Automatic Application Specific Floating-point Unit Generation

Yee Jern Chong, Sri Parameswaran
School of Computer Science & Engineering
University of New South Wales
Sydney, Australia
{yeejernc,sridevan}@cse.unsw.edu.au

ABSTRACT

This paper describes the creation of custom floating point units (FPUs) for Application Specific Instruction Set Processors (ASIPs). ASIPs allow the customization of processors for use in embedded systems by extending the instruction set, which enhances the performance of an application or a class of applications. These extended instructions are manifested as separate hardware blocks, making the creation of any necessary floating point instructions quite unwieldy. On the other hand, using a predefined FPU includes a large monolithic hardware block with considerable number of unused instructions. A customized FPU will overcome these drawbacks, yet the manual creation of one is a time consuming, error prone process. This paper presents a methodology for automatically generating floating-point units (FPUs) that are customized for specific applications at the instruction level. Generated FPUs comply with the IEEE754 standard, which is an advantage over FP format customization. Custom FPUs were generated for several Mediabench applications. Area savings over a fully-featured FPU without resource sharing of 26%-80% without resource sharing and 33%-87% with resource sharing, were obtained. Clock period increased in some cases by up to 9.5% due to resource sharing.

1. 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. Especially 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 evermore powerful devices, while keeping within tighter contraints.

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 floating point instructions can be emulated in software, the emulated performance leaves a lot to be desired. Therefore, dedicated floating-point hardware is highly sought after for floating-point intensive applications. However, high performance floating-point units (FPUs) are large and complex, and therefore, costly and power-hungry. They are also time-consuming to design.

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 Application Specific Instruction Set Processor (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.

To reduce cost, area and power, it is desirable for the FPU to have as few hardware blocks and interconnections as possible. This can be achieved by reusing 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.

This paper presents for the first time, a methodology for the automatic generation of FPUs customized at the instruction-level, with integrated resource sharing to minimize the area of the FPU. In this methodology, the application to be executed on the system is profiled and the required floating-point instructions are extracted. The netlist describing the datapath for each of the necessary instructions are obtained from a library and passed to a resource sharing process, where the datapaths are merged. The resulting datapath description is then used to generate the HDL for the custom FPU.

An interesting prospect of this methodology is that it would be possible to generate FPUs that contain a mix of single- and double-precision hardware, depending on the requirements of the application.

The rest of this paper is organized as follows: Section 2 discusses previous research in the area of automatic FPU customization and resource sharing. In section 3, we present the FPU generation methodology, including the resource sharing algorithm and bit-alignment. Section 4 describes the VHDL generation stage. The experimental setup is described in Section 5. Section 6 presents and discusses the results from its use in generating application specific FPUs for different media applications. The final section gives the conclusion.

2. RELATED WORK

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

Liang *et. al.* [1] presents a FPU generation tool for FPGAs that chooses an appropriate implementation algorithm and architecture based on user specified requirements. They focus only on choosing the most appropriate FP addition implementation, but omit other operations.

Gaffar *et. al.* [2] automatically customizes floating-point designs by customizing the representation (bit-width) of the FP operations based on user-specified accuracy requirements. They minimize the bit-width while keeping the error/loss of precision to an acceptable level. The downside is that it produces a non-standard FP format with arbitrary mantissa and exponent sizes. Non-standard formats are less desirable because in general, software is developed and verified on IEEE754 compliant machines.

Central to our FPU generation methodology is the resource sharing algorithm for merging the hardware datapaths of each instruction. Resource sharing is an important problem in high level synthesis (HLS). Various solutions to the resource sharing problem have been proposed [3, 4, 5, 6, 7, 8, 9, 10, 11].

Recently, Brisk *et. al.* [3] proposed an algorithm for merging a set of custom instructions into a single datapath. Another approach was

978-3-9810801-2-4/DATE07 © 2007 EDAA

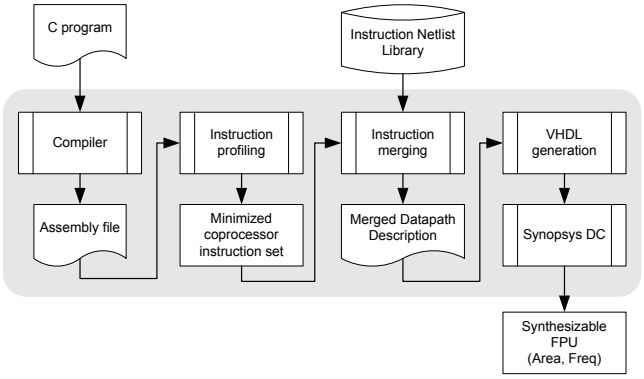

Figure 1: Methodology

presented by Moreano *et. al.* [4]. The approaches of [3] and [4] exclude many low-level details, which make them unsuitable for merging very complex architectures (such as floating-point architectures). Floating-point architectures have many low-level details to consider, such as input bit vectors made up of several smaller bit vectors. Our work is based on the technique presented by [4], but adapted to support the required low-level information required.

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 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.

Previous merging techniques do not consider the bit-alignment problem. This is a problem related to sharing hardware components and interconnects of differing bit-widths. When a wider component (e.g. 11-bit adder) is shared with a smaller component (e.g. 8-bit adder), the issue arises as to how the smaller component should be aligned within the larger one. Schoofs *et. al.* [12] investigated a similar issue when different bit length data is executed on the same DSP hardware. They placed re-alignment muxes, which they called 'routers', at the inputs to functional units align the input signals. This approach would be too costly in terms of area and delay for our FPU generation scheme.

In this paper, we describe how a customized FPU is generated, and show how the bit alignment problem can be solved. In particular our ***contributions*** are:

- a constructive methodology to create a customized FPU; and
- a novel algorithm which solves the bit alignment problem.

3. METHODOLOGY

The methodology for the FPU generation is outlined in Figure 1. 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 FP instructions from that subset are extracted and further reduced by omitting rarely used instructions that contribute a sizeable area cost. These are emulated in software with an acceptable sacrifice of performance for area (the design space exploration is beyond the scope of this paper).

The subset of instructions that the FPU needs to support is then passed to the instruction merging stage. In this stage, the netlist for each instruction is obtained from a library of instructions and passed through the instruction merging algorithm. The instruction merging algorithm merges the datapaths of each instruction into a single shared datapath and outputs a new netlist. In addition to the netlist, auxiliary data for placement and control of multiplexors are also generated.

The netlist is passed to the hardware generation stage, which generates the VHDL describing the shared datapath using a library of VHDL components. The auxiliary data from the previous stage is used to generate the multiplexors and the control signals for each of the multiplexors.

The generated VHDL is then ready to be packaged as a co-processor or a functional unit.

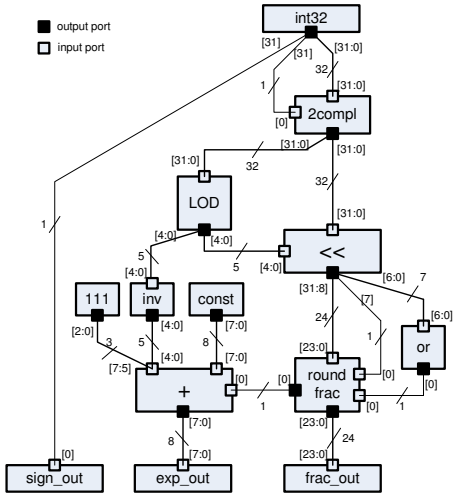

Figure 2: Structure of datapath to convert 32b integer to single precision FP format

3.1 Instruction Merging Algorithm

The technique used for merging the instruction datapaths is based on the maximum weight clique approach presented in [4]. However, rather than merging simple graphs as shown in [4], more detailed structures are merged. Due to the complex architectures associated with floating-point and DSP hardware, extra lower-level detail is necessary to ensure correct operation. The merging algorithm has to handle different bit-width components, bit-alignment issues, multiple output ports, and different bit-width ports on the same component. A datapath structure would look similar to Figure 2, which depicts the structure of a datapath for the *Convert integer to single-precision FP* operation. It is a visual representation of a netlist describing the datapath.

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

- Type of function of the component
- Bit-width
- No. of input ports and their bit-widths
- No. of output ports and their bit-widths
- Area of component

Connections in the netlist are characterized by:

- Bit-width
- Source component
- Source port no.
- Destination component
- Destination port no.
- Bit-range of source port
- Bit-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). Components in the netlist can be written as large functional units (coarse grained), such as a multiplier or rounding unit, or discrete gates (fine grained). The level of granularity that the netlists are written in will determine the granularity of the instruction merging. A connection represents an interconnect between

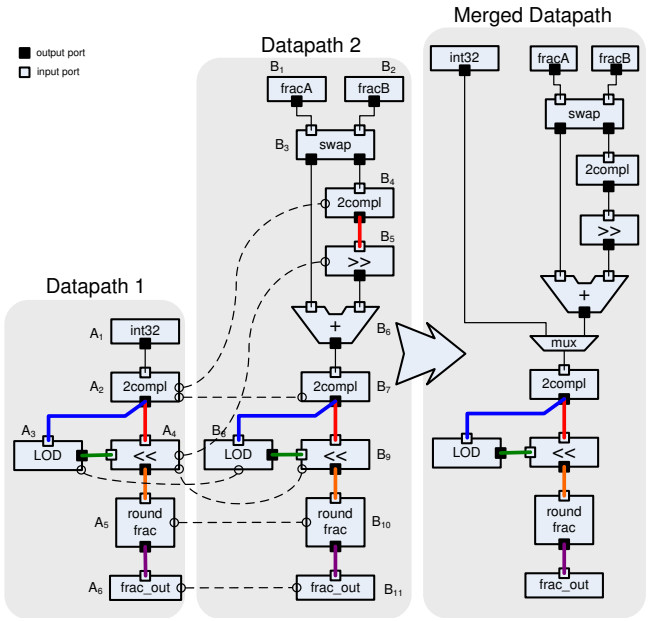

Figure 3: Datapath Merging Example

a range of bits on a component's port with a range of bits of another component's port.

A netlist for each instruction to be supported is created and added into a library. The inputs to the merging algorithm are the netlists for the instructions to be merged. These are selected from the pre-designed library of netlists.

The merging of two datapaths is illustrated in Figure 3, where the dotted lines represent possible hardware mappings and the thick interconnects represent possible interconnect mappings. Datapath 1 in Figure 3 represents a simplified version of Figure 2 and Datapath 2 represents a simplified version of the datapath for the fractional part of a floating point addition. The merged datapath in Figure 3 illustrates how the two datapaths could be merged.

For simplicity, the netlists for the datapaths to be merged can be modelled 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:

- a set of input ports $p_{in} = 1...N_{in}(v)$, if $N_{in}(v) > 0$.
- a set of output ports $p_{out} = 1...N_{out}(v)$, if $N_{out}(v) > 0$.
- attributes specifying its type, bit-width, estimated area and bit-width of each port.

An edge $e = (u, p_{out}, v, p_{in}) \in E$ represents an interconnect from output port p_{out} of vertex u to input port p_{in} of vertex v.

Instruction datapaths to be merged are represented by DFGs G_i for $i = 1...n$. 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 illustrated in Figure 4 and described in the following sub-sections. Figure 4(I) shows the two graphs to be merged. Figure 4(II) shows the hardware and interconnect mappings, where the dotted lines indicate which components and interconnects can be shared. Figure 4(III) shows the compatibility graph and Figure 4(IV) shows the resulting merged graph.

3.1.1 Hardware and interconnect mapping

The first step of the merging algorithm is to find all possible mappings between the two netlists to be merged, say $G_i = (V_i, E_i)$ and $G_j = (V_j, E_j)$. The example in Figure 4(II) shows possible mappings between two graphs (dotted lines). A vertex $v_i \in V_i$ can be

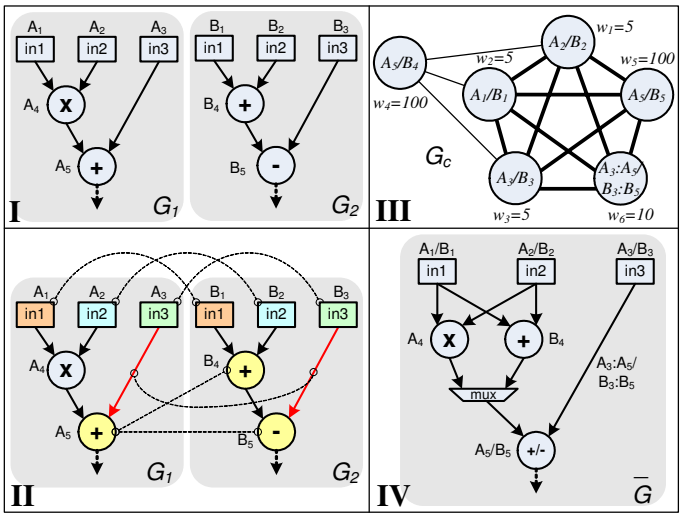

Figure 4: DFG merging process: I - DFG G_1, G_2; II - Hardware and Interconnect mapping; III - Compatibility graph and maximum weight clique solution; IV - Merged DFG

merged with a vertex $v_j \in V_j$ 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 savings by a mapping are 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 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 $e_i = (u_i, p_{out_i}, v_i, p_{in_i}) \in E_i$ and $e_j = (u_j, p_{out_j}, v_j, p_{in_j}) \in E_j$ can be mapped if they satisfy these conditions:

- source vertex u_i can be mapped to source vertex u_j.
- destination vertex v_i can be mapped to destination vertex v_j.
- source port p_{out_i} matches source port p_{out_j}.
- destination port p_{in_i} matches destination port p_{in_j}.

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: $Area\_saved(e_i/e_j) = Area(mux)$.

3.1.2 Non-beneficial 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 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. For example, if a small component like a 2-input AND gate can be shared, but the interconnects at its inputs cannot be shared, then it requires multiplexors at its inputs (Figure 5). In this case, the area of two unshared AND gates would be less than a shared AND gate with multiplexors.

3.1.3 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 that represents which mappings are

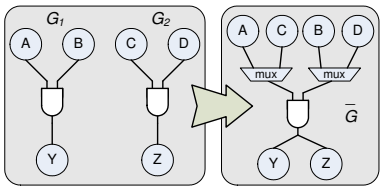

Figure 5: Case where merging results in larger area

compatible with each other. Figure 4(III) 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 mapping v_i/v_j between $v_i \in G_i$ and $v_j \in G_j$ or a connection mapping e_i/e_j between $e_i \in G_i$ and $e_j \in G_j$. 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 3.1.1). 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.

3.1.4 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 $G_{MWC} \subseteq G_c$, 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 Figure 4(III) shows the maximum weight clique for the example.

This is known to be an NP-complete problem and is solved using a heuristic polynomial-time algorithm. The Cliquer tool [13], 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.

3.1.5 Bit Alignment

Floating-point instructions require many components of varying bit-widths, and therefore it is desirable to merge components of different bit-widths. This brings up the problem of bit-alignment where datapaths with different bit widths share the same hardware and interconnects. The problem is how a narrower component or interconnect should be aligned within a wider component or interconnect, while maintaining correct operation, maximizing the sharing of wires and minimizing re-alignment points. Another issue is how to integrate the bit-alignment into the existing merging algorithm.

In the case of resource sharing, bit-alignment problems only exist in shared components or interconnects. Unshared resources have their own separate hardware, so they do not suffer the same problem. Therefore, we only need to examine the mappings in the compatibility graph when considering bit-alignment problems. Where there exists an interconnect mapping that cannot be aligned, a mux must be placed to ensure both interconnects align correctly to a component input. This can be done by deleting from the compatibility graph the interconnect mapping that cannot be aligned.

When a smaller component is mapped to a larger component, we refer to the larger component as the 'carrier' and the smaller component as the 'passenger'. The 'passenger' has to be aligned within the 'carrier'. We represent the alignment as an offset from either the LSB side or the MSB side. In most cases, components have zero offset from one of the sides.

Components are classed as either having *flexible* or *fixed* alignments. With a component having *flexible alignment*, the alignment of the data word is not critical to its correct operation. For example, adders, subtractors and bit-wise operations. The input data can be freely aligned (with appropriate padding bits) without affecting the result, though the alignment of the result at the output may change. Note that there may be other considerations, like the location of the carry-out bit changes with different alignments in adders.

With a component with *fixed alignment*, the alignment of data word is critical to its correct operation. For example, a leading-one detector (LOD) requires the MSB of the input to be aligned to the MSB of the LOD.

Each component in the netlist is tagged with the following information:

- its alignment (flexible or fixed).
- input and output ports that are affected by alignment.
- alignment offset (if fixed alignment).

In floating-point architectures, the majority of components have flexible alignments with a few fixed alignment components. Our method for bit-alignment is performed by analyzing the compatibility graph from Section 3.1.3. A breadth-first search traversal of the compatibility graph is performed, starting at a mapping of two fixed alignment components with different bit-widths. The alignment of the passenger within the carrier is set according to the offset tag. Next, the compatibility graph is traversed to all vertices affected by this alignment, propagating the alignment offset. Vertices affected by a component alignment are connection mappings connected to the component, and vice versa.

The connection mappings at the affected vertices are aligned based on the propagated alignment offset. The traversal continues by propagating the offset to all affected component mappings. The process continues until a termination condition is met. A termination condition occurs:

- if there exists no more shared connection along a path; or,
- if a component mapping has no input or output port that is affected by the alignment; or,
- an alignment is impossible (the passenger component extends beyond the boundaries of the carrier due to the required alignment).

When a termination condition is met, the traversal along that path is halted. If alignment is impossible, the connection mapping just before the termination point is marked for deletion. The traversal path and alignment information for each vertex is stored as an alignment tree T. The process is repeated starting at the next fixed alignment component mapping on a copy of the original compatibility graph. This continues until all vertices with a fixed alignment component mapping has its own alignment tree. The resulting alignment trees are then compared and merged. Any alignment trees that do not have vertices that overlap can be merged directly. Alignment trees that have overlapping nodes can be also be merged directly if the overlapping nodes have the exact same alignment, otherwise they are incompatible. To merge alignment trees that are incompatible, the alignment of one one of the trees over-rides the overlapping parts of the other incompatible trees. The final merged alignment tree is then applied to the compatibility graph by applying the alignment data to each vertex and deleting vertices that are marked for deletion.

Algorithm 1 Bit-alignment

```
Let 𝕋 = {T_1, ..., T_n} be a set of alignment trees
for all v_i ∈ G_c do
    if v_i is a fixed alignment component mapping with bit-width mis-match then
        T_i ← Build alignment tree rooted at v_i
T̄ ← merge_alignment_trees(𝕋)
Ḡ_c ← apply_alignment_tree(T̄, G_c)
return Ḡ_c
```

The bit-alignment algorithm is shown in Algorithm 1 and the overall algorithm for the instruction merging is shown in Algorithm 2.

Algorithm 2 Datapath merging of n instructions

//Merge instructions represented by $G_i = (V_i, E_i)$ *for* $i = 1...n$
Let $G_a = (V_a, E_a)$ and $G_b = (V_b, E_b)$ be the graphs to merge.
$G_a \leftarrow G_1$
for $i \leftarrow 2$ to n **do**
  $G_b \leftarrow G_i$
  //Hardware mapping
  Let $M = \{m_1, ..., m_n\}$ be a set of mappings,
  where $m_k = (v_i, v_j, w_k)$ *or* (e_i, e_j, w_k), and v_i/v_j is a mapping between hardware components, e_i/e_j is a mapping between interconnects and w_k is the area saved by the mapping.
  $M \leftarrow$ build_mappings(G_a, G_b)
  //Non-beneficial mapping removal
  for all $m_k \in M$ **do**
    if multiplexor(s) required **then**
      Adjust w_k to account for mux area
    if $w_k \leq 0$ **then**
      delete m_k from M
  $G_c \leftarrow$ construct_compatibility_graph(M)
  $\overline{G_c} \leftarrow$ bit_alignment(G_c)
  $G_{MWC} \leftarrow$ find_max_weight_clique ($\overline{G_c}$)
  $\overline{G} \leftarrow$ reconstruct_DFG(G_{MWC}, G_a, G_b)
  //Merged graph fed back for next iteration
  $G_a \leftarrow \overline{G}$
return $\overline{G}$

4. VHDL GENERATION

The VHDL generation stage generates a structural VHDL representation of the netlist. The VHDL generation tool was written in C. Its inputs are the netlist of the merged datapath from the instruction merging stage and auxiliary information for multiplexor placement and control. The netlist is converted into data structures describing components and interconnections. The component data structures are then used to generate the component declarations, and port mapping using a library of VHDL descriptions for each component is used. The input and output ports of each component is assigned a unique identifiable signal name in the port mapping to ease signal assignment. The connection data structures are used to generate the signal assignments. The auxiliary information passed from the instruction merging stage helps the VHDL generation tool with multiplexor placement and control. Multiplexors are placed at input ports that have more than one interconnect connected to it. The VHDL generation tool simply uses conditional signal assignments to create these muxes (e.g. *signal_in <= signal_in_1 when (instruction = fadd64) else signal_in_2;*).

5. EXPERIMENTAL SETUP

To test the methodology, FPUs for a Simplescalar base processor were generated targeting various applications. The FPUs were attached to the base processor as a tightly coupled co-processor. The processor is based on the SimpleScalar/PISA instruction set [14], which includes the following FP instructions:

- single and double precision FP arithmetic instructions: addition, subtraction, multiplication, divide and square root.
- conversion instructions between integer, single and double precision formats.
- data movement or manipulation instructions: move, absolute, negate.
- branch decision instructions.

The FPU has a separate register file and uses instructions such as, *move to co-processor1 (mtc1)* and *move from co-processor1 (mfc1)* to move data between the base processor and FPU. Of all the instructions, the arithmetic and conversion instructions would benefit most from instruction merging, as other instructions do not require complex hardware. The hardware to support these instructions were designed and added to the instruction library as netlists. The floating point architecture was designed to be IEEE754 [15] compliant.

Table 1: Simulation results

App	No resource sharing				Resource sharing			
	Area	%ΔA	T(ns)	%ΔT	Area	%ΔA	T(ns)	%ΔT
I	104331	NA	11.7	NA	NA	NA	NA	NA
II	83404	-20.0	11.8	0.09	59947	-42.5	13.6	15.5
Epic	43068	-58.7	11.6	-0.9	29039	-72.1	12.9	9.5
Unepic	20934	-79.9	9.4	-19.8	13684	-86.9	10.3	-12.6
mpeg2dec	69792	-33.1	11.7	-0.3	63156	-39.5	12.4	5.4
mpeg2enc	77062	-26.1	11.9	1.2	69725	-33.2	12.4	5.6
jpegdec	23459	-77.5	9.6	-18.1	17343	-83.4	10.3	-12.5

The FP addition/subtraction architecture is usually the most complicated of the FP instructions. The FP addition algorithm is based on the standard 5-stage design described in [16].

The multiplier is usually the most critical single component in floating point designs because of the size and delay. Double precision floating point multiplication requires a large 54x54 bit multiplier. Therefore, a fast Booth-encoded Wallace-tree multiplier was used. Radix-4 Modified Booth Encoding (MBE) was used to reduce the number of partial products, a Wallace tree was used to reduce the partial products and a fast carry-select adder was used for the final addition.

The floating point divide and square root were designed based on an iterative Radix-2 SRT algorithm. This architecture was chosen for its simplicity and small area. The downside is that the latency is equal to the number of bits it has to compute. However, in most applications, divide and square root instructions are much less frequent than additions and multiplications.

Rounding mode was set to the IEEE default round-to-nearest-even, but the other rounding modes can be easily implemented. Exceptions arising from overflows, invalid operands, NaNs and infinity are detected. Hardware support for denormals and underflow was not implemented and is assumed to be handled in software.

Applications were selected from the Mediabench suite [17] and compiled using the Simplescalar [14] compiler. The compiled binaries were then profiled using the Simplescalar profiling tool. The instruction profiles were analyzed to find the subset of floating point instructions to support. The instructions were fed into the instruction merging tool to obtain the merged datapath. The VHDL for the merged datapath was generated and synthesized using Synopsys Design Compiler [18] to obtain area and timing information. Synthesis was performed with the 0.18μm TOWER library available from Synopsys.

Custom FPUs were generated for several Mediabench applications - EPIC decoder and encoder, MPEG2 decoder and encoder and JPEG decoder. The JPEG application was executed with the *"-dct float"* flag to force it to use a floating point DCT instead of the default integer DCT. If both single and double precision versions of the same operation needs to be supported (e.g. single-precision FP add & double-precision FP add), only the double-precision datapath is synthesized because the single precision instruction can be executed on the double-precision datapath, but not vice-versa. To gauge the effectiveness of resource sharing, custom FPUs were generated with and without resource sharing.

6. RESULTS

Table 1 shows the area and timing results of the simulations. The first column shows the application for which the FPU was customized for. In this column, **I** is an FPU with all single- and double-precision FP instructions in the PISA instruction set. **II** is an FPU with all double-precision FP instructions. **II** is of interest because double-precision hardware can execute single-precision operations without loss of precision, but not vice-versa. Columns 2-5 show the area (no. of gates), % change in area of each FPU (compared to I), minimum clock period and % change in clock period (compared to I) without resource sharing applied. The last four columns show the same as columns 2-5 but with resource sharing applied.

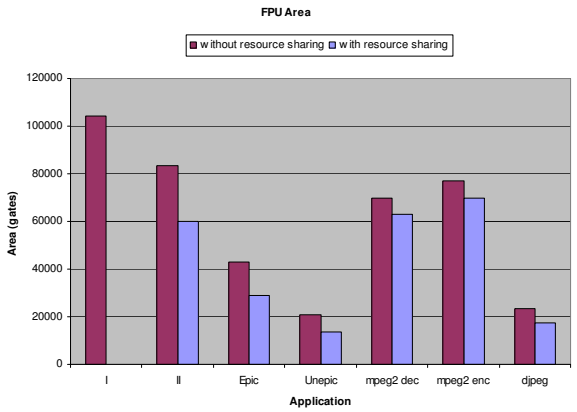

Figure 6: FPU area comparison

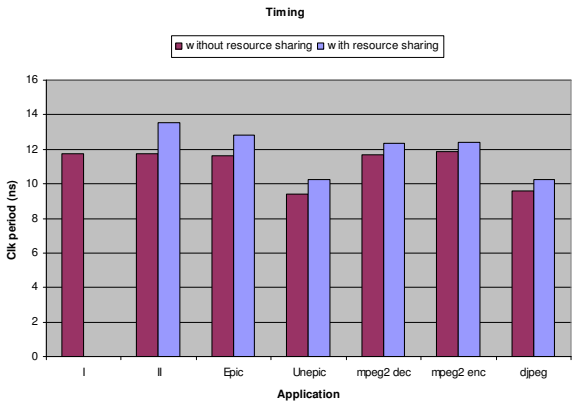

Figure 7: FPU timing

For all the clock period values for the FPUs with resource sharing, Synopys Design Compiler's static timing analysis was unable to extract accurate timing numbers due to the formation of false loops [19, 5]. False loops is a problem often associated with resource sharing. It occurs when combinatorial loops in the datapath are formed by merging datapaths, but never occur in actual operation because the execution of the datapaths are mutually exclusive. The timing results in Table 1 were adjusted manually to account for the false loops by examining the output of the timing analysis.

Figure 6 compares the area of the FPUs with and without resource sharing. **I** is used as the baseline for the comparisons. If single-precision instructions are executed on double precision hardware, the single-precision hardware can be removed, resulting in II. This reduces the size of the FPU by 20% without resource sharing, and another 42.54% with resource sharing. The area savings that were achieved for the custom FPUs ranged from 26% to 80% without resource sharing and 33% to 87% with resource sharing.

As shown in Figure 7, the clock periods for most of the generated FPUs increased. This can be attributed to the insertion of muxes into the critical path as a result of resource sharing. This is unavoidable unless the resources on the critical path are excluded from the sharing process. Assuming a new critical path is not formed, the clock period would not increase. In some cases (unepic & djpeg), the clock period reduced because the datapath that contributed most to the critical path delay in the other FPUs (double precision addition) was not included in the merging. The increase in clock period due to resource sharing can be reduced at the expense of increased area by excluding resources on the datapath that contributes the most critical path delay from the sharing. As long as this delay is not exceeded by the merging, the clock period will not increase. Pipelining could also be used to reduce the effects of the muxes on the clock period.

7. CONCLUSION

This paper presented a methodology for automatically generating FPUs customized at the instruction level, incorporating resource sharing to minimize area. FPUs were generated for different media applications and compared to typical general purpose FPUs. Area savings of up to 87% were observed compared to the fully featured reference FPU (without resource sharing). Clock periods increased in most cases due to insertion of muxes into the critical path. The increase in clock period due to resource sharing can be reduced at the expense of increased area by excluding resources on the datapath that contributes the most critical path delay from the sharing. Pipelining can also be used to reduce the effects of the mux delays and reduce clock period. At this stage, the generated FPU is not pipelined. Pipeline stages can be added manually if desired. We will explore integrating automatic pipelining in future work. In the future, this methodology could be expanded to generate DSP, or multimedia co-processors, or even one with a mixture of floating-point, DSP and multimedia instructions.

8. REFERENCES

[1] J. Liang, R. Tessier, and O. Mencer. Floating point unit generation and evaluation for FPGAs. In *Proceedings of 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003.

[2] A.A. Gaffar, W. Luk, P.Y.K. Cheung, and N. Shirazi. Customising floating-point designs. In *Proceedings of 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[3] P Brisk, A Kaplan, and M Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *Proceedings of Design Automation Conference*, pages 395 – 400, 2004.

[4] N Moreano, E Borin, C Souza, and G Araujo. Efficient Datapath Merging for Partially Reconfigurable Architectures. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 24, pages 969–980, July 2005.

[5] W. Geurts, F. Catthoor, and H. De Man. Quadratic zero-one programming based synthesis of application specific data paths. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93). Digest of Technical Papers*, Nov 1993.

[6] J. Um, J. Kim, and T. Kim. Layout-driven resource sharing in high-level synthesis. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, Nov 2002.

[7] Chia-Jeng Tseng and D.P. Siewiorek. Automated Synthesis of Data Paths in Digital Systems. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 5, July 1986.

[8] C.Y. Hitchcock III and D.E. Thomas. A Method of Automatic Data Path Synthesis. In *20th Conference on Design Automation*, June 1983.

[9] O. Bringmann and W. Rosenstiel. Resource sharing in hierarchical synthesis. In *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers*, Nov 1997.

[10] S. Raje and R.A. Bergamaschi. Generalized resource sharing. In *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers*, Nov 1997.

[11] E.M. Witte, A. Chattopadhyay, O. SchliebuschKammler, R. Leupers, G. Ascheid, and H. Meyr. Applying resource sharing algorithms to ADL-driven automatic ASIP implementation. In *International Conference on Computer Design*, Oct 2005.

[12] K Schoofs, G Goossens, and HG Man. Bit-Alignment in Hardware Allocation for Multiplexed DSP Architectures. In *Proceedings of the 4th European Conference on Design Automation with the European Event in ASIC Design*, pages 289 – 293, Feb 1993.

[13] Cliquer. http://users.tkk.fi/ pat/cliquer.html.

[14] SimpleScalar Tool Set. http://www.simplescalar.com.

[15] IEEE standard for binary floating-point arithmetic, 1985.

[16] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design*, chapter H.5. Morgan Kaufmann Publishers, 3rd edition, 2005.

[17] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, 1997.

[18] Synopsys Tool Set. http://www.synopsys.com.

[19] L. Stok. False loops through resource sharing. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD-92). Digest of Technical Papers*, Nov 1992.