Wannabee Larrabee

Authors: Alexander Gotsis, Electrical and Computer Engineering, Carnegie Mellon University

Cyril Agbi, Electrical and Computer Engineering, Carnegie Mellon University

David Gronlund, Electrical and Computer Engineering, Carnegie Mellon University

Abstract—**We aimed to replicate the architecture of the Larrabee project that was developed and ultimately abandoned by Intel, where we use the RISC-V ISA instead of x86. This allows for us to use an even simpler integer pipeline and improve the ratio of CPU logic to floating logic. Just like Larrabee, our processor cores are going to be capable of vector operations on single-precision floating-point numbers, and will use the RISC-V vector extension to program our cores in C. Our vector cores will use independent scratchpad memories, whose contents will be managed by code running on the supervisor.**

Index Terms—**ASIC, CPU, FPGA, FPU, GPU, VPU**

I. Introduction

With power limitations and a slowdown in Moore's law, computing has moved towards increasingly parallel architectures. Graphics processing units (GPUs) are a good example of a successful attempt at parallelism, with graphics problems providing a large number of completely independent operations that can be executed in specialized shader cores on the GPU. These cores and the general architecture of a GPU emphasizes numerical throughput over decision making, in part since multiple shader cores share instruction fetch and decode logic, limiting the amount of jumps any single core can effectively execute. GPUs provide a lot of their speedup by implementing application specific hardware, like texture and rasterization units, which are not programmed through assembly code but instead given smalls commands and then let to run independently on some data.

Intel Larrabee was intended as a GPU competitor, as its initial marketing and benchmarks were running different video games. As an effort to leverage existing designs, but also to simplify development for it, Intel decided to use general purpose x86 cores instead of purpose-built shaders in Larrabee, where there would be less x86 cores than shader cores in a comparable GPU, but far more cores than in a CPU of the same era. A GPU, along with having customized computing hardware, also has very specific pipelining and memory hierarchy, while Larrabee would instead give each core equal access to main memory. This allowed for user code to allocate cores as it saw fit to different applications, with cores potentially split up between applications just like memory is allocated.

To simplify the design of our architecture we used RISC-V instead of x86 as the ISA for each of our cores. We can leverage existing compilers for RISC-V to write code for each of our cores, and given the simplicity of the RISC-V ISA we can produce a minimal integer pipeline whose only ISA augmentation is floating-point and vector processing logic, essentially implementing RV32-IFV.

To demonstrate and debug our architecture we want to try to implement it on two different FPGA boards, with different sized FPGAs. The FPGA is a good way to demonstrate that the architecture not only runs real code, but that we can meet some basic timing requirements and that our logic can make efficient use of the resources given to it. Our RISC-V processor for this reason will be optimized for targeting an FPGA so that the vector coprocessor will be able to run at full speed. The FPGA we are targeting also has an ARM CPU attached to the FPGA fabric, which is easy to boot Linux on and will be our supervisor core, where jobs can be dispatched to the vector cores and results read back to potentially be drawn to the screen or saved to a file.

Our goal was to perform at about 50% of each individual FPGAs theoretical floating-point performance, by only using software running on our vector cores. This reduction from the theoretical performance is there to accommodate the FPGA design's clock speed being about half of what the theoretical DSP slice speed is, and also any inefficiency of the software and memory model of our architecture. This should result in a maximum floating throughput target for our architecture of 0.116 TFLOPs on the Ultra96 and 0.977 TFLOPs on the ZCU102.

II. Design Requirements and Benchmarks

Our primary design requirement was to be able to most effectively leverage the floating-point resources in the FPGA, which through each of our vector cores should be usable entirely through software. The platforms we wanted to benchmark our architecture on are two different Xilinx Ultrascale+ development boards, which both have the same ARM processor attached to the FPGA fabric. This would allow us to boot nearly identical software on each board, and hopefully demonstrate that our architecture can scale by simply providing more cores for the user. We also wanted to use two, and if time allowed three, different benchmarks to show that across different workloads our architecture can saturate the math resources in the FPGA.

Board/FPGA	**DSP Slices**	**Max. DSP Freq.**	**Max. Theoretical FLOPs (Multiply)**
Ultra96 ZCU3EG-1	360	645	0.232 TFLOPs
ZCU102 ZCU9EG-2	2,520	775	1.953 TFLOPs

For all of our benchmarks we wanted to write them in C and then run them on the ARM core alone to simply show that they work. Then we tried to make sure that they were being accelerated by the NEON FPU next to the ARM core, which would hopefully show some improvements in performance. Next, we wanted to work on porting the code over to our architecture, which would first involve managing moving code and data around between the cores, and then trying to use our custom vector intrinsics to speed up the normal floating-point instructions generated by the RISC-V compiler. Had extra time allow our plan would then be to also port the benchmark over to a comparable GPU and benchmark its performance versus our architecture over a metric involving theoretical FLOPS throughput for both architectures. Our vector and floating-point units have a performance counter which allows them to check how many of each operation they have performed, which we can compare to either our clock cycle counter or wall time for different benchmarks against an equivalent CPU or GPU implementation.

A. Mandelbrot Benchmark

Our first benchmark is hopefully the simplest, which is to compute the Mandelbrot fractal pattern on our architecture. Ideally, we have the Linux supervisor on the ARM core able to render that to a display in real time, but it would also be acceptable to simply save it to memory or a file to later validate the result. The reason we chose this algorithm for our first benchmark is due to its relative simplicity, with the value assigned to each pixel on the screen determined solely by its location, which provides a simple way to determine if the floating-point resources are saturated without relying on memory accesses. The only memory challenge with this benchmark is moving the pixel results out of each core as they are computed and assigning them to the final rendered image, which we could not complete in time.

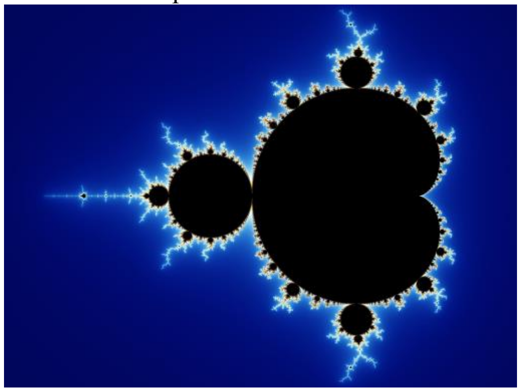


Fig. 1. Example rendering of the mandelbrot set. [2]

III. Architecture and/or Principle of Operation

After the design review we decided to move the Memory stage in the integer pipeline to be after the Execute stage instead of in parallel with it. This allowed us to not have to duplicate an adder from the Execute stage. We also looked at the vector instructions and decided to support only floating-point vector operations in our first revision instead of combined integer/floating-point vector operations.

Of particular note, our block diagrams follow the key below, which helps to distinguish between individual register stages in a design, what those stages do, and if those blocks in the diagram consist of multiple stages. The arrows are also clearly distinguished, with solid arrows using full flow control, and dotted arrows using no flow control.

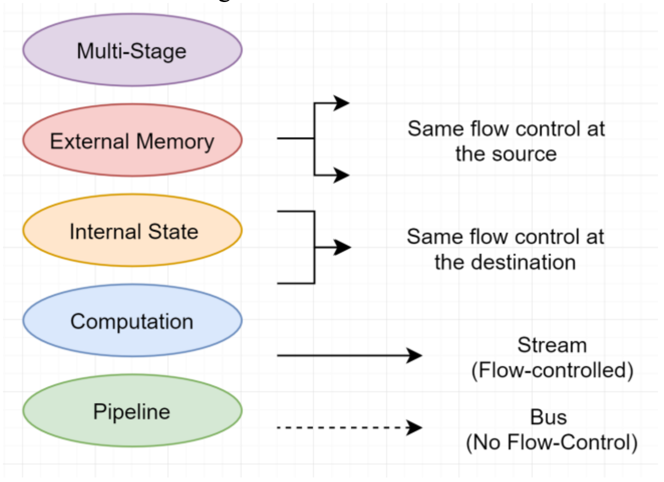


Fig. 2. Legend for processor and floating-point unit design.

A. Integer Pipeline

One of the biggest issues with FPGA logic is the routing delay, which for a long pipeline can far outweigh the delay through look-up-tables (LUTs). Tough combinatorial logic in an ASIC like a long carry chain are comparatively small problems in FPGAs, who have purpose-built carry-chains that are a much closer approximation of ASIC performance than the routing resources are. For this reason our CPU pipeline is very careful of having dependencies between stages, especially avoiding the complete forwarding approach taken in a lot of small five-stage processors. This allows the place and route algorithm to layout the design with less spatial considerations, which significantly improve the tools abilities to find better routing.

We also take special care to only have a single write-port into our register file, which allows the FPGA to use special purpose distributed RAM resources. Two write ports would force the FPGA to use normal registers to implement the register file, which would balloon the number of LUTs needed to decode the register file. Distributed RAM in the FPGA is also found in the FPGA in larger quantities than normal registers on a bit-for-bit basis. Given that a RISC-V register file is exactly a kilo-bit, this resource saving is super important to make sure we have enough integer-pipelines to service our floating-point vector logic.

Given the lack of complete forwarding a couple of small optimizations were then taken to improve instructions-per-clock (IPC) in the integer pipeline. First the Execute stage saves its last result indefinitely until a new result is produced, which the decode logic will instruct the Execute stage to use should the next operation be dependent on it. Second the Execute stage forwards its result, if it is purely a register-to-register operation, past the Memory stage and directly to the Writeback stage. This requires that the Writeback stage have logic allowing it to accept multiple result streams, and can write them to the register file in any order. To prevent a shorter latency, later executed instruction from pre-empting an earlier one, instructions writing to invalid registers are not allowed to execute until the register becomes valid again, or the previous invalidation was done through the Execute stage, and the next invalidation would be through the Execute stage too.

Given that the branch prediction table needs to be stored in the FPGA distributed RAM in order for the fetch stage to be single cycle we opted to make it small, but configurable in depth. We also allow for forwarding from buses that we found to be useful to forward from but that did not dramatically hurt timing. This limited and configurable forwarding approach further allows us to accommodate complications involved in FPGA implementation.

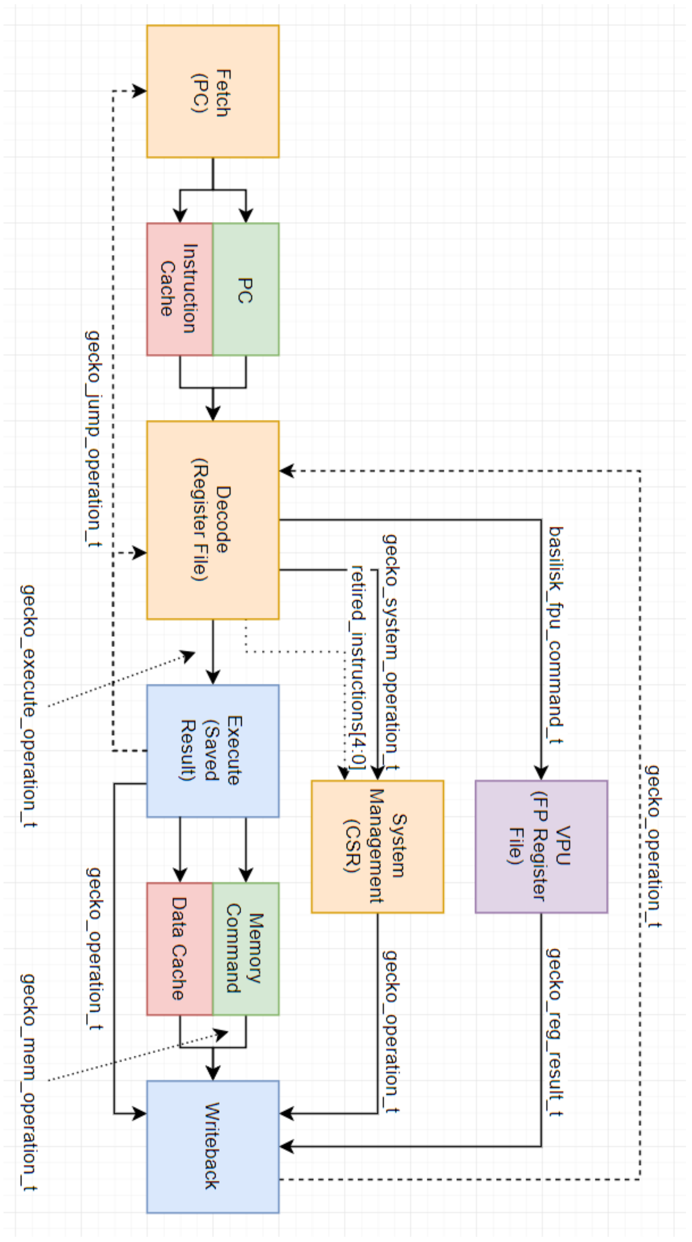


Fig. 3. Integer pipeline implementation.

B. Floating-Point

The FPU will fully support the RISC-V F-extension. In order to do that, we need it to be able to perform all of the floating-point operations, as well as interactions with memory and the integer registers. It also needs to be well optimized for the FPGA. The FPU design is important to the project because it is used for the vector extension. Also, the plan is to have it designed so that it can easily be integrated with the core. For this reason, we need it to use up as few resources as possible and be pipelined well so that it doesn't hurt our timing.

The FPU has 4 stages: Decode, Execute, Writeback, and Rounding. The Decode stage will receive the floating-point instruction and prepare the signals for the rest of the stages. One thing to note is that our FPU is like our integer pipeline in that it does out-of-order execute and in-order writeback. We have six execution paths (Addition, Multiplication, Division, Square Root, Encode, Decode, Memory). The encode execution path handles instructions who use data from an integer register to produce an output for a floating-point register (and vice-versa for the decoder). The memory path is the same as the one in the integer pipeline, but we use it to handle the floating-point load and store instructions since they rely on the floating-point registers. The Writeback stage selects which output from the Execute stage to write to a register (if necessary). Once selected, the Rounding stage will round the result if necessary before the result is written to the floating point register.

As mentioned before, performance of the FPU was crucial to the design. We wanted to seamlessly add in our FPU to the core such that it would not mess up our timing (don't want our critical path in the FPU). And since we wanted to vectorize our core, we needed the FPU to use up as few resources as possible. The FPU operations (add, multiply, divide, square root) required a lot of logic and time to compute. So, each operation path was pipelined into three stages. The first stage handles the resolution of the exponent. Then the second stage does the operation on the mantissa. And the final stage normalizes the result. As an attempt to save space we did rounding after the writeback stage since the rounding logic is the same for all of the operations.

For division and square root, the mantissa resolution requires a loop that would be expensive and time consuming to do in one clock cycle. Also, splitting up the loop into a bunch of pipeline stages would be expensive. We resolve this by having the results from one iteration of the mantissa resolution loop from the end of the stage back into the beginning of the stage for the necessary number of iterations (stalling the path as the looping occurs). This prevents the division and square root mantissa resolution from using up excess resources and hurting our timing.

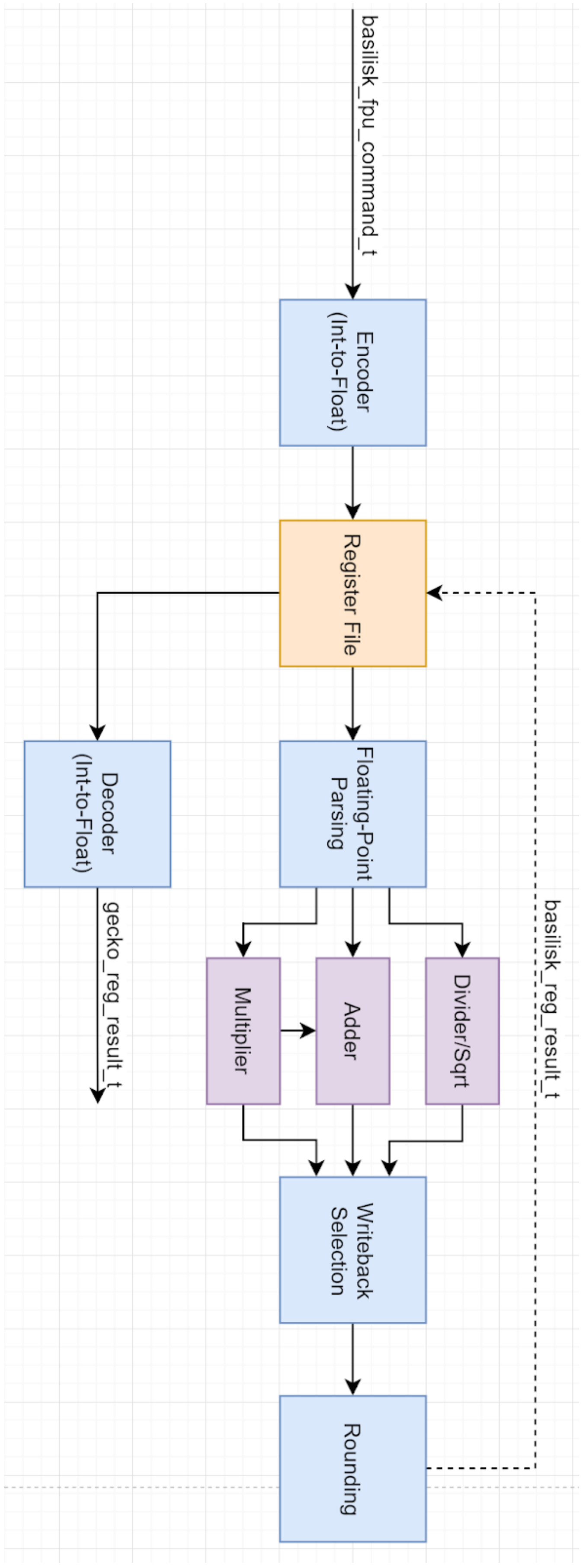


Fig. 4. Floating-Point Unit Implementation.

C. Vector

Bridging the gap between the FPU and the performance requirement we are trying to hit is the vector processing unit. Following the design of the Larrabee cores, our vector processing unit can work on 16 floating-point numbers at once, which requires each register file entry to be 512 bits wide. The RISC-V vector specification allows for these vector entries to wrap multiple floating-point registers, i.e. the first vector register of 16 entries might encompass the first 4 vector registers since each vector register is only 4 entries wide. This is mostly done to save on the use of vector registers for short vector operations, while also allowing for very wide vector operations on the same register file. At the expense of area but for design simplicity, a full 16-single precision float wide register file would only need 16 kilobits of distributed RAM, and our smallest FPGA will have nearly 1.8 megabits of it, which means a single vector core will only need 1% of the available distributed RAM. However, due to the accessibility requirements of distributed RAM inside the FPGA and the potential for an instruction to read from three separate registers, we actually needed three times this much distributed RAM to implement a single vector register file, or about 48 kilobits of distributed RAM.

Each VPU is designed primarily for addition, multiplication, and fused multiply-add. Division and square roots are also supported by both the VPU and the base FPU, but have a much higher latency and comparatively less resources are allocated to them. Two DSP slices are used per multiplication unit, and given that each FPU (of which there are 16 equivalents per VPU) is only ever issued a single operation per clock cycle, some of logic like rounding can be shared amongst the multiplication, addition, division, and square root logic for only a fractional reduction in performance.

In a best case regarding resource usage, we can expect to use two DSP slices for a fused multiply-adder (this unit could do either a single multiplication or addition a clock cycle, or a fused multiply-add in two clock cycles), which would allow us in the same FPGA we selected to build about 360 fused multiply-adders. Given our vector width, we could then expect to make about 11 VPUs with all the FPGA resources.

However, due to limitations in how other operations get mapped to FPGA resources, we wound up being limited primarily by the number of LUTs in the FPGA and not our DSP slices, but we will discuss this later.

We also realized during implementation that the RISC-V vector specification was far more complicated than we were expecting it to be, mostly involving how vector operations were allowed to be strided across the file. We eventually settled on not implementing this part of the specification, which was feasible since we were generating all of the assembly ourselves.

D. *Interconnect*

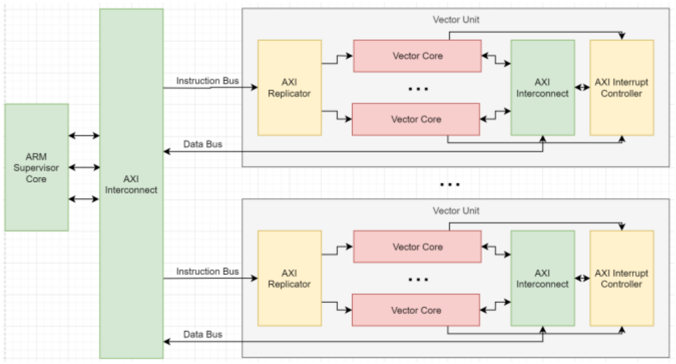
Each of the combined vector cores has a separate interface for both instruction and data memory. These interfaces are relatively simple request/response memory interfaces that connect to two different scratchpad memories. This separation is warranted since each core is never going to edit or issue its own instructions, instead getting those issued from the supervisor. First, given the small size of both memories, this allows each core to have somewhere between one to two cycles of latency without a cache, reducing logic consumption in the FPGA. Each memory also uses the dual-port configuration of the block RAM in the FPGA, but there are four separate entities that need access to it, so we implemented a memory multiplexor on each port. First, instruction memory and the supervisor get access to a single port, since each is probably not going to accessing it at the same time, as a core is in idle while it is being loaded. The data memory and floating point data memory both share the other port, which makes sense from a bandwidth standpoint as only a single instruction can be issued a cycle that would need to access this memory.

Instructions can be issued to a group of cores at a time, with the write requests being replicated across the group by the interconnect logic. This behavior is maskable by the interconnect, so a fine-grained group of cores can be issued the same behavior all at once. This is useful for instance when needing to perform operations like vertex transformations, as all of the cores are executing the same instructions on different data. This mirrors the parallelism that GPU shader cores implement, but still provides separate instruction memories to each core.

Each core is managed by polling memory addresses in its data memory for status updates, as well as having a small interrupt receiver from the core should it execute an undefined instruction or try to access memory outside of its scratchpad. To recover from this state, the supervisor has access to a reset controller for each core, which allows it to put any individual core into reset. All cores in fact start in reset, and need to be brought out of reset when the system starts. A core brought out of reset will then proceed to start executing instructions at the beginning of its instruction memory.

The request/response memory interface used by the individual cores is converted to AXI at each core's data memory so that independent memory operations can occur while the core is running. This independent AXI slave device can either be written to directly by the supervisor through the interconnect, or through a DMA engine that is provided per group of cores. This DMA engine sits on the same interconnect as that group of cores, which gives it faster access times for moving data between individual cores in that group, but at the expense of latency for moving data from the core to main memory. The DMA cores and AXI interconnect are Xilinx IP catalog cores.

Our block diagram that we were using to test out a single instance of the core is shown below. In the design we have the core, an AXI interconnect to bridge all the blocks together, some ILAs in order to debug the logic inside of the FPGA, and an AXI FIFO so that we can read out the messages that the FPGA is printing out.



E. *Software*

The supervisor core that is dispatching data and instructions to the vector cores is going to be the ARM core on the FPGA development board that is running Linux. This allows for much easier access to resources like networking and video output on the development board, and potentially even using high level languages like Python to manage the operations of the individual vector cores.

In order to run code on the vector cores it first needs to be compiled from C to RISC-V machine code, which there already exists a toolchain for. However, an additional complication is using our VPU logic from C. The single-data FPU implementation can be targeted by the RISC-V compiler, but the vector extension is not mainstreamed yet. To solve this we need to add the vector instructions to the RISC-V assembler, then use a set of intrinsics to interface that assembly into our C code. Luckily, this means we do not have to modify the RISC-V GCC implementation, and instead just point it at a different assembler. In a perfect world a compiler would exist already that could infer the use of a vector operation from a similar programming construct like a for-loop, but given even the state-of-the-art in compiler research we are mostly left to write vector code ourselves. This allows us to somewhat leverage the unique differences in our architecture most effectively, particularly in our we load and unload our vector core.

We spent a lot of time working on editing gas (the GNU assembler), so that GCC would respect the registers that our instructions required and still generate the correct code. This was much more involved than we expected and required editing a lot of different parts of the compiler toolchain. In the end we did have a properly functioning assembler that could assemble and avoid dependency issues in a whole new set of registers.

For our integer pipeline, we used ECALL and EBREAK as assembly macros, but we leveraged the lessons we learned there to handle our overlaid vector-float register file. This required our assembly macros to properly indicate to GCC which registers were being clobbered and which ones were not.

The supervisor code is going to be in charge of managing messages from each of the vector cores and understanding any exceptions that they generate. We wanted to implement a small set of software libraries for both the vector cores and supervisor to allow for utilities like mailboxes between the two, as well as allowing the vector cores to update the supervisor when they need common operations like a DMA transaction done.

IV. Management

A. Schedule

See back for schedule.

B. Bill of Materials and Tools

Our project uses two FPGA boards to demonstrate the architecture, the Ultra96 and a ZCU102. Both are Xilinx Ultrascale+ development boards with an integrated quad-core ARM processor to run our supervisor code on. We are using Xilinx Vivado to build for the boards as well as simulate a lot of our logic. We are also using VCS as a resource for simulating some of our logic due to its ready availability in the ECE clusters.

The Ultra96 is a small credit-card sized board which costs about $250 and is a small-scale demonstration of our architecture. It will help us show that even with a small number of vector cores we can accelerate our benchmarks, especially for embedded applications.

The ZCU102 is a much larger and more expensive development board, but has the same ARM cores next to the FPGA, and we are using it to show that our architecture can scale. The ZCU102 is otherwise the same FPGA fabric as the Ultra96, just bigger, which should help in migrating the design when we get to that point.

Our software toolchain consists primarily of the RISC-V GCC compiler along with our modifications to the assembler. The ARM cores will run Linux that we are booting using the PetaLinux kernel provided by Xilinx, as well as the Pynq libraries

V. Related Work

Given that we are trying to replicate Intel Larrabee but with a different ISA, it is the closest related work to our project. Some notable differences were that they had actually fabricated their architecture into custom silicon and demonstrated it that way, while we are limited to presenting our architecture in an FPGA and optimizing it for FPGA-specific resources.

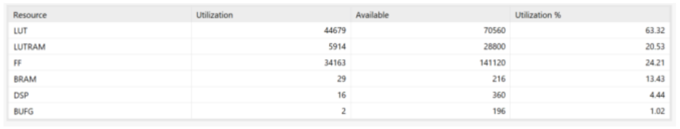
Intel would later abandon using the architecture they developed for graphics and instead upgrade and rebrand it to be a general computation accelerator called Xeon-Phi, which powers many of the fastest computers in the world. Xeon-Phi differs from our architecture in that it is a cache-coherent processor amongst its cores and can otherwise boot an operating system on its own, without a host computer acting as a supervisor, although it is usually run with a supervisor in most applications.

While we were working on our design the vector specification changed underneath us a few times, in some large ways, so we ultimately had to work with an older and limited version of the specification in order to be able to reasonably implement it in our FPGA.

Fig. 5. (Top) Xeon-Phi accelerator card, [8] (Bottom) Larrabee engineering sample [9]

VI. Results

Our goal is to implement a software driven approach to highly parallel computing in an FPGA, using common resources in the fabric, and specifically targeting the constraints



Resource	Utilization	Available	Utilization %
LUT	44679	70560	63.32
LUTRAM	5914	28800	20.53
FF	34163	141120	24.21
BRAM	29	216	13.43
DSP	16	360	4.44
BUFG	2	196	1.02

Fig. 6. FPGA Implementation Usage

While this would seem to indicate that we might be able to fit in two, maybe three cores if we used 4-wide compute units, the area consumed on the FPGA paints a different picture.

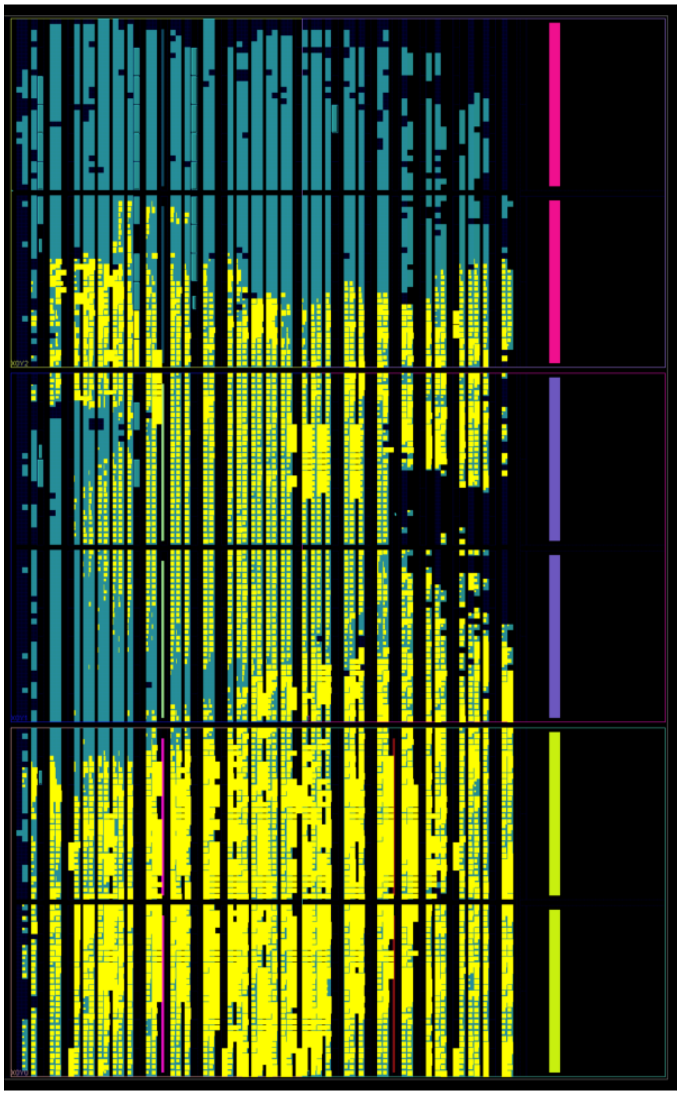


Fig. 7. FPGA Internal Layout (Our core is in yellow, various debug logic is in green)

While only 27% of the FPGA is being used according to the report, this is really indicating that routing congestion is going to be more of an issue in the final design, as the tools feel the need to spread everything out, taking over most the FPGA. The reports also indicated that some routing congestion still remained, mostly having to do with the vector register file, which given its large fanout we mostly expected. Despite this, our pipelining let us maintain a 200 MHz clock frequency at least, which while less than our original 300 MHz goal, mostly came from the integer decode pipeline, with the multiplication normalization being a close second, but far easier to pipeline in the future. Given that we expected the actual math of the floating point operation to be our critical path and not the integer pipeline, we feel that our floating pipelining was largely successful.

Our integer core was measured on the Dhrystone benchmark and we are able to get at least 0.9 DMIPs/MHz even with the double cycle memory we needed in order to meet timing inside the FPGA. With single cycle memory to reduce the branch penalty our core was performing at about 1.1 DMIPs/MHz. Our FPU had different metrics, and we have not tested it across the benchmarks we originally set out to due to a lack of time to complete them.

VII. Summary

Our goal of this project was to implement a software driven approach to highly parallel computing in an FPGA, using common resources in the fabric, and specifically targeting the constraints of the FPGA to improve our performance. We wanted to have a custom, FPGA-optimized RISC-V processor and a vector co-processor for it, along with the necessary interconnect to attach these vector cores to a supervisor core running Linux. A variety of different benchmark applications will then let us determine if our architecture effectively met its goals or determine what computational resource slowed down our performance. We aimed to demonstrate that the architecture proposed by Intel in its Larrabee project was feasible, and is potentially an even better idea now given the development of open source, RISC ISA specifications.

Among things that we learned was how involved it would be to modify GCC to even assemble different instructions, completely leaving alone compiler. While we did eventually get this working it took a lot of time in our schedule that we ultimately did not have to spend. We also learned about some more weird disconnects between the Xilinx simulator and the actual implementation tools, which led to a couple of weeks of debugging something that was not working inside the FPGA but worked completely fine in the simulator. Another big issue that hurt our progress was even being able to use an ILA inside the design while running code in Linux to test the design at the same time. We ultimately worked out that JTAG access was causing the CPU to fault while in the idle state, and the final solution was simply to use a shell script disabling the CPU idle states in the first place.

While we did not succeed in getting our core to run the benchmarks we set out to at the beginning of the project, we did succeed in getting our core to run on the FPGA, run basic floating point operations successfully and print their results, and then finally run vector operations and extract those results as well. Our vector unit had a configurable compute width as well as vector width, and having the ability to configure both let us try to fit in the FPGA more effectively. If we had had more time we would have tried to work on potentially sharing complex, non-DSP logic like dividers and square-root logic, so that we could have fit more cores into the FPGA. We would have also tried to get more rigorous tests of the vectorization performance done, that would have compared it to other floating-point RISC-V cores, as well as comparing our vector performance to normal floating point code.

References

[1] https://en.wikipedia.org/wiki/Larrabee_(microarchitecture)
[2] https://en.wikipedia.org/wiki/Mandelbrot_set
[3] https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-tray-tracing/ray-tracing-practical-example
[4] https://learnopengl.com/Advanced-Lighting/Deferred-Shading
[5] https://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.html
[6] https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/ray-tracing-practical-example
[7] https://en.wikipedia.org/wiki/Deferred_shading
[8] https://www.amazon.com/Intel-Xeon-Phi-7120P-Coprocessor/dp/B00FKG9R2Q
[9] https://www.vrandfun.com/check-intels-graphics-card-prototype-larrabee/

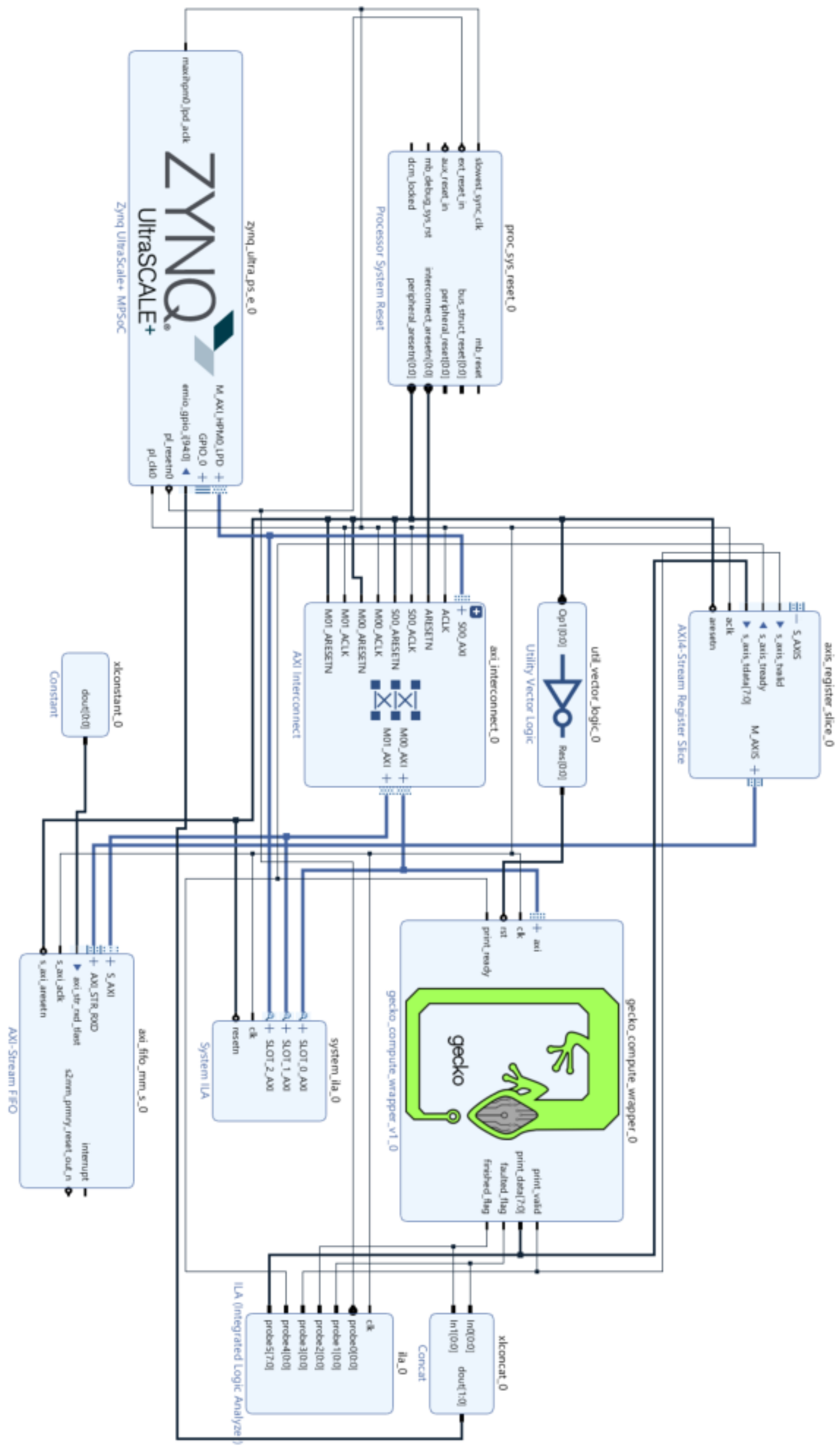


Fig. 8. Main system/interconnect diagram.

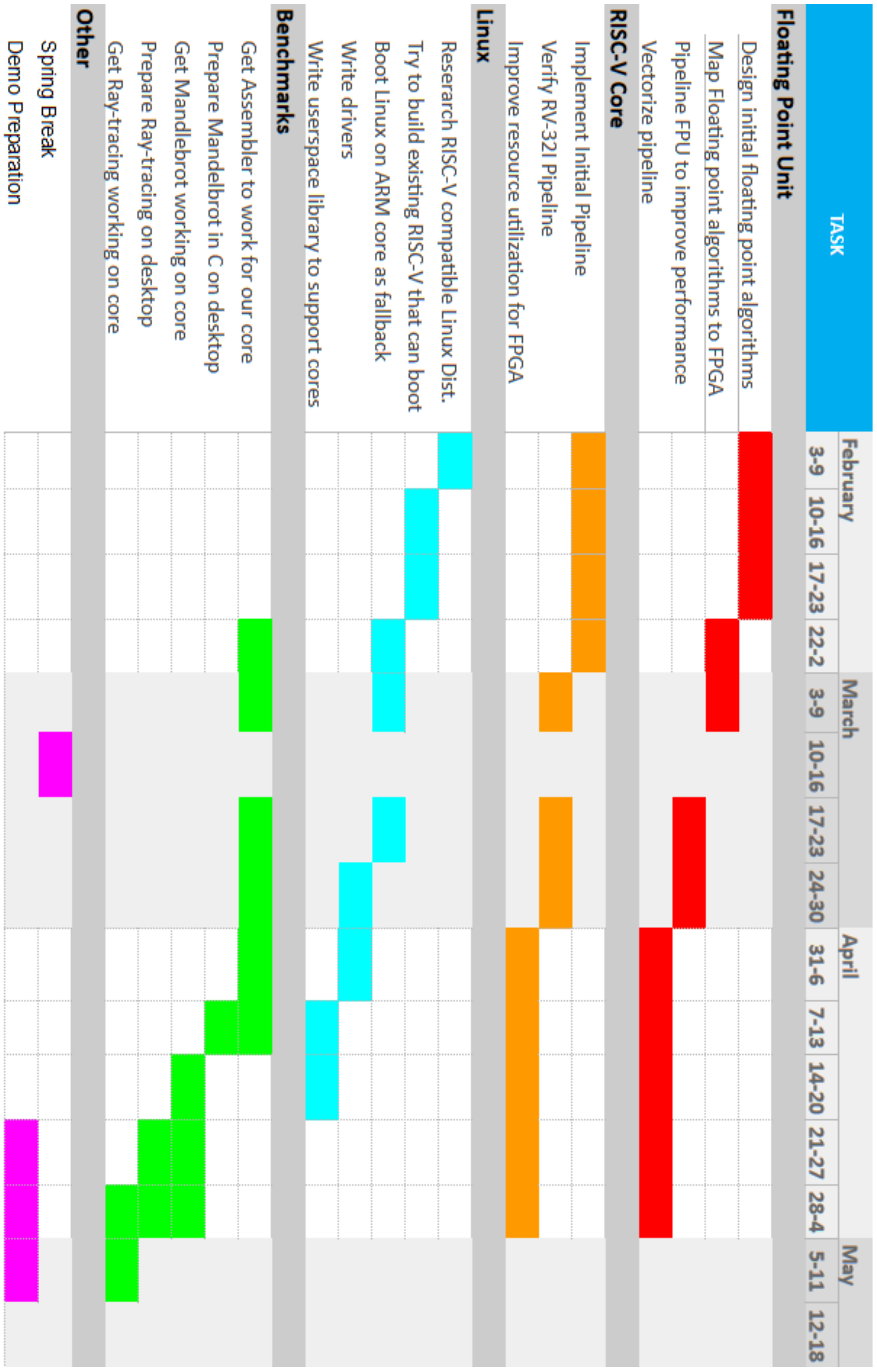


Fig. 9. Gantt Chart/Schedule for Project