Phonon Modular Synthesis DSL Design Document

Executive Summary

This document outlines the design for Phonon's next-generation modular synthesis DSL - a text-based, live-codeable system that enables arbitrary signal routing and cross-modulation between any audio sources, synthesis parameters, and pattern data. Unlike traditional systems where synthesizers are black boxes (SuperCollider/TidalCycles), Phonon will allow any signal to modulate any parameter in real-time, bringing DAW-style modulation capabilities to live coding.

Core Philosophy

1. **Everything is a Signal**: Audio, control data, pattern events - all are signals that can modulate each other
2. **No Black Boxes**: Every signal path is visible and modifiable
3. **100% Textual**: No GUI required - everything expressible in code
4. **Live Codeable**: All connections and parameters hot-swappable during performance
5. **Performance First**: Leverage Rust and FunDSP for real-time audio processing

Language Design

1. Signal Bus System

Signal buses are named connections that carry audio or control signals throughout the system. They are defined with the ~ prefix:

```
~lfo1: sine(2)                  // 2 Hz sine wave LFO
~env1: perc(0.01, 0.5)         // Percussion envelope
~bass: saw(110) * ~env1        // Saw wave modulated by envelope
```

Buses can reference other buses, creating complex modulation networks:

```
~mod_depth: sine(0.1) * 0.5 + 0.5     // Slow LFO controlling modulation depth
~vibrato: sine(6) * ~mod_depth * 20    // Vibrato with variable depth
~lead: sine(440 + ~vibrato)            // Lead synth with vibrato
```

2. Signal Analysis & Feature Extraction

Any signal can be analyzed to extract control data:

```
// RMS (Root Mean Square) - extract amplitude envelope
~bass_level: ~bass >> rms(0.05)        // 50ms RMS window

// Pitch detection
~vocal_pitch: ~vocal >> pitch

// Transient detection
~kick_transient: bd >> transient

// Spectral centroid (brightness)
~brightness: ~input >> centroid

// Zero crossing rate (noisiness)
~zcr: ~input >> zero_crossings
```

3. Signal Processing Chains

Signals flow through processors using the >> operator:

```
// Basic processing chain
~filtered: ~source >> lpf(2000, 0.7) >> delay(0.25, 0.3)

// Parallel processing with mixing
~stereo: ~mono >> [
    delay(0.020) * 0.5,
    delay(0.023) * 0.5
]

// Complex routing
~complex: ~input >> {
    dry: gain(0.5),
    wet: reverb(0.3) >> lpf(~lfo * 2000)
} >> mix
```

4. Cross-Pattern Modulation

Pattern events can modulate synthesis parameters and vice versa:

```
// Pattern drives filter cutoff
~rhythm: "1 0 1 0"
~filter: ~input >> lpf(~rhythm * 3000 + 500)

// Audio analysis affects pattern playback
~gate: ~input.rms > 0.1
hats: "hh*16" >> when(~gate)  // Hats only play when input is loud

// Sidechain compression via pattern
~kick_pattern: "bd ~ ~ bd"
~compressed: ~bass * (1 - ~kick_pattern * 0.7)  // Duck bass on kicks
```

5. Modulation Routing

Explicit routing statements for complex modulation:

```
// route source -> destination: amount
route ~lfo1 -> filter.freq: 0.3        // LFO modulates filter by 30%
route ~env1 -> osc.pitch: 12           // Envelope modulates pitch by 12 semitones
route ~bass.rms -> reverb.mix: 0.5     // Bass amplitude controls reverb amount

// Multiple destinations
route ~lfo1 -> {
    filter.freq: 0.3,
    delay.time: 0.1,
    pan.position: 0.5
}
```

6. Conditional Logic & Dynamic Routing

```
// Conditional processing
~processed: ~input.rms > 0.5 ? ~input >> distort(0.7) : ~input
```

```
// Switch between sources
~source: ~selector > 0.5 ? ~synth1 : ~synth2

// Threshold gates
~gated: ~input >> gate(~control > 0.3)

// Probability-based routing
~random: ~input >> prob(0.7) ? reverb(0.5) : delay(0.25)
```

7. Feedback Networks

Create recursive signal paths with implicit single-sample delay:

```
// Simple feedback delay
~feedback: ~delay_out * 0.7
~delay_out: (~input + ~feedback) >> delay(0.25) >> lpf(2000)

// Karplus-Strong string synthesis
~exciter: noise() * perc(0.001, 0.01)
~string_feedback: ~string_out * 0.99
~string_out: (~exciter + ~string_feedback) >> delay(1/440) >> lpf(8000)
```

8. Polyphonic Voice Management

```
// Define a polyphonic synth
poly[8] ~synth: {
    osc: saw($freq) * adsr($gate, 0.01, 0.1, 0.7, 0.5),
    filter: osc >> lpf($cutoff * ~lfo1, 0.8),
    out: filter * $velocity
}

// Pattern triggers voices
"c3 e3 g3" >> ~synth  // Automatically allocates voices
```

Complete Example: Bass-Modulated Percussion

```
// === LFOs and Control Signals ===
~lfo_slow: sine(0.25) * 0.5 + 0.5          // 0.25 Hz, normalized 0-1
~lfo_fast: sine(8) * 0.3                   // 8 Hz vibrato

// === Bass Synthesis ===
~bass_env: perc(0.01, 0.3)
~bass_osc: saw(55) * ~bass_env
~bass: ~bass_osc >> lpf(~lfo_slow * 2000 + 500, 0.8)

// === Extract Bass Features ===
~bass_rms: ~bass >> rms(0.05)              // Bass amplitude envelope
~bass_transient: ~bass >> transient        // Bass note attacks

// === Percussion Modulated by Bass ===
~kick: "bd ~ ~ bd" >> gain(1.0)
~snare: "~ sn ~ sn" >> lpf(~bass_rms * 4000 + 1000)  // Bass controls snare filter
~hats: "hh*16" >> hpf(~bass_rms * 8000 + 2000)       // Bass controls hat brightness
```

```
// === Cross-modulation ===
route ~bass_transient -> ~hats.gain: -0.5  // Duck hats on bass attacks
route ~kick.transient -> ~bass.gain: -0.3  // Sidechain compression

// === Master Processing ===
~mix: (~bass * 0.4) + (~kick * 0.5) + (~snare * 0.3) + (~hats * 0.2)
~master: ~mix >> compress(0.3, 4) >> limit(0.95)

// === Output ===
out: ~master
```

Implementation Architecture

Phase 1: Core Signal Graph (Week 1-2)

```rust
// Core signal graph representation
pub struct SignalGraph {
    nodes: HashMap<NodeId, Node>,
    connections: Vec<Connection>,
    buses: HashMap<String, BusId>,
    execution_order: Vec<NodeId>,
}

pub enum Node {
    Source(Box<dyn AudioUnit>),       // FunDSP audio units
    Bus(f32),                         // Signal bus value
    Processor(Box<dyn AudioUnit>),    // Effects, filters
    Analysis(AnalysisType),           // RMS, pitch, etc.
    Pattern(PatternNode),             // Pattern integration
}

pub struct Connection {
    from: NodeId,
    to: NodeId,
    amount: f32,
    tap_type: Option<TapType>,
}
```

Phase 2: Parser & Compiler (Week 2-3)

```rust
// Parse DSL into signal graph
pub fn parse_signal_definition(input: &str) -> Result<SignalGraph, ParseError> {
    // Tokenize and parse the DSL
    // Build signal graph
    // Optimize connection order
}

// Compile to efficient execution plan
pub fn compile_graph(graph: &SignalGraph) -> Result<ExecutionPlan, CompileError> {
    // Topological sort
    // Dead code elimination
    // Constant folding
```

```
    // SIMD optimization opportunities
}
```

Phase 3: Runtime Engine (Week 3-4)

```rust
pub struct ModularEngine {
    graph: SignalGraph,
    sample_rate: u32,
    block_size: usize,
    audio_units: HashMap<NodeId, Box<dyn AudioUnit>>,
    bus_values: Vec<f32>,
}

impl ModularEngine {
    pub fn process_block(&mut self, output: &mut [f32]) {
        // Update control rate signals
        self.update_control_signals();

        // Process audio rate signals
        for node_id in &self.graph.execution_order {
            self.process_node(node_id);
        }

        // Mix to output
        self.mix_output(output);
    }

    pub fn hot_swap_definition(&mut self, name: &str, definition: &str) {
        // Parse new definition
        // Crossfade from old to new
        // Update graph without audio glitches
    }
}
```

Phase 4: Pattern Integration (Week 4-5)

```rust
// Bridge between Strudel patterns and signal graph
pub struct PatternBridge {
    pattern_engine: StrudelEngine,
    signal_graph: Arc<RwLock<SignalGraph>>,
}

impl PatternBridge {
    pub fn process_pattern_event(&mut self, event: &PatternEvent) {
        // Convert pattern events to signal graph updates
        // Handle voice allocation for polyphonic patterns
        // Update bus values from pattern data
    }

    pub fn extract_signal_features(&self, bus_name: &str) -> f32 {
        // Get current value from signal bus
        // Use for pattern modulation
    }
}
```

Performance Considerations

Memory Layout

- **Cache-friendly**: Group frequently accessed data
- **SIMD alignment**: Ensure audio buffers are 16-byte aligned
- **Zero-copy**: Use slices and references where possible

Processing Strategy

- **Block processing**: Process in 64-512 sample blocks
- **Multi-rate**: Control signals at 60-120 Hz, audio at 44.1-48 kHz
- **Lazy evaluation**: Only compute active signal paths
- **Parallel processing**: Use Rayon for independent signal chains

Optimization Opportunities

- **Graph compilation**: Pre-compute static paths
- **JIT compilation**: Use cranelift for hot paths
- **Constant folding**: Optimize away static values
- **Dead code elimination**: Remove unused signal paths

Testing Strategy

Unit Tests

- Signal node processing
- Parser correctness
- Graph compilation
- Feature extraction accuracy

Integration Tests

- Complex signal routing
- Pattern integration
- Live coding hot-swaps
- Performance benchmarks

Example Test Cases

```rust
#[test]
fn test_cross_modulation() {
    let dsl = r#"
        ~lfo: sine(2)
        ~bass: saw(110) >> lpf(~lfo * 2000, 0.8)
        ~bass_rms: ~bass >> rms(0.05)
        ~hats: noise() >> hpf(~bass_rms * 8000)
    "#;

    let graph = parse_signal_definition(dsl).unwrap();
    let mut engine = ModularEngine::new(graph);

    // Process and verify modulation is applied
    let mut output = vec![0.0; 512];
```

```rust
    engine.process_block(&mut output);

    // Verify bass RMS affects hat filter
    assert!(engine.get_bus_value("~bass_rms") > 0.0);
}
```

Migration Path

Stage 1: Parallel Implementation

- Keep existing synthesis system
- Implement new DSL alongside
- Allow gradual migration

Stage 2: Integration

- Bridge old and new systems
- Allow mixed usage in patterns
- Maintain backward compatibility

Stage 3: Full Migration

- Convert existing synthdefs to new DSL
- Deprecate old system
- Full modular synthesis

Future Extensions

Visual Representation

- Generate signal flow diagrams from DSL
- Real-time visualization of signal values
- Interactive debugging interface

Machine Learning Integration

- Train models on signal analysis
- Automatic parameter mapping
- Gesture-to-synthesis mapping

Distributed Processing

- Split processing across cores/machines
- Network-based collaborative performances
- Cloud-based rendering for complex patches

Conclusion

This modular synthesis DSL will position Phonon at the forefront of live coding environments, offering unprecedented flexibility in signal routing and cross-modulation. By treating everything as a signal and allowing arbitrary connections, we enable new forms of musical expression impossible in traditional systems.

The key innovations are: 1. **Universal signal routing** - any signal can modulate any parameter 2. **Pattern-audio bidirectional modulation** - patterns affect audio, audio affects patterns 3. **100% textual** - no GUI required, everything in code 4. **Live codeable** - all connections hot-swappable 5. **Performance-focused** - leveraging Rust and FunDSP

This design brings the power of modular synthesis and DAW-style modulation to the live coding world, opening new creative possibilities for electronic music performance.