I still remember my third year of undergrad as an electrical engineering major — the time we first dipped our toes into digital circuit design using hardware description languages. The HDLs themselves weren’t too bad once you got the hang of them, but getting those circuits to actually run on an FPGA? That was straight-up chaos. We were using Altera FPGAs with the Quartus IDE on Linux machines, and just getting the software and drivers to play nice felt like solving a puzzle designed by someone who hates you.

Somehow, we stumbled our way through that course, and I was more than happy to leave FPGAs behind as I dove into semiconductor device research for my PhD at MIT. But as fate would have it (plot twist!), I recently found myself circling back into the FPGA vortex for a new project. All I had were distant, mildly traumatic memories of the ordeal — no actual memory of *how* to make anything work.

So here we are again. I had to relearn everything from scratch, piecing it together step by step and hoping I’m not just reinventing a more chaotic wheel. This blog is my way of documenting those steps — implementing a fairly simple digital design on an FPGA — so Future Me (and maybe Present You, curious reader) can skip the unnecessary suffering next time around.

Oh, and by the way: the Mersenne Twister (MT) pseudo-random number generator is just the guinea pig for this project. If you're here for a deep dive into PRNG metrics, this probably isn’t your ride. But if you're looking to get a basic design up and running on an FPGA without losing your mind — welcome aboard.

I’ll start with a quick primer on the Mersenne Twister — just enough to give you a feel for how the FSM ticks, so the Verilog code that follows actually makes some sense. Once that’s out of the way, we’ll dive into the simulation workflow: from basic behavioral simulation to the post-implementation stage. After that, I’ll walk through how to get solid, high-confidence energy estimates using Vivado’s Power Report tool, with the help of a SAIF (Switching Activity Interchange Format) file.

At this point, the FPGA will be happily generating MT19937 pseudo-random numbers internally. But unless you’re planning to sit and stare at LED blinks for entropy, we’ll need to slap on a UART transmitter to stream the numbers to a PC for verification. With that UART wrapper in place, we’ll be all set to flash the code and watch a stream of Mersenne magic roll out of the FPGA.

**The Mersenne Twister, FSMs, and Me**

Before diving into simulations and power estimates, let’s start with the core of this beast — the **Mersenne Twister** (specifically, the MT19937 variant) and how I built it up as a finite state machine (FSM) in Verilog.

**A Quick Tour of the MT19937**

The Mersenne Twister (MT) is one of the most widely used pseudo-random number generators out there, and MT19937 is its most popular variant. The “19937” refers to the fact that the algorithm has a ginormous period of 219937−12^{19937} - 1219937−1, which is a Mersenne prime — hence the name.

Here’s the flow of the MT19937 algorithm as implemented in hardware:

1. **Seeding**: A 624-word internal state is initialized using a recurrence formula, starting from a single external seed.
2. **Twisting**: New state values are generated in chunks using XORs, shifts, and a conditional matrix multiply with a constant matrix\_a.
3. **Tempering**: The raw output from the twist is passed through a series of XOR-and-shift operations to improve statistical quality. This step *is* implemented in my hardware design — more on that below.

**The FSM That Runs the Show**

To coordinate all this logic, I designed a finite state machine (FSM) that cycles through four states:

* **S0 – IDLE**: Waiting for the ext\_seed\_enable signal. This is the FPGA equivalent of twiddling thumbs.
* **S1 – SEED**: Seeds the internal 624-entry state array using a recurrence. Outputs done\_seed = 1 when finished.
* **S2 – TWIST**: Executes the twisting transformation. At each cycle, it:
  + Grabs three words from the state array,
  + Computes a twisted value via masked bit operations and XORs,
  + Applies **tempering** to the result,
  + Produces a final random\_number and raises valid\_rn.
* **S3 – TRANSIT**: A guard band between SEED and TWIST to ensure memory operations complete cleanly.

The transitions between states are driven by the done\_seed and done\_twist flags. Once initialized, the FSM loops between twisting and reseeding, generating new outputs on every cycle.

**Module Breakdown: Who Does What**

Here’s how the Verilog modules are sliced up:

* **mt\_fsm.v**: The central controller. It handles all state transitions and control signals for the rest of the design. Think of it as the stage manager cueing actors.
* **mt\_seed.v**: Responsible for seeding the MT state array. It iterates through all 624 indices, calculating and writing new values using the initialization recurrence. When done, it asserts done\_seed.
* **mt\_twist.v**: This one handles the magic. For each cycle:
  + It calculates the new twisted value using neighboring entries of the state array.
  + Performs the **tempering** step using constants from the MT19937 spec:

verilog

CopyEdit

y = y ^ (y >> U);

y = y ^ ((y << S) & B);

y = y ^ ((y << T) & C);

y = y ^ (y >> L);

* + Outputs a valid 32-bit pseudo-random number and writes the updated state back.
* **mt\_state\_mem.v**: A dual-ported memory array with 624 words of 32-bit state. Both the seed and twist modules read/write to it based on the FSM’s control.

Together, these modules recreate the MT19937 algorithm step by step, entirely in hardware. It’s overkill if you just want to generate random numbers — but gold if you care about deterministic, cycle-accurate, energy-measurable randomness for hardware benchmarking (which I do).

**Running the Behavioral simulation**

As the name suggests, behavioral simulation is about verifying whether your digital design *behaves* as intended — before you go anywhere near actual hardware. For the Mersenne Twister implemented here, the goal of simulation is to check whether the FSM cycles correctly between seeding the state memory and performing the twist/temper operation, and whether the output random\_number stream looks valid — ideally, uniform over the 32-bit range.

What makes this process powerful is the **waveform database**: it logs how every signal in your design changes over time. This becomes an essential tool for debugging, letting you visually trace issues like stuck states, incorrect transitions, or invalid outputs — long before you flash anything onto an FPGA.

To run a behavioral simulation, you need to write a **testbench** — a kind of wrapper module that mimics the real-world environment your design would operate in. The testbench doesn’t get synthesized or loaded onto the FPGA. Its only job is to stimulate your design with clock signals, resets, and inputs, and to observe outputs, just like a virtual lab setup.

In this project, the testbench file tb\_mt\_fsm.v instantiates the top-level module mt\_fsm. It drives the clk and rst lines, provides an external\_seed value, and toggles the external\_seed\_enable flag to kick off the FSM. It also captures the output signals random\_number and valid\_rn, and writes the random numbers to a text file so we can inspect them offline.

**Post-implementation simulation and Power Report generation**

By this point, the behavioral simulation has verified that the design works as expected. But before flashing it onto the FPGA, it’s wise to run a few more checks. We want to know whether the design will operate reliably at the target clock frequency, how much power it might consume, and whether it risks heating up during sustained operation. These questions are addressed through Vivado’s Synthesis and Implementation stages. Once Implementation completes, Vivado provides a Utilization Report that tells us how many FPGA resources — like LUTs, flip-flops, and DSPs — the design consumes. It also generates a Timing Report, where metrics like the Worst Negative Slack and Worst Hold Slack let us assess whether the design meets timing requirements.

Vivado also provides a Power Report at this stage, but the numbers in that report are based on internal heuristics about signal switching activity. In other words, it makes educated guesses about how often each signal toggles per second. This can be misleading, especially for designs like a pseudo-random number generator, where some signals might switch very frequently while others rarely do. To improve accuracy, we can generate a SAIF file — a Switching Activity Interchange Format file — that records actual signal transitions during a post-implementation functional simulation. This simulation uses the same design and testbench as before, but with the added instruction for Vivado to track how many times each signal flips. Once the simulation runs long enough — say, until one million random numbers have been generated — we end up with a detailed and realistic estimate of each signal’s activity. Feeding this SAIF file back into Vivado lets the tool refine its Power Report, giving you a much more trustworthy estimate of your design’s real-world power consumption.

**Preparing for FPGA deployment**

Having ascertained that the design operates as intended and gathered other vital information regarding timing, resources utilization and power consumption, we can prepare the design for flashing on the FPGA. In principle, the only addition we need to make to proceed is to the constraints file, where we designate the ports of the design (top-level module) to the FPGA’s I/O ports, and in effect to the development board’s peripherals (LEDs, switches, buttons, UART, etc.) However, for this example, I will make some further modifications to correctly use the differential clock on my board (Genesys2) and to facilitate streaming the random numbers on my PC using UART.

The Genesys2 board comes with a differential clock which operates at 200 MHz. Our Mersenne Twister, however, takes as input a single-ended clock. So, I created a new top-level module, mt\_uart.v, which takes as input the differential clock, derives a single-ended, clock, clk, out of it (also at 200 MHz), down-samples that clock to a lower frequency (seq\_clk) and feed this lower frequency clock to mt\_fsm. Down-sampling is simply done using a counter, while the differential clock is convert to single-ended using the following snippet

**UART Communication - Transmit from FPGA to PC**

With the clocks configured, digital design is in principle ready to happily generate 32-bit random numbers inside the FPGA — but we need a way to actually see those numbers on a computer. That’s where UART (Universal Asynchronous Receiver/Transmitter) comes in.

**A Quick Primer on UART**

UART is one of the simplest and most widely used serial communication protocols for talking to microcontrollers, FPGAs, or pretty much any embedded device. It's asynchronous, meaning there’s no clock line shared between sender and receiver — both ends just agree on a **baud rate** (bits per second) and stick to it. Each byte is sent as a packet that includes:

* **Start bit** (usually 0)
* **8 data bits** (least significant bit first)
* **Stop bit** (usually 1)

That makes each byte a 10-bit packet on the wire.

**My UART Transmitter Setup**

To send the MT19937-generated numbers to the PC, I wrote a uart\_tx.v module. It takes in 1 byte from a register and handles all the UART-level packing: adding the start and stop bits, and shifting the bits out serially on a tx line. It also has a busy flag so the controller knows when to wait before sending the next byte.

Then comes the top-level module, mt\_uart.v. This one connects the dots:

* It instantiates the mt\_fsm to generate 32-bit random numbers.
* It slices those 32 bits into four 8-bit chunks.
* It sends those chunks out one-by-one through uart\_tx, waiting for the busy signal to clear before loading the next byte.

So for every new random number generated, four UART packets are streamed out to the PC.

**Receiving the Data on the PC**

On the PC side, I wrote a short Python script (stream\_mt.py) using the pyserial library. It listens to the serial port, grabs the incoming bytes, assembles them back into 32-bit words, and writes them into a text file — one number per line.

This gives me a quick and easy way to verify that the MT output looks reasonable (or at least non-repeating!) and also lets me analyze the distribution offline.

Well, all our preparations are finally done! Just head over to Vivado’s Project Manager and run Synthesis, Implementation, and Generate Bitstream in sequence. Once the bitstream is ready, use the Hardware Manager to flash it onto the FPGA. You should see the LEDs spring to life — a visual confirmation that the Mersenne Twister is up and running on hardware.

Now, to start collecting those random numbers, open your terminal and run the stream\_mt.py Python script. That’s it. The whole setup is live, generating and streaming high-quality pseudo-random numbers straight from your FPGA to your PC.