# EXPERIMENTAL PROTOCOL

## ASIC-DIFFUSION: Latent Space Generation Experiments

Hardware Platform: Lucky Miner LV06 (SHA-256)

| | |
|---|---|
| **Protocol ID:** | ASIC-DIFF-EXP-001 |
| **Version:** | 1.0 |
| **Duration:** | 4-6 weeks |
| **Objective:** | Validate ASIC-generated latent spaces for diffusion model initialization |

## EXPERIMENT 1: Hardware Communication Validation

### 1.1 Objective

Establish reliable bidirectional communication with the LV06 ASIC via USB, verify hashrate consistency, and confirm deterministic behavior (identical inputs produce identical outputs).

### 1.2 Materials Required

- Lucky Miner LV06 ASIC miner
- USB cable (high quality, data-capable)
- Host computer with Python 3.10+
- USB power meter (optional but recommended)
- cgminer or bfgminer (for initial testing)

### 1.3 Procedure

#### Step 1: Physical Setup

1. Connect LV06 to host computer via USB.
2. Verify device recognition: `lsusb | grep -i miner` or check `/dev/ttyUSB*`
3. Record USB port assignment for consistent testing.
4. Allow 60 seconds for device initialization and thermal stabilization.

#### Step 2: Initial Communication Test

5. Launch cgminer with test pool: `cgminer -o stratum+tcp://testpool:3333 -u test -p x --usb :all`
6. Verify hashrate display (expected: ~500 GH/s nominal).
7. Record actual hashrate, hardware errors, and rejection rate.
8. Run for 10 minutes minimum to establish baseline stability.

#### Step 3: Custom Stratum Client Implementation

Create Python script implementing minimal Stratum protocol:

```
# lv06_stratum_client.py
import socket, json, hashlib, struct, time

def create_work_unit(seed_nonce):
'''Construct block header with controlled nonce'''
    version = struct.pack('<I', 0x20000000)
    prev_hash = bytes(32)  # Zeroed for testing
    merkle = hashlib.sha256(seed_nonce.encode()).digest()
    timestamp = struct.pack('<I', int(time.time()))
```

```
    bits = struct.pack('<I', 0x1d00ffff)
    nonce = struct.pack('<I', 0)
    return version + prev_hash + merkle + timestamp + bits + nonce
```

### Step 4: Determinism Verification

9. Submit identical work unit 100 times.
10. Record all returned hashes.
11. Verify 100% hash identity across runs.
12. Document any deviations with timestamp and environmental conditions.

## 1.4 Success Criteria

| Metric | Required Value |
|---|---|
| Hashrate stability | ±5% of nominal over 1 hour |
| Determinism rate | 100% (identical input → identical output) |
| Hardware error rate | < 0.1% |
| Communication latency | < 100ms round-trip |

# EXPERIMENT 2: Statistical Characterization of Hash Output

## 2.1 Objective

Comprehensively characterize the statistical properties of LV06-generated hashes to determine suitability as pseudo-random latent space initializers. Specifically, verify uniform bit distribution, absence of autocorrelation, and compliance with NIST randomness standards.

## 2.2 Procedure

### Step 1: Large-Scale Hash Collection

13. Generate sequential work units with nonces 0 through 10,000,000.
14. Collect all resulting SHA-256 hashes (256 bits × 10M = 320 MB raw data).
15. Store in binary format for efficient analysis.
16. Estimated collection time: 20-30 seconds at 500 GH/s.

### Step 2: Bit Distribution Analysis

17. Calculate per-bit frequency across entire dataset.
18. Expected: each bit position shows ~50% ones, ~50% zeros.
19. Apply chi-squared test for uniformity ($p > 0.01$ required).
20. Generate visualization: 256-element bar chart of bit frequencies.

### Step 3: Autocorrelation Analysis

21. Compute autocorrelation of bit stream at lags 1, 2, 4, 8, 16, 32, 64, 128, 256.
22. Expected: autocorrelation $< 0.01$ at all lags (indicates independence).
23. Generate visualization: autocorrelation plot with 95% confidence bounds.

### Step 4: NIST SP 800-22 Test Suite

Apply the following NIST statistical tests:

- Frequency (Monobit) Test
- Frequency Test within a Block
- Runs Test
- Longest Run of Ones in a Block
- Binary Matrix Rank Test
- Discrete Fourier Transform (Spectral) Test
- Cumulative Sums Test

### Step 5: Avalanche Effect Quantification

24. Generate 10,000 hash pairs where inputs differ by exactly 1 bit.
25. Calculate Hamming distance between each hash pair.
26. Expected mean: 128 bits (50% of 256 bits).
27. Expected standard deviation: ~8 bits (binomial distribution).
28. Generate histogram of Hamming distances.

## 2.3 Success Criteria

| Test | Pass Criterion |
|---|---|
| Bit frequency deviation | < 0.1% from 50% |
| Autocorrelation (all lags) | < 0.01 |
| NIST tests passed | All 7 tests ($p > 0.01$) |
| Avalanche mean Hamming distance | 128 ± 2 bits |

# EXPERIMENT 3: Visual Texture Generation (Project C.H.R.O.M.A.)

## 3.1 Objective

Visualize LV06 hash outputs as images to determine whether SHA-256 computation introduces visible structure or texture. This experiment tests the hypothesis that the ASIC's physical computation produces aesthetically interesting patterns beyond pure noise.

## 3.2 Procedure

### Step 1: Canvas Generation (1024×1024)

29. Target image size: 1024 × 1024 pixels = 1,048,576 pixels.
30. Each pixel requires 24 bits (RGB) or 32 bits (RGBA).
31. Each hash provides 256 bits = 8 RGB pixels or 10.67 pixels (grayscale).
32. Required hashes: ~131,072 for RGB, ~98,304 for grayscale.

### Step 2: Hash-to-Pixel Mapping

Implement multiple mapping strategies:

```
# Strategy A: Direct byte mapping (grayscale)
def hash_to_grayscale(hash_bytes):
    return list(hash_bytes)  # 32 grayscale pixels per hash

# Strategy B: RGB triplets
def hash_to_rgb(hash_bytes):
    pixels = []
    for i in range(0, 30, 3):  # 10 RGB pixels per hash
        pixels.append((hash_bytes[i], hash_bytes[i+1], hash_bytes[i+2]))
    return pixels
```

### Step 3: Visual Analysis

33. Generate 10 images with sequential seed nonces (0-9).
34. Perform visual inspection for patterns, gradients, or structure.
35. Apply 2D Fourier transform to detect spectral patterns.
36. Compare to reference images generated by software PRNG (numpy.random).

### Step 4: Structured Input Experiments

Test whether structured inputs produce structured outputs:

- Feed grayscale gradient image (0-255 sweep) as input nonces.
- Feed checkerboard pattern as input.
- Feed natural image (photograph) pixel values as nonces.
- Document any visual correlation between input structure and output texture.

## 3.3 Expected Results

**Hypothesis A (Null):** Hash outputs appear as perfect white noise with no visible structure. This would confirm SHA-256's cryptographic properties and suitability as a noise source.

**Hypothesis B (Alternative):** Subtle visual patterns emerge, potentially due to ASIC hardware characteristics, power supply ripple, or thermal effects. This would indicate unique artistic potential.

# EXPERIMENT 4: Stable Diffusion Integration

## 4.1 Objective

Replace Stable Diffusion's default Gaussian noise initialization with LV06-generated latent tensors and evaluate image quality, generation consistency, and aesthetic properties.

## 4.2 Procedure

### Step 1: Latent Tensor Construction

Stable Diffusion v1.5 latent space dimensions: 64 × 64 × 4 = 16,384 values.

```
def build_sd_latent_from_asic(hashes):
'''Convert LV06 hashes to SD-compatible latent tensor'''
    import numpy as np
    raw_bytes = b''.join(hashes[:512])  # 512 hashes × 32 bytes
    uint8_array = np.frombuffer(raw_bytes, dtype=np.uint8)
    # Scale to [-1, 1] range (approximate Gaussian)
    float_array = (uint8_array.astype(np.float32) - 127.5) / 127.5
    return float_array[:16384].reshape(1, 4, 64, 64)
```

### Step 2: Comparative Generation

37. Select 20 diverse prompts spanning different artistic styles and subjects.
38. For each prompt, generate 5 images with standard Gaussian initialization.
39. For each prompt, generate 5 images with ASIC-derived latents (sequential seeds).
40. Total: 200 images (100 Gaussian, 100 ASIC).

### Step 3: Quality Assessment

- **FID Score:** Compare Fréchet Inception Distance between ASIC and Gaussian batches.
- **CLIP Score:** Measure prompt-image alignment for both methods.
- **Human Evaluation:** Blind A/B preference test with 10 evaluators.

### Step 4: Energy Measurement

41. Measure LV06 energy consumption during latent generation phase.
42. Compare to GPU energy for equivalent torch.randn() calls.
43. Calculate Joules per latent tensor for both methods.

## 4.3 Success Criteria

| Metric | Target |
|---|---|
| FID score difference | < 5% degradation vs Gaussian |
| CLIP score | Equivalent to Gaussian baseline |
| Human preference | No significant preference ($p > 0.05$) |
| Energy efficiency gain | > 90% reduction in latent generation phase |

# 5. Data Recording Requirements

All experiments must record:

- Timestamp (ISO 8601 format)
- Ambient temperature
- LV06 reported hashrate
- Power consumption (if meter available)
- Input parameters (seeds, nonces)
- Output hashes (hex format)

- Any anomalies or errors observed

## Author Information

**Name:** Francisco Angulo de Lafuente
**Date:** December 2024

**GitHub:** https://github.com/Agnuxo1
**ResearchGate:** https://www.researchgate.net/profile/Francisco-Angulo-Lafuente-3
**Kaggle:** https://www.kaggle.com/franciscoangulo
**HuggingFace:** https://huggingface.co/Agnuxo
**Wikipedia:** https://es.wikipedia.org/wiki/Francisco_Angulo_de_Lafuente