# Automatic ADL-based Operand Isolation for Embedded Processors

A. Chattopadhyay, B. Geukes, D. Kammler, E. M. Witte, O. Schliebusch, H. Ishebabi,
R. Leupers, G. Ascheid, H. Meyr
Integrated Signal Processing Systems
RWTH Aachen University 52056 Aachen, Germany
anupam@iss.rwth-aachen.de

## Abstract

*Cutting-edge applications of future embedded systems demand highest processor performance with low power consumption to get acceptable battery-life times. Therefore, low power optimization techniques are strongly applied during the development of modern Application Specific Instruction Set Processors (ASIPs). Electronic System Level design tools based on Architecture Description Languages (ADL) offer a significant reduction in design time and effort by automatically generating the software tool-suite as well as the Register Transfer Level (RTL) description of the processor. In this paper, the automation of power optimization in ADL-based RTL generation is addressed.*

*Operand isolation is a well-known power optimization technique applicable at all stages of processor development. With increasing design complexitiy several efforts have been undertaken to automate operand isolation. In pipelined datapaths, where isolating signals are often implicitly available, the traditional RTL-based approach introduces unnecessary overhead. We propose an approach which extracts high-level structural information from the ADL representation and systematically uses the available control signals. Our experiments with state-of-the-art embedded processors show a significant power reduction (improvement in power efficiency).*

## 1. Introduction

Nowadays, ASIPs are ubiquitous due to the unique combination of performance and flexibility offered by them. ADLs [8] [2] are employed to model the ASIP in a higher level of abstraction than RTL, thereby reducing the design effort significantly. The architectural information available in this high-level abstraction has been successfully used to optimize the processor [13] in terms of area or delay. However, effective power reduction techniques using the ADL-based description are yet to be explored. On the other hand, demand for power-efficiency, especially for mobile hand-held embedded systems, is growing strongly.

Power-specific optimizations can be applied at all levels of abstraction during a processor design. In the gate-level design, low power can be achieved by introducing specially designed cell libraries, at the RTL by introducing selective blocking of operands or at the system level by shutting off sections of the design. In principle, the power optimization techniques can be classified into two major categories. In one group of power optimization techniques, the dynamic power of a design is controlled by reducing the *clocking* activity of the sequential elements. Exemplarily, *Clock gating* [11] is a power-reduction technique based on the above principle, which disables the clock entering into a register whenever an unnecessary storage takes place. In the other group, the combinational components of a design are controlled in order to minimize power consumption. *Operand Isolation* is one such technique. Employing operand isolation, *redundant* combinational blocks of a design are identified. Here, redundant means in temporal perspective i.e. a combinational operation is performed at a particular time, when its result is not getting used in the downstream circuit. By operand isolation, these temporarily redundant operations are identified and the operands of these operations are held at a particular value to reduce the switching activity of the combinational circuit. This, in turn, reduces the dynamic power significantly. The concept of operand isolation is shown using figure 1, where a part of a datapath is shown. Here, the leftmost adder can be labeled redundant at a particular time, when either *sel_1* or *sel_2* is set to the value 0. In that case, the primary output *out_1* does not use the result of the addition. The operators (e.g. adder), for which isolation logic can be inserted to reduce power are termed as *isolation candidates*. The signals, which control the isolation logic (*IS_1* and *IS_2*) are termed as *isolating signal*. This isolating signal is set to high, when the result of the isolation candidate is relevant in the primary output. The rectangular box inserted before the adders represent *isolation logic*. Typically AND-based, OR-based or latch-based isolation logic is employed. For AND-based isolation and OR-based isolation, the input operands are held at value 0 or 1 respectively. For latch-based isolation, the input operand value of the previous cycle is held. For both types of operand isolation, isolation logic and the circuit to prepare the isolating signal (shown as a cloud) introduces area and delay overhead. The power dissipation of these added circuitry may cancel the gain achieved by operand isolation. We implemented automatic generation of logic-based operand isolation as it introduces comparatively less area and delay overhead [9].

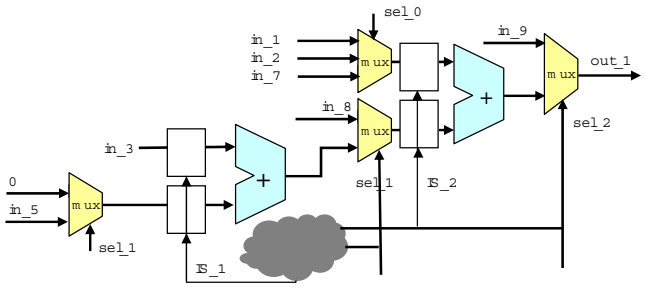


**Figure 1. Example of Operand Isolation**

### 1.1 Related Work

Implementation or automation of operand isolation techniques at various levels of abstraction has been reported in literature. Across

abstraction levels, the issues faced in operand isolation are quite similar in nature. First, one has to select the candidates for operand isolation. Second, the isolating signal has to be created or re-used from existing circuitry. Finally, the isolation logic has to be inserted in a suitable position of the circuit. The existing operand isolation approaches at different abstraction levels can be reviewed in the perspective of the three abovementioned steps.

During the implementation of IBM PowerPC 4xx microcontrollers, operand isolation has been implemented manually [4]. There, nothing specific is mentioned about the candidate selection. Typically the isolation candidates are shown to reside before a result bus multiplexer and the control signal of the multiplexer is re-used as the isolating signal (figure 4 of [4]). The isolating logic is inserted immediately before the isolation candidate. Tiwari et al [17] performed operand isolation automatically at gate-level. The approach exploits the gate-level granularity by allowing the isolation to be performed over any arbitrary logic circuit. The existing control signals are directly used as isolating signal and the isolation candidate is selected on the basis of potential power savings. The isolation logic is inserted at a point of circuit, where the isolating signal arrives pretty earlier than the actual transition. This way of derivation of the isolating signal limits the scope of identification of possible isolation candidates. In a recent work by Banerjee et al [10], novel gate-level circuits are introduced to perform operand isolation. This approach requires special cells to perform operand isolation and it can be complemented with high-level operand isolation schemes.

The approach adopted by Münch et al [9] allows automated RTL-based operand isolation. In this approach, the isolation candidates are determined on the basis of a detailed power model. The isolating signal is created out of a circuit based on the output Observability Don't Care (ODC) conditions of the isolation candidates. The concept of ODC can be explained using figure 1. In case of the rightmost adder of the figure 1, the result of the addition is not observable at *out_1* when *sel_2* is 0. Therefore, for the input operands of the adder, the ODC is simply $\overline{sel\_2}$. In Münch's approach, the isolating signal for each input operand of an operator is generated from the operator's output ODC. For a large datapath block, the primary inputs and outputs are defined by partitioning the complete circuit across sequential logic boundaries.

Operand Isolation techniques have been incorporated during high-level synthesis, too. In the approach mentioned at [6], operand isolation is performed independent of other high-level synthesis tasks and a blocking latch is inserted before the functional units. Details of isolating signal generation or isolation candidate selection are not presented. In the context of ADL-based high-level synthesis, the RTL generator GO [16], based on the ADL nML [1], is known to contain power-specific optimizations like disabling of functional units. Any detailed analyses of these optimizations are not publicly available.

## 1.2 Motivation

With the above discussion about available operand isolation approaches, the motivation of this work is now presented. A key issue in operand isolation is the derivation of the isolating signal itself. Consider figure 2 for example.

Here the steering signals (*in_steer_1* and *in_steer_2*) are primary inputs of the datapath block. The steering signals along with the result of a local addition produce the multiplexer controlling signal *sel_2* through a selection logic. As mentioned by Münch et al [9], the derivation of the relevant steering signals from the complex RTL structure is often impossibly difficult. The approach taken in [9] is to use the multiplexer control signals (e.g. *sel_1*) and to derive a logic circuit generating the isolating signal. Obviously, the insertion of logic circuit introduces area and delay overhead. In case of this

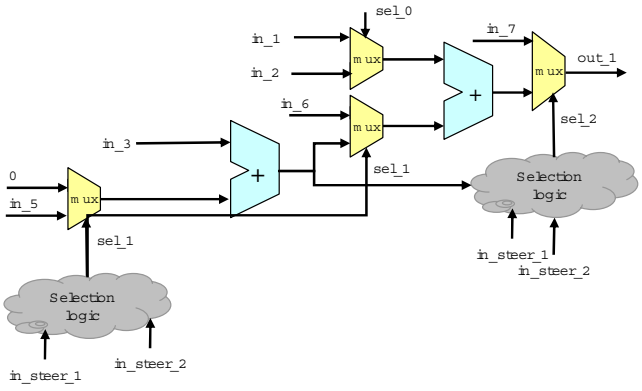


**Figure 2. Motivation for ADL-based Operand Isolation**

overhead being significant the operand isolation is not performed at all. Thus the trade-off between power, area and delay is heavily biased. Either the isolation is done at finest granularity or not done at all. In this paper, this flexibility is introduced by deriving the steering signals from high level architectural information. For low-overhead *coarse-grained* operand isolation, the steering signal is directly used as an isolating signal. Alternatively, with the knowledge of the dataflow in the datapath blocks, ODC-based operand isolation can be implemented. In summary, the major contribution of this paper is to present:

- An automatic ADL-based operand isolation framework.
- A flexible operand isolation approach for pipelined processors.

In addition to these, we propose a minor extension to the ODC-based operand isolation algorithm. Consider previous figure 1. Following the ODC-based approach outlined at [9], the isolation logic will be placed immediately before the isolation candidates. However, the cases where the input is already blocked for similar conditions do not need another isolation. As in figure 1, *IS_1* is derived correctly from *sel_1* and *sel_2* (as in ODC-based approach), whereas *sel_1* is already used to isolate *in_5*. These situations often occur for datapath involving bit-masking operations. Clearly, the isolation logic is redundant considering logic-based isolation. We identified this redundancy manually in the generated RTL description and eliminated the isolation overhead. A low overhead in isolation, in turn, results in further power reduction.

The rest of the paper is organized as follows: section 2 discusses the relevant features of ADL LISA, which is used for this work. Section 3 describes the framework for low power optimization. In section 4, the automatic generation of operand isolation from the ADL is described in detail. Section 5 elaborates and analyzes our case study. We conclude with the summary and outlook.

# 2. ADL Structure Overview

In this section, a brief overview of the ADL LISA [8] is provided. Furthermore, the relevant language elements and their corresponding mapping to a processor are discussed.

## 2.1 LISA Operation Graph

In LISA, an *operation* is the central element to describe the timing and the behavior of a processor instruction. The instruction may be split among several LISA operations. The resources (registers, memories, pins etc.) are declared globally in the *resource* section, which can be accessed from any LISA operation.

The LISA description is based on the principle that a specific common behavior or common instruction encoding is described in a single operation whereas the specialized behavior or encoding is

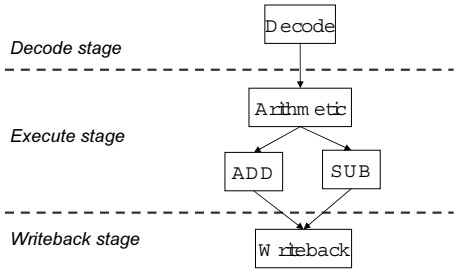


**Figure 3. LISA Operation DAG**

implemented in its *child* operations. Specialized operations may be referred to by more than one *parent* operation. The complete structure is a Directed Acyclic Graph (DAG) $\mathcal{D} = \langle V, E \rangle$. $V$ represents the set of LISA operations, $E$ the graph edges as set of child-parent relations. These relations represent *activations*, which refer to the execution of another LISA operation. Figure 3 gives an example of a LISA operation DAG. As shown, the operations can be distributed over several pipeline stages. A chain of operations, forming a complete branch of the LISA operation DAG represents an instruction in the modelled processor. A LISA Operation contains different subsections to capture the entire processor behavior. The ones relevant for RTL synthesis are discussed below.

**Instruction Coding Description:** The instruction encoding of a LISA operation is described as a sequence of *coding fields*. Each coding field is either a terminal bit sequence with "0", "1", "don't care"(X) bits or a nonterminal bit sequence referring to the coding field of another child LISA operation.

**Activations:** A LISA operation can *activate* other operations in the same or a later pipeline stage. In either case, the child operation may be activated *directly* or via a *group*. A *group* collects several LISA operations, with the elements being mutually exclusive. The elements are distinguished by a unique binary coding, forming a coding tree. The activation and instruction coding tree jointly form the decoder in the target processor. The activation chain also provides valuable information regarding the dataflow in the complete processor. The activation edge is transformed as a major steering signal in the processor datapath. For each LISA operation, a corresponding steering signal is existent. The instruction coding tree generates minor steering signals within the scope of one LISA Operation.

**Behavior Description:** The behavior description of a LISA operation corresponds to the datapath of the processor. Inside the behavior description plain C code can be used. Resources such as registers, memories, signals and pins as well as coding elements can be accessed in the same way as ordinary variables. The behavior section of every *LISA operation* is transformed into a *functional block* in the RTL datapath.

# 3. Power Optimization Framework

Before elaborating on the power-based optimization framework, a typical pipelined processor structure is explained. In the remaining part of this section, we introduce the data flow graph, which is used for operand isolation.

## 3.1 Processor Structure

The figure 4 shows a typical simplescalar pipelined processor structure. The detailed connections are avoided for simplicity. The pipeline is divided into 4 stages. In this processor, the *register file* is residing outside the pipeline. In this implementation, the decoder is distributed over the complete pipeline. For each datapath block, the local decoder issues a major steering signal. Corresponding to the ADL elements discussed in the previous section, the activation and

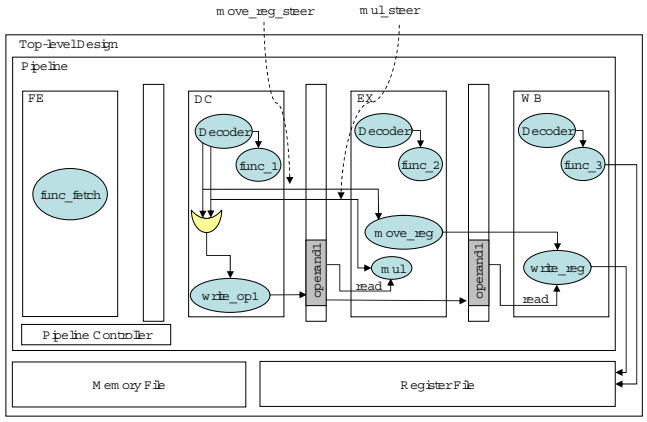


**Figure 4. Pipelined Processor with Distributed Decoding**

the instruction coding jointly contributes to this decoder formation. Clearly, ADL-based RTL synthesis allows an easy access to high-level structural information in order to identify the steering signals from a complex logic structure. For detailed understanding of RTL processor synthesis from ADLs, please refer to [13] and/or [12].

Note that, it is possible to derive and use these steering signals for other pipeline organizations, too. For a pipeline with centralized decoder, the steering signals need to be propagated to the relevant datapath. This concept of using steering signals for pipelined datapaths has been succesfully used for clock gating in [7]. However, no automatic approach is available. Understandably, it is difficult to derive these signals from complex RTL structure.

As observed in [9], the ODC-based operand isolation algorithms become extremely complex if the complete design is considered for ODC calculation. A much simpler and effective approach is to partition the datapath with the sequential cells in the boundary and then apply operand isolation locally. In the context of pipelined processors, this partition occurs naturally within one stage. The operand isolation algorithms are applied within these partitions i.e. within the scope of one *LISA operation*.

## 3.2 Behavioral Data Flow Graph

Inside a single LISA operation, the *behavior* section guides the data flow. The behavior section of a LISA operation is converted into a pure, directed acyclic Data Flow Graph (DFG). The graph vertices of $G_{\text{DFG}} = \langle V_{\text{op}}, E_{\text{ic}} \rangle$ are the basic *operators* for data manipulation e.g. additions while edges represent the flow of unchanged data in form of *interconnections* of inputs and outputs.

**Operators:** The following list summarizes the basic classes of operators represented by graph vertices. This special choice of vertices allows us to represent the data flow information in a level between RTL and logic-level representation. In that way, our representation is close to Bergamaschi's *Behavioral Network Graph* [5].

- Commutative $n$-ary Operator, $n \geq 2$
- Noncommutative $n$-ary Operator, $n \geq 1$
- Read Access to Registers and Memories
- Write Access to Registers and Memories
- Read and Write Access to Array of Variable
- Multiplexer

**Interconnections:** Interconnections represent the data flow on symbol-level opposed to bit-level representations used in gate-level synthesis. The information about the data type transferred is given by an annotation to the interconnection. Bit range subscriptions are included into the interconnection information, too.

The creation of the DFG from the plain C-code of a LISA operation's behavior section is shown in figure 5. As depicted there,

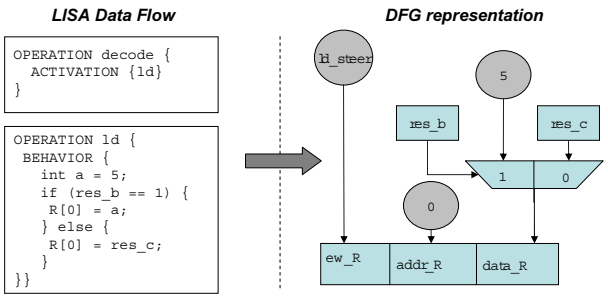


**Figure 5. Example of Data Flow Graph Creation**

the DFG is constructed after performing basic compiler-like optimizations. In this case, the constant value of local variable *a* is propagated. For the read access to non-array registers e.g. *res_c*, we need not pass any address value. For the write access to a one-dimensional resource *R*, the write enable and the address value is set in the scope of the same vertex. The value for write enable is set to *ld_steer*, which indicates that this operation is to be executed or not.

# 4. Automatic Operand Isolation

In this section, different operand isolation techniques are explained. The isolation constraints used during automatic operand isolation are mentioned. Finally, the algorithms for instantiating operand isolations are outlined and the overall flow is presented.

## 4.1 Operand Isolation Techniques

In figure 6, there is one major steering input (indicative of the complete datapath block execution) and one minor steering input (*in_steer_local*), which is used within the context of this datapath block. Both the steering inputs are directly available as primary inputs. The steering inputs are fed into combinational logic (shown as cloud) to prepare the multiplexer controlling signals *sel_1* and the enable signal for a target register.

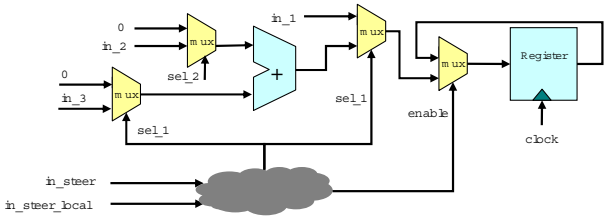


**Figure 6. Datapath without Operand Isolation**

**Coarse-grained Operand Isolation:** To perform coarse-grained operand isolation, we rely completely on the major steering signal for the datapath block. A coarse-grained isolation over the complete datapath block can be done as shown in figure 7. Obviously, blocking the primary inputs leading to a simple logic gate will not be beneficial. Therefore, one needs to identify the operators and bit-widths within the scope of a datapath partition, for which the isolation is to be done. In order to allow this selective isolation, several high-level constraints can be introduced, which are discussed afterwards.

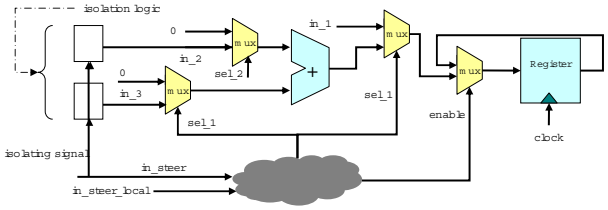


**Figure 7. Datapath with Coarse-grained Operand Isolation**

**ODC-based Operand Isolation:** For ODC-based operand isolation, the Observability Don't Cares of the primary outputs need to be considered. In this case, there is only one primary output (input of register). Traversing back from the enable signal of the primary output allows to create an isolating signal for the input operands of given logic blocks (as shown in figure 8). The isolating signals for all the input operands are derived to be the same. In this case, for the adder inputs, the isolating signal is obtained by performing logical *and* operation between *sel_1* and *enable*.

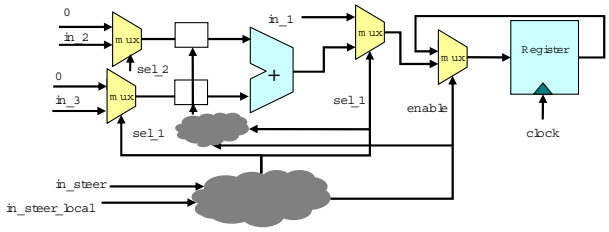


**Figure 8. Datapath with ODC-based Operand Isolation**

**Fine-grained Operand Isolation:** For this technique, the ODC-based isolation procedure is extended by considering the bit-masking at the input operands of the isolation candidates. As can be observed in figure 9, one input operand for the adder is held constant by a controlling signal (*sel_1*) also used in a later multiplexer. Therefore, it is sufficient to use *enable* as isolating signal. However, the same is not true for the input operand *in_2* where *sel_2* is the controlling signal. In this case, the area overhead will increase if *sel_2* is included in the isolating signal circuit. Hence, fine-grained operand isolation is useful only in the cases where the input operand is held constant under a condition, already included in the ODC condition.

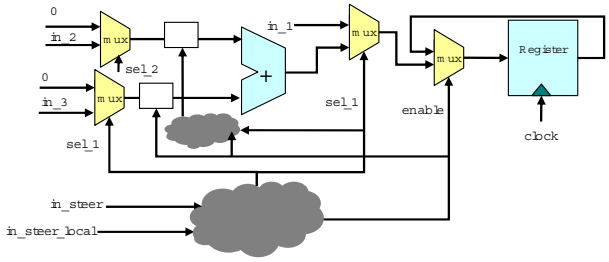


**Figure 9. Datapath with Fine-grained Operand Isolation**

## 4.2 Isolation Constraints

In some automated operand isolation approaches, a detailed power model has been considered for inserting operand isolation among isolation candidates [9]. The approach presented in this paper is integrated in a high-level synthesis framework, where the proper estimation of performance itself is a huge research area. We adopted an alternative approach by allowing several high-level constraints before inserting operand isolation. These constraints, unlike the model-based approach, grant more user interaction and therefore stronger control over the trade-offs. A brief summary of the constraints is mentioned here.

**Isolation Prevention:** A complete data flow block can be spared from operand isolation by inserting pragma in the ADL description. This is useful for dataflow blocks with high execution frequency.

**Operator Selection:** Selected operators can be pointed for operand isolation candidacy. For example, one can select only adders and multipliers to be isolated.

**Bit-width Selection:** A minimum bit-width (of operator) can be specified for being selected as an isolation candidate.

### 4.3 Algorithms for Automatic Operand Isolation

Here, the ADL-based coarse-grained and ODC-based operand isolation algorithms are outlined. Subsequently, the computation complexity for the algorithms are shown.

```
01 // In a pipeline stage, DP_j denotes a datapath block
02 // DP_j is completely represented by a dataflow graph
03 // node_k denotes a vertex of the dataflow graph
04 // pi_k denotes a primary input corresponding to node_k
05
06 InstantiateCoarseGrainedOperandIsolation(DP_j) {
07   // each datapath block can be stopped from isolation by a pragma
08   if (DP_j is covered by pragma) return;
09   for each node_k ϵ DP_j {
10     if (node_k satisfies isolation constraints) {
11       S(pi_k) = set_of_primary_inputs(node_k)
12       for each pi_km ϵ S(pi_k) {
13         if (pi_km is not already isolated) {
14           isolating_signal = steering_signal(DP_j)
15           insert_isolation_logic(pi_km, isolating_signal)
16 }}}}}
```

The coarse-grained operand isolation algorithm during ADL-driven high level synthesis is outlined in the above pseudo-code. For each datapath block, the nodes which satisfy given isolation constraints are treated. For every such node, primary inputs are tracked back and the isolation logic is coupled with the primary inputs. The isolating signal is given by the major steering signal of the datapath block, which is directly available via high-level structural information. The runtime complexity of the above algorithm is determined by the loop over primary inputs (line 12). For a dataflow graph with $n$ nodes, the worst case complexity is $O(n^2)$.

The following pseudo-code contains the essence of the ODC-based operand isolation algorithm. The ODC is propagated from the datapath block's primary outputs to the isolation candidate's input operands. The isolating signal is derived from the ODC itself. The runtime complexity of the above algorithm is determined by the computation of the ODC for each input operand of the isolation node. ODC propagation per node is done in linear complexity. Considering the number of input operands per operator to be $p$, the worst case complexity is $O(p \cdot n^2)$.

```
01 // In a pipeline stage, DP_j denotes a datapath block
02 // DP_j is completely represented by a dataflow graph
03 // node_k denotes a vertex of the dataflow graph
04 // po_k denotes a primary output corresponding to node_k
05
06 InstantiateFineGrainedOperandIsolation(DP_j) {
07   // each datapath block can be stopped from isolation by a pragma
08   if (DP_j is covered by pragma) return;
09   for each node_k ϵ DP_j {
10     if (node_k satisfies isolation constraints) {
11       S(po_k) = set_of_primary_outputs(node_k)
12       // calculating Observability Don't Care
13       ODC_out = Propagate_ODC(S(po_k), node_k)
14       S(i_k) = set_of_input_operands(node_k)
15       for each i_km ϵ S(i_k) {
16         isolating_signal = ¬ODC_out
17         insert_isolation_logic(i_km, isolating_signal)
18 }}}}
```

### 4.4 Overall Flow

The overall flow of the automatic ADL-based operand isolation is shown in the figure 10. A global analysis of a processor's ADL description is done at first. The datapath blocks of the ADL in each pipeline stage is then converted to corresponding DFG representation. By using the architectural information and the isolation constraints, a designer-selected algorithm is employed to instantiate the operand isolation. The DFG-representation is then mapped to register transfer level HDL description via HDL backend. It must be noted that to map the overall processor structure and the control path to HDL, various other steps are involved, which are not shown in this flow. The isolation constraints and the automatic operand isolation algorithms are integrated into a more elaborate framework for ADL-drived RTL synthesis.

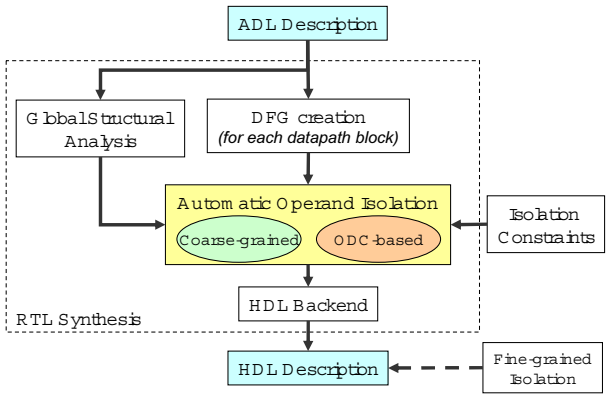


**Figure 10. ADL-based Operand Isolation Flow**

The selection of the isolation algorithm plays an important role to minimize the power. Currently, the isolation algorithm can be selected for the overall architecture, whereas it may be more beneficial to have a different isolation algorithms for different datapath blocks. This extension will be targeted and studied in our future work. Another important issue is the derivation of the isolation constraints. In particular, to derive the execution frequencies of datapath blocks, the designer has to run simulations. With the framework of ADL, fast cycle-accurate instruction-set simulation [3] can be performed. The instruction-set simulator is automatically generated from the ADL description.

## 5. Case Study

The power optimizations discussed in this paper are tested with two different ASIPs. The ICORE [15] architecture is dedicated for Terrestrial Digital Video Broadcast (DVB-T) decoding. It is based on a pipelined Harvard architecture implementing a set of general purpose arithmetic instructions as well as specialized trigonometric operations. We took two applications performing trigonometric calculations namely, cordic01 and cordic02 running on this architecture. The second architecture is an ASIP dedicated for Fast Fourier Transformation (FFT) algorithms. A 32-point FFT application is used as the test application. Both the ICORE and the FFT architecture have been developed using LISA. A brief summary of the architectures is shown in table 1.

**Table 1. Summary of the Benchmark Architectures**

| | ICORE | FFT |
|---|---|---|
| Basis Architecture | 32-bit RISC | 24-bit RISC |
| Pipeline Stages | 4 | 6 |
| Lines of LISA Code | 2200 | 1500 |
| Lines of Verilog Code | 25200 | 16600 |

The LISA models were synthesized to obtain the RTL with and without various low power optimizations. The automatically generated RTL description was verified through RTL and gate-level simulation using Synopsys VCS and then synthesized with Synopsys

Design Compiler, using a $0.13\mu$m technology library. Finally, Synopsys power measurement flow [14] was employed to measure the power on gate-level. During the case study, the isolation constraints have been used to guide the power optimization of the entire processor. The datapath blocks, which are executed with a high frequency, have not been considered for isolation. The operators with low bitwidth and less complexity, have been ommitted, too.

Operand isolation, by its nature, affects the datapath partitions strongly. However, in both the architectures, the RegisterFile accounts for a high percentage of the power (65% for FFT and 37% for ICORE). To determine the clear effect of the operand isolation algorithms, the power improvements for the pipelined datapath is studied. The best power results obtained for both architectures are presented in table 2. The corresponding changes in area are given, too. Interestingly, the best power improvements for ICORE are available with ODC-based and fine-grained operand isolation, whereas for FFT processor it is the coarse-grained operand isolation. The reason behind this is the fine-grained operand isolation for the FFT introduced area overhead and thereof power increment, which offsets the gain. Moreover, for FFT, there are some datapath partitions, which got isolated but are active almost entirely during the application. Those partitions are identified manually and removed by inserting pragmas in the ADL description.

**Table 2. Effect of Operand Isolation over Pipeline**

| Benchmark | No Operand Isolation | | With Operand Isolation | |
|---|---|---|---|---|
| | Area [Gates] | Power [mW] | Area [Gates] | Power [mW] |
| cordic01 | 29145 | 9.32 | 34599(+18.71%) | 5.41(-41.95%) |
| cordic02 | | 8.35 | 34671(+18.96%) | 5.79(-30.67%) |
| fft32 | 12026 | 2.66 | 13198(+9.75%) | 2.39(-10.07%) |

In the following, a study of the effects of operand isolation in the overall architecture for ICORE is presented. Table 3 summarizes the power values obtained by applying different isolation algorithms for ICORE. The relative improvement of power with respect to the case of no operand isolation is shown in percentage. As expected, the coarse-grained isolation algorithm produces least benefit whereas, in general, the fine-grained operand isolation improves power most. It must be noted here, that these values reflect the overall architecture measurement and the power improvement is significantly higher for individual datapath partitions.

**Table 3. Overall Power (mW) results for ICORE**

| | cordic01 | | cordic02 | |
|---|---|---|---|---|
| | AND-based | OR-based | AND-based | OR-based |
| Original | 14.74 | | 13.24 | |
| Coarse-grained | 11.21(-23.95%) | 12.62(-14.38%) | 10.80(-18.43%) | 12.53(-5.36%) |
| ODC-based | 11.14(-24.42%) | 12.16(-17.50%) | 10.37(-21.68%) | 12.41(-8.31%) |
| Fine-grained | 10.97(-25.58%) | 12.08(-18.05%) | 10.73(-18.96%) | 12.04(-9.06%) |

Table 4 summarizes the results of area and delay obtained under different isolation algorithms. The delay due to isolation logic insertion is never significant. The main underlying reason for this is the datapath blocks in case of ICORE do not have complicated conditional chains. However, the isolation logic increased the area significantly for all isolation algorithms. Interestingly, the coarse-grained isolation algorithm resulted in bigger area than fine-grained or ODC-based isolation algorithm. Note that, for coarse-grained operand isolation, the isolation logic appears at the primary inputs. For ODC-based or fine-grained operand isolation, the isolation logic and isolating signal are generated inside the datapath block, thereby giving some scope of logic optimization. A closer look revealed that for coarse-grained isolation, such optimizations did not occur. As a result, the area increase is less compared to coarse-grained operand isolation.

These results shows that depending on the architecture, the flexibility to shift between different operand isolation techniques plays a major role to reap the optimum benefit. Using the high-level architectural information, it is possible to achieve a strong reduction in power and to shift between the coarse-grained power optimization method (as found in system-level designs) or a fine-grained power optimization method (as done in RTL abstraction).

**Table 4. Overall Area-Delay results for ICORE**

| | Area (Gates) | | Delay (ns) | |
|---|---|---|---|---|
| | AND-based | OR-based | AND-based | OR-based |
| Original | 60441 | | 3.90 | |
| Coarse-grained | 66994 (+10.84%) | 66751(+10.44%) | 3.96(+1.54%) | 3.88(-0.51%) |
| ODC-based | 66260 (+9.63%) | 64981 (+7.51%) | 3.90 (0.00%) | 3.89 (-0.26%) |
| Fine-grained | 66147 (+9.44%) | 64762 (+7.15%) | 3.80 (-2.56%) | 3.94 (+1.03%) |

# 6. Summary and Future Work

Growing complexity of cutting-edge embedded processors have promoted the usage of high processor abstraction level, thus making the use of system level tooling inevitable. At the same time, dwindling power budgets for modern embedded processors have increased the significance of power optimization techniques. *Operand Isolation*, an effective power reduction technique is explored in this work from the perspective of ADL-based processor synthesis.

In our future work, we will focus on the combination of clock gating and operand isolation and study the power improvements thereof. We will also extend our framework to include high-level power models.

# 7. REFERENCES


[1] A. Fauth et al. Describing Instruction Set Processors Using nML. In *Proc. of the European Design and Test Conference (ED&TC)*, 1995.

[2] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 1999.

[3] A. Nohl et al. A universal technique for fast and flexible instruction-set architecture simulation. In *DAC '02: Proceedings of the 39th conference on Design automation*, 2002.

[4] Anthony Correale, Jr. Overview of the power minimization techniques employed in the IBM PowerPC 4xx embedded controllers. In *ISLPED '95: Proceedings of the 1995 international symposium on Low power design*, pages 75–80, New York, NY, USA, 1995. ACM Press.

[5] R. A. Bergamaschi. Behavioral Network Graph: Unifying the Domains of High-Level and Logic Synthesis. In *DAC*, 1999.

[6] C. Chen and K. Küçükçakar. An Architectural Power Optimization Case Study using High-level Synthesis. In *ICCD*, 1997.

[7] H. Li, S. Bhunia, Y. Chen, T.N. Vijaykumar and K. Roy. Deterministic Clock Gating for Microprocessor Power Reduction. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA)*.

[8] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.

[9] M. Münch, B. Wurth, R. Mehra, J. Sproch and N. Wehn. Automating RT-level operand isolation to minimize power consumption in datapaths. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, 2000.

[10] N. Banerjee, A. Raychowdhury, S. Bhunia, H. Mahmoodi and Kaushik Roy. Novel Low-Overhead Operand Isolation Techniques for Low-Power Datapath Synthesis . In *IEEE International Conference on Computer Design (ICCD)*, San Jose, California, USA, October 2005.

[11] P. Babighian, L. Benini and E. Macii. A Scalable ODC-Based Algorithm for RTL Insertion of Gated Clocks. In *Proceedings of the conference on Design, automation and test in Europe*, 2004.

[12] P. Mishra, A. Kejariwal and N. Dutt. Synthesis-driven Exploration of Pipelined Embedded Processors. In *Int. Conf. on VLSI Design*, 2004.

[13] Schliebusch, O., Chattopadhyay, A., Witte, E.M., Kammler, D., Ascheid, G., Leupers, R. and H. Meyr. *Optimization Techniques for ADL-driven RTL Processor Synthesis*. Montreal, Canada, June 2005.

[14] Synopsys. *PrimePower http://www.synopsys.com/products/power/primepower_ds.pdf*.

[15] T. Gloekler, S. Bitterlich and H. Meyr. ICORE: A Low-Power Application Specific Instruction Set Processor for DVB-T Acquisition and Tracking. In *Proc. of the ASIC/SOC conference*, Sep. 2000.

[16] Target Compiler Technologies. *http://www.retarget.com*.

[17] V. Tiwari, S. Malik and P. Ashar. Guarded Evaluation: Pushing Power Management to Logic Synthesis/Design. In *International Symposium on Low Power Design*, pages 221–226, 1995.