Pico Miner

FPGA-Based Proof of Work Mining Accelerator

Proof of Concept Document

Pico Miner Project

February 18, 2026

Contents

1 Introduction

Cryptocurrency mining is the process by which new transactions are verified and added to the blockchain ledger. The core computational challenge is known as **Proof of Work (PoW)**: miners must find a value (called a *nonce*) such that the hash of the block data concatenated with the nonce produces a result that satisfies a certain difficulty condition — typically, the hash must be numerically less than a *target* value.

In Bitcoin, this involves computing the `SHA-256` hash function twice (double SHA-256) over an 80-byte block header while iterating over a 32-bit nonce space. The difficulty target is adjusted so that, on average, a valid nonce is found every 10 minutes across the entire network.

Pico Miner is a proof-of-concept project that demonstrates the fundamental principles of PoW mining accelerated on an FPGA. Rather than implementing the full SHA-256 algorithm (which is complex and resource-intensive), we use a **custom lightweight hash function** that preserves the essential properties needed for PoW while being practical for implementation in Vivado HLS on a Zynq-7020 FPGA.

2 Background: How Bitcoin Mining Works

2.1 The Blockchain

A blockchain is a distributed, append-only ledger where each block contains:

- A reference (hash) to the previous block
- A set of transactions
- A timestamp
- A *nonce* value

2.2 Proof of Work

The PoW mechanism requires that:

$$\text{Hash}(\text{block\_header} \,\|\, \text{nonce}) < \text{target} \tag{1}$$

where $\|$ denotes concatenation and the *target* is a threshold derived from the current difficulty. Since cryptographic hash functions behave as pseudo-random oracles, the only way to find a valid nonce is by **brute-force iteration** — trying nonce values one by one until the condition is met.

2.3 Why FPGAs?

FPGAs offer several advantages for PoW mining:

- **Parallelism**: Multiple hash computations can run simultaneously in hardware
- **Pipelining**: The hash computation can be deeply pipelined to achieve high throughput (one hash per clock cycle)

- **Energy efficiency**: Dedicated hardware is far more energy-efficient than general-purpose CPUs or GPUs for this repetitive computation
- **Low latency**: Direct hardware implementation avoids instruction fetch/decode overhead

3 Project Scope and Simplifications

This project is a **proof of concept** designed for educational purposes. The following table compares real Bitcoin mining with our simplified implementation:

Table 1: Comparison: Bitcoin Mining vs. Pico Miner

Aspect	**Bitcoin**	**Pico Miner**
Hash function	Double SHA-256 (512-bit blocks)	Custom 32-bit hash (PicoHash)
Block header	80 bytes (640 bits)	16 bytes (4 × 32-bit words)
Nonce size	32 bits	32 bits
Difficulty	256-bit target comparison	32-bit target comparison
Output	256-bit hash	32-bit hash
Network	Peer-to-peer blockchain	Standalone demonstration

3.1 What is Preserved

Despite the simplifications, our project preserves the **core algorithmic structure** of PoW mining:

1. **Brute-force nonce search**: The miner iterates through nonce values, just like a real miner
2. **Hash computation per nonce**: Each nonce attempt requires a full hash computation
3. **Difficulty-based acceptance**: The hash must be below a target to be "valid"
4. **Hardware acceleration**: The hash + compare loop runs in dedicated FPGA hardware, controlled by ARM software
5. **HLS optimization**: We demonstrate how pipeline and unroll directives dramatically improve throughput

4 PicoHash: Custom Hash Function

4.1 Design Rationale

Our custom hash function, **PicoHash**, is inspired by the DJB2 and FNV hash families. It is designed to be:

- **Simple**: Easily synthesizable in HLS with minimal resources

- **Deterministic**: Same input always produces the same output
- **Avalanche-prone**: Small input changes cause large output changes (desirable for PoW)
- **Non-invertible (practically)**: Difficult to reverse analytically, requiring brute force to find a valid nonce

4.2 Algorithm Definition

Algorithm 1 PicoHash($data[0..N-1]$, $nonce$)

$h \leftarrow$ `0x5A3C_F1E7` {Initial seed}
for $i = 0$ **to** $N - 1$ **do**
  $h \leftarrow h \oplus data[i]$ {XOR with data word}
  $h \leftarrow h \times$ `0x01000193` {FNV prime multiply}
  $h \leftarrow h \oplus (h \gg 16)$ {Mix upper bits into lower}
end for
$h \leftarrow h \oplus nonce$ {Incorporate nonce}
$h \leftarrow h \times$ `0x5BD1E995` {MurmurHash finalizer constant}
$h \leftarrow h \oplus (h \gg 13)$
$h \leftarrow h \times$ `0x5BD1E995`
$h \leftarrow h \oplus (h \gg 15)$
return h

The algorithm has two phases:

1. **Absorption**: Iterates over the block data words, mixing each into the hash state via XOR and multiplication
2. **Finalization**: After incorporating the nonce, applies a MurmurHash-inspired bit-mixing sequence to ensure good avalanche behavior

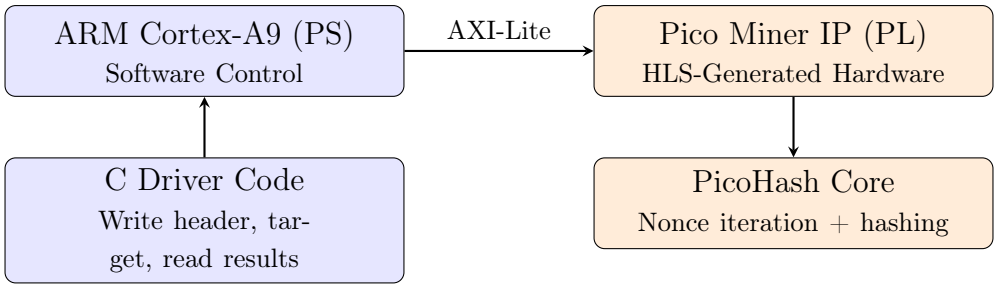
4.3 Properties

- **Output size**: 32 bits (unsigned integer)
- **Input**: Array of 32-bit words + 32-bit nonce
- **Latency**: $O(N)$ for N data words, plus constant finalization steps
- **Collision resistance**: Not cryptographically secure (32-bit output), but sufficient for educational PoW demonstration

5 System Architecture

5.1 Hardware/Software Partitioning

The system follows the Zynq PS/PL architecture:



5.2 HLS IP Interface

The HLS IP function signature is:

```
void pico_miner(
    int block_header[4],    // Input: simplified block header
    int difficulty_target,  // Input: hash must be < this value
    int nonce_start,        // Input: beginning of nonce range
    int nonce_end,          // Input: end of nonce range
    int *found_nonce,       // Output: the winning nonce (if found)
    int *found_hash,        // Output: the winning hash value
    int *status             // Output: 1 = found, 0 = not found
);
```

All ports are mapped to AXI-Lite (`s_axilite`) bundled into a single bus called `myaxi`, following the same methodology as the course examples.

5.3 Mining Operation Flow

1. The ARM processor writes the block header (4 words), difficulty target, and nonce range to the HLS IP via AXI-Lite registers
2. The ARM issues `ap_start` to begin computation
3. The HLS IP iterates through the nonce range:
   (a) Computes $h = \text{PicoHash}(\text{header}, \text{nonce})$
   (b) If $h <$ target: stores the nonce and hash, sets status to 1, and terminates early
4. The ARM polls `ap_done`, then reads the results
5. If no valid nonce was found in the range, status is 0

6 HLS Optimization Strategy

Following the course methodology, we explore multiple optimization levels:

6.1 Solution 1: Baseline

No directives applied. The nonce search loop executes sequentially with no pipelining or parallelism.

6.2 Solution 2: Loop Pipelining

```
#pragma HLS PIPELINE II=1
```

Applied to the main nonce search loop. This allows the next nonce iteration to begin before the previous one completes, achieving an **Initiation Interval (II) of 1** — one new hash per clock cycle after the pipeline fills.

6.3 Solution 3: Array Partitioning

```
#pragma HLS ARRAY_PARTITION variable=block_header complete
```

Combined with loop pipelining, this ensures all block header words can be read simultaneously, eliminating memory port bottlenecks.

6.4 Expected Performance Scaling

Table 2: Expected Performance Across Solutions (estimated)

Solution	Latency/Hash	II	Throughput
Baseline (no directives)	∼20 cycles	∼20	∼5 MH/s
Loop pipelining	∼20 cycles	∼1	∼100 MH/s
Pipeline + array partition	∼10 cycles	∼1	∼100 MH/s

(At 100 MHz clock, II=1 yields ∼100 million hashes/second.)

7 Verification Strategy

7.1 C Simulation (csim)

A software testbench (`pico_miner_tb.cpp`) contains:

- A `pico_miner_sw()` reference function implementing the same algorithm in pure C
- Test cases with known difficulty levels and expected results
- Bit-exact comparison between SW and HW outputs

7.2 C/RTL Co-Simulation (cosim)

After C synthesis, the generated RTL is verified against the C testbench using Vivado HLS co-simulation to confirm cycle-accurate correctness.

7.3 ARM Integration Test

The ARM driver code (`helloworld.c`) performs the same SW vs. HW comparison on the actual Zynq hardware, printing results via UART.

8 Conclusion

Pico Miner demonstrates that the fundamental principles of cryptocurrency Proof of Work mining can be effectively mapped to FPGA hardware using Vivado HLS. By using a simplified hash function, we preserve the core computational pattern — brute-force nonce search with hash comparison — while keeping the design practical for a Zynq-7020 target.

The project showcases:

- High-Level Synthesis of a mining algorithm from C to RTL
- AXI-Lite integration for PS/PL communication
- HLS directive optimization (pipeline, array partition)
- SW/HW verification methodology with golden model comparison

This approach could be extended to implement real SHA-256 hashing for actual Bitcoin mining, though that would require significantly more FPGA resources and design effort.