Your Flowgraph is the Chip: GNU Radio Compiled Straight to Silicon

Charles McClish CHUCK@LATTREX.COM
Lattrex Inc., 1143 Oak Ridge TPKE, STE 107A PMB 310, Oak Ridge, TN 37830 USA

Abstract

Kyttar is a chip you program by drawing a flowgraph. It is a homogeneous array of small asynchronous compute cells, and a GNU Radio flowgraph maps onto it directly: a block becomes a region of cells, and a connection becomes a route between them. The graph you draw in GNU Radio Companion is essentially the layout of the design on the silicon, so there is no separate hardware-design step to manage. placeKYT is the open-source tool that does the mapping. It reads a flowgraph, places and routes it onto the array, runs it, and sends the output back into GNU Radio through custom source and sink blocks, so from the outside it looks like ordinary GNU Radio.

The harder problem is the block library. Every Kyttar block so far has been written by an AI agent. The placeKYT source includes the block-generation workflow: the prompts, the rules an agent follows to lay a block out across cells, and testbenches that check each generated block against its GNU Radio equivalent. A running knowledge base captures what works as blocks are built, so lessons learned on one carry into the next. Running unattended, the workflow has produced and verified 113 blocks, measuring each block's quantization noise against its GNU Radio floating-point original. We instrumented 62 of those builds: a median of 232,000 tokens and 31 minutes of wall-clock time per block, and 51 of 62 verified on the first attempt.

This runs today on simKYT, a free, cycle-accurate model of the chip. A 120-cell prototype is in fabrication, with first silicon expected at the end of November 2026.

1. Introduction

Software-defined radio has always lived in tension between two implementation targets. CPUs and DSPs are easy to program, but process samples serially. For highly parallel workloads this requires sequencing operations carefully or more hardware to handle the workload. FPGAs offer massive parallelism, but are programmed at the logic level. Algorithms are written at a higher level of abstraction and mapped to the logic level in a hardware description language (HDL).

This paper describes a third option. Kyttar is a massively parallel processor array (MPPA) built from a homogeneous array of identical asynchronous compute cells. To map a GNU Radio flowgraph onto this chip, we have developed a tool called placeKYT. This tool takes the flowgraph as an input, maps each block onto compute cells, routes the data streams between them, and produces a bitstream to load onto the Kyttar chip itself.

We make four contributions:

- A GNU Radio native programming model for a spatial, asynchronous fabric, in which a flowgraph maps onto a grid of compute cells (Section 3).
- placeKYT, the tool that places, routes, hosts, and debugs a flowgraph on the array (Section 4).
- An increasingly automated block-generation pipeline, with measured cost per generated block (Section 5).
- A set of complete SDR transceivers built and validated on a single prototype array as evidence (Section 6).

As of the publishing of this paper, the chip has not yet returned from the fab. The results reported here are from a cycle-accurate simulator, simKYT, characterized against SPICE and SDF timing/power data from the design itself.

2. Kyttar Chip Details

The prototype Kyttar chip is a 120-cell array on the SkyWater SKY130 process, arranged in a 10x12 grid. Each compute cell contains 32 words of memory, a 16-bit ALU with a fixed-point (Q15) multiply-accumulate unit, and the ability to execute out of its internal memory. Cells communicate only with their nearest neighbors in the 4 cardinal directions using two special instructions: JUMP and WRITE. The JUMP instruction initiates execution on another cell at a specified address. The WRITE instruction will write a

word of data to another cell at a specified address. Data can be received on any of the 4 faces. The cell transmits data out of a single face specified by a configuration register, which can be modified during run time. Each cell can forward JUMP and WRITE instructions to write data to or initiate execution on cells further away. Data transactions, both inside and outside of cells, utilize a bundled-data handshake. There is no clock in this system to trigger events or time data transactions. A cell acts when an input arrives, otherwise the cell is idle, drawing only leakage current.

Another aspect worth noting is arbitration and locking. A cell can receive data on any of its 4 faces. While executing, the cell blocks incoming data transactions. Once internal execution is complete, it will receive the next batch of data from one of the faces. Arbitration is asynchronous and uncontrolled: the first handshake to pass through the arbiter wins while the others wait, naturally forming a round-robin priority scheme. The cells have the ability to lock arbitration to a particular face. This is useful for multi-cell blocks that execute over several iterations, blocks that need to receive data from multiple sources before calculating a final result, or to handle control flow and branching operations. The lock function and which face to lock to are controlled by cell level configuration registers and can be modified during execution.

The mental model a GNU Radio user needs is simple. A flowgraph is a graph of streaming blocks connected by data flows. The Kyttar array is a grid of cells connected to their neighbors through a handshaking protocol. Compiling one onto the other is the placement and routing of a dataflow graph onto that grid. A block becomes one or more cells containing the assembly instructions in its memory to perform the necessary functionality. A connection between blocks becomes a route through the grid. Figure 2 shows the same BPSK receiver both ways: the flowgraph the user draws, and the design placed and routed on the array.

What this buys an SDR developer:

- **Spatial parallelism.** Independent parts of a flowgraph run at the same time on different cells. Cells not executing only consume leakage current, so idle cells do not eat into the power budget.
- **Fast reconfiguration.** The array can be reprogrammed from reset in roughly 100 microseconds. A different waveform is a different program, not a different chip.
- **No timing closure.** There is no clock to constrain and no static timing to sign off in the user flow; correctness is a property of the dataflow, not a cycle budget.

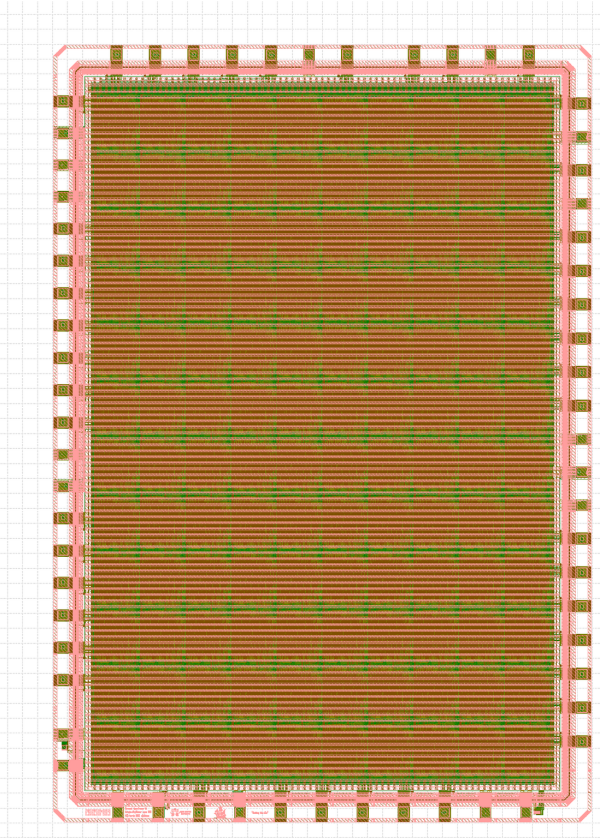

Figure 1. The KYT16A120 prototype die: 120 cells in a 10x12 array on SkyWater SKY130, placed and routed entirely with open-source tools.

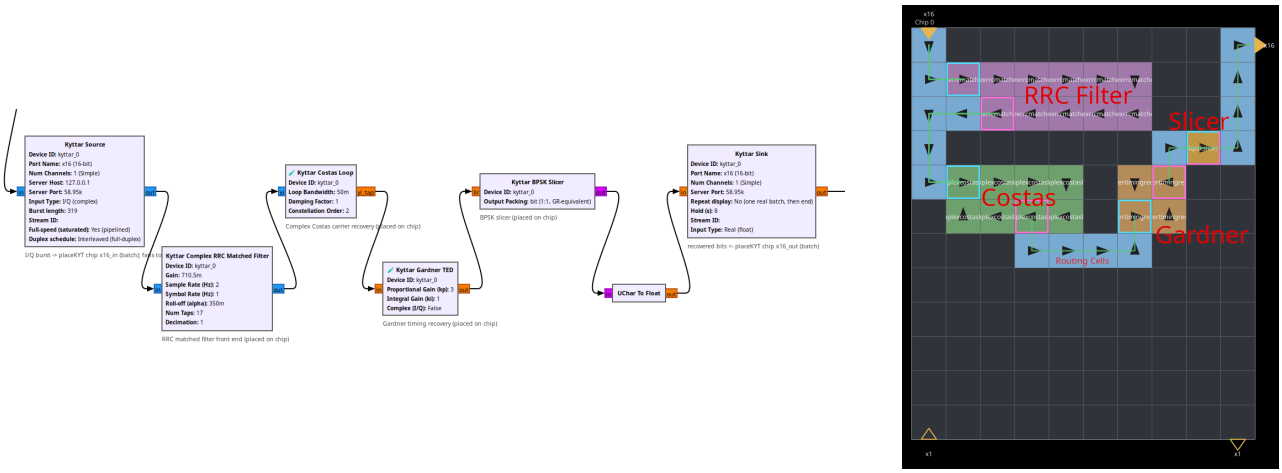

Figure 2. One BPSK receiver, 2 views. Left: the GNU Radio flowgraph the user draws, a Kyttar Source feeding an RRC matched filter, a complex Costas loop, a Gardner timing recovery, and a BPSK slicer, out to a Kyttar Sink. Right: the same design placed and routed on the 120-cell array. Each block becomes a coloured region of cells sized to the work it does, the RRC matched filter spanning 11 cells and the Costas loop 8, while the connections between blocks become routes across the grid (green). The blue cells carry data without computing on it. The graph the user drew is the layout on the silicon.

3. From Flowgraph to Fabric

The programming flow is deliberately close to ordinary GNU Radio development. The user builds a flowgraph in GNU Radio Companion from a library of Kyttar blocks, each of which corresponds to a known GNU Radio block. The flowgraph is imported into placeKYT, which places every block onto cells and routes the sample streams between them, across the grid and, on multi-chip boards, across chip boundaries.

3.1. The Block Library

As of the writing of this paper the library contains 113 blocks (Section 5). Table 1 summarizes the library by category. Every block names the exact GNU Radio factory it is checked against, so the equivalence is never a matter of guesswork; the full per-block table, including the measured error against the reference, ships with the tool. The library and examples are public: https://github.com/Lattrex/placekyt

There are 2 special blocks that define the boundaries of a data stream to be processed on the chip. The Kyttar Source block is used to define an input coming into the Kyttar chip, while a Kyttar Sink block is used to define data leaving the chip. These are not represented in the placeKYT array, but act as intermediaries between the chip itself and GNU Radio. Each data stream going through the source and sink blocks is tagged to the specific cell receiving or transmitting the data, as well as by which port and which chip (in the case of multi-chip configurations) the data is transmitting through. For the purposes of this paper, the chip is hosted in the simKYT simulator. On real hardware, these blocks are implemented in a companion CPLD, FPGA, or MCU that connects the PC hosting GNU Radio to the Kyttar chips, over the desired physical channel.

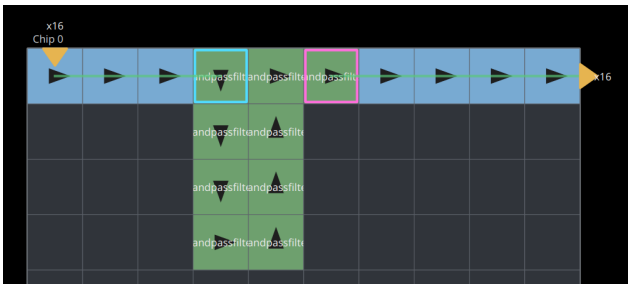

Figure 3. A band-pass FIR filter as a multi-cell spatial pipeline. The filter (green) folds down and back through 8 cells, each holding a share of the taps and passing its partial sum to the next. Data enters from the port at the left, transits the routing cells (blue), runs the fold, and exits right. Longer filters extend the fold rather than changing the pattern.

3.2. Multi-Cell Blocks

A block larger than a single cell is spread across several cells as a spatial pipeline. A long FIR filter, for example, is distributed so that partial sums flow from cell to cell (Figure 3). The taps are divided among the cells, each cell accumulating its share of the partial sum and handing the running total to the next, so the filter folds into a compact region of the array rather than occupying one very long row. This pattern defines how the design scales onto larger arrays and across chips.

Category	Representative blocks
Arithmetic and type	add, subtract, multiply, gain, constants (real and complex); float/char and complex/float conversion; conjugate, magnitude, magnitude-squared, argument, absolute value, $n \log_{10}$
FIR and IIR filters	low, high, band-pass, band-reject; root-raised-cosine matched and pulse-shaping; IIR biquad; moving average; DC blocker; complex/IQ variants of each
Mixing and frequency	complex mixer, NCO, IQ upconversion, frequency-translating FIR, frequency modulator
Rate conversion	upsampler, decimating and interpolating FIR, rational resampler, repeat, keep-one-in-N
Synchronization and equalization	Costas loop, 16-QAM constellation receiver, band-edge FLL, Mueller-and-Müller timing recovery, 4-FSK syncword timing recovery, LMS linear equalizer
Symbol mapping and slicing	PSK, 4-FSK, and 16-QAM mappers and slicers; binary slicer; soft-decision demodulator
Coding and framing	Hamming, Golay, CRC-16, block interleaver, differential encode/decode, LFSR scrambler, pack/unpack k bits, Varicode
Conditioning	AGC (real and complex), squelch, RMS
Amateur-mode native	CW keyer and decoder, raised-cosine envelope
Transforms	streaming radix-2 FFT at 16, 32, 64, and 128 points; butterfly; twiddle multiply; multi-cell complex delay line
Machine learning	gated recurrent unit (GRU) cell with classifier head; sigmoid; tanh; dot-product multiply-accumulate
Chirp spread spectrum	chirp generator, symbol mapper, conjugate-chirp dechirp mixer, preamble sync, bin argmax
Logic and routing	XOR, NOT, AND-const, map, delay, stream splitter, crossover

Table 1. The Kyttar block library by category. 113 blocks, each verified against a named GNU Radio reference or a numerical golden.

3.3. The Two-Terminal Workflow

Running a design feels like normal SDR work. In one terminal the user hosts the chip in placeKYT; in another, the same flowgraph runs in GNU Radio Companion, streaming stimulus into the hosted device and plotting the result that comes back. The Kyttar GNU Radio blocks run on the array, while the non-Kyttar GNU Radio blocks are the front end that feeds the chip and displays its output.

4. placeKYT: Place, Route, Host, Debug

placeKYT is the tool that turns a flowgraph into a running design. It imports the flowgraph, places blocks onto cells, routes the streams between them, generates the program for the device, and hosts the design in a cycle-accurate simulator (and, on hardware, programs the physical part). The user then drives it from GNU Radio.

4.1. Placement and Routing on a Spatial Fabric

Mapping a dataflow graph onto a grid with only nearest-neighbor links is a placement and routing problem. For most designs placeKYT places and routes automatically from the imported flowgraph. A few dense, hand-tuned designs (for example the Weaver SSB transceiver and the 4-FSK and 16-QAM modems) are shipped as pre-placed designs that are opened directly rather than auto-routed. Figure 4 is the hand-optimized Weaver SSB transceiver, contrasting with the auto-routed BPSK receiver of Figure 2. With the current algorithm, utilizations above 60% with multiple large blocks are not fully reliable for automated place and route.

Manual routing is supported in placeKYT with flylines and block level transforms (rotation, mirroring). Multiple streams can be multiplexed on a single routing corridor. Moving blocks or reconnecting them automatically triggers output direction and distance calculations to be performed, so the user only needs to make a legal placement and route; everything else is handled under the hood. For violations, the tool provides a DRC window which specifies where the violations are and highlights them for review.

4.2. Visibility and Debug

Because the whole design is spatial, placeKYT can animate it: as the design runs, cells light up as they execute and per-word markers show data hopping from cell to cell along its route to the output. Combined with instruction-level stepping, this makes it possible to see exactly where a signal enters, which cells compute, and how it flows to the egress port, a level of visibility that bitstream flows do not offer. Alongside the array view, placeKYT provides register and instruction-memory inspection for any selected cell, breakpoints, and a waveform viewer with traces captured directly off the chip ports, so a signal can be followed from the port through the cells that compute it and back out.

This feature is very useful for detecting lockup conditions when debugging new blocks or certain routes during fully saturated computation. Cells that lock up typically get into a condition where 2 or more data streams converge on the same cell, with the input and output paths colliding on the same face. When multiplexing data streams on the same routing channel it is very important that data run in the

Figure 4. The Weaver SSB transceiver, a dense hand-placed design occupying most of the 120-cell array. Both the transmit and receive chains are resident at once: complex mixers (orange and green), complex low-pass filter banks (green and tan), and IQ upconversion (red and purple), with routing cells (blue) threading between them. This is the layout that produces the SSB row of Table 3.

same direction. The same is true inside of complex blocks with feedback loops. Being able to visualize the exact cells involved makes pinpointing the problem much easier.

4.3. The Simulator

The host simulator is cycle-accurate and characterized against SPICE and SDF timing and power data at the typical corner (1.8 V, 25 °C, TT), so the throughput and power figures it reports correlate to the physical device. This same simulator served as the predictor model during design verification.

5. Automated Block Generation

A spatial fabric is only as useful as its block library, and hand-authoring blocks is the bottleneck. Our main tooling contribution is that block development is now a largely automated pipeline. An agent is dispatched into an isolated git worktree with one instruction: build this GNU Radio block for Kyttar, or quarantine it, reporting why and where the agent got stuck. It reads a knowledge base of substrate rules accumulated from earlier builds, writes the cell assembly and the block's placement across cells, and then verifies its own work: it stands up the block on the cycle-accurate simulator as the device under test, runs the named GNU Radio block as the golden reference on the same stimulus, and compares them sample by sample against a quantization-noise budget. It also runs a set of deliberately broken mutations of its own block and requires each one to fail, which catches tests that pass for the wrong reason. A build ends in one of two states: verified and merged, or quarantined with a written reason. What it learned is appended to the knowledge base for the next block.

Wave	N	**Tokens** (median)	**Wall** (median)	**First-pass**
1	14	166,010	14.4 min	10/14
2	6	249,165	29.7 min	2/6
3	4	202,140	34.5 min	4/4
4	8	195,578	23.6 min	8/8
5	10	305,138	41.7 min	10/10
6	20	398,699	84.6 min	17/20
All	62	235,080	32.1 min	51/62

Table 2. Cost per generated block, by wave. Tokens are agent output tokens. Waves 1 to 5 are the DSP library; wave 6 is the transform, inference, and spread-spectrum work. All blocks verified against their reference; none were abandoned. Total 21.3 M tokens and 70 hours of wall-clock time. The Gardner timing loop is split into its first pass (wave 2) and its completion (wave 6), which is when each was actually run.

5.1. Measured Cost

The pipeline is instrumented so the automation claims can be verified. Every dispatched build writes a record of token usage, tool-call turns, wall-clock time, attempts, human interventions, and outcome.

We report 62 instrumented library-block builds run across 6 waves between 2026-08-06 and 2026-08-25 (Table 2). A small number of blocks were co-built inside a sibling block's session and carry no separately attributable token count.

Across the 62 builds: 21.3 M output tokens total, a median of 235,080 tokens and a mean of 343,308 per block, and a median of 32.1 minutes of wall-clock against a mean of 68. 51 verified on the first attempt, 6 needed a human, and all 62 verified.

There are some interesting data points from these runs, as shown in Figure 5. Cost per block rose over the early runs, from a median of 176,396 tokens across waves 1 and 2 to 287,101 across the later waves. There were several reasons for this. The early waves were mostly single-cell primitives (XOR, NOT, bit packing), while waves 3 to 5 were AGC, band-edge FLL, Golay coding, block interleaving, and the LMS equalizer. Larger more complicated blocks burn more tokens. In addition, the knowledge base itself grows, so each later build read more accumulated context.

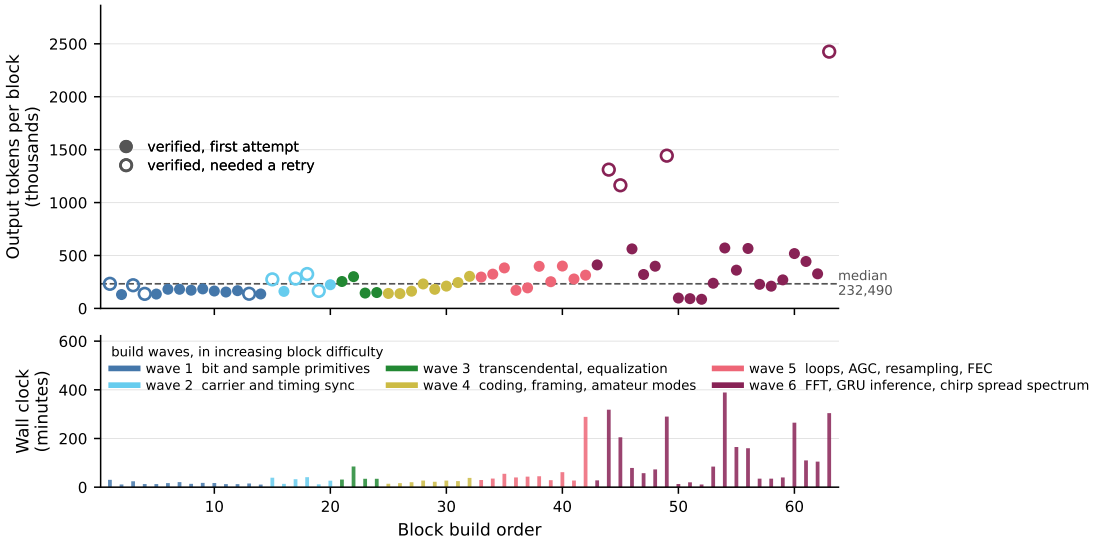

Figure 5. Cost per generated block, in build order, for the 62 instrumented library-block builds. Top: agent output tokens, coloured by wave; filled markers verified on the first attempt, open markers needed one or more retries. Bottom: wall-clock time for the same builds. Cost is flat and cheap through the DSP library (waves 1 to 5) and rises in wave 6, which changes the workload class to streaming FFTs, recurrent inference, and chirp spread spectrum: the median goes from 166,010 tokens in wave 1 to 398,699 in wave 6. The most expensive builds are the Gardner timing loop (2.43 M tokens over 5 dispatches for the wave 6 completion alone, the point at the far right), the GRU classifier example (1.44 M, 4), the 128-point FFT (1.31 M, 4), and the 64-point FFT (1.16 M, 3). Every one of them finished, and none failed on arithmetic: the cost went into placement rules, the multi-chip data path, and in the Gardner case a sequence of confidently wrong diagnoses. The Gardner block appears twice, as its first pass in wave 2 and its completion in wave 6, which is when each was run.

The retries in those early waves reflect the agents discovering how this substrate behaves and what the reference actually does. Building the scrambler required pinning GNU Radio's LFSR convention by probing it with an all-zeros input, which showed the existing hand-written block had used the wrong polynomial. The complex gain block wrapped instead of saturating above unity gain, an error of 27,525 LSB that every prior modem had hidden because its constellation never left the safe range. The root-raised-cosine matched filter turned out to have invented its own taps rather than using `firdes`. Unpacking k bits overflowed a cell's 31 usable addresses when fully unrolled and had to be rewritten as a counted loop. Each of those cost a second dispatch, and each produced an invariant that later blocks were checked against.

During wave 6 the difficulty ratcheted up again, with the streaming FFTs, the recurrent-inference cell, and the chirp spread spectrum chain. The median token count jumped to 398,699 tokens and wall-clock time to 84.6 minutes. The mean of 573,575 is pulled well above that by a small number of very expensive builds.

The 4 most expensive blocks in the campaign were the ones that had to be reworked. The Gardner timing loop took 2.59 M tokens over 7 dispatches counting both passes, the GRU classifier example 1.44 M over 4, the 128-point FFT 1.31 M over 4, and the 64-point FFT 1.16 M over 3. **All 4 finished.** For the 3 large blocks what consumed the effort was an improper placement constraint and the multi-chip data path. The Gardner block was a different case: it was twice recorded as a quarantine, once blaming Q15 precision and once blaming a placement rule, and both diagnoses were later disproved by the agent's own measurements. The actual fault was in the timing detector's strobe structure, and the block now verifies bit-exact against `symbol_sync_cc`. For all previous blocks, the library reserved a routing channel on either side and capped a block at 8 cells across, which is correct for a block that has to sit next to other blocks. A block that occupies half the array on a die does not need this constraint. It needs only its input and output reachable from one side. Given the relaxed rule for large sole-occupant blocks, they placed as quickly as everything else, and the later large blocks in the same wave built successfully on the first attempt.

The 128-point transform was another new case. This block is so large it does not fit on one die at all. No prior block had spanned dies, so the agent had to choose a stage boundary to cut at, place both halves, and work out the cross-die handoff, with nothing in the knowledge base to draw on. It found that a complex sample is a 3-part transaction and that each part must be pumped to quiescence before the next is

injected, which is the kind of substrate detail that costs a great deal to discover once and is much cheaper to reuse.

For both the early and later retries, the knowledge base proved its value. Builds after the findings were recorded completed in line with their complexity and did not require retries until the next unknown or improper constraint was encountered.

6. Worked Examples

As evidence, we built 7 complete transceivers, each from a GNU Radio flowgraph through the flow above, and each fitting on one 120-cell array. Every one is a full transceiver: the transmit and receive chains are co-resident on the same array and share one input port and one output port. Table 3 reports them.

Design	Mod.	RX rate	RX pwr	TX rate	TX pwr	Corr./BER
BPSK	1 b/sym	188 kSa/s	9.6 mW	481 kSa/s	7.6 mW	BER 0
QPSK	2 b/sym	172 kSa/s	8.2 mW	460 kSa/s	9.1 mW	BER 0
4-FSK	2 b/sym	542 kSa/s	15.2 mW	225 kSa/s	4.2 mW	BER 0
16-QAM	4 b/sym	146 kSa/s	8.9 mW	460 kSa/s	11.5 mW	BER 0
AM	analog	460 kSa/s	10.1 mW	479 kSa/s	5.9 mW	0.998
FM	analog	1.93 MSa/s	6.3 mW	429 kSa/s	5.7 mW	0.996
SSB	analog	346 kSa/s	14.0 mW	346 kSa/s	14.4 mW	0.97

Table 3. 7 complete transceivers, each on one 120-cell array. Simulation measurements on simKYT, characterized against SPICE and SDF timing and power data. Rates are the sink (output) sample rate at the simplex saturated operating point, each direction driven alone. Power is the chip total (active plus idle) while that chain runs. Idle power is 0.4 to 0.6 mW across all designs. Digital modems are verified at bit-error-rate 0; analog transceivers by audio correlation against the GNU Radio floating-point reference.

There are 2 points to consider regarding this data. First, the figures are *simplex and saturated*: each direction is driven alone with the whole burst queued back to back, so each chain reaches its own compute-bound ceiling with no cross-chain contention. The full-duplex rate depends on how the 2 chains time-slice the single shared port, which is contention-window sensitive and easy to misread, so we report the clean per-chain number and ship the interleaved schedule as a user-selectable option. Second, each direction runs on its own freshly hosted chip, which is what makes the RX and TX power figures separable.

The same flow produced blocks well outside the modem set. Streaming radix-2 transforms run at 16, 32, 64, and 128 points, the first bit-exact against `numpy.fft` on 44 cells, the 64-point version on 84 cells at 52 to 87 dB SNR depending on stimulus against a 51 dB floor derived beforehand. A large transform had been the standing objection to this architecture, and the objection was about the wrong transform: a block FFT holds all N samples at once and will not fit cells of 32 words, while a single-path delay-feedback pipeline holds a delay line of $N/2^k$ words per stage and shrinks geometrically down the chain. The 128-point transform does not fit one die and runs split across 2, 30 cells on the first and 84 on the second, bit-exact end to end on the placed and routed build.

A gated recurrent unit with a 4-class head classifies SSB, BPSK, 4-FSK, and noise from a 2-feature stream, occupying 102 of 120 cells and agreeing exactly with a chip-exact offline model. Weights are trained offline in PyTorch and loaded as constants; no training happens on chip. A GRU timestep is 3 gated matrix-vector products, 2 nonlinearities, and an elementwise blend, which decomposes into a multiply-accumulate chain, a table-and-interpolation approximation, and a feedback path, all patterns the library already had. A chirp spread spectrum receiver reuses framing, CRC, Hamming coding, scrambling, and magnitude unchanged, adding only the chirp-specific blocks and a transform.

These were not built as a separate exercise. They went through the same generation and verification flow against the same equivalence bar as the filters and slicers, and what they cost is reported in Table 2 alongside everything else. Their value here is that they define the edge of what is feasible with this prototype configuration: a 128-point transform needs 2 dies, the classifier needs 102 of 120 cells, and the delay lines press on per-cell memory. Those are measurements of what workloads a 32-word cell and a 10x12 array can process effectively.

The spread across the table follows from the architecture. FM receive is the fastest demodulator at 1.93 MSa/s because its receive path is a bare quadrature discriminator, a multiply-accumulate of the conjugate product with no feedback loop, running on 3 active cells. SSB is the heaviest at roughly 14 mW in each direction because the Weaver third-method topology needs complex FIR filtering across 35 to 38 active cells. The rate a chain achieves is set by its longest sequential dependency, and the power it draws is set by how many cells that takes.

7. Discussion

7.1. Where the Fabric Fits

The ideal workloads for this architecture are streaming, nearest-neighbor, low-precision, latency-tolerant, and parallel across many narrowband channels. It is a strong fit for many-channel SDR, spectrum monitoring, and the open, evolving software-defined voice and data modes that have no fixed-function silicon precisely because they keep changing.

The real constraint is per-cell memory, and the transforms show exactly where this cell design buckles. A 32-word cell holds a short delay line and little else, so the early stages of a large transform consume cells quickly: the 64-

point version needs 84 of the 120 available, and the 128-point version does not fit a single die. Anything holding significant state per sample runs into the same wall. The workloads still run, and they run bit-exact, but they run wide across many cells.

Before the wave 6 block builds, the working assumption was that transforms of any size were out of reach on cells this small. That holds for a block transform, which holds all N samples at once, but a streaming pipeline spreads the samples across cells and stages. Knowing what a 128-point transform, a recurrent classifier, and a spread-spectrum receiver each cost on the current cell provides guidance for the next revision's memory sizing calculation. Porting the library forward should also be cheap, because the same generation flow and the same knowledge base apply to a new instruction set.

7.2. Reconfigurability and Power

The two properties we believe matter most are reconfigurability and power. One part can be reprogrammed to a different waveform without a respin, which suits open modes and multi-mode devices. And because only active cells draw current, many channels can share one die while power is paid only for the ones that are busy. The value proposition is size, weight, and power.

8. Conclusion

A GNU Radio flowgraph can be compiled and run directly on a clockless spatial processor. placeKYT makes that a workflow that feels like ordinary SDR development, block development for the fabric is now largely automated at measured cost, and a suite of complete transceivers validates the approach on a single prototype array. Streaming transforms, a recurrent classifier, and a spread-spectrum receiver run on the same fabric at a cost we can now quote rather than estimate. First silicon returns at the end of November 2026. We invite the community to try the tooling and to help identify where this architecture fits best.