Data Path Allocation using an Extended Binding Model*

Ganesh Krishnamoorthy

Mentor Graphics Corporation
Warren, NJ 07059

John A. Nestor

Illinois Institute of Technology
Chicago, Illinois 60616

Abstract

Existing approaches to data path allocation in high-level synthesis use a binding model in which values are assigned to the same register for their entire lifetimes. This paper describes an extended binding model in which segments of a value's lifetime may reside in different registers if there is a cost advantage in doing so. In addition, the model supports multiple copies of values and the use of functional units to "pass through" unmodified values to reduce interconnect. This model is exploited in an allocation tool that uses iterative improvement to search for low-cost designs. Results show that allocation costs can be substantially reduced using this model.

1. Introduction

Data path allocation [1] is the problem of assigning hardware to a scheduled control/data flow graph (CDFG) to implement a specified behavior while meeting performance and timing constraints and minimizing implementation cost. The CDFG specifies operators that manipulate data, data values that require storage, and data transfers that move information between operators and values.

The traditional approach to allocation of a scheduled CDFG is based on a binding model that is described many places in the literature (e.g. [2,3,4]). As shown in Figure 1, the binding model specifies the following assignments between CDFG elements and datapath hardware:

1. Operators must be assigned to functional units during the control steps in which they are scheduled. Operators which are scheduled in the same control step may not be assigned to the same functional unit.

2. Values must be assigned to registers that store the values. Each value must be assigned to a register for the duration of its lifetime. Values with overlapping lifetimes may not be assigned to the same register.

3. Data transfers must be assigned to some path of connections, buses, and multiplexers so that data may flow between allocated operators in functional units and allocated values in registers as specified by the edges of the CDFG.

Figure 1 - The Traditional Binding Model

The form of data transfer allocation depends on the connection style used. In a general interconnection style [4,5], any interconnection structure of wires, buses, and multiplexers may be used. In a bus-oriented interconnection style [6], a restricted interconnection style is used in which module (register and functional unit) outputs drive one or more buses, which are connected by a single level of multiplexers to module inputs. In a point-to-point interconnection style [2], module outputs are connected to module inputs by a single level of multiplexers. The point-to-point style has the advantage that it may be derived directly from functional unit and register allocation. For this reason, it is often used as a starting point for allocation, followed by a postprocessing phase that uses a more general interconnection style (e.g. [2,5]).

The cost of a data path allocation is usually taken to be a weighted sum of the number of functional units, registers, and interconnection elements (multiplexers, buses, and wires) used to implement the behavior of the CDFG. The goal of allocation algorithms is to minimize this cost. Since the minimum number of functional units and registers is fixed by scheduling,
much of the effort in allocation involves minimizing interconnection cost.

Several allocation algorithms and heuristics have been developed that attempt to minimize the cost of an allocation using the traditional binding model. These include constructive, greedy heuristics [7,2,4], iterative approaches such as simulated annealing [8,9,10], simulated evolution [11], and "rip-up-and-reallocate" [12], and exact approaches such as bipartite graph matching [13] and integer programming [14]. These approaches can find optimal or near-optimal allocations for this binding model and cost function.

However, the binding model itself restricts the design decisions that are available in allocation and so prevents the allocation of lower-cost configurations. It does not allow techniques commonly used by human designers, such as the storage of copies of values and the use of ALU's to "pass-through" values during data transfers. The failure to consider these decisions prevents tools from finding the high-quality designs that are demanded by users of high-level synthesis tools.

Some of these problems have been addressed in previous approaches. For example, the HAL allocator [2] uses the concept of "local storage" operators, which allow the use of multiple copies of values to reduce interconnect costs. However, these local copies are restricted to be one control step in length and are placed only at the beginning of a larger value's lifetime. Other approaches (e.g. [5]) allow functional-unit pass-throughs to be considered as part of interconnect allocation. However, these opportunities are only explored after functional units are assigned.

This paper describes an extension of the traditional binding model called the SALSA binding model. The key idea in the SALSA approach is to introduce additional flexibility by breaking the lifetime of each value into segments of one control step and to allow different segments of a value to be assigned to different registers if there is a cost advantage in doing so. This approach is similar to the ASYL [15] allocator, which also considers register assignment of value segments independently. However, ASYL stipulates that all segments of a value must be stored contiguously in a single register, so this flexibility is not fully exploited.

The SALSA binding model also supports multiple copies of values and allows functional unit pass-throughs to be allocated in a natural way. Results show that this model can find allocations that require fewer interconnections than those supported by the traditional allocation model.

2. The SALSA Binding Model

The SALSA binding model is quite similar to the traditional CDFG binding model described in Section 1 with one key departure: the addition of a new type of node to the CDFG called the slack node. Slack nodes explicitly represent slack in the data flow, control flow, and timing constraints implied by CDFG edges. Figure 2(a) shows a CDFG with added slack nodes, each marked with an "S". This explicit representation of slack can be used in scheduling [16] used to rapidly evaluate whether an operator node may be rescheduled without violating data or timing constraints. However, slack nodes on data edges also prove useful in allocation.

![Figure 2 - The SALSA Binding Model](image)

Specifically, slack nodes can be thought of as "No-op" operators that pass an input value unmodified to an output value. This breaks values which may be several control steps in length into value segments that are each one control step in length. Thus instead of considering a value as a monolithic element that must be assigned to one register for the duration of its lifetime, a value is divided into multiple segments that may be assigned to separate registers, as shown in Figure 2(c). When segments associated with the same value are assigned to different registers, then a data transfer is created that must be considered during allocation. However, the ability of values to move during their lifetime introduces a new degree of freedom that can reduce the amount of interconnection needed in an allocated datapath. For example, moving a value to a different register during a control step with few data transfers may reduce the demand for new interconnections in another control step with many data transfers.

Slack nodes can also be used to take advantage of "pass-through" operations. In a pass-through operation, an idle functional unit is used to transfer a value from its input to its output without altering the value. This allows existing connections to be combined, reducing the need for additional connections. Exploiting pass-throughs is easily accomplished in the SALSA binding model by interpreting slack nodes as bindable operators that perform a "No-Op" function. Slack nodes which represent data transfers may then be assigned to functional units that can perform this "No-Op" function, as shown in the second step in Figure 2(b). Slack nodes representing contiguous value segments that are stored in
the same register do not need to be assigned to functional units and are not considered as part of the allocation.

Pass-through operations can then be used to implement data transfers using existing connections instead of adding a new connection. For example, Figure 3 shows a partially allocated CDFG and datapath in which value segments V1.1 and V1.2 are assigned to different registers - V1.1 to register R2, and V1.2 to register R1. Thus a transfer of data is required. In the normal allocation approach, this assignment requires a multiplexer at the input of R1 to select either the output of functional unit FU1 or register R2, as shown in Figure 3(a). However, since FU1 is idle at the time of the transfer and there are connections from R2 to FU1 and FU1 to R1, the slack node (S) can be assigned to FU1 and FU1 can be used as a "pass-through" device, as shown in Figure 3(b).

It is sometimes useful to create multiple copies of a value to reduce the global demand for interconnect in an allocated design. For example, Figure 4 shows a value V1 stored in register R1 which is used by operators bound to two functional units, FU1 and FU2. If a copy of V1 is created and stored in register R2, then a connection between R1 and F2 can be eliminated at the expense of an added connection at the input of R2. This can result in a lower cost implementation when the overall number of multiplexers is reduced, as in Figure 4.

The SALSA representation supports value copies by providing a value splitting transformation that creates a copy of a value segment. Repeated applications of the value splitting transformation allows a copy to be stored over multiple control steps. In addition, since value splitting may be applied to any value segment, copies of a value may be made at any point in its lifetime. A value merging transformation is also available which reverses the effects of value splitting by combining two value segments that represent the same value.

3. Exploring Allocation Alternatives

The new flexibility introduced by the SALSA binding model is exploited using an iterative approach to allocation in which a set of moves (transformations) successively modify a complete allocation to produce a new allocation. This approach is similar to earlier simulated annealing approaches (e.g. [8]), but differs in that both functional unit and register allocation are considered simultaneously and additional moves are defined that are unique to the SALSA representation. Two sets of moves were developed: moves that alter functional unit (FU) bindings, and moves that alter register bindings. Table 1 summarizes these moves.

<table>
<thead>
<tr>
<th>Move</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>FU Exchange: Exchange binding of 2 FUs</td>
</tr>
<tr>
<td>F2</td>
<td>FU Move: Reassign operator to unused FU</td>
</tr>
<tr>
<td>F3</td>
<td>Operand Reverse: Switch FU inputs</td>
</tr>
<tr>
<td>F4</td>
<td>Bind to Pass-Through: Assign slack operator/data transfer to FU</td>
</tr>
<tr>
<td>F5</td>
<td>Unbind Pass-Through: Eliminate pass-through binding</td>
</tr>
<tr>
<td>R1</td>
<td>Segment Exchange: Exchange binding of 2 value segments</td>
</tr>
<tr>
<td>R2</td>
<td>Segment Move: Reassign value segment to unused register</td>
</tr>
<tr>
<td>R3</td>
<td>Value Exchange: Exchange bindings of two selected values</td>
</tr>
<tr>
<td>R4</td>
<td>Value move: Assign all segments of a value to unused register</td>
</tr>
<tr>
<td>R5</td>
<td>Value Split: Copy of a value segment</td>
</tr>
<tr>
<td>R6</td>
<td>Value Merge: Eliminate copy of value segment</td>
</tr>
</tbody>
</table>

Table 1 - Allocation Moves
Functional unit moves alter the assignment of operators to functional units. Moves F1, F2, and F3 are similar to moves used previously in simulated annealing [8]. Moves F4 and F5 are new moves that create and remove pass-through bindings. Initially, moves that alter operator scheduling were also included in this set as in [8]. However, in our experience these moves did not lead to better allocations and so were omitted.

Register moves alter the assignment of value segments to registers. While similar moves have been used in previous approaches, the use of value segments means that moves R1 and R2 are local to a single control step. Moves R3 and R4 reassign all segments of a value to the same register. Moves R5 and R6 implement the value split and merge transformations. The combination of segment-oriented and value-oriented moves allows the exploration of tradeoffs between contiguous and non-contiguous register bindings.

4. Allocation Implementation

Allocation is implemented in a two step approach. First, a simple constructive algorithm is used to create an initial allocation. Second, an iterative improvement phase applies the moves described in the previous section to explore the design tradeoffs made possible by the SALSA representation.

The initial allocation is generated using a simple procedure that attempts to find a good starting point for allocation improvement. In this procedure, operators are assigned to functional units in each control step on a first-available basis. Values that represent loop inputs and outputs are then assigned to registers so that consistency is maintained across iterations. Next, values that occur in control steps with maximum register demand are assigned to arbitrary registers. Finally, any remaining values are bound to registers in a way that attempts to avoid adding more interconnections. Segments of each value are initially assigned to the same register unless there is no contiguous space available in a single register. In this case, a value is split into segments that fit in available registers.

Allocation improvement is implemented by applying the move set described in Section 3 and evaluating moves using a cost function that is a weighted sum of functional unit, register, and interconnect costs. Interconnect costs are computing assuming a point-to-point model since this can be quickly computed from functional unit and register allocation. This is important in an iterative approach where costs are recalculated after every move. After allocation improvement, multiplexers are combined using a simple merging procedure.

It was originally thought that allocation improvement would be implemented using simulated annealing. However, attempts to use annealing produced poor results and seldom converged on a good solution. An iterative improvement scheme was developed instead that produced better results for this application. In this approach, several trials are attempted that are analogous to temperature levels in annealing.

Several moves are attempted in each trial. Moves are selected by randomly picking a move type and then randomly picking the CDFG and datapath elements required by the particular move type. The random selection process is weighted to pick complex moves such as value move and value interchange less often than to control execution times. After the move is selected and applied, the effect on the cost function is calculated. Moves that decrease the cost function (i.e. "downhill" moves) are always accepted and retained. Moves that increase the cost function (i.e. "uphill" moves) are sometimes accepted (as described below) or else rejected by reversing the move application.

A fixed number of uphill moves are accepted in each trial. After this number is reached, only downhill moves are accepted. The idea behind this strategy is to allow a random search of the configuration space of possible allocations. Uphill moves at the beginning of the trial allow the search to move to a different region of the configuration space. Once the new neighborhood is reached, downhill-only moves search for a local optimum. During the search, the best allocation found so far is recorded. Iterative improvement terminates after no improvement is found in three successive trials or a maximum number of trials is reached. The best allocation found during the search is then returned. While no guarantee can be made that iterative improvement will converge on a high-quality solution, experience has shown that this is generally the case.

After allocation improvement, the number of multiplexers can be reduced by merging together compatible multiplexers. This is done using a simple heuristic in which an arbitrary multiplexer is selected and combined with as many other compatible multiplexers as possible. Then, another multiplexer is selected and merged with other compatible multiplexers. This process continues until merging has been attempted with all multiplexers.

5. Results

The SALSA binding model and allocator have been implemented in C and work with the existing SALSA scheduler [16]. This implementation has been tested with a number of examples.

Table 2 summarizes the results of an experiment allocating the Elliptic Wave Filter (EWF) benchmark [2,17] under a wide variety of conditions. Assumptions about operator delay are equivalent to those in [2]: adders require one control step, multipliers require two control
steps, and pipelined multipliers have a latency of one control step. Allocation CPU times for these examples ranged between 8 and 10 CPU minutes on a Sun Sparcstation 1. However, due to the random nature of the iterative improvement scheme, multiple trials are sometimes necessary to find the best result, increasing the actual CPU time required.

In this experiment, schedules were generated for 17 and 19 control steps assuming both non-pipelined and pipelined (marked with "P") multiplier units, and also for 21 control steps assuming only non-pipelined multipliers. In each case, the schedule fixes the minimum number of functional units and registers. Allocations were generated for these schedules using the minimum number of registers, and also with additional registers allowed to trade off storage vs. interconnect. Adder units in these allocations were also used to implement pass-through operations. Table 2 shows these values and the number of equivalent 2-1 multiplexers in the resulting allocations.

For comparison, the best number of 2-1 multiplexers reported by other researchers is presented in the last column. In 5 of the 14 cases, the SALSA allocator finds an allocation with fewer multiplexers than the best result previously reported. In 7 other cases, the result is equal to the best reported results, while in 2 cases the SALSA allocator required 1 more multiplexer than the best reported case. The advantage of the SALSA allocator over other approaches is particularly dramatic for the 17 step schedule with 10 registers for both pipelined and non-pipelined multipliers.

A larger example was also used to demonstrate the effectiveness of the approach with more complex designs. The Discrete Cosine Transform (DCT) [18] is used in image coding and compression applications. The specific implementation of the DCT that we used is drawn from [19] and is shown in CDFG form in Figure 5. This CDFG consists of 25 addition operators, 7 subtraction operators and 16 multiply operators and provides a challenging problem for both scheduling and allocation. This example was scheduled and allocated using hardware assumptions identical to those for the EWF example. As in the EWF example, constants for multiplication were not considered to contribute to the cost of the allocation. Table 3 summarizes schedule and allocation results for four different schedules. Execution times for the DCT example ranged from 10-17 CPU minutes on a Sun Sparcstation 1. Again, multiple trials are sometimes necessary to find the best results.

<table>
<thead>
<tr>
<th>Steps</th>
<th>FU *</th>
<th>FU +</th>
<th>Reg</th>
<th>Mux 2-1</th>
<th>Mux Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>2P</td>
<td>3</td>
<td>10</td>
<td>18</td>
<td>26 [15]</td>
</tr>
<tr>
<td>17</td>
<td>2P</td>
<td>3</td>
<td>11</td>
<td>17</td>
<td>16 [11]</td>
</tr>
<tr>
<td>17</td>
<td>2P</td>
<td>3</td>
<td>12</td>
<td>16</td>
<td>16 [2,15]</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>21</td>
<td>29 [15]</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>3</td>
<td>11</td>
<td>18</td>
<td>18 [15]</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>3</td>
<td>12</td>
<td>17</td>
<td>17 [15]</td>
</tr>
<tr>
<td>19</td>
<td>1P</td>
<td>2</td>
<td>10</td>
<td>19</td>
<td>20 [15]</td>
</tr>
<tr>
<td>19</td>
<td>1P</td>
<td>2</td>
<td>11</td>
<td>16</td>
<td>16 [5]</td>
</tr>
<tr>
<td>19</td>
<td>1P</td>
<td>2</td>
<td>12</td>
<td>18</td>
<td>19 [15]</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>20</td>
<td>20 [5]</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>2</td>
<td>11</td>
<td>16</td>
<td>16 [5]</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>2</td>
<td>12</td>
<td>17</td>
<td>20 [15]</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>15</td>
<td>15 [15]</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>2</td>
<td>11</td>
<td>16</td>
<td>16 [11, 15]</td>
</tr>
</tbody>
</table>

Table 2 - Results for EWF Example

<table>
<thead>
<tr>
<th>Steps</th>
<th>FU *</th>
<th>FU +/-</th>
<th>Reg</th>
<th>Mux 2-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
<td>6</td>
<td>12</td>
<td>29</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>4</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>3</td>
<td>14</td>
<td>29</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>2</td>
<td>14</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 3 - Results for DCT Example

7. Conclusions
This paper described an extended allocation model that allows new flexibility through the use of value segments, value copies, and functional unit pass-throughs. An iterative improvement scheme was used to search for allocations that take advantage of this flexibility. Results show that very good allocations can be found using this approach even though a relatively simple point-to-point interconnect model is used. The good solutions found by this approach suggest that the binding model may also be useful in other approaches to allocation.

Future work is possible in a number of areas. First, extensions to interconnection allocation should be investigated to improve on the point-to-point model currently use. Second, the iterative improvement scheme used in allocation improvement could be replaced by a more powerful approach that takes advantage of the added flexibility of the binding model. Finally, further
extensions to the binding model should be considered which more accurately model the actual layout.

Acknowledgements
The authors would like to thank R. Cloutier, M. McFarland, and K. Vissers for useful discussions regarding allocation, and J. van Meerbergen for suggesting the DCT example.

Figure 5 - Discrete Cosine Transform CDFG

References