Low-Area, High-Throughput Field-Programmable Gate Array Implementation of Microprocessor Without Interlocked Pipeline Stages

Prateek Sikka, Abhijit R. Asati, and Chandra Shekhar

1 Introduction

The processor architecture for microprocessors without interlocked stages (MIPS) developed by MIPS Technologies and Imagination Technologies has recently evolved from a 32-bit version to a 64-bit version. Compared with ARM processors, the MIPS core is compact, requires a smaller die size, and consumes less power, all while offering multi-threading capabilities, which increases its functionality. MIPS processors are often used in applications involving consumer audio devices, such as audio players, set-top boxes, DVD recorders and players, and digital displays, which are typically implemented with a multifunction system on chip.

Given that low power and high speed are important goals in these applications, reduced instruction set computer (RISC) architectures are preferred. The instructions are simple in RISC, and each instruction requires a similar number of clock cycles, making it easy to pipeline the instructions, thereby obtaining high throughput. Field-programmable gate arrays (FPGAs) are also popular as platforms for pre-silicon prototyping to accelerate verification and software development. Thus, exploiting low-area FPGAs to accelerate the implementation of MIPS processors is of significant importance.

Numerous research efforts have focused on MIPS architecture in the past. In 2019, Indira et al. implemented a 32-bit MIPS processor and targeted the same on a Xilinx Virtex 7 FPGA [1]. They also discussed possible pipeline hazards and the associated remedies. In 2017, Rashidah et al. proposed a simulator for the RISC-16 instruction set [2] that was based on visual basic programming and five pipeline stages. In 2016, Husainali et al. proposed a three-stage, 32-bit pipelined processor [3] that they designed in Verilog and implemented on a Xilinx Virtex 7 FPGA using Xilinx ISE

P. Sikka (✉) · A. R. Asati · C. Shekhar
Electrical and Electronics Engineering Department, Birla Institute of Technology and Science, Vidya Vihar Campus, Pilani, Rajasthan, India

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2022

A. Dhawan et al. (eds.), *Recent Trends in Electronics and Communication*, Lecture Notes in Electrical Engineering 777,
https://doi.org/10.1007/978-981-16-2761-3_58

software. In 2018, Mangalwedhe et al. proposed a low-power RISC processor [4] that they designed in Verilog. They used clock gating to reduce the dynamic power consumption and implemented the design on a Spartan 6 FPGA.

In 2014, Rakesh et al. proposed a novel architecture for a 17-bit address RISC processor [5]. They implemented their Harvard architecture-based design on a Xilinx FPGA. In the present work, we use high-level synthesis (HLS) to design a MIPS core and implement it on a Xilinx Virtex 7 FPGA target, and we compare this implementation with previous implementations. In the proposed design, we use multiple HLS directives to reduce the area and increase the speed. The results of the proposed FPGA are clearly superior to those of previous FPGAs.

The rest of the paper is organized as follows: Sect. 2 introduces HLS, and Sect. 3 presents the architecture of the MIPS processor that we designed and explains the method used in the proposed design. In Sect. 4, we discuss the techniques used to optimize the synthesis results. Specifically, we discuss in detail the HLS directives used during the design process, which lead to the optimal synthesis between design and the target FPGA. Section 5 presents the results of the simulation and synthesis and compares the results with those of other works for the same application. Finally, Sect. 6 concludes the paper.

2 High-Level Synthesis

HLS is gaining popularity in the design community as a method that ensures continued verification in the design flow and increasing the level of abstraction that designers can use to describe the design behavior. This method of code generation is free from errors and is faster than manual register transfer language (RTL) coding [6]. HLS tools like Vivado HLS [7] and MATLAB HDL [8] coder are commonly used in the design community to design and prototype algorithms that target different application areas, such as image processing, computer vision, and microprocessors. The code complexity is reduced by almost an order of magnitude, and reuse of behavioral IPs across projects is simplified by using modeling techniques in HLS, such as transaction-level modeling [9].

For modern system on chip designs, especially those containing embedded processors running firmware, the use of high-level programming languages in the automated HLS process enables designers and architects to explore area, power, and throughput trade-offs using different hardware–software boundaries. The industrial focus on HLS tools gained importance with the enhancements to RTL-based synthesis tools and flows. Proprietary tools were introduced by major chip design houses, such as Motorola [10], IBM [11], Philips [12], and Siemens [13]. Major electronic design automation (EDA) companies have also commercialized their HLS tools in the past few years. In 1995, Synopsys introduced the behavioral compiler [14] that generates synthesizable RTL implementations from C code and connects with downstream synthesis tools. Mentor Graphics came out with Catapult HLS [15], and Cadence introduced Stratus HLS [16].

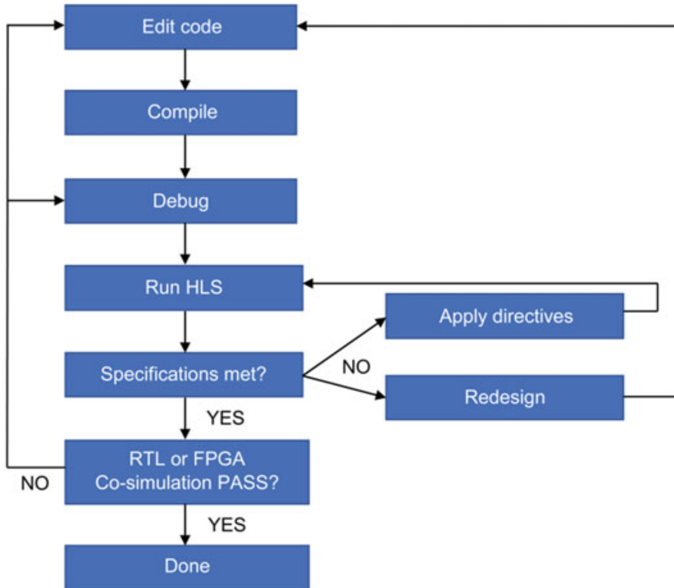

Fig. 1 High-level synthesis flow for very large scale integration designs

Figure 1 shows a typical HLS flow for very large scale integration (VLSI) designs. The HLS tool extracts all the parallelism that is accessible from the input description and schedules the operations. The next step includes allocating and sharing the necessary resources and optimization to minimize the area and improve performance. During these intermediate transform stages, different optimization directives can be applied to guide the HLS tool to meet the design specifications for decreasing area, increasing speed, and reducing power consumption. Within the HLS flow, tools typically perform the following functions: compile the specifications, allocate hardware resources (functional units, storage components, buses, etc.), schedule the operations to clock cycles, bind the operations to functional units, bind variables to storage elements, and bind transfers to buses.

Although the overall design cycle time during algorithm development may increase, the subsequent design implementation cycles are quicker, so the time to market is reduced. As seen in Fig. 1, not satisfying the desired specifications may trigger multiple iterations for HLS directives in the design flow.

3 Architecture of MIPS Processor

The primary goal in VLSI design has always been to reduce the power consumption of devices (specially mobile), such as laptops, mobile phones, and tablets, so that they would support real-time applications in video, image processing, telecom,

networking, etc. Different types of integrated circuits provide different complex signal processing units to fulfill the various demands of complex use cases such as mathematical computations. When it comes to real-time operation of these integrated circuit designs, speed and power dissipation are the major bottlenecks. Thus, the major aim here is to obtain computational speed and decrease power consumption.

To satisfy this requirement, the MIPS processor was proposed by MIPS Technologies in the 1980s. To date, six versions of the MIPS have been released (denoted I–VI), with the current version (VI) having been released in 2017. Earlier versions had only 32-bit support, but the later versions support 64 bits. The MIPS architecture is also known as load-store architecture because, except for memory access, all instructions operate on registers. The salient feature of a MIPS core is its use of the RISC architecture with a non-interlocked five-stage pipelining technique to reduce delay [17].

Pipelining has proven to be more efficient than traditional sequential architecture because of that fact that CPU becomes idle during instruction cycles that include other services such as read and write to memory and storage or input–output devices. This is clearly evident from Tables 1 and 2, which show the fetch, decode, and execute cycle with and without the pipelining for multiple instructions, and the associated throughput.

All MIPS instructions are 32 bit long. The instruction set is a compiler-based encoding of the machine instructions (i.e., code generation efficiency is used to choose alternative instructions). Multiple simple instruction sections are packed together into an instruction word. To do several operations simultaneously in pipelining, the simultaneous implementation of devices such as memory, integer units, and other units is essential. Pipelining involves five stages: fetch, decode, execute, memory, and write back.

- Fetch: In this stage, we fetched the instruction from the memory based on the address provided in the program counter (PC). Incrementing by four to the

Table 1 Instruction execution without pipeline; the throughput is two instructions per clock cycle

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6
Instruction 1	Fetch	Decode	Execute			
Instruction 2				Fetch	Decode	Execute

Table 2 Instructions execution with pipeline; the throughput is four instructions every six clock cycles

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6
Instruction 1	Fetch	Decode	Execute			
Instruction 2		Fetch	Decode	Execute		
Instruction 3			Fetch	Decode	Execute	
Instruction 4				Fetch	Decode	Execute

previous instruction address may update the PC. Alternately, it can be updated by a branch address provided by conditional or unconditional branch instructions. The instruction is fetched from the memory and relayed on to the next stage:

$PC \le (PC + 4)$,
$PC \le$ Branch address.

- Decode: In instruction decode, the opcode is decoded by the decoder unit. The register bank is also present in this stage to find the effective address. When in indirect-addressing mode, two clock cycles are required. For different addressing modes, operands are identified through the register bank and handed to the next stage for computation.
- Execute: All logical and arithmetic operations are performed in this stage. The operation performed on the data are dependent on the control signals generated in the decode stage for each instruction [2]. The integer unit consists of the arithmetic logic unit (ALU), shifter, and multiplier devices. In addition, the effective addresses used for load and store operations are computed.
- Memory: All memory related instructions are executed in this stage. Depending on the control signals, instructions such as load and store are performed to read and write the data.
- Write back: The registers are updated in this stage. The corresponding data are updated in the register array located in decode stage. The data may be coming from a memory location or another register.

MIPS cores have three types of instructions: I-type, J-type, and R-type (see Table 3). The six most significant bits for all types specify the opcode to select the type of instruction. R instructions are used when all the values used by the corresponding instruction are used from registers. I instructions are used when the instruction operates on a register and an intermediate value. Immediate values may be up to 16 bits long; larger number values cannot be manipulated by immediate instructions. J instructions are used for jumps; it has the most bits available for an immediate value to be stored because addresses are typically large numbers.

Table 3 Instruction types for MIPS architecture

<table>
<tr><td colspan="6">R-type instructions</td></tr>
<tr><td>Op (6 bits)</td><td>Rs (5 bits)</td><td>Rt (5 bits)</td><td>Rd (5 bits)</td><td>Shamt (5 bits)</td><td>funct (6 bits)</td></tr>
<tr><td colspan="6">I-type instructions</td></tr>
<tr><td>Op (6 bits)</td><td>Rs (5 bits)</td><td>Rt (5 bits)</td><td colspan="3">Address/immediate (16 bits)</td></tr>
<tr><td colspan="6">J-type instructions</td></tr>
<tr><td>Op (6 bits)</td><td colspan="5">Target address (26 bits)</td></tr>
</table>

4 Design Methodology

The end-to-end processor model for the MIPS core was created by using Simulink with MATLAB function blocks. Figures 2, 3, and 4 show the top-level CPU (Controller + Memory), MIPS datapath, and controller, respectively. The design is optimized by applying directives such as loop unroll and pipelining (three stages). An instruction parser operates within the datapath and consisted of the opcode, source register, destination register, immediate operand, and jump address. The parser is directly linked to a sign extend block and a jump calculator block. A 32-bit register file is also available which consists of one write port and two read ports. The ALU RESULT consists of three inputs: ALU control, Scr A, and Scr B, and this block performs four major operations on the input: addition, subtraction, AND, and OR

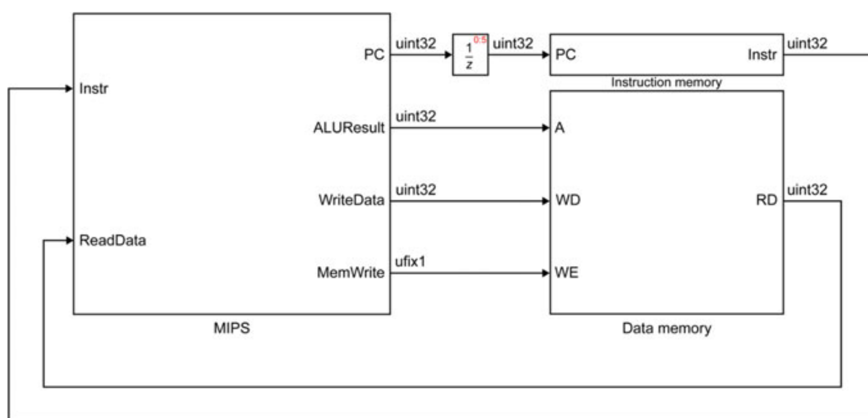

Fig. 2 Top-level implementation of processor system with memory

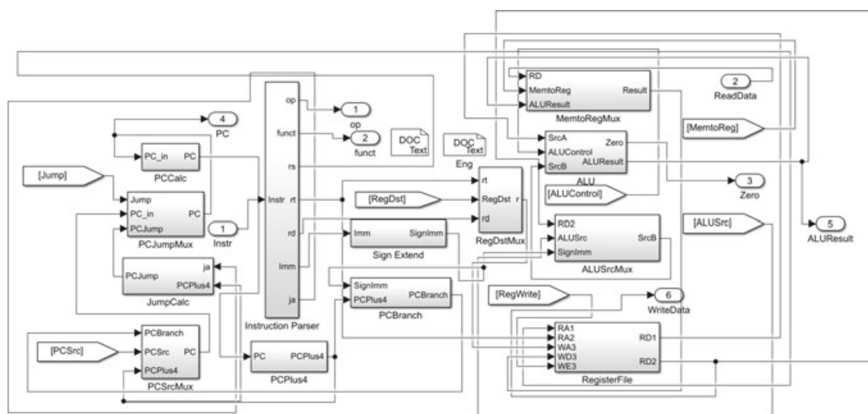

Fig. 3 MIPS data path model

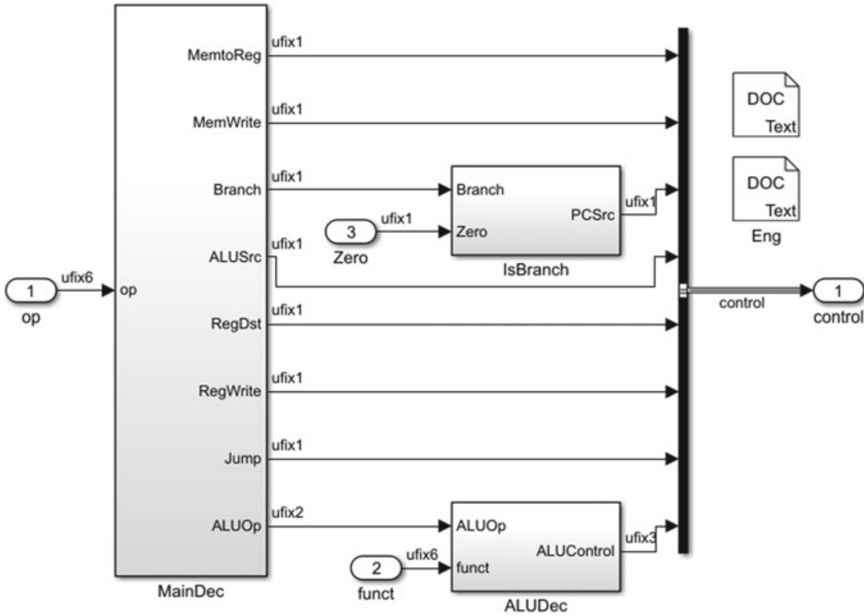

Fig. 4 MIPS controller

between the Scr operands A and B. The register file provides Scr A, and the Scr B output data are obtained from the sign extended immediate value. The three-bit ALU control specifies the operation to perform on operands while the ALU generates a 32-bit result and a zero flag (to indicate if ALU Result = zero). The ALU Scr multiplexer is used to handle the R-type instructions, which write the ALU Result to the register file. Therefore, we add this multiplexer to select between Read Data and ALU Result and call the output as "Result." This multiplexer was controlled by the signal "Mem to Reg," which is zero for R-type instructions to choose Result from the ALU Result, and unity for lw to choose Read Data. After the base implementation is created, we use the HLS tool MATLAB HDL coder to convert the MIPS core to synthesizable Verilog code and subsequently run it through FPGA synthesis using Xilinx Vivado.

As a second step, we apply the following HLS directives to optimize the results of the MATLAB HDL coder:

(A) Pipeline. The HLS directive allows the concurrent execution of operations by reducing the initiation interval for a loop or function, so a trade-off exists between area and speed. To optimize this CPU design implementation, we apply a pipeline directive with an initiation interval of two for all loops in the design.

(B) Loop unroll allows the loop iterations to run in parallel by creating multiple copies of the same loop body in the generated RTL. This directive helps to increase the throughput by making the loops either partially or fully unrolled.

Table 4 Resource use for FPGA implementation; maximum synthesis frequency = 420.028 MHz, power = 0.0233 W; LUT: lookup table, IOBs: input–output blocks, BUFG: global clock buffer

Resource	Resources used	Total resources
Slice registers	43	408,000
Slice LUTs	178	204,000
Fully used FF pairs	41	178
Number of bonded IOBs	47	600
Number of BUFG	1	32

For the proposed application of the MIPS controller and datapath, we use the partial unroll directive to improve design performance, which incurs a minor trade-off on resource usage. An unroll factor of two is used in the implementation.

Section 5 presents detailed implementation results for the base implementation and the implementation after application of the HLS directives.

5 Results

5.1 Simulation Results

After generating the RTL code, we performed an RTL simulation for the design using a non-synthesizable Verilog test bench in xSim software. The simulation results were identical to the simulation results obtained using a high-level simulation in Simulink.

5.2 Results of FPGA Implementation

We implemented the generated RTL from the HLS model on a Xilinx Virtex 7 FPGA board (7vx330tffg1157-3). Table 4 summarizes the implementation results for the target FPGA device.

5.3 Comparison of Results

As can be seen from Tables 4 and 5, although the proposed implementation operates at almost the same operating frequency as that of Indira et al. [1], its resource usage is 40–50% less. In addition, the proposed implementation is about fourfold faster than that proposed by Rakesh et al. [5], with almost the same resource use. These

Table 5 Comparison of FPGA implementation results

Metric	Proposed implementation	Indira et al. [1]	Rakesh et al. [5]
Slice registers	43	81	56
Slice LUTs	178	321	203
Fully used flip flops	41	81	43
Bonded IOBs	47	71	51
BUFG	1	2	1
Power (W)	0.021	0.0233	1.318
Maximum frequency (MHz)	404.1	420.028	100

improvements are attributed to the use of HLS directives in the proposed implementation, which produces better results in terms of both area and speed of operation (i.e., synthesis frequency).

6 Conclusion

Over the past few years, MIPS processor architectures have evolved as an important choice for multiple computing applications, such as communication and information processing. Being built from a RISC architecture, MIPS are friendly for pipelining instructions and thus provide faster throughput for targeted applications. Additionally, to reduce time to market for VLSI designs, FPGAs have also become popular as platforms for pre-silicon software development and accelerated verification. Therefore, the optimal FPGA implementation of MIPS cores is vital for obtaining optimized designs.

This paper proposes a resource optimal, high-throughput implementation of a MIPS core on a Virtex 7 FPGA. The proposed design was created using Simulink and was implemented using the MATLAB HDL coder and Xilinx Vivado. The design targeted Virtex 7 so that the results could be compared directly with those of other studies. We optimized the targeted implementation for performance and resource usage using appropriate HLS directives for pipelining and loop unrolling. After applying the directives, the final implementation results proved superior in terms of throughput and target area on the FPGA. The results of FPGA synthesis clearly indicate that the proposed implementation is superior to previous implementations, despite having exactly the same design specifications. The implementation results can be further improved using one or more HLS directives, which will be explored in future work.

References

1. O. Indira, V.V. Dwivedi, M. Kamaraju, Verilog implementation of a MIPS RISC 32-bit pipelined processor architecture. IOSR J. Electron. Commun. Eng. **14**, 31–40 (2019)
2. R.F. Olanrewaju, F.E. Fajingbesi, S.B. Junaid, R. Alahudin, F. Anwar, B.R. Pampori, Design and implementation of a 5-stage pipelining architecture simulator for RISC-16 instruction set. Indian J. Sci. Technol. **10**, 1–9 (2017)
3. H.S. Bhimani, H.N. Patel, A.A. Davda, Design of 32-bit 3-stage pipelined processor based on MIPS in Verilog HDL and implementation on FPGA Virtex7. Int. J. Appl. Inf. Syst. **10** (2016)
4. S. Mangalwedhe, R. Kulkarni, S.Y. Kulkarni, Low power implementation of 32-bit RISC processor with pipelining, in *Proceeding of the Second International Conference on Microelectronics*. Lecture Notes in Electrical Engineering (2019), pp. 307–320. https://doi.org/10.1007/978-981-10-8234-4_27
5. M.R. Rakesh, B. Ajeya, A.R. Mohan, Novel architecture of 17 bit address RISC CPU with pipelining technique using Xilinx in VLSI technology. Int. J. Eng. Res. Appl. **4**, 116–121 (2014)
6. F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems* (Springer, The Netherlands, 2005)
7. Xilinx, *Vivado Design Suite: High-Level Synthesis* (2018). Available from: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf
8. Math Works, *HDL Coder*. Available from: https://www.mathworks.com/products/hdl-coder.html
9. K. Wakabayashi, C-based behavioral synthesis and verification analysis on industrial design examples, in *Proceedings of the ASPDAC* (2004), pp. 344–348
10. K. Kucukcakar, C.-T. Chen, J. Gong, W. Philipsen, T.E. Tkacik, Matisse: an architectural design tool for commodity ICs. IEEE Des. Test Comput. **15**, 22–33 (1998). https://doi.org/10.1109/54.679205
11. R.A. Bergamaschi, R.A. O'Connor, L. Stok, M.Z. Moricz, S. Prakash, A. Kuehlmann, D.S. Rao, High-level synthesis in an industrial environment. IBM J. Res. Dev. **39**, 131–148 (1995). https://doi.org/10.1147/rd.391.0131
12. P.E. Lippens, J.L. van Meerbergen, A. van der Werf, W.F. Verhaegh, B.T. McSweeney, J.O. Huisken, O.P. McArdle, PHIDEO: a silicon compiler for high speed algorithms, in *Proceedings of the European Conference Design Auto* (IEEE Computer Society Press, Amsterdam, 1991), pp. 436–441
13. J. Biesenack, M. Koster, A. Langmaier, S. Ledeux, S. Marz, M. Payer, M. Pilsl, S. Rumler, H. Soukup, N. Wehn, P. Duzy, The Siemens high-level synthesis system CALLAS. IEEE Trans. Very Large Scale Integr. **1**, 244–253 (1993). https://doi.org/10.1109/92.238438
14. D.W. Knapp, *Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler* (Prentice Hall, Englewood Cliffs, NJ, 1996)
15. Catapult, High-Level Synthesis (2020). Available from: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/
16. Stratus High-Level Synthesis. Available from: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html
17. J. Hennessy, N. Jouppi, F. Baskett, J. Gill, *MIPS: A VLSI Processor Architecture* (Springer, Berlin, Heidelberg; Stanford University, Departments of Electrical Engineering and Computer Science, 1981), pp. 337–346