High-Level Synthesis of Software-Customizable Floating-Point Cores

Samridhi Bansal, Hsuan Hsiao, Tomasz Czajkowski[†], Jason H. Anderson
Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada
[†]Intel Corp., Toronto, ON Canada
{samridhi.bansal, julie.hsiao}@mail.utoronto.ca, janders@ece.utoronto.ca

Abstract—**Parameterized cores with fixed capabilities are typically used for floating-point (FP) operations on FPGAs. However, such standard cores can be overprovisioned or lack specific specializations as required by applications. We consider FP cores described in the `C` language, synthesized to hardware using the LegUp high-level synthesis (HLS) tool [1]. Their software specification permits straightforward customization to non-compliant variants having superior area and performance characteristics, such as reduced-precision floating point, or cores without full IEEE 754 exceptions support. We create and evaluate the IEEE 754 FP standard cores for the key operations of addition, subtraction, division and multiplication, targeted to an FPGA and compare with widely used optimized RTL FP cores from Altera [7] and FloPoCo [3]. The software-specified HLS-generated cores are surprisingly close to the optimized RTL cores in terms of area/performance, and superior in certain cases, such as FP division.**

I. Introduction

Floating-point (FP) computations, when implemented on FPGAs, have typically been realized using intellectual property (IP) cores provided by either the FPGA vendor (e.g. Xilinx or Altera/Intel), or a third-party vendor or project, such as FloPoCo [3]. The usual design methodology is for the user to specify their FP-core performance requirements (desired $Fmax$ or cycle latency) to a core-generator tool, and the tool will produce an optimized RTL module that meets the performance requirements, while minimizing circuit area. The user then instantiates the generated RTL module within their design. In this paper, we evaluate an alternative to the traditional approach for FPGA FP computing: namely, we use high-level synthesis (HLS) to *automatically* generate FP cores from software.

Synthesizing FP cores from software offers a number of advantages vs. the use of IP cores specified in RTL. First, the cores are considerably easier to modify, as desired alterations can be made in software, as opposed to a user attempting to make changes in Verilog or VHDL. For example, one may wish to modify the core to use non-standard bitwidths to tailor precision to specific application needs, saving area and improving performance. Or likewise, for certain applications, full compliance to the IEEE 754 floating-point standard may be unnecessary, and the user may wish to remove handling for certain scenarios that they know will never arise for their particular application (e.g. detecting $-\infty$).

Another key advantage is that HLS-generated FP cores may be synthesized concurrently with the surrounding circuit in which they reside. With RTL IP instantiated cores, the user must know the cycle latency and initiation interval of the core they are instantiating into their design. They must then design the surrounding circuit to inject inputs and expect outputs at the correct times, consistent with the core's operation. With the HLS approach, the FP core is scheduled along with the rest of the circuit, relieving the user from such concerns, improving ease of use. The use of HLS also offers the potential for co-optimization across the boundaries between the FP core and the surrounding design, with possible area/speed benefits relative to the instantiation of a "black box" RTL core, which may be left intact by downstream RTL synthesis tools.

In this work, we specify IEEE 754-compliant single-precision FP cores in `C` software for the key operations of addition, subtraction, multiplication and division. The cores are synthesized to hardware using the LegUp HLS tool from the University of Toronto [1]. The synthesized cores are pipelined and can accept new inputs every clock cycle. Through the adjustment of clock-period constraints provided to the HLS tool, cores with different area/performance trade-offs are generated. In an experimental study, we target the Altera Cyclone V 28nm FPGA and compare the HLS-generated cores with Altera commercial FP cores, as well as open-source FP cores from FloPoCo. Results show that in the case of addition/subtraction, the HLS-generated cores are relatively close in area/performance to the RTL cores. In the case of division, the HLS-generated cores are superior to FloPoCo RTL cores. We also evaluate the area/performance characteristics of a reduced-precision non-754-compliant FP core variant we believe would be valuable to end-users.

A frequently raised concern regarding HLS pertains to the circuit quality "gap" between hand-designed RTL hardware and HLS-generated hardware. While HLS has shown itself to be a competitive design methodology for certain application domains, such as streaming/dataflow [2], to the authors' knowledge, this is the first work to demonstrate the potential for HLS in FP computing.

The four FP cores (including additional non-compliant variants) were developed by one engineer over a period of 16 weeks with no prior knowledge of IEEE 754 floating point, underscoring the enormous productivity boost afforded by the HLS design methodology. Upon publication, the `C` source code for the FP core implementations will be made publicly accessible to the research community.

II. Background and Related Work

A. IEEE 754 Standard

IEEE defines a standard for the representation of floating-point numbers and computations thereupon. In this paper, we are concerned primarily with *single precision* 32-bit floating point; however, the results and work presented are extensible to the double-precision case (64-bit representation). In single-precision FP, the 32-bits are partitioned as follows: 1-bit sign (s), 8-bit exponent (e_{7-0}) and 23-bit mantissa (m_{22-0}). The interpretation of the bits for a floating point number z is as follows:

$$z = (-1)^s \times 2^{e_{7-0}-(127)_2} \times 1.m_{22-0} \tag{1}$$

Observe that the exponent is stored in a *biased* form and ranges from 0 to 255. To remove the bias, 127 is subtracted from the specified exponent.

In the "normal" floating point, as above, there is an implicit 1. in the significand (see the right term in above equation). In such a representation, it may not be possible to represent numbers very close to 0, which are smaller in absolute value than $|1.0 \times 2^{-126}|$. To handle the representation of such numbers – the *subnormal* numbers – the 754 standard specifies special handling to eliminate the leading 1. in the significand. Typically, FPGA hardware implementations of floating point clamp the subnormal numbers to 0 [3], [7]; we likewise do not consider subnormal numbers in our FP core implementations.

B. High-Level Synthesis

High-level synthesis (HLS) refers to the synthesis of a behavioral (untimed) functional specification in software into a hardware circuit, typically specified in RTL with VHDL or Verilog. A key step in HLS is *scheduling*, which assigns computations in the software into time steps, which then form states in a finite-state machine. The *binding* step assigns computations of a given type (e.g. multiply or add) from the input specification to specific hardware units.

With respect to floating point, the primary factors affecting HLS include: 1) the latency and injection-rate (*initiation interval*) of floating-point units, and 2) the permitted number of floating-point units of a given type. The former factor affects scheduling: the scheduler must be aware of the number of clock cycles each operation takes to complete, and the rate at which new operands may be injected. The latter factor, unit count, affects scheduling and binding. The scheduler must limit the number of floating-point operations that occur in a given cycle based on the number of available units. Binding then decides which operations, from the input source code, are to be executed on each specific hardware unit.

C. Floating Point in HLS

The approach normally taken in research/open-source HLS tools is to make use of third-party FP libraries. Floating point operations in the software being synthesized are recognized during compilation, and the appropriate third-party core is then instantiated by the HLS tool. The BAMBU HLS framework [11], from Polytecnico di Milano, instantiates cores from FloPoCo [3]. FloPoCo offers cores for all fundamental FP operations (+, -, /, *), as well as some transcendental functions. The FloPoCo core generator accepts a clock-period constraint as input, and generates a core in VHDL appropriately pipelined for the target constraint.

DWARV HLS [10], from the Technical University of Delft, targets Xilinx FPGAs and instantiates FP cores that are pre-generated by Xilinx's CoreGen tool. The LegUp HLS tool [1] instantiates Altera FP cores when targeting Altera devices, but also has support for the use of FloPoCo cores, when targeting FPGAs from other vendors. The manner in which commercial HLS tools handle FP computations is proprietary and not publicly disclosed.

III. Software-Specified FP Cores

In this section, we delve into the details of the software FP core specification destined for HLS. Floating-point addition is outlined in the next section, as it is used to illustrate some of the coding examples involved. It is well known that coding style and HLS constraints have a significant role in the quality of circuit generated by an HLS tool. For example, [8] demonstrated orders-of-magnitude improvements in circuit performance through code changes and HLS constraints. Thus, we highlight key approaches in two categories that we applied to raise the hardware area efficiency and performance: code transformations and HLS constraints.

A. Floating Point Addition

Floating point addition is analogous to performing addition in scientific notation. Assume we wish to add X = 12.25 and Y = 3.75 together. In single-precision floating point, the two numbers are represented as

$$X = 0(10000010)10001000000000000000000$$

$$Y = 0(10000000)11100000000000000000000$$

where the sign bit is on the left, the exponent appears in parentheses, and the mantissa is on the right. The main steps for floating-point addition are as follows:

1) Extract the sign, exponent and mantissa bits from the operands and explicitly represent the hidden '1.' in their mantissas.

$$X = (-1)^0 \times 1.1000100000.. \times 2^3$$

$$Y = (-1)^0 \times 1.1110000000.. \times 2^1$$

2) Equalize the operand exponents by shifting the decimal point of the mantissa of the *smaller* operand to the left by the difference in exponent values. Exponent difference $= 3 - 1 = 2$, therefore:

$$Y = (-1)^0 \times 0.011110000000.. \times 2^3$$

3) Add the adjusted mantissas and set the result exponent as the exponent of the larger operand.

$$\text{Result mantissa} = 1.10001.. + 0.01111.. = 10.00..$$

$$\text{Result exponent} = 3$$

4) Normalize and round the resulting mantissa to bring it back to standard form (with a leading 1.) by adjusting the result exponent.

$$\text{Before normalizing} = (-1)^0 \times 10.0000000.. \times 2^3$$

$$\text{After normalizing} = (-1)^0 \times 1.000000000.. \times 2^4$$

5) Remove the implicit 1. from the mantissa and reconstruct the result into floating point format.

While performing floating-point addition, there exists the potential loss of bits when shifting the smaller operand's mantissa (as done in the example above). To combat this, three extra bits are used: a guard, a rounding and a sticky bit at the least-significant end of the mantissa. These bits are used to check for any loss and to obtain the correctly rounded result, and the interested reader is referred to [9] for complete details.

IV. FP Cores Implementation in HLS-Style C

Our C implementations of the FP cores first extract the sub-fields of input FP operands (sign, exponent and mantissa) into unsigned values. Then, all necessary computational work is performed in fixed-point arithmetic. Finally, the sub-fields of the computed result are assembled into a 32-bit FP output value. The following code snippet illustrates masking/shifting to extract the sub-fields of operand a:

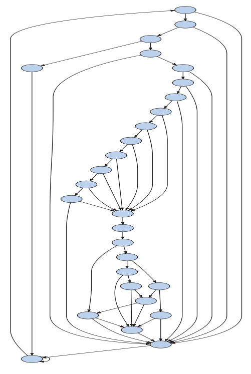
```
unsigned int FPAdder(unsigned int a, ...) {

unsigned char a_sign = a >> 31;
unsigned short a_exp = (a & 0x7f800000) >> 23;
unsigned int a_mantissa = a & 0x007fffff;
...
```

We began with the FP adder and implemented a functionally correct 754-compliant version in standard C with typical software use of *if-else* conditional constructs. Compilers such as LLVM typically translate *if-else* statements to *branch* instructions, resulting in multiple basic blocks in the program. Fig. 1(a) shows the control-flow graph (CFG) of the initial version we created, comprising many basic blocks and complex control. Clearly, the control flow was too complex, making it impossible for the HLS tool to apply pipelining and generate a streaming FP core capable of having multiple operands "in flight" at various stages of the pipeline.

In contrast to *if-else*, ternary operators (`<cond> ? <val1> : <val2>`) are typically implemented using *select* instructions, which do not introduce control flow. We removed *all* control flow through extensive use of the ternary operator, ultimately realizing the entire FP core in a single basic block – straight-line code with no control flow (Fig. 1(b)). Such branch-free code can be transformed into a pipelined circuit by HLS. To illustrate, the following snippet shows how the sign bit of the result is computed using the exponents exp and mantissas mantissa of operands a and b.

```
result_sign = ((a_exp > b_exp) |
    ((a_exp == b_exp) &
       (a_mantissa >= b_mantissa))) ?
       a_sign : b_sign;
```


(a) before

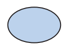

(b) final CFG

Fig. 1: CFG of FP adder before and after code restructuring.

Note the use of bitwise operators (`|`, `&`) rather than logical operators (`||`, `&&`) in the above. We observed that the compiler sometimes inserted control flow when logical operators were used.

Then, having created a pipelined hardware implementation through HLS, we proceeded to make code changes to optimize circuit area. We manually identified common subexpressions and factored these out, eliminating the possibility of duplicate computational work. We also used the narrowest datatypes possible for each piece of computational work. For example, the unsigned char type is wide enough to hold the exponent in single-precision FP. By explicitly identifying variables of narrow widths, HLS specific compiler analysis can more effectively reduce the width of circuit datapaths.

One of the more interesting area-reducing code transformations we applied was in the leading-zero count, which is necessary for the normalization step, discussed above. This step finds the number of leading zeros in the result mantissa, so that it can be brought into standard form with the leading 1., adjusting the result exponent accordingly. We use logical operations and successively smaller types, yielding lower circuit area. The snippet below counts the number of leading zeros in 32-bit-type X and puts the count into n. Line 2 set condition cond to true (1) if X has 16 leading zeros. Line 3 sets a bit (corresponding to 16) in n if cond is true. Line 4 declares a variable X1 with a narrower 16-bit type, into which the remaining portion of X to be examined is placed. In line 7, an yet narrower 8-bit C type is used for the remaining computational work.

```
1: unsigned int X = ...
2: bool cond = !(X & 0xFFFF0000);
3: n |= (cond << 4);

4: unsigned short X1 = (cond) ?  X : X >> 16;
5: cond = !(X1 & 0xFF00);
6: n |= (cond << 3);

7: unsigned char X2 = (cond) ? X1 : X1 >> 8;
...
```

Although we have focused on the adder here, similar coding style and transformations were applied for all FP core types.

The divider unit is worthy of some elaboration, as several division algorithms are commonly used [9], each with its own strengths and weaknesses:

1) Digit-recurrence algorithm: produces one digit in each iteration, roughly as follows:
   a) Determine the next quotient digit.
   b) Multiply it with the divisor.
   c) Subtract it from the current partial remainder to obtain the remainder for the next iteration.
2) Functional iteration: uses Newton's method to find the value of *1/x*. It requires fewer iterations than the previous algorithm making it faster, but it generally requires more circuit area.
3) Polynomial approximation: like to functional iteration, yet using polynomial approximation to evaluate *1/x*, saving area.

For the divider unit, we employ a restoring iterative algorithm which falls under the first category.

All cores were functionally verified in software using millions of randomly generated operands. The cores were also verified after HLS via ModelSim simulation.

A. Ease of Modifiability

As all cores are specified in software, making changes to the cores is a straightforward matter of changing the `C` and re-synthesizing. In the next section, in addition to 754-compliant cores, we evaluate a core variant that lacks the functionality to detect and handle the FP special cases of $\pm\infty, \pm$ NaN (not a number), and ± 0.0, thereby saving circuit area. We believe such a core would be useful in application scenarios where a user knew in advance that exceptions would not arise. Creating the variant was a simple matter of commenting out the relevant lines of the software.

We also evaluate an adder core variant that uses non-754-standard bitwidth, namely, 4 bits for the exponent and 10 bits for the mantissa, permitting higher performance and lower area than a 754-compliant core, at the cost of reduced precision. We believe the ease with which such cores can be generated will be particularly useful in approximate-computing applications, such as machine learning [6], where full precision has been shown as unnecessary in many cases.

B. HLS Constraints

Apart from the code transformations above, we also apply HLS constraints. Loop pipelining is applied to each of the FP cores synthesized, so the resulting hardware core is able to accept new inputs every clock cycle (initiation interval (II)=1). We also applied LegUp's constraint for `if-conversion` [4], which invokes a compiler pass to remove control flow from the CFG, where possible. A key HLS constraint we applied for area reduction was LegUp's static bitwidth analysis and minimization [5]. The optimization reduces datapath widths by propagating constants forward/backward in the program's dataflow graph. We found it to be very effective in providing area reductions that arise from non-standard widths, e.g. 23-bit mantissa. For example, the optimization "sees" the shifts/masks in our code to extract the 23-bit mantissa, and from this, it "realizes" that computations involving the mantissa need not be full 32-bit width. The cores were compiled with `-O3` optimization.

In the experimental study, we apply various clock-period constraints to LegUp to synthesize a range of FP cores having different area/performance trade-offs (resulting from varying the degree of pipelining). For each core, we experimented with clock-period constraints ranging from 1-50ns.

V. Experimental Study

We compare HLS-generated floating-point cores with heavily optimized RTL cores from Altera and FloPoCo. Altera cores are IEEE 754 compliant, yet without support for subnormal numbers. FloPoCo cores are not IEEE 754 compliant, as they lack checks for floating-point exception cases, and also lack support for subnormal numbers. We used LegUp to generate two sets of floating-point cores. *LegUp (1)* corresponds to floating-point cores that are IEEE 754 compliant, without subnormal support. These are functionally equivalent and therefore comparable to the Altera RTL cores. *LegUp (2)* corresponds to floating-point cores without exception checking, making them equivalent to the FloPoCo cores.

All cores evaluated were targeted to the Altera Cyclone V 28nm FPGA using Altera's Quartus II CAD software v15.0. Aside from these two LegUp-generated cores, we also consider a core with non-standard reduced precision, as such cores are straightforward to specify by tweaking the software specification. Note that all cores considered, including Altera and FloPoCo, are *pipelined* having initiation interval of 1: new inputs can be injected every clock cycle. Cores with different area/delay trade-offs were created by specifying a target clock period constraint as input and allowing Altera, Flopoco and LegUp to select an appropriate pipeline depth for the circuit according to the target. When mapping to the Cyclone V FPGA, the same RTL synthesis constraints were applied in all cases, using the default Quartus settings.

We gauge speed performance using two metrics: 1) clock frequency ($Fmax$), and 2) pipeline depth. Silicon area evaluation is slightly more complicated. The Cyclone V FPGA contains adaptive logic modules (ALMs), DSP blocks, and block RAMs. ALMs in Cyclone V comprise a dual-output look-up-table with two flip-flops, capable to implement two independent 4-input logic functions, one six-input function, and other combinations. The DSP blocks can realize two 18×19 multiplies, three 9×9 multiplies, and other combinations. RAM blocks are 10Kb, distributed through the array, with configurable aspect ratio.

Fig. 2 shows results for floating-point adder cores. The horizontal axis reflects area (ALMs); the vertical axis reflects core speed (MHz). The pipeline depths of the fastest and smallest core in each category are shown numerically beside the relevant datapoint. Qualitatively, we observe that the speed performance of the LegUp cores (diamond and triangle points) falls within the same range of values as many of the RTL cores ($\times$ and square points), though the LegUp cores generally

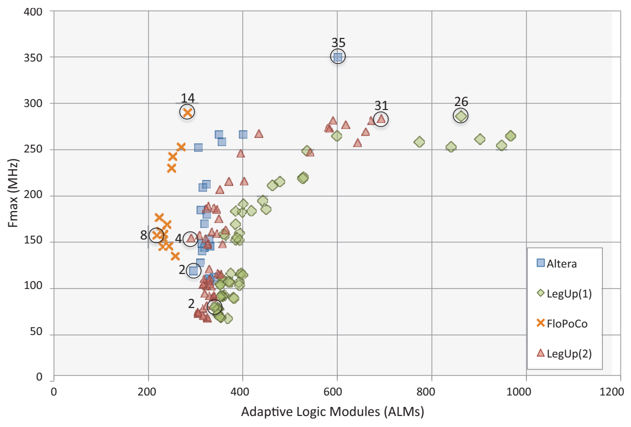


Fig. 2: Adder results; # pipeline stages shown for implementations with the best area/performance.

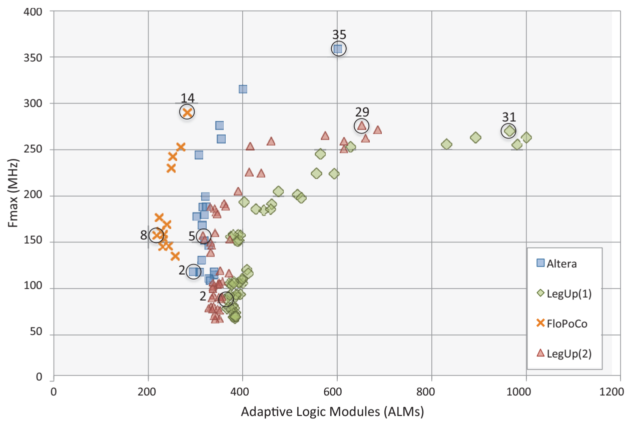


Fig. 3: Subtractor results; # pipeline stages shown for implementations with the best area/performance.

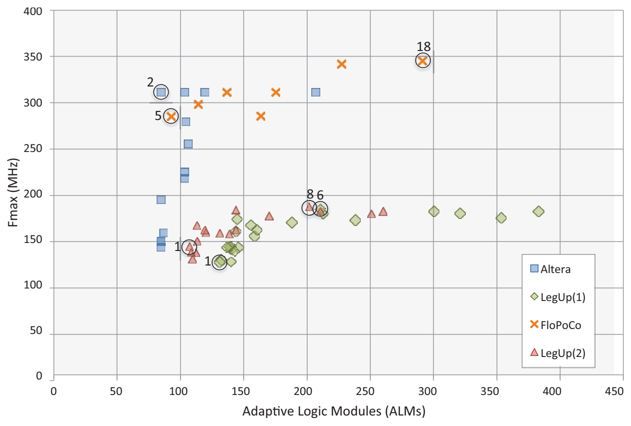


Fig. 4: Multiplier results; # pipeline stages shown for implementations with the best area/performance. Note: Altera cores use 1 DSP block; FloPoCo cores use 2 DSP blocks; LegUp cores use 3 DSP blocks.

consume more circuit area. Comparing LegUp (1) to Altera, we observe that the fastest Altera core operates at 349MHz, whereas the fastest LegUp (1) operates at 284MHz – about 19% slower. The fastest FloPoCo core operates at 288MHz, and the fastest LegUp (2) core operates at 282MHz – about 2% slower.

On the area front, the smallest Altera core consumes 300 ALMs, and the smallest LegUp (1) core consumes 345 ALMs (15% more). The smallest FloPoCo core consumes 221 ALMs and the smallest LegUp (2) core consumes 294 ALMs (33% more). Similar trends are observed for the case of subtraction (Fig. 3): the maximum speed of LegUp (1) cores are within 25% of Altera best-performing cores; the speed of LegUp (2) cores are within 5% of FloPoCo cores.

Overall, we find the results for addition/subtraction cores encouraging and surprisingly close to the established cores. Altera and FloPoCo cores are specified in RTL and heavily optimized for performance and area. In particular, we expect that the Altera cores are developed by expert IP design teams, specifically to leverage features of Altera FPGA architectures. Given the ease of core specification and customization in software, as well as the ability to fuse FP core functionality with other surrounding logic in HLS, we believe the performance/area gap between the HLS-generated and RTL adder/subtractor cores may be an acceptable trade-off for many users.

The results for multiplication cores are given in Fig. 4. Note that the Altera cores require 1 DSP unit; FloPoCo cores require 2 DSP units; and, LegUp cores require 3 DSP units, making it less straightforward to compare the area among the various cores. Certainly, the smallest LegUp cores offer a significant advantage in pipeline depth – 1 stages – as compared with Altera and FloPoCo at 2 and 5 stages respectively. In terms of $Fmax$, the RTL cores are clearly superior to the HLS-generated cores. The fastest LegUp (1) core is about 41% slower than the fastest Altera core. The fastest LegUp (2) core is about 47% slower than the fastest FloPoCo core. The HLS-generated cores contain wide combinational multiply operations that would need to be automatically split into narrower multiplies and pipelined to achieve higher $Fmax$.

The divider cores from Altera and FloPoCo require various combinations of ALMs, block RAMs and DSP units, and therefore, we present the area/performance results in tabular form in Fig. 5. The LegUp-generated cores require no DSP blocks or block RAMs, whereas the Altera cores use block RAMs and DSPs in all cases. The use of heterogeneous resources makes it difficult to directly compare the Altera and LegUp (1) cores for division. FloPoCo cores only use block RAMs in one case; many of the FloPoCo cores require no block RAMs or DSPs, making it possible to compare with LegUp (2) cores. The fastest FloPoCo core operates at 166MHz, as compared with the fastest LegUp (2) core, operable at 244MHz – 47% faster than FloPoCo. The smallest FloPoCo core that does not require RAM blocks uses 973 ALMs, compared with the smallest LegUp (2) core requiring 802 ALMs – 28% smaller than FloPoCo. Overall, division appears to be the most favorable result for the HLS-generated cores, superior in speed and area to FloPoCo.

Finally, Fig. 6 shows the results for the reduced-precision adder with 10-bit mantissa, 4-bit exponent, as synthesized

	ALMs	Fmax (MHz)	DSPs	RAM blocks	#pipeline stages
Altera	**580**	**300.66**	**4**	**5**	**29**
	414	230.1	4	5	19
	319	188.57	4	5	13
	280	150.15	4	5	11
	272	**124.86**	**4**	**5**	**10**
FloPoCo	**959**	**152.21**	**0**	**12**	**33**
	1059	158.55	0	0	23
	973	**165.98**	**0**	**0**	**13**
	1141	81.73	0	0	7
	1169	55.77	0	0	5
LegUp(1)	**2753**	**230.57**	**0**	**0**	**85**
	1114	177.53	0	0	28
	1053	157.33	0	0	21
	952	141.7	0	0	16
	837	**78.57**	**0**	**0**	**8**
LegUp(2)	**2158**	**243.61**	**0**	**0**	**82**
	1257	222.67	0	0	39
	876	184.81	0	0	26
	919	149.54	0	0	18
	802	**91.73**	**0**	**0**	**8**

Fig. 5: Divider results (fastest and smallest in each category are shaded).

by LegUp, labelled as *LegUp (3)* Observe that speed and area are considerably superior to both Altera and FloPoCo cores. Tremendous speed and area advantages are apparent for the reduced-precision core relative to all single-precision cores. The fastest reduced-precision adder is operable at 340MHz, compared with 349MHz (Altera), 288MHz (FloPoCo), 284MHz (LegUp (1)), and 282MHz (LegUp (2)). The smallest reduced-precision core requires just 171 ALMs, compared with 300 ALMs (Altera), 221 ALMs (FloPoCo), 345 ALMs (LegUp (1)), and 294 ALMs (LegUp (2)). We note that with Altera, one can generate a limited set of reduced-precision cores, subject to the constraint that the exponent ranges from 5 to 11 and the mantissa ranges from 10 to 52.

The software-specified cores were developed and optimized by a single junior engineer over a period of 16 weeks. The engineer had no knowledge of IEEE 754 floating point at the project onset, and we estimate that 8 of the 16 weeks was spent learning the details of the standard. We believe the productivity advantages afforded by HLS are significant in terms of reducing development costs and time-to-market. In summary, the proposed software-specified HLS-generated floating-point cores will be highly useful to achieve performance and area reductions in application scenarios where customizability is desired.

VI. Conclusions and Future Work

We considered the HLS of pipelined floating-point cores from a `C` software specification using the LegUp HLS tool. The HLS-generated cores were compared with optimized RTL cores from Altera and FloPoCo. In the case of addition and subtraction, the $Fmax$ of the fastest LegUp-generated cores fell within 25% and 5% of Altera and FloPoCo, respectively. For multiplication, the LegUp cores had considerably lower $Fmax$ than the RTL cores, however, the smallest LegUp-generated multipliers had comparable area to the RTL cores,

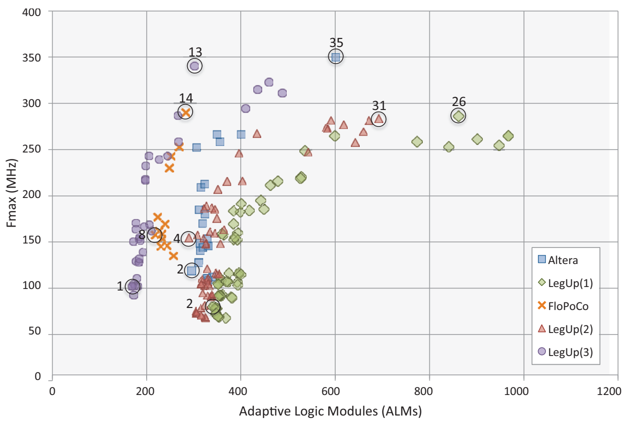


Fig. 6: Reduced-precision adder results; # pipeline stages shown for selected implementations

yet offered better pipeline latency: 1 stage vs. 2 and 5 stages in the RTL case. For division, the LegUp-generated cores were superior to FloPoCo cores in terms of area and $Fmax$. With software specification, it is straightforward to create cores customized to specific application scenarios, such as reduced precision, or with varying levels of exception handling. In the experimental study, a core with 10-bit mantissa and 4-bit exponent was synthesized and shown to offer appreciably better speed and area vs. the single-precision cores. Future work will investigate the benefits of using HLS-generated cores within larger applications, where portions of the core may be fused into the surrounding circuit.

References

[1] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *ACM FPGA*, pages 33–36, 2011.
[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April 2011.
[3] F. de Dinechin and B. Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, July 2011.
[4] B. Fort, A. Canis, J. Choi, N. Calagar, R. Lian, S. Hadjis, Y. T. Chen, M. Hall, B. Syrowik, T. Czajkowski, S. Brown, and J. Anderson. Automating the design of processor/accelerator embedded systems with LegUp high-level synthesis. In *IEEE EUC*, pages 120–129, 2014.
[5] M. Gort and J. H. Anderson. Range and bitmask analysis for hardware optimization in high-level synthesis. In *IEEE/ACM ASP-DAC*, pages 773–779, 2013.
[6] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.
[7] https://www.altera.com/documentation/eis1410764818924.html. *Altera Float-Point IP Cores Users Guide*, 2017.
[8] J. Matai, P. Meng, L. Wu, B. Weals, and R. Kastner. Designing a hardware in the loop wireless digital channel emulator for software defined radio. In *2012 International Conference on Field-Programmable Technology*, pages 206–214, 2012.
[9] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.
[10] R. Nane, V. M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels. Dwarv 2.0: A cosy-based c-to-vhdl hardware compiler. In *FPL*, pages 619–622, 2012.
[11] C. Pilato and F. Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *FPL*, 2013.