

In-Flight FPGA Task Scheduling for Prime Counting in Ranges

1st Alex Shin

EECS, MIT

Cambridge, MA, US

shina999@mit.edu

2nd Haani Mahmmood

EECS, MIT

Cambridge, MA, US

haani@mit.edu

Abstract—This work presents the foundation of an FPGA system for counting prime numbers within arbitrary 32-bit unsigned integer ranges using an in-flight task scheduler structured as a balanced multiway (BMW) heap. The design integrates on-chip distributed RAM for internal FIFOs, off-chip DRAM for task storage, as well as UART transmission and reception for demonstration and evaluation of the system’s performance via software. The proposed system was demonstrated to give a X speedup when compared with a FIFO reference.

Index Terms—Digital systems, Field programmable gate arrays, task scheduling, prime counting

I. INTRODUCTION

Task scheduling is a highly relevant concept in hardware, especially in the context of real-time systems with large dataflows where optimizing for time, cost, and throughput metrics is crucial. It is generally a hard problem to solve due to the dynamic nature of the problem - the optimal order of scheduling can not be computed ahead of time. So, a good task scheduler has the job of managing task loads using assigned priorities (usually with heuristics such as expected time to run, or the cost of missing deadlines) that maximizes use of the available computation resources.

BMW heaps are one such example of a task scheduling system. They are able to achieve high throughput with the use of low-latency RAM, and their modularized, pipeline-friendly architecture. An example of their use is given in [2] for network packet sorting.

For the purpose of implementation, we paired our task scheduling system with execution units being prime counters for a given range and deadline. Prime counting is a classic problem in the fields of math and computer science, and although there are good estimates of the density of prime numbers, to get an exact count in a range requires running a deterministic algorithm. Each computation takes a significant number of clock cycles, especially in a memory-limited system where caching results is not an option. This provides a good way to evaluate the effectiveness of the scheduler.

II. SYSTEM OVERVIEW

A. High-Level Architecture

Our system spans:

- a memory management module using the opensource UberDDR3 memory controller [4] for off-chip DRAM usage

- a software interface module for UART communication to load and unload data between hardware and software
- a task scheduler module including the BMW heap placeholder, task fetcher, priority encoder, and task dispatcher
- multiple prime counting modules that are able to run in parallel

Each module uses an AXI-like handshake mechanism where possible. All modules apart from the memory controller run on a 100 MHz clock.

The full project source code can be found on Github [1].

B. Timekeeping

We use an unsigned 64-bit number to keep track of time. While the maximum allowed time will never practically be reached, this method allowed us to add a concept of ‘no deadline’. For big enough ranges, it is not possible to compute the result in a reasonable amount of time, so no imposed deadline should be a signal to a good scheduler to put it off until everything else is completed, rather than immediately issuing it, and stalling an execution module.

This is implemented as a simple event counter that is constantly running in the system’s top level.

III. TASKS AND EXECUTION RESULTS (HAANI)

Each software-provided task is structured as shown in figure 1; each execution result which is communicated back to software is structured as shown in figure 2. The task ID is omitted for the former, as that is just its address in memory, and is appended in hardware. It is necessary to include when communicating results as each task may be executed in an arbitrary order.

For the tasks, we need to specify both start and end times, and the required deadline. unsigned 32-bit values for start and end give us ample space to do so. Additionally, the memory controller uses native 128-bit data addressing, so it aligns well with our DRAM usage. For results, the choice of 128-bits is more for convenience; we defined a `word_t` 32-bit logic at the start of the project, and found it easy to use for consistency and readability.

When a task deadline is not met (either during execution or at the start), we immediately set the completion time to 0 and pass it out. Any valid task (even if the range is $[0, 0]$) is ran for more than a cycle, so this is fine to do.

32 bits	32 bits	64 bits
Range Start	Range End	Deadline

Fig. 1. Task breakdown.

32 bits	32 bits	64 bits
Task ID	Prime Count	Time of Completion

Fig. 2. Execution result breakdown.

IV. DATA TRANSFER, STORAGE, AND USAGE (HAANI)

To bring the effectiveness of the scheduler to light, a large number of tasks is needed, and at a pace faster than a naive first-in-first-out scheduling system would be able to manage. That motivated our use of off-chip DRAM for holding the tasks.

Our system has two stages. First, we initialize it with all the task data, which is sent from the computer using UART at a BAUD rate of 2,000,000, with a sleep time of 500 μ s in between each 16 byte transfer. During this stage, only write requests are issued to the UberDDR3 memory controller. There are multiple distributed RAM-based FIFOs along this data path to ensure no data is dropped, while also achieving a short setup time.

Once the memory is filled (indicated by an onboard LED), we start running the scheduling and execution modules using a button trigger. In this stage, the scheduler makes only read requests to the controller. As tasks are completed, results are routed back to the software interface module, and transferred to the computer using UART at a BAUD rate of 2,000,000. These transfers can be much slower than the system itself, so this datapath also has FIFOs to manage that. At some points, the task fetching also needs to be throttled to ensure all results are transferred back.

This data management system allows us to decouple execution from slow data transfers between hardware and software to a good extent.

The associated modules here are:

- `uart_rx_w` and `uart_tx_w` which implement UART transmission and reception for 16 bytes at a time
- `sw_interface_top` which wraps the UART modules, and interfaces with memory, the execution units, and holds associated FIFOs. It also provides the `software_stall` signal discussed above
- `mmu_top`, which is where the memory controller lives. We employ the same `clockdomain_fifo` file as used in 6.205 lab 6 to deal with the necessary clock domain crossing between the controller and everything else
- the scheduler's `fetch`, which keeps track of memory requests and responses while the system is running

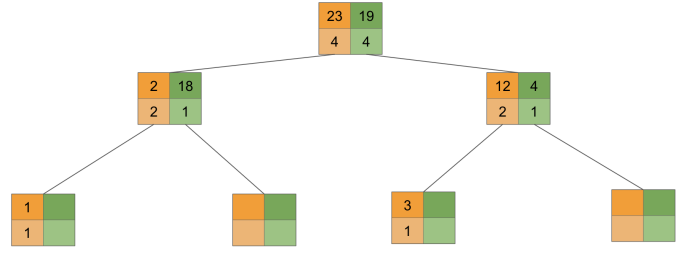


Fig. 3. Example BMW Tree with 2 Ways and 3 Layers.

V. BMW HEAP QUEUE (ALEX)

The (Balanced MultiWay) BMW heap is a binary min-heap storing task nodes indexed into block RAM, enabling efficient priority queue operations. Deadlines are used as priority keys. Tasks with no immediate deadline receive a maximal timestamp to ensure background scheduling. Priority errors could lead to missed deadlines, which reduces system reliability but is acceptable in this experimental context. The heap can have a task pushed in through the top, and the highest priority can use pop to be returned.

For the implementation of the hardware, very different from theoretical tree. Instead of branching a ton with registers, using a controller called a Ranking Processing Unit (RPU). Each one is connected to a BMW Tree's layer which is represented contiguously in BRAM. Figure 4 shows how it works, where the higher layers near the root have less elements to store so the BRAM is smaller, and later layers have a larger BRAM.

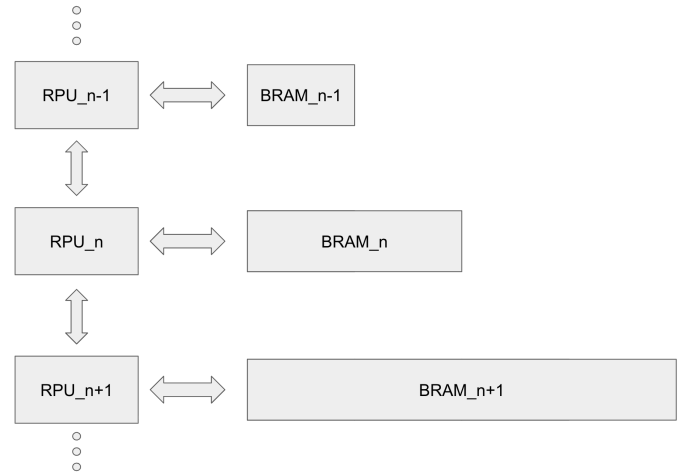


Fig. 4. Initialization waveform.

How priority encoding is implemented in the scheduler is an important design parameter. Very conservative priorities (e.g. purely based on range size) would cause a lot of tasks with short runtime but high urgency to fall through and fail. Thus the choice to use an unsigned 64-bit integer is deliberate: even though the maximum possible time will never practically be reached, it allows us to put more consideration into encoding priorities. The encoding mechanism will primarily be determined via software simulation.

For the sake of an initial full-system buildout, our FPGA builds currently only use a basic FIFO mechanism in place of the BMW heap.

VI. PRIME RANGE COUNTERS (HAANI)

There are three parts to our execution units, which use:

- a fully-pipelined 33 stage division module
- a primality checker module which wraps a single divider to check if a given number is prime
- a range prime counter module which wraps a single primality checker to count the number of primes in a given range

There are a total of 7 instances of the range counter which are wrapped by our `workers_top` module. This module is able to issue and retire a single task and result per cycle using gated signals. It provides a clean interface in our top level file.

Each execution module runs in parallel, so there can be at most 7 independent tasks being actively worked on.

There are optimizations that can be made here in terms of the counting/checking algorithms, however it was not possible in the scope of our work.

VII. VERIFICATION (HAANI)

This was an extremely important of the project's development. Each module was individually tested for the expected high-level behaviour, including correct value calculations, and for data handoffs handshakes (particularly ensuring that no piece of data got lost or double-counted). Additionally, every simulatable part of the project was put into a `sim_top` module, which allowed us to test the system as a whole. The only part that wasn't included here was the memory module. This allowed us to keep hardware debugging to an absolute minimum, and gave us the ability to quickly track down issues (which were found mostly in the memory module).

We used the `cocotb` Python library, along with `icarus` and `verilator` backends for all our simulations. Additional libraries that were used include `sympy` for their `isprime` function, and `cocotb_bus` for the handshake-based drivers and monitors we used for a large chunk of our simulations. The latter are Python classes slightly adapted from code provided by [5].

Waveform captures from a simulation run of `sim_top` are shown in figures 3, 4, and 5. Note in particular the timescales of the captures, as well as:

- `sw_interface` making memory write requests at the start with `uart_rxd` data coming in
- `memory_full` and `running` going high, after which the scheduler starts making memory read requests
- The worker completion signal in `tx_trigger` followed by `uart_txd` data going out

VIII. HARDWARE RESOURCE USAGE (HAANI)

We include the hardware resource usage of our final build in tables 1 and 2. The parallel execution units make up most of the logic usage.

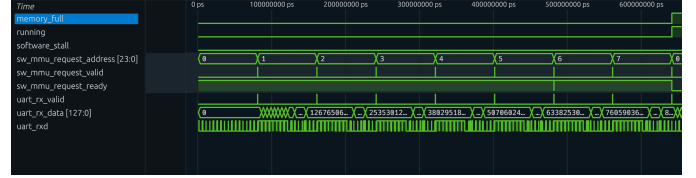


Fig. 5. Initialization waveform.

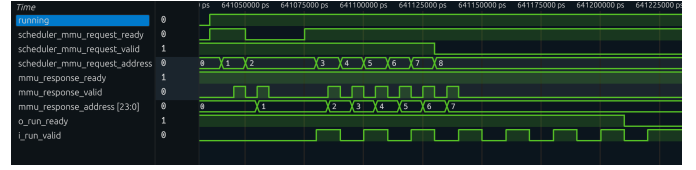


Fig. 6. Scheduler fetching waveform.

The BRAM usage is purely from the clockdomain FIFO. All the other FIFOs used distributed RAM.

BRAM usage otherwise was mostly limited to leave space for the BMW to maximize its size.

TABLE I
LOGIC AND DSP UTILIZATION

Resource	Used	Available	Utilization (%)
Slice LUTs	19,312	32,600	59.24%
Slice Registers	24,707	65,200	37.89%
DSPs	7	120	5.83%

TABLE II
DISTRIBUTED AND BLOCK RAM UTILIZATION

Resource	Used	Available	Utilization (%)
LUT as Memory	742	9,600	7.73%
Block RAM Tile	9	75	12.00%
RAMB36E1	7	75	9.33%
RAMB18E1	4	150	2.67%

IX. DESIGN EVALUATION (ALEX AND HAANI)

The final system build unfortunately did not include the BMW heap.

Our way of evaluating the prime counter was to read out the data stream from the FPGA, visually frame the data if needed, and then compare the expected output to the actual output.

X. REFLECTIONS AND IMPLEMENTATION INSIGHTS

1) *Alex*: While our final integrated build did not include a fully operational BMW heap, the attempt to implementing it in hardware provided valuable lessons about scaling, modularity, and correctness in a deeply pipelined design. Our approach treated the BMW as a “shell” module with a nested RPU responsible for core push/pop behavior. This seemed like a reasonable path toward eventual integration into the broader system, but in practice the RPU quickly became large and



Fig. 7. Execution completion waveform.

difficult to reason about as additional corner cases and control paths were added.

We also recognized that more systematic hazard handling could have been incorporated around pop behavior and memory request timing. The combination of push and pop pipelines, along with generators used to parameterize variable dimensions, added complexity that was manageable in isolation but became challenging under tight development deadlines. The recursive structure of a BMW-style design amplifies this difficulty: each module must both handle its own pipeline arithmetic and cleanly interface with a smaller instance of the same logic, which makes it easy for small control assumptions to propagate into larger correctness issues.

This experience reinforced the importance of aggressive sub-modularization and early verification. We believed the RPU was complete more than once, only to discover failing cases that traced back to subtle internal timing and state assumptions. As hazard-handling temporary variables accumulated, we were reminded how critical it is to reason clearly about the scope and sequencing of nonblocking assignments in SystemVerilog, especially when building multi-stage control logic that must remain precise across cycles.

Overall, these challenges were constructive. They clarified which aspects of the BMW heap are primarily architectural, which are implementation hazards introduced by real memory latencies, and which demand more disciplined decomposition. Although we did not fully realize the BMW in the final system and were commented out, the development process provided a concrete roadmap for a cleaner, more verifiable implementation that better matches the modular intent of the broader project.

2) *Haani*: The project's initial scope was a bit undefined. Past making a scheduler, we didn't know what to make a scheduler *for*. I personally wasn't familiar with task scheduling at all so I didn't know what to focus on. I think that made it really hard for us to start off development at a good pace, and for the staff to provide actionable feedback. Deciding on prime counting was pretty good because it brings execution over many clock cycles into play. Unfortunately, it took a bit to get there since it's not something that immediately came to mind.

Developing the project mostly independently from the hardware was a really good decision. It just gave a great sense of confidence in the parts that were up and running, and made it super simple to pin down the problems when building on hardware. It would have been a massive headache to work on both the memory unit and the other modules when neither of them worked effectively.

Interestingly enough, the UART communication turned out

to not be too much of a problem at all. I was expecting a lot more data to be dropped and/or misframed from both ends (especially with a BAUD rate of 2,000,000), but that didn't happen. Using the memory controller in a different manner than the lab 6 setup was also something I expected to be really hard to get right, but that wasn't too bad either. It helped a lot that we weren't switching between DRAM reads and writes apart from going the setup to running. That's probably good for memory throughput as well or something.

I'm happy with the code organization overall. However, I did feel like a lot of the modules could have been consolidated and/or made combinational to save a clock cycle of basically no work. Still, I think it was good for the sake of matching up the conceptual understanding of the workflow with its implementation.

ACKNOWLEDGEMENTS

We would like to thank Professor Joe Steinmeyer and the 6.205 staff for all the help with the class and the project.

REFERENCES

- [1] 6.205 Team 62, A. Shin, H. Mahmood. [Online]. Available: <https://github.mit.edu/6205F25/fa25-6205-team62>
- [2] R. Yao, Z. Zhang, G. Fang, P. Gao, S. Liu, Y. Fan, Y. Xu, and H. J. Chao, "BMW Tree: Large-scale, High-throughput and Modular PIFO Implementation using Balanced Multi-Way Sorting Tree," in *Proceedings of the ACM SIGCOMM 2023 Conference*, New York, NY, USA: ACM, 2023, pp. 208–219, doi: 10.1145/3603269.3604862.
- [3] "Wikipedia: Sieve of Eratosthenes," Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
- [4] UberDDR3, Angelo Jacobo. [Online]. Available: <https://github.com/AngeloJacob0/UberDDR3>
- [5] 6.S965 Lab 8, 6.S965 Staff. [Online]. Available: https://rfsoc.mit.edu/6S965/_static/F25/assignments/week08/rfsoc_03/starter_code.zip