Garp — A reconfigurable coprocessor for RISC-V

Qijing Huang, Sean Roberts
Department of Electrical Engineering and Computer Science
University of California, Berkeley
{qijing.huang, seanroberts}@berkeley.edu

***Abstract*—Due to the breakdown of Dennard scaling and the demand for low power devices in the new era, the architecture research refocused its attention with an emphasis on energy efficiency. Aside from circuit level optimizations, coprocessor design became another heated topic for building efficient systems. The main benefit of coprocessors is that their dedicated functions introduce less overhead. However, the selection of coprocessors is application-oriented. Implementing fixed function coprocessors can be unsuitable and costly in the long run for algorithms that vary quickly. Therefore, we need reconfigurability in hardware functions to make coprocessors adaptable to real workloads. In this paper, we presented our VLSI implementation of a reconfigurable coprocessor called Garp. Garp is a reconfigurable coprocessor architecture originally proposed by Hauser and Wawrzynek [1] for the MIPS II processor with coarse-grain programmability. We simplified the Garp design for a RISC-V processor, built the Garp coprocessor with the 32nm educational process and evaluated the performance and feasibility of the architecture in this work.**

***Index Terms*—Garp, reconfigurable coprocessor, VLSI, RISC-V.**

I. Project Objectives

According to the 21st century computer architecture whitepaper [2], one new architecture research direction is to enable specialization to improve performance and energy efficiency. However, full-custom accelerators are infeasible for all but the highest-volume applications due to the upsurging NRE costs incurred by the increasing complexity of silicon processes. Reconfigurable platforms, such as FPGAs, can drive the costs down but with the expense of undesirable energy and performance overheads. The paper predicts that accelerators in the future will use coarser-grain semiprogrammable building blocks to reduce the overheads incurred by the fine-grain programmability. However, the selection of building blocks remains an open research question. The amount of customization needed and its corresponding costs need to be studied carefully in order to make efficient reconfigurable accelerators possible. A powerful synthesis tool is also crucial to the advance of such architectures.

Garp, a reconfigurable coprocessor design proposed 20 years ago, falls in this category of semiprogrammable accelerators. Its basic building blocks, such as add, shift, select and table lookup, are prudently designed to support basic common operations. Complicated functions such as DES, MD5, and SHA can be constructed efficiently using these building blocks as shown in [3]. A hardware compiler is developed in bundle with the architecture which facilitates the use and testings of the hypothetical Garp architecture. Due to the completeness and cleanliness of its design, Garp is a perfect stepping stone for us to study the reconfigurable hardware design. Our goal for the project is to understand the design of Garp together with its compiler, implement it using the up-to-date technology, and evaluate the design.

II. The Design

A. Overall Architecture

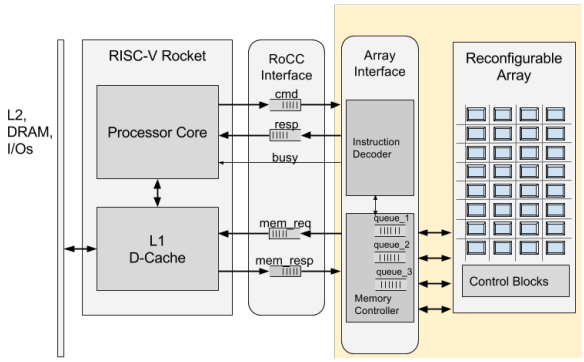

Fig. 1. Overall Architecture

Garp acts as a coprocessor to the single issue RISC-V Rocket as shown in Figure 1. It communicates to the processor and memory through the RoCC interface. The RoCC interface defines a standard set of ready/valid signals for instruction and data transfer. We implemented the Garp together with the instruction decoder and memory controller to the RoCC interface highlighted in yellow in Figure 1.

As a reconfigurable coprocessor, Garp takes two levels of inputs:

1) the configuration to set the coprocessors function
2) the data to perform calculations on

It requires custom RISC-V instructions to provide control for the configuration and data movement. It also requires memory access to request configurations and data from the memory. Both can be satisfied by adding glue logic to the RoCC interface. However, the ability to change cache allocation policy and to prefetch data, which is demanded by the original Garp design, is not directly supported by the RoCC interface. We decided to drop the corresponding Garp functions as modifying the RoCC interface is out of the project scope.

A typical flow to utilize Garp is described as follows:

1) The processor loads a configuration onto the array.

2) The processor initializes data input and a counter value to specify the number of cycles the coprocessor should run for.
3) The coprocessor starts running and the counter decrements after every clock cycle.
4) Once the counter value reaches zero, the execution on Garp is done.
5) The processor copies the results back to processor registers.

B. Array Organization

As shown in Figure 2, the structure of Garp resembles an array, so we will refer to it as the "array" in the following sections. The array is organized as 24 x n blocks. The number of rows is implementation specific. Each row of the array is composed of 1 control block and 23 logic blocks. A typical 32-bit data thus will take up 16 logic blocks as highlighted in blue in the figure. The memory buses, shown as double-headed arrows in the figure, are running in between columns to transfer data to/from the array.

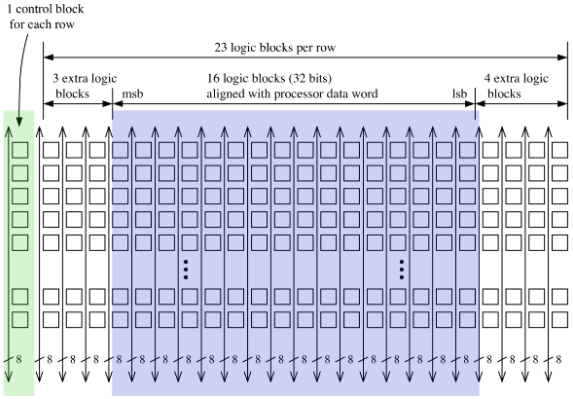

Fig. 2. Garp Organization [3]

C. Wire Connections

The local data transfers among blocks are supported by the array wires. Between the rows, there are horizontal wires called G wires and H wires to transfer data between columns. As shown in Figure 3, the G wires (highlighted in red) span the entire 24 columns of the array, while the H wires (highlighted in green) span exactly 11 blocks. Between the columns, there are vertical wires called V wires to move data between rows. The V wires come in a range of lengths as shown in Figure 4.

The wire network is passive, meaning a value cannot jump from one wire to another without passing through a logic block. There can be multiple readers on the same wire, but only one driver is allowed.

Connecting the array wires, especially the H wires and V wires, is one of the most challenging parts in our implementation.

1) H Wire: For the H wires, the logic block select the input based on its local index, but the connections are made using global index. Each H wire can be driven from center, left or right. With the information from Garp configurator, we were able to figure out the starting position of the first H wire and the index of the driving block as in Figure 5. The figure shows the proposed physical layout of the H wires running below each row of 24 blocks. The numbers listed above the array are the indices of the logic blocks. The control block is indexed as 'c'. In each row, every stripe in one color represents one wire; the characters 'R', 'C' and 'L' mark the three different driving points of the wire. Index lookup in this representation of wires is rather easy. For example, from Figure 5, we can tell the first blue H wire in the 0th row spans from logic block 21 to logic block 11 and can be driven by logic block 20, 16 and 12. Every logic block is able to connect to all the wires passing through its column. As the leftmost wire seen from the array block is always indexed as 0 locally, we cannot simply connect the wires to the blocks based on the absolute row number.

We tried giving the H wires global indices based on their arrangement in Figure 5 but found it to difficult to make the connections due to the irregularity of the wires highlighted in green and yellow. We then came up with an unrolled layout of the H wires, which is demonstrated in Appendix C. The unrolled indexing turns the global index to local index conversion to continuous mapping, which makes implementing and debugging the wire connections more straightforward.

Fig. 3. H Wire and G Wire [3]

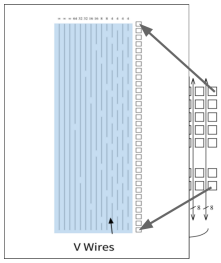

Fig. 4. V Wire [3]

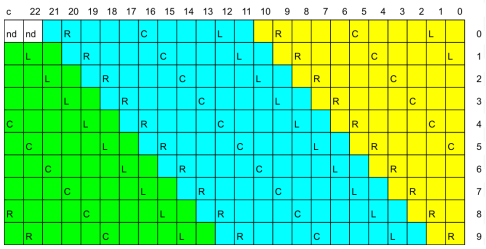

Fig. 5. H Wire Indexing

2) V Wire: The V wires are laid out in such a way that configurations assume a smaller number of rows would still have valid encodings when placed at the top of larger arrays. In other words, the V wire structure is recursively defined such that the layout of V wires for a small array is most of the top half of the layout of the next largest size of array. While this pattern has nice properties, it makes specifying the V wires in RTL somewhat challenging. We therefore looked into the code for the golden reference Garp array simulator. This code

lists all of the wires contained within the array as well as their length, nominal length, and nominal starting point. We were able to use this information and the row number of a logic block to determine which in port the V wire should be connected to. The golden reference code showed us that the global V wires were actually illusory. They are actually just wires with a long enough nominal length that they span the whole array for the current number of rows. This meant that we did not need to develop a special case to handle them.

D. Logic Block

Each logic block implements a function of up to four 2-bit inputs. Its functions and inputs is specified by a 64-bit configuration as in Figure 6. Select signals to the input multiplexers are tied to the configuration during execution. Figure 7 shows the datapath of a logic block. Four 2-bit inputs (A, B, C, D) are taken from adjacent wires and can be used to derive two outputs. One output (Z) is calculated from the function unit, and the other (D) is a direct copy of an input. Each output value can be optionally buffered in a register, after which the outputs can be used to drive onto as many as three wires leading to other logic blocks. The logic block registers can also be read or written over the memory buses.

Figure 8 shows the system diagram of the function unit within the logic block. Users can construct adders, shifters, selectors and arbitrary logic functions by setting the mode field in the configuration bits. Different modules in the function unit will be activated accordingly by the control signals generated from the configuration. The system diagrams of the function unit in each mode are included in Appendix A.

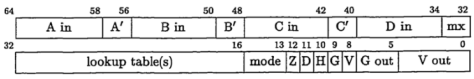

Fig. 6. 64-bit Configuration Encoding [3]

Noted from the logic block datapath, there are multiple paths that form combinational loops. The logic block can take input from any adjacent wire while outputting to the same wire, and not every path within the logic block goes through a register. As the actual datapath is defined by the configuration, such encoding is invalid in the real use of the Garp array. However, Chisel does not synthesize any design with potential combinational loops in it. In order to make Chisel work, we have to insert black boxes of wires to 'break' the loops.

E. Control Block

The primary purpose of the control blocks, which occupy the periphery of the array, is to convert select H wire signals into a multitude of control signals depending on the configuration encoding. First, each of the four inputs selected with multiplexers is passed through a reducer block that converts the two bit signal into 1 bit. The first reduced signal always serves the same purpose of enabling the other three signals depending on mode. In addition to an idle no function mode, each control block can also be in processor mode or memory

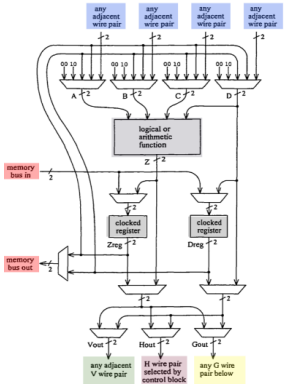

Fig. 7. Logic Block [3]

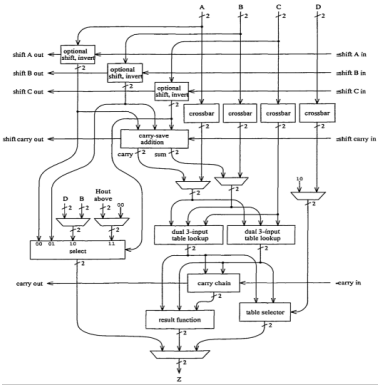

Fig. 8. Function Unit [3]

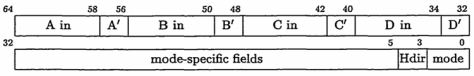

Fig. 9. 64-bit Configuration Encoding [3]

interface mode. In processor mode, control blocks can be used to zero the array counter or interrupt the main processor. In memory interface mode, the control blocks can initiate memory accesses and load or store data to or form the registers of its associated row of logic blocks.

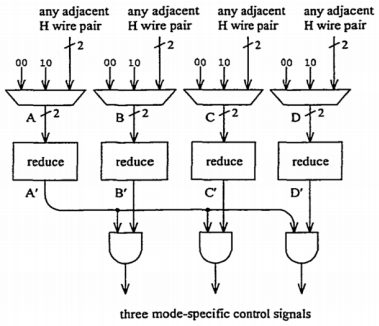

Fig. 10. Control Block [3]

F. Instruction Decoder

The original specification for the Garp coprocessor included many instructions that could be used by the main processor to interact with it. These instructions conformed to the MIPS ISA and required adaptation to be used as RoCC interface, RISC-V instructions. Additionally, we decided to focus on the subset of instructions that would be necessary for running the tests on the Garp array.

The first instruction is `gaconf` which loads configuration from memory into the garp array. This takes one register value in `rs1` which is the pointer to the configuration in memory. The size of the configuration encoding is not needed because it is stored along with the configuration. The next instruction `mtga` is used to send data from the main processor to the registers contained within the logic blocks of one row of the array. The value to be sent to the array is stored in `rs1` while

the row it is stored in is set using the `rs2` field as a literal. The array clock is also set in this command by using the `rd` field as a literal. The final command `mfga` is the complement of `mtga`, retrieving the value stored in the registers of a given row of logic blocks. In this case, `rs1` is used instead as a literal to set the array clock so that the `rd` field is free to specify the processor register to return the array data to. For both `mtga` and `mfga`, the lowest bit of the `funct7` field is used to determine whether Z or D registers are being used.

G. Memory Bus and Controller

We decided to have only one memory bus instead of four with the extra memory queues at current stage to simplify the memory controller complexity. Since the Rocket is 64-bit RISC-V core, one transaction from/to the Rocket memory system through the RoCC interface is able to fill the 48-bit memory bus. Figure 11 shows the block diagram of the memory bus and its corresponding controls. For loading data from the processor to the Z or D registers of the middle 16 blocks, *Load_Data* signal should be asserted high. The 5-bit address is provided from the data transfer instruction `mtga` and `mfga` issued from the Rocket. There is no need to use another *Write_Data* signal to support the `mfga` instruction as we can directly use not *Load_Data* to indicate a write from the array.

For loading configuration, the signal *Load_Config* should be asserted. Each block requires 64 bits of configuration data, so if all the blocks are loaded in parallel, it would require 32 cycles to load the configuration for a full array row. However, currently the configuration data for one block is compiled to one consecutive 64-bit word in memory. Without preloading 24 words from memory, it will require 32 x 24, which equals 768 cycles to load one row. Preloading will add 24 cycles to the original 32 cycles if the scheduling is not interleaved properly. It will add complexity to the memory controller. Therefore, we decided to implement the parallel loading scheme assuming the configuration is lay out as the memory bus needed in memory. It will need modification to compiler to put the first 2 bits of every configuration in the same row to one word, the second 2 bits to the next word and so on. We like this idea better as it is easier to make changes on software compiler side.

III. Design Verification

We performed a battery of different tests across the pieces of RTL that were used in this project. The Chisel poke and peek testing framework was used to unit test smaller modules. Complex modules with reasonable input vectors like the functional unit were tested by exhaustive sweep where feasible. Modules with larger input vectors like the logic block were tested with partially randomized input vectors since it would be too time consuming to enumerate through all of the possibilities. The software standard that the modules were compared against was written in Scala by the person who did not implement the module to avoid making overlapping interpretation mistakes.

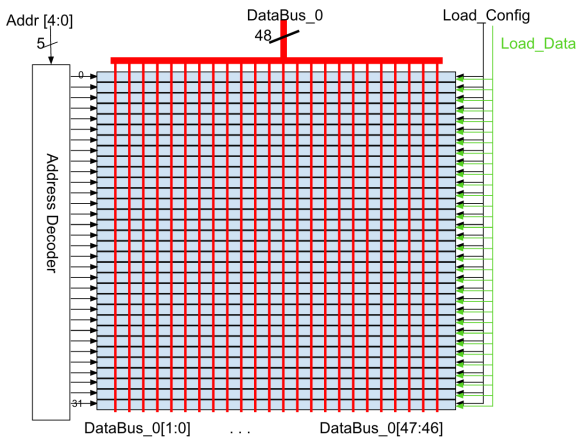

Fig. 11. Memory Bus and it controls

After the interconnect wires were implemented, we were able to test our Garp array against a software full design golden reference previously implemented and provided to us. Along with the software simulated Garp array, we also received a configurator which generates configuration encodings for the various array blocks from a human readable text specification. This configuration is consumed by both the golden reference and our simulated hardware Garp array, allowing cycle by cycle comparisons of the two. In this way, a couple dozen full array test operations were performed, enabling us to verify the interconnect network and further verify the array blocks.

IV. Synthesis Results

This section describes our effort to synthesize the array using the standard tools and the synthesis results we gathered.

A. Hierarchical Flow

In order to shorten the time to synthesize the Garp design and take advantage of the regularity of the design, we investigated a hierarchical flow called MIM (Multiply-instantiated Module) flow. The MIM flow is consisted of the following steps:

1) Run synthesis on tile (dc-syn.tile)
2) Run synthesis on top (dc-syn.top)
3) Run floorplanning on top (icc-dp)
4) Run P&R on tile (icc-par.tile)
5) Run synthesis on top with tile macro (dc.syn.top.hier)
6) Run P&R on top with tile macro (icc.top)

We managed to carry out an small experiment to tile four AND modules (highlighted in while) in a 2x2 grid as shown in Figure 12. However, as we ported the MIM flow scripts from the 28 nm real process to our 32 nm educational process, many metal layers are missing. For larger design with long wires, the tool crashed on step 3) presumably because our re-assigned metal layers are invalid.

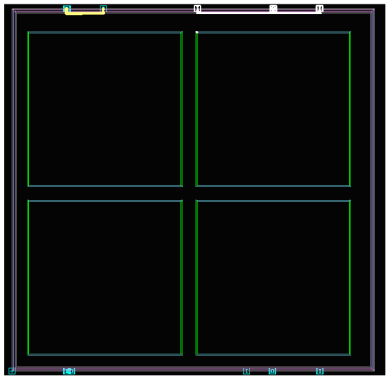

Fig. 12. MIM Flow Experiment Output

As of the time the paper is written, two compilations for our design with 1 rows and 4 rows using MIM flow are still running. As seen from the terminal output, we are still getting the errors for creating power straps, but the IC compiler did not crash after we specified reasonable combinations of metal layers.

B. Top-down Flow

1) **Design Compiler:** In this part, we present the DC results for our design space exploration. We used DC as it provides a reliable prediction of the circuit performance in relatively short compilation time.

a) Number of Rows: One feasible design space exploration for our design is to vary the number of rows. Table I shows a performance comparison between the array with 4 rows and 8 rows. The results meet our expectation, as double the number of rows should also double the area. The extra area for the 8 rows may come from the extra V wire connections. The clock period is invalid as the actual critical path is constrained by the configuration. The energy figures are also inaccurate because of the invalid timing information.

Number of Rows	4	8	ratio
Area (μm^2)	179350	388432	2.17
Clock Period (ns)	471.8	641.01	1.36
Power(μW)	1460	3150	2.16

TABLE I
PERFORMANCE COMPARISON

Overall compile time for 4 rows and 8 rows using Design Compiler is 1.88h and 12.32h respectively, but increasing the rows to a full array size 32 took more than 1 week without finishing. This might be because the combinations of connections are grown exponentially to the number of rows. Most of the existing Garp examples for simple operations utilize less than 10 rows of the array, so we think that 8 rows is a proper configuration for Garp. More rows will incur much more NRE time without providing too much functional benefit.

b) The Use of Tri-state Buffers: The other design space exploration we performed is to use tri-state buffers in place of the multiplexers to see the impact on the area of the G wire input multiplexer. Table II shows that the forced use of tri-state buffers to build the multiplexer will double the area. The design compiler is able to make better decision in the selection of standard cells to synthesize a multiplexer as it has a standard cell library to extract useful specifications from and can optimize the area by attempting many combinations of cells.

	Multiplexer	Tri-state Buffer	Ratio
Area (μm^2)	143.8455	282.0998	1.96

TABLE II
MULTIPLEXER AREA COMPARISON

c) The Use of Don't Care Signals: After we synthesized the logic block, we discovered that the input multiplexer occupied most of the area. In Figure 7, the blue input box for any adjacent wire actually contains 46 wires (16 V wires + 22 H wires + 8 G wires). Together with the other four inputs, they form a 2-bit wide 50-to-1 multiplexer. We then discovered that the 50 inputs did not fill the input encoding field of 6 bits, 2^6 - 50 = 14 inputs can be don't care signals in order to optimize the area. Therefore, we created a Verilog blackbox of the multiplexer with don't care signals and compared the area results to the Chisel implementation without don't care signals. Result in Table III shows that, with the use of don't care signals, the area can be reduced by 20 %.

	Chisel	Verilog	ratio
Area (μm^2)	1059.2722	833.3382	0.79

TABLE III
MULTIPLEXER AREA COMPARISON

2) **IC Compiler:** This section discusses the IC Compiler results for a Garp design with 4 rows using the top-down flow.

At the beginning we were not able to compile the same design with the top-down flow within one week using the IC Compiler because constraints we specified were too hard to reach. We thus relaxed the constraints for time and area to accelerate the compilation process. We set the clock period to 500 ns and the chip height and width to 1000 μm.

Table IV lists the ICC results for the ICC run. From the area figure of the 4-row array, we estimated the area of a full size 32-row array to be at least 2.59 mm^2, which is around 1.9x larger than the Rocket core. In terms of area, the Garp is feasible to be a coprocessor, but its timing and power figures generated by ICC seems to be extremely large. For normal hardware design, a critical path of 469.59 ns means it operates at 2.1 MHz, which is orders of magnitude slower than nowadays processors. Compared to the Rocket chip's 2.21 mW power consumption, The power figure 8.63 mW for 4 rows also seems to be fairly high. Nonetheless, the timing result generated by the ICC compiler is not valid for the reconfigurable architecture, as the critical path of an application is not only defined by the delay between registers but also by its configuration. Since the timing affects the power estimation, the power figure might not be accurate either.

Rows	Total Area (μm^2)	Cell Area (μm^2)	Clock Period (ns)
4	324213	197090	469.59

	Switch Power	Int Power	Leak Power	Total Power
μW	13.549	62.56	8550	8630

Cell Type	lvt_cell	rvt_cell	hvt_cell
Number	744	4055	59233

TABLE IV
ICC RESULTS

To better understand the cost of each module, we extracted the area number from the ICC report and generated two breakdown charts in Figure 13 and Figure 14. Figure 13 shows a breakdown of area for the logic block. The input multiplexers still take up the largest portion of area even with the unused input signals specified to don't care signals. The 64-bit configuration registers and its corresponding logic to load the configuration from the memory bus take up the second largest area. The function unit, which is the actual computing module, only occupies 26.7% of the total logic block area. The area result shows that most of the area is consumed by the wire selection and the configuration in Garp design to support customization, not the real computing unit.

Within the function unit, the two 3-input lookup table modules take up 46.1% of the area. The 3 shift-invert modules that are composed of 12 2-input multiplexers, and the 4 crossbar modules that are composed of 8 2-input multiplexers, occupy the second and third largest portion of area in the function unit.

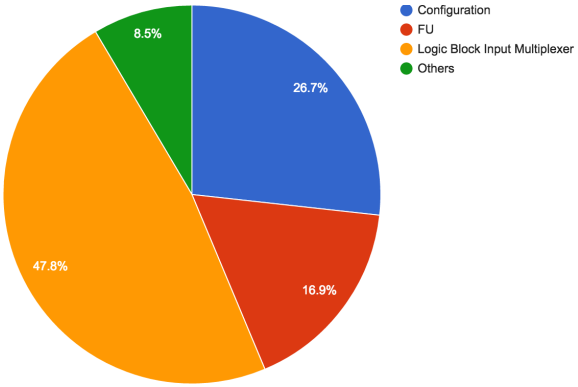

Fig. 13. Logic Block Area Breakdown

Figure 15 shows the physical layout of the Garp design. From the figure, we can see global wires running vertically and horizontally across the chip, but the standard cells are not visible in this view. We thus generated a layout view with highlighted standard cells in Figure 16. The figure shows that most of the chip area is utilized by the standard cells. Figure 17 shows an interesting layout of how the four rows are placed in the design. The array rows are shaped into four squares instead of four rows in the figure, which is very different to the optimal placement in our imagination. Another layout view worth noticing is the highlight of critical path. As shown in Figure 18, the highlighted critical path runs through all the

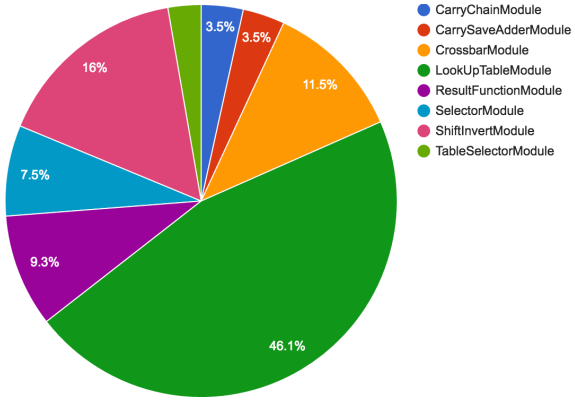

Fig. 14. Function Unit Area Breakdown

array rows and is able to travels back and forth from row to row for several times. This is the main reason the clock period we got from ICC is high. Since the P&R is timing directed, the critical path spanning as a nested loop on chip could also contribute to the four square layout in Figure 17.

According to the original Garp design [3], only the following sequences of connections are valid to make:

1) short wire, simple function, short wire, simple function
2) long wire, any function not using the carry chain
3) short wire, any function

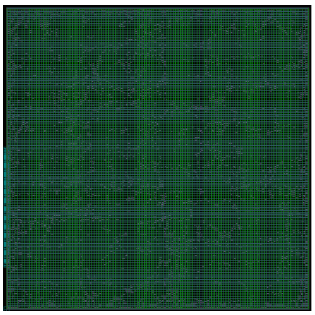

Fig. 15. Chip Layout

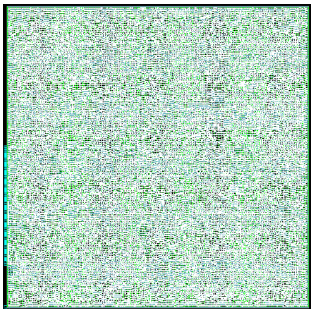

Fig. 16. Highlighted Std Cells

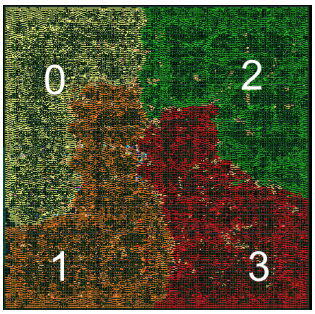

Fig. 17. Colored Hierarchical Cells

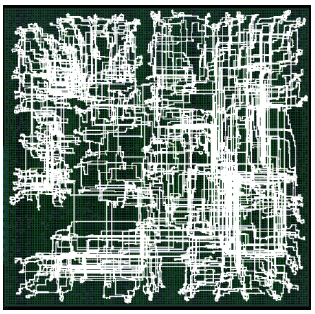

Fig. 18. Highlighted Critical Path

Without creating constraints to the IC compiler, it will not know which sequence of connections is valid. Thus, in order to direct the IC compile to optimize the circuits as we needed,

we tried using the ICC commands set_disable_timing and set_false_path to set the wrong paths in the design. We are able to set all the paths driven from the configuration registers to be false because the configuration value is fixed during execution. However, since there is no command to set the valid paths and the valid paths are depending on the real time configuration of the logic block functions according to Garp specifications, we are not able to find a good way to specify valid paths so far. Using the MIM flow is the only workaround, which might prevent the IC computer from using invalid timing paths as compilation directives.

In all, areawise, the Garp design is feasible to be a coprocessor to the Rocket core. The full size 32 rows design is not necessary for many applications and may incur high NRE design cost. In order to justify the timing and power aspects of the array, we need to be able to specify the valid critical paths, which turned out be a rather challenging task. Though we did not have valid information of the timing, we are not optimistic about the power as the power figure is already high even when circuit is running at an extremely low frequency.

An efficient layout should resemble the original hand-drawn layout in Appendix B. It is only accomplishable with the use of hierarchical flow. However, the design tools are way more complex then we expected. Thus, we conclude that the proficiency with the tools is crucial to the success of a VLSI project. There is a lot more knowledge for us to acquire in order to finish the design.

V. Retrospective

A. Garp Reflections

Perhaps the Garp architectures most noteworthy divergence from more traditional reconfigurable hardware like FPGAs is its smaller and simpler configurations at the expense of reduced generality. This tradeoff is reasonable for Garp because it is typically used for short 32 bit pipelines that handle the inner loops of programs.

Testing of the Garp array required us to generate many configurations. These took on the order of seconds and the resulting configurations were on approximately $192 \times R$ bytes where R is the number of rows, small enough to be embedded as literals in the programs that use them. As promised, this is a dramatic difference from configuring something like an FPGA. This is good news considering that the utility of Garp is contingent on its ability to switch between configurations during runtime.

While developing the RTL, we realized that there is a high degree of reuse of circuitry across the different modes that the logic blocks can take on. In general, the design of the array blocks, excluding the interconnect portions, was pretty novel and well adapted for the task of a reconfigurable pipeline element. One downside of the array block designs was the large area devoted to multiplexing input signals. For our logic blocks, the multiplexers accounted for a majority of the total area, even after space saving measures like blackboxing for dont care signals were introduced. The inclusion of input multiplexers in the array blocks allow for the interconnect network to be entirely devoid of routing circuitry and it is still an open question which would have required more area. The multiplexers could also possible be implemented with different elements to save area as ours were still implemented mostly with logic gates. Either way, the amount of area used by the multiplexers exceeded the expectations we held at the beginning of this project.

We also found some disadvantages associated with a network of interconnect wires distributed across the array. The move away from the kinds of mesh networks seen in FPGAs certainly decreased the time and complexity of routing, but meant that many wires experienced low utilization for a given configuration. This could be an artifact of the tests that we looked at or a necessary consequence of the rigid nature of the network. Ultimately, the cost of the wire network might not be that great since they span a very regular array. We only attempted to use the MIM flow to place the array blocks but made no special effort to place the interconnect wires so there could be room for improvement on this front in the future.

B. Chisel Reflections

Chisel proved to be a good tool for the types of designs like the Garp array, which required a number of repeated and parameterized modules within a hierarchical structure. There were some parameters of the Garp array like the array granularity which we ended up leaving fixed for our project. Developing the code to handle a range of different granularities would require extensive modifications to the functional unit and the advanced generation capabilities of Chisel would be vital.

Exotic aspects of our designs sometimes proved difficult to implement in Chisel. We used large non-power-of-two multiplexers and the default signal requirement effectively forced us to include additional unused signals to arrive at the next power of two. We also had bus constructs which could be driven by tens of inputs and no support for tri-state buffers would mean that additional multiplexers would need to be used. Also, because of the reconfigurable nature of the design, there are numerous combinational loops in our implementation that would never be traversed for valid configurations. All of the above problems were handled in our design with the help of Chisel blackboxes. This fix was not without problems. Firstly, the use of blackboxes immediately prevents Chisel C++ simulations from being performed. This will likely become a nonissue with the next version of chisel because the software simulator will be derived from the emitted Verilog. Secondly, it meant that the design constantly jumps into and out of Chisel and a module hierarchy is imposed which does not play well with the flows that we were experimenting with. For example, it may be advantageous to embed tri-states used to drive interconnects within array block modules for the purpose of tiling, but making bus blackboxes prevented us from doing so. Although additional support from Chisel would have been helpful to us for this design, we recognize that Chisel needs to strike a balance between low level details and ease of use and it is possible that our design is sufficiently

on the fringe that expecting Chisel to accommodate all of its idiosyncrasies is unreasonable.

During simulation we were faced with some additional problems with Chisel. At one point the encoding configuration of all the array blocks under test was part of the test vector. This results in a Java index out of bound error because the large amount of encoding signals was larger than a fixed size buffer used for testing. Of course it is possible to change our test vector to fit within the buffer but that should not be necessary. We resolved this issue by using dynamic Java containers to store the simulation signals. It is possible this problem arose due to the version of Chisel used in CS250 and that the issue has already been resolved as there was discussion of it on the public Chisel repository. Additionally, when testing larger sizes of arrays we ran into Java out of memory errors. Our solution was to add `export SBT_OPTS="-Xmx4G -Xms4G"` and `export JAVA_OPTS="-Xmx4G -Xms4G"` to `bash_profile`. Even though both of these problems were eventually resolved, it took quite a while to determine the source of these bugs. Part of the trouble is that since Chisel is embedded in Scala and running on the JVM, there are three different families of error messages and they are frequently not very illuminating.

C. Flow Reflections

Possibly the greatest lesson learned during this project was that adapting CAD tools, whose notion of meeting timing relies the delay between the output of one register and the input of the next, requires sizable manipulation in order to arrive at reasonable results for reconfigurable hardware circuits. Even if we assumed that the timing number was nonsensical for our true purposes, it was often hard to reason about how this false constraint might compel the tools to make unnecessary tradeoffs and lessen the validity of other results.

The array portion of Garp is both large and regular which we attempted to capitalize on through the use of the MIM flow. Using MIM flow was ultimately quite a headache because it was simultaneously complex and closed. This meant that we could not find examples or ask question online. Scripts were eventually adapted from other BWRC projects, but these used a different standard library that was under NDA and had different layer definitions. We were forced to adapt the scripts as best we could to the educational standard cell library, with no documentation to verify if the changes we made were appropriate. As advice for future iterations of CS250, we recommend that projects either do not require extensive changes to the class flow, or else making those changes should be a first priority so that unfruitful paths can be avoided sooner.

ACKNOWLEDGMENT

The authors would like to thank all the people who helped them for the project, especially Prof. John Wawrzynek, Dr. John Hauser and Christopher Yarp for their valuable advice and Palmer and Ben Keller for sharing their knowledge.

REFERENCES

[1] J. R. Hauser and J. Wawrzynek, "Garp: A mips processor with a reconfigurable coprocessor," in *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*. IEEE, 1997, pp. 12–21.

[2] M. D. Hill, "21st century computer architecture," *SIGPLAN Not.*, vol. 49, no. 8, pp. 1–2, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2692916.2558890

[3] J. R. Hauser. Augmenting a microprocessor with reconfigurable hardware. [Online]. Available: http://brass.cs.berkeley.edu/documents/AugmentingProcWithReconfigHardware.pdf

APPENDIX A
FUNCTION UNIT MODES

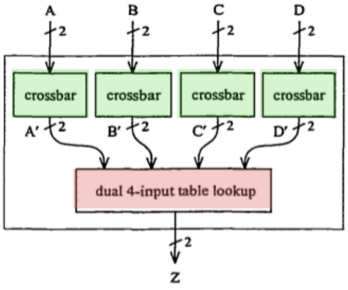

Fig. 19. Table Mode

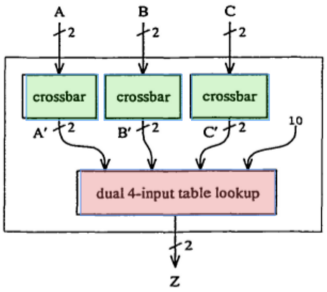

Fig. 20. Split Table Mode

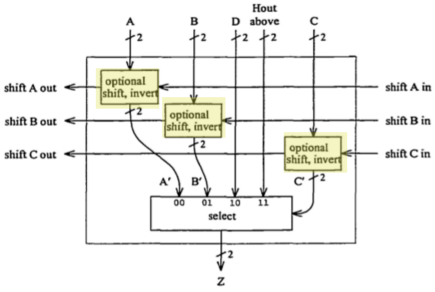

Fig. 21. Select Mode

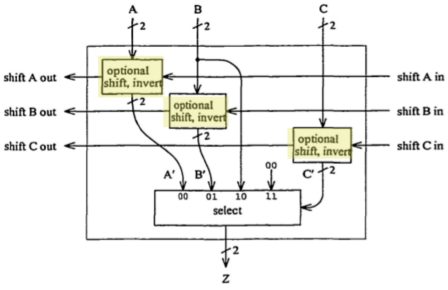

Fig. 22. Partial Select Mode

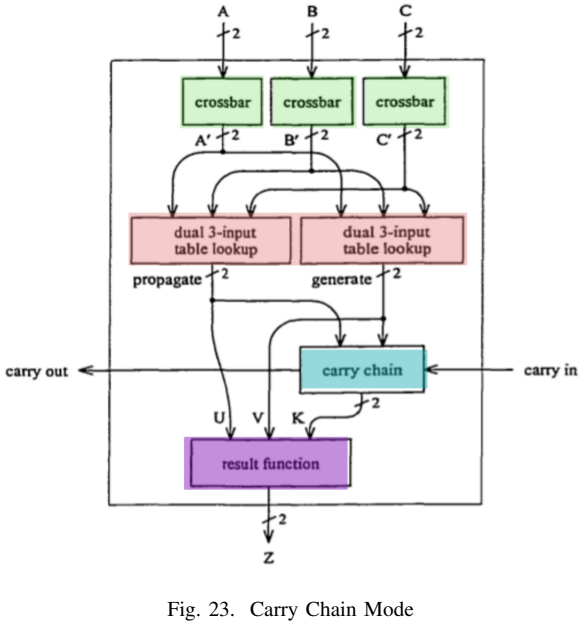

Fig. 23. Carry Chain Mode

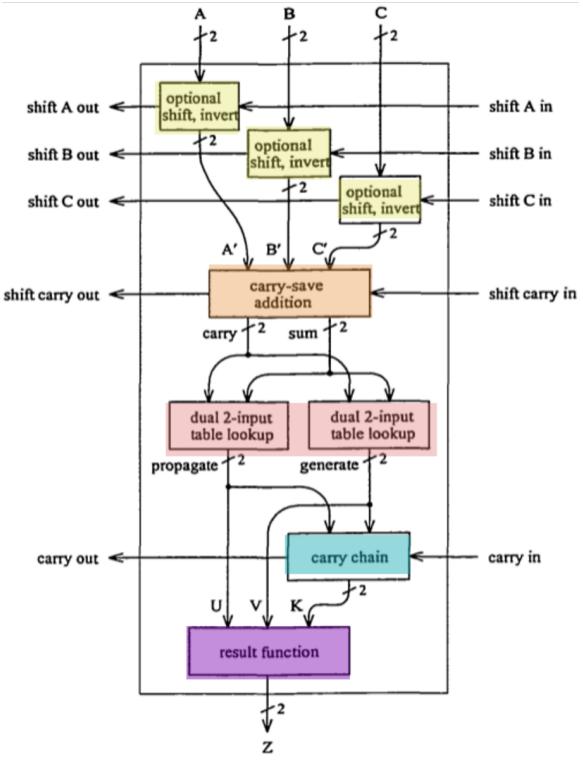

Fig. 24. Triple Add Mode

APPENDIX B
PROPOSED GARP LAYOUT

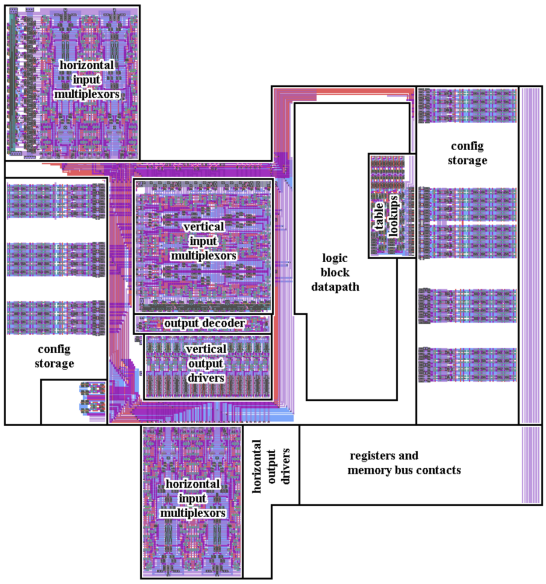

Fig. 25. Partial Garp Layout

APPENDIX C
H WIRE INDEXING

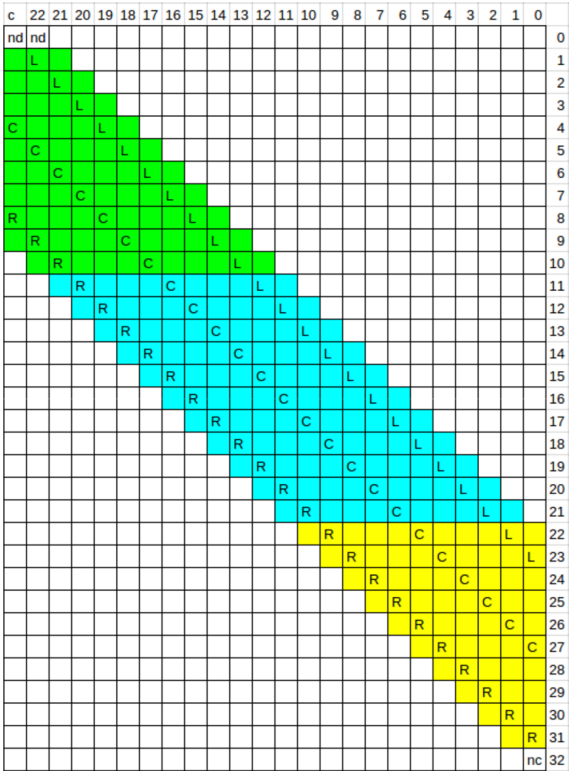

Fig. 26. H Wire Indexing Unrolled