

REWIREd – Register Write Inhibition by Resource Dedication

Pushkar Tripathi

Rohan Jain

Srikanth Kurra

Preeti Ranjan Panda

Dept. of CSE
Indian Inst. of Tech. Delhi
New Delhi 110016, India

Dept. of CSE
Indian Inst. of Tech. Delhi
New Delhi 110016, India

Oracle India Pvt. Ltd.
Bangalore 560029, India

Dept. of CSE
Indian Inst. of Tech. Delhi
New Delhi 110016, India

Abstract— We propose REWIREd (REgister Write Inhibition by REsource Dedication), a technique for reducing power during high level synthesis (HLS) by selectively inhibiting the storage of function unit (FU) output data into registers. Registers are generally inferred in HLS when data produced in one clock cycle is used in a later cycle. However, when it can be established that the input registers to an FU are not changing values during a certain period, the outputs during this period can be directly read off the FU output pins without needing to store them in registers. When the life-times of such data are short, it may be possible to completely eliminate the register storage operation, thereby reducing power. We present a genetic algorithm formulation and a heuristic for maximizing the number of register stores that can be inhibited in a scheduled data flow graph (DFG) during behavioral synthesis.

I. INTRODUCTION

High level synthesis consists primarily of scheduling, function unit allocation/binding, and register allocation steps. Scheduling determines the mapping of operations to clock cycles; FU binding determines the actual resources performing each operation; and register allocation assigns physical registers to store data produced at FU outputs. As the power reduction techniques at behavioral level have a significant impact on the synthesized results, researchers have targeted the traditional behavioral synthesis sub-tasks for power optimization in addition to the conventional metrics of area and performance.

In HLS, data generated by FUs in each cycle are stored in registers if they are used at a later time. Consequently, the action of writing (storing) into registers occurs very frequently in any design. This makes the register sub-system an attractive target for power optimization, and motivates us to take a closer look at register allocation decisions. In this paper we propose to depart from the conventional HLS strategy of storing FU output data of scheduled operations in registers every cycle, arguing that the data can still be read off the FU output pins if we have established that the FU input registers have not changed in the intervening cycles. When such opportunities exist, we can save power by avoiding the writing of the data into registers.

Power aware register allocation has received considerable attention from researchers. In [4], the authors presented a way to calculate the switching activity based on the assumption that the joint probability density function of primary input random variables is known or a sufficiently large number of input vectors is given. After computing the switching activity between pairs of data values that could potentially share the same register, the power-aware register assignment problem is formulated as a minimum cost clique covering of a compatibility

graph. The power optimization problem was formulated as a max-cost multi-commodity flow problem in [5, 6]. In [7], the authors presented an allocation algorithm for low power by allowing all the storage elements to be implemented in latches while minimizing the spurious operations in functional units. A formulation for generalising low power register allocation to multiple basic blocks is given in [3]. Further, an attempt can be made to reduce power by keeping the inputs of FU's stable to the extent possible by doing an appropriate register allocation [10, 11].

We propose REWIREd, a new behavioral synthesis optimisation that attempts to reduce power dissipation through reducing the frequency of register stores. Our technique is orthogonal to most of the above ideas, and can be applied in conjunction with them.

II. MOTIVATION

In traditional HLS, when a DFG edge crosses a clock cycle boundary after scheduling, a register is inferred for storing the value. This helps to free up functional units after performing the operation in the current cycle, so that those functional units can be reused to perform other computations in the later cycles.

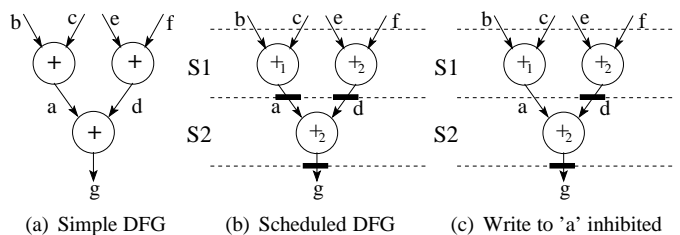


Fig. 1. Data Flow Graph

Consider an example DFG with three add operations as shown in Figure 1(a). Assume that the DFG has been scheduled with two adders and each add operation requires one cycle. One possible schedule with binding of the DFG is shown in Figure 1(b). The controller and datapath for this schedule is shown in Figure 2. PS and NS are the present state and next state columns; M1 and M2 are the MUX select signals; and R1 and R2 are register load signals.

An alternative implementation, corresponding to the schedule of Figure 1(c), is shown in Figure 3, with no register inferred for a , and the output of $+_1$ being sent directly to the MUX at $+_2$'s input. Assuming the values of b and c are not

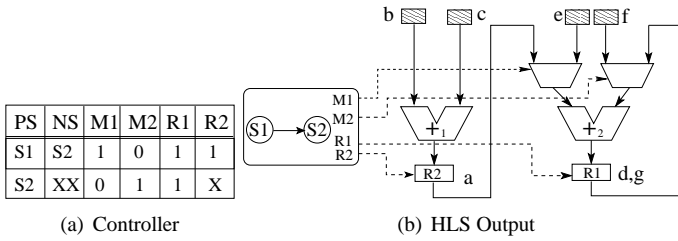


Fig. 2. Synthesized Design

changing in between state S1 and S2, the output at $+_1$ is still available in state S2, because its inputs are held constant. Effectively, the resource $+_1$ is *dedicated* for the $b + c$ operation in state S2 also, and is not assigned to any other operation. We term this optimization REWIRED – *register write inhibition by resource dedication*, the latter referring to the strategy of keeping the inputs to an FU constant explicitly to read its output value in a future cycle.

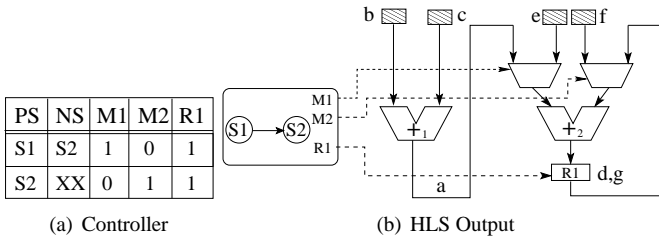


Fig. 3. Register Write Inhibition

The synthesized design in Figure 3 is more energy efficient than the one in Figure 2 due to the reduction in the datapath and controller energy. The datapath energy has been reduced by avoiding the register load operation. The controller energy has been reduced as we have eliminated the generation of the associated register load signal for R2. Significant opportunities for register inhibition arise when there are complex expressions being evaluated. Resource utilization is frequently uneven, making FUs available for dedication. Also, many temporary variables in such a scenario have short life-spans, thus creating opportunities for the REWIRED optimization. Further, resource constraints imposed on the system might typically keep in view the most resource-hungry parts of the application, leaving spare resources available for dedication in the rest of the program.

REWIRED may lead to a reduction in the number of registers in the datapath in cases where all the write operations assigned to a physical register are inhibited. Correspondingly, the controller area also reduces due to a reduction in the number of register load control signals to be generated by the FSM – the state table has fewer columns.

After REWIRED, there may be some register-to-register combinational paths in the datapath whose delay exceeds the clock period. These are, however, false paths because the schedule does incorporate the spreading out of the computation over multiple cycles. Such paths can be communicated to a timing analysis tool operating on the resulting design.

Cascaded function units not separated by a register sometimes lead to glitch activity that might cause power overheads. However, in our generated architecture, the propagation of such glitches is shielded by intervening MUXes; the MUX select signals change only when the appropriate state is reached. In the example of Figure 3, $+_1$ and $+_2$ are cascaded, but the signal activity propagation into $+_2$ is shielded by multiplexer M1.

REWIRED attempts to reduce power by eliminating register stores at the end of each clock cycle boundary. In pipelined systems, this could lead to a reduction in throughput due to the dropping of register writes. For example the scheduled DFG with the FU binding and register allocation decisions in Figure 5(a), with two single-cycle adders and subtractors, can be pipelined with an initiation interval of 1. When $+_1$ is dedicated to eliminate register a (Figure 5(b)), the pipeline initiation interval increases to 3.

III. FORMULATION AND APPROACH

The REWIRED optimization can be formulated as the problem of choosing the largest set of operations for which we can legally inhibit the storage of the results into registers. In our present formulation, this must be done while honoring the latency and resource constraint of the schedule and preserving functional correctness.

A. Register Contention

A dedicated resource must continue to perform the same operation until all the operations requiring the result are completed. An operation is said to be completed when its result is stored in a storage unit (memory/register) or there are no other computations operating on the computed result. During this period its inputs must be held constant. By “continue to perform”, we only mean that the FU’s output will not change – no activity takes place in the module and zero additional dynamic power is dissipated.

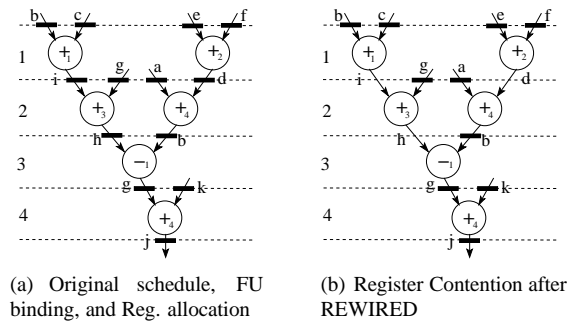


Fig. 4. Register Contention

Consider an example DFG shown in Figure 4(a) after scheduling, FU binding and register allocation. Let the resource constraint be four single-cycled adders and one subtractor. The FU binding is indicated by $+_1$, $+_2$ etc. The labels on the dark boxes at DFG edges represent the register allocation decisions – edges with the same name represent values stored in the same physical register. Assume that $+_1$, $+_2$, and $+_3$ are dedicated adders and $+_4$ is shared by additions in cycles 2 and

4. As we have one subtract operation and subtractor resource; if suppose operation $h - b$ (-1) is dedicated then the inputs to $+1$ cannot be changed until the end of the third cycle. The situation is shown in Figure 4(b). Since $+4$ is reused in cycle 4, its output in cycle 2 needs to be stored in a register. The assigned register is b , which also provides input data to $+1$ in the existing register allocation; this leads to contention for the register b . The REWIRED algorithm needs to be able to avoid such contention.

B. Resource Contention

In the proposed approach, resource dedication to the operations is performed under the given resource and latency constraints of the schedule. Resource dedication to an operation may not be possible due to lack of available resources.

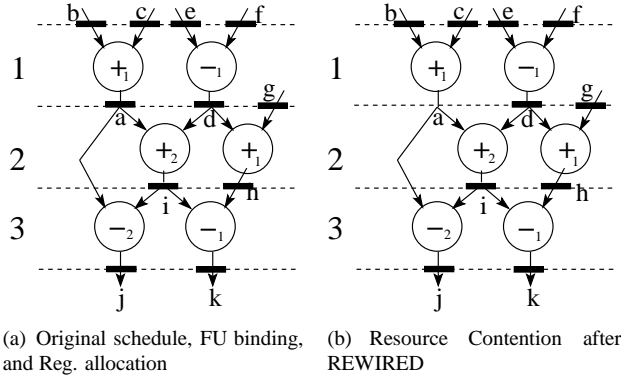


Fig. 5. Resource Contention

Consider an example scheduled DFG shown in Figure 5(a) with two single-cycle adders and subtractors as resource constraint. Assume that the resource $+1$ is dedicated to the operation $b + c$ of the first cycle as shown in Figure 5(b). It must continue to perform the same operation until the end-of third cycle. Thus the two addition operations cannot be assigned to cycle 2, leading to an increase in latency.

C. Genetic Algorithm for REWIRED

The register inhibition problem can be shown to be NP-complete. We omit the proof due to lack of space. We first present a genetic algorithm formulation for the REWIRED optimization.

C.1 Encoding

Encoding encompasses two important aspects, Genes and Chromosomes.

- **Chromosomes:** A chromosome is a bit-string of 0's and 1's, where each bit corresponds to a value (edge crossing a cycle boundary in the scheduled DFG).
- **Genes:** Genes in our framework refer to bits. A '0' in the i -th position of the bit-string indicates that the corresponding value is stored in a register, and a '1' value indicates that the storage of the value into registers is inhibited.

C.2 Fitness Function

The fitness of a chromosome x depends on the output of the following functions

- **Dedication(x):** A larger number of 1's in a chromosome means a larger number of register writes are inhibited. This function returns the number of 1's in the chromosome.
- **Valid(x):** We require that two conditions be satisfied for a chromosome to be valid: (1) *resource constraints*: the number of resources utilized in each cycle should not be greater than the resource constraints; (2) *forced 0's*: Since we are not re-assigning registers (we may only drop a register from storing a value), it may be required that registers be set to certain outputs, hence forcing the FUs generating their values to write into registers, i.e. a '0' is implied in the corresponding gene. Hence we need to check whether the chromosome satisfies this condition. The *Valid* function returns 1 or -1 depending on whether the chromosome is valid or not.
- **ResConstr(x):** If the number of resources utilized in a cycle is close to the maximum allowed, then it will be difficult to increase the number of resource dedications on this chromosome. Hence the expression:

$$\sum_{i=1}^m Res_{con} - Res_i$$

must be as large as possible for a fitter chromosome, where, Res_{con} = Total number of resources available for all FU types, Res_i = Number of resources used in the i th cycle, m = number of cycles (schedule length)

The fitness function is evaluated as:

$c_1 \times Dedication(x) + c_2 \times Valid(x) + c_3 \times ResConstr(x)$, where c_1, c_2, c_3 are constants determined by the number of genes and the schedule length.

C.3 Reproduction

To proceed to future generations, we select parents equal in number to the current population. The selection is performed in proportion to fitness function defined above, ensuring that the fitter parents have a high probability of recurring in the pool. Following this, the children are generated using two processes.

- **Crossover:** We select two parents $f[1..n]$ and $g[1..n]$ from the current pool, where n is the number of values under consideration, and generate a child p which takes bits from f and g in such a way that the probability of selection of a gene from the respective parents is proportional to their fitness values. If the fitnesses are in ratio $r : (n - r)$, then the child has r bits from f and $(n - r)$ bits from g . As in the standard crossover operation, a second child p is generated with the remaining $(n - r)$ bits from g and r bits from f .
- **Mutation:** After the crossover process we examine each gene and flip it with a certain mutation probability.

The above two processes yield children equal in number to the original population. We then combine the two populations, sort them according to their fitness values, and retain the fitter half to form the next generation. The process continues until there is no further improvement in the quality of results.

Algorithm 1 REWIRED Heuristic

Input: Scheduled and Register Allocated CDFG

Output: CDFG with REWIRED optimization

- 1: Compute *Available* resources in each cycle
 - 2: $Available[i][j] = R_j - \sum_{k=1}^{R_j} Scheduled_{i,j,k}$
/* *Available[i][j]* gives number of available resources of type *j* in i^{th} cycle */
 - 3: Compute life-time of each operation's result
 - 4: Sort all the operations in the increasing order of their result's life-time
 - 5: $DedicatedOps \leftarrow \Phi$
 - 6: **for all** operations *currOp* in sorted list **do**
 - 7: $DedicatedOps \leftarrow DedicatedOps \cup \{currOp\}$
 - 8: Store *Available[][]* values into *avail[][]*
 - 9: **for all** $Op \in DedicatedOps$ **do**
 - 10: **if** *Op* stores result in Global Register **then**
 - 11: Continue with next *Op* in *DedicatedOps*
 - 12: Recompute life-time of *Op* result
 - 13: **for** $c = Op.life-time.start$ to $Op.life-time.end$ **do**
 - 14: $avail[c][Op.res] \leftarrow avail[c][Op.res] - 1$
 - 15: **if** $avail[c][Op.res] < 0$ **then**
 - 16: **Goto Failure** /* Resource Contention */
 - 17: **for each** input register *r* of *Op* **do**
 - 18: Extend life-time of *r* to $Op.life-time.end$
 - 19: **if** any overlap for extended life-time **then**
 - 20: **Goto Failure** /* Register Contention */
 - 21: Allocate dedicated resource to *currOp* and inhibit register store
 - 22: Continue with next operation in sorted list
 - 23: **Failure:** Allocate resource and store result in register
 - 24: $DedicatedOps \leftarrow DedicatedOps - \{currOp\}$
 - 25: **return** CDFG
-

D. REWIRED Heuristic

We present a fast heuristic for performing the REWIRED optimization (Algorithm 1). The intuition is to greedily choose those operations whose dedication leads to a lower probability of register contention and resource contention. We assign a higher priority to operations whose results have shorter life-times as that would reduce the possibility of conflicts with other binding and register allocation decisions. This reduces the resource pressure in the successive cycles and thereby maximizes the total number of possible dedications.

Algorithm 1 begins by first computing the resources that are available in each cycle of the scheduled DFG. This is computed as the difference between the number of instances available of resource type j ($= R_j$), and the number of resources actually scheduled in cycle i . $Scheduled_{i,j,k}$ is defined as 1 if the resource instance is scheduled in cycle i and 0 if it is free in that cycle. The initial life-times of each operation's result

are computed (as discussed earlier, these may change as a result of FU dedication decisions). We sort all the operations in the increasing order of their life-times (operation's life-time being synonymous with the life-time of its resulting value). We then scan the sorted list in increasing order, and for each new node/operation encountered, check if dedicating this resource would lead to register- or resource contention against existing dedication decisions. If not, then a resource is dedicated to the computation to inhibit the register store; otherwise the resource is allocated and the computed result is stored in a register. Each operation is considered only once and dedication decisions are never re-evaluated.

The original *Available* array is never modified. For every tentative dedication attempt, we copy it over to a temporary array *avail*, and check the effect of the dedication on resource availability. If we exceed the resource constraint, the dedication is infeasible and the register write cannot be inhibited. If the current operation writes into a global register, then the writing cannot be inhibited.

IV. EXPERIMENTS

To validate the REWIRED optimization, we used examples from different suites such as Livermore, HLS-95, Mediabench and Mibench. *DiffEq*, *Elliptic*, and *Tap* are relatively smaller examples; and *ADI*, *FFT*, *IDCT*, and *MESA* are larger – Table I shows the number of register writes in each example. The hardware for both the RTL VHDL models was synthesized using Synopsys Design Compiler and a 0.18μ ASIC library. The power simulations were performed by feeding the results from the VHDL simulation to Synopsys Prime Power.

Figure 6 shows a comparison of the power dissipation in the synthesized circuits corresponding to the conventional synthesis, with that of the circuits resulting from REWIRED. An average power reduction of 15% is observed. There is an average area reduction of 2% due to the reduced registers.

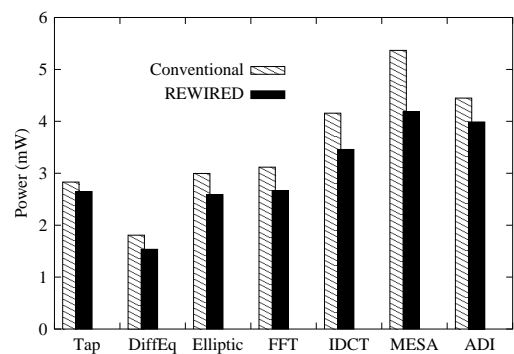


Fig. 6. Power reduction with REWIRED

When resource constraints are specified by considering the computationally heavy part of an application, the remaining basic blocks have a surplus of resources increasing the possibility of resource dedication. To validate this, we performed experiments by increasing the allocated resources for three examples. On increasing the available resources, the schedule latency does not reduce further due to the dependencies in between the

operations. Thus, there is no change in the power consumption in the traditional HLS. But, when we applied REWIRED, the power did reduce further, as shown in Figure 7.

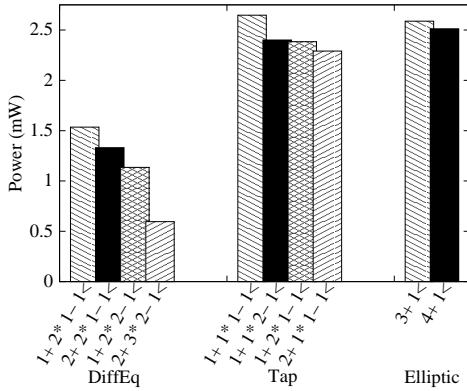


Fig. 7. Power improvement with different resource allocations

A. Genetic Algorithm Vs Heuristic

Table I shows comparison of the number of register load/stores saved with the genetic algorithm and the proposed heuristic. Even though the heuristic is greedy, the results come close to that produced by the genetic algorithm – the percentage of inhibited writes as a fraction of the total number of writes comes within an average of 4%. For this experiment, the genetic algorithm was run up to a maximum of 114,000 generations by when it showed near complete convergence; the GA result can be considered near-optimal. The mutation probability was 0.25. The genetic algorithm took more than an hour for executing some of the examples compared to less than 1.5 seconds for the heuristic.

App. Name	Resource Constraints	Total Writes	Write Inhibits	
			Gen	Heu
Elliptic	4 +, 1 <	44	10	9
	3 +, 1 <	44	9	7
Tap	1 +, 1 -, 1 *, 1 <	30	3	2
	1 +, 2 -, 1 *, 1 <	30	4	3
	2 +, 1 -, 1 *, 1 <	30	5	5
	1 +, 1 -, 2 *, 1 <	30	5	4
Diffeq	1 +, 1 -, 2 *, 1 <	16	3	3
	2 +, 1 -, 2 *, 1 <	16	6	4
	2 +, 2 -, 3 *, 1 <	16	6	6
	1 +, 2 -, 2 *, 1 <	16	6	4
FFT	3 +, 4 *, 3 -, 1 <	49	5	4
IDCT	3 +, 3 *, 3 -, 5 <<, 1 <	80	29	26
MESA	2 +, 3 *, 2 -, 1 <	51	13	12
ADI	3 +, 3 *, 3 -, 1 <	69	18	14

TABLE I
GENETIC ALGORITHM VS. HEURISTIC

V. CONCLUSION AND FUTURE WORK

We presented REWIRED, a technique for low power behavioral synthesis that attempts to reduce the number of actual reg-

ister stores by exploiting the knowledge of stable input registers to function units. Dynamic power is reduced by avoiding the register store activity. We outlined two techniques for register write inhibition – a genetic algorithm formulation and a heuristic approach. Experimental results show promising power reduction results, with the heuristic approach giving results similar to the genetic algorithm, while running much faster. There is also a small area reduction corresponding to cases where avoiding register stores results in fewer registers. Power reduction due to REWIRED increases further if more resources are available.

Power reduction could be possibly enhanced if scheduling decisions could be revisited, making the scheduler aware of the inhibition possibility; the schedule latency could be permitted to increase while reducing the total energy. The analysis could also be extended in the future to span across basic block boundaries.

REFERENCES

- [1] E. Musoll and J. Cortadella, "Scheduling and resource binding for low power," ISSS, Cannes, France, 1995
- [2] A. Dasgupta and R. Karri, "Simultaneous scheduling and binding for power minimization during microarchitecture synthesis," ISLPED, Dana Point, USA, 1995
- [3] Y. Zhang, et al., "Global register allocation for minimizing energy consumption," ISLPED, San Diego, USA, 1999.
- [4] J.-M. Chang and M. Pedram, "Register allocation and binding for low power," DAC, San Francisco, 1995
- [5] M. Pedram and J. Chang, "Module assignment for low power," EURO-DAC, Geneva, Switzerland, 1996
- [6] C. H. Gebotys, "Low energy memory and register allocation using network flow," DAC, Anaheim, USA, 1997
- [7] W. Yang, et al., "Low-power high-level synthesis using latches," ASPDAC, Yokohama, Japan, 2001
- [8] N. Chabini and W. Wolf, "Unification of scheduling, binding, and retiming to reduce power consumption under timing and resources constraints," IEEE TVLSI, 13(10), 2005.
- [9] A. Davoodi and A. Srivastava, "Effective techniques for the generalized low power binding problem," ACM TODAES, Jan 2006
- [10] J. Luo, et al., "Register binding-based rtl power management for control-flow intensive designs," IEEE TCAD, Aug 2004.
- [11] G. Lakshminarayana, et al., "Transforming control-intensive designs to facilitate power management," ICCAD, San Jose, USA, 1998
- [12] D. Chen and J. Cong, "Register binding and port assignment for multiplexer optimization," ASPDAC, Yokohama, Japan, 2004