Custom Floating-Point Unit Generation for Embedded Systems

Yee Jern Chong, *Student Member, IEEE*, and Sri Parameswaran, *Member, IEEE*

***Abstract*—While application-specific instruction-set processors (ASIPs) have allowed designers to create processors with custom instructions to target specific applications, floating-point (FP) units (FPUs) are still instantiated as noncustomizable general-purpose units, which, if underutilized, wastes area and performance. Therefore, there is a need for custom FPUs for embedded systems. To create a custom FPU, the subset of FP instructions that should be implemented in hardware has to be determined. Implementing more instructions in hardware reduces the cycle count of the application but may lead to increased latency if the critical delay of the FPU increases. Therefore, a balance between the hardware-implemented and the software-emulated instructions, which produces the best performance, must be found. In order to find this balance, a rapid design space exploration was performed to explore the tradeoffs between the area and the performance. In order to reduce the area of the custom FPU, it is desirable to merge the datapaths for each of the FP operations so that redundant hardware is minimized. However, FP datapaths are complex and contain components with varying bit widths; hence, sharing components of different bit widths is necessary. This introduces the problem of bit alignment, which involves determining how smaller resources should be aligned within larger resources when merged. A novel algorithm for solving the bit-alignment problem during datapath merging was developed. Our results show that adding more FP hardware does not necessarily equate to lower runtime if the delays associated with the additional hardware overcomes the cycle count reductions. We found that, with the Mediabench applications, datapath merging with bit alignment reduced area by an average of 22.5%, compared with an average of 14.1% without bit alignment. With the Standard Performance Evaluation Corporation (SPEC) CPU2000 FP (CFP2000) applications, datapath merging with bit alignment reduced area by an average of 7.6%, compared with an average of 3.9% without bit alignment. The less pronounced improvement with the SPEC CFP2000 benchmarks occurs because the SPEC CFP2000 applications predominantly use double-precision operations only. Therefore, there are fewer resources with different bit widths, which benefit less from bit alignment.**

***Index Terms*—Bit alignment, floating-point (FP) arithmetic, merging, resource sharing.**

I. Introduction

THE RACE to improve productivity and lifestyle has led to the proliferation of embedded microprocessors into many common everyday devices. The increasing performance demands, while still satisfying cost, area, and power constraints, have led to an interest in application-specific processor customizations. Particularly demanding are portable devices, which are becoming increasingly popular, continuously reducing in size, and burgeoning in functionality. The demand for such portable devices presents the challenge of creating ever more powerful devices, while keeping within tight constraints.

Floating-point (FP) operations are crucial for many scientific applications, such as processing experimental data, mathematical computations, and physical simulations, as well as multimedia applications, such as audio, video, and graphics processing. While FP instructions can be emulated in software, the emulated performance leaves a lot to be desired. Therefore, dedicated FP hardware is highly sought after for FP intensive applications. In FP intensive applications, emulated FP operations can consume over 90% of the application's total clock cycles [based on Mediabench and Standard Performance Evaluation Corporation (SPEC) CFP2000], which is unacceptable in most situations. However, high-performance FP units (FPUs) are large and complex and, therefore, costly and power hungry. They are also time consuming to design.

Application-specific instruction-set processors (ASIPs) have recently burst onto the system-on-a-chip design stage as an alternative for designers who only used ASICs and general-purpose processors. A number of vendors, such as Tensilica [1], ARC [2], ASIP Solutions [3], etc., have all produced systems which are capable of creating processors with custom instructions.

However, to support FP operations, a noncustomizable general-purpose FPU is often instantiated, e.g., in Tensilica [1]. This is often less than ideal, particularly in the embedded systems domain, where specific applications are executed and not all FP instructions may be necessary for the application, therefore resulting in redundant hardware occupying valuable real estate. To optimize the size and performance of an FPU, it is desirable to customize the FPU for the specific application that is to be executed on it. This is possible for most embedded systems because they execute a single application or a class of applications which are well-known *a priori*.

Following the ASIP philosophy, the operations supported by the FPU can be reduced to the minimum needed to execute the desired application. This reduces redundant resources and results in area and power savings.

In creating a custom FPU, we implement a subset of the FP instructions necessary for the application in the hardware FPU, and the FP instructions that are not implemented in the hardware FPU are emulated in software. Therefore, as FP

Manuscript received August 1, 2008; revised November 3, 2008. Current version published April 22, 2009. This paper was recommended by Associate Editor P. Eles.

The authors are with the School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia (e-mail: yeejernc@cse.unsw.edu.au; sridevan@cse.unsw.edu.au).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2009.2013999

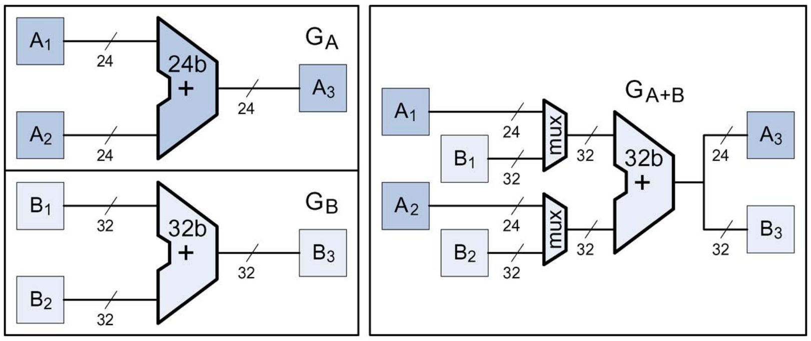

Fig. 1. Merging of 24- and 32-b adders.

instructions are off loaded onto the dedicated FP hardware, there are several tradeoffs to consider. In general, the more operations that are implemented in hardware, the greater the area consumed but the lower the cycle count needed to complete the execution of the application. However, more hardware could cause the clock period to increase. This is particularly true in the case of datapath merging, where multiplexors added to the critical path can increase delay. Therefore, in this paper, we use a rapid design space exploration to investigate these tradeoffs in our FPU-generation methodology.

To further reduce cost, area, and power, it is advantageous for the FPU to have as few hardware blocks and interconnections as possible. Each operation often has its own discrete datapath. Area can be reduced by sharing resources between datapaths to reduce replication of similar resources. This reuses as much of the hardware and interconnects between the datapaths of different operations as possible. This resource sharing creates a shared datapath for all of the operations and reduces the amount of redundant resources.

There are various datapath-merging techniques that have been proposed. However, resource sharing has traditionally been restricted to components and interconnects of identical bit widths. While this provides some area savings, it does not allow all available sharing to be exploited. Sharing of components consisting of differing bit widths is an important point in merging FP datapaths, since their complex datapaths contain components of varying bit widths.

To maximize the utilization of resources, sharing components with different bit widths is necessary. For example, given the datapaths of two different operations, one containing a 24-b adder and the other containing a 32-b one, the two adders should be replaced with a single shared 32-b adder, as shown in Fig. 1, when the datapaths are merged. However, the merging of different-sized components can result in a problem referred to as the bit-alignment problem. The bit-alignment problem involves finding how a smaller component or interconnect should be aligned within the larger component or interconnect when merged. Some types of components have to follow a strict alignment in order to operate correctly [e.g., leading-one detector (LOD), rounding unit], while others can function properly independent of alignment (e.g., bitwise logical operations, adders). The problem is complicated further when a chain of successive components and interconnects are shared and need to be aligned correctly in relation to each other.

This paper presents a methodology for the automatic generation of FPUs customized at the instruction level, with integrated resource sharing to minimize the area of the FPU.

This paper also presents a novel method for bit alignment during the resource-sharing process to allow the merging of nonmatching bit widths, thus maximizing area reduction and opening up resource sharing to more complex applications.

Thus, this paper attempts to achieve the following. Given an application that requires the set of FP operations $O_v = \{O_1, O_2, \ldots, O_n\}$, determine the subset $O_h \subseteq O_v$ that should be implemented in a hardware FPU such that the runtime of the application is minimized (assuming that the remaining operations $O_s = O_v - O_h$ are emulated in software), while using datapath merging and bit alignment to minimize the area of the FPU by merging the datapaths necessary to execute operations in O_h. The subproblem of datapath merging with bit alignment is as follows. Given a set of datapaths implemented in hardware $G_h = \{G_1, G_2, \ldots, G_n\}$, which correspond to the set of operations $O_h = \{O_1, O_2, \ldots, O_n\}$, merge components and connections between datapaths in G_h to form a single shared datapath $\overline{G}$ that can perform all operations in O_h such that the area of $\overline{G}$ is minimal.

Note that we use a rapid design space exploration to explore how the selection of the subset O_h affects runtime and area so that the subset O_h that meets any given area–runtime constraints can be selected. In this paper, we select the configuration with the lowest runtime.

The rest of this paper is organized as follows. Section II discusses previous research in the area of FPU generation, datapath merging, and bit alignment. Section III provides an overview of our FPU-generation methodology. Section IV details the datapath-merging algorithm used in the FPU-generation scheme. Section V explains the concepts and background behind bit alignment before the detailed explanation of the bit-alignment algorithm in Section VI. Section VII describes our rapid design space exploration methodology. The experimental setup is described in Section VIII, and the results are discussed in Section IX. The final section gives the conclusion.

II. Related Work

Previous research into FPU generation and customization has focused upon issues such as bit-width customization and choice of FP algorithm.

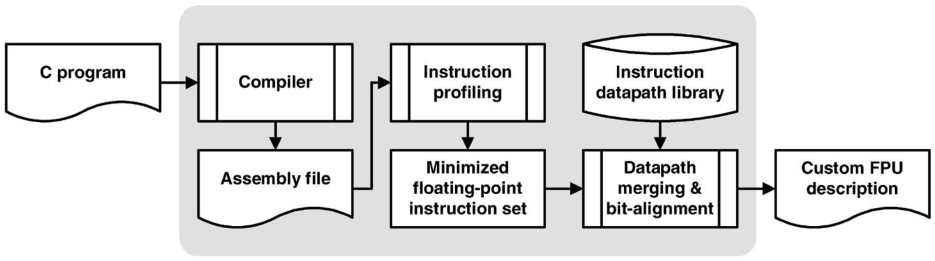
C program
Compiler
Assembly file
Instruction profiling
Minimized floating-point instruction set
Instruction datapath library
Datapath merging & bit-alignment
Custom FPU description

Fig. 2. Custom FPU-generation methodology.

Liang *et al.* [4] present an FPU-generation tool for field-programmable gate arrays, which chooses an appropriate implementation algorithm and architecture based on user-specified requirements. They focus only on choosing the most appropriate FP addition implementation and omit other operations.

There have been various works that customize FP designs by customizing the representation (bit width) of the FP operations based on user-specified accuracy requirements [5]–[9]. They minimize the bit width while keeping the loss of precision to an acceptable level. The downside is that their system produces a nonstandard FP format with arbitrary mantissa and exponent sizes. Nonstandard formats are less desirable, in general, because software is developed and verified on IEEE754 compliant machines, making the system more difficult to validate. Reducing the bit width of the FP format also reduces the accuracy and/or dynamic range.

Baidas *et al.* [10] synthesize FP modules from high-level behavioral descriptions. The technique of Baidas *et al.* selects between three techniques to implement each function – table lookup, iterative series, or CORDIC algorithm – depending on the specified parameters and constraints.

Instead of varying the implementation of FP operations as in [4] or by customizing the FP format as in [5]–[9], our FP-generation methodology customizes the FPU by selecting the necessary FP operations to be implemented and merging the hardware datapaths of these operations to reduce area. Our technique operates at the hardware datapath level, while that of Baidas *et al.* operates at a high-level functional description level.

Central to our FPU-generation methodology is the resource-sharing algorithm for merging the hardware datapaths of each instruction. While some degree of resource sharing in FPUs is common practice, such as sharing the rounding unit or the large multiplier between multiple instructions, these sharings have only been explored manually by FPU designers. As the size of the design increases, it becomes more difficult to manually find the best sharing candidates.

Resource sharing is an important problem in high-level synthesis (HLS) and register transfer level synthesis. Various solutions to the resource-sharing problem have been proposed [11]–[17]. In addition, datapath-merging techniques have been used to reduce the reconfiguration overhead in reconfigurable architectures [18] and to merge customized instructions into a single datapath [19].

The approaches of Moreano *et al.* [18] and Brisk *et al.* [19] exclude support for certain features that are necessary for merging very complex architectures (such as FP architectures). These features that need to be supported include bit vectors, multiple output ports, merging of different bit-width components, and alignment. Our work is based on the technique presented by Moreano *et al.* [18] but adapted to support the required features.

Unfortunately, previous merging techniques neglect to consider the bit-alignment problem. They generally assume that the components in the datapaths to be merged are of uniform bit width. In many cases (e.g., in HLS), this would be a fair assumption; however, in more complex architectures, such as FP designs, such an assumption would be very restrictive in terms of the hardware that could be shared.

Schoofs *et al.* [20] investigated the bit-alignment problem in multiplexed digital-signal-processor (DSP) architectures. In DSP systems, it is often necessary to execute data words with different bit widths on the same execution units; therefore, it is necessary to align the data words entering an execution unit. To overcome this problem, Schoofs *et al.* placed routers at the input ports of execution units, where each router contained several different interconnection patterns connected to a multiplexor. They presented an algorithm to determine the interconnection patterns required in each of the routers. They consider bit alignment as a postallocation task in HLS. This paper differs from [20] in that we consider bit alignment during the merging of complex hardware datapaths. We aim to determine a configuration that is properly aligned with minimal realignment multiplexors while reducing area.

III. Methodology

The methodology for the FPU generation is shown in Fig. 2. The input is the source code for the application that is to be executed on the processor. It is compiled into an assembly file, which is profiled using an instruction profiling tool to determine the subset of the FP instruction set that is required to execute the program.

The subset of instructions that the FPU needs to support is then passed to the instruction-merging stage. In this stage, the datapath descriptions (in the form of netlists) for each instruction is obtained from a library of predesigned datapaths and passed through the datapath-merging and bit-alignment stage. The instruction-merging algorithm merges the datapaths of each instruction into a single shared datapath and outputs a new datapath description for the custom FPU. The custom FPU description can then be used to generate very high speed integrated circuit hardware description language (VHDL) and then synthesized.

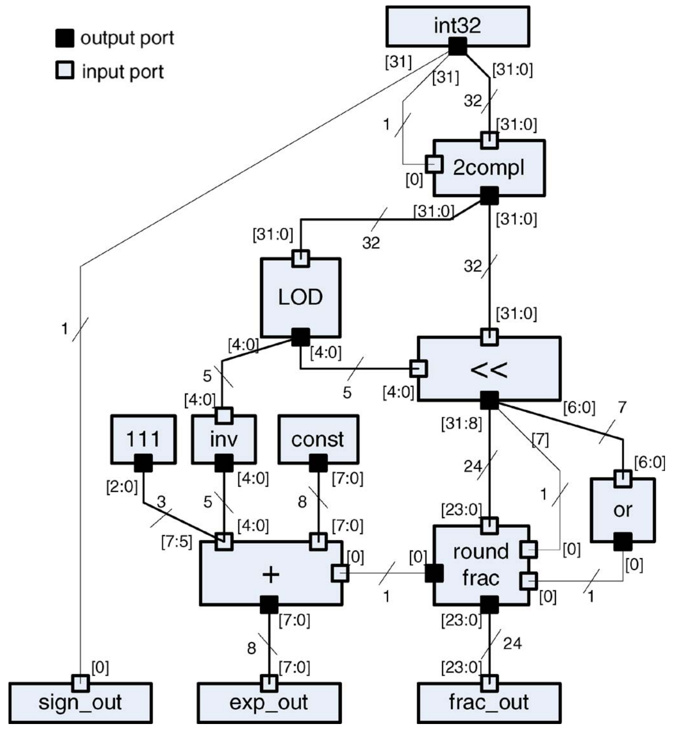

Fig. 3. Structure of datapath to convert 32-b integer to single-precision FP format.

IV. Datapath-Merging Algorithm

The technique used for merging the instruction datapaths is based on the maximum weight clique approach presented in [18]. However, rather than merging simple graphs, the technique from [18] was modified to support the merging of more detailed structures with multiple output ports with different bit widths, bit vectors, and alignment data. A datapath structure would look similar to Fig. 3, which shows the structure of a datapath for the *convert integer to single-precision FP* operation. The merging algorithm was also modified to handle the merging of different bit-width components and bit alignments.

The netlists that describe the structures contain both components and connections. Components are characterized by:

1) function of the component;
2) bit width;
3) number of input ports and their bit widths;
4) number of output ports and their bit widths;
5) area of the component;
6) alignment information.

Connections in the netlist are characterized by:

1) bit width;
2) source component;
3) source port number;
4) destination component;
5) destination port number;
6) bit-vector range of source port;
7) bit-vector range of destination port.

A component can represent a functional unit, source (e.g., input operand, constant), or sink (e.g., output result). The area of each component is estimated using synthesis tools (in our case, Synopsys Design Compiler). A connection represents an interconnect between a range of bits on a component's port with a range of bits of another component's port.

A netlist describing the hardware datapaths required to execute each FP instruction is created and added into a library. The inputs to the merging algorithm are the netlists for the instructions that are needed for the application. These are selected from the predesigned library of netlists.

The merging of two datapaths is shown in Fig. 4, where the dotted lines represent possible hardware mappings and the thick interconnects represent possible interconnect mappings. As shown in Fig. 4, Datapath 1 represents a simplified version of Fig. 3, and Datapath 2 represents a simplified version of the datapath for the fractional part of an FP addition. The merged datapath in Fig. 4 shows how the two datapaths could be merged. Some components may have more than one potential mapping, such as A_2 with either B_4 or B_7 and A_4 with B_5 or B_9. In this example, A_2 and A_4 should be merged with B_7 and B_9 to allow more interconnects to be shared. The merged datapath is more compact than the two discrete datapaths, with a multiplexor to select the operation to be executed.

For simplicity, the netlists for the datapaths to be merged can be modeled by data-flow graphs (DFGs). A DFG is a directed graph $G = (V, E)$, where a vertex $v \in V$ represents a component and an edge $e \in E$ represents an interconnect between two vertices. Each vertex v has:

1) a set of input ports $P_{in} = \{p_1, \ldots, p_n\}$;
2) a set of output ports $Q_{out} = \{q_1, \ldots, q_n\}$;
3) attributes specifying its type/function, bit width, area, bit width of each port, and alignment info.

An edge $e = (u, q_u, v, p_v) \in E$ represents an interconnect from the output port q_u of vertex u to the input port p_v of vertex v.

Instruction datapaths to be merged are represented by DFGs G_i for $i = 1, \ldots, n$. Each of the graphs G_i are iteratively merged, two at a time, into a shared datapath represented by $\overline{G}$.

The instruction-merging algorithm is shown in Fig. 5 and described in the following sections. Fig. 5(a) shows the two graphs to be merged; Fig. 5(b) shows the hardware and interconnect mappings, where the dotted lines indicate which components and interconnects can be shared. Fig. 5(c) shows the compatibility graph, and Fig. 5(d) shows the resulting merged graph. Each of these steps are explained in detail, as follows.

A. Hardware and Interconnect Mapping

The first step of the merging algorithm is to find all possible mappings between the two graphs to be merged, for example, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. The example in Fig. 5(b) shows (dotted lines) possible mappings between two graphs. A vertex $v_i \in V_1$ can be merged with a vertex $v_j \in V_2$ into a mapping v_i/v_j if they are of the same type (e.g., both are adders) or if they are of compatible types (e.g., adder and subtractor can be replaced with a combined adder/subtractor). If v_i and v_j do not have identical bit widths, the mapping v_i/v_j will take on the wider bit width when merged, i.e., $bit\_width(v_i/v_j) = \max\{bit\_width(v_i), bit\_width(v_j)\}$. The estimated area saved by a mapping is calculated based on the area of each vertex to be merged. A mapping of two vertices results in an area saving equal to the combined area of the two

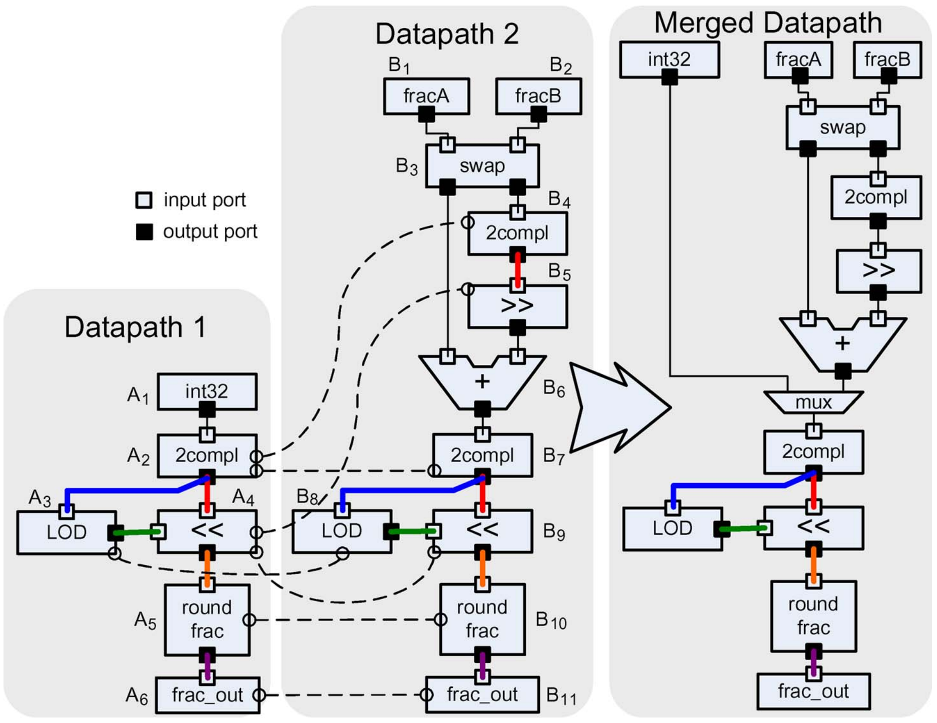

Fig. 4. Datapath-merging example.

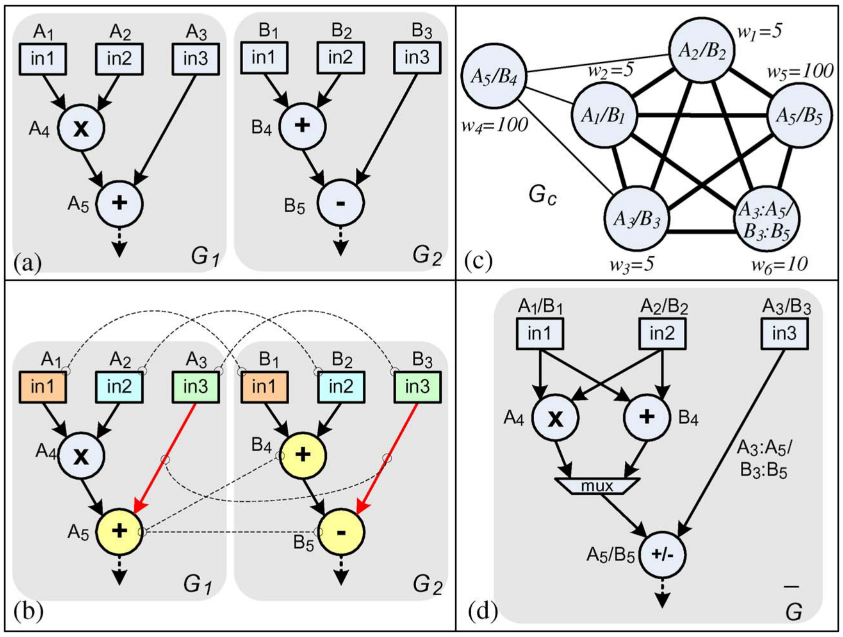

Fig. 5. DFG merging process. (a) DFGs G_1, G_2. (b) Hardware and interconnect mapping. (c) Compatibility graph and maximum weight clique solution. (d) Merged DFG.

vertices minus the area of the resulting combined component $Area\_saved(v_i/v_j) = Area(v_i) + Area(v_j) - Area(v_i/v_j)$.

Two edges $e_i = (u_i, q_i, v_i, p_i) \in E_i$ and $e_j = (u_j, q_j, v_j, p_j) \in E_j$ can be mapped if they satisfy these conditions:

1) source vertex u_i can be mapped to source vertex u_j;
2) destination vertex v_i can be mapped to destination vertex v_j;
3) source port q_i matches source port q_j;
4) destination port p_i matches destination port p_j.

A port matches another port if they are the same type of port. For example, if u_1 and u_2 are adders, the carry-in port of u_1 matches the carry-in port of u_2 but does not match the operand port of u_2. The area saved by mapping two connections is equal to the area of a multiplexor, which would be required if the connection is not shared $Area\_saved(e_i/e_j) = Area(mux)$.

B. Nonbeneficial Mapping Removal

The second step checks the mappings to find which of the vertex mappings require a multiplexor at one or more of its input ports. If a vertex mapping v_i/v_j has an input edge

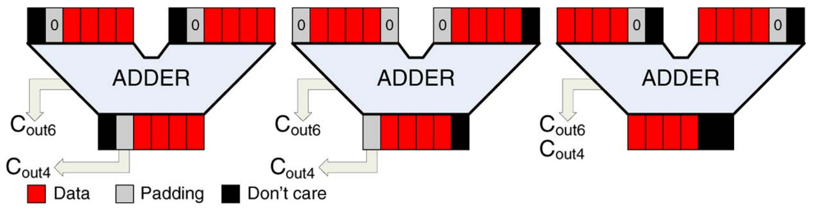

Fig. 6. Adder with different alignments. (Left to right) $ali = (\text{LSB } 0)$; $ali = (\text{LSB } 1)$; $ali = (\text{MSB } 0)$.

that cannot be shared (i.e., it has more than one arc going into an input port), it will require a multiplexor at that input port. The area savings calculated earlier are then adjusted to account for the need for multiplexors. All mappings that end up with zero or negative area savings are removed. This ensures that mappings that do not contribute to area savings due to additional multiplexors are not considered.

C. Compatibility Graph

The third step involves constructing the compatibility graph using the mappings determined in the earlier steps. The compatibility graph is an undirected weighted graph, which represents which mappings are compatible with each other. Fig. 5(c) shows an example of a compatibility graph.

Let $G_c = (V_c, E_c)$ be the compatibility graph for a pair of DFGs G_i and G_j. A vertex $v_c \in V_c$ represents either a component v_i/v_j or a connection e_i/e_j mapping. Each vertex in the compatibility graph $v_c \in G_c$ has a weight w_c that corresponds to the area reduction achieved by that mapping (see Section IV-A). An edge $e_c = (u_c, v_c) \in G_c$ between two vertices indicates that the two mappings represented by the vertices u_c and v_c are compatible. If there exists a conflict between two mappings, they are incompatible with each other, i.e., two vertices in G_c are incompatible if they map the same component to different components.

D. Maximum Weight Clique Solution

To find the set of compatible mappings that provide the greatest area reduction, the maximum weight clique for the compatibility graph is solved. The maximum weight clique of the graph $G_c = (V_c, E_c)$ is a subgraph $G_{\text{MWC}} \subseteq G_c$, where all vertices in G_{MWC} are pairwise adjacent and the total weight of all the vertices in G_{MWC} is maximum. The thick lines in Fig. 5(c) shows the maximum weight clique for the example.

Maximum weight clique determination is known to be an NP-complete problem and is solved using a heuristic polynomial-time algorithm. The Cliquer tool [21], which is based on a branch-and-bound technique, was used to solve the problem.

The resulting graph G_{MWC} is then used to reconstruct the netlist describing the new merged datapath.

V. Bit Alignment

The datapath-merging process described in Section IV would be sufficient if only components of identical bit width are allowed to be merged. However, if components with different bit widths are allowed to be shared in order to minimize area, the datapath merging needs to consider how the merged components are aligned. For example, FP arithmetic datapaths have paths for computing the exponent and mantissa parts of the FP number. For single precision, the exponent and mantissa components would mostly be 8 and 24 b, respectively. For double precision, the exponent and mantissa components would mostly be 11 and 53 b, respectively. Conversion operations to convert between integer and single- and double-precision formats would contain a mix of 8-, 24-, 11-, 53-, and 32-b components. This creates a considerable mix of different bit-width components that could possibly be shared and requires a bit-alignment technique to allow sharing between them.

Our bit-alignment technique introduces an additional processing step on the compatibility graph G_c prior to solving the maximum weight clique. Before presenting our bit-alignment algorithm in the next section, some basic definitions and concepts are described.

When two components of different bit widths are mapped to each other, we call the larger component, which would be the component that is actually synthesized, the *carrier*. The smaller component, whose function will be executed on the carrier's hardware, is called the *passenger*. One can imagine the passenger as being aligned within the carrier. Bit alignment is the process of determining how the signals for a narrower operation should pass through the larger than necessary component but can also be thought of as the process of aligning the passenger within the carrier.

We classify components as either having *flexible* or *fixed* alignments. A component that has *flexible alignment* is not dependent on the alignment of its incoming data to generate a correct result (e.g., adders, subtractors, or logical operations). A component that has *fixed alignment* requires that the incoming data are aligned in a specified way in order to operate correctly [e.g., a LOD or an FP rounding unit].

The alignment of the passenger is defined as an offset from either the MSB or LSB side. In this paper, we will use the following notation to represent the alignments: $ali = |\text{MSB/LSB } n|$ for fixed-alignment components and $ali = (\text{MSB/LSB } n)$ for flexible-alignment components or connections, where n is the offset from either the MSB or LSB side. For example, $ali = |\text{MSB } 0|$ represents a fixed alignment with zero offset from the MSB side. Similarly, $ali = (\text{LSB } 2)$ represents a 2-b offset from the LSB side assigned to a flexible-alignment component or connection. Note that connections always have flexible alignments.

Fig. 6 shows an example of different alignments possible for data passing through an adder. In this case, the carrier is 6 b wide, while the passenger is 4 b wide. As can be seen, the

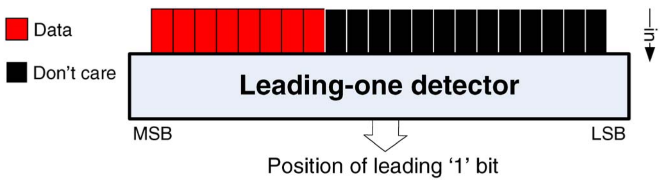

Fig. 7. LOD circuit with $ali = |\text{MSB } 0|$.

alignment of the passenger does not matter for correct operation as long as both inputs have the same offset and the result is read using the same offset as the inputs. The position of the passenger's carry out changes with alignment; thus, if the carry out is required, the carry out has to be read from the correct position. Care must be taken also not to contaminate the data with carry ins; hence, zero padding the adjacent bit on the LSB side of each input may be necessary. Likewise, if the carry out is needed, zero padding the adjacent bit on the MSB side of each input may be necessary. Padding would not be necessary for operations where there are no dependences between bits, such as in bitwise logical operations. If certain special bits are required, such as an overflow bit from an adder, internal logic may be added or modified to produce the special bits for the different alignments.

Fig. 7 shows an example of a fixed-alignment component—a LOD circuit, which outputs the position of the leading-one bit of the input word (e.g., an input of 00101111_2 produces an output of 010_2, indicating that the leading-one bit is in the second bit position, assuming that the MSB is in position zero). In this case, the output would be erroneous if the MSB of the passenger was not aligned to the MSB of the carrier. The rounding unit found in many FP operations is another example of a fixed-alignment component. The LSB of the input data is necessary to make rounding decisions; therefore, the passenger's LSB must be aligned to the LSB of the carrier.

When a component is aligned, it will affect the alignments of interconnects and other components around it. For example, given two shared components v_1/v_2 and v_3/v_4 connected to each other with a shared connection $e_{1,3}/e_{2,4}$, if v_1/v_2 is aligned with a particular offset $ali = (\text{MSB } 0)$, the shared connection $e_{1,3}/e_{2,4}$ must be aligned to $ali = (\text{MSB } 0)$ since it is attached to v_1/v_2. Subsequently, v_3/v_4 must also be aligned to $ali = (\text{MSB } 0)$ since it is attached to $e_{1,3}/e_{2,4}$, and so on. We define this chain of events as *alignment propagation*. However, it is important to note that not all components or ports should propagate an alignment. For example, the alignment of the output from the LOD does not change because the bit position output would not make sense if realigned. Similarly, the shift amount input of a shifter should not be realigned or else the amount shifted would be erroneous. Therefore, we define a port as *alignment sensitive* if it propagates an alignment.

Each shared interconnect must fit the alignment of the shared components on each end, as shown in Fig. 8(a). If a shared interconnect cannot fit with the required alignments of both components, then the interconnect cannot be shared. In this case, a multiplexor must be added to select between the carrier and the passenger data lines, as shown in Fig. 8(b).

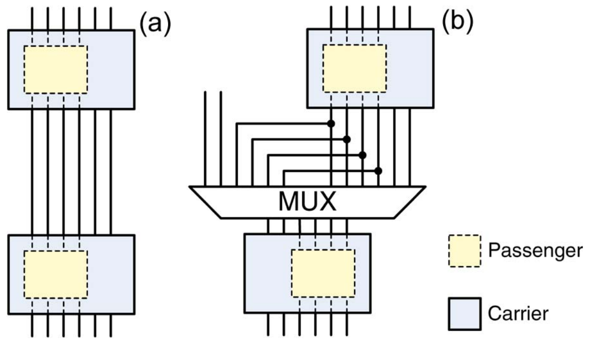

Fig. 8. (a) Shared interconnect between two shared components. (b) Interconnect cannot be shared due to incompatible alignments of shared components at each end.

VI. Bit-Alignment Algorithm

This section describes the bit-alignment algorithm, which is presented in Algorithm 1. In order to integrate bit alignment into the resource-sharing process, each component in the datapaths is tagged with the following additional information: its alignment class (flexible or fixed); input and output ports that are alignment sensitive; and its alignment offset. The bit-alignment problem occurs only with shared components and connections. Since the compatibility graph contains all the potential mappings, only the compatibility graph needs to be analyzed. The compatibility graph is traversed to determine necessary alignments. The traversal path and alignment data are stored as an alignment tree. The alignment tree is then used to prune and annotate the compatibility graph. This process is described in detail in this section.

Algorithm 1 Bit-alignment algorithm

```
$G_c = (V_c, E_c)$ is the compatibility graph.
Let $\dot{V} = \{\dot{v_1}, \ldots, \dot{v_n}\}$, where $\dot{V} \subset V_c$ is a set of vertices with fixed-alignment component mappings.
for all $\dot{v_i} \in \dot{V}$ do
    Build alignment tree $T_i$ rooted with $\dot{v_i}$:
    Propagate alignment starting at $\dot{v_i}$ throughout $G_c$.
    Add each vertex along traversal path to $T_i$ along with alignment offsets.
Combine alignment trees $\{T_1, \ldots, T_n\} \rightarrow \bar{T}$.
Apply $\bar{T}$ to $G_c \rightarrow \overline{G_c}$.
return The new compatibility graph $\overline{G_c}$.
```

Fig. 9 shows how the alignments are propagated through the compatibility graph and how an alignment tree is formed. The components with fixed alignments need to be aligned with specific alignments for correct functionality. Therefore, the bit-alignment algorithm first starts by selecting a fixed-alignment component ($\dot{v_1}$). Since the compatibility graph shows the potential mappings that can coexist, alignments only need to propagate along the edges of the compatibility graph. The algorithm searches each vertex that is connected to $\dot{v_1}$ by an edge for another vertex that it can propagate its alignment to. An aligned component can only propagate its alignment to an interconnect, and vice versa. For example, if the current aligned vertex is

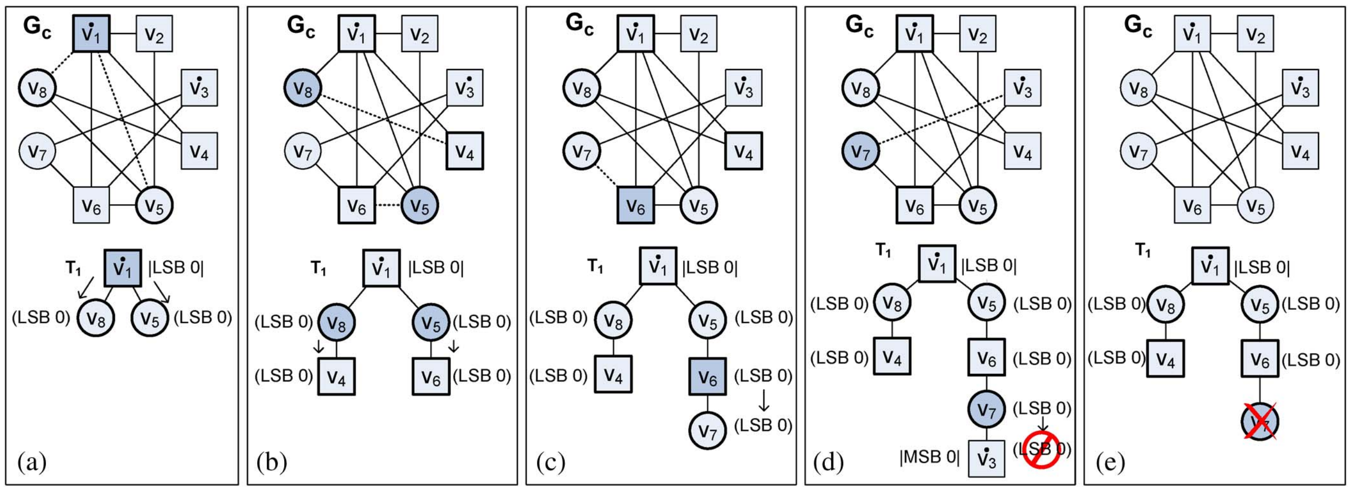

Fig. 9. Illustration of alignment propagation and alignment tree formation. (Squares) Component mappings. (Circles) Connection mappings.

a shared component, it will search for vertices with shared interconnects, and vice versa. In addition, alignments are only propagated through *alignment-sensitive* ports. For example, in Fig. 9, the component vertex $\dot{v_1}$ can propagate its alignment to interconnect vertices v_5 and v_8, which are assumed, in this example, to be attached to the alignment-sensitive ports of $\dot{v_1}$. Similarly, in the second step in Fig. 9, v_5 can only propagate its alignment to v_2 if v_5 is attached to an alignment-sensitive port of v_2. In this example, we assume that it is not; however, we assume that v_5 is attached to an alignment-sensitive port of v_6 and, therefore, v_5 can propagate its alignment to v_6.

The alignment propagation process occurs as a traversal of the compatibility graph. Each traversal path continues until one of the following occurs: alignment propagation is no longer possible; the path intersects with another path (i.e., reaches an already aligned vertex); the path reaches a fixed-alignment component; or alignment fails.

An alignment may fail if the alignment offset pushes the passenger beyond the bounds of the carrier. For example, given a mapping between a 10-b passenger and a 16-b carrier, an alignment of (LSB 7) or greater will fail. If the vertex with the failed alignment was an interconnect mapping, the mapping is marked for deletion. If the vertex with the failed alignment was a component mapping, the interconnect mapping immediately preceding it, which propagated the alignment, is marked for deletion. This is the case in Fig. 9(d), where v_7 is unable to propagate its alignment to $\dot{v_3}$ because the alignment $ali =$ (LSB 0) conflicts with the fixed alignment of $\dot{v_3}$ of $ali =$ |MSB 0|. Therefore, the shared interconnect v_7 connecting v_6 and $\dot{v_3}$ is invalid and is marked for deletion. The reason it is marked for deletion is because when alignment fails, the interconnect can no longer be shared and a multiplexor must be inserted to allow for the misaligned connections to be multiplexed into the functional unit, such as in Fig. 8. Deleting the interconnect mapping results in two unshared interconnects connected to the same input port of a component, and a multiplexor will be automatically placed at that input when the merged datapath is reconstructed.

The traversal path and alignment propagation information is stored as an alignment tree T_i, where each node in the tree corresponds to a vertex in the compatibility graph and contains the alignment information for that vertex and the root of the tree corresponds to the fixed-alignment vertex, where the alignment propagation originated from. Fig. 9 shows the formation of an alignment tree during alignment propagation. The process is repeated starting at other fixed-alignment vertices, which will create more alignment trees. Thus, in the example shown in Fig. 9, a new alignment tree would be formed starting from $\dot{v_3}$ next (not demonstrated in Fig. 9).

All of the alignment trees $T_i, \ldots, T_n$ are then combined. Alignment trees that do not overlap can be combined directly. Alignment trees that overlap can be combined directly if the overlapping leaves are not in conflict, i.e., the alignment offsets of the overlapping leaves are the same. If the overlapping leaf/leaves have conflicting alignments, one tree will have priority over the other, allowing its leaves to override the overlapping leaves. Larger trees are given priority over smaller ones. The resulting alignment tree $\bar{T}$ is then applied to the compatibility graph G_c. This is done by traversing the alignment tree and copying the alignment offsets at each node to the corresponding vertex in the compatibility graph G_c. Vertices that are marked for deletion are removed from the compatibility graph to give the bit-aligned compatibility graph $\overline{G_c}$. This is shown in Fig. 10.

The bit-aligned compatibility graph $\overline{G_c}$ is solved for the maximum weight clique as in Section IV, and the resulting G_{MWC} is then used to reconstruct the new merged datapath $\overline{G}$.

VII. Rapid Design Space Exploration

To investigate the tradeoff between the performance and the operations implemented in hardware, a rapid design space exploration was performed. Instead of exploring every single configuration in the design space, which is time consuming, we strategically select the most likely configurations.

The rapid design space exploration methodology is shown in Fig. 11. The application is compiled and profiled to determine the necessary FP instructions. Without an FPU, the processor would have to emulate all of these FP instructions in software; hence, we determine how many cycles it would require for

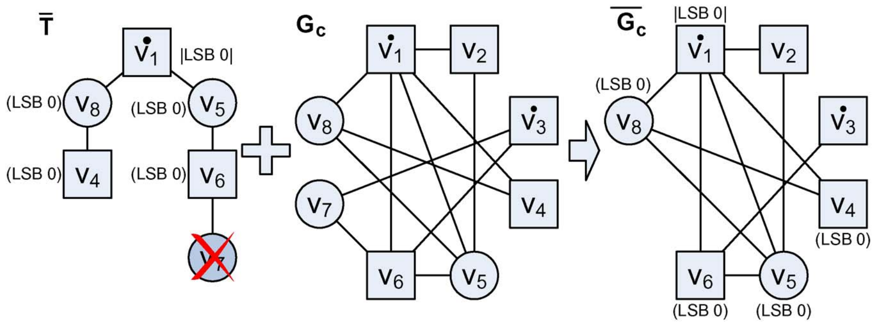

Fig. 10. Applying alignment tree to compatibility graph.

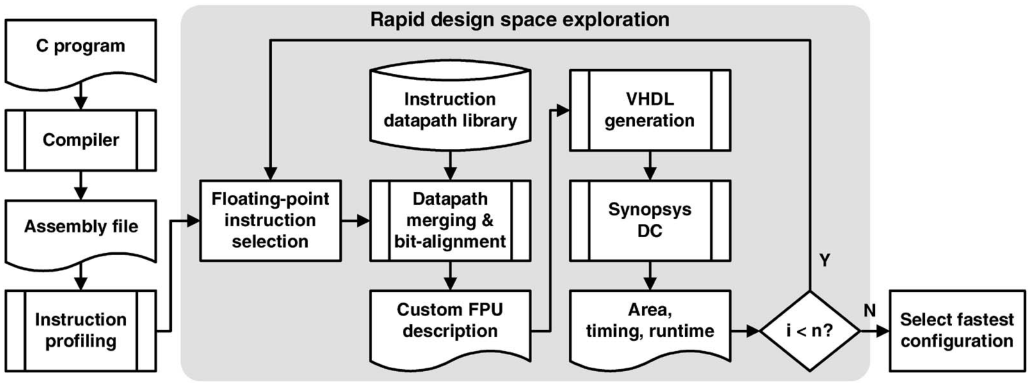
Rapid design space exploration
C program
Compiler
Assembly file
Instruction profiling
Floating-point instruction selection
Instruction datapath library
Datapath merging & bit-alignment
Custom FPU description
VHDL generation
Synopsys DC
Area, timing, runtime
i < n?
Y
N
Select fastest configuration

Fig. 11. Rapid design space exploration methodology.

the application to execute on the base processor with purely software emulation. Then, we estimate the number of cycles saved by hardware implementation of each operation. The operations are ranked by the number of cycles saved. During the rapid design space exploration, the highest ranked operation is first implemented in the custom FPU. Operations are added to the FPU, one at a time, on the order of their rank until we reach a configuration that contains all necessary FP operations. With each configuration, datapath merging with bit alignment is used to minimize area. The FPU is synthesized to obtain the area and critical path delay. If the critical path delay exceeds the minimum clock period of the base processor, the critical path delay of the FPU is used as the minimum clock period. The total runtime of the application is then determined. When the design space exploration is complete, we select the configuration that produced the fastest application runtime. The rapid design space exploration algorithm, with datapath merging and bit alignment, is presented in Algorithm 2.

The datapath-merging algorithm merges only two datapaths at a time; therefore, the merging of multiple datapaths may be influenced by the order in which the datapaths are merged. An exhaustive search of all permutations of the datapath ordering is performed, and the best solution is chosen based on the lowest total area. Since the number of permutations is $n!$, where n is the number of datapaths to be merged, the runtime of the algorithm is $O(n!)$ for large n. However, as n is generally small for FP applications (< 10) and it only needs to be run at design time, the runtime is not critical. The algorithm took 3.5 h to complete the merging and bit alignment of seven datapaths when running on a 2.2-GHz dual-core Advanced Micro Devices Opteron machine. To expand this algorithm to applications with large n, a heuristic algorithm may need to be used.

Algorithm 2 Rapid design space exploration with datapath merging and bit alignment

```
Let $\mathbf{O_h} = \{O_1, \ldots, O_n\}$ be the set of operations needed by the application.
for all $O_k \in \mathbf{O_h}$ do
    Compute cycles saved by implementing $O_k$ in hardware
Rank each $O_k \in \mathbf{O_h}$ according to cycles saved.
Let $\mathbf{R} = \{R_1, \ldots, R_n\}$ be the list of ranked operations.
Let $G_i$ be the datapath that implements the operation $R_i$.
Let $\mathbf{G_M}$ be the set of datapaths to be merged.
Let $\overline{\mathbf{G}}$ be the set of merged datapaths.
for $i = 1 \rightarrow n$ do
    Add datapath $G_i$ to $\mathbf{G_M}$
    if $i = 1$ then
        $\overline{G}_{\min} = G_1$
    else
        for all permutations of datapaths in $\mathbf{G_M}$ do
            $G_a = G_1$
            for $G_b = G_j, j = 2 \rightarrow n$ do
                Build hardware mappings between the components and the connections in $G_a$ with $G_b$.
                Remove mappings that do not contribute to area reductions.
                Construct compatibility graph $G_c$.
                Perform bit alignment on $G_c$. Returns $\overline{G_c}$.
                Find maximum weight clique for $\overline{G_c} \rightarrow G_{MWC}$
            Reconstruct merged datapath $\overline{G}$ using $G_{MWC}$.
            $G_a = \overline{G}$.
        Calculate total area of $\overline{G}$
        Save $\overline{G}$ to $\overline{\mathbf{G}}$.
    $\overline{G}_{\min} \leftarrow$ Select from $\overline{\mathbf{G}}$ the merged datapath with
    lowest total area.
  Package $\overline{G}_{\min}$ into FPU and synthesize $\rightarrow$ area and
  timing.
  Compute runtime for the application.
return Configuration with fastest runtime.
```

VIII. Experimental Setup

The Simplescalar platform [22] was chosen for our experiments. The base processor is a six-stage pipelined processor, based on the portable instruction set architecture (PISA) instruction set architecture (ISA) [23], and has a clock period of 8 ns (125 MHz). While the PISA ISA is used more in the academic community than in commercial microprocessors, the PISA ISA is a very similar to the MIPS ISA, which is a widely used ISA in embedded microprocessors. The architecture is similar to many reduced-instruction-set-computer-based architectures. The FP instructions in the PISA ISA include single and double-precision FP arithmetic instructions (add, subtract, multiply, divide, square root, absolute value, and negate); conversion instructions to convert between integer, single, and double-precision formats; comparison instructions; load and store instructions; data-transfer instructions (move); and control instructions (branch and jump). Refer to [24] for the full list of PISA instructions and detailed descriptions of each. Datapaths for each of the arithmetic and conversion instructions were designed and placed in the instruction datapath library in Fig. 11. Only the datapaths for the arithmetic (except for absolute value and negate) and conversion instructions are included for datapath merging because the other instructions are trivial to implement.

The FP datapaths were designed to be IEEE754 [25] compliant. The FP adder was based on the standard single-path design [26]. The FP multiplier used a fast and area-efficient Wallace-tree design with radix-4 modified Booth encoding [27] and a carry–select adder for the final addition. The FP divider and square root were based on iterative radix-2 SRT designs [28].

Benchmark applications from the Mediabench suite [29] and the FP benchmarks from the SPEC CPU2000 suite (SPEC CFP2000) [30] were selected and compiled using the Simplescalar [22] compiler. Simplescalar's profiler was used to profile the compiled binaries and determine the subset of FP instructions required. The datapath for each instruction was loaded from the library into the datapath-merging algorithm. The merged datapath was then synthesized using Synopsys Design Compiler [31], and area and timing information was obtained. The 0.18-μm Tower library, available from Synopsys, was used for synthesis.

The rapid design space exploration was performed for each of the several Mediabench applications (epic, unepic, mpeg2dec, mpeg2enc, cjpeg, and djpeg) and SPEC CFP2000 applications (art, equake, wupwise, and swim). The "*-dct float*" switch was used to force the JPEG applications to use the FP discrete cosine transform (DCT) instead of the integer DCT.

The generated custom FPUs are not pipelined internally and complete all operations in one cycle, except for the FP divide and square root, which take 28 and 27 cycles, respectively, for single precision and 57 and 56 cycles, respectively, for double precision. Each custom FPU is closely coupled into the execution stage of the base processor. Therefore, the minimum clock period of the processor + FPU combination is constrained by the critical path delay of the FPU if this delay exceeds the longest pipeline stage delay of the base processor.

A publicly available IEEE754 compliant FP emulation package [32] was profiled to determine the average number of cycles needed to emulate each FP operation. This was used to estimate the number of cycles, which would be saved by implementing each operation in hardware.

To evaluate the impact of datapath merging and bit alignment on the area and delay, for each of the benchmark applications, two additional FPUs were created based on the configuration with the lowest runtime.

1) One where datapath merging was performed without bit alignment (i.e., only components of identical bit widths could be shared). This FPU was generated using the same methodology as in Section III, except that the datapath merging was restricted to components of identical bit widths and that the bit-alignment step was bypassed.
2) One where no datapath merging was performed at all (i.e., each operation has a discrete datapath). This FPU contained discrete handcrafted FP datapaths for each of the FP operations required for the specific application. This FPU served as the reference FPU with which to compare against.

IX. Results and Discussion

The results of the rapid design space exploration are shown in Fig. 12. The graphs show the runtime against the area. Circles indicate each of the configurations that produced the fastest runtimes. As expected, the more operations implemented in hardware, the greater the area and the lower the runtime. Most of the graphs show diminishing returns after a certain point. This is because the clock period increases as more datapaths are merged into the FPU and the number of cycles saved using hardware (instead of software emulation) diminish as we implement lower ranked operations in hardware. In all of the cases, except for epic and mpeg2enc, the runtime eventually starts increasing slightly. In these cases, the cycles saved by using hardware cannot overcome the added delay.

The effects of the FPU's critical path delay on the minimum clock period of the overall system can be reduced by pipelining the FPUs internally, which would increase area but should result in considerably better performance. In Fig. 12, some of the benchmarks show only a moderate improvement in runtime initially, considering that the clock frequency of the system drops after adding an unpipelined double-precision FPU, offsetting some of the gains from the lower cycle count. By pipelining the FPUs, we could avoid the drop in clock frequency and expect to see faster runtimes. The effect on the graphs shown in Fig. 12

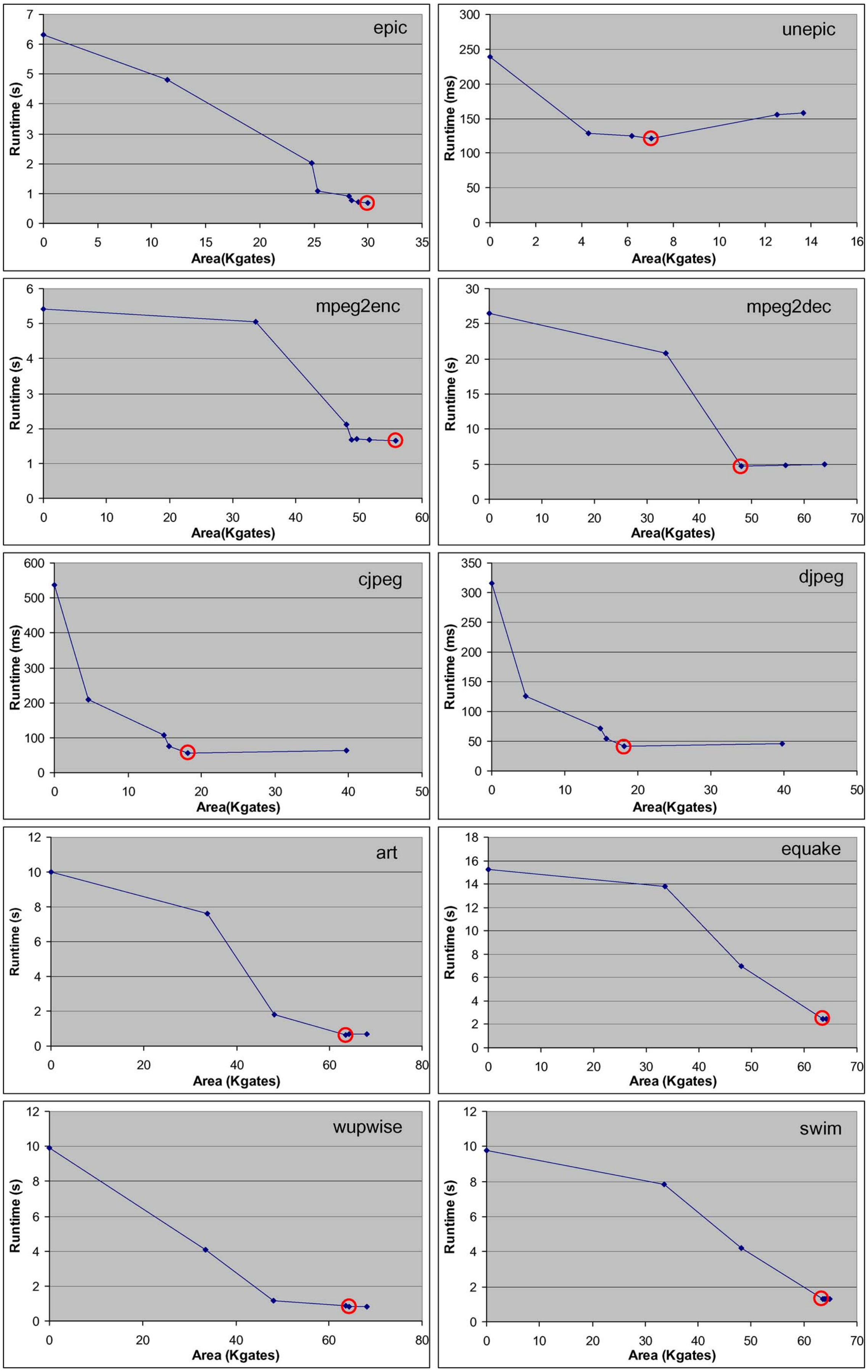

Fig. 12. Rapid design space exploration results—runtime versus area.

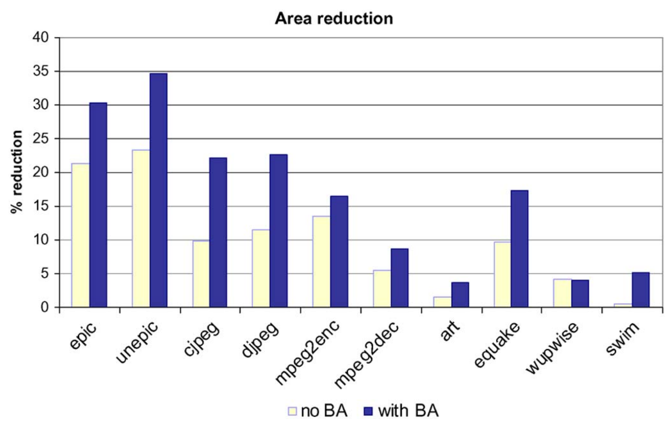

Fig. 13. Percentage area improvement for FPUs with bit alignment enabled (with BA) and disabled (no BA).

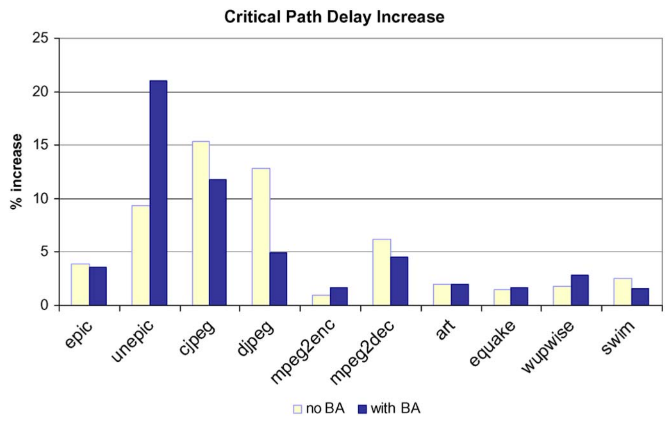

Fig. 14. Percentage increase in critical path delay for FPUs with bit alignment enabled (with BA) and disabled (no BA).

would be lower runtimes for each point, a slight increase in area for each point (due to overheads of the pipeline registers), and curves shaped more like the cjpeg and djpeg graphs, with large initial improvements before flattening off, instead of the kinked curves such as the mpeg2enc, mpeg2dec, art, equake, and swim graphs. A simple way to pipeline the FPUs would be to use the automatic pipelining and retiming features available in synthesis tools. This would reduce or eliminate the penalty on clock frequency caused by the FPU. Pipelining would also allow the FPUs to be coupled to faster base processors than the one used in our experiments.

The results of the bit-alignment evaluation are shown in Figs. 13 and 14. Fig. 13 shows the percentage reduction in area, and Fig. 14 shows the percentage increase in critical path delay caused by (light bars) *datapath merging without bit alignment* and (dark bars) *datapath merging with bit alignment*, compared with when no datapath merging is performed.

Datapath merging without bit alignment reduced area by an average of 10.1%; however, bit alignment provided even greater area reductions with an average of 16.5% compared with the FPU that was generated without any datapath merging. The Mediabench benchmarks showed significant area reductions (average of 14.1% without bit alignment versus 22.5% with bit alignment), while the SPEC CFP2000 benchmarks showed less pronounced improvement (average of 3.9% without bit alignment versus 7.6% with bit alignment). Most of the SPEC CFP2000 benchmarks used, predominantly, double-precision FP instructions. Therefore, there would be more resources with similar bit widths, resulting in only minor gains from nonidentical bit-width merging. The benchmarks that show the best results tend to have a more diverse mix of bit widths, which would be where the non-bit-aligned algorithm would struggle. The results indicate that the performance of the bit-alignment algorithm is highly dependent on the type of instructions being merged and the bit widths of the resources used by each instruction.

The results show that datapath merging increased the critical path delay in all cases, which is caused by the insertion of multiplexors into the critical path during resource sharing. This is most evident in the unepic benchmark, where the area reduction was greatest, as well as the increase in delay.

Intuitively, we should see the bit alignment increasing the critical path delay over the non-bit-aligned datapath merging due to increased resource sharing. However, the results show that, in some cases (epic, cjpeg, djpeg, mpeg2dec, and swim), the delay was lower with bit alignment. In these cases, bit alignment opened up additional resource-sharing opportunities that did not contribute to the critical path delay.

Apart from its use in merging FP datapaths, the bit-alignment technique may also be useful in other applications where components of different bit widths have to be merged, for example, complex custom datapaths, multiword-length DSP datapaths, custom bit-width FP datapaths, and variable bit-width datapaths. It may be useful for merging datapaths generated by variable bit-width synthesizers, such as variable-length C (Valen-C) [33], [34] and Hewlett Packard Lab's Program In, Chip Out (PICO) [35]. Valen-C allows programmers to specify variables of arbitrary bit widths, instead of being limited to the standard bit widths available in C, to better match the needs of the application and to reduce redundancy. PICO performs bit-width analysis on the C program before synthesis to determine the bit widths needed for each functional unit.

X. Conclusion

The bit-alignment problem has been largely ignored in resource sharing, thus limiting resource sharing to simple datapaths. This paper has presented a novel solution to the bit-alignment problem during datapath merging. The results show that significant area reductions are possible with bit alignment during resource sharing. The performance of the bit-alignment algorithm is dependent on the types of instructions being merged and the bit widths of the hardware resources being used by each instruction. The greater the mix of bit widths in the datapaths to be merged, the greater the potential benefit resource sharing with bit alignment will provide.

A design space exploration was performed to investigate the tradeoffs between the area and the performance when generating custom FPUs. The results showed that adding more hardware to the custom FPU does not necessarily improve performance if the additional hardware introduces excessive delay.

To expand the design space exploration to a suite of applications instead of a single application, the entire suite of applications could be run and profiled. In Section VII, the operations were ranked on the order of the number of cycle saved by implementing each operation in hardware when executing that application. For a suite of applications, the operations would be ranked on the order of the number of cycles saved when executing all the applications in the suite. If the designer knows that certain applications will be executed more often than others in the suite, the rankings could be biased toward those applications.

Future work may include investigation of pipelining in the custom FPU generation and how it affects the design space exploration. A heuristic could also be developed to handle a larger number of instructions than the current algorithm.

REFERENCES

[1] *Xtensa Processor*, Tensilica Inc., ASIP Solutions, Santa Clara, CA. [Online]. Available: http://www.tensilica.com
[2] *ARCtangent*, ARC Int., San Jose, CA. [Online]. Available: http://www.arc.com
[3] *ASIP Meister*, ASIP Solutions, Osaka, Japan. [Online]. Available: http://www.asip-solutions.com
[4] J. Liang, R. Tessier, and O. Mencer, "Floating point unit generation and evaluation for FPGAs," in *Proc. FCCM*, Apr. 2003, p. 185.
[5] A. Gaffar, W. Luk, P. Cheung, and N. Shirazi, "Customising floating-point designs," in *Proc. FCCM*, Apr. 2002, pp. 315–317.
[6] A. A. Gaffar, O. Mencer, W. Luk, and P. Y. K. Cheung, "Unifying bit-width optimisation for fixed-point and floating-point designs," in *Proc. 12th Annu. IEEE FCCM*, 2004, pp. 79–88.
[7] A. Gaffar, O. Mencer, W. Luk, P. Cheung, and N. Shirazi, "Floating-point bitwidth analysis via automatic differentiation," in *Proc. EEE Int. Conf. FPT*, Dec. 2002, pp. 158–165.
[8] F. Fang, T. Chen, and R. Rutenbar, "Floating-point bit-width optimization for low-power signal processing applications," in *Proc. IEEE ICASSP*, 2002, vol. 3, pp. III-3208–III-3211.
[9] G. Leyva, G. Caffarena, C. Carreras, and O. Nieto-Taladriz, "A generator of high-speed floating-point modules," in *Proc. 12th Annu. IEEE Symp. FCCM*, 2004, pp. 306–307.
[10] Z. Baidas, A. Brown, and A. Williams, "Floating-point behavioral synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 7, pp. 828–839, Jul. 2001.
[11] W. Geurts, F. Catthoor, and H. D. Man, "Quadratic zero–one programming based synthesis of application specific data paths," in *Proc. ICCAD*, Nov. 1993, pp. 522–525.
[12] J. Um, J. Kim, and T. Kim, "Layout-driven resource sharing in high-level synthesis," in *Proc. ICCAD*, Nov. 2002, pp. 614–618.
[13] C.-J. Tseng and D. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 5, no. 3, pp. 379–395, Jul. 1986.
[14] C. Y. Hitchcock and D. E. Thomas, "A method of automatic data path synthesis," in *Proc. Conf. Des. Autom.*, Jun. 1983, pp. 484–489.
[15] O. Bringmann and W. Rosenstiel, "Resource sharing in hierarchical synthesis," in *Proc. Int. Conf. CAD*, Nov. 1997, pp. 318–325.
[16] S. Raje and R. Bergamaschi, "Generalized resource sharing," in *Proc. Int. Conf. CAD*, Nov. 1997, pp. 326–332.
[17] E. Witte, A. Chattopadhyay, O. K. Schliebusch, R. Leupers, G. Ascheid, and H. Meyr, "Applying resource sharing algorithms to ADL-driven automatic ASIP implementation," in *Proc. Int. Conf. Comput. Des.*, Oct. 2005, pp. 193–199.
[18] N. Moreano, E. Borin, C. Souza, and G. Araujo, "Efficient datapath merging for partially reconfigurable architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 7, pp. 969–980, Jul. 2005.
[19] P. Brisk, A. Kaplan, and M. Sarrafzadeh, "Area-efficient instruction set synthesis for reconfigurable system-on-chip designs," in *Proc. DAC*, 2004, pp. 395–400.
[20] K. Schoofs, G. Goossens, and H. Man, "Bit-alignment in hardware allocation for multiplexed DSP architectures," in *Proc. Eur. Conf. Des. Autom.*, Feb. 1993, pp. 289–293.
[21] *Cliquer*. [Online]. Available: http://users.tkk.fi/~pat/cliquer.html
[22] *SimpleScalar Tool Set*. [Online]. Available: http://www.simplescalar.com
[23] J. Peddersen, S. Shee, A. Janapsatya, and S. Parameswaran, "Rapid embedded hardware/software system generation," in *Proc. Int. Conf. VLSI Des.*, 2005, pp. 111–116.
[24] D. Burger and T. Austin, "The Simplescalar Tool Set, Version 2.0," Madison Comput. Sci. Dept. Univ. Wisconsin, Madison, WI, Tech. Rep. 1342, Jun. 1997. [Online]. Available: http://www.cs.wisc.edu/~mscalar/simplescalar.html
[25] *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std. 754, Aug. 1985.
[26] D. Patterson and J. Hennessy, *Computer Organization and Design*. San Mateo, CA: Morgan Kaufmann, 2005, ch. H.5.
[27] W.-C. Yeh and C.-W. Jen, "High-speed booth encoded parallel multiplier design," *IEEE Trans. Comput.*, vol. 49, no. 7, pp. 692–701, Jul. 2000.
[28] P. Soderquist and M. Leeser, "Area and performance tradeoffs in floating-point divide and square root implementations," *ACM Comput. Surv.*, vol. 28, no. 3, pp. 518–564, Sep. 1996.
[29] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *Proc. Int. Symp. Microarchitecture*, 1997, pp. 330–335.
[30] *The Standard Performance Evaluation Corporation (SPEC)*. [Online]. Available: http://www.spec.org/index.html
[31] *Synopsys Tool Set*. [Online]. Available: http://www.synopsys.com
[32] *SoftFloat*. [Online]. Available: http://www.jhauser.us/arithmetic/SoftFloat.html
[33] A. Inoue, H. Tomiyama, E. F. Nurprasetyo, H. Yasuura, and H. Kanbara, "A programming language for processor based embedded systems," in *Proc. APCHDL*, 1998, pp. 89–94.
[34] H. Yasuura, H. Tomiyama, A. Inoue, and E. F. Nurprasetyo, "Embedded system design using soft-core processor and Valen-C," *IIS J. Inf. Sci. Eng.*, vol. 14, pp. 587–603, Sep. 1998.
[35] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood, "Bitwidth cognizant architecture synthesis of custom hardware accelerators," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 11, pp. 1355–1371, Nov. 2001.

Yee Jern Chong (S'08) received the B.E. (with first-class honors) degree in electrical and electronic engineering from the University of Canterbury, Christchurch, New Zealand, in 2002 and the M.Eng.Sc. degree in electrical engineering from The University of New South Wales, Sydney, Australia, in 2004, where he is currently working toward the Ph.D. degree in the School of Computer Science and Engineering.

His research interests include design automation, embedded systems, computer architecture, and floating-point arithmetic circuits.

Sri Parameswaran (M'92) received the B.E. degree from Monash University, Australia, in 1986 and the Ph.D. degree from the University of Queensland, Australia, in 1991.

He is a Professor with the School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, where he also serves as the Program Director for Computer Engineering. His research interests include system level synthesis, low-power systems, high-level systems, and network on chips. He is also an Associate Editor of the Association for Computing Machinery *Transactions on Embedded Computing Systems* and the European Association for Signal Processing *Journal on Embedded Systems*.

Prof. Parameswaran has served on the program committees of numerous international conferences, such as the Design Automation Conference; Design and Test in Europe; the International Conference on Computer Aided Design; the International Conference on Hardware/Software Codesign and System Synthesis (as the Technical Program Committee Chair); and the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems.