Analyzing Resource-Performance Tradeoffs in Golden Gate

B.Tech Project 2020-21

Krish Kansara

December 14, 2020

Introduction

Introduction

- The goal of our project is to look at the processing of the Golden Gate compiler for hardware, and more specifically, at the resource performance trade-offs it makes, attempting to optimize it for the given design

Introduction

- The goal of our project is to look at the processing of the Golden Gate compiler for hardware, and more specifically, at the resource performance trade-offs it makes, attempting to optimize it for the given design
- The paper introducing Golden Gate laid this down as a point of future interest for their project

Introduction

- The goal of our project is to look at the processing of the Golden Gate compiler for hardware, and more specifically, at the resource performance trade-offs it makes, attempting to optimize it for the given design
- The paper introducing Golden Gate laid this down as a point of future interest for their project
- We also aim to look at certain FPGA-hostile components and see if making certain changes to Golden Gate can help in placing them onto FPGA

Current Progress

Current Progress

1. Writing source code in Chisel3 (**done**)

Current Progress

1. Writing source code in Chisel3 (**done**)
2. Verification by simulation using Vivado and Verilator (**done**)

Current Progress

1. Writing source code in Chisel3 (**done**)
2. Verification by simulation using Vivado and Verilator (**done**)
3. Synthesizing and converting the Chisel3 code into a bitstream to be put onto FPGA (**ongoing**)

1.1. Chisel3

1.1. Chisel3

- Hardware construction language embedded into Scala programming language (akin to a Scala library)

1.1. Chisel3

- Hardware construction language embedded into Scala programming language (akin to a Scala library)
- Allows hardware to be expressed in the paradigm of object-oriented programming

1.1. Chisel3

- Hardware construction language embedded into Scala programming language (akin to a Scala library)
- Allows hardware to be expressed in the paradigm of object-oriented programming
- Has extremely parameterizable configuration, which is a plus for better design space exploration

1.1. Chisel3

- Hardware construction language embedded into Scala programming language (akin to a Scala library)
- Allows hardware to be expressed in the paradigm of object-oriented programming
- Has extremely parameterizable configuration, which is a plus for better design space exploration
- Has a dedicated community working on it constantly, and is completely open-source

1.2. Writing Source Code in Chisel3

1.2. Writing Source Code in Chisel3

- Since our end goal is to optimize an existing design using the Golden Gate compiler, we decided to start with a design that we are very familiar with ourselves -

1.2. Writing Source Code in Chisel3

- Since our end goal is to optimize an existing design using the Golden Gate compiler, we decided to start with a design that we are very familiar with ourselves - the 5-stage RV32I CPU

1.2. Writing Source Code in Chisel3

- Since our end goal is to optimize an existing design using the Golden Gate compiler, we decided to start with a design that we are very familiar with ourselves - the 5-stage RV32I CPU
- We wrote the code for it in Chisel3, which also served as a good entry point for Scala programming and using the Chisel3 library

1.2. Writing Source Code in Chisel3

- Since our end goal is to optimize an existing design using the Golden Gate compiler, we decided to start with a design that we are very familiar with ourselves - the 5-stage RV32I CPU
- We wrote the code for it in Chisel3, which also served as a good entry point for Scala programming and using the Chisel3 library
- We used the Scala build tool (sbt) to compile the Chisel3 code and emit FIRRTL and Verilog files, as well as for other things such as fulfilling package dependencies.

1.3. Examining Chisel3 Source Code + A Demo

1.4. A Few Pointers

1.4. A Few Pointers

- The design is completely parameterized - that is, a flip of a single Boolean parameter can change the design from single-cycle to pipelined.

1.4. A Few Pointers

- The design is completely parameterized - that is, a flip of a single Boolean parameter can change the design from single-cycle to pipelined.
- This is in accordance with the philosophy of Chisel3 which aims to parameterize all hardware designing, since more parameterization means a larger design space to explore and optimize over

1.4. A Few Pointers

- The design is completely parameterized - that is, a flip of a single Boolean parameter can change the design from single-cycle to pipelined.
- This is in accordance with the philosophy of Chisel3 which aims to parameterize all hardware designing, since more parameterization means a larger design space to explore and optimize over
- We are planning to write code for testing the module in Chisel3 as well, though not immediately, as that purpose is currently served by Verilator

1.5. Relating This Work To Project Aim

1.5. Relating This Work To Project Aim

- As this is a design we are fairly well-acquainted with, we believe it will be easier for us to do initial work on Golden Gate in the context of this design, since we have an idea of what optimizations can be performed

1.5. Relating This Work To Project Aim

- As this is a design we are fairly well-acquainted with, we believe it will be easier for us to do initial work on Golden Gate in the context of this design, since we have an idea of what optimizations can be performed
- As an example, for the pipelined setup, we can make the compiler check if placing the modules in different stages can lead to an improvement in the design - essentially, making it perform retiming on the design

1.5. Relating This Work To Project Aim

- As this is a design we are fairly well-acquainted with, we believe it will be easier for us to do initial work on Golden Gate in the context of this design, since we have an idea of what optimizations can be performed
- As an example, for the pipelined setup, we can make the compiler check if placing the modules in different stages can lead to an improvement in the design - essentially, making it perform retiming on the design
- More such use cases include making the external memory a blackbox instead of known asynchronous-read-synchronous-write setup and allowing the compiler to create the best possible design for the same

1.5. Relating This Work To Project Aim

- As this is a design we are fairly well-acquainted with, we believe it will be easier for us to do initial work on Golden Gate in the context of this design, since we have an idea of what optimizations can be performed
- As an example, for the pipelined setup, we can make the compiler check if placing the modules in different stages can lead to an improvement in the design - essentially, making it perform retiming on the design
- More such use cases include making the external memory a blackbox instead of known asynchronous-read-synchronous-write setup and allowing the compiler to create the best possible design for the same, or the effect of implementing ISA extensions and seeing where the optimal design curve moves

2.1. Simulation using Vivado

2.1. Simulation using Vivado

- Once the Chisel3 code had been succesfully transformed into Verilog, the next step was to verify functionality by software simulation

2.1. Simulation using Vivado

- Once the Chisel3 code had been succesfully transformed into Verilog, the next step was to verify functionality by software simulation
- This was done using two variants of software - proprietary (Vivado) and open-source (Verilator)

2.1. Simulation using Vivado

- Once the Chisel3 code had been succesfully transformed into Verilog, the next step was to verify functionality by software simulation
- This was done using two variants of software - proprietary (Vivado) and open-source (Verilator)
- Simulating in Vivado needed no learning overhead, since we have already done that in the past, though it was still nice to see the functionality of the generated Verilog module match up with the expected results

2.2. Verilator

2.2. Verilator

- Verilator is an open-source program that converts Verilog HDL files into C++/SystemC files, which can be used to perform software simulation of the HDL modules using the API generated in the process

2.2. Verilator

- Verilator is an open-source program that converts Verilog HDL files into C++/SystemC files, which can be used to perform software simulation of the HDL modules using the API generated in the process
- We chose to learn and use Verilator in addition to Vivado, since Verilator is more commonly used in the open-source community which we are targeting. Specifically, the MIDAS compiler on which Golden Gate is based uses Verilator as a part of its workflow as well

2.2. Verilator

- Verilator is an open-source program that converts Verilog HDL files into C++/SystemC files, which can be used to perform software simulation of the HDL modules using the API generated in the process
- We chose to learn and use Verilator in addition to Vivado, since Verilator is more commonly used in the open-source community which we are targeting. Specifically, the MIDAS compiler on which Golden Gate is based uses Verilator as a part of its workflow as well
- In our case, we converted the CPU module generated by Chisel3 elaboration into C++, then wrote a testbench module using functions from the generated API, and finally ran the testbench to verify functionality

2.3. Examining Verilator Source Code + A Demo

2.4. Pointers and Future Directions

2.4. Pointers and Future Directions

- As of now, IMEM and DMEM are manually provided through the simulator, since a top module integrating the CPU, IMEM and DMEM hasn't been created

2.4. Pointers and Future Directions

- As of now, IMEM and DMEM are manually provided through the simulator, since a top module integrating the CPU, IMEM and DMEM hasn't been created
- Support for tracing via GTKWave makes the usage almost the same as XSim

2.4. Pointers and Future Directions

- As of now, IMEM and DMEM are manually provided through the simulator, since a top module integrating the CPU, IMEM and DMEM hasn't been created
- Support for tracing via GTKWave makes the usage almost the same as XSim
- Understanding how Verilator works will be important later while analyzing the MIDAS compiler, since it incorporates the former in its working

2.4. Pointers and Future Directions

- As of now, IMEM and DMEM are manually provided through the simulator, since a top module integrating the CPU, IMEM and DMEM hasn't been created
- Support for tracing via GTKWave makes the usage almost the same as XSim
- Understanding how Verilator works will be important later while analyzing the MIDAS compiler, since it incorporates the former in its working
- As for this specific use case, a top module can be created to simulate all three - CPU, IMEM and DMEM - simultaneously

3.1. Synthesis and FPGA Testing

3.1. Synthesis and FPGA Testing

- The final step to be taken before moving to the core of Golden Gate is to observe the process of putting the Chisel3 design created onto hardware, from synthesis to writing the bitstream onto FPGA

3.1. Synthesis and FPGA Testing

- The final step to be taken before moving to the core of Golden Gate is to observe the process of putting the Chisel3 design created onto hardware, from synthesis to writing the bitstream onto FPGA
- Since Golden Gate is essentially a (latency-insensitive) FPGA simulator, it is necessary to understand how it tackles this process

3.1. Synthesis and FPGA Testing

- The final step to be taken before moving to the core of Golden Gate is to observe the process of putting the Chisel3 design created onto hardware, from synthesis to writing the bitstream onto FPGA
- Since Golden Gate is essentially a (latency-insensitive) FPGA simulator, it is necessary to understand how it tackles this process
- Work on accomplishing this is currently ongoing. Here is the progress report so far -

3.2. Current Progress

3.2. Current Progress

- We are currently using the **MIDAS Examples** repo to test out the MIDAS compiler on a smaller scale before eventually moving onto Golden Gate

3.2. Current Progress

- We are currently using the **MIDAS Examples** repo to test out the MIDAS compiler on a smaller scale before eventually moving onto Golden Gate
- The simulation step, which uses Verilator, is working as expected for a given design

3.2. Current Progress

- We are currently using the **MIDAS Examples** repo to test out the MIDAS compiler on a smaller scale before eventually moving onto Golden Gate
- The simulation step, which uses Verilator, is working as expected for a given design
- For further steps, we are facing the following issues -

3.3. Issues with Current Goals

3.3. Issues with Current Goals

1. The synthesis step using Vivado is giving an error for a file either already present in the repo or autogenerated by the program. The offending code will have to be properly analyzed and the error spotted and resolved

3.3. Issues with Current Goals

1. The synthesis step using Vivado is giving an error for a file either already present in the repo or autogenerated by the program. The offending code will have to be properly analyzed and the error spotted and resolved
2. The Zynq board for which we are performing these steps has its repo missing on GitHub, which is causing an issue with the implementation. We are trying to find it on GitHub and Xilinx's website (we are not sure if what they require is the Zynq BSP on the latter)

3.3. Issues with Current Goals

1. The synthesis step using Vivado is giving an error for a file either already present in the repo or autogenerated by the program. The offending code will have to be properly analyzed and the error spotted and resolved
2. The Zynq board for which we are performing these steps has its repo missing on GitHub, which is causing an issue with the implementation. We are trying to find it on GitHub and Xilinx's website (we are not sure if what they require is the Zynq BSP on the latter)
3. For power modelling using their Strober software, the developers require the Synopsys Design Compiler. To resolve this, we plan to -

3.3. Issues with Current Goals

1. The synthesis step using Vivado is giving an error for a file either already present in the repo or autogenerated by the program. The offending code will have to be properly analyzed and the error spotted and resolved
2. The Zynq board for which we are performing these steps has its repo missing on GitHub, which is causing an issue with the implementation. We are trying to find it on GitHub and Xilinx's website (we are not sure if what they require is the Zynq BSP on the latter)
3. For power modelling using their Strober software, the developers require the Synopsys Design Compiler. To resolve this, we plan to -
   - Use a trial version of Synopsys

3.3. Issues with Current Goals

1. The synthesis step using Vivado is giving an error for a file either already present in the repo or autogenerated by the program. The offending code will have to be properly analyzed and the error spotted and resolved
2. The Zynq board for which we are performing these steps has its repo missing on GitHub, which is causing an issue with the implementation. We are trying to find it on GitHub and Xilinx's website (we are not sure if what they require is the Zynq BSP on the latter)
3. For power modelling using their Strober software, the developers require the Synopsys Design Compiler. To resolve this, we plan to -
   - Use a trial version of Synopsys
   - See if Vivado can be ported into this application

3.3. Issues with Current Goals

1. The synthesis step using Vivado is giving an error for a file either already present in the repo or autogenerated by the program. The offending code will have to be properly analyzed and the error spotted and resolved
2. The Zynq board for which we are performing these steps has its repo missing on GitHub, which is causing an issue with the implementation. We are trying to find it on GitHub and Xilinx's website (we are not sure if what they require is the Zynq BSP on the latter)
3. For power modelling using their Strober software, the developers require the Synopsys Design Compiler. To resolve this, we plan to -
   - Use a trial version of Synopsys
   - See if Vivado can be ported into this application
   - Use FPGAs on AWS via FireSim

3.3. Issues with Current Goals

1. The synthesis step using Vivado is giving an error for a file either already present in the repo or autogenerated by the program. The offending code will have to be properly analyzed and the error spotted and resolved
2. The Zynq board for which we are performing these steps has its repo missing on GitHub, which is causing an issue with the implementation. We are trying to find it on GitHub and Xilinx's website (we are not sure if what they require is the Zynq BSP on the latter)
3. For power modelling using their Strober software, the developers require the Synopsys Design Compiler. To resolve this, we plan to -
   - Use a trial version of Synopsys
   - See if Vivado can be ported into this application
   - Use FPGAs on AWS via FireSim
   - Forego power modelling and focus on other design aspects (**i.e. not an option**)

Conclusion

Conclusion

- In closing, a few steps remain to be taken before we can start poking into the working of Golden Gate and seeing what changes can be made to it in order to perform better resource and performance optimization as well as find better ways to design hardware that is normally hard to actuate on an FPGA

Conclusion

- In closing, a few steps remain to be taken before we can start poking into the working of Golden Gate and seeing what changes can be made to it in order to perform better resource and performance optimization as well as find better ways to design hardware that is normally hard to actuate on an FPGA
- The work done so far has been extremely helpful in understanding the working of the open-source hardware development tools developed by UCB BAR and using them to develop a design of our own, and with a little more of the kind of work done till now, we should be able to resolve the final issues with the implementation and move on to tacking the main aim of the project - Golden Gate

Thank You!

Our work is available on GitHub at
`https://github.com/vasid99/ScalaCPU`