

Designing Silicon Brains using LLM: Leveraging ChatGPT for Automated Description of a Spiking Neuron Array

Mike Tomlinson
mtomlin5@jh.edu
Johns Hopkins University
Baltimore, MD, USA

Joe Li
qli67@jh.edu
Johns Hopkins University
Baltimore, MD, USA

Andreas Andreou
andreou@jh.edu
Johns Hopkins University
Baltimore, MD, USA

ABSTRACT

Large language models (LLMs) have made headlines for synthesizing correct-sounding responses to a variety of prompts, including code generation. In this paper, we present the prompts used to guide ChatGPT4 to produce a synthesizable and functional verilog description for the entirety of a programmable Spiking Neuron Array ASIC. This design flow showcases the current state of using ChatGPT4 for natural language driven hardware design. The AI-generated design was verified in simulation using handcrafted testbenches and has been submitted for fabrication in Skywater 130nm through Tiny Tapeout 5 using an open-source EDA flow.

ACM Reference Format:

Mike Tomlinson, Joe Li, and Andreas Andreou. 2024. Designing Silicon Brains using LLM: Leveraging ChatGPT for Automated Description of a Spiking Neuron Array. In *Proceedings of (ARXIV)*. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

Over the last three decades, advances in CMOS technology and CAD tools have led to advances in processor technology that in turn fed research into design and automation tools that enable the sophisticated System On Chip for general computing and AI. Verilog and VHDL, both released in the 1980s, have become standard synthesis tools in digital design. These tools allow the writer to describe behavioral functionality that can be directly mapped to digital standard cells and physical layout generation through place and route. Verilog and VHDL are the schematic entry point into modern CAD tools. Writing and maintaining code in Verilog and VHDL introduces significant overhead as these abstract design at a rather low level. There are a number of projects trying to address this with a range of adoption and commercial support. These projects include efforts such as Chisel[1] and High Level Synthesis (with specific tools from Cadence, Vivado, and Synopsys). Overall, the general trend in these methods is to move towards a higher level language that can then be used to generate VHDL or Verilog.

In November of 2023, OpenAI’s ChatGPT, a LLM captured the attention of users and business alike because it offered a simple

but powerful interface to LLMs for performing generative AI tasks. This interactive interface to LLMs is capable of executing a variety of tasks such as writing prose and generating code. This model has shown to be effective at generating python, albeit with problems of attention span and adaptability [5], [9].

Recent works addressing LLM assisted hardware design include a LLM based optimization framework that integrates ChatGPT with existing EDA tools. In the work by [4], authors employ LLM tools to implement several simple modules. For each implementation, power, performance, and area are compared to modules implemented with ChatGPT alone, Xilinx HLS, and Chisel. Other research explores a different set of simple blocks, including a simple microprocessor with a ChatGPT-defined ISA [2]. Similarly, Yang et al. investigate ChatGPT’s effectiveness for systolic arrays and ML accelerators [11].

1.1 Contributions

In this paper we explore the use of generative AI and ChatGPT (version 4 is used in this work) [7] to design a hardware system, namely a spiking neural network chip, a neuromorphic electronic system [6], for hardware AI inference [8]. Our effort differs from the current research in the use of LLMs for CAD by focusing on digital spiking neurons [3], an unconventional computing architecture, and by emphasizing complete system design. We document the steps taken to go from a conversational design description to a functional and synthesizable Verilog description of a programmable array of spiking neurons. The final AI-generated HDL has a standardized interface, SPI, and multiple levels of hierarchy. This work represents one of the first ASICs synthesized entirely from natural conversational language. By documenting this process, we hope to showcase the current state of using LLMs as a higher level, conversational, alternative to handcrafted HDL.

2 NATURAL LANGUAGE HARDWARE DESCRIPTION

This work targets a neuromorphic design with a model trained on a large amount of public data. Only a small fraction of this data is likely to be related to neuromorphic engineering and an even smaller fraction to neuromorphic Verilog. Table 1 gives an idea of the volume of training code available for these topics. The table lists relevant keywords and the corresponding number of matching publicly available repositories on Github. There are hundreds of spiking neuron related repositories, but around 2 orders magnitude fewer dealing specifically with Verilog.

We start by prompting ChatGPT to generate a leaky integrate and fire (LIF) neuron. This module is then instantiated in a network module, where 2 layers of 3 neurons are instantiated in a fully

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ARXIV, Jan 25, 2024.

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Table 1: Number of Search Results on Github

Search Term	Language	Number of Results
"integrate and fire"	Verilog	3
"integrate and fire"	Any	750
"snn"	Verilog	26
"snn"	Any	2.4k
"spiking neuron"	Verilog	6
"spiking neuron"	Any	478
"spiking"	Verilog	29
"spiking"	Any	8.7k
"alu"	Verilog	1.1k
"accelerator"	Verilog	339
"cnn"	Verilog	198
"spi"	Verilog	530

connected fashion. We then prompt for an SPI peripheral that will be used to program the network parameters. Finally, we ask ChatGPT to synthesize a top-level module instantiating both the SPI and network modules. All transcripts and source code can be found on Github : https://github.com/AndreouLab/tinytapeout_05_chatgpt_snn

2.1 The Neuron

The LIF neuron model is one of the more popular spiking neuron models and one of the simplest. The basic LIF neuron [3] has one state variable, the membrane potential, V_m , and one output, S , a boolean variable indicating if a spike is generated or not. In discrete time, these variables can be modeled by the following equations:

$$V_m[t] = \begin{cases} \max(I_{in}[t] + V_m[t-1] - I_{leak}, 0), & S[t] = 0 \\ 0, & S[t] = 1 \end{cases}$$

$$S[t] = V_m[t] > V_{th}[t]$$

Additional functionality can be added by including a refractory filter. After a spike occurs, $S[t]$ transitions from 0 to 1; $V_m[t]$ will remain at 0 for a fixed period of time. This can be thought of as an additional state variable.

This discrete time approximation of the LIF neuron model naturally lends itself to digital circuitry. The membrane potential can be implemented with an accumulator, the refractory filter, a counter, and the spiking output with a comparator. With this in mind, we prompted ChatGPT4 with a high level request for a Verilog module for a spiking LIF neuron:

Prompt: *Can you write a verilog module for a spiking leaky integrate and fire neuron? Please include a multibit current input and a refractory period.*

ChatGPT: *Sure, I'll provide a basic Verilog module for a spiking leaky integrate-and-fire neuron. Here's a simple version:*
(Shown in Table 2)

After a cursory look at the code, it may seem very impressive! But,

after careful consideration and simulation, a number of problems can be found.

- (1) **Line 5:** syntax error, spike is declared as a wire (by default), it needs to explicitly declared as a register
- (2) **Lines 27-38:** logical error, update logic does not account for overflow
- (3) **Lines 34:** logical error, faulty logic for checking for an underflow (line 34)
- (4) **Lines 18, 45:** structural error, spike is multiply driven (used in two processes)

These problems were all fixable through further prompting. Issues 1 and 4 each required only one additional prompt to correct. The logical errors, 2 and 3, were considerably more difficult, requiring several iterations to correctly handle overflow, underflow, and working with unsigned numbers.

Once the code was functional and synthesizable, we prompted ChatGPT to make the neuron parameters (threshold, leak rate, and refractory period) programmable. ChatGPT was able to do this without error, deleting the parameter **declarations and adding** input ports to drive the parameter values. We also **increased the bit width of the refractory period from 6 to 8 bits through prompting**. The complete Verilog for this module is shown in Table 2.

ChatGPT had little trouble understanding the basic concept of the LIF neuron and making specific modifications when given a detailed prompt, but struggled to provide complete modules without syntax errors and showed conceptual misunderstanding when working with limited precision unsigned numbers. For an experienced digital designer these problems are relatively easy to fix, but for a user without the same background, these problems may pose considerable difficulty. Requiring extensive code modifications heavily limits the ability for ChatGPT to act as a full natural language abstraction.

2.2 Network

For the network, we wanted to implement something relatively simple. We decided on a 2 layer network with 3 neurons per layer and fully programmable weights. A diagram of the envisioned network is shown in Fig. 1.

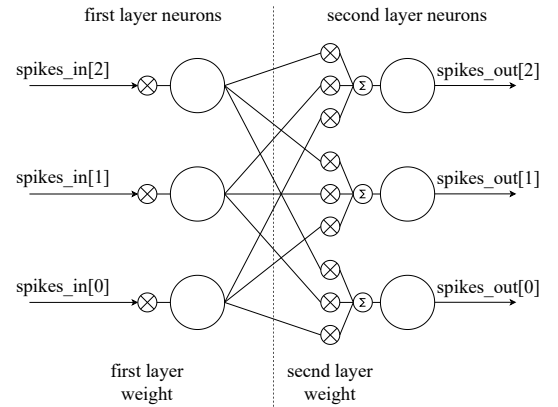
**Figure 1: Two layer fully connected neuron network.**

Table 2: Initial and Complete Generated LIF Neuron Verilog

Initial Module	Complete Module
<pre> 1 module leaky_integrate_fire_neuron (2 input clk, // clock input 3 input reset, // asynchronous reset 4 input [7:0] current, // 8-bit current input 5 output spike // output spike signal 6); 7 // Parameters 8 parameter THRESHOLD = 8'd255; // spiking threshold 9 parameter LEAK_RATE = 8'd1; // decrease by 1 per cycle 10 parameter REFRAC_PERIOD = 32; // refractory period in cycles 11 12 // Internal states 13 reg [7:0] membrane_potential; // current value of the membrane potential 14 reg [5:0] refrac_counter = 6'd0; // refractory period counter 15 reg in_refrac = 0; // refractory period flag 16 17 // On every clock cycle 18 always @(posedge clk or posedge reset) begin 19 if (reset) begin 20 membrane_potential <= 8'd0; 21 refrac_counter <= 6'd0; 22 in_refrac <= 0; 23 end else begin 24 if (in_refrac) begin 25 refrac_counter <= refrac_counter - 1'b1; 26 if (refrac_counter == 6'd0) in_refrac <= 0; 27 end else begin 28 // update membrane potential based on current input and leak rate 29 if (membrane_potential + current > THRESHOLD) begin 30 membrane_potential <= 8'd0; 31 spike <= 1; 32 in_refrac <= 1; 33 refrac_counter <= REFRAC_PERIOD; 34 end else if (membrane_potential + current - LEAK_RATE < 8'd0) begin 35 membrane_potential <= 8'd0; 36 end else begin 37 membrane_potential <= membrane_potential + current - LEAK_RATE; 38 end 39 end 40 end 41 end 42 43 // Reset spike signal after one cycle 44 always @(posedge clk or posedge reset) begin 45 if (reset) spike <= 0; 46 else if (spike) spike <= 0; 47 end 48 endmodule </pre>	<pre> 1 module leaky_integrate_fire_neuron (2 input clk, // clock input 3 input reset, // asynchronous reset 4 input [7:0] current, // 8-bit current input 5 input [7:0] THRESHOLD, // spiking threshold 6 input [7:0] LEAK_RATE, // decrease by 1 per cycle 7 input [7:0] REFRAC_PERIOD, // refractory period in cycles 8 output reg spike // output spike signal 9); 10 // Internal states 11 reg [7:0] membrane_potential = 8'd0; 12 reg [7:0] refrac_counter = 8'd0; 13 reg in_refrac = 0; 14 15 // On every clock cycle 16 always @(posedge clk or posedge reset) begin 17 if (reset) begin 18 membrane_potential <= 8'd0; 19 refrac_counter <= 8'd0; 20 in_refrac <= 0; 21 spike <= 0; 22 end else begin 23 spike <= 0; // By default, reset the spike signal 24 if (in_refrac) begin 25 refrac_counter <= refrac_counter - 1'b1; 26 if (refrac_counter == 8'd0) in_refrac <= 0; 27 end else begin 28 // Check for potential underflow 29 if (membrane_potential < LEAK_RATE) begin 30 membrane_potential <= current; // Directly set to current 31 end else if (membrane_potential + current < membrane_potential) begin 32 // Potential overflow detected, clamp to THRESHOLD 33 membrane_potential <= THRESHOLD; 34 end else begin 35 membrane_potential <= membrane_potential + current - LEAK_RATE; 36 end 37 38 // Check for spiking 39 if (membrane_potential >= THRESHOLD) begin 40 membrane_potential <= 8'd0; 41 spike <= 1; 42 in_refrac <= 1; 43 refrac_counter <= REFRAC_PERIOD; 44 end 45 end 46 end 47 end 48 endmodule </pre>

Each circle represents one LIF neuron. For a given neuron, the conversion from spikes to current can be defined as the sum of incoming spikes weighted by the strength of the synaptic connection. Mathematically, the input current for each neuron is given by:

$$I_{in}[t] = \sum_{i=1}^N w_i x_i[t]$$

Where I_{in} is the neuron input current, x_i is the incoming spike from neuron i in the previous layer, w_i is the programmed weight for the i th input to this neuron, and N is the total number of synaptic connections to the previous layer. To describe this module in

Verilog, an experienced designer might consider using two generate statements, one per layer, and some combinational logic to calculate the input currents. We opened a new chat, provided a new copy of the neuron module and then gave the following prompt:

Prompt: Now that we have a working neuron, can you create a new module that instantiates a network of neurons with 2 layers in a fully connected fashion? Please connect them with programmable synapses.

The generated code is included in Table 3. For this module, there are a number of apparent issues.

- (1) **Line 7:** syntax error, parameter N is used but never declared
- (2) **Lines 7, 9:** syntax error, ports, input_currents and weights, are declared as 2d arrays, which is invalid in Verilog (although valid in systemVerilog)
- (3) **Line 8:** syntax error, similar to the previous issue, spikes is declared as an unpacked array, which is not allowed for ports in verilog (although valid in systemVerilog)
- (4) **Lines 13-21:** syntax error, the instantiation of the neuron is not valid syntax, no generate variable is used, nor are variables used to index into the current and spike connections
- (5) **Lines 40-48:** syntax error, issue 4 also applies to the second set of instantiations
- (6) **Lines 25-37:** syntax error, spike variable access is syntactically incorrect.

Issues 1 and 3 were relatively easy to fix, each only requiring one additional prompt. Issue 2 was more difficult. Eventually, the issue was resolved by suggesting that ChatGPT flatten the 2D ports into 1D arrays. This required the prompter to be familiar with Verilog in order to provide the solution, limiting the use of this process to already experienced engineers.

After flattening the array, ChatGPT created the following code to reassign the flattened array to an internal 2D variable. This code has a fundamental problem. An initial block is used when continuous assignment is desired. This error causes weights to be initialized, but never updated after initialization, resulting in a module that cannot be programmed. Prompting ChatGPT to fix this issue caused it to then declare spikes (Line 8) as type reg, introducing another syntax error (spikes is driven through a port connection). In fixing one issue, two others were introduced.

```

1 reg [7:0] weights[2:0][2:0];
2 integer i, j;
3 initial begin
4     for (i = 0; i < 3; i = i + 1) begin
5         for (j = 0; j < 3; j = j + 1) begin
6             weights[i][j] = weights_flat[(i*3 + j)*8 +: 8];
7         end
8     end
9 end
10 ;
```

Issue 4 was also difficult to resolve, requiring 7 additional prompts. We requested ChatGPT explicitly declare each neuron separately and then concatenate those instantiations back together using a generate statement. While working on this part of the code, ChatGPT again introduced syntax errors in other parts of the code. At one point, replacing *begin* in the code snippet above with an open bracket while keeping the corresponding end statement, mixing C style coding and Verilog. Similar to the neuron module, these issues required knowledge of Verilog to identify, debug, and resolve through prompting, reducing the level of abstraction.

Once the errors were corrected, we used ChatGPT to make design revisions to the generated code. For this module we added a register file to store the neuron parameters (shared for all neurons), and the network weights. We first gave the following prompt:

Prompt: *Ok, can you please provide a programmable register file that*

stores all the parameters to our network

This prompt produced a reasonable looking register file, that stored the neuron parameters for each neuron (threshold, leak rate, and refractory period). The produced code was syntactically correct, but not exactly what we were asking for. This highlights both an impressive and problematic aspect of this technology. The previous prompt was ambiguous. It is not clear what parameters we wanted stored; and rather than asking for clarification, ChatGPT assumes we want to store independent parameters for each neuron, instead of storing the weights and a single set of neuron parameters to be shared. This was an easy clarification to make. The next prompt resulted in an almost correct module with our desired functionality and only a minor syntax error, confusing systemVerilog and Verilog coding styles.

Prompt: *Can you include weights as a parameter and let's say both layers use the same parameters*

Again, this prompt was somewhat vague. From the structure of the sentence, we can infer that the intent is to store independent values for each weight and a single common copy of the neuron parameters. ChatGPT was able to make the same inference. Once this module was functionally correct, we prompted ChatGPT to merge this functionality into the network module. This was done without error.

2.3 Serial Peripheral Interface (SPI)

We then prompted ChatGPT to create an SPI peripheral module, allowing us to program the network parameters and weights over a standard interface.

Prompt: *OK, I want you to now create a SPI interface to communicate with the network module above.*

The generated code is presented in Table 3, some comments were removed to reduce the size of the code. This module also has a number of subtle errors.

- (1) **Line 13:** syntax error, use of the SystemVerilog enum feature which is not supported in Verilog
- (2) **Lines 27-52:** logical errors in SPI finite state machine
- (3) **Line 48, 62:** syntax error, write_enable is multiply driven

Errors 1 and 3 are similar to errors seen previously. The first is another mixup with SystemVerilog syntax and the third is a multiply driven net. Both of these issues were resolved with a single follow-up prompt addressing each. These consistent mistakes highlight ChatGPT's lack of familiarity with Verilog.

Error 2 also showcases a lack of experience with Verilog and with SPI. This code section confuses one segment and two segment coding styles. Lines 19-25 are typical for a two segment style, but inconsistent with lines 27-52 where *next_state* is assigned sequentially, as if *next_state* is the state variable in a one segment process. The state logic also has no dependence on *sclk*, even though it created a port for this signal. *sclk* is intended to be the SPI clock, which synchronizes the controller and peripheral. This signal is crucial to writing a correct SPI state machine.

Table 3: Initial Generated Network and SPI modules

Network Module	SPI Module
<pre> 1 module two_layer_neural_network (2 input clk, // clock input 3 input reset, // asynchronous reset 4 input [7:0] THRESHOLD, // spiking threshold 5 input [7:0] LEAK_RATE, // decrease by 1 per cycle 6 input [5:0] REFRAC_PERIOD, // refractory period in 7 // cycles 8 input [7:0] input_currents[N], // Input currents for N 9 // neurons in the first layer 10 output spikes[M], // Spike outputs for M 11 // neurons in the second layer 12 input [7:0] weights[N][M] // Programmable synapse 13 weights 14); 15 16 // First layer of neurons 17 leaky_integrate_fire_neuron first_layer[N](18 .clk(clk), 19 .reset(reset), 20 .THRESHOLD(THRESHOLD), 21 .LEAK_RATE(LEAK_RATE), 22 .REFRAC_PERIOD(REFRAC_PERIOD), 23 .current(input_currents), 24 .spike() // We'll compute this shortly 25); 26 27 // Logic to compute effective current for second layer 28 // neurons based on spikes and weights 29 reg [7:0] second_layer_currents[M]; 30 genvar i, j; 31 generate 32 for (i = 0; i < M; i = i + 1) begin : 33 compute_second_layer_current 34 always @(posedge clk) begin 35 second_layer_currents[i] = 0; 36 for (j = 0; j < N; j = j + 1) begin 37 if (first_layer[j].spike) begin 38 second_layer_currents[i] = 39 second_layer_currents[i] + weights[j][i]; 40 end 41 end 42 end 43 end 44 endgenerate 45 46 // Second layer of neurons 47 leaky_integrate_fire_neuron second_layer[M](48 .clk(clk), 49 .reset(reset), 50 .THRESHOLD(THRESHOLD), 51 .LEAK_RATE(LEAK_RATE), 52 .REFRAC_PERIOD(REFRAC_PERIOD), 53 .current(second_layer_currents), 54 .spike(spikes) 55); 56 endmodule </pre>	<pre> 1 module spi_slave (2 input clk, // System Clock 3 input reset, // System Reset 4 input sclk, // SPI Clock 5 input cs_n, // SPI Chip Select (Active Low) 6 input mosi, // SPI Master Out Slave In 7 output reg miso, // SPI Master In Slave Out 8 output reg [3:0] addr_out, 9 output reg [7:0] data_out, 10 output reg write_enable 11); 12 13 // SPI state definitions 14 typedef enum {IDLE, TRANSFER, COMPLETE} spi_state; 15 reg [1:0] state, next_state; 16 reg [15:0] shift_reg; 17 reg [7:0] addr_reg; 18 integer bit_count; 19 // SPI state machine 20 always @(posedge clk or posedge reset) begin 21 if (reset) begin 22 state <= IDLE; 23 end else begin 24 state <= next_state; 25 end 26 end 27 28 // SPI logic 29 always @(posedge clk) begin 30 case (state) 31 IDLE: begin 32 if (!cs_n) begin 33 bit_count <= 15; // Reset bit count 34 next_state = TRANSFER; 35 end 36 end 37 TRANSFER: begin 38 shift_reg[bit_count] <= mosi; 39 if (bit_count == 0) begin 40 next_state = COMPLETE; 41 end else begin 42 bit_count <= bit_count - 1; 43 end 44 end 45 COMPLETE: begin 46 if (cs_n) begin 47 next_state = IDLE; 48 addr_reg <= shift_reg[15:8]; 49 data_out <= shift_reg[7:0]; 50 write_enable <= 1; 51 end 52 end 53 endcase 54 end 55 56 // MISO signal assignment 57 always @(posedge clk) begin 58 if (state == TRANSFER) begin 59 miso <= shift_reg[15]; 60 end 61 end 62 63 always @(*) begin 64 addr_out = addr_reg; 65 if (state != COMPLETE) begin 66 write_enable = 0; 67 end 68 end 69 endmodule </pre>

Additionally, the state machine doesn't come out of reset. When reset is applied, state is driven to IDLE and next state is undriven. When reset is deasserted, the current state will go to whatever *next_state* was at the time of applying reset. These issues were all eventually resolved through subsequent prompts.

2.4 Top Module

The top module needs to instantiate both the SPI module and the network module, appropriately connecting internal signals and declaring input and output ports. In the same chat, the following prompt was used:

Prompt: *Can you create a top file to connect this spi module with the network module?*

The generated code had only one error. The intermediate signals between the modules were declared as reg instead of wire. This was an easy fix. After one more prompt, the top module was complete.

3 TINY TAPEOUT: IMPLEMENTATION

This design was submitted to TinyTapeout 5, a multi-project die effort for Skywater 130nm through efabless. The flow is intended to handle most of the intricacies of digital implementation, exposing only a minimal set of configuration options in a user level script. This flow uses yosys [10] for synthesis, which introduced an interesting complication. Through an iterative debugging process we determined the following code produced a multiply driven net error.

```

1 always @(posedge clk or posedge reset) begin
2     if (reset) begin
3         ...
4         for (idx1 = 0; idx1 < 3; idx1 = idx1 + 1) begin
5             FIRST_LAYER_WEIGHTS[idx1] <= 8'd0;
6             ...
7         end
8     end else if (write_enable) begin
9         ...
10    end
11 end
12
13 always @(*) begin
14     for (idx1 = 0; idx1 < 3; idx1 = idx1 + 1) begin
15         input_currents[idx1] = spikes_in[idx1] ?
16             FIRST_LAYER_WEIGHTS[idx1] : 8'd0;
17     end
18 end
19 Error: : Yosys checks have failed: Encountered check error:
20 Warning: Drivers conflicting with a constant 1'0 driver:
21 port Q[2] of cell $procdff$614 ($dff)

```

After extensive debugging, we found that this error could be resolved by removing the duplicate loop variables in the code above. Once identified, simply asking ChatGPT to create new loop variables resolved the issue. Final simulations using the gate level netlist were conducted through the TinyTapeout flow. The design occupies 33% of 320um x 200um and is expected to be fabricated by Summer 2024.

4 CONCLUSION

This paper explores the use of ChatGPT to convert from natural language to functionally correct and synthesizable Verilog. We

successfully use natural language entry to generate a complete HDL description of a programmable spiking neuron array, ready for implementation. It is clear technologies like ChatGPT have the potential to increase design efficiency, correctly producing simple modules, quickly generating foundational code from scratch and offering near instantaneous, accurate modifications of existing code when prompted detailed instructions. However, the current quality of ChatGPT's output often falls short. ChatGPT's responses tend to frequently include some form of error, either syntactically or logically. ChatGPT also confidently demonstrates ignorance of more advanced concepts, leading to potentially obfuscated bugs, increasing the difficulty of verification. These problems compound, placing a significant burden on the prompter. **If the prompter knows the solution, they can guide ChatGPT to the answer; but without that knowledge, it can be difficult to use this technology** as a tool for abstracting Verilog description. Overall, our findings suggest that natural language to Verilog synthesis has potential; but in its current form, it leaves much to be desired.

ACKNOWLEDGMENTS

This work was supported by NSF Grant 2020624 AccelNet:Accelerating Research on Neuromorphic Perception, Action, and Cognition through the Telluride Workshop on Neuromorphic Cognition Engineering, NSF Grant 2332166 RCN-SC: Research Coordination Network for Neuromorphic Integrated Circuits and NSF Grant 2223725 EFRI BRAID: Using Proto-Object Based Saliency Inspired By Cortical Local Circuits to Limit the Hypothesis Space for Deep Learning Models.

REFERENCES

- [1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzyniec, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, San Francisco California, 1216–1225. <https://doi.org/10.1145/2228360.2228584>
- [2] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. <http://arxiv.org/abs/2305.13243> arXiv:2305.13243 [cs].
- [3] Andrew S. Cassidy, Julius Georgiou, and Andreas G. Andreou. 2013. Design of Silicon Brains in the {nano-CMOS Era}: Spiking Neurons, Learning Synapses and Neural Architecture Optimization. *Neural Networks* 45 (June 2013), 4–26. <https://doi.org/10.1016/j.neunet.2013.05.011>
- [4] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2023. ChipGPT: How far are we from natural language hardware design. <http://arxiv.org/abs/2305.14019> arXiv:2305.14019 [cs].
- [5] Ali Kashefi and Tapan Mukerji. 2023. ChatGPT FOR PROGRAMMING NUMERICAL METHODS. *Journal of Machine Learning for Modeling and Computing* 4, 2 (2023), 1–74. <https://doi.org/10.1615/JMachLearnModelComput.2023048492>
- [6] Carver A. Mead. 1990. Neuromorphic Electronic Systems. *Proc. IEEE* 78, 10 (1990), 1629–1636. <https://doi.org/10.1109/5.58356>
- [7] OpenAI. 2023. GPT-4 Technical Report.
- [8] Kayode A. Sanni and Andreas G. Andreou. 2019. A Historical Perspective on Hardware AI Inference, Charge-Based Computational Circuits and an 8bit Charge-Based Multiply-Add Core in {16nm FinFET CMOS}. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 3 (Sept. 2019), 532–543. <https://doi.org/10.1109/JETCAS.2019.2933795>
- [9] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant – How far is it? <http://arxiv.org/abs/2304.11938> arXiv:2304.11938 [cs].
- [10] Clifford Wolf and Johann Glaser. 2013. Yosys - A Free Verilog Synthesis Suite. *Proceedings of the 21st Austrian Workshop on Microelectronics(Austrochip)* (Oct. 2013).
- [11] Kaiyuan Yang, Haotian Liu, Yuqin Zhao, and Tiantai Deng. 2023. A new design approach of hardware implementation through natural language entry. *IET Collaborative Intelligent Manufacturing* 5, 4 (2023), e12087. <https://doi.org/10.1049/>

cim2.12087 _eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1049/cim2.12087>.