# Register Binding based Power Management for High-level Synthesis of Control-Flow Intensive Behaviors

Lin Zhong, Jiong Luo, Yunsi Fei and Niraj K. Jha

Department of Electrical Engineering
Princeton University, Princeton, NJ 08544

## ABSTRACT

*A circuit or circuit component that does not contain any spurious switching activity, i.e., activity that is not required by its specified functionality, is called perfectly power managed (PPM). We present a general sufficient condition for register binding to ensure that a given set of functional units is 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. The proposed algorithm is 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 45.9% power reduction was achieved by our method at the cost of 7.7% average area overhead, compared to power-optimized register-transfer level (RTL) circuits which did not use PM register binding.*

## 1. Introduction

High-level synthesis for low power 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 [1-5]. Power management has been recognized as a very useful technique for reducing power consumption [1], [6-11]. It is well known that register binding may introduce spurious switching activity in functional units they feed [8], [9]. One way to eliminate such spurious switching activity is to add transparent latches at the functional unit input ports [8]. 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.

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. In [9], a technique is proposed to redesign the control logic to configure existing multiplexer networks to minimize (may not eliminate) spurious switching activity in the data path. On the other hand, a register binding method which guarantees a PPM RTL circuit for DFI behaviors, if the multiplexer network is properly designed, was presented in [10]. This method was extended in [11] to CFI behaviors. However, it could not ensure that a given set of functional units is PPM, *i.e.*, did not provide a sufficient condition for CFI behaviors.

We present a sufficient condition to ensure that any given set of functional units in an RTL implementation of both DFI and CFI behaviors is PPM. No such condition has been presented before for CFI behaviors. Based on the sufficient condition, we propose a register binding algorithm for PM circuits.

Our method does not impact the choice of the scheduling or functional unit binding algorithms and is easy to incorporate into existing high-level synthesis systems. Our experimental results show an average power reduction of 45.9% at an average area overhead of 7.7% compared to power-optimized RTL circuits.

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 one way 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

Figure 2.1 shows an example behavior and 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 scheduled state. For example, (+1:*a,b*) means that operation +1 takes variable *a* and *b* as inputs.
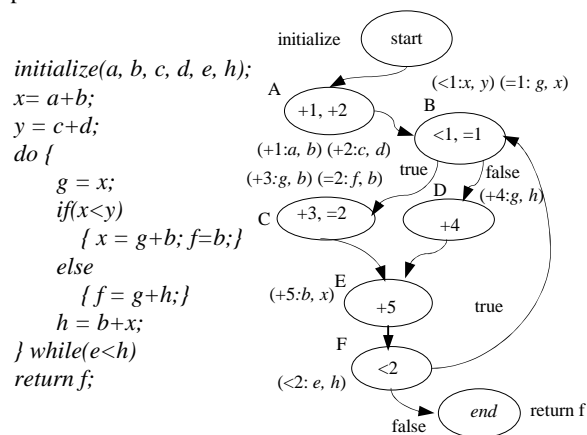


Figure 2.1: An example CFI behavior and its STG

Suppose we have two adders and one comparator. <1 and <2` are bound to the comparator, and +1 and +5 to one of the adders, say *adder1*, while +2, +3 and +4 to the other one, say *adder2*. Suppose we bind variables *c* and *x* to the same register, and let *d* have its own, as shown in Figure 2.2. 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 the value of 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 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 $Reg1$ and $Reg3$ are still selected to feed input ports $in1$ and $in2$ of $adder2$, respectively, as in state $A$. Moreover, since variable $f$ is not defined in state $A$, register $Reg1$ still holds the same value in state $B$ as in state $A$. Similarly, register $Reg3$ still holds the same value since no other variable is bound to it. 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.
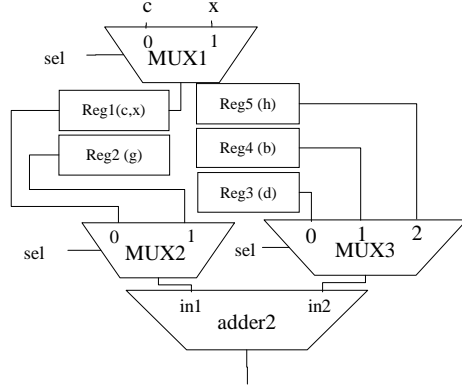


Figure 2.2: Part of the RTL implementation

# 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 given 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. Retentive multiplexers are multiplexers which can preserve their last select signal values in the control steps in which the select signals are don't-care. We then address the concept of selective PM circuits.

## 3.1 Register Binding for PM circuit

As mentioned before, the schedule of a behavior can be represented in the form of an STG. For each variable $v$, $defstates(v)$ is defined as the set of states in which $v$ is defined, and $usestates(v)$ 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 $defstates(v)$ except state $p$ itself. $livestates(v)$ is defined as the set of states in which $v$ is live. $livestates(v)$ can be generated using liveness analysis algorithms from compiler theory [13].

In the example in Section 2, we have
$livestates(a) = \{start\}$; $livestates(b) = \{start,A,B,C,D,E,F\}$;
$livestates(c) = livestates(d) = \{start\}$;
$livestates(e) = \{start,A,B,C,D,E,F\}$;
$livestates(f) = \{C,D,E,F\}$; $livestates(g) = \{B\}$;
$livestates(h) = \{start,A,B,E,F\}$;
$livestates(x) = livestates(y) = \{A,B,C,D,E,F\}$.
For variable $c$, $defstates(c) = \{start\}$, $usestates(c) = \{A\}$; for variable $x$, $defstates(x) = \{A,C\}$, $usestates(x) = \{B,E\}$; and for variable $f$, $defstates(f) = \{C,D\}$, $usestates(f) = \{end\}$.

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 *defstate*s and *livestates* of variables $a$ and $b$ as follows [13].

**Theorem 1**: Two variables $a$ and $b$ do not have overlapping lifetimes during circuit execution if $livestates(a) \cap defstates(b) = \Phi$, and $livestates(b) \cap defstates(a) = \Phi$.

We next introduce the notion of *extended set of live states*. For a variable $v$, its *extended set of live states* with respect to the functional unit $F$ that it feeds is defined and computed recursively by Algorithm 1 in Figure 3.1. In this algorithm, $son(p)$ is defined as the set of states which are the direct successors of state $p$. For each functional unit $F$, $idlestates(F)$ is defined as the set of states in which $F$ is idle, $activestates(F)$ as the set of states in which $F$ is used, and $last\_idlestates(F)$ as the subset of states in $idlestates(F)$ which are the direct predecessors of any state $s \in activestates(F)$. For our running example, we have $idlestates(adder1) = \{start,B,C,D,F,end\}$, $idlestates(adder2) = \{start,B,E,F,end\}$, $last\_idlestates(adder1) = \{start,C,D\}$, and $last\_idlestates(adder2) = \{start,B\}$.

> *Algorithm* 1: *Extended_set_of_livestates(F, livestates(v)) {*
>    *initialize extlivestates(v,F);*
>    *while(1) {*
>        *temp = extlivestates(v,F);*
>        *for each p∈ extlivestates(v,F) {*
>            *temp1=(idlestates(F)−last_idlestates(F))∩son(p);*
>            *temp = temp U temp1;*
>        *}*
>        *if(extlivestates(v,F)==temp)*
>            *break;*
>        *extlivestates(v,F) = temp;*
>    *}*
> *}*

Figure 3.1: Computing *extended set* of *live states* of a variable

In Algorithm 1, $extlivestates(v,F)$ 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 $activestates(F)$. All successors of states in $extlivestates(v,F)$ are included recursively into $extlivestates(v,F)$, until any state in $last\_idlestates(F)$ is reached. For the example in Section 2, the *extended set of live states* for variable $c$ with respect to $adder2$ is $extlivestates(c, adder2) = \{A\}$, since in state $A$, variable $c$ is used to feed $adder2$, and state $B$, the successor of state $A$, belongs to $last\_idlestates(adder2)$.

For any two variables $a$ and $b$, assume the set of functional units which $a$ feeds is $func_a$, and the set of functional units which $b$ feeds is $func_b$. Suppose we want to make a set of functional units, $ppm\text{-}func$, PPM. Let $ppm\_func_a = func_a \cap ppm\_func$, and $ppm\_func_b = func_b \cap ppm\_func$. We say that variables $a$ and $b$ do not interfere with each other if the following two conditions are satisfied: (a) $livestates(a) \cap defstates(b) = \Phi$ and $livestates(b) \cap defstates(a) = \Phi$, and (b) for any $F_a \in ppm\_func_a$, $extlivestates(a,F_a) \cap defstates(b) = \Phi$, and for any $F_b \in ppm\_func_b$, $extlivestates(b,F_b) \cap defstates(a) = \Phi$. We call these two conditions **non-interference conditions**. Theorem 2 below gives a sufficient condition for the set of functional units, $ppm\_func$, to be PPM. For our running example, assume $ppm\_func = \{adder1, adder2\}$. Then since variable $c$ feeds $adder2$, and variable $x$ feeds $adder1$, $ppm\_func_c = \{adder2\}$, $ppm\_func_x = \{adder1\}$. However $ppm\_func_f = \Phi$.

**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 $F_a \in ppm\_func_a$, $extlivestates(a,F_a) \cap defstates(a) = \Phi$, then no spurious switching activity will occur in the functional units in $ppm\_func$, 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 *ppm_func*. For any functional unit $F \in ppm\_func$, 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 *extlivestates(a,F)* should contain state *q*, according to the definition 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*. □

In Theorem 2, condition (1) can be satisfied through proper register binding, while condition (2) is determined by the schedule and can be satisfied through variable re-naming and register re-assignment. If condition (2) is not satisfied, we cannot guarantee the PPM property of the given set of functional units. However, satisfaction of condition (1) only will still significantly reduce the spurious switching activity in the circuit, as shown in Section 5. We call this PM register binding.

## 3.2 Retentive Multiplexers

The essence of a PM functional unit is to make sure that the input values to it remain the same as they were in the last state in which it was active. To achieve this, 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 [9]. The don't-cares of the multiplexer select signals in the state transition table of the controller are identified and assigned proper values. The spurious switching activity can be statistically reduced if not eliminated. Such a multiplexer is called *static retentive* since the don't-cares are assigned values statically.

In the second implementation, we introduce an extra control signal to indicate whether the concerned functional unit is idle or not. A one-bit delay latch [12] is then added to the select signal input of the multiplexers in question. When the functional unit is idle, the latch is disabled by the added control signal and holds the previous value. Such multiplexers are called *dynamic retentive* and are the ones assumed in Theorem 2. The disadvantage of this implementation is the overhead introduced in the controller (one extra bit per functional unit) and multiplexers (one-bit latch for some 2-to-1 multi-bit multiplexers). However, compared with placing transparent latches before a functional unit input port [8], the extra hardware for our approach is very small.

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. The power/area of some register implementations can be comparable to those of some functional units. 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. 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 which can handle both DFI and CFI behaviors [5]. To ensure that for different register binding methods, the same functional unit optimization is done, we performed all functional binding first and then register binding, instead of interleaving them.

After reading the specification in the form of a CDFG and resource or timing constraints, the tool starts with either time-constrained scheduling or resource-constrained scheduling. 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 selection, resource sharing/splitting, multiplexer tree selection, *etc.*, first for functional units and then for registers. If a PM 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, three register binding methods are compared. The first (maximal) allows sharing of registers as much 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 IIR filter used in the industry. *dct_dif, dct_lee* and *dct_wang* are different algorithms for computing Discrete Cosine Transform [14]. *diffeq* is a differential equation solver from the NCSU CBL high-level synthesis benchmark repository [15]. *wavelet* performs the Discrete Wavelet Transform. These are all DFI behaviors. We also constructed a behavior, called *con_loop*, of two concurrent loops which are both control and data intensive. The VHDL code for it can be downloaded from [16]. The power consumption and area of the RTL circuits are computed using an NEC 0.35μm RTL library.

In Table 5.1, we show the percentage reduction in total power as well as the overhead in area for PM circuits (*i.e.*, all

functional units are PM) compared to the maximal register binding case. On an average, the PM circuits reduce total power by 45.9% at an area overhead of 7.7%, compared to circuits which are power-optimized with spurious switching activity ignored.

The total power for the four cases, maximal, PM with static retentive multiplexers (PM), selective PM, and PM with dynamic retentive multiplexers, is shown in Figure 5.1. The selective PM method is seen to be as good as the PM method in power reduction but requires less area overhead. Dynamic retentive multiplexers do help in two cases, but not in others.

We found that the average spurious switching power as a fraction of total power for maximal binding was 52.7%, whereas for the PM and selective PM circuits, the average was reduced to only 11.1% and 11.7%, respectively. Use of dynamic retentive multiplexers reduces spurious switching power to zero. The CPU times for high-level synthesis ranged from 9 seconds to 145 seconds on a Pentium-III machine with 256 MB memory running under Linux.

Table 5.1: Maximal vs. PM register binding

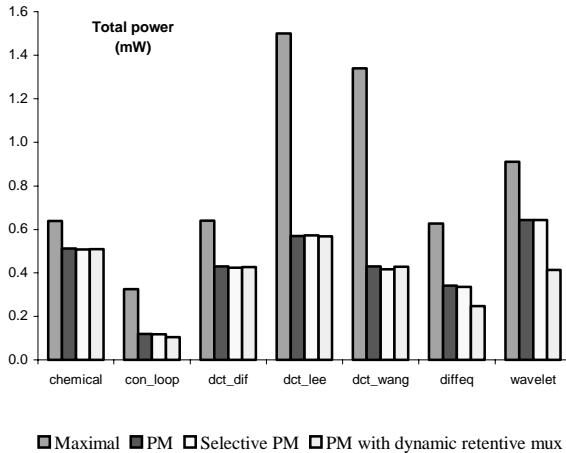| Benchmarks | Maximal versus PM binding | |
| --- | --- | --- |
| | Power reduction (%) | Area overhead (%) |
| *chemical* | 20.1 | 7.4 |
| *con_loop* | 63.3 | 5.9 |
| *dct_dif* | 33.1 | 10.5 |
| *dct_lee* | 62.0 | 8.6 |
| *dct_wang* | 68.0 | 9.2 |
| *diffeq* | 45.8 | 4.5 |
| *wavelet* | 29.3 | 7.5 |
| Average | 45.9 | 7.7 |



Figure 5.1: Total power consumption for different techniques

# 6. Conclusions

We 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 given 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 45.9% power reduction at an average 7.7% 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

[1] A. Raghunathan, N. K. Jha, and S. Dey, *High-level Power Analysis and Optimization*, Kluwer Academic Publishers, 1998.

[2] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey and R. Brodersen, "Optimizing power using transformations," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 12-51, Jan. 1995.

[3] R. S. Martin and J. P. Knight, "Power profiler: Optimizing ASICs power consumption at the behavioral level," in *Proc. Design Automation Conf.*, pp. 42-47, June 1995.

[4] A. Raghunathan and N. K. Jha, "SCALP: An iterative improvement based low-power data path synthesis algorithm," *IEEE Trans. Computer-Aided Design*, vol. 16, no. 11, pp. 1260-1277, Nov. 1997.

[5] K. S. Khouri, G. Lakshminarayana, and N. K. Jha, "High-level synthesis of low power control-flow intensive circuits," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 12, pp. 1715-1729, Dec. 1999.

[6] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, and M. Papaefthymiou, "Precomputation-based sequential logic optimization for low power," *IEEE Trans. VLSI Systems*, vol. 2, pp. 426-436, Dec. 1994.

[7] C. Papachristou, M. Spining, and M. Nourani, "An effective power management scheme for RTL design based on multiple clocks," in *Proc. Design Automation Conf.*, pp. 337-342, June 1996.

[8] E. Mussoll and J. Cortadella, "High-level synthesis techniques for reducing the activity of functional unit," in *Proc. Int. Symp. Low Power Design*, pp. 99-104, Apr. 1995.

[9] S. Dey, A. Raghunathan, N. K. Jha, and K. Wakabayashi, "Controller-based power management for control-flow intensive designs," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 10, pp. 1496-1508, Oct. 1999.

[10] G. Lakshminarayana, A. Raghunanthan, N. K. Jha, and S. Dey, "Power management in high level synthesis," *IEEE Trans. VLSI Systems*, vol. 7, no. 1, pp. 7-15, Mar. 1999.

[11] G. Lakshminarayana, A. Raghunanthan, N. K. Jha, and S. Dey, "Transforming control-flow intensive designs to facilitate power management," in *Proc. Int. Conf. Computer-Aided Design*, pp. 657-664, Nov. 1998.

[12] V. P. Nelson *et al.*, *Digital Logic Circuit Analysis and Design*, Prentice-Hall, 1995.

[13] A. W. Appel, *Modern Compiler Implementation in ML*, Cambridge University Press, 1998.

[14] K. R. Rao and P. Yip, *Discrete Cosine Transform*, Academic Press, 1990.

[15] http://www.cbl.ncsu.edu/benchmarks/

[16] http://www.ee.princeton.edu/~lzhong/publications/con-loop.vhd