μARCHDB EVENT VIEWER FOR GEMMINI

Kevin He, Victor Hu*, Nicolas Castaneda, Ryan Ma**

kevinjhe@berkeley.edu, victorhu3@berkeley.edu,
nicolas.a.castaneda@berkeley.edu,
ryan.ma3011@berkeley.edu

1. INTRODUCTION

Gemmini is a systolic-array based DNN accelerator hardware generator. The Gemmini accelerator code, not including the Rocket core which it interfaces with, is comprised of 15,664 lines of Chisel HDL, split across 55 files and many more modules [2]. For open-source developers and users, understanding Gemmini's microarchitecture and codebase represents a major obstacle to debugging and optimization. We present a Gemmini implementation of our hardware-based event tagging and tracking tool with visualization for debugging, education, and performance analysis.

2. MOTIVATION

Gemmini is highly complex system with custom RISC-V ISA based RoCC instructions. It has a DMA, a store, load, and execute controller, scratchpad, accumulator, mesh systollic array, transposer, reservation station, Im2Col unit, and scaling arithmetic units. It also has hardware controllers for decoding CISC matmul and convolution commands into RISC instructions, giving the programmer more options to use coarse-grained or fine-grained commands. This complexity makes for an exceptionally difficult debugging process. Instructions are both received from the CPU and generated dynamically by Gemmini's internal FSMs. Instructions are complex with 64+64+32 bits of encoding space. Each instruction is responsible for spawning many more possible loads and stores with data being sent to different functional units, depending on the configuration.

Currently, debugging and developing on Gemmini requires an engineer to parse through simulation waveforms and thus, a highly detailed knowledge of the microarchitecture to track each signal. There is also no instruction diassembler like Spike for RISC-V instructions, which forces developers to hand splice addresses and correlate function codes with instruction dumps to track the instruction flow. There is no easy way to visualize where instructions and data go inside Gemmini. Lastly, with Gemmini's complex instruction handling, it becomes quite difficult to follow how instructions are decoded. For all but the most experienced Gemmini developers, these barriers slow down code optimization and any future modifications or improvements to the microarchitecture.

3. PRIOR WORK

We were inspired by the Gem5 O3 pipeline viewer, which visualizes out-of-order CPU instruction execution[4]. We use the Konata instruction pipeline visualizer for our frontend. This pipeline visualizer was also designed for Gem5 processors [5]. [1] proposed a similar hardware token passing method to tracking event dependencies, which they used to identify critical instruction paths. The authors were able to use the instruction criticality data to reduce resource contention in instruction scheduling.

We previously worked on annotating the Rocket and Sodor in-order RISC-V cores. Rocket and Sodor have simpler instruction flow.

4. IMPLEMENTATION DETAILS

Our microarchitectural event tracking tool is composed of three parts: Chisel tagging code, a Python graph processing script, and the frontend pipeline visualizer.

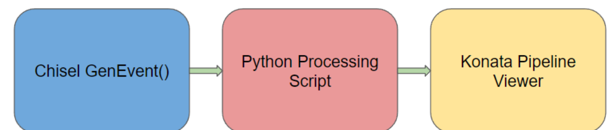

Figure 1: Tool Workflow

4.1. Chisel GenEvent

The Chisel tagging code is comprised of a GenEvent object, the event annotations, and related token-passing hardware in the Gemmini microarchitecture. The GenEvent object creates a small snippet of hardware code and includes a Chisel printf, an unique ID generator, and a cycle counter. The GenEvent can be called like a function in the Gemmini RTL and can be conditioned with Chisel's *when*, *elsewhen*, and *otherwise* clauses. On each cycle that the GenEvent is "called" or enabled, it prints a JSON string with the event name, a unique id, parent event, cycle count, and an optional data field. A parent event ID can be optionally input into the GenEvent and is used to connect two events such that the event's ID and its parent's ID can be used to generate an edge in the event graph. The GenEvent then outputs the unique event ID that can be passed through the hardware to the next GenEvent location in the microarchitecture. The event name field specifies the label for the location or microarchitecture event that we are capturing. The data field can be used to print values of wires or registers in the design such as tags, IDs, or addresses. GenEvent also allows developers to specify the event ID rather than having it generate a unique ID. This can be useful if there already exists IDs in the microarchitecture that can be used to uniquely identify event paths. For example,

*EE290 Project Group Members

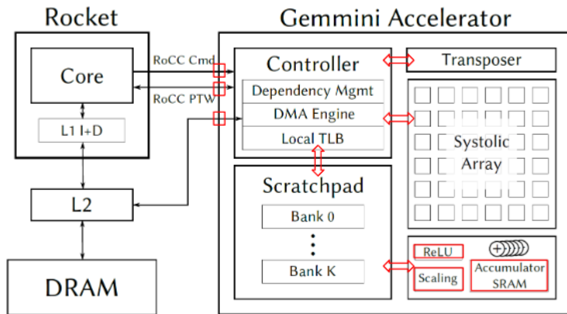

Figure 2: Gemmini Annotation Locations

the *rob_id* in Gemmini is used to track commands throughout their execution and is used in our implementation when the Chisel logic is not easily modified to support tag/token passing. Figure 8 in the Appendix lists the GenEvent and EventTag Chisel code.

4.1.1. Gemmini Microarchitecture

We added GenEvent annotations to critical control flow and datapath locations in Gemmini. In *Controller.scala*, we annotated the IO where Gemmini RoCC commands are received from the CPU, where they are turned into fine-grain instructions by the FSMs, reservation station issue, load, execute, and store controller issue from the reservation station and finally, instruction retire from the three controllers. In *ExecuteController.scala*, we annotated when the systolic array was flushed or fired, scratchpad and accumulator read/writes for the A, B, D matrices, mesh data inputs, and Im2Col requests.

To enable event tag passing between different areas of RTL, we at times had to modify Chisel bundles and data structures to store and pass tags (See Figure 9 of the Appendix for an example). For instance, in Figure 10 of the Appendix, we have different events for each matrix and scratchpad bank number for scratchpad reads. The generated event tags are enqueued onto mesh_cntl_signals_q which is eventually passed to Figure 11 of the Appendix, where the mesh input fire event is different depending on the matrix and where the data originated, with the parent event set to the pipeline_tag. This event tag passing is important to reconstruct the event graph later on. The code for annotated Gemmini can be found at `https://github.com/kevinhe5/gemmini/tree/pipeline`

4.2. Python Graph Processing

The purpose of the Python script is to take the Verilator or VCS log generated during RTL simulation, build a graph of events, and parse it into an output file compatible with the pipeline visualizer. We named the script iris.py.

The script is adaptable to other microarchitectures and takes a JSON configuration as input. Figure 12 shows the config file used for Gemmini. The config specifies event names, start stages, split stages, and end stages. Event names correspond to the eventName parameter of the Chisel GenEvents. Start stages specify where an instruction can originate; for Gemmini, these stages are CMD in the controller.scala, LOOP_CONV, and LOOP_MATMUL which refer to the LoopConv and LoopMatmul CISC instruction FSMs. End stages specifies where an instruction can retire. In a CPU pipeline, this is could be the writeback stage. For Gemmini, instructions can end up in multiple locations. For example, some config instructions only go to configure the LoopConv and LoopMatmul modules. Controller config instructions are sent to the execute, store, and load controllers, where they set the internal controller states for future execution. These instructions are retired almost immediately. Move-in, move-out, preload, and compute instructions are much longer latency and are retired in LD_RET, EX_RET, and ST_RET stages when the respective controllers send a completed signal to the main controller.

We also decode Gemmini instructions in iris.py. RoCC instructions contain a 32 bit instruction and two 64 bit register values: rs1 and rs2. These bits are used for setting the number of rows, columns, strides, addresses, and various other configuration parameters in Gemmini. Our GenEvent prints out the raw instruction encoding, which we mask in iris.py to provide the instruction type and its parameters in the event viewer. Based on the various instruction decodings, the script is coded to fetch the various fields of all the different instructions, making our system able to detect fine-grained instruction flows.

We construct the event graph with the NetworkX [3]. Since each event output from the GenEvent has a parent ID and its own event ID, we can draw an edge between the event and its parent. However, we must first pre-process the event log from our RTL simulator. Figure 13 shows a small portion of an output from VCS for scratchpad reads events and the related scratch pad rows being sent to the mesh. Since the GenEvents allow users to specify their own event IDs and allow the event ID to be the same as the parent event ID, we must first uniquify the event IDs. Event IDs that are the same are given a unique ID while preserving the relationship/edge between the two adjacent events by sorting by cycle time. For example, as an instruction passes from the reservation station to the execute controller to the spatial array mesh and retires, the ROB ID is used as the event ID for all the events. Because, each event will occur after the other, they will have different cycle counter values. Thus, their uniqified IDs construct edges in the order of execution.

Once the graph is constructed, we perform depth first search through the graph to extract the instruction paths. Each event is a graph node, containing the corresponding cycle counter value and data field. From these paths, we can generate the Konata log, where each path goes on one line in the viewer.

The code for the iris.py can be found at `https://github.com/ncastaneda02/uarchdb/tree/Gemmini`. The JSON config file for Gemmini is gemmini.json. An example Konata log for Gemmini can be found in gemmini.log.

4.3. Pipeline Viewer

We created a custom fork of the Konata instruction pipeline visualizer. This fork contains some quality of life changes we made such as freezing the vertical height when zooming to make the experience similar to other waveform viewers. Konata ingests the Kanata Log Format file generated from the previous step. The fork can be found at `https://github.com/victorhu3/Konata`.

5. RESULTS

Refer to Figure 5 of the Appendix for an example of a weight stationary tiled matmul workflow being visualized. The blue bars

represent the instructions sitting in the ROB and the green bars are when they reach their respective controllers. Figure 7 is an example of a convolution workflow. In Figure 6, the instruction decoding feature is overlaid.

The pipeline viewer allows the user to see interesting interactions in Gemmini. In Figure 3, an instruction waits in the reservation station before it can be issued to the execute controller, until the load instruction returns. Konata allows for visualizations of dependencies that would have otherwise been hard to identify in a waveform.

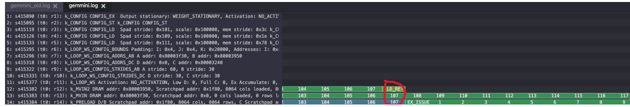

Figure 3: Execute Stalling

Using the Python script, we also were able to generate a top-down visualization of graph of events with GraphViz. The roots of each tree are instructions being issued by the host processor.

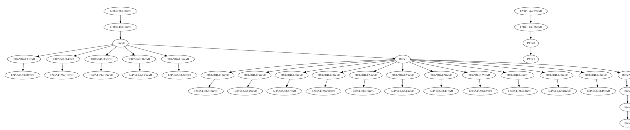

Figure 4: GraphViz Output

Since we are ultimately constructing a graph, our GenEvent tagging solution allows for designers to choose the level of abstraction they want to annotate in the microarchitecture. We mostly painted broad strokes, annotating the largest controllers and most common memory transactions; however, a more experienced Gemmini developer should have no difficulty with annotating the most minutia of Gemmini signals and dependencies. NetworkX is extensible to very large graphs, and there was no noticeable delay with the iris.py script for our event paths.

6. CONCLUSION

In this paper, we introduced a tool to annotate and visualize events for Gemmini. We hope that this tool will speedup debugging, encourage open-source development, and provide valuable insights for the Gemmini microarchitecture.

7. CONTRIBUTIONS

Kevin He annotated Gemmini with GenEvent tags and worked with Nico to design the iris.py graph parsing script.

Victor Hu helped annotate Gemmini, improved the iris.py script, and worked on quality of life improvements of the Konata pipeline viewer.

Ryan Ma wrote the Gemmini instruction dissassembler in iris.py.

We thank Nico Castaneda, Jerry Zhao, and Dima Nikiforov for their help and advice on this project.

References

[1] B. Fields, S. Rubin, and R. Bodik. “Focusing processor policies via critical-path prediction”. In: *Proceedings 28th Annual International Symposium on Computer Architecture*. 2001, pp. 74–85. DOI: 10.1109/ISCA.2001.937434.

[2] Hasan Genc et al. “Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration”. In: *Proceedings of the 58th Annual Design Automation Conference (DAC)*. 2021.

[3] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. “Exploring Network Structure, Dynamics, and Function using NetworkX”. In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. 2008, pp. 11–15. URL: http://conference.scipy.org/proceedings/SciPy2008/paper_2/.

[4] *O3 Pipeline Viewer*. URL: https://www.gem5.org/documentation/general_docs/cpu_models/visualization/.

[5] Ryota Shioya. *Konata: An instruction pipeline visualizer for Onikiri2-Kanata/GEM5-o3pipeview formats.* URL: https://github.com/shioyadan/Konata.

8. APPENDIX

8.1. Reproducibility

For our development of the tool, we used the GemminiRocketConfig config and tiled_matmul_ws_At-baremetal test in Chipyard:
make run-binary CONFIG=GemminiRocketConfig BINARY=/tools/designs/kevinhe/chipyard6/generators/gemmini/software/gemmini-rocc-tests/build/bareMetalC/tiled_matmul_ws_At-baremetal

To run the Python processing script, ensure that the pandas, numpy, and networkx packages are installed, then:
python3 iris.py –log_file [path/to/VCS_out_file] –schema_file gemmini.json –output_file gemmini.log –verbose –gemmini

This will output the Konata log to gemmini.log. Once Konata is launched, the log file can be drag and dropped in for visualization.

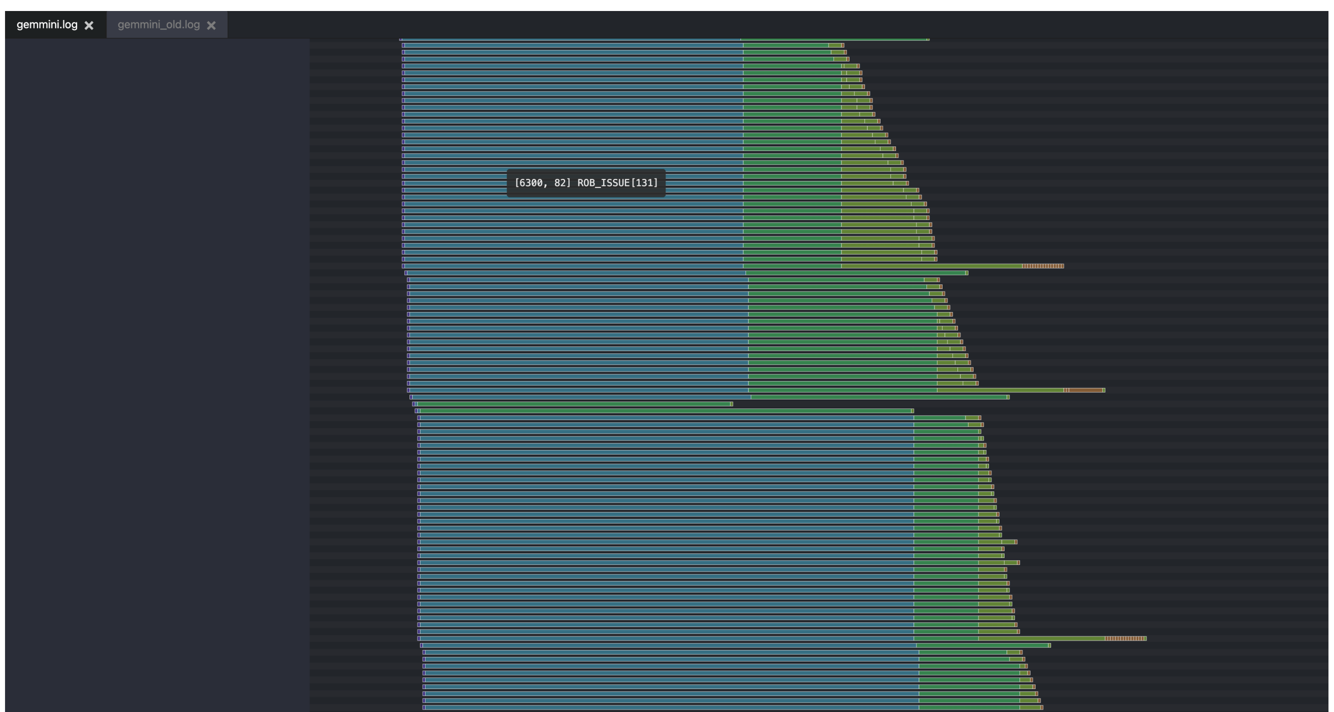

Figure 5: tiled_matmul_ws_At Konata View

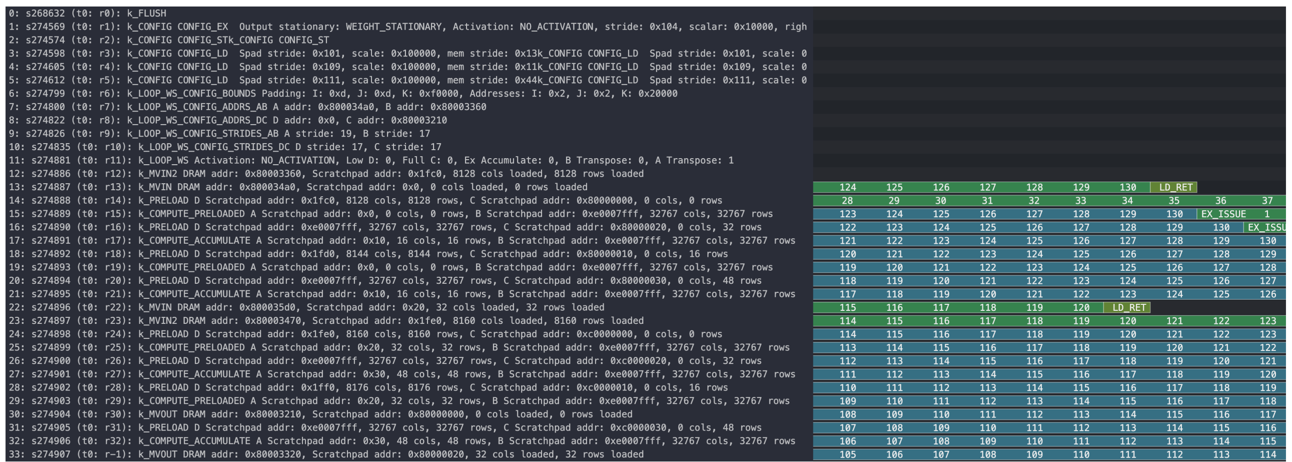

Figure 6: Konata View

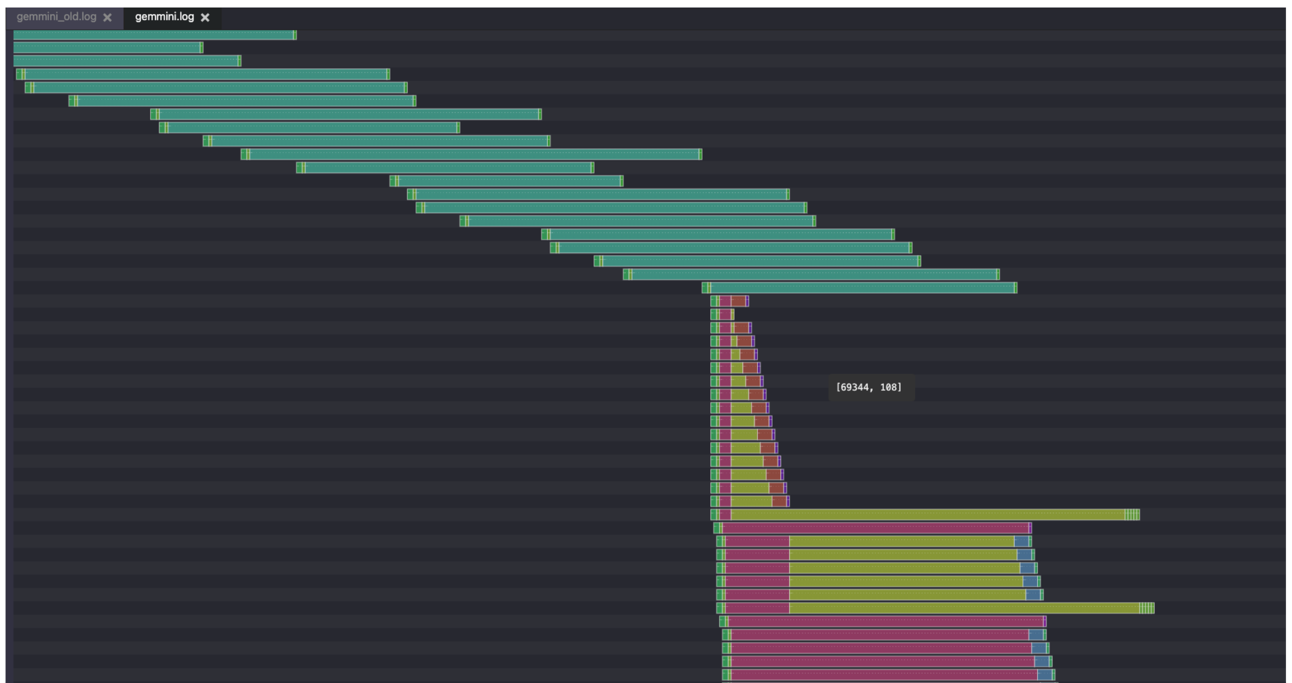

Figure 7: conv-baremetal Konata View

8.2. Code Snippets

```
object GenEvent {
  var instance_ctr: Int = 0
  def apply(eventName: String, data: UInt, parent: Option[EventTag], id: Option[UInt] = None): EventTag
     = {
    var new_id = Wire(UInt(64.W))
    val id_ctr = RegInit(0.U(32.W))
    id_ctr := id_ctr + 1.U
    new_id := Cat(instance_ctr.asUInt(32.W), id_ctr)
    if (parent.isDefined) {
      if (id.isDefined) {
        printf(cf"{\"id\": \"0x${id.get}%x\", \"parents\": \"0x${parent.get.id}%x\", \"cycle\": \"
    $id_ctr\", \"event_name\": \"$eventName\", \"data\": \"0x$data%x\"}\n")
      } else {
        printf(cf"{\"id\": \"0x$new_id%x\", \"parents\": \"0x${parent.get.id}%x\", \"cycle\": \"$id_ctr
    \", \"event_name\": \"$eventName\", \"data\": \"0x$data%x\"}\n")
      }
    } else {
      if (id.isDefined) {
        printf(cf"{\"id\": \"0x${id.get}%x\", \"parents\": \"None\", \"cycle\": \"$id_ctr\", \"
    event_name\": \"$eventName\", \"data\": \"0x$data%x\"}\n")
      } else {
        printf(cf"{\"id\": \"0x$new_id%x\", \"parents\": \"None\", \"cycle\": \"$id_ctr\", \"event_name
    \": \"$eventName\", \"data\": \"0x$data%x\"}\n")
      }
    }
    instance_ctr += 1
    return EventTag(new_id)
  }
}
class EventTag extends Bundle {
  val id = UInt(64.W)
}
object EventTag {
  def apply(id: UInt): EventTag = {
    val tag = Wire(new EventTag)
    tag.id := id
    return tag
  }
}
```

Figure 8: GenEvent and EventTag Chisel code

```
  class ComputeCntlSignals extends Bundle {
    ...
    //For pipeline viewer
    val pipeline_tag_a = new EventTag
    val pipeline_tag_b = new EventTag
    val pipeline_tag_d = new EventTag
  }
```

Figure 9: New EventTag fields for the ComputeCntlSignals bundle in ExecuteController.scala

```
when(io.srams.read(i).req.fire) {
  when (read_a && a_ready) {
        mesh_cntl_signals_q.io.enq.bits.pipeline_tag_a := GenEvent(s"SP_RD_A$i", io.srams.read(i).req.bits.addr, Some(EventTag(cmd.bits(preload_cmd_place).rob_id.bits)))
      }
      when (read_b && b_ready) {
        mesh_cntl_signals_q.io.enq.bits.pipeline_tag_b := GenEvent(s"SP_RD_B$i", io.srams.read(i).req.bits.addr, Some(EventTag(cmd.bits(preload_cmd_place).rob_id.bits)))
      }
      when (read_d && d_ready) {
        mesh_cntl_signals_q.io.enq.bits.pipeline_tag_d := GenEvent(s"SP_RD_D$i", io.srams.read(i).req.bits.addr, Some(EventTag(cmd.bits(preload_cmd_place).rob_id.bits)))
      }
    }
```

Figure 10: GenEvent tagging code for Scratchpad reads in the ExecuteController.scala. For these GenEvents, the parent tag is the rob_id and its own event id is a uniquely generated tag

```
//For pipeline viewer
when(mesh.io.a.fire) {
  when(cntl.a_garbage) {
  }.elsewhen(cntl.a_unpadded_cols === 0.U) {
    GenEvent("A_0_PAD", 0.U, Some(EventTag(cntl.rob_id.bits)))
  }.elsewhen(cntl.im2colling) {
    GenEvent("A_IM2COL", 0.U, Some(cntl.pipeline_tag_a))
  }.elsewhen(cntl.a_read_from_acc) {
    GenEvent("A_ACC->MESH", cntl.a_bank_acc, Some(cntl.pipeline_tag_a))
  }.otherwise {
    GenEvent("A_SP->MESH", cntl.a_bank, Some(cntl.pipeline_tag_a))
  }
}

```

Figure 11: GenEvent tags for mesh input fires corresponding to Scratchpad read GenEvents in the previous figure. The GenEvent IDs from the previous figure are passed as the parent ID in the above GenEvents to construct a causal relationship where after a row is read from the Scratchpad, it is sent into the mesh.

```
{
    "event_names": ["CMD", "LOOP_CONV", "LOOP_MM", "ROB_ISSUE", "LD_ISSUE", "ST_ISSUE", "EX_ISSUE", "ST_RET",
                    "LD_RET", "EX_RET", "MESH_FIRE", "A_GARBAGE", "B_GARBAGE", "D_GARBAGE", "A_0_PAD", "B_0_PAD", "D_0_PAD",
                    "A_ACC->MESH", "B_ACC->MESH", "D_ACC->MESH", "A_SP->MESH", "B_SP->MESH", "D_SP->MESH", "ACC_WR_0",
                    "ACC_WR_1", "SP_RD_A0", "SP_RD_D1", "LOOP_MM_CMD"],
    "event_types": ["inst_bytes", "bytes", "bytes", "inst_bytes", "bytes", "bytes", "bytes", "bytes", "bytes", "bytes",
                    "bytes", "bytes", "bytes", "bytes", "bytes", "bytes", "bytes", "bytes", "bytes", "bytes",
                    "bytes", "bytes", "bytes", "bytes", "bytes", "bytes", "bytes", "bytes"],
    "start_stages": ["CMD", "LOOP_CONV_CMD", "LOOP_MM_CMD"],
    "split_stages": ["CMD", "LOOP_CONV", "MM_LOOP"],
    "end_stages": ["ST_RET", "LD_RET", "EX_RET", "ST_ISSUE", "LD_ISSUE", "LOOP_MM", "LOOP_CONV", "A_ACC->MESH",
                    "B_ACC->MESH", "D_ACC->MESH", "A_SP->MESH", "B_SP->MESH", "D_SP->MESH", "A_0_PAD", "B_0_PAD", "D_0_PAD", "ACC_WR_0", "ACC_WR_1"]
}

```

Figure 12: Iris.py Config JSON for Gemmini Stages

```
{"id": "0x14", "parents": "0x0000000000000014", "cycle": "    275062", "event_name": "MESH_FIRE", "data": "0x15200000042018140"}
{"id": "0x0000001d00043276", "parents": "0x0000000700043271", "cycle": "    275062", "event_name": "A_SP->MESH", "data": "0x0"}
{"id": "0x0000000c00043277", "parents": "0x0000000000000014", "cycle": "    275063", "event_name": "SP_RD_D1", "data": "0xfde"}
{"id": "0x0000002300043277", "parents": "0x0000000c00043272", "cycle": "    275063", "event_name": "D_SP->MESH", "data": "0x1"}
{"id": "0x0000000700043278", "parents": "0x0000000000000014", "cycle": "    275064", "event_name": "SP_RD_A0", "data": "0x002"}
{"id": "0x0000000c00043278", "parents": "0x0000000000000014", "cycle": "    275064", "event_name": "SP_RD_D1", "data": "0xfdd"}
{"id": "0x0000001d00043278", "parents": "0x0000000700043273", "cycle": "    275064", "event_name": "A_SP->MESH", "data": "0x0"}
{"id": "0x0000000700043279", "parents": "0x0000000000000014", "cycle": "    275065", "event_name": "SP_RD_A0", "data": "0x003"}
{"id": "0x0000000c0004327b", "parents": "0x0000000000000014", "cycle": "    275067", "event_name": "SP_RD_D1", "data": "0xfdc"}
{"id": "0x000000070004327c", "parents": "0x0000000000000014", "cycle": "    275068", "event_name": "SP_RD_A0", "data": "0x004"}
{"id": "0x0000000c0004327c", "parents": "0x0000000000000014", "cycle": "    275068", "event_name": "SP_RD_D1", "data": "0xfdb"}

```


Figure 13: Snippet from VCS output from GenEvents