Modeling DRAM Discharge

1st Gary Mejia
Baskin School of Engineering
University of California
Santa Cruz, USA
gmejiama@ucsc.edu

2nd Mitchell Tansey
Baskin School of Engineering
University of California
Santa Cruz, USA
metansey@ucsc.edu

***Abstract*—Spiking neural networks have gained popularity in recent years. Their power efficiency over traditional deep learning models is from their inherit co location of memory and computation. The leaky-integrate fire model is the simplest model of a spiking neuron. This consists of a resistor and a capacitor in parallel to each other. We are attempting to model the leaky-integrate fire neuron on DRAM cells. By abstracting away the DRAM cells as resistors and capacitors in connection with a switch, we can use DRAM memories as large spiking neural networks. Using FPGAs and DRAM, we are attempting to bypass memory manufacturers' refresh mechanisms to disable refresh to allow the correct memory behavior seen in the leaky-integrate fire model. We iterated our model on several FPGAs including the Samsung SmartSSD platform with a Xilinx Kintex UltraScale+, Alveo U250, and ULX3S with Lattice ECP5. Our current design uses the Digilent Genesys2 board with Kintex 7 FPGA; the two designs consists of an open source PicoRV32 CPU to communicate with DRAM and the Rocket Chip CPU with bootable Linux. With DRAM refresh turned off, we found that the Linux bootloader fails after 21 seconds on average after bitstream programming.**

***Index Terms*—spiking neural networks, DRAM, security, field-programmable gate arrays**

I. Introduction

Spiking neural networks (SNNs) are artificial neural networks inspired by the brain. Due to their inherit combination of memory and computation elements, SNNs are more power efficient than traditional deep learning models [1]. An additional benefit of this combination is the simplicity of modeling. Leaky-integrate fire neurons are the simplest model for spiking neurons. Using a resistor and capacitor, we model both the computation and memory of a spiking neuron.

Dynamic random access memory (DRAM), is a common form of memory used in most modern computing devices. Each DRAM cell consists of only a transistor an a capacitor. Common delay models for transistors abstract them away as resistors attached with switches. As a result, our goal to use a DRAM cell to model a spiking neuron using the leaky-integrate fire model. This requires turning off the DRAM module's auto-refresh capabilities. We use FPGAs to modify DRAM module's controllers to disable refresh and also run memory intensive tasks to check data corruption. This time for data corruption is inline with the leaky-integrate fire neurons ability to spike.

A. Motivation

Computing tasks on a deep learning model is expensive through compute and memory costs. While the power efficiency of SNNs is a boon, the possibility to run an SNN on a dedicated DRAM module would save a lot of computation. Additionally this project would likely discover additional hardware security vulnerabilities with current DRAM designs. Refresh is a command that is carefully monitored by all DRAM manufactures for security purposes. Disabling entirely would require disabling logic in the DRAM IC itself or a work around commands such as RowHammer attacks [13].

II. Background

There are two main fields to have background knowledge for this project, SNNs and DRAM architecture. However, this paper will focus entirely on the FPGA design to disable refresh in DRAM.

A. Dynamic Random Access Memory

There are several forms of memory used in computer systems. DRAM along with SRAM (static random access memory) fall under volatile memories, where they lose their data upon power loss. Figure 1 shows the differences between the two at the transistor level.

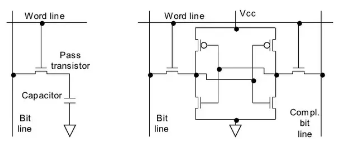


Fig. 1. Left: DRAM Cell Right: SRAM Cell

SRAMs are typically constructed with six total transistors, four to create two inverters in a feedback loop, and two access transistors. The feedback in the two inverters is what holds the value. When a one is stored in the cell the positive bit line will always output a one, the opposite is true for a zero. These are called static since the value stored never changes value until a write command is issued.

In contrast, DRAM cells are typically made with one transistor attached to a capacitor. Where in the SRAM cell, we had inverters in a feedback loop to remember our value, the capacitor's sole purpose is to remember a voltage level. Since capacitors leak voltage over type, the cell must be periodically recharged. This is the reason why DRAMs are dynamic memory. While recharging cells increase power consumption in DRAM, they are smaller and hence denser memories compared to SRAMs.

B. Elmore Delay Model

A common delay model in VLSI is the Elmore Delay model for transistors. This model abstracts transistors into resistors,capacitors, and switches [2]. Delay is often found by taking the summation of all RC time constants along a connected path. Since transistors have inherit resistance and capacitance, we believe we can use a DRAM module as a large spiking neural network.

C. Synchronous DRAM

Most DRAM modules in modern computers are synchronous DRAM (SDRAM) [12]. These modules are unique in that they sample commands at the positive edge of the clock. Sampling at the positive edge helps to synchronize the data flow with other components throughout the system. The DRAM module in the Samsung SmartSSD FPGA platform is a DDR4 SDRAM module. This paper will use DRAM to refer to both SDRAM and regular DRAM memories.

D. Double Data Rate

Double Data Rate (DDR) DRAM are memory modules that output data on both the positive and negative edge of the clock. These devices are among the few that utilize the negative edge of the clock in hardware. Memory bandwidth is an important metric, such that memory manufacturers are taking extreme methods to utilize both edges of the clock to output as much data into and out of memories. However, this comes at a complexity cost in the memory controller. The initialization of a DDR memory module requires precise analog timing, which the controller must respect. In addition, most synchronous logic designs are only synchronized to the positive edge of the clock, adding additional complexity to neighboring modules.

III. Initial FPGA Design

Many FPGA boards include some form of DRAM. Our main design uses the Samsung SmartSSD. This board ships with a Kintex UltraScale+ FPGA for custom logic designs and 4GB of DDR4 SDRAM. The main draw of this board is that the memory interface is not locked down, meaning it is not already synthesized for us. As a result, we can modify characteristics of the DRAM controller, such as enabling user-refresh commands and which interfaces it uses. Development is streamlined using Xilinx's Vitis flow.

Fig. 2. Samsung SmartSSD FPGA Development Board

A. Vitis Flow

Traditional FPGA toolflows consist of three main steps: synthesis, place and route, and bitstream generation. These flows are due to the compilation of RTL to netlists to place on the logic fabric. Vitis differs in this regard as it compiles any design to a kernel. Kernels are then placed on top of an already existing FPGA platform. Platforms are responsible for handling IO interconnectivity from the host to kernel design.

The traditional FPGA flow is also different. While one still does the traditional synthises, place and route, and bitstream generation, one must first compile and link the kernel. This is due to Vitis being a high-level synthesis tool for compiling C++ to Verilog. In our design we do not use C++, instead we directly use RTL. Compiling a design as a kernel requires several TCL scripts. These scripts are responsible for mapping memory addresses and saving the design into Xilinx's local IP repository.

Most FPGA designs include source RTL for actual hardware synthesis and different RTL for simulation and verification. For verification, we simply use Vitis's hardware emulation tool to generate a waveform of our design after linking. This requires us to target the linker for hardware emulation, as opposed to hardware for hardware synthesis. Our FPGA design consists of two main components: the main FPGA platform and the FPGA kernel design.

B. FPGA Kernel

The FPGA kernel is the design that goes on top of the provided platform. Our kernel consists of two main sets of components, the softcore CPU and interconnects. Figure 3 showcases all components connected together.

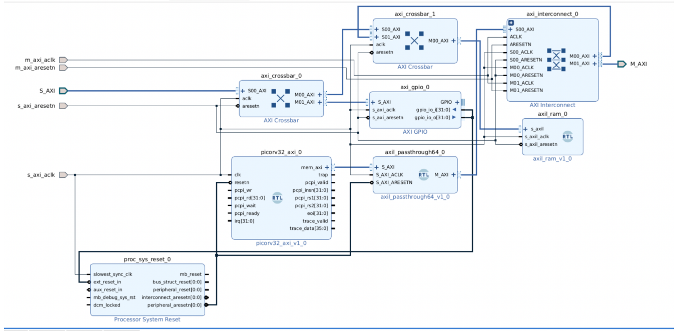


Fig. 3. FPGA Kernel Design

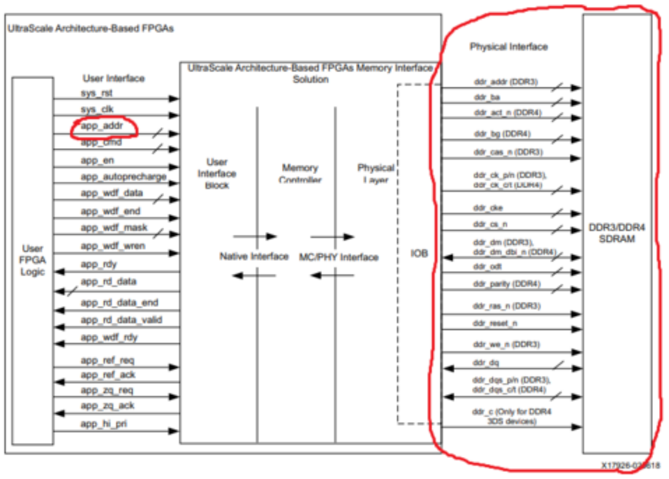


Fig. 5. UltraScale+ Memory Mappings from MC to Physical Device

1) Softcore - PicorRV32: The PicoRV32 is an open source single cycle CPU designed by YosysHQ, it supports the RISCV32IMC instruction set. Due to the open source nature of the CPU, there are extensive projects and documentation for using it. Among its parameters, the PicoRV32 supports multiple memory interfaces including both AXI Lite and Wishbone. Since most of Xilinx's IP uses AXI and AXI Lite, the PicoRV32 with AXI Lite is used.

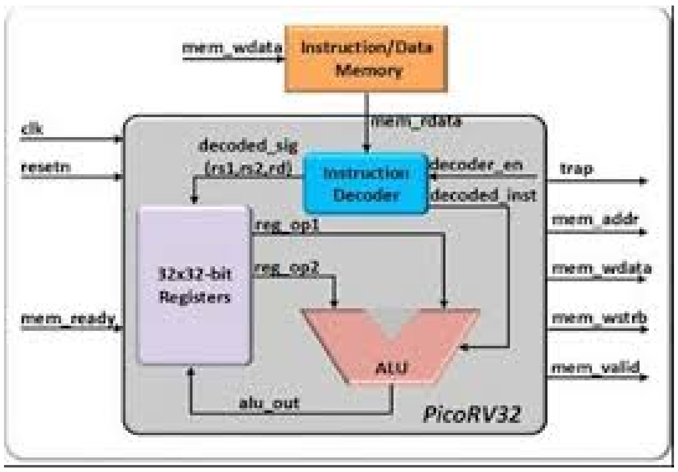


Fig. 4. Internal Architecture of PicoRV32

For the C program, we run a basic heap sort as it requires additional memory to store the heap data structure. To compile we use the RISC32-GCC cross compiler and linker. Since the target architecture is unknown, the linker requires an additional linker script to denote what memory regions are available to the CPU. We set the read only "code" region to be from addresses 0x0 to 0x10000. Main memory is both readable and writable from addresses 0xF0000000 to 0xFFFFFFFF. This allows our memory to be split into two regions by using the MSB of the address.

An additional constraint of our program is where it should read. Reads in DRAM are destructive, meaning they will refresh the cell and other local cells. As a result, we need to know where the C program's addresses are mapped to on the DRAM module itself. UltraScale+ FPGA's use a special address bus, "app_addr", to index into DRAM as shown in Figure 5.

To check corruption, the PicoRV32 includes an interrupt timer. When a read into DRAM is triggered, we can enter the interrupt handler. Once in the interrupt handler, we wait a certain amount of time and check the data read to an expected. If a mismatch is found, we can either increment a counter to make the PicoRV32 enter a trap.

2) Interconnects: Interconnects and cross bars are blocks designed to multiplex or de-multiplex signals. We have several interconnects in this design. Instruction memory is a simple AXI Lite BRAM that is written to prior to the CPU being ran. This memory has two users, the IO pins that write to it and the CPU itself. To achieve this behavior, we use a 2 to 1 interconnect. Similarly, the CPU can write and read from two different memories, instruction memory, or DRAM. A 1 to 2 interconnect solves this problem.

3) Additional Utilities: Outside of interconnects, the only utilities we need are processor reset IP and width adapters. The CPU only has a 32-bit address space while DRAM is a 64-bit address space. To adapt to this space, we wrote a pass-through module that simply appends 32 zeros to the address and data busses of the CPU's AXI Lite bus. For reset, we use Xilinx's Processor System Reset IP to make the system reset signal sequential. This allows the CPU to be reset before the interconnects are, preventing garbage data from being incorrectly written.

C. FPGA Platform

Despite this project being written in mostly block diagram and RTL, it is compiled down into a Vitis kernel. This kernel is then placed onto the pre-existing platform, the normal FPGA flow runs after this step. Figure 6 shows our FPGA kernel placed into the platform and synthesized.

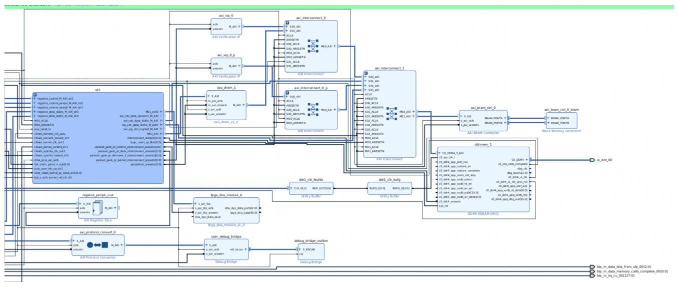


Fig. 6. Platform with Kernel Placed

This is the final and placed design on the FPGA. However, during kernel compilation and FPGA synthesis, we are able to modify characteristics of the design. Using a TCL script, we are able to disable the DRAM's controller's ability to issue ECC commands by shortening its address bitwidth from 72 to 64. Additionally, we are able to support user's abilities to issue refresh commands. We believe this disables the DRAM's own auto-refresh mechanisms and relies on the user to issue them. As a result, in the TCL script we add in a constant zero block and connect it to the DRAM's user refresh port.

1) PYNQ: PYNQ is a Python library aimed at interacting with Vitis kernels. We use PYNQ to interact with the board once it is flashed. PYNQ allows us to inject the board with inputs. Our main use of PYNQ is to write the compiled executable in instruction memory and run the CPU. An additional benefit is that PNYQ allows us to declare IO busses for DRAM to see data being transferred back and forth. Kernels can be targeting hardware or emulated hardware which PYNQ can distinguish.

IV. Functioning Design

After finding the SmartSSD approach to be too difficult, we shifted our design to the Diligent Genesys 2 board [8].

A. Genesys 2

The Genesys 2 is a high-performance FPGA development board from Digilent. It features a Xilinx Kintex-7 XC7K325T FPGA, which is part of the same FPGA family as the UltraScale+ but optimized for cost-efficiency and prototyping. This board includes 1GB of DDR3 memory [9], USB-JTAG programming, and extensive peripheral connectivity such as HDMI, Ethernet, and Pmod ports.

One of the most important features of the Genesys 2 for our purposes is the exposed DDR3 memory interface. Unlike some development boards with hardcoded memory controllers or vendor-specific firmware, the Genesys 2 allows users to implement and configure the memory interface using Xilinx's Memory Interface Generator (MIG). This makes it possible to adjust advanced memory parameters, such as refresh behavior, which is central to our goal of disabling auto-refresh for DRAM-based spiking neuron modeling.

Development on the Genesys 2 is supported by the Vivado Design Suite and does not require Vitis or a custom kernel platform. This simplified toolflow proved beneficial after running into the integration issues present in the SmartSSD platform. With full control over the hardware design and memory interface, the Genesys 2 allowed us to validate key aspects of our system, including memory access timing, refresh disablement, and CPU-controlled access to DRAM cells.

B. Vivado RISC-V Project

To test our DRAM corruption experiments, we utilized the open-source repository *vivado-risc-v* [3] by Eugene Tarassov. This project demonstrates how to integrate the Rocket Chip softcore processor into various Xilinx FPGA platforms, including the Genesys 2 board used in our setup. In this design, a bootable Linux image is stored on an SD card and loaded into memory at startup. The Rocket Chip then begins execution directly from the image on the FPGA. We confirmed successful DRAM corruption when the system crashed during Linux boot due to invalid virtual memory accesses, indicating that critical data had been altered in memory.

C. Disabling DRAM Refresh

Disabling refresh on the Genesys 2 board proved to be significantly more straightforward than on any other platform we tested. The process involves two key steps: enabling user-controlled refresh and synthesizing the design in a global context to propagate the changes.

1) Enabling User Refresh: As mentioned previously, the Genesys 2 board uses Vivado's in-house Memory Interface Generator (MIG) to interface with DDR memory. While MIGs typically provide a GUI option for configuring refresh behavior, this particular version did not expose any visible settings for user refresh in the block design editor.

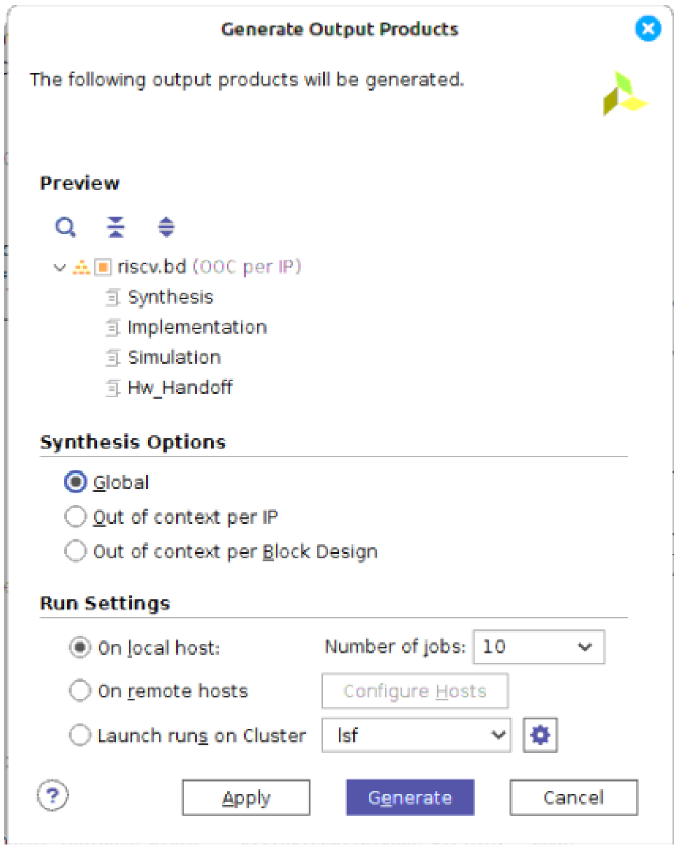
However, by directly inspecting the RTL generated by the MIG, we discovered the hidden ‘USER_REFRESH‘ parameter that controls this behavior. Although undocumented in the GUI, this parameter is present and functional in the MIG's RTL source. The relevant file is:

rtl/controller/mig_7series_v4_2_mc.v

By setting ‘USER_REFRESH‘ to ON within this file, we were able to enable user-controlled refresh, effectively allowing us to turn refresh off by never sending a user controlled refresh.

2) Global Context Synthesis: After modifying the MIG RTL, Vivado must be configured to re-synthesize the full design with these changes. This requires synthesizing the block design in a global context, ensuring that edits to generated sources are preserved and included in the implementation flow.

This can be done by selecting ”Generate Block Design” and then choosing ”Global” under ”Synthesis Options,” as shown in Figure 9. Without this step, Vivado may overwrite or ignore custom RTL changes during regeneration.



Generate Output Products

The following output products will be generated.

Preview

riscv.bd (OOC per IP)
- Synthesis
- Implementation
- Simulation
- Hw_Handoff

Synthesis Options

[✓ Global]
Out of context per IP
Out of context per Block Design

Run Settings

[✓ On local host:] Number of jobs: 10
On remote hosts — Configure Hosts
Launch runs on Cluster — lsf

Fig. 7. Selecting Global Context for Block Design Synthesis

D. Results

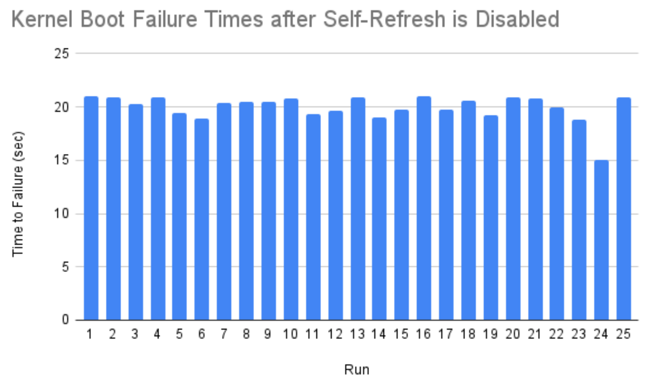


Fig. 8. Linux Bootloader Failure Times

We have collected a small batch of results to show the spread of times that linux took to crash. To collect these, we ran the vivado-risc-v project and used the linux boot timer to track when the first invalid virtual address read occurred. We see that on average, it takes about 20 seconds for the first error to throw. Below is a table with the most common virtual addresses incorrectly accessed during boot.

Virtual Address	Count
0xfffffffffffffc4	12
0x80	3
0x800000018	3

TABLE I
COMMON INVALUD VIRTUAL ADDRESSES ACCESSED

Most errors stem from the bootloader reading an invalid instruction. Once the bootloader found the illegal instruction, it would throw an error and stop. However, this is common on invalid accesses to low or high virtual addresses. Invalid accesses to middle virtual addresses often errored out due to shared objects unable to be found. This caused the kernel to drop an emergency shell after about 80 seconds. Our design is hosted publicly on a forked version of [3] on Github [7].

V. OTHER APPROACHES

Before we were able to get refresh successfully turned off, there were many different approaches that we tested.

A. *Custom SmartSSD Platform*

Since we are trying to modify the platform in between a Vitis flow, we believed that modifications should be made prior. This led us to develop a custom platform and place our kernel on top. We have tried two main methods.

The first method is modifying the provided ”hw.xsa” file that comes with the SmartSSD's platform package. This file holds all files requires for the hardware to synthesize such as: RTL, block diagrams, simulations, example designs, and information for the GUI. These files are just propitiatory zip files that can be opened with any archive manager. We grepped and changed any file relating to the Xilinx Memory Interface Generator (MIG) responsible for generating the memory controller for the DRAM memory. Additionally, we found the block diagram responsible for representing the actual DRAM memory on-board and changed its characteristics to turn off ECC. Upon re zipping the new files, we found we could run Vitis as is with no errors outside of a warning stating our ”hw.xsa” file can be unzipped and modified. However this still showed a memory controller and memory with ECC turned on.

Our second attempted involved us working with Gregor Haas, Ph.D student at University of Washington. As a part of his work with Vitis platforms, he has been modifying platforms to better work with on-board hardcores such as Arm Cortex and Esp32's. We thought his work could translate here and turn off refresh in the SmartSSD. He had successfully modified the same parameters we did with extracting the ”hw.xsa” file. However, our issues here stemmed from the constraint file being modified. Whereas our old custom platform did not do anything, here we could not place our clock or the kernel could not find it.

B. *ULX3S*

The ULX3S [10]is an open-source FPGA development board equipment with on-board SDRAM. This SDRAM module is non-DDR, which means that the controller will be less complicated than one for DDR [11]. However, the open-source nature of the board means that there is little vendor-supported IP. Fortunately, at least for interconnects, there are open-source projects that offer substitutes for AMD IP. To use the on-board SDRAM module, we need to create the memory controller from scratch. While the complexity is smaller than a DDR controller, it is not a trivial task. We attempted to create the SDRAM controller using Chisel.

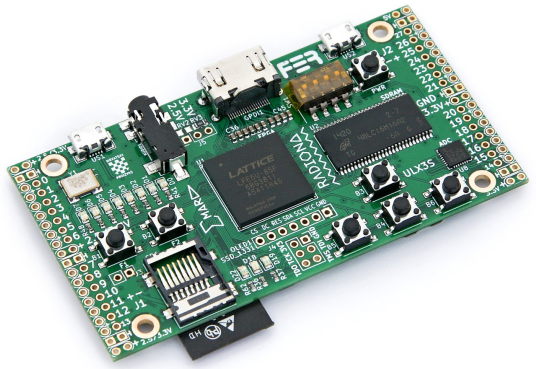


Fig. 9. ULX3S ECP5 FPGA Development Board

1) Chisel: Chisel is Scala embedded DSL aimed at increasing productivity in the creation of hardware generators. Whereas traditional hardware description languages such as Verilog and VHDL emit netlists, Chisel emits Verilog. This allows our controller to be manufacturer agnostic. A previous project of ours aims to generate memory for non-DDR SDRAM memory modules.

2) SDRAM Memory Controller Generator: The SDRAM Memory Controller Generator project [4] is a Chisel hardware generator that takes in JSON files as input and outputs SystemVerilog. JSON files describe the timing characteristics desired by the SDRAM module. Among these characteristics are strings to indicate the manufacturer and module number targeted.

Not only does the emitted SystemVerilog file include the synthesizable behavior of the controller, but also formal verification checks. These formal checks are primarily for the purpose of verifying the correct state machine transition.

While the formal checks passed, upon placing the design on the ULX3S we did not see anything as well. The issues here are we can not tell the polarity of the SDRAM module itself. Thus, we cannot verify wether we have correctly implemented the controller.

VI. Future Work

So far we only have metrics for the Rocket Chip failing to boot Linux. Our PicoRV32 design requires more work due to the JTAG to AXI IP required to write to the PicoRV32's instruction memory. We host this design publicly on Github as well [5].

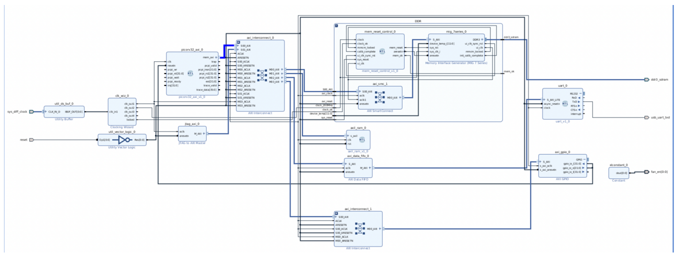


Fig. 10. PicoRV32 Design with JTAG to AXI Controller

Figure 10 shows our most up to date PicoRV32 design. The JTAG to AXI controller is placed to send commands to CPU, BRAM, and DRAM if needed. It is the second master of the main interconnect block that connects all 3 main components together.

A. C Programs for PicoRV32

Given that compared to the Rocket Chip design, the PicoRV32 will have no OS and thus run programs on baremetal. We have started a small repository of C programs targeting the PicoRV32 [5]. Among these is a simple bubble sort, that sorts 64 integers from least to greatest. We can test DRAM cell corruption by cross-checking the final array with one in ROM.

B. Debugging JTAG to AXI

Currently, our TCL script is a small modification of Xilinx's example script. This script just writes the first two instructions of a bubble sort into BRAM and reads it out. However, errors in the script do not specify any information to aid in debugging. To continue down this route, one must get more verbose error messages.

C. Using JTAG for AXI Transactions

Once the JTAG to AXI controller is fixed, we can start issuing the appropriate writes to BRAM. First, the PicoRV32 must be held in the reset state. During reset, the TCL script can then issue all the writes needed to fully write the entire program into BRAM. Since the transactions are AXI, a max of 256 bytes can be sent per transaction. Once the program is written in BRAM, the script can take the CPU out of reset and let it run. A question for the future is: how can the JTAG controller be used to probe DRAM during CPU runtime?

D. Pinpointing Failures in DRAM

The Rocket Chip design already shows that without refresh capabilities, the kernel will read from the wrong address. Since these are virtual addresses, they are indirect pointers to physical addresses in DRAM. These physical addresses would be what the FPGA sends out to the MIG and the memory module itself. Future work may include tracing where these virtual addresses map to in physical memory and then using the memory controller's addressing scheme to find what bank is failing. Finding where the flip occurs gives insight into using corrupting DRAM modules as SNNs.

VII. Conclusion

DRAM cells are likely candidates for modeling spiking neurons. Given the simplicity of the leaky-integrate fire model for spiking neurons, DRAM cells have the same physical characteristics as spiking neurons. DRAM memories can be suitable targets for creating spiking neural networks. Our work so far has shown that it likely takes around 20 seconds for DRAM memory to corrupt in a room temperature environment.

We have demonstrated that by disabling DRAM refresh and observing Linux bootloader behavior on the Genesys 2 FPGA, we can quantify DRAM decay through real-world system crashes. This method provides a practical, observable metric for degradation in memory charge retention, giving insight into both neuromorphic computing possibilities and areas in which DRAM efficiency can increase.

Future work on improving the PicoRV32 JTAG-based flow and enabling runtime DRAM monitoring will help us move closer to a fully controllable spiking DRAM framework. Ultimately, this work bridges the gap between conventional memory systems and brain-inspired models in a novel and hardware-efficient way.

Acknowledgment

We acknowledge Assistant Professors Jason Eshraghian and Dustin Richmond for supervising this project. Not only do they assist with ideas and debugging, but they also provided us with resources. They have been kind enough to provide us with a dedicated server for the SmartSSD and access to all materials used in this project. In addition to them, we like to acknowledge Ph.D students Tyler Sheaves and Gregor Haas for their guidance and expertise on topics involved.

References

[1] Ganguly C, Bezugam SS, Abs E, Payvand M, Dey S, Suri M. Spike frequency adaptation: bridging neural models and neuromorphic applications. Commun Eng. 2024 Feb 1;3:22. doi: 10.1038/s44172-024-00165-9. PMCID: PMC11053160.

[2] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. Boston: Addison Wesley, 2011.

[3] eugene-tarassov, “GitHub - eugene-tarassov/vivado-risc-v: Xilinx Vivado block designs for FPGA RISC-V SoC running Debian Linux distro,” GitHub, Mar. 26, 2024. https://github.com/eugene-tarassov/vivado-risc-v (accessed May 12, 2025).

[4] Gary Mejia, “GitHub - gmejiamtz/sdram_controller_generator: A Chisel hardware generator for SDRAM controllers from their datasheets,” GitHub, 2024. https://github.com/gmejiamtz/sdram_controller_generator (accessed Dec. 14, 2024).

[5] Gary Mejia, “Github - gmejiamtz/picorv32i_programs: Some example programs to compile for picorv32,” GitHub, 2024. https://github.com/gmejiamtz/picorv32i_programs (accessed Jun. 8, 2025).

[6] Gary Mejia, “Github - gmejiamtz/genesys2-dram-discharge-picorv32: A Vivado 2019.2 Project for DRAM Discharge using the PicoRV32 CPU” GitHub, 2025. https://github.com/gmejiamtz/genesys2-dram-discharge-picorv32 (accessed Jun. 9, 2025).

[7] Gary Mejia, “Github - gmejiamtz/vivado-risc-v-dram-discharge: Xilinx Vivado block designs for FPGA RISC-V SoC running Debian Linux distro with DRAM refresh disabled” GitHub, 2025. https://github.com/gmejiamtz/vivado-risc-v-dram-discharge (accessed Jun. 9, 2025).

[8] Digilent, “Genesys 2 Reference Manual,” Genesys 2 Reference Manual, 2023. https://digilent.com/reference/programmable-logic/genesys-2/reference-manual (accessed Jun. 09, 2025).

[9] Micron, ”DDR3 SDRAM Reduced tFAW Addendum”, March 14, 2013

[10] Crowd Supply, “Ulx3s,” ULX3S, https://www.crowdsupply.com/radiona/ulx3s (accessed Jun. 10, 2025).

[11] Winbond, ”W9825G6KH”, Dec 17, 2021

[12] JEDEC Standard - DDR4 SDRAM, https://xdevs.com/doc/Standards/DDR4/JESD79-4%20DDR4%20SDRAM.pdf (accessed Jun. 10, 2025).

[13] O. Mutlu, “Fundamentally Understanding and Solving RowHammer,” arXiv, Jan. 2023.