Practical Comparison of High-Level Synthesis and Hardware Generation Frameworks: CPU Floating Point Unit Case

Oleg Morozov[a], Alexander Antonov[a]

[a] *ITMO University, Kronverksky Pr. 49, bldg. A, Saint-Petersburg, 197101, Russia*

Abstract

The research is devoted to analyzing and matching advantages and drawbacks of various high-level design environments for the components of modern CPU cores. In the paper, high-level synthesis (HLS) and hardware generation frameworks (HGF) are compared for the case of floating-point execution unit (FPU). We use HGF-based FPU available in open-source SonicBOOM RISC-V CPU design from Berkeley as reference. Original HLS-based design of FPU module is proposed. This design is functionally equivalent to HGF-based one, but is described in behavioral (untimed) style, and its microarchitecture is optimized automatically by the HLS tool. The designed FPU has been synthesized in Vivado HLS and successfully tested in FPGA device. The research has shown that raising abstraction level up to behavioral one has provided the design with comparable frequency and resource characteristics, however, with significantly more concise design specification and automatic generation of microarchitecture. Based on these estimations, we envision HLS to be promising not only for accelerators that are external from CPUs, but also for selective, execution-centric components of modern CPUs themselves.

Keywords

High-level synthesis, hardware generation, hardware microarchitecture, floating-point unit, RISC-V

1. Introduction

Hardware designing based on register-transfer level (RTL) and corresponding design languages (SystemVerilog, VHDL) has been dominant in industry in the last decades due to efficient abstraction from basic structural devices (gates, multiplexers, etc.), understandable concepts by a wide community of developers, and good support by the design tools. However, time-to-market, cost, and complexity restrictions are motivating exploration of approaches to improve the design process. These improvements include support of algorithmic specifications as design entry, automation of microarchitectural synthesis from high-level specifications and configurations, and ensuring scalability of designs to meet various performance, power, and area constraints.

2. Theoretical background

2.1. High-level synthesis and hardware generation approaches

High-level synthesis (HLS) and hardware generation frameworks (HGF) are two widely known approaches to improvement of hardware design process. Despite some common priorities (abstract specification, improving configurability, utilizing software experiences in hardware domain), these approaches differ significantly.

Proceedings of the 12th Majorov International Conference on Software Engineering and Computer Systems, December 10-11, 2020, Online & Saint Petersburg, Russia
EMAIL: igelomori@gmail.com (A. 1); antonov@itmo.ru (A. 2)
ORCID: 0000-0003-2362-4361 (A. 1); 0000-0002-4596-9275 (A. 2)

High-level synthesis is typically understood as automated synthesis of hardware structure from behavioral (algorithmic), untimed specifications, effectively forming a new distinct abstraction level [1]. C/C++/SystemC programming languages are typically used as design entry. Microarchitectural synthesis is performed by the tool automatically, and, though it is directed to a certain extent via pragmas and constraints, design entry is abstracted from it. Majority of HLS tools perform a typical set of operations, including allocation of basic functional units, scheduling of operations regarding their dependencies and time constraints, and binding of these operations to allocated functional units. Optimizations are applied to programmatic models (such as Control and Data Flow Graph, CDFG). Shorter design cycle using behavioral synthesis allows many alternative circuit implementations to be explored, enlarging design space for better implementations.

Hardware generation frameworks improve RTL designing via exposing its abstractions (registers, modules, combinational circuits, etc.) to general-purpose programming environments. Typically, they are implemented as an embedded domain-specific language (eDSL), i.e. as a library. Unlike HLS, microarchitectural synthesis is not abstracted in design entry, but can be embedded in multiple custom generators. HGFs provide feature-rich environment for specification of RTL generation, offering programmatic construction of hardware, improving flexibility in defining and processing of configurations, layering new eDSLs, etc. Facilitation of programming generators instead of “fixed” designs enables deep adaptation of the hardware to the project needs and constraints. RTL-like models (such as FIRRTL) are typically used as intermediate representations for application of optimizations.

With their advantages and drawbacks, both HLS and HGF approaches have gained significant traction in academic and industrial designing. However, their typical application domains have some variations. Though HGF is more like a general-purpose approach (similar to generic RTL), it still requires digital design expertise from the designers. Also, the designers should be simultaneously programming experts and know the details of how RTL abstractions are embedded in certain HGF. HLS (ideally) does not require the designer to be a hardware expert, but targets acceleration co-processors with static scheduling of operations and pipelined microarchitecture. As a result, HLS is not usually positioned for designing hardware units with custom and dynamic scheduling of computational process, including CPUs. Even simple, in-order implementations suffer from suboptimal performance, mostly because of conservative, static branch scheduling [2].

To adopt HLS for CPU-like hardware applications, the following strategies can be implemented [3].

Definition of microarchitecture explicitly in high-level language. To reflect dynamic scheduling mechanisms, they can be explicitly programmed in high-level language. For CPU applications, these mechanisms can include dynamic speculation, instruction reordering, data forwarding, stalling, etc. Though this approach does not impose restrictions on complexity of these mechanisms (custom ones can be freely included as well), this approach effectively lowers the design level, transforming behavioral approach into microarchitectural one. Expertise in hardware microarchitecture is required to implement this approach.

Allocation of statically scheduled structural units and designing them separately in high-level environment. Though this approach requires hardware microarchitecture expertise for allocation of these units and their integration, these units themselves can be extracted for abstract high-level definition of their behavior and automation of their optimization. For CPU applications, “computational” execution pipelines (integer, floating-point, DSP, custom ones) can hypothetically be good candidates for such extraction, since even in complex out-of-order microarchitectures operations are issued to such units when the data operands are ready, and the number of clock cycles needed does not depend on other CPU subsystems [4]. In this paper, we explore the case of floating-point unit – an important mathematical CPU block that was often implemented as external co-processor in the past, and now is typically a part of CPU die and can occupy more that 10% of chip area [5].

2.2. CPU floating point unit functionality

CPU floating-point unit (FPU) provides basic operations for numbers represented in floating-point format. The common format for single precision floating-point number is defined by IEEE-754 standard [6]:

$$(-1)^S * M * 2^E, \tag{1}$$

where S stands for sign, E is exponent, and M is mantissa. The binary IEEE-754 representation defines a 32-bit word, with one bit for sign, 8 bits for exponent, and 23 bits for mantissa. As a basic set of floating-point operations, we use those defined in RISC-V architecture – a modern and open instruction set architecture being widely used both in academia and industry in recent years. An extension that includes floating-point operations on single precision numbers is denoted RV-F, which derives from the name of the "Float" data format. RISC-V uses 32 registers for floating-point numbers, denoted f0 – f31, with a size of 32 bits each. FPU works with both a separate floating-point register file and a common register file. Therefore, the module must accept and return data in both float and integer formats.

Table 1 gives a summary of these operations.

Table 1
Floating-point operations defined in RISC-V architecture

Operation	Description
FADD, FSUB, FMUL, FMIN, FMAX	Arithmetic functions, input and output are float
FSGNJ, FSGNJN, FSGNJX	Sign-injection instructions, input and output are float
FEQ, FLT, FLE, FCLASS	Comparison operations, input is float, output is integer
FCVT.W.S, FCVT.S.W, FCVT.WU.S, FCVT.S.WU	Transfer operations from float to integer and vice versa.
FMADD, FMSUB, FNMSUB, FNMADD	Floating-point fused multiply-add instructions, input and output are float

3. Design of HGF-based FPU in BOOM

SonicBOOM is the third iteration of Berkeley Out-Of-Order Machine (BOOM) project. BOOM is a high-performance, synthesizable and parameterizable RV64GC RISC-V core, which means it supports multiplication and division extensions, atomic, single and double precision floating point operations, and short instructions. BOOM is currently one of the most complete and productive open-source RISC implementations and demonstrates the use of the main contemporary mechanisms such as superscalar processing of instructions, speculation, branch prediction, cache memory, etc. The core is designed based on Chisel hardware generation framework.

Chisel allows to flexibly construct class hierarchies of modules for various templates and communication mechanisms with the rest of the system (see Fig. 1).

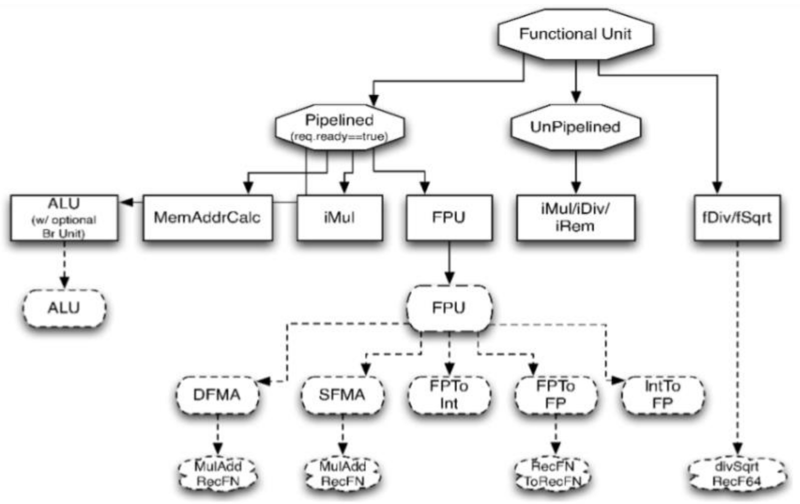

Figure 1: Class hierarchy for functional units of SonicBOOM RISC-V CPU [7].

In BOOM, execution of a floating-point instruction occurs in two different modules: fDiv/fSqrt for calculating the square root and division, and the FPU module that executes all other instructions. For simplicity, only FPUs without fDiv/fSqrt will be considered.

BOOM's FPU consists of for subblocks: sfma for single-precision operations, dfma for double-precision operations, fpiu for fp-to-int operations, and fpmu for fp-to-fp operations. Calculation algorithms are specified in "combinational" style and successively copied in register chains using Chisel's Pipe primitive with configurable delay. After EDA tool applies retiming, fully pipelined implementation with initiation interval of one clock cycle is obtained. To simplify write port processing, the delay is set to the same value for all subblocks.

BOOM uses interfaces and modules from the RocketChip processor core, which uses interfaces and modules from the Hardfloat core. Clock and reset signals are specified implicitly.

The interface consists of two buses, the output ExeUnitResp and the input FpuReq. ValidIO is a built-in Chisel function that implements the creation of an interface with the valid enable signal and the specified bus type. The output interface resp has type ExeUnitResp, the standard interface for all BOOM function blocks. ExeUnitResp consists of a data bus and a ValidIO bus with flags. The flag bus is specified in the same execution unit file and consists of a MicroOp bus for transmitting service information and a flags for Floating Point exception flags from the RISC-V specification. The flags are part of the FCSR register.

The Input interface req consists of the valid FpuReq interface. It has a MicroOp bus, three buses for transferring data from floating-point registers and one 5-bit bus for transferring the value of the exception flags.

Generation of certain FPU implementation is controlled by 4 parameters:

- minimum instruction length,
- maximum instruction length,
- arithmetic block latency based on SFMA operations,
- arithmetic block latency based on DFMA operations.

In Fig. 2, the configuration used for FPU implementation is shown.

```
case class FPUParams(
      minFLen: Int = 32,
      fLen: Int = 64,
      …
      sfmaLatency: Int = 3,
      dfmaLatency: Int = 4)
```
Figure 2: FPU configuration used for generation of implementation.

Using SonicBOOM generator, FPU implementation has been generated and implemented for educational Digilent Nexys4-DDR board with Artix-7 FPGA device. We used Vivado 2020.2 for this task. Resulting characteristics have been compared to a similar implementation synthesized using Vivado HLS tool (see subsequent Sections).

4. Designing a FPU module with an HLS tool

4.1. Designed behavioral model of FPU

To compare the reference HGF-based design to HLS-generated one, functionally equivalent unit for HLS has been designed. According to HLS methodology, HLS-based design is a software function that specifies solely the behavior of the module and does not fixate its microarchitecture (see Fig. 3).

```
return_floats FPU(t_floats val){
    return_floats val_out = inizialize();
    if (val.funct3 == 0 && val.funct7 == 0)
        val_out.rd_f = val.rs1 + val.rs2;
    else if (val.funct3 == 0 && val.funct7 == 4)
        val_out.rd_f = val.rs1 - val.rs2;
    else if (val.funct3 == 0 && val.funct7 == 8)
        val_out.rd_f = val.rs1 * val.rs2;
    else if (val.funct7 == 16)
        val_out = FSGNJ_FSGNJN_FSGNJX(val, val_out);
...
     val_out = FCVTWS_FCVTSW_FCVTWUS_FCVTSWU(val, val_out);
    else if (val.funct3 == 1 && val.funct7 == 112)
        val_out = FCLASS(val, val_out);
    else
        val_out.err = 0;
    if (isnan(val_out.rd_f) != 0)
        val_out.nan = 1;
    return (val_out);
}
```

Figure 3: Behavioral FPU design for Vivado HLS (similar code fragments are omitted).

The structure of the designed block is implemented as a branching function, where an operation is selected based on the func7 and func3 RISC-V instruction fields, as well as the value of the rs2 operand.

There are four sub-functions: calculating the equality operation FEQ, branching for sign change operations FSGNJ/FSGNJN/FSGNJX, format change operations FCVTWS/FCVTSW/FCVTWUS/FCVTSWU and defining the type of variable FCLASS.

Input and output signals are specified as structures. The input structure includes:

- floating-point operands
- integer operand
- funct7 and funct3 RISC-V instruction fields

The output structure includes:

- floating-point result
- integer result
- instruction error flag
- NaN flag

The functions signbit, copysignf, fabsf, fpclassify, islessequal, isgreaterequal, isnan from the C library “math.h” were used. Compared to native C functions, the math.h library functions can reduce the use of LUT by 40%, FF by 50%, and achieve a higher clock speed by 62.5%.

HLS-based implementation has also been synthesized to RTL, implemented and tested in hardware on Digilent Nexys4-DDR FPGA board.

4.2. Hardware test infrastructure

To provide interactive control, observation and debug capability for designed FPGA modules from PC programming environment, custom infrastructure has been used.

The key element in this infrastructure is UDM (UART-based Debug Module) FPGA module (see Fig. 4). This module can initiate simple bus transactions in FPGA fabric under the control of PC program. UDM is managed via UART interface that is lightweight, easy to implement, and available in all FPGA boards. The protocol working between UDM and PC allows to initiate transactions and receive responses. This allows PC to “emulate” CPU host in custom system-on-chip designs. On PC, UDM is supported in Python 3 environment. Read or write function calls on PC become requests appearing on UDM system bus.

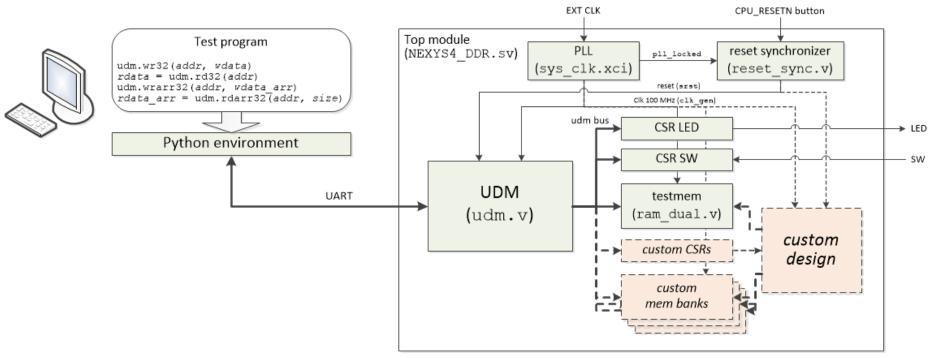
```
udm.wr32(addr, wdata)
rdata = udm.rd32(addr)
udm.wrarr32(addr, wdata_arr)
rdata_arr = udm.rdarr32(addr, size)
```

Figure 4: Infrastructure for interactive hardware testing of custom FPGA-based designs.

UDM module consumes minimum amount of hardware resources (<1% of LUTs and flip-flops on Artix-7 FPGA device), can be implemented in minutes, and requires minimum setup (restricted to COM port number definition).

Table 2
CSRs allocated for FPU hardware testing.

Address	Mnemonic	Description
0x08	FPU_START	Enabling signal
0x0C	FUNC7	Unsigned 7-bits RISC-V instruction field
0x10	FUNC3	Unsigned 3-bits RISC-V instruction field
0x14	RS1 [31:0]	First source floating-point register value
0x18	RS1 [63:32]	
0x1C	RS2 [31:0]	Second source floating-point register value
0x20	RS2 [63:32]	
0x24	RS3 [31:0]	Third source floating-point register value
0x28	RS3 [63:32]	
0x2C	RSI	Source integer register value
0x30	RESULT [31:0]	Result floating-point register value
0x34	RESULT [63:32]	
0x38	RESULT_I	Result integer register value
0x3C	FLAG_NAN	Flag indicating that the result is NAN
0x40	FLAG_ERROR	Flag indicating that the function code is invalid

For HLS-based FPU, test several control and status registers (CSRs) have been allocated (see Table 2). These registers have been connected to the FPU and UDM system bus. Each test iteration sends the instruction number, the values of the operands, then starts the FPU and reads the error flags and the result values.

5. Comparison of HLS and HGF based implementations

Resulting characteristics for HGF-based and HLS-based implementations are shown in Table 3.

Table 3
Comparison of HGF and HLS based implementations.

Characteristics	HGF-based module (reference)	HLS-based module (designed)
Top frequency	92 Mhz	136 Mhz
Initiation interval	1 clock cycle	1 clock cycle
Latency	4 clock cycles	10 clock cycles
LUT	4738	3441
Flip-flops	1454	2929
DSP	11	26
Lines of code	230 (+1200 in HardFloat)	120

It can be seen that the modules have the same initiation interval of one clock cycle, comparable frequency and resource characteristics.

HLS-based implementation is faster, but has bigger latency. According to our experiments, restricting maximum latency is impractical, since it is possible only with close to fold reduction of frequency. This makes absolute latency almost the same, but reduces bandwidth.

Also, HLS-based implementation consumes less LUTs, but more flip-flops and DSP blocks. While DSP utilization (at the expense of general-purpose LUTs) is predictably better for high-level environment, more than two-fold consumption of DSPs requires additional investigation. Increased flip-flops consumption of HLS-based implementation is likely due to deeper pipelining.

When it comes to design specification mechanisms, for HLS, as well as for HGF, it is possible to set custom latency. In HLS this is possible through the use of pragmas, while in HGF it is done through explicit parameterization of the pipeline. Actually, reference HGF-based implementation heavily relies on retiming in lower-level RTL synthesis tool. In HLS, since pragma is a synthesizer directive, it is easier to change the computation schedule with this method, rather than directly adding parameters to the module structure. However, since the synthesis is carried out automatically by the tool, the desired result in HLS must be achieved heuristically.

To sum up, designing CPU execution units in high-level synthesis looks promising to implement high-level, easily extendable, scalable CPU projects, while preserving sufficient quality-of-results.

6. Future work

In the future, the research is planned to develop in the following directions:

1. The designed HLS-based module is supposed to be integrated in Rocket and/or BOOM project and validated as part of actual RISC-V CPU;
2. In-depth exploration of the synthesized netlists in HGF and HLS projects and identification of the discrepancies in their structures;
3. Experimental explicit programming of floating-point computation algorithms in synthesizable C/C++ instead of relying on HLS tool to synthesize this logic;
4. Exploration of floating-point capabilities in alternative high-level tools, including open-source ones (LegUp [9], GAUT [10]);
5. Exploration of feasibility of high-level synthesis tools for alternative CPU execution pipelines (integer, DSP, custom ones);
6. Exploration of high-level execution units design targeting ASIC devices.

7. Conclusion

Raising abstraction level, improving configurability of component base and adopting various design techniques from software domain is often considered inevitable in hardware designing to satisfy hardware project constraints at the moment and in the future. Despite the recent improvements in RTL design offered by hardware generation frameworks, design specification on behavioral level seems especially promising. However, this transition should be done with regard to quality of results, which may not be sufficient for the entire diversity of hardware.

Using the example of CPU floating-point execution unit, we are showing that comparable implementation results for selected elements of CPU can be achieved on behavioral level and using automatic synthesis of the unit's microarchitecture. This motivates further comparative exploration of configurability and efficiency of HGF and HLS environments for execution-related and other selected subsystems of modern CPUs, as well as other complex hardware projects.

8. Acknowledgements

The work has been done in Software Engineering and Computer Systems Faculty of ITMO University. Design of hardware test infrastructure for interactive control, observation and debug of custom hardware modules based on FPGA devices (conducted by A. Antonov) has been supported by Russian Science Foundation, grant № 20-79-00219.

9. References

[1] M. Fingeroff, High-Level Synthesis Blue Book. Xlibris Corporation (2010).

[2] S. Skalicky, T. Ananthanarayana, S. Lopez, and M. Lukowiak, Designing Customized ISA Processors using High Level Synthesis. In: International Conference on ReConFigurable Computing and FPGAs (ReConFig), pp. 0–5 (2015).

[3] A. Antonov, Methods and Tools for Computer-Aided Synthesis of Processors Based on Microarchitectural Programmable Hardware Generators, Ph.D dissertation, ITMO University, Saint-Petersburg, http://fppo.ifmo.ru/dissertation/?number=63419, last accessed 2019/05/27.

[4] J.P. Shen, M.H. Lipasti, Modern Processor Design: Fundamentals of Superscalar Processors. Waveland Press (2013).

[5] Hwa-Joon Oh, et al., A Fully Pipelined Single-Precision Floating-Point Unit in the Synergistic Processor Element of a CELL Processor. IEEE Journal of Solid-State Circuits, Vol. 41, No. 4 (2006).

[6] IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008, pp. 1-70 (2008).

[7] RISCV-BOOM's documentation, URL: https://docs.boom-core.org/en/latest/sections/execution-stages.html, last accessed 2020/11/14.

[8] A. Antonov, ActiveCore, URL: https://github.com/AntonovAlexander/activecore, last accessed 2020/11/14.

[9] A. Canis, et al., LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. In: Trans. Embed. Comput. Syst., vol. 13, no. 2 (2013).

[10] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, E. Martin, GAUT: A High-Level Synthesis Tool for DSP Applications, From C algorithm to RTL architecture. In: High-Level Synthesis, pp. 147–169, Eds. Springer Netherlands (2008).