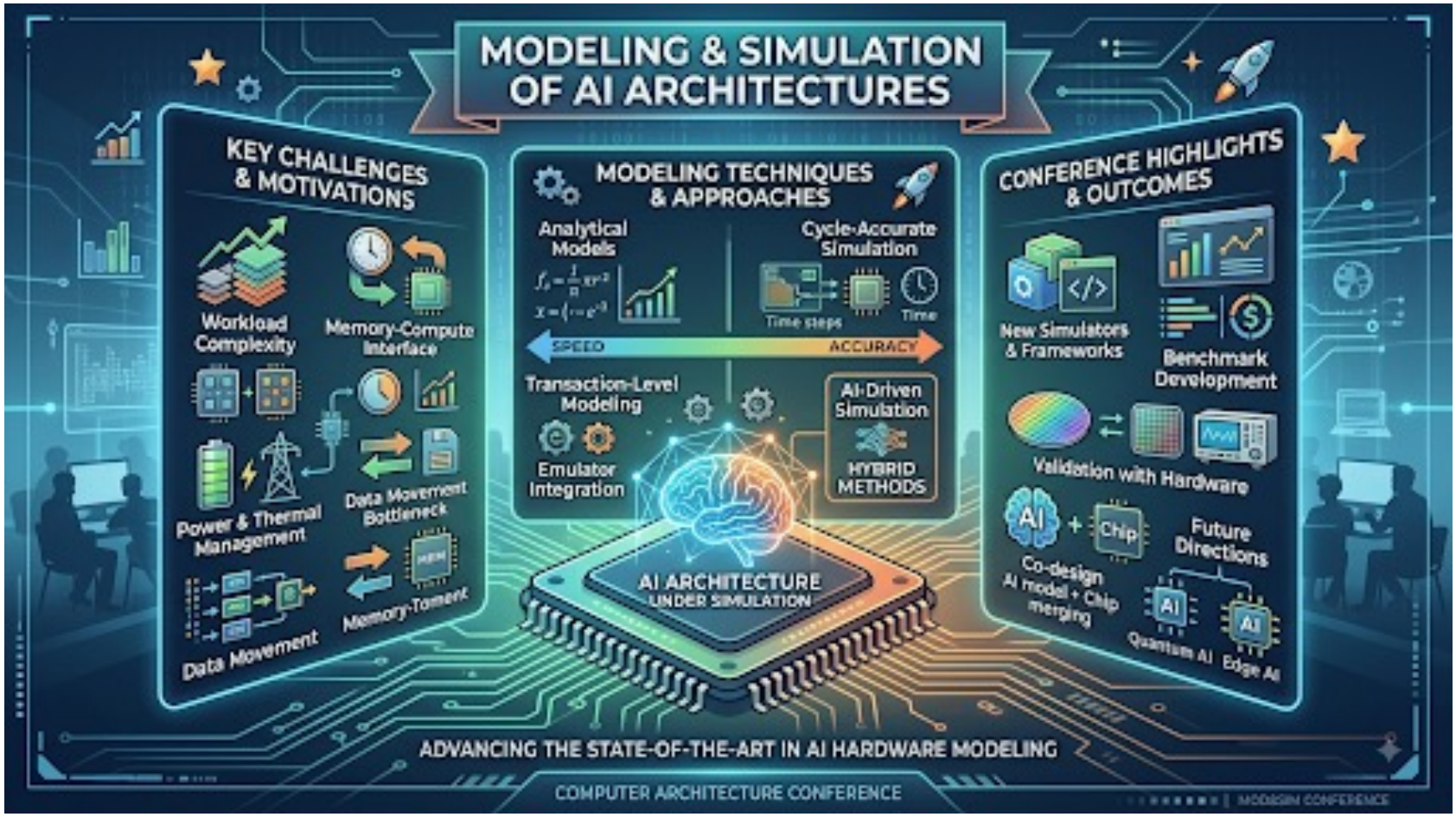
# The Future of ModSim when Considering Advanced AI Architectures

David Donofrio

ModSim 2026

Seattle, WA

**Minimal design to prove not AI generated*


MODELING & SIMULATION OF AI ARCHITECTURES
KEY CHALLENGES & MOTIVATIONS
Workload Complexity
Memory-Compute Interface
Power & Thermal Management
Data Movement Bottleneck
Data Movement
Memory-Torment
MODELING TECHNIQUES & APPROACHES
Analytical Models
Cycle-Accurate Simulation
Time steps
Time
SPEED
ACCURACY
Transaction-Level Modeling
Emulator Integration
AI-Driven Simulation
HYBRID METHODS
AI ARCHITECTURE UNDER SIMULATION
CONFERENCE HIGHLIGHTS & OUTCOMES
New Simulators & Frameworks
Benchmark Development
Validation with Hardware
AI
Chip
Co-design AI model + Chip merging
Future Directions
Quantum AI
Edge AI
ADVANCING THE STATE-OF-THE-ART IN AI HARDWARE MODELING
COMPUTER ARCHITECTURE CONFERENCE

# Simulation – Circa 2025

- Plethora of models / libraries
  - gem5
  - SST
  - SystemC
  - Etc.
- Interaction between models limited and brittle
- Simulation development labor intensive
  - Makes intersection with product, paper, or other deadlines challenging
- Explosion of AI architectures is making this problem worse
  - Heterogeneity increasing
- Testing, analysis, workloads, remain persistent issues

# Many Emerging Hardware Architectures

- New architectures continue to emerge
  - GraphCore, Groq, Tenstorrent, …
- Accelerators owning a larger share of the compute performance gains
  - Simulation models growing in importance
  - Capture the *actual* performance gains of these accelerators
- How can simulation models keep up?

# Traditional Method

- Pick framework / library
- Understand new architecture
- Code from scratch
- Test
- Fix (some) bugs
- Re-test
- ….

# AI Powered Code Generation

- ***Stop writing simulators, Start writing specs***
- AI can generate websites.. But can it write simulators?
  - Turns out… it can!
- Claude Code utilized, Opus 4.8+
- Already has basic arch knowledge
  - Boilerplate components, parameters are well understood
- Add guardrails through SST Skills Files

# Introducing…. Clawd-SST

- Create Claude skills files for the SST Simulator
- Start simple – can it create a new cache model?
  - Rev + simple cache / memory hierarchy


**Not associated with SNL*
***Generated by AI*

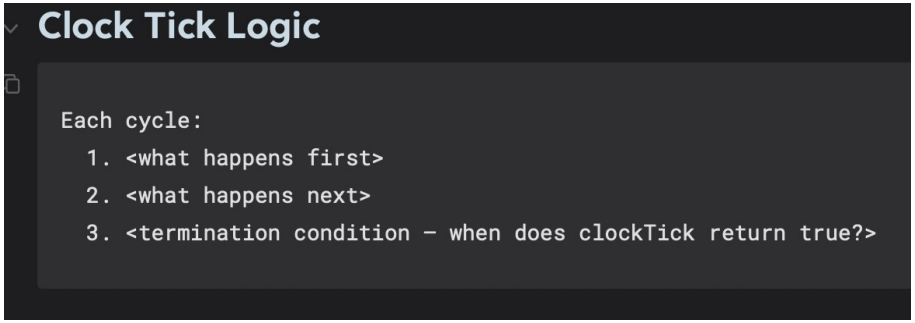
# Clawd-SST

- Top level CLAUDE.md
  - Generic stuff:
    - Copyright header, build system, namespace, etc.
  - More Interesting stuff:
    - Component and SubComponent templates!
  - Coding conventions
    - SST::output vs. std::stdout, memory allocation rules, std::unique_ptr usage, etc.
  - SST Specific items:
    - Python configuration, clock calling conventions, file organization, etc.

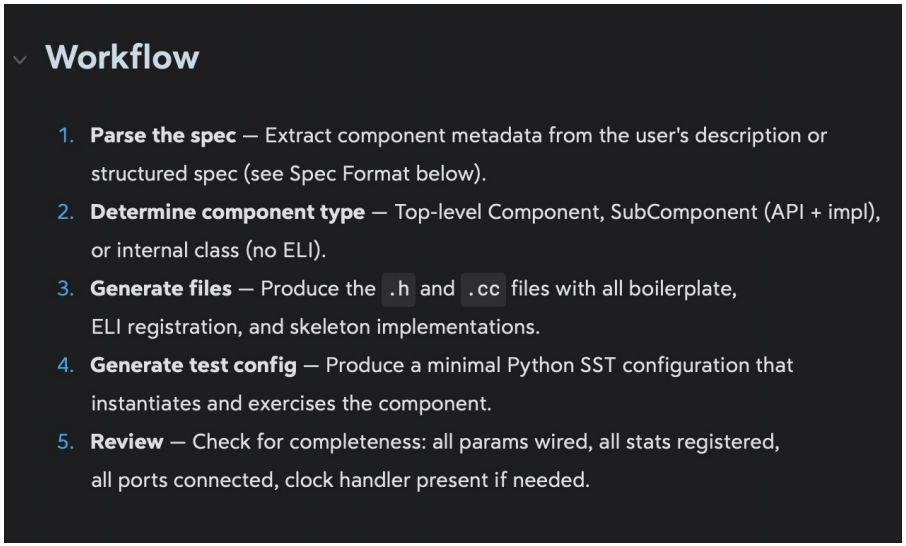
```
class MyComponent : public SST::Component {
public:
  MyComponent(SST::ComponentId_t id, const SST::Params& params);
  ~MyComponent() = default;

  // Lifecycle
  void setup() final;
  void finish() final;
  void init(uint32_t phase) final;

  // Clock handler - return true to disable component
  bool clockTick(SST::Cycle_t currentCycle);

  // ELI Registration (REQUIRED)
  SST_ELI_REGISTER_COMPONENT(
    MyComponent,          // class
    "mylib",              // library name (used in Python config)
    "MyComponent",        // component name
    SST_ELI_ELEMENT_VERSION(1, 0, 0),
    "Description string",
    COMPONENT_CATEGORY_UNCATEGORIZED  // or COMPONENT_CATEGORY_PROCESSOR,
etc.
  )

  SST_ELI_DOCUMENT_PARAMS(
    {"param_name", "Description", "default_value"},
  )

  SST_ELI_DOCUMENT_PORTS(
    {"port_name", "Description", {"event.type"}},
  )

  SST_ELI_DOCUMENT_SUBCOMPONENT_SLOTS(
    {"slot_name", "Description", "SST::Namespace::SubCompClass"},
  )

  SST_ELI_DOCUMENT_STATISTICS(
    {"stat_name", "Description", "unit", enable_level},
  )
};
```

# Clawd-SST

- Specific SKILL.md files
  - Component Generation Skill
    - Composed of a specification:
      - Statistics, SubComponent, Connections / Links to other components
      - How to structure a ClockTick
      - How to handle events
      - Default ports and parameters to include
    - … and a specific skill:
      - Template for a component spec
      - Constructor template
      - ELI Macros + Rules



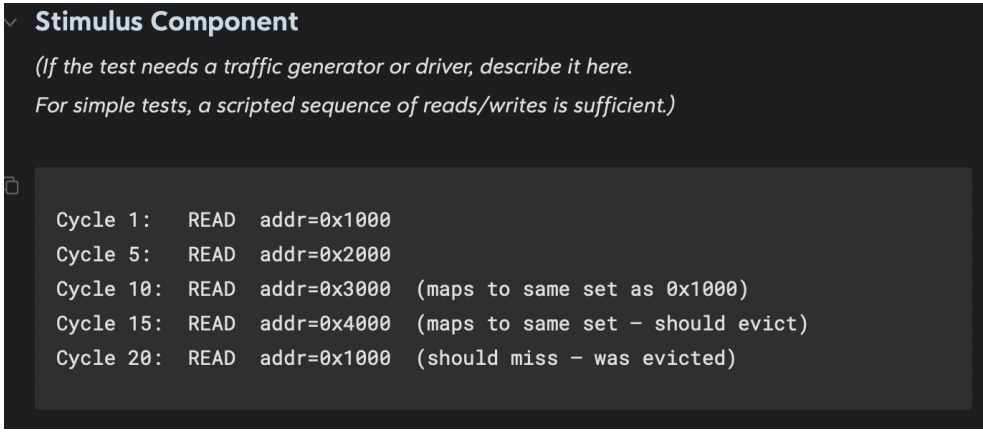

*Spec: Outline for Clock Tick Functionality*



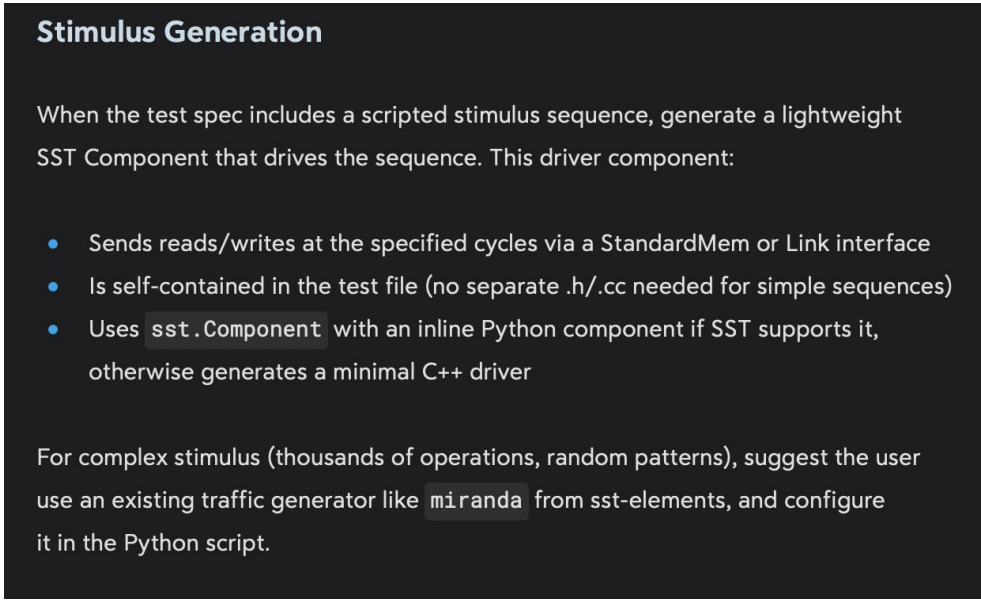

*Skill: Outline for component generation*

# Clawd-SST

- Don’t forget about testing…
- Testing SKILL
  - Test template
    - Test objective, configuration, stimulus and expected results, behavioral checks
    - Generic ”Notes” section for edge cases or specific checks
  - SKILL.md
    - Read the template and generate the test
    - File organization, etc
    - Create a component that creates the targeted test pattern
    - Create the Python config (template included)
    - Create scripts to run tests and verify results




*Template: Example stimulus generator*




*Skill: Example stimulus component generation*

# Rev – A RISC-V Processor Model

- SST Component
- Supports RV64GCA
- Runs native ELF Binary
  - Use vanilla GCC
- Runs bare-metal (newlib)
  - Support for some OS creature comforts
- Snappy at 3+MIPS
  - Scalable – 10K+ core simulations successfully run
- Supports a co-processor interface
  - Unknown instructions are trapped
  - Instruction passed to attached co-processor for execution / validity
  - Host core stalls until result
- Coded the old-fashioned way – by hand



# Test Drive: Rev + Cache Hierarchy

- Attach a RevCore to a MESI-enabled cache
- Build on existing skills for components and testing
- Create a cache_spec.md
- Note that you can describe standard features directly
- Unique features can be included with pseudo-code or a text description
  - For example, a custom replacement policy

**Parameters**

| Name | Type | Default | Description |
|---|---|---|---|
| verbose | uint32_t | 0 | Verbosity level |
| clock | string | 1GHz | Clock frequency |
| lineSize | uint32_t | 64 | Number of bytes per cache line |
| lineCount | uint32_t | 16 | Number of lines in the cache |
| associativity | string | full | Describes the level of cache associativity. Valid options are 2-way, 4-way and fully |
| hitLatency | uint32_t | 3 | Number of clock cycles to access the cache, provided it is a hit |
| inclusive | bool | true | specifies if cache is inclusive |
| replacement | string | LRU | "Replacement policy: LRU, FIFO, Random" |
| noncacheable_regions | string | "[]" | "vector of (start, end) address pairs for noncacheable address ranges. Vector format should be [start0, end0, start1, end1, ...]" |

# Results

- Success!
- Clean coded MESI Cache model
- Multiple Replacement policies
- Non-cacheable region support
- Connects via SST Standard Memory interface

- SST Ready
  - ELI
  - Stats
  - Clock
  - Serializable
- Tests
  - Traffic generator hooked up to cache model

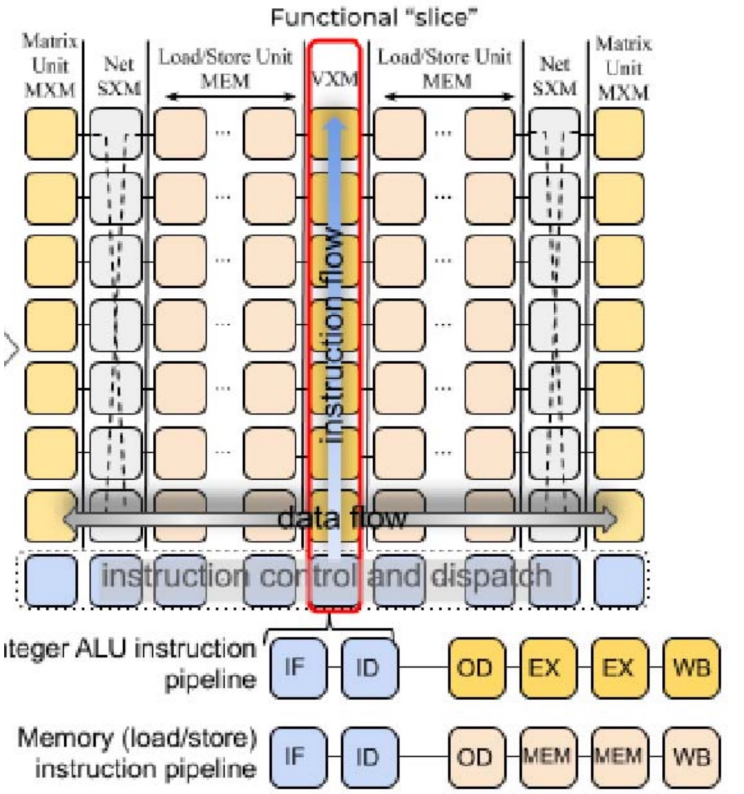
```
import sst

# --- MESICache component ---
cache = sst.Component("test_cache", "Memory.MESICache")
cache.addParams({
    "verbose":              "5",
    "clock":                "1GHz",
    "lineSize":             "64",
    "lineCount":            "16",
    "associativity":        "full",
    "hitLatency":           "3",
    "inclusive":             "true",
    "replacement":          "LRU",
    "noncacheable_regions": "[]",
})
```

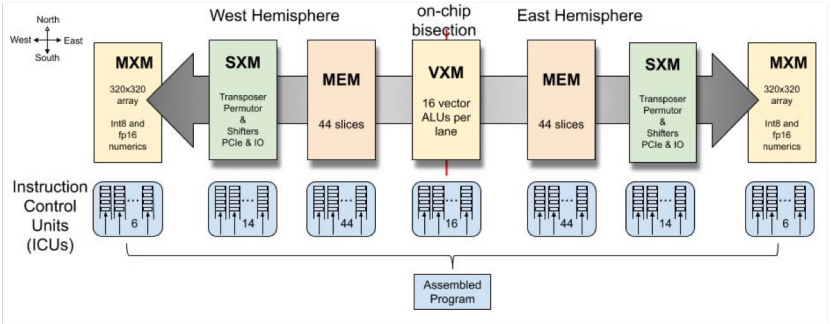
# Bigger Challenge

- We see that the models have significant knowledge of ”vanilla” architecture parameters
- Many interesting architectures are less well defined
- Dataflow architectures
  - Significant departure from traditional organization
  - Still well documented and defined
- Current SST dataflow-esque element: Llyr
  - Maps a computational graph to a collection of PEs
  - Don’t want the model to “cheat”
- Different dataflow target: Groq

# Groq Background

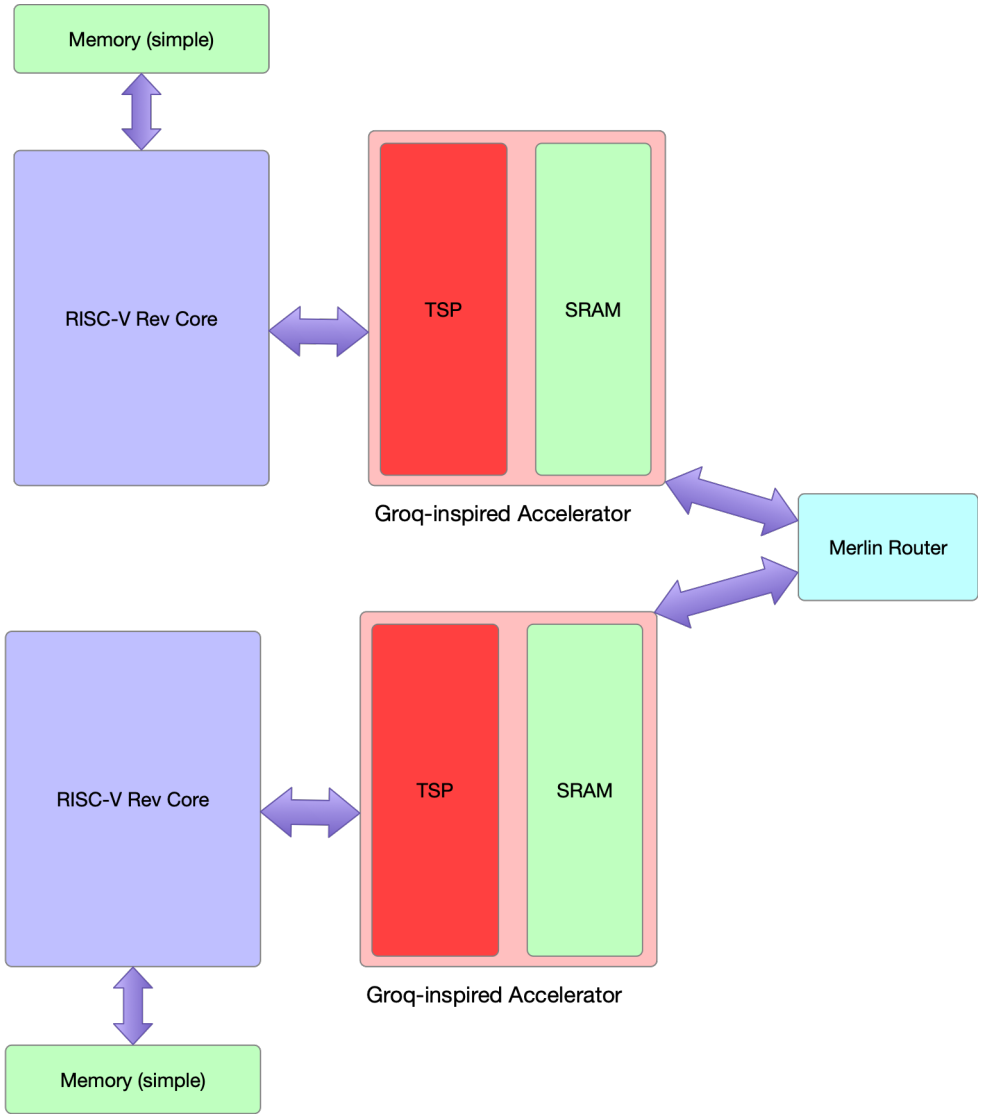
- Architecture centered around the Tensor Streaming Processor (TSP)
- Computational elements are arranged spatially by function
  - Take advantage of dataflow locality as tensors flow past
  - Elements have specific functions
    - Mix of PEs and pipeline stages
- Deterministic latency and static scheduling



# Model Target

- RISC-V Core
- Connect to Groq-esque accelerator via Co-Proc
- Scale to multiple accelerators via merlin for on-chip network



# Clawd-SST Results

- Clear enumeration of features implemented:
  - Deterministic architecture
  - A representation of the MXM | SXM | MEM×44 | VXM | MEM×44 | SXM | MXM die layout
  - Functional slicing with 320 lanes
  - Flat 220 MiB software-managed SRAM with no cache
  - Staggered execution with the the T = N + dfunc + δ(j,i) latency equation
  - Merlin utilized to connect blocks, latency set to 0 to remove contention
- Vs. Not Implemented:
  - Timing delays assumed are marked [est] in the model. This covers blocks with undocumented latency
  - The model does real byte math for everything except the MXM dot products, TanH/Exp/RSqrt, Rotate, Transpose
    - Timing is supported here, so performance is accurate
  - DMAs in the model are async, which (sort of) breaks the determinism of the model

# Clawd-SST Results

- RISC-V Opcodes
  - Not explicitly requested, but the model generated RISC-V opcode encodings to control the co-processor
- Tests
  - Smoke, Vector Add, scatter/gather, timing determinism, exchange between multiple accelerators
- Expected results and stats
- Configs include a 320 Rev Core model
  - Recommended to run with –num-threads option

```
// Two MEM Reads directed inward toward the VXM, an Add, then a Write.
#define TSP_R(f7, f3, rd, rs1, rs2) /* .insn r 0x0B, f3, f7, rd, rs1, rs2
*/

uint64_t d_s1 = (0)  | (320ULL << 8);              // stream 1 East,
VL=320
uint64_t d_s2 = (4)  | (320ULL << 8);              // stream 4 East,
VL=320
uint64_t d_add= (0)  | (320ULL << 8) | (4ULL<<18); // dst=s0, rhs=s4

asm volatile(".insn r 0x0B, 1, 0, x0, %0, %1" :: "r"(addr_X),
"r"(d_s1)); // MEM Read
asm volatile(".insn r 0x0B, 1, 0, x0, %0, %1" :: "r"(addr_Y),
"r"(d_s2)); // MEM Read
asm volatile(".insn r 0x0B, 2, 1, x0, x0, %0"  ::
"r"(d_add));            // VXM binary
asm volatile(".insn r 0x0B, 1, 1, x0, %0, %1" :: "r"(addr_Z),
"r"(d_s1)); // MEM Write
```

Simple Z = X + Y example

# Additional SST Outputs

- Multi-Core RISC-V
  - Multiple RISC-V Cores connected with a coherent interface
- UCIe Interface
  - UCIe Spec utilized
  - Specify spec features to implement
- Posit Computation
  - Create extension to Rev to allow Posit computation on RISC-V

# Where do we go from here?

- Creating simulation components is no longer the bottleneck
  - An exciting future!
  - Interfaces, interoperability, etc. no longer a barrier
  - Opportunities for better intercept with new products
- Focus on robust simulation libraries as a foundation
  - Manual development of simulation libraries will allow AI users to better drive the tools, providing better outcomes
- Tests are more important than ever
  - Ai can help with that too…
- We get to focus on *using* the simulators

# Next Steps (personally)


ZOOX

Questions?