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 are replicating and improving 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 are using 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. 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 are going to be implementing 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 is 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

Our primary design requirement is 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 want 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 allows 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 are also using two, and if time allows 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 are going to write them in C and then first run them on the ARM core alone to simply show that they work. Then we will try to make sure that they are being accelerated by the NEON FPU next to the ARM core, which should hopefully show some improvements in performance. Next, we are going to work on porting the code over to our architecture, which should 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. Should 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 will 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 running 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.

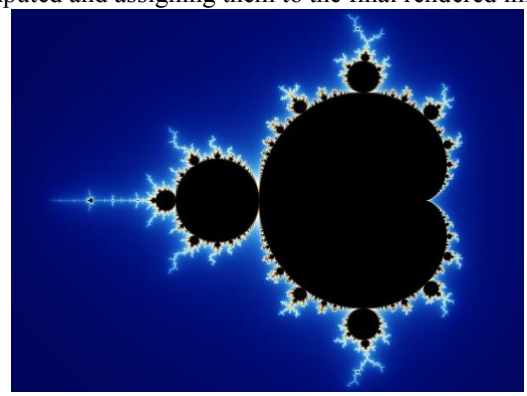


Fig. 1. Example rendering of the mandelbrot set. [2]

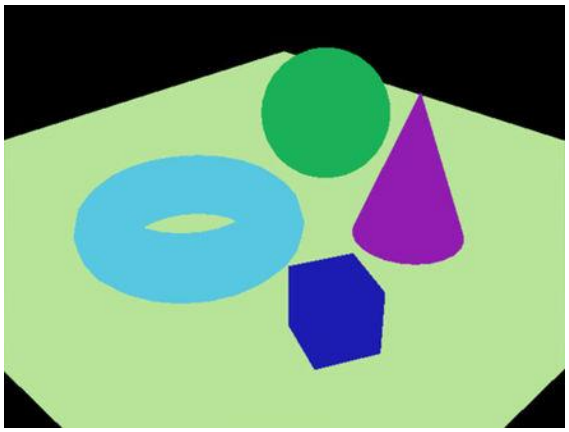
B. Ray-Tracing Benchmark

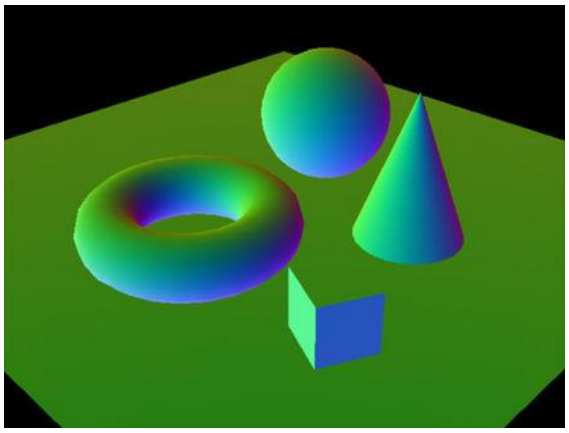
Our next benchmark is to implement a simple ray tracer using our cores, which should provide a good benchmark of our floating performance as well as how efficiently we can move data from one shader core to the next. As the ray from the light source traces its way across the screen it moves around a lot, potentially from one side of the view frustum to the other. If a shader core is assigned an individual ray it would have to constantly get information for different parts of the scene. Likewise, if our software stored information for a section of the scene on a core-by-core basis the individual cores would have to share rays between them, managed by the supervisor logic. Simple ray tracers are also available in 500 lines of C code or less, which should hopefully give us a clear reference implementation despite the apparent complexity of ray tracing.

Fig. 2. Example ray traced scene with light source and partially transparent objects, example written in only 300 lines of code. [6]

C. Traditional Rendering Benchmark

The last benchmark we would like to implement would be a traditional deferred rendering pipeline. This is what GPUs are built to do primarily, and they are highly optimized with special hardware to do so. It is unlikely that we can compute a scene, say the Stanford Bunny with a single light source, as efficiently as even the GPU integrated with ARM processor on the FPGA development board, but it would be a good point of comparison to show that even though our architecture sacrifices some abilities since it is general purpose it can still effectively perform a traditional workload. This is also the benchmark we are least likely to have the time to implement.





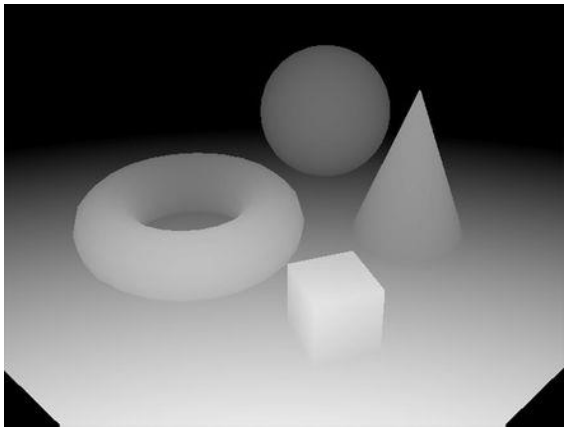


Fig. 3. Example outputs of a deferred shading pipeline, with color, depth, and surface normals which can then be composited in parallel, pixel-by-pixel. [7]

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.

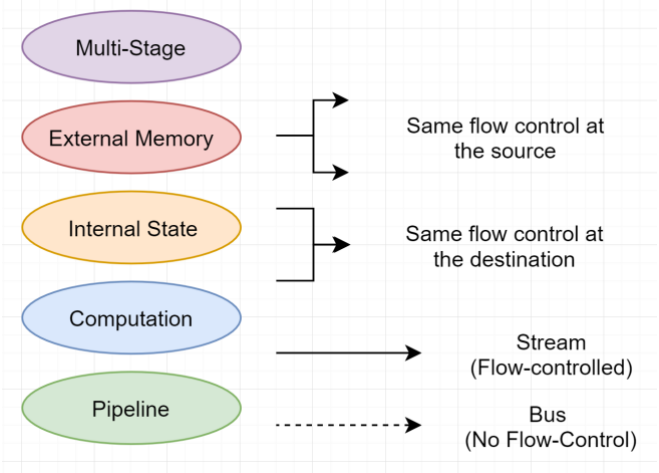


Fig. 4. Legend for processor and floating-point unit design.

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 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 these cores are not designed to be branching heavy and instead focus on math, some shortcuts were taken to improve performance and core simplicity at the sacrifice of potential IPC. When a conditional branch is encountered, any register-to-register instructions are still allowed to be execute but come with a flag that will prevent them from being written back to the register file until the branch is resolved. Given that a block RAM would have to be used in the FPGA for storing the branch prediction table, the pipeline currently assumes that all branches are not taken, but if FPGA resources remain available and time permits then branch prediction can be added later.

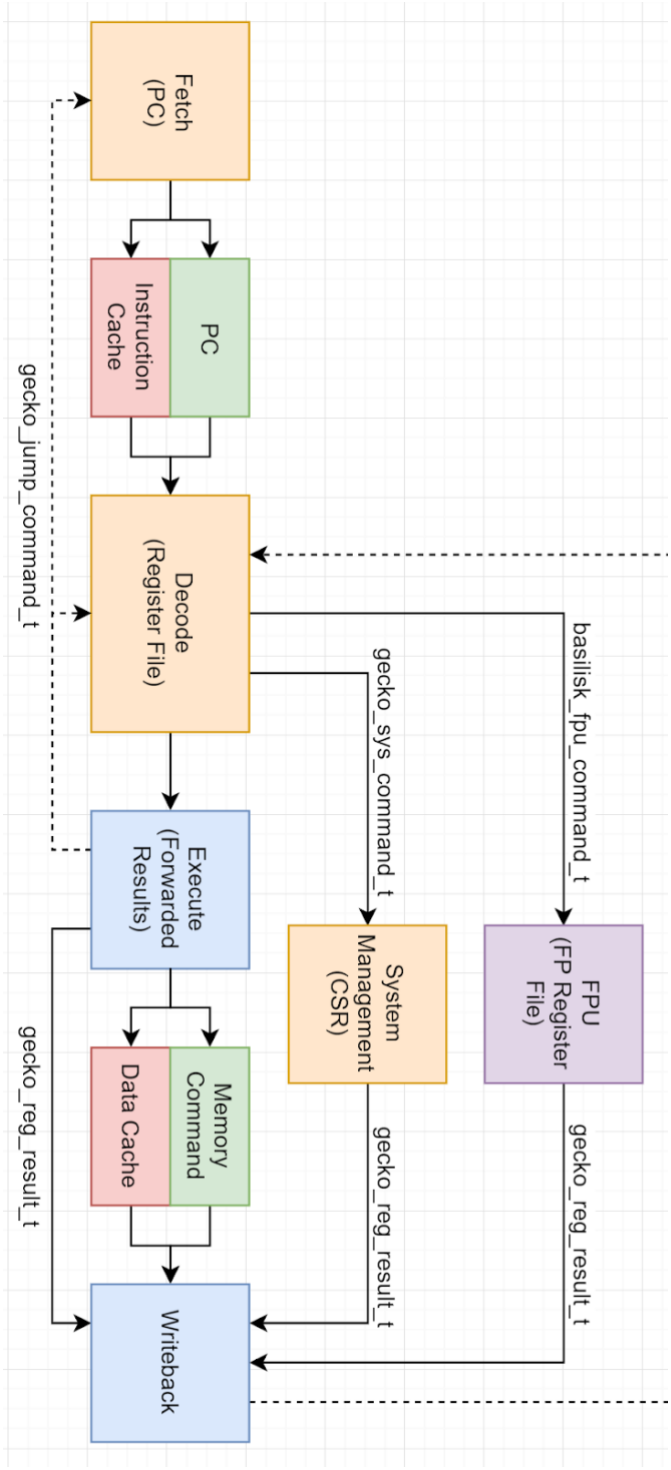


Fig. 5. Integer pipeline implementation.

B. Floating-Point

We know that the system will need a good floating-point unit in order to perform floating-point arithmetic well. Most of the improved performance will come from the RISC-V core design and the memory optimizations, but we shouldn't let the floating-point unit hinder the performance of the system. We aim to have a floating-point unit that can correctly do addition, subtraction, multiplication, division, and square root on the IEEE 32-bit floating-point format. We also plan to have it support 4 different rounding modes: round up, round down, round to zero, round to nearest even.

To speed up the floating-point unit, we will try to have it target the DSP slices on the Ultra96 FPGA. These slices are dedicated hardware made to speed up arithmetic operations. They are capable of 48-bit addition and 27-bit by 18-bit multiplication. Using the DSP slices to do the mantissa operations should be faster than any other implementation synthesized on the FPGA. Because of the 48-bit addition, this should be easy to integrate with our addition, division, and square root algorithms. However, we cannot directly do a 24 by 24-bit multiplication. Our solution to this would be to use two DSP slices to perform the mantissa multiplication. The first multiplication will be the upper 6 bits of the first operand multiplied by the second operand. The result from this will be left shifted by 18. The second operation would be the lower 18 bits of the first operand multiplied by the second operand and added with the earlier shifted result. Getting the Vivado tool to synthesize our design with as many DSP slices as possible will allow for optimal floating-point arithmetic.

The operation time for each operation will vary a lot. Regardless of the operation, parsing the inputs and normalizing are done the same way regardless of the operation. Solving the exponent varies based on the operation but should take around the same amount of time to complete. Rounding the result is done the same for all operations but varies minimally based on the type of rounding. The mantissa calculation is the most troublesome part of the FPU. Multiplication, division, and square root require some type of iteration and will take significantly longer to complete. This is the nature of doing these operations and the use of multiple DSP slices should mitigate the overall time to do the calculation.

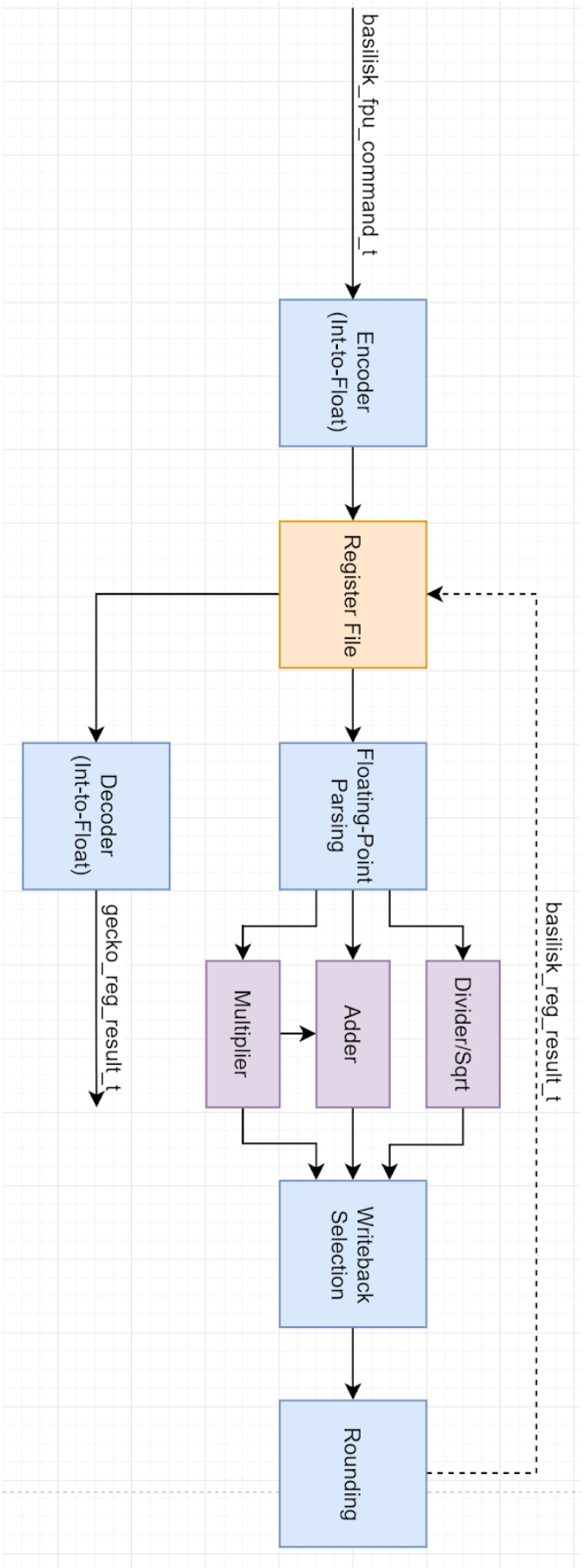


Fig. 6. Floating-Point Unit Implementation.

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.

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. A single DSP slice is 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 a single DSP slice 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 22 VPUs with all the FPGA resources.

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, which means each core owns an interface to its memories, and the supervisor owns the other port.

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 but could be rewritten if resource consumption becomes an issue and time allows.

See back for full diagram.

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.

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 are going 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 going to use 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 going to try to boot using the PetaLinux kernel provided by Xilinx.

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.

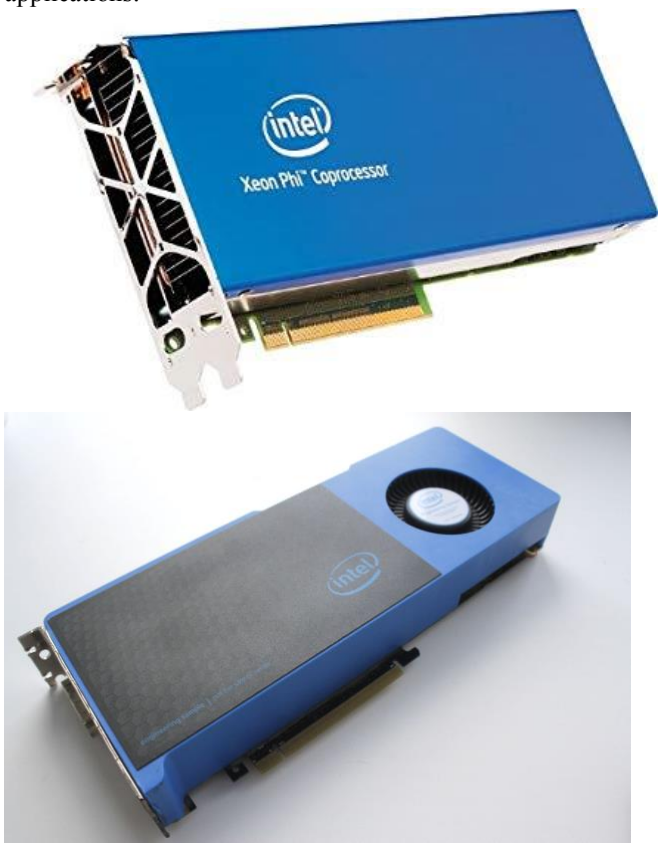


Fig. 7. (Top) Xeon-Phi accelerator card, [8] (Bottom) Larrabee engineering sample [9]

VI. Summary

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 of the FPGA to improve our performance. We will have implemented 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 aim 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.

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/

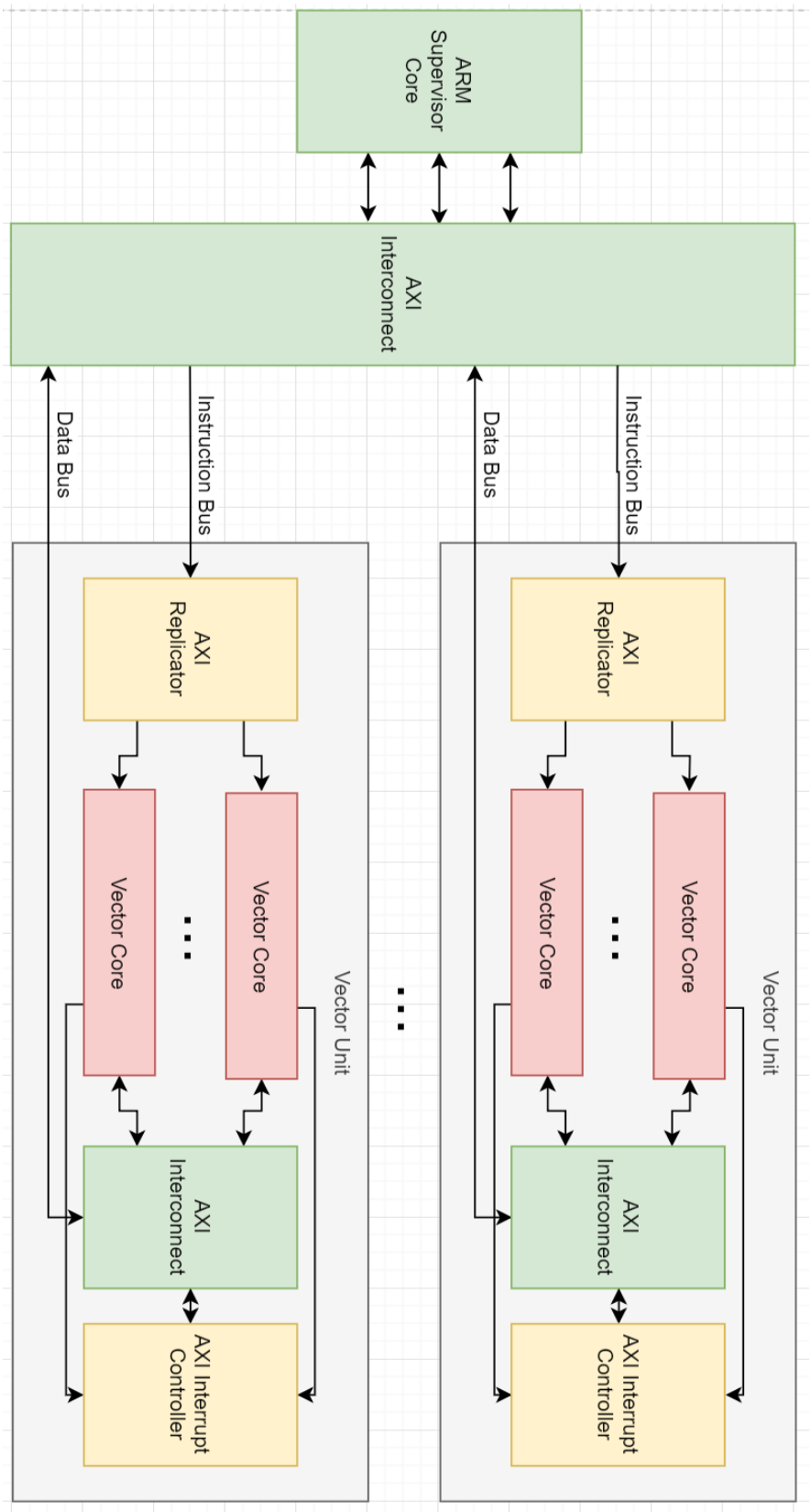


Fig. 8. Main system/interconnect diagram.

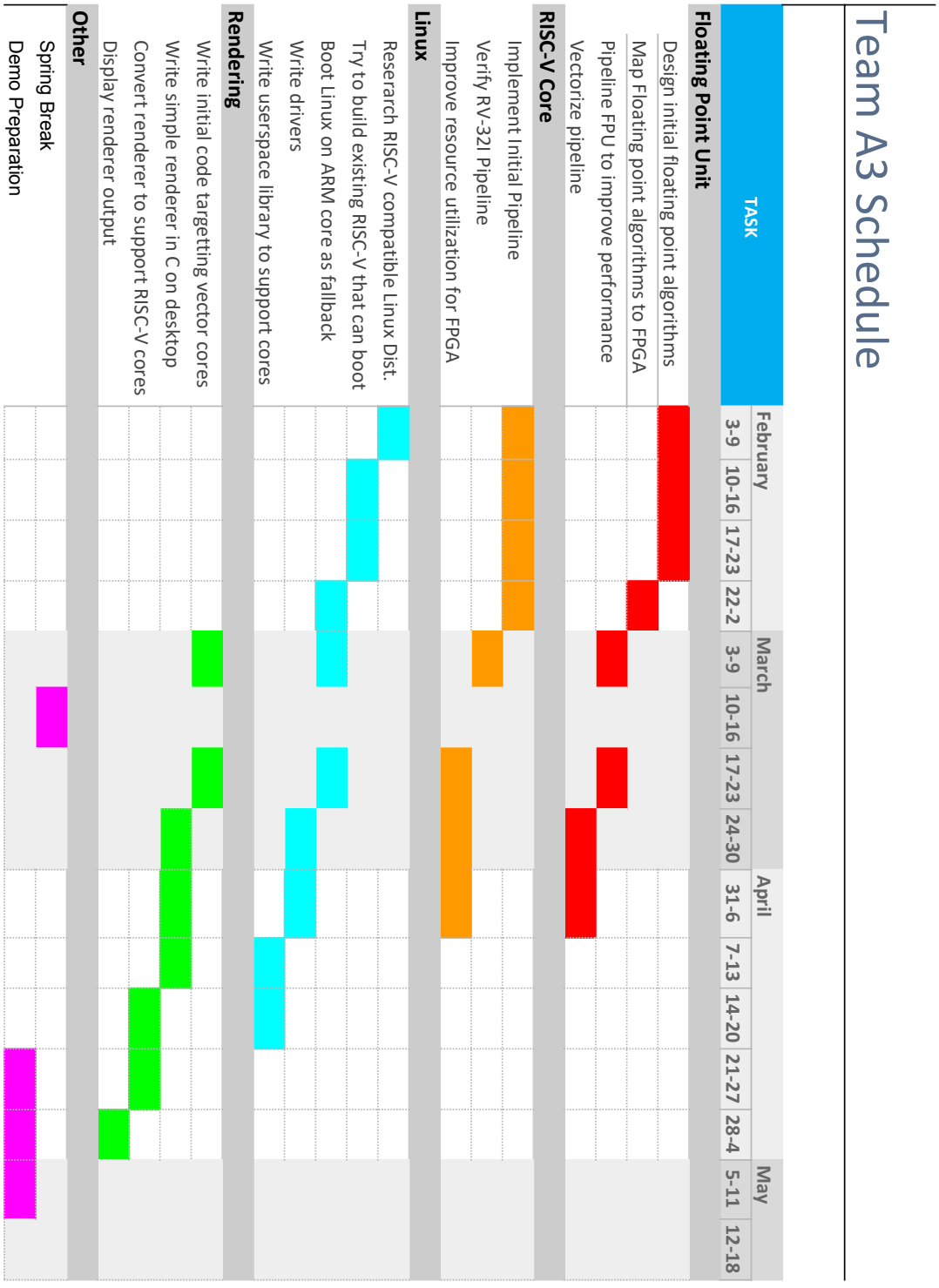


Fig. 9. Gantt Chart/Schedule for Project