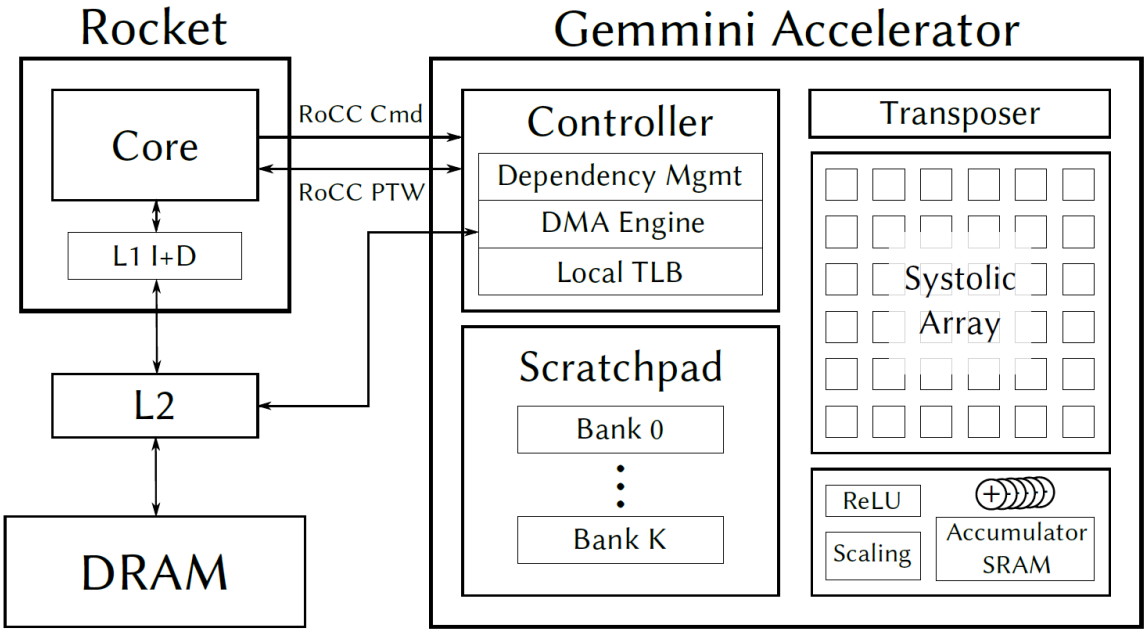
Gemmini

The Gemmini project is developing a full-system, full-stack DNN hardware exploration and evaluation platform.
Gemmini enables architects to make useful insights into how different components of the system and software stack (outside of just the accelerator itself) interact to affect overall DNN performance.

Gemmini is part of the Chipyard ecosystem, and was developed using the Chisel hardware description language.

This document is intended to provide information for beginners wanting to try out Gemmini, as well as more advanced in-depth information for those who might want to start hacking on Gemmini's source code.



Quick Start

We provide here a quick guide to installing Gemmini's dependencies (Chipyard and Spike), building Gemmini hardware and software, and then running that software on our hardware simulators.

Dependencies

Before beginning, install the Chipyard dependencies that are described here.

```
git config --global url.https://mirror.ghproxy.com/https://.insteadof https://

git config --global --unset url.https://mirror.ghproxy.com/https://.insteadof
```

Installing Chipyard and Spike

Run these steps to install Chipyard and Spike (make sure to checkout the correct Chipyard and Spike commits as shown below):

```
git clone https://github.com/ucb-bar/chipyard.git
cd chipyard
git checkout 117624d8eea27bafd613eec09e9b9b3e31239e08
./scripts/init-submodules-no-riscv-tools.sh
./scripts/build-toolchains.sh esp-tools

source env.sh

cd generators/gemmini
git fetch && git checkout v0.6.4
git submodule update

cd -
cd toolchains/esp-tools/riscv-isa-sim/build
git fetch && git checkout 090e82c473fd28b4eb2011ffcd771ead6076faab
make && make install
```

Setting Up Gemmini

Run the steps below to set up Gemmini configuration files, symlinks, and subdirectories:

```
cd chipyard/generators/gemmini
./scripts/setup-paths.sh
```

Building Gemmini Software

Run the steps below to compile Gemmini programs, including large DNN models like ResNet50, and small matrix-multiplication tests.

```
cd chipyard/generators/gemmini/software/gemmini-rocc-tests
./build.sh
```

Afterwards, you'll find RISC-V binaries in `build/`, for "baremetal" environments, Linux environments, and "proxy-kernel" environments.

Linux binaries are meant to be executed on SoCs that run Linux.
These binaries are dynamically linked, and support all syscalls.
Typically, our users run them on FireSim simulators.

Baremetal binaries are meant to be run in an environment without any operating system available.
They lack support for most syscalls, and do not support virtual memory either.
Our users typically run them on cycle-accurate simulators like Verilator or VCS.

"Proxy-kernel" binaries are meant to be run on a stripped down version of Linux, called the "RISC-V Proxy Kernel."
These binaries support virtual memory, and are typically run on cycle-accurate simulators like Verilator.

Warning: Proxy-kernel binaries have limited heap space, so some Gemmini programs that work correctly in baremetal or Linux environments may fail on the proxy-kernel.

Building Gemmini Hardware and Cycle-Accurate Simulators

Run the instructions below to build a cycle-accurate Gemmini simulator using Verilator.

```
cd chipyard/generators/gemmini
./scripts/build-verilator.sh

# Or, if you want a simulator that can generate waveforms, run this:
# ./scripts/build-verilator.sh --debug
```

After running this, in addition to the cycle-accurate simulator, you will be able to find the Verilog description of your SoC in `generated-src/`.

Building Gemmini Functional Simulators

Run the instructions below to build a functional ISA simulator for Gemmini (called "Spike").

```
cd chipyard/generators/gemmini
./scripts/build-spike.sh
```

Spike typically runs *much* faster than cycle-accurate simulators like Verilator or VCS.
However, Spike can only verify functional correctness; it cannot give accurate performance metrics or profiling information.

Run Simulators

Run the instructions below to run the Gemmini RISCV binaries that we built previously, using the simulators that we built above:

```
cd chipyard/generators/gemmini

# Run a large DNN workload in the functional simulator
./scripts/run-spike.sh resnet50

# Run a smaller workload in baremetal mode, on a cycle-accurate simulator
./scripts/run-verilator.sh template

# Run a smaller workload with the proxy-kernel, on a cycle accurate simulator
./scripts/run-verilator.sh --pk template

# Or, if you want to generate waveforms in `waveforms/`:
# ./scripts/run-verilator.sh --pk --debug template
```

Next steps

Check out our IISWC 2021 tutorial to learn how to:

- build different types of diverse accelerators using Gemmini.
- add custom datatypes to Gemmini.
- write your own Gemmini programs.
- profile your workloads using Gemmini's performance counters.

Also, consider learning about FireSim, a platform for FPGA-accelerated cycle-accurate simulation. We use FireSim to run end-to-end DNN workloads that would take too long to run on Verilator/VCS.
FireSim also allows users to check that their Gemmini hardware/software will work when running on a Linux environment.

Or, continue reading the rest of this document for descriptions of Gemmini's architecture, ISA, and configuration parameters.

Architecture

Gemmini is implemented as a RoCC accelerator with non-standard RISC-V custom instructions. The Gemmini unit uses the RoCC port of a Rocket or BOOM *tile*, and by default connects to the memory system through the System Bus (i.e., directly to the L2 cache).

At the heart of the accelerator lies a systolic array which performs matrix multiplications. By default, the matrix multiplication support both *output-stationary* and *weight-stationary* dataflows, which programmers can pick between at runtime.
However, the dataflow can also be hardened at elaboration time.

The systolic array's inputs and outputs are stored in an explicity managed scratchpad, made up of banked SRAMs.
A DMA engine facilitates the transfer of data between main memory (which is visible to the host CPU) and the scratchpad.

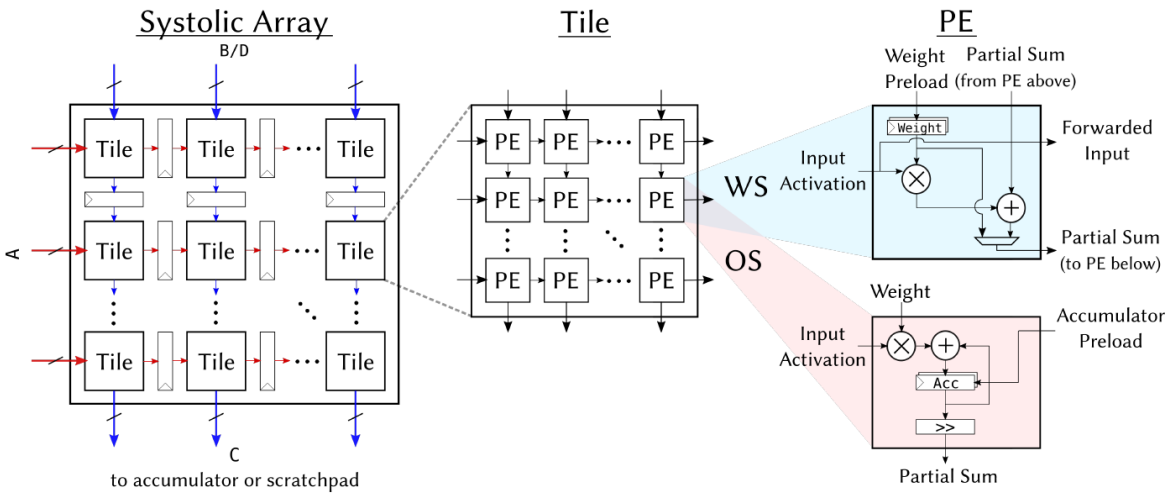
Because weight-stationary dataflows require an accumulator outside the systolic array, we add a final SRAM bank, equipped with adder units, which can be conceptually considered an extension of the scratchpad memory space. The systolic array can store results to any address in the accumulator, and can also read new inputs from any address in the accumulator. The DMA engine can also tranfer data directly between the accumulator and main memory, which is often necessary to load in biases.

Gemmini also includes peripheral circuitry to optionally apply activation functions such as ReLU or ReLU6, scale results down by powers-of-2 to support quantized workloads, or to transpose matrices before feeding them into the systolic array to support the output-stationary dataflow.

Generator Parameters

Major parameters of interest include:

- Systolic array dimensions (`tileRows`, `tileColumns`, `meshRows`, `meshColumns`): The systolic array is composed of a 2-level hierarchy, in which each tile is fully combinational, while a mesh of tiles has pipeline registers between each tile.



- Dataflow parameters (`dataflow`): Determine whether the systolic array in Gemmini is output-stationary or weight-stationary, or whether it supports both dataflows so that programmers may choose between them at runtime.
- Scratchpad and accumulator memory parameters (`sp_banks`, `sp_capacity`, `acc_capacity`): Determine the properties of the Gemmini scratchpad memory: overall capacity of the scratchpad or accumulators (in KiB), and the number of banks the scratchpad is divided into.
- Type parameters (`inputType`, `outputType`, `accType`):
  Determine the data-types flowing through different parts of a Gemmini accelerator.
  For example, `inputType` may be an 8-bit fixed-point number, while `accType`, which determines the type of partial accumulations in a matrix multiplication, may be a 32-bit integer.
  `outputType` only determines the type of the data passed between two processing elements (PEs); for example, an 8-bit multiplication may produce a 16-bit result which must be shared between PEs in a systolic array.
  - Examples of possible datatypes are:
    - `SInt(8.W)` for a signed 8-bit integer
    - `UInt(32.W)` for an unsigned 32-bit integer
    - `Float(8, 24)` for a single-precision IEEE floating point number
  - If your datatype is a floating-point number, then you might also want to change the `pe_latency` parameter, which specifies how many shift registers to add inside the PEs. This might be necessary if your datatype cannot complete a multiply-accumulate operation within a single cycle.
- Access-execute queue parameters (`ld_queue_length`, `st_queue_length`, `ex_queue_length`, `rob_entries`): To implement access-execute decoupling, a Gemmini accelerator has a load instruction queue, a store instruction queue, and an execute instruction queue. The relative sizes of these queue determine the level of access-execute decoupling. Gemmini also implements a reorder buffer (ROB) - the number of entries in the ROB determines possible dependency management limitations.
- DMA parameters (`dma_maxbytes`, `dma_buswidth`, `mem_pipeline`): Gemmini implements a DMA to move data from main memory to the Gemmini scratchpad, and from the Gemmini accumulators to main memory. The size of these DMA transactions is determined by the DMA parameters. These DMA parameters are tightly coupled with Rocket Chip SoC system parameters: in particular `dma_buswidth` is associated with the `SystemBusKey` `beatBytes` parameter, and `dma_maxbytes` is associated with `cacheblockbytes` Rocket Chip parameters.

There are also optional features, which can be either enabled or left out of Gemmini at elaboration-time.
For example:

- Scaling during "move-in" operations (`mvin_scale_args`, `mvin_scale_acc_args`):
  When data is being moved in from DRAM or main memory into Gemmini's local scratchpad memory, it can optionally be multiplied by a scaling factor.
  These parameters specify what the datatype of the scaling factor is, and how the scaling is actually done.
  If these are set to `None`, then this optional feature will be disabled at elaboration time.
  If both the scratchpad inputs are accumulator inputs are to be scaled in the same say, then the `mvin_scale_shared` parameter can be set to `true` so that the multipliers and functional units are shared.

Major Components

This subsection is aimed towards those who wish to start hacking on Gemmini's RTL.
Here, we briefly describe Gemmini's main hardware components, and how they fit together.
If you have no interest in changing Gemmini's hardware (besides just changing configuration parameters), then feel free to skip this section.

Decoupled Access/Execute

Gemmini is a decoupled access/execute architecture, which means that "memory-access" and "execute" instructions happen concurrently, in different regions of the hardware.
We divide the hardware broadly into three "controllers": one for "execute" instructions, another for "load" instructions, and a third for "store" instructions.
Each of these controllers consume direct ISA commands from the programmer, decode this commands, and execute them, while sharing access to the scratchpad and acccumulator SRAMs.

- `ExecuteController`: This module is responsible for executing "execute"-type ISA commands, such as matrix multiplications.
  It includes a systolic array for dot-products, and a transposer.
- `LoadController`: This module is responsible for all instructions that move data from main memory into Gemmini's private scratchpad or accumulator.
- `StoreController`: This module is responsible for all instructions that move data from Gemmini's private SRAMs into main memory.
  This module is also responsible for "max-pooling" instructions, because Gemmini performs pooling when moving unpooled data from the private SRAMs into main memory.

Scratchpad and Accumulator

Gemmini stores inputs and outputs for the systolic array in a set of private SRAMs, which we call the "scratchpad" and the "accumulator".
Typically, inputs are stored in the scratchpad, while partial sums and final results are stored in the the accumulator.

The scratchpad and accumulator are both instantiated within `Scratchpad.scala`.
The scratchpad banks are implemented by the `ScratchpadBank` module, and the accumulator banks are implemented by the `AccumulatorMem` module.

Each row of the scratchpad and accumulator SRAMs is `DIM` "elements" wide, where `DIM` is the number of PEs along the width of the systolic array.
Each "element" represents a single scalar value that Gemmini operates upon.

Each "element" in the scratchpad is of type `inputType` (which, in the default config, is an 8-bit integer).
Each "element" in the acccumulator is of type `accType` (which, in the default config, is a 32-bit integer).

So, for example, in the default config, which has a 16x16 systolic array, the scratchpad banks have a row-width of `16*bits(inputType)=128` bits, and the accumulatorr banks have a row-width of `16*bits(accType)=512` bits.

Both inputs and outputs to the scratchpad must be of type `inputType`.

Both inputs and outputs from the accumulator can be either of type `accType` *or* `inputType`.
If `inputType` values are input to the accumulator, they will be cast up to `accType`.
If `inputType` values are output from the accumulator, they will first be "scaled" down to be of type `inputType`.
The exact "scaling" function can be configured as the as the user wishes, but in the default config, the scaling function is a simple multiplication by a `float32` value that casts an `int32` down to an `int8`.

The scratchpad banks are very simple, comprising little more than an SRAM and a queue.

The accumulator banks are a bit more complex: in addition to the underlying SRAM, they also include a set of adders to support in-place accumulations.
In addition, they have a set of "scalers" (described above), and activation function units.
The scaling and activation functions are applied when the programmer wishes to transform `accType` values down to `inputType` values while reading data out of the accumulator.
This is typically done to transform the partial-sum outputs of one layer into the low-bitwidth quantized inputs of the next layer.

Systolic Array and Transposer

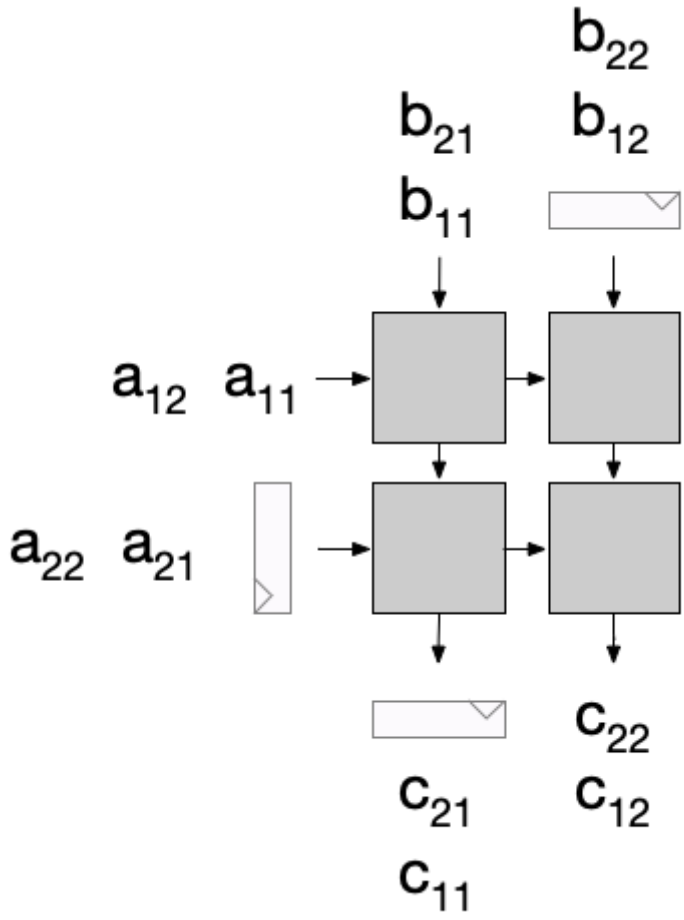
`MeshWithDelays`, which is instantiated within the `ExecuteController`, contains the systolic array (`Mesh`), a transposer (`Transposer`), and a set of delay registers which shift the inputs to the systolic array.
The `MeshWithDelays` module takes in three matrices one row at a time per cycle (`A`, `B`, and `D`), and outputs the result `C = A * B + D` one row at a time per cycle.

In the weight-stationary mode, the `B` values are "preloaded" into the systolic array, and `A` and `D` values are fed through.
In the output-stationary mode, the `D` values are "preloaded" into the systolic array, and `A` and `B` values are fed through.

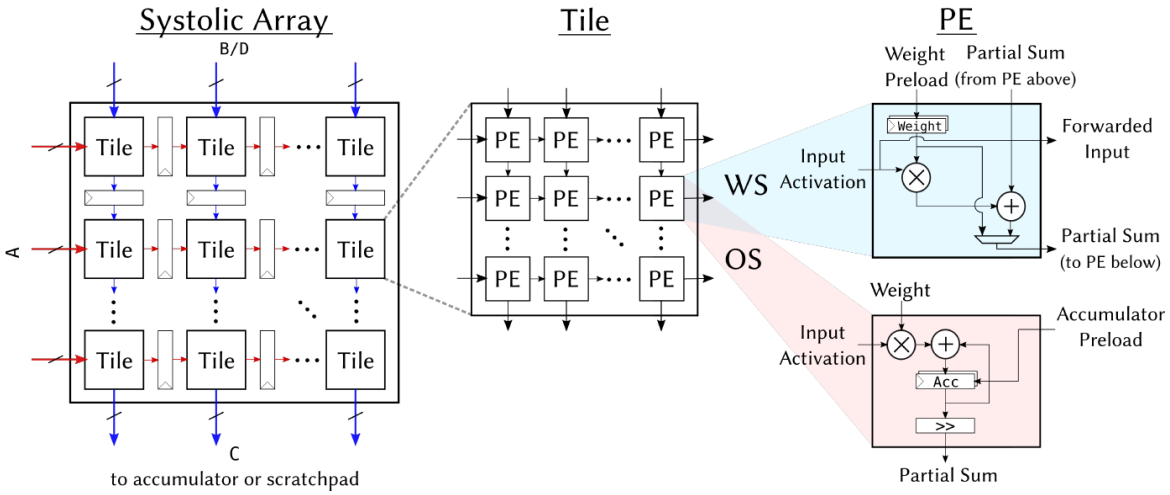
`A`, `B`, and `D` are all of type `inputType`, while `C` is of type `outputType`.
If the programmer wishes to write `C` into the scratchpad, then `C` is cast down to `inputType`.
However, if the programmer instead wishes to write `C` into the scratchpad, then `C` is cast up to `accType`.

Note that in the weight-stationary mode, an `inputType` D usually has insufficient bitwidth to accurately represent partial sums.
Therefore, in the weight-stationary mode, `D` is usually just the 0-matrix, while the `accType` accumulator SRAMs are used to accumulate partial sum outputs of the systolic array instead.

The inputs (`A`, `B`, and `D`) must be delayed with shift-registers so that each input from one matrix reaches the correct PE at exactly the right time to be multiplied-and-accumulated with the correct input from another matrix.
The diagram below shows an example of a 2x2 output-stationary matmul (ignoring `D`), with the appropriate delay registers at the inputs and outputs of the systolic array:

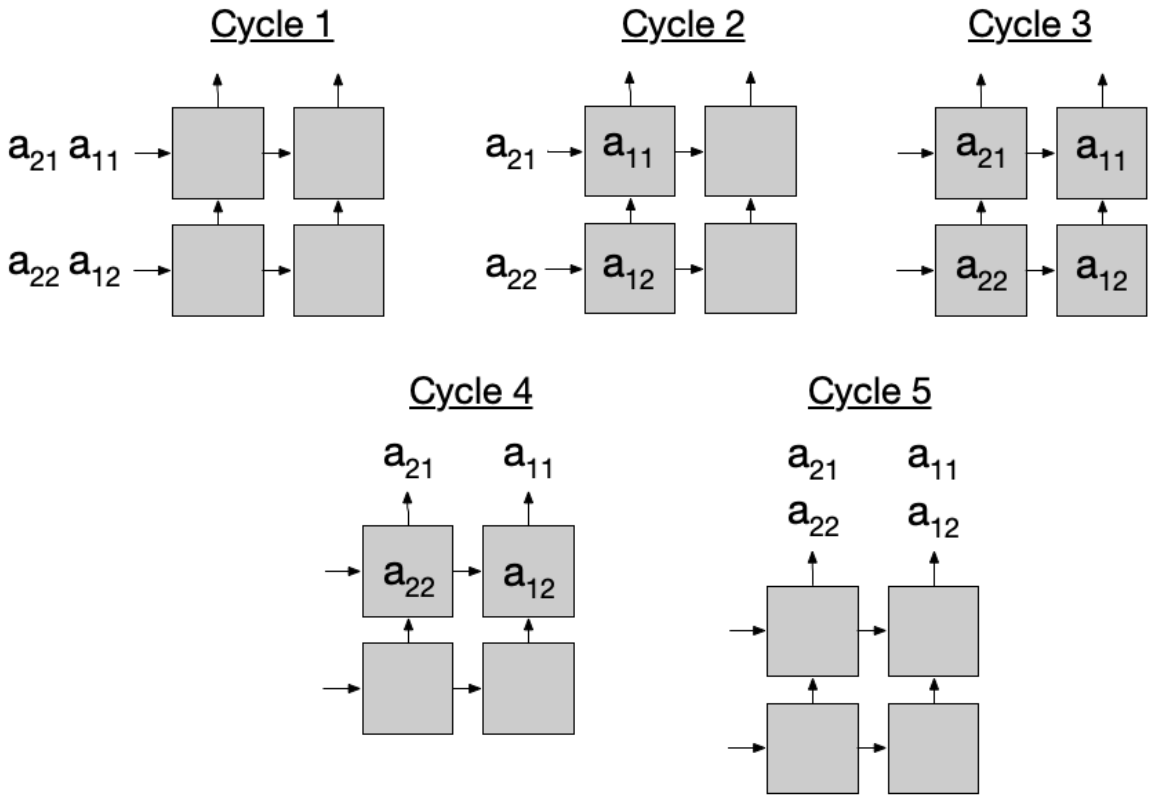


The systolic array itself (implemented in `Mesh.scala`), is composed of a two-tier hierarchy of `Tiles` and `PEs`.
The `Mesh` is composed of a set of `Tiles`, separated by pipeline registers.
Every `Tile` is composed of a combinational set of `PEs`, where each PE performs a single matmul operation, with either the weight-stationary, or output-stationary dataflow.



The `MeshWithDelays` module also includes a number of counters and configuration registers. `MeshWithDelays` assumes that every matmul operation will be exactly of size `DIM x DIM`, where `DIM` is the number of PEs across the width of the systolic array itself (16 in the default config). These counters count up to `DIM`, and then update the configuration registers from the inputs to `MeshWithDelays`.
These configuration registers control which of `A` and `B` are to be transposed before being fed into the systolic array.
They also control whether the preloaded values in the systolic array are to be maintained for the next matmul, or whether they are to be overwritten and replaced.

The transposer itself is implemented as a very simple systolic array, which transports inputs from left-to-right for `DIM` cycles, and then down-to-up for another `DIM` cycles.
This is illustrated in the diagram below:



Note that for output-stationary matmuls, the transposer is used even when the programmer does not request a transposition.
This is because the systolic array expects inputs from the same row of `A` to enter the same PE in the output-stationary mode, but all values in a single row of `A` are stored within the same scratchpad SRAM row.
Therefore, the rows have to be transposed after being read out of the scratchpad, so that elements on the same row can be fed into the same PE one-after-another, rather than being fed into adjacent PEs.

DMA

Gemmini includes two DMAs, one for reading data from main memory into Gemmini's private SRAMs, and another for moving data from Gemmini's private SRAMs into main memory.
Both these modules are implemented in `DMA.scala`.

Both DMAs operate on virtual addresses, and share access to a TLB to translate these into physical main memory addresses.
If the TLB misses, it transparently falls back to a PTW that is shared with Gemmini's host CPU.

After physical addresses are obtained from Gemmini's private TLB, the DMAs break large memory requests up into smaller TileLink read and write requests.
To satisfy the TileLink protocol, each memory request must be aligned to the number of bytes requested from/to main memory, and the size of each memory request (in bytes) must be a power of 2.
The DMAs generally attempt to minimize the number of TileLink requests as much as possible, even if this requires reading a larger total amount of data from main memory.
Empirically, we have found that an excessive number TileLink requests can limit performance more than reading a small amount of extra data.

The DMAWriter, which writes data from private SRAMs to main memory, also includes a set of `>` comparators that are used for max-pooling data during a memory-write operation.

ROB

Due to Gemmini's decoupled access-execute architecture, instructions in the `LoadController`, `StoreController`, and `ExecuteController` may operate concurrently and out-of-order with respect to instructions in other controllers.
Gemmini includes an ROB which is meant to detect hazards between instructions in different controllers.
The instructions in the ROB are only issued to their respective controllers once they have no dependencies on instructions in other controllers.

Note that instructions that are destined for the same controller are issued in-order.
The ROB does not check hazards between instructions within the same controller, because each controller is obligated to handle it's own dependencies and hazards internally, assuming that it receives it's own instructions in program-order.

Matmul and Conv Loop Unrollers

Gemmini's systolic array can only operate on matmuls that are up to `DIMxDIM` elements large. When performing matmuls and convolutions that are larger than this, programmers must tile their matmuls into a sequence of smaller `DIMxDIM` matmuls.

However, tiling these operations efficiently can be difficult for programmers, due to CPU and loop overheads, and the difficulty of unrolling and pipelining software loops.

To alleviate this difficulty, Gemmini's ISA includes high-level CISC-type instructions, which automatically tile and unroll large matmuls and convolutions.
These are implemented in the `LoopMatmul` and `LoopConv` modules.

These modules are implemented as FSMs, which double-buffer matmul/conv tiles to maximize performance, and which monitor the proportion of load/store/execute instructions in the ROB to maximize overlap between memory accesses and dot-product computations.
For example, if the ROB is dominated by matmul instructions, without leaving any slots for incoming load instructions, then the FSMs will pause the issuing of matmul instructions to allow more space for concurrent load instructions in Gemmini's datapath.

Software

The Gemmini ISA is specified in the `ISA` section below.
The ISA includes configuration instructions, data movement instructions (from main memory to/from Gemmini's private memory), and matrix multiplication execution instructions.

Since Gemmini instructions are not exposed through the GNU binutils assembler, several C macros are provided in order to construct the instruction encodings to call these instructions.

The Gemmini generator includes a C library which wraps the calls to the custom Gemmini instructions into common DNN operators like matmuls, convolutions (with or without pooling), matrix-additions, etc.
The `software` directory of the generator includes the aforementioned library and macros, as well as baremetal tests, and some FireMarshal workloads to run the tests in a Linux environment. In particular, the C library can be found in the `software/gemmini-rocc-tests/include/gemmini.h` file.

The Gemmini generator generates a C header file based on the generator parameters. This header files gets compiled together with the C library to tune library performance. The generated header file can be found under `software/gemmini-rocc-tests/include/gemmini_params.h`

Gemmini can also be used to run ONNX-specified neural-networks through a port of Microsoft's ONNX-Runtime framework. The port is included as the onnxruntime-riscv repository submoduled in the `software` directory.
To start using ONNX-Runtime, run `git submodule update --init --recursive software/onnxruntime-riscv`, and read the documentation here.

Build and Run Gemmini Tests

To build the Gemmini tests:

```
cd software/gemmini-rocc-tests/
./build.sh
```

Afterwards, the test binaries will be found in `software/gemmini-rocc-tests/build`.
Binaries whose names end in `-baremetal` are meant to be run in a bare-metal environment, while binaries whose names end in `-linux` are meant to run in a Linux environment.
You can run the tests either on a cycle-accurate RTL simulator, or on a (much faster) functional ISA simulator called Spike.

We use a special fork of Spike, found here, which has support for Gemmini instructions.
(You can find the required commit hash in `SPIKE.hash`).
If you are using Chipyard, you can easily build Spike by running `./scripts/build-toolchains.sh esp-tools` from Chipyard's root directory.
Then, to run the `mvin_mvout` test, which simply moves a matrix into Gemmini's scratchpad before moving it back out into main memory, run the following commands:

```
cd build/bareMetalC
spike --extension=gemmini mvin_mvout-baremetal
```

Writing Your Own Gemmini Tests

`software/gemmini-rocc-tests/bareMetalC/template.c` is a template Gemmini test that you can base your own Gemmini tests off of. To write your own Gemmini test, run:

```
cd software/gemmini-rocc-tests/
cp bareMetalC/template.c bareMetalC/my_test.c
```

Then, add `my_test` to the `tests` list at the top of `bareMetalC/Makefile`. Afterwards, running `./build.sh` will install `my_test-baremetal` in `build/bareMetalC`.

DNN Tests

Example DNNs, such as ResNet50, can be found in `software/gemmini-rocc-tests/imagenet` and `software/gemmini-rocc-tests/mlps`.
These tests are built and run the same way as the other tests described above, but they typically take too long to run in a software simulator like VCS or Verilator.
We recommend instead that you run these tests through Firesim, an FPGA-accelerated simulation platform, which will reduce your runtime from days to minutes.

Note that the DNN tests rely upon our C library of common DNN operators (found in `gemmini.h`). They call very few direct Gemmini ISA instructions, and mostly call the wrappers around them found in the C library.

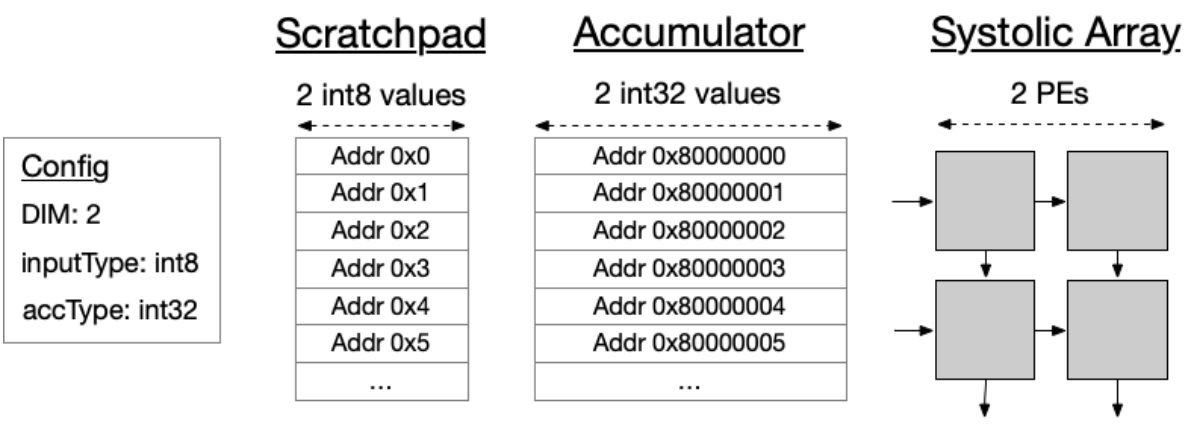
Memory Addressing Scheme

Gemmini's private memory is "row-addressed", where each row is `DIM` elements wide, where `DIM` is the number of PEs across the width of the systolic array (16 in the default config).
These elements will be of type `inputType` in the scratchpad, and of type `accType` in the accumulator.

Every private Gemmini memory address is 32 bits long.
The three most signficant bits are reserved, and have special meanings:

- Bit 31 (the MSB) is 0 if we are addressing the scratchpad, and 1 if we are addressing the accumulator.
- Bit 30 is ignored if we are addressing the scratchpad, or if we are reading from the accumulator. If, instead, we are writing to the accumulator, then bit 30 is 0 if we want to overwrite the data at that address, and 1 if we want to accumulate on top of the data already at that address.
- Bit 29 is ignored if we are addressing the scratchpad, or if we are writing to the accumulator. If, instead, we are reading from the accumulator, then bit 29 is 0 if we want to read scaled-down `inputType` data from the accumulator, and 1 if we want to read `accType` data from the accumulator.
    - If bit 29 is 1 for an accumulator read address, then we do not apply activation functions or scaling to the output of the accumulator.

The memory addressing scheme for a Gemmini config with a 2x2 systolic array is illustrated below:



Gemmini accesses main memory addresses (which are also visible to the CPU) through their software-visible virtual addresses.
Physical translation addresses are handled by Gemmini, transparently to the programmer.

ISA

This section describes Gemmini's assembly-level ISA which is made up of custom RISC-V instructions.

Data Movement

`mvin` Move Data From Main Memory to Scratchpad

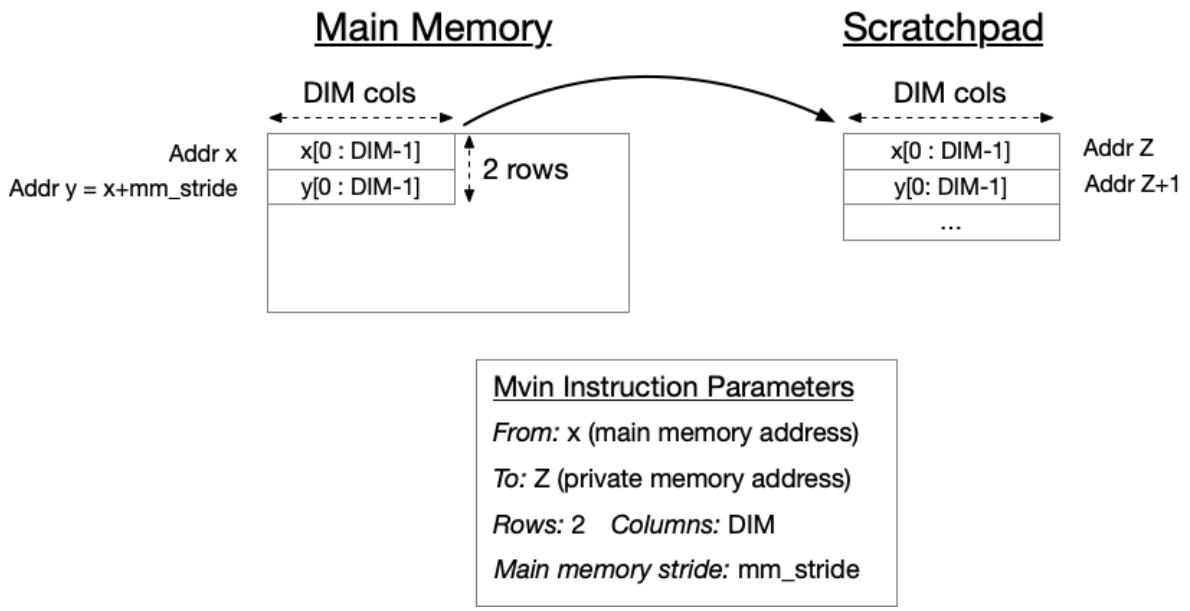
Format: `mvin rs1, rs2`

- `rs1` = virtual DRAM address (byte addressed) to load into scratchpad
- `rs2[31:0]` = local scratchpad or accumulator address
- `rs2[47:32]` = number of columns to load in
- `rs2[63:48]` = number of rows to load in. Must be less than or equal to `DIM`.
- `funct` = 2

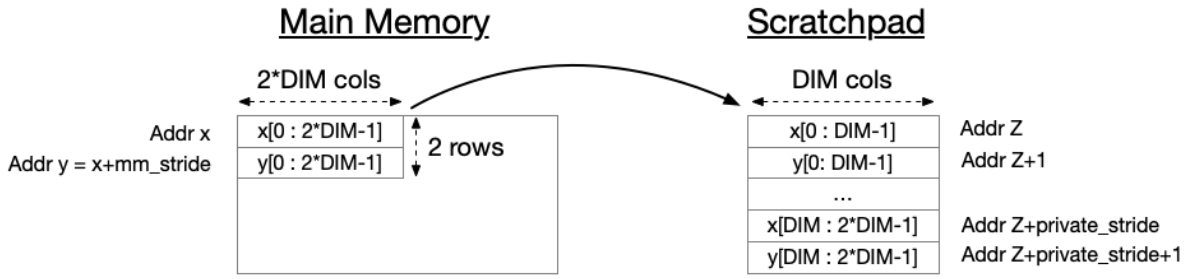
Action: Scratchpad[rs2] <= DRAM[Translate[rs1]]

- Loads a 2D matrix from main memory into Gemmini's private memory.
- Load is sequential from the rs1/rs2 base addresses.
- Main memory stride must be set by the `config_mvin` command.
- If the number of columns we load in are greater than `DIM`, then multiple submatrices will be moved in.
  The private-memory stride between these submatrices is set by the `config_mvin` command.

The figure below illustrates how the `mvin` command works:



In addition, the figure below illustrates the special case where the number of columns moved-in is greater than `DIM`:



Mvin Instruction Parameters

From: x (main memory address)

To: Z (private memory address)

Rows: 2 *Columns:* 2*DIM

Main memory stride: mm_stride

Private memory stride: private_stride

Notes:

- There are actually **three** `mvin` instructions in Gemmini: `mvin`, `mvin2`, and `mvin3`. `mvin2` and `mvin3` are completely identical to `mvin`, except that they have their own independent set of configuration registers.
  When calling `config_mvin` (described below), the programmer can choose which `mvin` instruction they want to configure.
- The reason we have three `mvin` instructions is so that the programmer can overlap loads for A, B, and D matrices (for a `A*B+D` matmul), where A, B, and D may all have different main-memory-strides.

`mvout` Move Data from Scratchpad to L2/DRAM

Format: `mvout rs1, rs2`

- `rs1` = virtual DRAM address (byte addressed) to write to from scratchpad
- `rs2[31:0]` = local scratchpad address
- `rs2[47:32]` = number of columns to store
- `rs2[63:48]` = number of rows to store
- `funct` = 3

Action: DRAM[Translate[rs1]] <= Scratchpad[rs2]

- Stores a 2D matrix from the scratchpad to main-memory
- Store is sequential from the rs1/rs2 base addresses. Stride must be set by the `config_mvout` command

Configuration

`config_ex` configures the Execute pipeline

Format: `config_ex rs1 rs2`

- `rs1[1:0]` must be `00`
- `rs1[2]` determines if output (0) or weight (1) stationary
- `rs1[4:3]` = activation function: either relu (1), relu6 (2), or no activation function (0)
- `rs1[8]` = should A be transposed?

- `rs1[9]` = should B be transposed?
- `rs1[31:16]` = the stride (in scratchpad addresses) by which the rows of A are fed into the systolic array.
  "A" in this context refers to the left-hand matrix A in the matmul represented by A * B = C.
  If this stride is 1, then we feed consecutive rows in the scratchpad, starting from the starting address of A, into the systolic array as the A matrix.
  If the stride is 2, then we feed every other row into the systolic array instead.
- `rs1[63:32]` = the scalar value by which we scale the `accType` output of the accumulator down to `inputType` values when reading from the accumulator.
  - In the default config, `rs1[63:32]` is of type `float32`
- `rs2[31:0]` = the number of bits by which the accumulated result of a matmul is right-shifted when leaving the systolic array
  - This parameter is only relevant in output-stationary mode, when partial sums must be accumulated within the systolic array itself, and scaled-down when leaving the systolic array and being written into the scratchpad.
- `rs2[63:32]` = the number of bits by which 6 should be left-shifted before applying relu6
  - This parameter is ignored if the relu6 activation function is not being used.
- `funct` = 0

Action: mode <= rs1(2); shift <= rs2; A_stride <= rs1[31:16]

Notes:

- As of now, certain combinations of transpose options cannot be performed unless the right dataflow is chosen.
  This limitation may be lifted in the future.

Dataflow	Transpose A	Transpose B	Permitted?
OS	No	No	Yes
OS	No	Yes	No
OS	Yes	No	Yes
OS	Yes	Yes	Yes
WS	No	No	Yes
WS	No	Yes	Yes
WS	Yes	No	Yes
WS	Yes	Yes	No

`config_mvin` configures the Load pipeline

Format: `config_mvin rs1 rs2`

- `rs1[1:0]` must be `01`
- `rs1[2]` is 0 if `mvin`s to the accumulator are of type `accType`, and 1 if they are `inputType`
- `rs1[4:3]` is 0 if the stride is being set for `mvin`, 1 if the stride is being set for `mvin2`, and 2 if the stride is being set for `mvin3`

- `rs1[63:32]` is the "scale" by which to multiply data as it's being moved in to the scratchpad. This is ignored if Gemmini isn't configured to have the ability to scale values during `mvin`s.
- `rs2` = the stride in bytes
- `funct` = 0

Action: stride <= rs2; scale <= rs1[63:32]

`config_mvout` configures the Store pipeline

Format: `config_mvout rs1 rs2`

- `rs1[1:0]` must be `10`
- `rs2` = the stride in bytes
- `funct` = 0

During `mvout` operations, Gemmini can also perform max-pooling.
This is an experimental feature, and is subject to change.
This feature assumes that data is stored in the scratchpad or accumulator in NHWC format.
The parameters controlling this feature are:

- `rs1[5:4]` = max-pooling stride. If this is 0, then max-pooling is deactivated.
- `rs1[7:6]` = max-pooling window size
- `rs1[9:8]` = upper zero-padding
- `rs1[11:10]` = left zero-padding
- `rs1[31:24]` = output dimension of image after pooling
- `rs1[39:32]` = number of pooled rows to output
- `rs1[47:40]` = number of pooled columns to output
- `rs1[55:48]` = number of unpooled rows to pool
- `rs1[63:56]` = number of unpooled columns to pool

Action: stride <= rs2; max-pooling parameters <= rs1

`flush` flushes the TLB

Format: `flush rs1`

- `rs1` = If `rs1[0]` is 1, then the current TLB request is skipped (if it has hit a page-fault and is waiting for an interrupt).
  Otherwise, the current TLB request is repeated.

Notes:

- This instruction executes *as soon as it is received* without waiting for other instructions which may be queued up.
  It is the programmer's responsibility to insert fences if necessary.

Core Matmul Sequences

Every single matrix multiply operation is a combination of `matmul.preload` and `matmul.compute` (due to the length of a single instruction, it was split into two instructions).
`matmul.preload` should precede the `matmul.compute`.

Example:

```
//// OS matmul example ////
// rs1 = InputD
// rs2 = OutputC
// rs3 = InputA
// rs4 = InputB
// matmul InputA InputB OutputC InputD
1. matmul.preload $rs1 $rs2
2. matmul.compute $rs3 $rs4
```

Action: Scratchpad[rs2] <= Scratchpad[rs3] * Scratchpad[rs4] + Scratchpad[rs1]

Notes on addressing:

- For B or D, the address can be replaced with all high bits to input a 0 matrix instead.
- For A, the address can be replaced with all high bits to input a matrix with undefined garbage data instead.

Preloading

Format: `matmul.preload rs1, rs2`

- `rs1[31:0]` = local scratchpad address of D matrix (when output-stationary), or B matrix (when weight-stationary)
- `rs1[47:32]` = number of columns of D/B matrix
- `rs1[63:48]` = number of rows of D/B matrix
- `rs2[31:0]` = local scratchpad address of C matrix.
  If this is set to all high bits, then C will not be written to the scratchpad or accumulator.
- `rs2[47:32]` = number of columns of C matrix
- `rs2[63:48]` = number of rows of C matrix
- `funct` = 6

Commit Behavior: This instruction commits on the cycle after the systolic array receives it. The systolic array remains idle until the subsequent OS/WS specific instructions are seen.

Computing

Explicitly Preloaded

Format: `matmul.compute.preloaded rs1, rs2`

- `rs1[31:0]` = local scratchpad address (systolic array single-axis addressed) of A matrix
- `rs1[47:32]` = number of columns of A matrix
- `rs1[63:48]` = number of rows of A matrix
- `rs2[31:0]` = local scratchpad address (systolic array single-axis addressed) of B matrix (when output-stationary), or D matrix (when weight-stationary)
- `rs2[47:32]` = number of columns of B/D matrix
- `rs2[63:48]` = number of rows of B/D matrix
- `funct` = 4
- This instruction will compute on the value preloaded (D if output-stationary, or B if weight-stationary)

Re-use Previous Preloads

Format: `matmul.compute.accumulated rs1, rs2`

- `funct` = 5
- `rs1` and `rs2` have the same encoding as the `matmul.compute.preloaded` encoding
- If output-stationary, this instruction will compute on the previously computed result (C) in the systolic array, accumulating on top of it
- If weight-stationary, this instruction will compute on the previously preloaded weights (B) in the systolic array

Loop Instructions

Gemmini includes CISC-type instructions which can perform matmuls and convolutions on data that is much larger than `DIMxDIM`.

There's nothing these CISC instructions do which a programmer couldn't do by tiling and looping through the other ISA instructions described above;
however, these CISC instructions may achieve higher throughput than such tiled loops written by non-expert programmers.
The CISC instructions should be considered performance enhancers; they do not give the accelerator any new functionality that it wouldn't have otherwise.

The CISC instructions have too many operands to fit into a single RISC-V custom instruction. Therefore, they are implemented as a sequence of many RISC-V custom instructions which must be called consecutively by the programmer.

These instructions can be found `software/gemmini-rocc-tests/include/gemmini.h`, together with example usages.
We list below their arguments.

These loop instructions are experimental and subject to change.

`gemmini_loop_ws` Matmul Loop (WS Dataflow)

This instruction calculates `A * B + D = C`, but `A`, `B`, `D`, and `C` can all be larger than `DIMxDIM`. `A`, and `B` must be of type `inputType`, but both `D` and `C` can be *either* `inputType` or `accType`.

The sizes of these matrices are represented by `I`, `J`, and `K`:

```
scratchpad rows of A = I * K * DIM
scratchpad rows of B = K * J * DIM
accumulator rows of D = I * J * DIM
accumulator rows of C = I * J * DIM
```

However, the total number of scratchpad rows taken up by a single `gemmini_loop_ws` must be at most **half** of the total scratchpad size, because Gemmini performs double-buffering during CISC instructions.
To compute larger matrix multiplies, the loop instructions must also be tiled within an outer loop.

To support outer-tiling of the `gemmini_loop_ws` instruction, we include an argument called `ex_accumulate`, which determines whether to perform a matmul on top of the partial sums that already exist within the accumulator (from previous calls to `gemmini_loop_ws` within the same outer-loop).

`gemmini_loop_conv_ws` Conv Loop (WS Dataflow)

Gemmini also includes a CISC instruction for convolutions, implemented similarly to the matmul CISC instruction.
`gemmini_loop_conv_ws` will perform a convolution with the WS dataflow, and also supports features such as max-pooling, transpose convolutions, and various preprocessing transformations on the weight and input data.

Like `gemmini_loop_ws`, the inputs to a single `gemmini_loop_conv_ws` call must fit within half of Gemmini's private memory, to support double-buffering.
If the programmer would like to perform larger convolutions, they must tile and wrap `gemmini_loop_conv_ws` within an outer-loop.

Citing Gemmini

If Gemmini helps you in your academic research, you are encouraged to cite our paper. Here is an example bibtex:

```
@INPROCEEDINGS{gemmini-dac,
  author={Genc, Hasan and Kim, Seah and Amid, Alon and Haj-Ali, Ameer and Iyer,
Vighnesh and Prakash, Pranav and Zhao, Jerry and Grubb, Daniel and Liew, Harrison
and Mao, Howard and Ou, Albert and Schmidt, Colin and Steffl, Samuel and Wright,
John and Stoica, Ion and Ragan-Kelley, Jonathan and Asanovic, Krste and Nikolic,
Borivoje and Shao, Yakun Sophia},
  booktitle={Proceedings of the 58th Annual Design Automation Conference (DAC)},

  title={Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via
Full-Stack Integration},
  year={2021},
  volume={},
  number={},
  pages={}
}
```

Acknowledgements

- This project was, in part, funded by the U.S. Government under the DARPA RTML program (contract FA8650-20-2-7006). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.
- The Gemmini logo was designed by Dima Nikiforov (@CobbledSteel).

Hierarchy

```
Queue_4_inTestHarness
    |---ChipTop(chiptop)
        |---DigitalTop(system)
            |---InterruptBusWrapper
            |---SystemBus
            |---PeripheryBus
            |---FrontBus
```

```
            |---PeripheryBus_1
            |---MemoryBus
            |---CoherenceManagerWrapper
            |---TilePRCIDomain(tile_prci_domain)
                |---TileResetDomain(tile_reset_domain)
                    |---RocketTile(tile)
                        |---TLXbar_7
                        |---IntXbar_1
                        |---BundleBridgeNexus_6
                        |---DCache
                        |---Gemmini
                        |---Frontend
                        |---FPU
                        |---HellaCacheArbiter
                        |---PTW
                        |---RRArbiter_5
                        |---RoccCommandRouterGemmini
                        |---Frontend
                        |---FPU
                        |---HellaCacheArbiter
                        |---PTW
                        |---RRArbiter_5
                        |---RoccCommandRouter
                        |---SimpleHellaCacheIF
                        |---Queue_65
                        |---Rocket
                |---FixedClockBroadcast_5
                |---TLBuffer_20
                |---TLBuffer_21
                |---IntSyncAsyncCrossingSink
                |---IntSyncSyncCrossingSink_1
                |---IntSyncCrossingSource_1
            |---ClockSinkDomain
            |---CLINT
            |---TLDebugModule
            |---IntXbar
            |---BundleBridgeNexus_15
            |---IntSyncCrossingSource_5
            |---IntSyncCrossingSource_1
            |---ClockSinkDomain_1
            |---ClockSinkDomain_2
            |---ClockSinkDomain_3
            |---IntSyncSyncCrossingSink_1
            |---ClockGroupAggregator_6
            |---ClockGroupParameterModifier
            |---ClockGroupParameterModifier_1
            |---ClockGroupResetSynchronizer
            |---TileClockGater
            |---TileResetSetter
            |---DebugTransportModuleJTAG
        |---ClockGroup_6
        |---DividerOnlyClockGenerator
        |---ResetCatchAndSync_d3
        |---AsyncResetSynchronizerShiftReg_w1_d3_i0
        |---ResetSynchronizerShiftReg_w1_d3_i0
        |---EICG_wrapper
        |---GenericDigitalOutIOCell
        |---GenericDigitalInIOCell
```

```
    |---SimJTAG
    |---plusarg_reader
    |---AsyncQueue_inTestHarness
    |---SerialRAM_inTestHarness
    |---SimSerial
    |---plusarg_reader
    |---SimDRAM
    |---UARTAdapter_inTestHarness
```