variables $c$ and $x$ in our running example cannot be bound to the same register. Instead, variables $c$ and $f$ can be bound to the same register even after the live states of variable $c$ are extended to include $A$.

It is worth noting that in Theorems 1 and 2, the non-interference conditions are based on analyzing the overlap of defstates and livestates (extlivestates), and not on statically analyzing the overlap of lifetimes and extended lifetimes of variables. In this way, two variables are considered interfering with each other if and only if their lifetimes and extended lifetimes overlap during actual execution.

3.2 Retentive Multiplexers

The essence of a PM functional unit is to make sure that it does not have any spurious switching activity, i.e., the input values to it remain the same as they were in the previous state in which it was active. For the input values to remain the same, first, the select signals of the multiplexers feeding its input ports should retain the same values, and second, the register value which feeds the selected input of the multiplexer should remain the same. The latter can be ensured by using the proposed sufficient condition (Theorem 2). However, the former needs help from the controller since retentive multiplexers are assumed. We next provide two different implementations of retentive multiplexers.

The first implementation is based on the controller re-specification method given in [16]. The don’t-cares in the outputs for multiplexer select signals in the state transition table of the controller are identified and assigned proper values. If a multiplexer select signal is a don’t-care in state $A$, the corresponding functional unit fed by this multiplexer is idle in state $A$. If the don’t-care is replaced with the value assigned to the select signal in the previous state, the multiplexer will output the value from the same input port that it did in the previous state. A problem arises when a state has multiple predecessor states. In this case with this method, it becomes impossible to
guarantee elimination of all spurious switching activity in the functional unit. Then, the preceding state with the highest transition probability into the current state is identified and the don’t-care is replaced with the value in that state. The spurious switching activity can be statistically reduced in this case. Such a multiplexer is called static retentive since the don’t-cares are assigned values statically.

In the second implementation, we can introduce extra hardware to make the multiplexer completely retentive. An extra control signal can be added to indicate whether the functional unit under consideration is idle or not. It is assigned a 0 in the states in which the functional unit is idle. Select signals to a functional unit’s multiplexers are don’t-care in a state if and only if the functional unit is idle in that state. A one-bit delay latch [19] can be added to the select signal input of the multiplexer. When the functional unit is idle, the latch is disabled and it holds the previous value. In this way, the multiplexer remembers the select signal value of the last state the functional unit was in. The extra bit acts as the enable signal for the latch. We call such multiplexers dynamic retentive. These are the multiplexers assumed in Theorem 2. The disadvantage of such a design is the overhead introduced in the controller (one extra bit per functional unit) and multiplexers (one-bit latch per 2-to-1 multi-bit functional unit multiplexer). However, compared with placing transparent latches before a functional unit input port [15], the extra hardware for this approach is very small. This is because a transparent latch has the same bit-width as the input port it feeds, whereas we require only a one-bit latch per 2-to-1 multiplexer. The overheads in the controller for the two approaches are similar.

In practice, we have found that using static retentive multiplexers eliminates most of the spurious switching activity in a functional unit. The fact that they do not require much overhead makes them doubly attractive.

3.3 Selective PM Circuits

PM register binding tries to reduce spurious switching activity in functional units with the help of some extra registers needed for this purpose. This causes some area and power overhead in registers. The power/area of some register implementations can be comparable to those of some functional units. For example, we found that an 8-bit register on an average consumes more power than an 8-bit ripple-carry adder if implemented in NEC CB-11 0.18µm cell-based technology [21]. Therefore, in order to make an adder PM, if we have to introduce an extra register, the trade-off is not worth it. The ideal way for balancing register power consumption overhead and spurious switching activity in functional units is to compare the two for each register binding. If the former is smaller than the latter, the register binding is accepted although it causes some spurious switching activity. This
method requires spurious switching activity estimation for every affected functional unit for every register binding choice.

A simpler way is to only make power-hungry functional units PM instead of all functional units. That is, only binding for registers that feed power-hungry functional units is checked using Theorem 2. For registers feeding other functional units, binding is done without regard to whether it will cause spurious switching activity or not. In the RTL library we used, the following functional units were more power-hungry than registers: different types of multipliers, dividers and general-purpose ALUs. They were the functional units that were targeted in our experiments.

4. Experimental Platform

To evaluate the proposed method, we incorporated it into an existing low power high-level synthesis tool that is an extended version of the one described in [10] and can handle both DFI and CFI behaviors. The main modification we needed to make to the synthesis flow was that instead of interleaving functional unit binding with register binding, we first performed all functional binding and then register binding. This ensures that for different register binding methods, the same functional unit optimization is done.

Figure 5 shows a high-level view of the tool. After reading the specification in the form of a CDFG and resource or timing constraints, it starts with either time-constrained scheduling (for DFI behaviors) or resource-constrained scheduling (for CFI behaviors). The resulting STG is simulated to collect data to evaluate the power and timing of the RTL architectures generated later. Data path optimization is based on variable-depth iterative improvement which is capable of escaping local minima. It starts with a fully parallel architecture in which each operation is bound to its own fastest possible functional unit from the RTL design library and each variable is bound to its own register. This architecture is iteratively improved for power optimization purposes through various moves such as functional unit synthesis, resource sharing/splitting, multiplexer tree synthesis, etc., first for functional units and then for registers. If a PM circuit or a selective PM circuit is desired, condition (1) in Theorem 2 is checked before two variables are allowed to share a register. The power-optimized RTL data path is output at the end. This is the best architecture seen during iterative improvement. Then controller re-specification is done as described in Section 3.2. That is, static retentive multiplexers are used for different register bindings, and don’t-cares in the controller state transition table are appropriately specified.
Our register binding technique can be incorporated in other high-level synthesis tools as well. Once scheduling and functional unit binding information is available, register binding can simply be based on the sufficient condition given in Theorem 2.

5. Experimental Results

In this section, we present experimental results for our proposed techniques. Three register binding methods are compared. The first (maximal) allows as much sharing of registers as possible without regard to spurious switching activity. The second is register binding to obtain a PM circuit using Theorem 2. The third is register binding to obtain a selective PM circuit geared towards only power-hungry functional units in the data path. The controller is re-specified as described in Section 3.2 in all the cases. We show results for both static and dynamic retentive multiplexers.

We used various high-level synthesis benchmarks to establish the efficacy of our technique. chemical is an infinite impulse response (IIR) filter used in the industry. dct_dif, dct_lee and dct_wang are different algorithms for
computing Discrete Cosine Transform [22]. \textit{diffeq} is a differential equation solver from the NCSU CBL high-level synthesis benchmark repository [23]. \textit{paulin} is a variant of \textit{diffeq} from [24]. \textit{wavelet} performs the Discrete Wavelet Transform. These are all DFI behaviors. They have been widely used by the research community. \textit{con_loop} is a large synthetic CFI example to test concurrent loop optimization.

The power consumption and area of the RTL circuits are computed using an NEC RTL library for 0.35\textmu m technology.

Table 1 shows total power and area of the controller/data path implementations of the different behaviors for the maximal register binding case. In Table 2, we show the percentage reduction in total power as well as the overhead in area for PM circuits (\textit{i.e.}, all functional units are PM) compared to the maximal register binding case. On an average, the PM circuits reduce total power by 42.8\% at an area overhead of 7.6\%, compared to circuits which are power-optimized except that they ignore spurious switching activity.

The total power for the four cases, maximal, PM with static retentive multiplexers (PM), selective PM, and PM with dynamic retentive multiplexers, is plotted in Figure 6. The selective PM method is seen to be as good as the PM method in power reduction. Dynamic retentive multiplexers do help in two cases, but not in others.

Figure 7 shows the percentage of the total functional unit switching activity that is spurious for three different register bindings. One can see that PM and selective PM register binding reduce the spurious switching activity significantly. If dynamic retentive multiplexers are employed, there will be no spurious switching activity at all.
Table 2: Maximal vs. PM register binding

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Maximal versus PM binding</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Power reduction (%)</td>
</tr>
<tr>
<td>chemical</td>
<td>20.1</td>
</tr>
<tr>
<td>con_loop</td>
<td>63.3</td>
</tr>
<tr>
<td>dct_dif</td>
<td>33.1</td>
</tr>
<tr>
<td>dct_lee</td>
<td>62.0</td>
</tr>
<tr>
<td>dct_wang</td>
<td>68.0</td>
</tr>
<tr>
<td>diffeq</td>
<td>45.8</td>
</tr>
<tr>
<td>paulin</td>
<td>20.8</td>
</tr>
<tr>
<td>wavelet</td>
<td>29.3</td>
</tr>
<tr>
<td>Average</td>
<td>42.8</td>
</tr>
</tbody>
</table>

Figure 6: Total power consumption for different techniques

Figure 7: Spurious switching activity percentage for different register bindings
6. Conclusions

We have demonstrated that by properly binding variables to registers in high-level synthesis, spurious switching activity in functional units can be significantly reduced. We gave a general sufficient condition for a set of functional units to be free of spurious switching activity in implementations of both DFI and CFI behaviors. Based on this condition, we proposed a register binding algorithm to reduce spurious switching activity in a given set of functional units. We achieved an average 42.8% power reduction at an average 7.6% area overhead compared to already power-optimized architectures. This condition can be applied selectively to power-hungry functional units in the data path.

We also discussed how the controller and/or multiplexers can be redesigned to cooperate with register binding to reduce spurious switching activity in the data path. Dynamic retentive multiplexers can eliminate most spurious switching activity but introduce some extra hardware which is very small compared to that of placing transparent latches before functional units’ input ports. Static retentive multiplexers hardly require any extra hardware, but permit some spurious switching activity. In practice, we found that PM register binding with static retentive multiplexers can eliminate most of the spurious switching activity in the benchmarks.
REFERENCES


