Nexus zkVM 3.0 Specification

Michel Abdalla, Arka Rai Choudhuri, Jens Groth, Yoichi Hirai, Ben Hoberman,
Samuel Judson, Daniel Marin, Duc Tri Nguyen, Evan Schott, and Kristian Sosnin

Nexus
research@nexus.xyz

February 12, 2025

Abstract

We describe the architecture of the Nexus zkVM 3.0, the third iteration of the Nexus zero-knowledge virtual machine based on the `Stwo` prover. The Nexus zkVM is a general-purpose verifiable processor, designed to prove the correct execution of arbitrary computations. Specifically, the Nexus zkVM emulates virtual machine execution and produces succinct proofs of correct computation.

Like previous versions of the Nexus zkVM, the instruction set used by the Nexus Virtual Machine is based on the popular RISC-V instruction set architecture so that existing developer tooling can be used with little modification. Version 3.0, however, uses a Harvard architecture in which the program being executed resides in a read-only memory space separate from the data.

In addition to the architectural change, another major change to the zkVM frontend is that constraints are now specified using Algebraic Intermediate Representation (AIR) and can work over the Mersenne prime field M31 used by the `Stwo` prover.

In order to describe the new arithmetization, we split the constraints for the zkVM into several components:

- **CPU**: Responsible for fetching, decoding, and preparing instructions for execution
- **Register memory**: Manages access to the read-write registers
- **Program memory**: Manages access to the read-only byte-addressable program memory
- **Data memory**: Manages access to the read-write byte-addressable random-access memory
- **Instruction execution**: Deals with the actual execution of instructions

Since these components may depend on each other, we also specify the interaction between the different components used in the Nexus zkVM.

Contents

1 Introduction

This document specifies the new version of the Nexus zero-knowledge Virtual Machine, Nexus zkVM v3.0, which uses the `Stwo` prover [STW24] in the backend.

Like previous versions of the Nexus zkVM [Nex24], the instruction set used by the Nexus Virtual Machine is based on the RISC-V RV32I instruction set [RIS19] so that existing developer tooling can be used with little modification. There are, however, several important changes being introduced:

- The new zkVM version now uses a (modified) Harvard architecture in which the program being executed resides in a read-only memory space separate from the data.
- Similar to Jolt [AST24], the new Nexus zkVM is designed around a 'pay only for what you use' memory architecture, where unused memory does not need to be proven. In particular, the Nexus zkVM now operates on a two-pass tracing architecture, first executing the program to obtain statistics about the resultant memory usage and then executing it again in a modified Harvard architecture with a fixed-memory organization determined from the statistics of the first execution.
- The backend now uses the `Stwo` prover by StarkWare, which is based on the Circle STARK protocol [HLP24].
- The constraints are now specified using Algebraic Intermediate Representation (AIR) and can work over the Mersenne prime field `m31` used by the `Stwo` prover.

In this document, we focus on the description of the frontend for the new Nexus zkVM, which is responsible for transforming the program execution into an arithmetic representation of the execution which is suitable for the `Stwo` backend prover.

1.1 Design overview

In order to describe Nexus zkVM 3.0, we divide the specification into several components, each of which is responsible for dealing with a particular task:

- **CPU**: Responsible for fetching, decoding, and preparing instructions for execution
- **Register memory**: Manages access to the read-write registers
- **Program memory**: Manages access to the read-only byte-addressable program memory
- **Data memory**: Manages access to the read-write byte-addressable random-access memory
- **Execution**: Deals with the actual execution of instructions

Moreover, since the different components may depend on each other, we also specify the interaction between the different components used in the zkVM.

Component representation: In our specification, each component will represented in terms of trace matrices and constraints.

- Trace matrix: This is a matrix of wire values used by the component. In particular, each trace column is a vector of wire values sharing some common attribute. For example, we may have one column which keeps track of the instruction opcode and another column that keeps track of the value of the destination register used in each cycle.
- List of Constraints: These constraints resemble the gate functionalities. Each constraint is applied to a subset of rows over a subset of columns of the trace matrix. For example, we may require that the sum of the first and second columns equal the third column for all rows.

  In our specification, the following types of constraints are used:

  - Finite field arithmetic constraints: Addition and multiplication in some finite field.
  - Lookup Constraints, including both unindexed and indexed lookup relation.

Finite field representation of wire values: The Nexus zkVM represents each wire value as a field element in the Mersenne prime field `m31` used by the `Stwo` prover.

1.2 zkVM components

As stated above, there are 5 components in the Nexus zkVM: a *CPU* component, 3 memory components for handling accesses to the *program*, *register*, and *data* memories, and an *execution* component.

1.2.1 CPU component

The CPU component of the Nexus zkVM is the component that emulates the behavior of the CPU of the virtual machine, and hence, plays a central role. In particular, this component is responsible for fetching, decoding, and preparing instructions for execution.

In order to achieve these goals, the CPU component performs the following tasks in each CPU cycle:

- Interacts with the program memory component to fetch the next instruction at the address pointed by the program counter;
- Decodes the instruction and check the correctness of its format;
- Interacts with the register memory component to read the values associated with the instructions operands, when necessary;
- Interacts with the execution component to execute the instruction; and
- Interacts with the register memory component to update register contents, when necessary.

Enforcing the correctness of tasks, such as the sign extension of immediate values or the encoding of instructions, is handled exclusively by the CPU component.

1.2.2 Memory components

The Nexus zKVM has different components for handling the behavior of the three types of memory used by the Nexus Virtual Machine: the *program memory*, the *register memory* and *data memory*. While the program memory is a read-only memory space storing the program being executed, both the register and data memories are read-write random-access memories of size 32 and 2^{32} respectively. Both the program memory and the data memory are byte-addressable while the register memory stores 32-bit words.

In order to maintain the consistency of the accesses to the register and data memories, the Nexus zkVM uses well known offline memory checking techniques [BEG+94], which we recall in Section 3.2. The main advantage of this technique is that one does not need to keep track of the actual status of the running memory. Instead, the memory checking algorithm only keeps a trace digest of memory accesses, which is inexpensive and can be updated at a cost that is independent of the size of the memory.

In the case of the program memory component, a simpler memory checking technique can be used to maintain the consistency of the memory accesses. In particular, instead of keeping a timestamp for each memory cell, it suffices to associate a counter to each memory cell to keep track of the number of times that each cell has been read.

In this version of the Nexus zkVM, the computation of the digest is implemented using logarithmic derivatives aka logups [EKRN24, Hab22].

1.2.3 Execution component

The final component of the Nexus zkVM is the execution component, which is responsible for enforcing the correct execution of the instructions supported by the Nexus Virtual Machine.

Currently, this component provides support for the Nexus Virtual Machine instruction set described in Section 2, which is closely related to the RISC-V RV32I instruction set in the Volume I, Unprivileged Specification version 20191213 [RIS19]. As a result, existing tooling for RISC-V RV32I can be used with usually no modification.

1.3 Outline

- Section 2 provides an overview of the new zkVM architecture, highlighting in particular the design features of the machine architecture, such as the new memory layout and the set of supported environment calls. This section also introduces a new two-pass tracing mechanism used to improve the memory usage of the Nexus zkVM. Section 2 also recalls the Nexus Virtual Machine instruction set and its encoding.
- Section 3 explains a few useful tools and concepts used in the new zkVM, such as offline memory checking and its implementation based on logups. This section also discusses lookup tables for range checks and bitwise operations, which are used throughout the new Nexus zkVM design.
- Section 4 specifies the CPU component, which is responsible for fetching, decoding, and preparing instructions for execution.
- Section 5 details the read-write register memory component, which is responsible for managing access to the registers.
- Section 6 describes the program memory component, which is responsible for managing access to the read-only program memory.
- Section 7 reviews the specification of the read-write data memory component, which is responsible for managing access to the data memory.
- Section 8 specifies the instruction execution component, which is responsible for enforcing constraints that guarantee the correct execution of the instructions supported by the Nexus Virtual Machine.

,

2 The Nexus zkVM machine architecture

The Nexus zkVM defines a map $\mathsf{zkVM} : (P, \mathsf{cfg}, x, y) \mapsto (z, \pi)$ where P is a program, cfg is the machine configuration which specifies details such as memory size and maximum execution time, x is a public input, y is a private input, z is the output (or an error), and π is a succinct proof. This proof asserts that when running P in configuration cfg on inputs x, y the output is z, or $P[\mathsf{cfg}](x, y) = z$. In isolation we can consider the zkVM's *machine architecture* $\mathsf{zkVM}_{\mathsf{MA}} : (P, \mathsf{cfg}, x, y) \mapsto z$. The primary use of this machine architecture is to support the verifiable computation of (z, π). However, it is a well-defined virtual machine in its own right, and we describe it in detail here. In particular, we highlight the design features of the machine architecture that — though idiosyncratic for traditional general purpose computation — are useful for verifiable computation, such as the particular memory layout and the set of supported environment calls.

2.1 Architectural overview

The $\mathsf{zkVM}_{\mathsf{MA}}$ is built around an instruction set, not the same as, but close enough to the RISC-V RV32I instruction set in the Volume I, Unprivileged Specification version 20191213 [RIS19] that existing tooling can be used with usually no modification. The primary difference is the lack of a few supported instructions, such as `fence` and `ebreak`. The $\mathsf{zkVM}_{\mathsf{MA}}$ uses a modified Harvard architecture (similar to the one adopted by Jolt [AST24]), in which the program, data, and input-output memory segments are distinct and permissioned with respect to whether they can be read from, written to, or both. The $\mathsf{zkVM}_{\mathsf{MA}}$ has 32-bit words, 32 registers $\mathtt{x0}, \mathtt{x1}, \ldots, \mathtt{x31}$, and memory addresses range over $[0, 2^{32} - 1]$.

As a virtual machine, the $\mathsf{zkVM}_{\mathsf{MA}}$ is ultimately a *host* that executes *guest* programs. Guest programs will usually be compiled with the *Nexus zkVM runtime*. This runtime, `nexus-rt`, is a mixed assembly/Rust program into which the guest program is linked and which provides the guest program with a suite of helpful macros and methods. Although the $\mathsf{zkVM}_{\mathsf{MA}}$ does not strictly require use of this runtime, its use greatly eases program development by ensuring compliance with the modified

Harvard architecture. The runtime does so by abstracting away the most idiosyncratic features of the $\mathsf{zkVM}_{\mathsf{MA}}$, such as its environment calls and its two distinct input interfaces: the public input segment containing x and the private input tapes containing y. The inputs x and y are separated to distinguish between public and private information when executing the full, proving zkVM. Since x is part of the public initialization of the $\mathsf{zkVM}_{\mathsf{MA}}$ and must be known to the verifier, we incorporate it into (its own segment of) the memory of the machine. Conversely, y is kept external in the environment of the $\mathsf{zkVM}_{\mathsf{MA}}$, and must be accessed by the guest program via an environment call.

A (guest) program is a sequence of RISC-V instructions. From a practical perspective, we envision P being given to $\mathsf{zkVM}_{\mathsf{MA}}$ encoded as an ELF file and then cfg, x, and y being given to P on invocation, resulting in the output z. Each instruction is specified via an opcode and takes up to three arguments, one of which can be an immediate value. The opcode and possible arguments are:

- opcode is a 7-bit string defining the instruction;
- func3 and func7 are optional bits that further specify the instruction;
- rd is a register selector specifying the destination register;
- rs1 is a register selector specifying the first operand;
- rs2 is a register selector specifying the second operand;
- imm is an immediate value (5, 12 or 20 bits depending on the opcode).

Each instruction is encoded as a 32-bit-long string, starting with 7-bit-long opcode string, followed by an encoding of the arguments, whose format varies with the instruction type. At initialization, all the general-purpose registers are set to 0. The program counter pc is set to the entrypoint of the binary being executed as specified in its ELF representation. The first instruction to be executed will be the one stored at that position in the program memory. The program counter pc is always advanced by 4 bytes after the execution of each instruction, unless the instruction itself sets the value of pc.

2.2 Execution model

2.2.1 Two-pass tracing

Following Jolt, the Nexus zkVM is designed around a 'only prove what you use' memory architecture, where unused memory does not need to be proven. As a trade-off, and again like Jolt, the zkVM requires that the amounts of memory used by most of the program segments (all but the heap) must be known before execution — even though the sizes of the stack and output are often execution-dependent. To avoid this chicken-and-egg problem, the $\mathsf{zkVM}_{\mathsf{MA}}$ operates on a two-pass tracing architecture: the program is first executed in a (mostly) traditional Harvard architecture, and statistics are kept as to the resultant memory usage. The program P is then executed again using the same x, y in a modified Harvard architecture with a fixed-memory organization determined from the statistics of the first execution, which is more conducive to proving.

Formally, we consider only the second of these passes to be the $\mathsf{zkVM}_{\mathsf{MA}}$ that defines what a 'correct' execution is. The first pass is considered a usability optimization, but we describe it as well throughout this section for completeness. The first pass can even be skipped and a fixed memory model can be used from the beginning, which may also be useful when the execution-dependent information about memory usage inherent in the 'only prove what you use' model constitutes an unacceptable privacy leakage that relegates the machine from being a zkVM into just a so-called 'succinctVM'.

2.2.2 Execution environment

Through environment calls (*i.e.*, the RISC-V `ecall` instruction), the executed program can interact with the execution environment. The $\mathsf{zkVM}_{\mathsf{MA}}$ supports six environment calls. Two of these calls are used for debugging (logging) and optimization (cycle counting), and these are no-ops during the second, proven tracing. The other four, `OverwriteStackPointer`, `OverwriteHeapPointer`, `ReadFromPrivateInput`, and `Exit`, functionally affect the traced program execution. The first pair of these calls, `OverwriteStackPointer` and `OverwriteHeapPointer`, are used by the runtime to move the heap and stack pointers to their reserved memory segments (see Section 2.3) before control is handed over to the user-supplied guest program. The other two, `ReadFromPrivateInput` and `Exit`, are more effectual to the result (z) of the execution itself. As the name indicates, `ReadFromPrivateInput` reads a byte of the private input y off the private input tape into the registers. These private input bytes are otherwise unchecked — like all other register values they will form part of the private witness, and the prover does not otherwise constrain them.

Meanwhile, `Exit` is used to immediately halt the program's execution. Although `Exit` takes an input for debugging purposes during the first tracing pass, this value is ignored in proving: instead, the contents of the exit code and public output memory segments at the time the exit call is made form the output of the program. Exit code `0` is used by the runtime to indicate a successful execution, while code `1` is used to indicate a panic thrown by the program. The runtime also exposes a convenience function that allows users to have guest programs exit with a supplied exit code, analogous to the exit system calls (like Rust's `std::process::exit(code: i32) -> !`) provided by most programming languages. It is important to note that a non-zero exit code indicates only a failure of the guest program, and the zkVM will prove such executions the same as it proves ones that halt successfully. This allows the zkVM to be used to prove faulty, buggy, or otherwise dangerous executions of programs, enabling applications of zero-knowledge verifiable computation to, *e.g.*, coordinated disclosure of security vulnerabilities as foreseen by the DARPA SIEVE program [SIE21].

2.2.3 Precompiles

In addition to the RISC-V RV32I instruction set, the $\mathsf{zkVM}_{\mathsf{MA}}$ is also designed to support precompiles, which are custom instructions that implement more complex functionality such as cryptographic hash functions or elliptic curve operations. The runtime supports defining and linking in precompiles within a compiled ELF binary, as well as integrating with an extensibility hardpoint exposed by zkVM that invokes the precompile library to both (a) execute the computation during tracing; and (b) provide suitable constraints for use to prove the correctness of the instruction evaluation.

These custom instructions are generated by the runtime using an LLVM directive (`.insn`), and so are treated as first-class instructions within $\mathsf{zkVM}_{\mathsf{MA}}$ for any execution they are linked in for, rather than being invoked via environment call. Precompiles offer exceptional performance benefits when the cost of constraining their outputs can be expressed much more simply than as the sum of constraints for the (usually long) list of instructions that computing their output natively would involve.

2.3 Memory layout

Fig. 1 shows the memory layout of the $\mathsf{zkVM}_{\mathsf{MA}}$ for both the first *(a)* and second *(b)* passes. Each memory has three attributes: in which address space it exists, with what permission structure (read-write, read-only, write-only, or no access), and whether it is of fixed or variable size. For the first pass, the organization of the memory is a mostly traditional Harvard architecture, with five distinct address spaces: (i) the cpu registers, (ii) the length (is) and content of the public input, (iii) the error code (ec) and content of the public output, (iv) the associated data, and (v) the RAM containing both the program and data segments (including the stack and heap). Other than the joining of the program and data memories this forms a relatively traditional Harvard architecture. Each of these memory segments is discussed in greater detail in the remainder of the section. In terms of permissions and sizes, (i) is read-write and fixed, (ii) is read-only and fixed, (iii) is write-only and variable, (iv) is no-access and fixed, and (v) is variable and mixed read-only (for static global variables) and read-write (for the remaining global variables and the entire data memory). Further, for (v) the guest program itself has no access to its instructions: they can only be accessed by the CPU.

For the second tracing pass, the memories are combined into a fixed-size, linear memory layout with one single, unified address space ranging over $\left[0, 2^{32} - 1\right]$. As this is the memory layout used in tracing, the $\mathsf{zkVM}_{\mathsf{MA}}$ is best understood as a modified Harvard architecture, as the permission

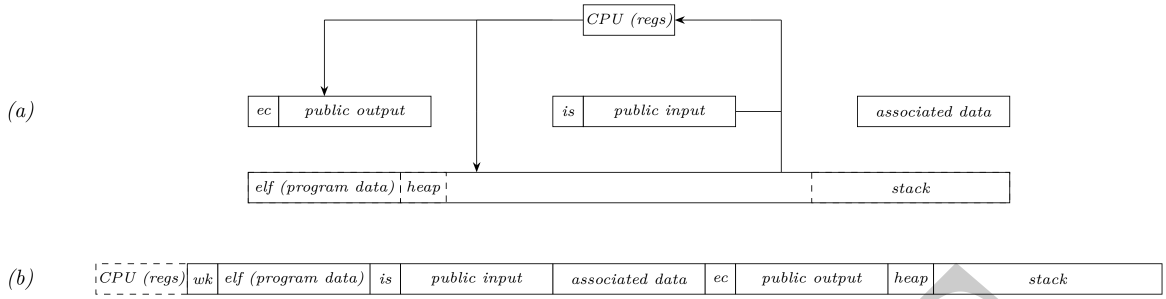

Figure 1: The memory architectures of the two-pass model. During the first Harvard pass shown in *(a)*, the memory is split into a variable-sized read-write RAM containing the program, heap, and stack, a read-only public input (prefixed by a word *is* containing the input size), a write-only public output (prefixed by a word *ec* to where the exit code is written), and a no-access associated data section. In the modified Harvard pass shown in *(b)*, the memory segments are now unified into a single linear fixed-size address space, albeit with the same reading and writing permissions structure (omitted from the figure). A space is reserved for the CPU registers at the beginning of this address space, but individual provers can choose whether to identify the registers with those addresses or maintain them in a separate namespace (such as the `x0`-`x31` naming used by the RISC-V spec).

structures on the segments are maintained but they no longer exist in distinct memories. Also, a small additional read-only 8-byte segment containing well-known (*wk*) location pointers is introduced to enable the runtime to successfully access the now relocated public input and output segments.

2.3.1 Registers

The zkVM$_{\mathsf{MA}}$ has 32 registers that hold 32-bit word values. The registers are addressed by 5-bit register selectors $\{\texttt{x0}, \ldots, \texttt{x31}\}$. We use the convention that the variables rs1 and rs2 refer to 'source' register selectors for values that are read in an instruction but left unchanged, while rd is a 'destination' register selector referring to a value that may be changed. We write $R[\ ]$ for the array of current register values, *e.g.*, $R[\mathrm{rd}]$ refers to the value of the register indicated by the 5-bit register selector rd. Register `x0` always holds the value 0, *i.e.*, if an instruction updates $R[\mathrm{rd} = \texttt{x0}]$ to a non-zero word the instruction will go through but the register will immediately be reset to $R[\texttt{x0}] = 0$.

When tracing the program the zkVM$_{\mathsf{MA}}$ stores the state of the registers separate from its record of memory operations. However, it also reserves the addresses `0x00` through `0x7F`, so that prover integrations are able to identify the registers with those addresses should the prover need to consider the entire state of the machine to lie within a single address space (as, *e.g.*, does Jolt [AST24]). For example, a prover might treat $R[\texttt{x1}]$ as being stored in the word beginning at `0x04`. At present however this is unused – the zkVM treats the registers as existing in an independent namespace, and the first 0x80 addresses as containing zero-initialized, non-writeable memory.

2.3.2 Well-known location pointers

When tracing, the zkVM$_{\mathsf{MA}}$ also reserves two words of memory, the first from `0x80` to `0x83` and the second from `0x84` to `0x87`, which contain pointers to other memory locations for use by the runtime to manage input and output handling. These pointers existing in a well-known location for the runtime to access enables the zkVM$_{\mathsf{MA}}$ to dynamically situate the input and output memory segments within the unified, linear architecture while still allowing the guest program easy access to their contents, as further discussed below. The pointers contained within these two memory locations are fixed by the zkVM$_{\mathsf{MA}}$ during initialization and can be considered part of the configuration cfg of the zkVM.

2.3.3 Program memory

The $\mathsf{zkVM}_\mathsf{MA}$ executes a program P encoded in an ELF binary. The core of such a binary is the program consisting of a sequence of instructions and supporting read-only data (such as that contained in the `.rodata` segment). Before the $\mathsf{zkVM}_\mathsf{MA}$ executes P, this data must be loaded into a read-only segment of program memory, and after that remains unchanged during execution. All instructions are encoded as 32-bit words and addressed via a 32-bit program counter pc. The program memory is byte-addressable. However, since each opcode is 32-bits long, the program memory enforces 4-byte-memory alignment and raises an instruction-address-misaligned exception whenever this condition is not satisfied. We write $P[\ ]$ for the array of program instructions. The $\mathsf{zkVM}_\mathsf{MA}$ will start executing P at the entrypoint specified in the ELF, *i.e.*, if the entrypoint is 0 then the first instruction to be executed is $P[0]$. The program must have at least 1 instruction and its size $|P|$ cannot exceed 2^{32}. If a program counter $\mathrm{pc} \geq |P|$ is used, the $\mathsf{zkVM}_\mathsf{MA}$ will raise an exception. The $\mathsf{zkVM}_\mathsf{MA}$ halts execution when either pc reaches an unimplemented instruction or when an exit environment call is made.

Additionally, the binary may contain read-write segments, such as the `.bss` and `.data` segments commonly used by programming languages to store global variables. Being writable, these segments must also be private over the course of the execution post-initialization. As such, the $\mathsf{zkVM}_\mathsf{MA}$ functionally treats these segments as a small part of the RAM that is non-contiguous with it, but with additional constraints to guarantee they are initialized as specified in the binary. In order to simplify management of the 'dual-nature' of these segments as both part of the program but also writeable, the $\mathsf{zkVM}_\mathsf{MA}$ keeps the program memory and data memory in the same address space during the first-pass tracing, but with distinct permissions for the writeable vs. read-only components. Those permissions are maintained in the unified address space of the linear memory model used in the second pass.

2.3.4 Public input

The public input segment $PI[\ ]$ contains an input of length n bytes, where $PI[k]$ for $k \in [0, n)$ stores the kth byte of the public input x. It is prefaced by a four-byte (one 32-bit word) segment $IS[\ ]$, that stores n so that the guest program can determine the length of the input made available to it. To read the kth byte of the input the runtime loads $k' = k + \texttt{offset}$ into a register $R[\mathrm{rs1}]$, and then invokes a custom instruction `rin` rd rs1. During the first tracing pass, $\texttt{offset} = 0$ so $k' = k$, and `rin` is routed by the Harvard architecture to the independent PI segment, setting $R[\mathrm{rd}] = PI[R[\mathrm{rs1}]] = PI[k'] = PI[k]$ as needed. During the second tracing pass – which is the one proven – `rin` is treated as a pseudoinstruction equivalent to `lb`, and the offset is loaded from the first word of the well-known segment (addresses `0x80` through `0x83` in the unified address space of the linear memory model). So if, *e.g.*, in the unified address space the contents of PI begin at address `p`, then the $\mathsf{zkVM}_\mathsf{MA}$ will set $\texttt{offset} = \texttt{p}$ and the ensuing load at $k' = k + \texttt{offset} = k + \texttt{p}$ will be from the memory location containing the kth byte of the input.

During both tracing passes, the contents of PI and IS are treated as read-only – the $\mathsf{zkVM}_\mathsf{MA}$ will halt if an attempt is made by the guest program to write to those memory segments. When proving with the zkVM, in the memory checking the prover constrains the input to be the same at the beginning and end of the execution (see Section 7), which is a weaker guarantee but sufficient for preventing a malicious prover from breaking the memory consistency. Moreover, during verification the values those constraints refer to (that is, the public input and its length) are provided to the circuit by the verifier, and so the constraints will not be satisfied if a malicious prover uses an input x when proving but claims to the verifier to have used $x' \neq x$ as the input instead.

2.3.5 Associated data

For many zkVM use cases it can be useful to be able to bind arbitrary contextual information about an execution or the context of the proving into the proof itself. For example, from the perspective of the zkVM the program is just a compiled binary. By incorporating the hash of the program as originally written in a high-level language – such as Rust or Python – the proof can carry a reference to that code for use by the verifier, such as for auditing its functional correctness or the correctness of its compilation. In the particular case where the program forms a standalone software package, such as a Cargo crate or Python wheel, then binding in the hash of that package can even relate the proof to the broader software ecosystem.

To support binding arbitrary external information into the proof, the $\mathsf{zkVM}_{\mathsf{MA}}$ contains an *associated data* memory segment that the prover can populate with an arbitrary bytestring. This segment is no-access within the Harvard architecture – it can neither be written to nor read from during an execution. But, the verifier otherwise treats it as a public input segment with checked contents that can then be used post-verification for application-focused infrastructure built on top of the proof, such as the aforementioned auditing. We place the associated data before the public output and exit code, so that the memory regions after it form a contiguous space of writeable segments.

2.3.6 Public output and exit code

In principle, the public output and exit code work in much the same way as the public input, except the relevant custom instruction is `wou` rs1 rs2 – writing the content of $R[\text{rs2}]$ into the $R[\text{rs1}]$-th byte of the output and interpreted on the second pass as `sb` – and the offset is loaded from the second word of the well-known segment (addresses `0x84` through `0x87` in the unified address space of the linear memory model). Otherwise, the most significant difference is that the single word exit code segment $EC[\,]$ and the public output segment $PO[\,]$ are write-only, rather than read-only. During the first tracing pass the public output segment can grow arbitrarily (up to addressing limits) to support additional written output. The length of the resultant output is then reserved ahead of time for the memory segment for the second and proven tracing pass.

2.3.7 Data memory

The $\mathsf{zkVM}_{\mathsf{MA}}$ includes a RAM, denoted $M[\,]$. The RAM contains bytes indexed by 32-bit addresses such that $M[\text{addr}]$ refers to the current byte value at address addr. The data memory size $|M|$ can be at most 2^{32} so that the entire memory is addressable. Addresses are reduced modulo $|M|$ to handle overflows, i.e., $M[\text{addr}] = M[\text{addr} \bmod |M|]$. Due to this reduction, for consistency the same $|M|$ must be used for both evaluation passes, but as $|M|$ is only explicitly determined at the end of the first pass based on its maximum memory usage, this is guaranteed by design. Instructions can read from the data memory, write to the data memory, or leave the data memory untouched. The primary use of the data memory is to store the stack and the heap. During the first tracing pass, its size is variable and the stack and heap grow towards each other, as is standard. During the second tracing pass the stack and heap still grow towards each other, but are given fixed-size segments within which to do so. As a consequence, the size of the data memory is limited to only what is needed, enabling quicker proving and smaller proofs in the zkVM as only memory that is used is 'paid for' by needing to prove its contents.

2.4 Basic instruction set and binary encoding

The Nexus Virtual Machine (NVM) instruction set extends the RISC-V `RV32I` Instruction Set Architecture by introducing an `RV32Nexus` extension that captures new instructions (especially future precompiles). Note that `fence` has no effect since our machine only has one CPU core (a.k.a. Hart in RISC-V terminology). The `ebreak` instruction is unsupported by the $\mathsf{zkVM}_{\mathsf{MA}}$, but the zkVM is nonetheless able to prove traces that utilize the instruction as its constraints are nearly identical to that of `ecall`. The `unimp` instruction is supported by neither the $\mathsf{zkVM}_{\mathsf{MA}}$ nor the zkVM as a whole, though for completeness the decoding circuits do include some handling for it. The NVM instruction set is summarized in Table 1 and the binary encoding of all instructions specified in Table 2.

The Nexus VM will enforce 1-byte alignment for the data memory and 4-byte alignments for the program memory. In particular, the program counter must be a multiple of 4. For instructions which

Table 1: Summary of the Nexus Virtual Machine Instruction Set, where operations are mod 2^{32} and sext indicates a sign extension. In an arithmetic shift, the sign bit is copied into the vacated upper bits. In a logical shift, zero is copied into the vacated bits.

Instruction mnemonic	Arguments	Description of functionality
`nop`		no operation, implemented as (`addi x0 x0 0`)
`lui`	rd i	loads i into top 20 bits of R[rd], fills lower 12 bits with 0's
`auipc`	rd i	loads i into top 20 bits of R[rd], fills lower 12 bits with 0's, adds pc to R[rd]
`jal`	rd i	stores $pc + 4$ into R[rd], jumps to $pc + \text{sext}(i)$
`jalr`	rd rs1 i	stores $pc + 4$ into R[rd], jumps to $(R[\text{rs1}] + \text{sext}(i))$ \& `0xFFFFFFFE`
`ecall`		system call (see Table 3)
`ebreak`		system call (see Table 3)
`fence`	$pr\ sc\ fm$	mapped to `nop`
`unimp`		the machine halts when encountering this instruction; pc is not updated
`beq`	rs1 rs2 i	branches to $pc + \text{sext}(i)$ if $(R[\text{rs1}] = R[\text{rs2}])$
`bne`	rs1 rs2 i	branches to $pc + \text{sext}(i)$ if $(R[\text{rs1}] \neq R[\text{rs2}])$
`blt`	rs1 rs2 i	branches to $pc + \text{sext}(i)$ if $(R[\text{rs1}] < R[\text{rs2}])$ (signed comparison)
`bge`	rs1 rs2 i	branches to $pc + \text{sext}(i)$ if $(R[\text{rs1}] \geq R[\text{rs2}])$ (signed comparison)
`bltu`	rs1 rs2 i	branches to $pc + \text{sext}(i)$ if $(R[\text{rs1}] < R[\text{rs2}])$ (unsigned comparison)
`bgeu`	rs1 rs2 i	branches to $pc + \text{sext}(i)$ if $(R[\text{rs1}] \geq R[\text{rs2}])$ (unsigned comparison)
`lb`	rd rs1 i	loads the sign extension of the byte at $M[R[\text{rs1}] + \text{sext}(i)]$ into R[rd]
`lh`	rd rs1 i	loads the sign extension of the half-word at $M[R[\text{rs1}] + \text{sext}(i)]$ into R[rd]
`lw`	rd rs1 i	loads the word at $M[R[\text{rs1}] + \text{sext}(i)]$ into R[rd]
`lbu`	rd rs1 i	loads the zero extension of the byte at $M[R[\text{rs1}] + \text{sext}(i)]$ into R[rd]
`lhu`	rd rs1 i	loads the zero extension of the half-word at $M[R[\text{rs1}] + \text{sext}(i)]$ into R[rd]
`sb`	rs1 rs2 i	stores the first byte of R[rs2] at $M[R[\text{rs1}] + \text{sext}(i)]$
`sh`	rs1 rs2 i	stores the first half-word of R[rs2] at $M[R[\text{rs1}] + \text{sext}(i)]$
`sw`	rs1 rs2 i	stores R[rs2] at $M[R[\text{rs1}] + \text{sext}(i)]$
`addi`	rd rs1 i	sets R[rd] to $R[\text{rs1}] + \text{sext}(i)$
`slti`	rd rs1 i	sets $R[\text{rd}] = 1$ if $(R[\text{rs1}] < \text{sext}(i))$ and 0 otherwise (signed comparison)
`sltiu`	rd rs1 i	sets $R[\text{rd}] = 1$ if $(R[\text{rs1}] < \text{sext}(i))$ and 0 otherwise (unsigned comparison)
`xori`	rd rs1 i	sets R[rd] to the bitwise XOR of R[rs1] and sext(i)
`ori`	rd rs1 i	sets R[rd] to the bitwise OR of R[rs1] and sext(i)
`andi`	rd rs1 i	sets R[rd] to the bitwise AND of R[rs1] and sext(i)
`slli`	rd rs1 i	sets R[rd] to $R[\text{rs1}] \ll (i$ \& `0x0000001F`) (logical shift)
`srli`	rd rs1 i	sets R[rd] to $R[\text{rs1}] \gg (i$ \& `0x0000001F`) (logical shift)
`srai`	rd rs1 i	sets R[rd] to $R[\text{rs1}] \gg (i$ \& `0x0000001F`) (arithmetic shift)
`add`	rd rs1 rs2	sets R[rd] to $R[\text{rs1}] + R[\text{rs2}]$
`sub`	rd rs1 rs2	sets R[rd] to $R[\text{rs1}] - R[\text{rs2}]$
`sll`	rd rs1 rs2	sets R[rd] to $R[\text{rs1}] \ll (R[\text{rs2}]$ \& `0x0000001F`) (logical shift)
`slt`	rd rs1 rs2	sets $R[\text{rd}] = 1$ if $(R[\text{rs1}] < R[\text{rs2}])$ and 0 otherwise (signed comparison)
`sltu`	rd rs1 rs2	sets $R[\text{rd}] = 1$ if $(R[\text{rs1}] < R[\text{rs2}])$ and 0 otherwise (unsigned comparison)
`xor`	rd rs1 rs2	sets R[rd] to the bitwise XOR of R[rs1] and R[rs2]
`srl`	rd rs1 rs2	sets R[rd] to $R[\text{rs1}] \gg (R[\text{rs2}]$ \& `0x0000001F`) (logical shift)
`sra`	rd rs1 rs2	sets R[rd] to $R[\text{rs1}] \gg (R[\text{rs2}]$ \& `0x0000001F`) (arithmetic shift)
`or`	rd rs1 rs2	sets R[rd] to the bitwise OR of R[rs1] and R[rs2]
`and`	rd rs1 rs2	sets R[rd] to the bitwise AND of R[rs1] and R[rs2]

Table 2: Binary Encoding of Nexus Virtual Machine Instructions, where $\langle d\rangle$, $\langle s_1\rangle$, and $\langle s_2\rangle$ denote respectively the binary representation of the 5-bit register selectors rd, rs1, rs2, and $\langle i\rangle[a{:}b]$ denotes the binary representation of the bits a through b of the immediate value i.

<table>
<tr><th>Instruction
mnemonic</th><th colspan="3">Arguments</th><th colspan="8">Binary Encodings (Bits 31–0)</th></tr>
<tr><td>lui</td><td>rd</td><td>i</td><td></td><td colspan="6">$\langle i\rangle[31:12]$</td><td>$\langle d\rangle$</td><td>0110111</td></tr>
<tr><td>auipc</td><td>rd</td><td>i</td><td></td><td colspan="6">$\langle i\rangle[31:12]$</td><td>$\langle d\rangle$</td><td>0010111</td></tr>
<tr><td>jal</td><td>rd</td><td>i</td><td></td><td colspan="6">$\langle i\rangle[20|10:1|11|19:12]$</td><td>$\langle d\rangle$</td><td>1101111</td></tr>
<tr><td>jalr</td><td>rd</td><td>rs1</td><td>i</td><td colspan="4">$\langle i\rangle[11:0]$</td><td>$\langle s_1\rangle$</td><td>000</td><td>$\langle d\rangle$</td><td>1100111</td></tr>
<tr><td>ecall</td><td></td><td></td><td></td><td colspan="4">000000000000</td><td>00000</td><td>000</td><td>00000</td><td>1110011</td></tr>
<tr><td>ebreak</td><td></td><td></td><td></td><td colspan="4">000000000001</td><td>00000</td><td>000</td><td>00000</td><td>1110011</td></tr>
<tr><td>fence</td><td>pr</td><td>sc</td><td>fm</td><td>fm</td><td colspan="2">sc</td><td>pr</td><td>00000</td><td>000</td><td>00000</td><td>0001111</td></tr>
<tr><td>unimp</td><td></td><td></td><td></td><td colspan="4">000000000011</td><td>00000</td><td>001</td><td>00000</td><td>1110011</td></tr>
<tr><td>beq</td><td>rs1</td><td>rs2</td><td>i</td><td colspan="2">$\langle i\rangle[12|10:5]$</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>000</td><td>$\langle i\rangle[4:1|11]$</td><td>1100011</td></tr>
<tr><td>bne</td><td>rs1</td><td>rs2</td><td>i</td><td colspan="2">$\langle i\rangle[12|10:5]$</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>001</td><td>$\langle i\rangle[4:1|11]$</td><td>1100011</td></tr>
<tr><td>blt</td><td>rs1</td><td>rs2</td><td>i</td><td colspan="2">$\langle i\rangle[12|10:5]$</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>100</td><td>$\langle i\rangle[4:1|11]$</td><td>1100011</td></tr>
<tr><td>bge</td><td>rs1</td><td>rs2</td><td>i</td><td colspan="2">$\langle i\rangle[12|10:5]$</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>101</td><td>$\langle i\rangle[4:1|11]$</td><td>1100011</td></tr>
<tr><td>bltu</td><td>rs1</td><td>rs2</td><td>i</td><td colspan="2">$\langle i\rangle[12|10:5]$</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>110</td><td>$\langle i\rangle[4:1|11]$</td><td>1100011</td></tr>
<tr><td>bgeu</td><td>rs1</td><td>rs2</td><td>i</td><td colspan="2">$\langle i\rangle[12|10:5]$</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>111</td><td>$\langle i\rangle[4:1|11]$</td><td>1100011</td></tr>
<tr><td>lb</td><td>rd</td><td>rs1</td><td>i</td><td colspan="4">$\langle i\rangle[11:0]$</td><td>$\langle s_1\rangle$</td><td>000</td><td>$\langle d\rangle$</td><td>0000011</td></tr>
<tr><td>lh</td><td>rd</td><td>rs1</td><td>i</td><td colspan="4">$\langle i\rangle[11:0]$</td><td>$\langle s_1\rangle$</td><td>001</td><td>$\langle d\rangle$</td><td>0000011</td></tr>
<tr><td>lw</td><td>rd</td><td>rs1</td><td>i</td><td colspan="4">$\langle i\rangle[11:0]$</td><td>$\langle s_1\rangle$</td><td>010</td><td>$\langle d\rangle$</td><td>0000011</td></tr>
<tr><td>lbu</td><td>rd</td><td>rs1</td><td>i</td><td colspan="4">$\langle i\rangle[11:0]$</td><td>$\langle s_1\rangle$</td><td>100</td><td>$\langle d\rangle$</td><td>0000011</td></tr>
<tr><td>lhu</td><td>rd</td><td>rs1</td><td>i</td><td colspan="4">$\langle i\rangle[11:0]$</td><td>$\langle s_1\rangle$</td><td>101</td><td>$\langle d\rangle$</td><td>0000011</td></tr>
<tr><td>sb</td><td>rs1</td><td>rs2</td><td>i</td><td colspan="2">$\langle i\rangle[11:5]$</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>000</td><td>$\langle i\rangle[4:0]$</td><td>0100011</td></tr>
<tr><td>sh</td><td>rs1</td><td>rs2</td><td>i</td><td colspan="2">$\langle i\rangle[11:5]$</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>001</td><td>$\langle i\rangle[4:0]$</td><td>0100011</td></tr>
<tr><td>sw</td><td>rs1</td><td>rs2</td><td>i</td><td colspan="2">$\langle i\rangle[11:5]$</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>010</td><td>$\langle i\rangle[4:0]$</td><td>0100011</td></tr>
<tr><td>addi</td><td>rd</td><td>rs1</td><td>i</td><td colspan="4">$\langle i\rangle[11:0]$</td><td>$\langle s_1\rangle$</td><td>000</td><td>$\langle d\rangle$</td><td>0010011</td></tr>
<tr><td>slti</td><td>rd</td><td>rs1</td><td>i</td><td colspan="4">$\langle i\rangle[11:0]$</td><td>$\langle s_1\rangle$</td><td>010</td><td>$\langle d\rangle$</td><td>0010011</td></tr>
<tr><td>sltiu</td><td>rd</td><td>rs1</td><td>i</td><td colspan="4">$\langle i\rangle[11:0]$</td><td>$\langle s_1\rangle$</td><td>011</td><td>$\langle d\rangle$</td><td>0010011</td></tr>
<tr><td>xori</td><td>rd</td><td>rs1</td><td>i</td><td colspan="4">$\langle i\rangle[11:0]$</td><td>$\langle s_1\rangle$</td><td>100</td><td>$\langle d\rangle$</td><td>0010011</td></tr>
<tr><td>ori</td><td>rd</td><td>rs1</td><td>i</td><td colspan="4">$\langle i\rangle[11:0]$</td><td>$\langle s_1\rangle$</td><td>110</td><td>$\langle d\rangle$</td><td>0010011</td></tr>
<tr><td>andi</td><td>rd</td><td>rs1</td><td>i</td><td colspan="4">$\langle i\rangle[11:0]$</td><td>$\langle s_1\rangle$</td><td>111</td><td>$\langle d\rangle$</td><td>0010011</td></tr>
<tr><td>slli</td><td>rd</td><td>rs1</td><td>i</td><td colspan="2">0000000</td><td colspan="2">$\langle i\rangle[4:0]$</td><td>$\langle s_1\rangle$</td><td>001</td><td>$\langle d\rangle$</td><td>0010011</td></tr>
<tr><td>srli</td><td>rd</td><td>rs1</td><td>i</td><td colspan="2">0000000</td><td colspan="2">$\langle i\rangle[4:0]$</td><td>$\langle s_1\rangle$</td><td>101</td><td>$\langle d\rangle$</td><td>0010011</td></tr>
<tr><td>srai</td><td>rd</td><td>rs1</td><td>i</td><td colspan="2">0100000</td><td colspan="2">$\langle i\rangle[4:0]$</td><td>$\langle s_1\rangle$</td><td>101</td><td>$\langle d\rangle$</td><td>0010011</td></tr>
<tr><td>add</td><td>rd</td><td>rs1</td><td>rs2</td><td colspan="2">0000000</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>000</td><td>$\langle d\rangle$</td><td>0110011</td></tr>
<tr><td>sub</td><td>rd</td><td>rs1</td><td>rs2</td><td colspan="2">0100000</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>000</td><td>$\langle d\rangle$</td><td>0110011</td></tr>
<tr><td>sll</td><td>rd</td><td>rs1</td><td>rs2</td><td colspan="2">0000000</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>001</td><td>$\langle d\rangle$</td><td>0110011</td></tr>
<tr><td>slt</td><td>rd</td><td>rs1</td><td>rs2</td><td colspan="2">0000000</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>010</td><td>$\langle d\rangle$</td><td>0110011</td></tr>
<tr><td>sltu</td><td>rd</td><td>rs1</td><td>rs2</td><td colspan="2">0000000</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>011</td><td>$\langle d\rangle$</td><td>0110011</td></tr>
<tr><td>xor</td><td>rd</td><td>rs1</td><td>rs2</td><td colspan="2">0000000</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>100</td><td>$\langle d\rangle$</td><td>0110011</td></tr>
<tr><td>srl</td><td>rd</td><td>rs1</td><td>rs2</td><td colspan="2">0000000</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>101</td><td>$\langle d\rangle$</td><td>0110011</td></tr>
<tr><td>sra</td><td>rd</td><td>rs1</td><td>rs2</td><td colspan="2">0100000</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>101</td><td>$\langle d\rangle$</td><td>0110011</td></tr>
<tr><td>or</td><td>rd</td><td>rs1</td><td>rs2</td><td colspan="2">0000000</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>110</td><td>$\langle d\rangle$</td><td>0110011</td></tr>
<tr><td>and</td><td>rd</td><td>rs1</td><td>rs2</td><td colspan="2">0000000</td><td colspan="2">$\langle s_2\rangle$</td><td>$\langle s_1\rangle$</td><td>111</td><td>$\langle d\rangle$</td><td>0110011</td></tr>
</table>

operate over half-words or words, we will load one byte at a time and reassemble them into half-words or words. Nevertheless, as indicated in Table 1, load-half (`lh`) and load-word (`lw`) instructions assume respectively 2-byte and 4-byte alignments and raise an exception whenever this condition is not met.

If the opcode is invalid, we will raise an exception. If there exists an ISA overflow in address calculation, we will also throw an exception.

As it is standard for RISC-V programs, a program halts whenever the `unimp` instruction is loaded into the CPU. The output of the program can then be read from the dedicated memory location used for outputs.

Table 3: Behavior of the `ecall` and `ebreak` instructions in the Nexus Virtual Machine.

$R[\texttt{x17}]$ value	Description of the `ecall` / `ebreak` functionality	pc update
`0x200`	system call to write to the memory, used for debugging	$\text{pc} \leftarrow \text{pc} + 4$
`0x201`	system call to halt the virtual machine, similar to `unimp`	pc is not updated
`0x400`	system call to read from private input, loads a 32-bit value onto $R[\texttt{x10}]$	$\text{pc} \leftarrow \text{pc} + 4$
`0x401`	system call to obtain the current cycle count	$\text{pc} \leftarrow \text{pc} + 4$
`0x402`	system call to overwrite the stack pointer, loads a 32-bit value onto $R[\texttt{x2}]$	$\text{pc} \leftarrow \text{pc} + 4$
`0x403`	system call to overwrite the heap pointer, loads a 32-bit value onto $R[\texttt{x10}]$	$\text{pc} \leftarrow \text{pc} + 4$

2.5 Instruction encoding

In order to abstract away the details of the encoding of virtual machine instructions, zkVMs often adopt an abstract representation of these instructions inside the prover. For instance, in Jolt [AST24] each instruction is viewed as a 5-tuple $(\text{opcode}, \text{rs1}, \text{rs2}, \text{rd}, \text{imm})$ denoting an operation code, two source registers, a destination register, and an immediate value. Although they may lack complete formal specifications, the code and documentation of some other zkVM projects show similar encoding approaches, demonstrating it to be a common arithmetization pattern.[1]

Our zkVM follows the same pattern and represents each instruction by a tuple of 5 field elements, denoting the instruction opcode opcode, three possible operands $(\text{op-a}, \text{op-b}, \text{op-c})$, and a flag imm-c indicating whether the third operand is an immediate value or register address.

The following table shows how to map Nexus Virtual Machine instructions to the above encoding.

[1] For example, in Valida (https://lita.gitbook.io/lita-documentation/architecture/valida-zk-vm/technical-design-vm) each instruction is encoded as six field elements, denoting the instruction operation code, three possible operands, and two flags. These flags indicate respectively whether the second and third operands are immediate values or offsets.

Table 4: Representation of Nexus Virtual Machine Instructions in the format (opcode, op-a, op-b, op-c, imm-c).

NVM Instruction mnemonic	Arguments	Instruction Encoding				
		opcode	op-a	op-b	op-c	imm-c
`nop`		mapped to (`addi x0 x0 0`)				
`lui`	rd *i*	`LUI`	rd	0	*i*	1
`auipc`	rd *i*	`AUIPC`	rd	0	*i*	1
`jal`	rd *i*	`JAL`	rd	0	*i*	1
`jalr`	rd rs1 *i*	`JALR`	rd	rs1	*i*	1
`ecall`		`ECALL`	0 | `x2` | `x10`	`x17`	0	1
`ebreak`		`EBREAK`	0 | `x2` | `x10`	`x17`	0	1
`fence`	*pr sc fm*	mapped to `nop`				
`unimp`		`UNIMP`	0	0	0	1
`beq`	rs1 rs2 *i*	`BEQ`	rs1	rs2	*i*	1
`bne`	rs1 rs2 *i*	`BNE`	rs1	rs2	*i*	1
`blt`	rs1 rs2 *i*	`BLT`	rs1	rs2	*i*	1
`bge`	rs1 rs2 *i*	`BGE`	rs1	rs2	*i*	1
`bltu`	rs1 rs2 *i*	`BLTU`	rs1	rs2	*i*	1
`bgeu`	rs1 rs2 *i*	`BGEU`	rs1	rs2	*i*	1
`lb`	rd rs1 *i*	`LB`	rd	rs1	*i*	1
`lh`	rd rs1 *i*	`LH`	rd	rs1	*i*	1
`lw`	rd rs1 *i*	`LW`	rd	rs1	*i*	1
`lbu`	rd rs1 *i*	`LBU`	rd	rs1	*i*	1
`lhu`	rd rs1 *i*	`LHU`	rd	rs1	*i*	1
`sb`	rs1 rs2 *i*	`SB`	rs1	rs2	*i*	1
`sh`	rs1 rs2 *i*	`SH`	rs1	rs2	*i*	1
`sw`	rs1 rs2 *i*	`SW`	rs1	rs2	*i*	1
`addi`	rd rs1 *i*	`ADD`	rd	rs1	*i*	1
`slti`	rd rs1 *i*	`SLT`	rd	rs1	*i*	1
`sltiu`	rd rs1 *i*	`SLTU`	rd	rs1	*i*	1
`xori`	rd rs1 *i*	`XOR`	rd	rs1	*i*	1
`ori`	rd rs1 *i*	`OR`	rd	rs1	*i*	1
`andi`	rd rs1 *i*	`AND`	rd	rs1	*i*	1
`slli`	rd rs1 *i*	`SLL`	rd	rs1	*i*	1
`srli`	rd rs1 *i*	`SRL`	rd	rs1	*i*	1
`srai`	rd rs1 *i*	`SRA`	rd	rs1	*i*	1
`add`	rd rs1 rs2	`ADD`	rd	rs1	rs2	0
`sub`	rd rs1 rs2	`SUB`	rd	rs1	rs2	0
`sll`	rd rs1 rs2	`SLL`	rd	rs1	rs2	0
`slt`	rd rs1 rs2	`SLT`	rd	rs1	rs2	0
`sltu`	rd rs1 rs2	`SLTU`	rd	rs1	rs2	0
`xor`	rd rs1 rs2	`XOR`	rd	rs1	rs2	0
`srl`	rd rs1 rs2	`SRL`	rd	rs1	rs2	0
`sra`	rd rs1 rs2	`SRA`	rd	rs1	rs2	0
`or`	rd rs1 rs2	`OR`	rd	rs1	rs2	0
`and`	rd rs1 rs2	`AND`	rd	rs1	rs2	0

3 Preliminaries

In this section, we explain a few useful tools and concepts used in the context of the new zkVM.

3.1 `Stwo` underlying algebraic structures

This specification does not describe the inner details of the `Stwo` prover but we briefly recall here the finite fields used in `Stwo`, which are relevant for the arithmetization of our circuits and the specification of constraints.

Fields: There are three main fields used in the `stwo` and and `stwo-cairo` repositories for the `Stwo` prover:

`m31`: This is the base prime field $\mathbb{F}_p$ used in the `stwo` and and `stwo-cairo` repositories. The size of the field is $p = 2^{31} - 1$, which is a Mersenne prime.

`cm31`: This is the field $\mathbb{F}'$ which is a complex extension of $\mathbb{F}_p$, also denoted by $\mathbb{F}_p(i)$. Note that the complex extension is also equivalent to $\mathbb{F}_p[X]/(X^2+1)$. Further, each element in $\mathbb{F}'$ can be written as $a + ib$, with addition and subtraction performed in a straightforward component-wise manner.

Multiplication: $(a_1 + ib_1)(a_2 + ib_2) = (a_1a_2 - b_1b_2) + i(a_1b_2 + a_2b_1)$

Inverse: $(a + ib)^{-1} = (a - ib)(a^2 + b^2)^{-1}$ for $a + ib \neq 0$

The components themselves are multiplied using the underlying field multiplication of $\mathbb{F}_p$.

`qm31`: This is the field $\mathbb{F}''$, which is the quadratic extension of $\mathbb{F}'$. This can also be written as $\mathbb{F}'(\phi)$, where ϕ is the root of the equation $X^2 = 2 + i$, with $\mathbb{F}''$ equivalent to $\mathbb{F}'[X]/(X^2 - 2 - i)$. Each element of $\mathbb{F}''$ can be written as $(a + ib) + \phi(c + id)$. For simplicity, this can be written as $u + \phi v$, where $u = (a + ib)$ and $v = (c + id)$.

Multiplication: $(u_1 + \phi v_1)(u_2 + \phi v_2) = (u_1u_2 + (2 + i)v_1v_2) + \phi(u_1v_2 + u_2v_1)$

Inverse: $(u + \phi v)^{-1} = (u - \phi v)(u^2 - (2 + i)v^2)^{-1}$ for $u + \phi v \neq 0$

Note that the above operations, such as multiplication between u and v above are done as in the multiplication in $\mathbb{F}'$. Further, note that this field is of size $p^4 \approx 2^{124}$.

In the open source code, `m31` and `qm31` are often referred to as the '`BaseField`' and '`SecureField`'. Notice that `SecureField` is large enough to be used in randomized checks with negligible probability of cheating.

Circle Groups: One of the most important components in STARK proofs [BBHR18] is the ability to compute FFT over some cyclic group. While a prime field $\mathbb{F}_p$ already has a cyclic multiplicative subgroup of order $p - 1$, for efficiency reasons, we would like to work with a subgroup with an order that can be divided by a large power of two 2^k. Unfortunately, for `m31` of size $p = 2^{31} - 1$, the cyclic group is not of such a form since $p - 1 = 2(2^{30} - 1)$ giving a maximal even power 2^k with $k = 1$.

A recent work [HLP24] shows how to get around this by instead working in the 'circle group'. The main insight is that one can represent every point on this circle group as a pair of underlying field elements such that with an appropriately chosen generator, the cycle group has order of the form 2^k. For instance, with elements from `m31`, one can generate a cyclic circle group of order 2^{30}.

3.2 Offline Memory Checker

Offline memory checking allows us to keep track of memory and register read/write consistency. In this section, we briefly recall the offline memory checking problem and an efficient solution for it.

Definition 1 (Offline Memory Checking) *The offline memory checking problem relates to a user of an external untrusted memory and how they can ensure consistency between accesses without keeping a copy of the entire memory. In our case the zkVM can be seen as the user that in the execution trace sees claims about register values and memory values that are hard to verify just using local information in a single row.*

The user issues a list containing n memory accesses to the untrusted memory of size m with strictly monotonically increasing timestamps timer*. We denote by M_{Init} the initial state of memory. Each memory access is in one of the following forms:*

- $\text{Read}_{\text{M}}(\text{addr}, \text{timer}) \to \mathsf{val}$*: In this case, the untrusted memory* M *should return the memory cell value* val *stored at address* addr *at time* timer *to the user.*
- $\text{Write}_{\text{M}}(\text{addr}, \mathsf{val}, \text{timer})$*: In this case, the untrusted memory* M *is instructed to rewrite the memory cell value stored at address* addr *and time* timer *with the value* val*.*

The offline memory checker observes the communication between the end user and the untrusted memory, and after the user issues all n memory accesses, the offline checker makes a decision at the end. We require two properties from the offline memory checker:

Completeness*: If for every read operation, the untrusted memory returns the tuple last written to that location, then the checker always outputs* 1*.*

Soundness*: If the untrusted memory ever returns a value* val^* *for a memory read such that* val^* *does not equal the value* val *last written to the address* addr *(or the initial value if never written to), then the checker outputs* 0*, with overwhelming probability.*

In Section 3.2.1, we describe a simple offline memory checker, which satisfies both completeness and soundness, but not the efficiency requirement since the algorithm keeps track of the status of the running memory.

In order to make it efficient, it suffices to keep a trace digest of the memory accesses, which is cheap and can be updated at a cost that independent of the size of the entire memory. In Section 3.2.2, we describe how the Nexus zkVM does so using logarithmic derivatives aka logups [EKRN24, Hab22].

3.2.1 A simple offline memory checker

We will maintain a global counter ts, which gets incremented at every clock cycle, along with two sets, a "read set" (RS) and a "write set" (WS). The purpose of the counter ts is to stamp each memory access with a unique identifier. The purpose of RS and WS is to record a trace of the memory access pattern.

We will also augment the memory to include an additional timestamp for each cell. More specifically, we now view the memory as a vector consisting of triples of the form $(\text{addr}, \mathsf{val}, t)$ where addr is the memory address, val is the associated value, and t is a timestamp which indicates the last time the address was read or updated. The offline memory checker initializes $\mathsf{RS} = \emptyset, \mathsf{WS} = M_{\text{Init}}$ and $ts = 0$.

To deal with a read instruction at address addr, the offline memory checker performs the following steps:

1. Read the triple $(\text{addr}, \mathsf{val}, t)$ from the untrusted memory, where all three values are treated as non-deterministic advice (witness) from the untrusted memory.
2. Verify that $t \in [0, ts - 1]$
3. Add $(\text{addr}, \mathsf{val}, t)$ to RS;
4. Update the counter to $ts = ts + 1$;

5. Add $(\mathrm{addr}, \mathsf{val}, ts)$ to WS;
6. Write $(\mathrm{addr}, \mathsf{val}, ts)$ to the untrusted memory.

To deal with a write instruction which overwrites the value at address addr into val', the offline memory checker performs the following steps:

1. Read the triple $(\mathrm{addr}, \mathsf{val}, t)$ from the untrusted memory;
2. Verify that $t \in [0, ts - 1]$
3. Add $(\mathrm{addr}, \mathsf{val}, t)$ to RS;
4. Update the counter to $ts = ts + 1$;
5. Add $(\mathrm{addr}, \mathsf{val}', ts)$ to WS;
6. Write $(\mathrm{addr}, \mathsf{val}', ts)$ to the untrusted memory.

Let's denote by M_{Init} the initial state of memory with all timestamps set to 0, and M_{Final} the final state of the memory with the final timestamps. The decision predicate for the offline memory checker is:

$$\mathsf{RS} \cup M_{\mathrm{Final}} \stackrel{?}{=} \mathsf{WS} \cup M_{\mathrm{Init}}$$

In order to understand the intuition as to why this algorithm satisfy completeness and soundness, first observe that the read and write sets maintained by the offline checker preserve two important invariants: (1) Every element added to RS and WS is unique because the timestamp of that element is always set to the current global counter, which is incremented after each memory access; (2) For each key used, RS "trails" WS by exactly the last write operation. In other words, we always have $\mathsf{RS} \subseteq \mathsf{WS}$.

Let WS and RS denote the multi-sets maintained by the offline memory checker. If for every read operation, the untrusted memory returns the tuple last written to that location, then we have $\mathsf{RS} \cup M_{\mathrm{Final}} = \mathsf{WS} \cup M_{\mathrm{Init}}$ and completeness follows.

Moreover, if the untrusted memory ever returns a value val^* for a memory read such that val^* does not equal the value val last written to the address addr (or the initial value if never written to), then there does not exist any set M such that $\mathsf{RS} \cup M = \mathsf{WS} \cup M_{\mathrm{Init}}$ and soundness follows.

3.2.2 A concrete proposal based on logups

As mentioned above, the simple algorithm described above keeps track of the status of the running memory, which is inefficient. In order to make it efficient, it suffices to maintain a digest of the read and write sets instead of the full sets. The advantage is that the digests can be updated at a cost that is independent of the sizes of the sets.

In this version of the Nexus zkVM, the computation of the digest is implemented using logarithmic derivatives aka logups [EKRN24, Hab22], which we now describe.

Even though the Nexus zkVM has different categories of memory (e.g., program memory, register memory, and random access memory), this section focuses on the implementation of the algorithm for the case of random access memory. Let us for simplicity assume each row can do one of three possible memory operations.

- Load a value from the RAM. The row will have values t, a, v, t', where (if derived from an honest execution trace) t is the timestamp of the CPU cycle being executed, a is an address, v is a memory value read from address a, and t' represents the last time the address a was accessed.
- Store a value in the RAM. The row will have values t, a, v, v', t', where (if derived from an honest execution trace) t is the timestamp of the CPU cycle being executed, a is an address, v is a memory value written to address a, and t' represents the last time the address a was accessed and the prior value is v'.
- Not access the RAM.

Now, let us for simplicity assume these elements can be represented as single field elements in a finite field $\mathbb{F}$ (we deal later with the case where they have to be represented as multiple field elements) of size p. Moreover, let $\mathbb{E}$ be an extension field of size p^e, where $e \geq 3$. And let f be an injective map $f : (t, a, v) \in \mathbb{F}^3 \to \mathbb{E}$. When using the Nexus zkVM with the `Stwo` prover, $\mathbb{F}$ and $\mathbb{E}$ will correspond to `m31` and `qm31`, respectively.

The high level strategy of the proof for memory consistency is

- Let $\mathsf{WS}_{\mathsf{init}}$ be a set of tuples $(t = 0, a, v)$ specifying initial values to some of the memory addresses, and let $\mathsf{RS}_{\mathsf{final}}$ be a claimed end state of the memory specified via tuples (t, a, v), where (in the case of an honest trace) t is the last time the address was written to.
- Commit to the table representing the execution trace (or at least commit to the memory access values given above)
- Get a logup challenge $z \in \mathbb{E}$ via Fiat-Shamir on the initial and final states and the commitments to the trace
- Going through all the rows, compute the read set and write set logup sums as follows
  - Set
    $$\sigma_{\mathsf{RS}} := 0 \qquad \text{and} \qquad \sigma_{\mathsf{WS}} := 0$$
  - For a row with a read (t, a, v) on an address last touched at t' (note $t' < t$, which can be verified via local constraints on the row) increment
    $$\sigma_{\mathsf{RS}} \mathrel{+}= \frac{1}{f(t', a, v) + z} \qquad \text{and} \qquad \sigma_{\mathsf{WS}} \mathrel{+}= \frac{1}{f(t, a, v) + z}$$
  - For a row with a write (t, a, v) on an address last touched at t' having value v' (again $t' < t$ can be verified via local constraints on the row) increment
    $$\sigma_{\mathsf{RS}} \mathrel{+}= \frac{1}{f(t', a, v') + z} \qquad \text{and} \qquad \sigma_{\mathsf{WS}} \mathrel{+}= \frac{1}{f(t, a, v) + z}$$
  - For a row not touching memory, make no changes to the logup sums
  - Return $\sigma_{\mathsf{RS}}, \sigma_{\mathsf{WS}} \in \mathbb{E}$
- The proof π for correct execution will include evidence to convince the verifier that
  - For each row that some local constraints are satisfied, e.g., type checks on a, v, t, v', t' via range proofs, and for reads and writes that $t' < t$
  - The Fiat-Shamir challenge z is computed based on the initial memory and final memory and commitments
  - The logup sums are computed correctly wrt z.
- The verifier will check
  $$\sum_{(t=0,a,v)\in\mathsf{WS}_{\mathsf{init}}} \frac{1}{f(a, v, t) + z} + \sigma_{\mathsf{WS}} \;=\; \sigma_{\mathsf{RS}} + \sum_{(t,a,v)\in\mathsf{RS}_{\mathsf{final}}} \frac{1}{f(t, a, v) + z}.$$

3.3 Lookup tables for range checks

Several parts of the Nexus zkVM design require 8-bit range checks, whose goal is to check whether a given field element lies in the range $\left[0, 2^8 - 1\right]$. In order to implement 8-bit range checks, we follow the MidenVM design based on logups [Mid].

More precisely, let `x` be a trace column containing 8-bit elements which need to be range checked. In order to implement this range check, we will include two columns (`v` and `m`) to the trace containing the list of values $\mathtt{v}[i] \in \{0, \ldots, 255\}$ in the range together with their multiplicities $\mathtt{m}[i]$, where i specifies the row index. In our design, these values will be in the `m31` field.

In addition to the columns $(\mathtt{x}, \mathtt{v}, \mathtt{m})$, we also include an interaction column $\mathtt{x}_{\mathtt{range}}$, which will keep track of the running sums used by the logup argument, whose elements will be in the larger extension field. We also assume that columns $(\mathtt{v}, \mathtt{m})$ might contain padding rows with the values $(255, 0)$, which will be enforced via constraints.

Let α be a random value chosen by the verifier after the prover commits to the execution trace of the program. The value of $\mathtt{x}_{\mathtt{range}}$ at row i will be computed as follows:

$$\mathtt{x}_{\mathtt{range}}[i] = \frac{1}{(\mathtt{x}[1] + \alpha)} - \frac{\mathtt{m}[1]}{(\mathtt{v}[1] + \alpha)} + \ldots + \frac{1}{(\mathtt{x}[i] + \alpha)} - \frac{\mathtt{m}[i]}{(\mathtt{v}[i] + \alpha)}.$$

This implies that the difference in the $\mathtt{x}_{\mathtt{range}}$ column between consecutive rows will be:

$$\mathtt{x}_{\mathtt{range}}[i] - \mathtt{x}_{\mathtt{range}}[i-1] = \frac{1}{(\mathtt{x}[i] + \alpha)} - \frac{\mathtt{m}[i]}{(\mathtt{v}[i] + \alpha)}.$$

Since several of the 8-bit elements that we want to range check are in fact limbs of a larger 32-bit value, we assume in this case that we can maintain a single $\mathtt{x}_{\mathtt{range}}$ column for all the limbs for efficiency reasons. More precisely, let $\mathtt{x}$ be a 32-bit element that has been split into 4 8-bit limbs $(\mathtt{x}^{(1)}, \mathtt{x}^{(2)}, \mathtt{x}^{(3)}, \mathtt{x}^{(4)})$. In this case, the difference in the $\mathtt{x}_{\mathtt{range}}$ column between consecutive rows will be:

$$\mathtt{x}_{\mathtt{range}}[i] - \mathtt{x}_{\mathtt{range}}[i-1] = \frac{1}{(\mathtt{x}^{(1)}[i] + \alpha)} + \frac{1}{(\mathtt{x}^{(2)}[i] + \alpha)} + \frac{1}{(\mathtt{x}^{(3)}[i] + \alpha)} + \frac{1}{(\mathtt{x}^{(4)}[i] + \alpha)} - \frac{\mathtt{m}[i]}{(\mathtt{v}[i] + \alpha)}.$$

Adding up multiple terms of $1/(\text{something}+\alpha)$ in the secure field requires some additional auxiliary columns. Those additional columns are implemented by, for example, the Stwo logup library.

Though we could further optimize and reuse the same range check for multiple large elements in the trace, we assume that each 32-bit value $\mathtt{x}$ will have separate multiplicity and $\mathtt{x}_{\mathtt{range}}$ columns. The column $\mathtt{v}$ could however be shared across different values being range checked.

Remark 3.1 Though the more general design allows for some of the numbers in the range to be omitted, we opt here for the simpler construction in which all the values in the 8-bit range are listed in range check columns. This optimization however would make sense for 16-bit range checks.

Boundary constraints Let n be the index of the last row.

- $\mathtt{v}[1] = 0$
- $\mathtt{v}[n] = 255$
- $\mathtt{x}_{\mathtt{range}}[n] = 0$

Transition constraints $(1 < i \leq n)$:

- $(\mathtt{v}[i] - \mathtt{v}[i-1]) \cdot (\mathtt{v}[i] - \mathtt{v}[i-1] - 1) = 0$
- $\mathtt{x}_{\mathtt{range}}[i] - \mathtt{x}_{\mathtt{range}}[i-1] = \frac{1}{(\mathtt{x}^{(1)}[i]+\alpha)} + \frac{1}{(\mathtt{x}^{(2)}[i]+\alpha)} + \frac{1}{(\mathtt{x}^{(3)}[i]+\alpha)} + \frac{1}{(\mathtt{x}^{(4)}[i]+\alpha)} - \frac{\mathtt{m}[i]}{(\mathtt{v}[i]+\alpha)}$
- $(\mathtt{v}[i] - \mathtt{v}[i-1] - 1) \cdot \mathtt{m}[i] = 0$

3.4 Lookup tables for bitwise operations

The execution component of the Nexus zkVM design also makes use of 4-bit lookup tables to implement the bitwise operations XOR, AND, and OR.

Let $(\mathtt{a}, \mathtt{b}, \mathtt{c})$ represent three trace columns for 4-bit values for which we want to verify that $\mathtt{a}\ \text{bit-op}\ \mathtt{b} = \mathtt{c}$, where $\text{bit-op} \in \{\oplus, \&, |\}$ denotes one of the bitwise operations $(\mathtt{XOR}, \mathtt{AND}, \mathtt{OR})$. In this section, we describe how to implement this check using logups. In order to do so, the main idea is to create a table indexed by $(\mathtt{a}, \mathtt{b})$ containing field elements representing valid triples of the form

$(\mathtt{a}, \mathtt{b}, \mathtt{a}\ \mathsf{bit\text{-}op}\ \mathtt{b})$. Then, when given a triple $(\mathtt{a}, \mathtt{b}, \mathtt{c})$, we can check whether that triple is valid by simply checking whether the field element representing the triple $(\mathtt{a}, \mathtt{b}, \mathtt{c})$ is present in the table.

In the actual implementation below, we will work with the binary decompositions of the values $\mathtt{a}$ and $\mathtt{b}$ since this will make it easier to map triples of the form $(\mathtt{a}, \mathtt{b}, \mathtt{a}\ \mathsf{bit\text{-}op}\ \mathtt{b})$ to a single field element in order to compute logups.

3.4.1 Bitwise lookup trace elements

In order to implement 4-bit lookup tables for bitwise operations on a triple $(\mathtt{a}, \mathtt{b}, \mathtt{c})$, we will first add columns $\mathtt{a0}, \ldots, \mathtt{a3}, \mathtt{b0}, \ldots, \mathtt{b3}$ to the trace, where $(\mathtt{a0}, \ldots, \mathtt{a3})$ and $(\mathtt{b0}, \ldots, \mathtt{b3})$ correspond to the binary decomposition of the values that we intend to search. These trace columns will be common to all bitwise operations.

Note that we do not need to introduce trace columns corresponding to the binary decomposition of c since all the entries in the lookup table will correspond to field elements representing triples of the form $(\mathtt{a}, \mathtt{b}, \mathtt{a}\ \mathsf{bit\text{-}op}\ \mathtt{b})$ for $\mathsf{bit\text{-}op} \in \{\oplus, \&, |\}$. For these triples, the value $\mathtt{c}$ is a known function of the values $(\mathtt{a0}, \ldots, \mathtt{a3})$ and $(\mathtt{b0}, \ldots, \mathtt{b3})$. Likewise, we do not need to explicitly introduce a trace column for the index being searched since this index is also a known function of the $(\mathtt{a0}, \ldots, \mathtt{a3})$ and $(\mathtt{b0}, \ldots, \mathtt{b3})$.

In addition to the binary decompositions for $\mathtt{a}$ and $\mathtt{b}$, we will add multiplicity columns for each bitwise operation: $\mathtt{m}_{\mathtt{xor}}$, $\mathtt{m}_{\mathtt{and}}$, $\mathtt{m}_{\mathtt{or}}$. In our design, the values will $(\mathtt{a0}, \ldots, \mathtt{a3})$, $(\mathtt{b0}, \ldots, \mathtt{b3})$, $\mathtt{m}_{\mathtt{xor}}$, $\mathtt{m}_{\mathtt{and}}$, $\mathtt{m}_{\mathtt{or}}$ are in the $\mathtt{m31}$ field.

Finally, we will also introduce trace columns $\mathtt{digest}_{\mathtt{xor}}$, $\mathtt{digest}_{\mathtt{and}}$, $\mathtt{digest}_{\mathtt{or}}$ to capture the logup computation for bitwise operations. Unlike the other columns defined above, these are elements of the secure extension field.

Hence, the following set of trace elements will be needed by the lookup component of bitwise operations:

- $\mathtt{a}, \mathtt{b}, \mathtt{c}$: trace columns corresponding to the 4-bits values for which we want to verify $\mathtt{c} = \mathtt{a}\ \mathsf{bit\text{-}op}\ \mathtt{b}$
- $\mathtt{a0}, \ldots, \mathtt{a3}$: the bit decomposition for the $\mathtt{a}$ component of the index value
- $\mathtt{b0}, \ldots, \mathtt{b3}$: the bit decomposition for the $\mathtt{b}$ component of the index value
- $\mathtt{m}_{\mathtt{xor}}, \mathtt{m}_{\mathtt{and}}, \mathtt{m}_{\mathtt{or}}$: multiplicity values for a given index value of the lookup table
- $\mathtt{digest}_{\mathtt{xor}}, \mathtt{digest}_{\mathtt{and}}, \mathtt{digest}_{\mathtt{or}}$: logarithmic derivatives for the bitwise lookup tables

3.4.2 Bitwise operation mapping function

Let $(\mathtt{a}, \mathtt{b}, \mathtt{c})$ represent three 4-bit values for which we want to verify that $\mathtt{c} = \mathtt{a}\ \mathsf{bit\text{-}op}\ \mathtt{b}$, where $\mathtt{a} \in [0, 2^4 - 1]$, $\mathtt{b} \in [0, 2^4 - 1]$, $\mathtt{c} \in [0, 2^4 - 1]$. In order to check whether a triple of the form $(\mathtt{a}, \mathtt{b}, \mathtt{c})$ is part of the lookup table for a given operation, we first need to convert these triples into a field element in the secure extension field. In order to do so, we define a function $\mathsf{triple\text{-}to\text{-}field}(\mathtt{a}, \mathtt{b}, \mathtt{c})$ as follows:

$$\mathsf{triple\text{-}to\text{-}field}(\mathtt{a}, \mathtt{b}, \mathtt{c}) \coloneqq \mathtt{a} + \mathtt{b} \cdot z + \mathtt{c} \cdot z^2,$$

where z is a challenge chosen by the verifier after seeing the commitments to the trace.

3.4.3 Bitwise operation helper functions

In order to compute a logup contribution related to a triple $(\mathtt{a}, \mathtt{b}, \mathtt{c})$, we define a few helper functions to capture the expected values of $(\mathtt{a}, \mathtt{b}, \mathtt{c})$ as a function of the values $(\mathtt{a0}, \ldots, \mathtt{a3})$, $(\mathtt{b0}, \ldots, \mathtt{b3})$ and the bitwise operation. These intermediate functions are defined as follows:

- $\mathsf{a\text{-}func}(\mathtt{a0}, \ldots, \mathtt{a3}) = \mathtt{a0} + \mathtt{a1} \cdot 2 + \ldots + \mathtt{a3} \cdot 2^3$
- $\mathsf{b\text{-}func}(\mathtt{b0}, \ldots, \mathtt{b3}) = \mathtt{b0} + \mathtt{b1} \cdot 2 + \ldots + \mathtt{b3} \cdot 2^3$
- $\mathsf{bit\text{-}xor\text{-}func}(\mathtt{ai}, \mathtt{bi}) = \mathtt{ai} + \mathtt{bi} - 2 \cdot \mathtt{ai} \cdot \mathtt{bi}$
- $\mathsf{bit\text{-}and\text{-}func}(\mathtt{ai}, \mathtt{bi}) = \mathtt{ai} \cdot \mathtt{bi}$

- $\text{bit-or-func}(\mathtt{ai}, \mathtt{bi}) = \mathtt{ai} + \mathtt{bi} - \mathtt{ai} \cdot \mathtt{bi}$
- $\text{tuple-xor-func}(\mathtt{a0}, \ldots, \mathtt{a3}, \mathtt{b0}, \ldots, \mathtt{b3}) =$
  $\text{bit-xor-func}(\mathtt{a0}, \mathtt{b0}) + \text{bit-xor-func}(\mathtt{a1}, \mathtt{b1}) \cdot 2 + \ldots + \text{bit-xor-func}(\mathtt{a3}, \mathtt{b3}) \cdot 2^3$
- $\text{tuple-and-func}(\mathtt{a0}, \ldots, \mathtt{a3}, \mathtt{b0}, \ldots, \mathtt{b3}) =$
  $\text{bit-and-func}(\mathtt{a0}, \mathtt{b0}) + \text{bit-and-func}(\mathtt{a1}, \mathtt{b1}) \cdot 2 + \ldots + \text{bit-and-func}(\mathtt{a3}, \mathtt{b3}) \cdot 2^3$
- $\text{tuple-or-func}(\mathtt{a0}, \ldots, \mathtt{a3}, \mathtt{b0}, \ldots, \mathtt{b3}) =$
  $\text{bit-or-func}(\mathtt{a0}, \mathtt{b0}) + \text{bit-or-func}(\mathtt{a1}, \mathtt{b1}) \cdot 2 + \ldots + \text{bit-or-func}(\mathtt{a3}, \mathtt{b3}) \cdot 2^3$
- $\text{xor-bits-to-field}(\mathtt{a0}, \ldots, \mathtt{a3}, \mathtt{b0}, \ldots, \mathtt{b3}) =$
  $\text{triple-to-field}(\text{a-func}(\mathtt{a0}, \ldots, \mathtt{a3}), \text{b-func}(\mathtt{b0}, \ldots, \mathtt{b3}), \text{tuple-xor-func}(\mathtt{a0}, \ldots, \mathtt{a3}, \mathtt{b0}, \ldots, \mathtt{b3}))$
- $\text{and-bits-to-field}(\mathtt{a0}, \ldots, \mathtt{a3}, \mathtt{b0}, \ldots, \mathtt{b3}) =$
  $\text{triple-to-field}(\text{a-func}(\mathtt{a0}, \ldots, \mathtt{a3}), \text{b-func}(\mathtt{b0}, \ldots, \mathtt{b3}), \text{tuple-and-func}(\mathtt{a0}, \ldots, \mathtt{a3}, \mathtt{b0}, \ldots, \mathtt{b3}))$
- $\text{or-bits-to-field}(\mathtt{a0}, \ldots, \mathtt{a3}, \mathtt{b0}, \ldots, \mathtt{b3}) =$
  $\text{triple-to-field}(\text{a-func}(\mathtt{a0}, \ldots, \mathtt{a3}), \text{b-func}(\mathtt{b0}, \ldots, \mathtt{b3}), \text{tuple-or-func}(\mathtt{a0}, \ldots, \mathtt{a3}, \mathtt{b0}, \ldots, \mathtt{b3}))$

Note that, when when $\mathtt{ai}$ and $\mathtt{bi}$ are bits, it follows that

- $\text{bit-xor-func}(\mathtt{ai}, \mathtt{bi}) = \mathtt{ai} \oplus \mathtt{bi}$
- $\text{bit-and-func}(\mathtt{ai}, \mathtt{bi}) = \mathtt{ai} \,\&\, \mathtt{bi}$
- $\text{bit-or-func}(\mathtt{ai}, \mathtt{bi}) = \mathtt{ai} \mid \mathtt{bi}$.

3.4.4 Bitwise logup computations – 1 triple per row

Let α be a random value chosen by the verifier after the prover commits to the execution trace of the program.

Let $(\mathtt{a}[i], \mathtt{b}[i], \mathtt{c}[i])$, $(\mathtt{a0}[i], \ldots, \mathtt{a3}[i])$, $(\mathtt{b0}[i], \ldots, \mathtt{b3}[i])$, $(\mathtt{digest}_{\mathtt{xor}}[i], \mathtt{m}_{\mathtt{xor}}[i])$, $(\mathtt{digest}_{\mathtt{and}}[i], \mathtt{m}_{\mathtt{and}}[i])$, $(\mathtt{digest}_{\mathtt{or}}[i], \mathtt{m}_{\mathtt{or}}[i])$ represent respectively the values of the trace columns $(\mathtt{a}, \mathtt{b}, \mathtt{c})$, $(\mathtt{b0}, \ldots, \mathtt{b3})$, $(\mathtt{digest}_{\mathtt{xor}}, \mathtt{m}_{\mathtt{xor}})$, $(\mathtt{digest}_{\mathtt{and}}, \mathtt{m}_{\mathtt{and}})$, $(\mathtt{digest}_{\mathtt{or}}, \mathtt{m}_{\mathtt{or}})$ at row i. Using the intermediate functions above, the value of the bitwise digests at row i must satisfy the following constraints for $\mathtt{op} \in \{\mathtt{xor}, \mathtt{and}, \mathtt{or}\}$:

$$\begin{aligned} \mathtt{digest}_{\mathtt{op}}[i] \quad = \quad & \frac{1}{(\text{triple-to-field}(\mathtt{a}[1], \mathtt{b}[1], \mathtt{c}[1]) + \alpha)} - \\ & \frac{\mathtt{m}_{\mathtt{op}}[1]}{(\text{op-bits-to-field}(\mathtt{a0}[1], \ldots, \mathtt{a3}[1], \mathtt{b0}[1], \ldots, \mathtt{b3}[1]) + \alpha)} + \ldots + \\ & \frac{1}{(\text{triple-to-field}(\mathtt{a}[i], \mathtt{b}[i], \mathtt{c}[i]) + \alpha)} - \\ & \frac{\mathtt{m}_{\mathtt{op}}[i]}{(\text{op-bits-to-field}(\mathtt{a0}[i], \ldots, \mathtt{a3}[i], \mathtt{b0}[i], \ldots, \mathtt{b3}[i]) + \alpha)} \end{aligned}$$

This implies that the difference in the $\mathtt{digest}_{\mathtt{op}}$ column between consecutive rows for $\mathtt{op} \in \{\mathtt{xor}, \mathtt{and}, \mathtt{or}\}$ is as follows:

$$\begin{aligned} \mathtt{digest}_{\mathtt{op}}[i] - \mathtt{digest}_{\mathtt{op}}[i-1] \quad = \quad & \frac{1}{(\text{triple-to-field}(\mathtt{a}[i], \mathtt{b}[i], \mathtt{c}[i]) + \alpha)} - \\ & \frac{\mathtt{m}_{\mathtt{op}}[i]}{(\text{op-bits-to-field}(\mathtt{a0}[i], \ldots, \mathtt{a3}[i], \mathtt{b0}[i], \ldots, \mathtt{b3}[i]) + \alpha)} \end{aligned}$$

If we use a preprocessed trace, the lookup tables will contain $(A, B, C_{\mathtt{xor}}, C_{\mathtt{and}}, C_{\mathtt{or}})$ triples where $0 \le A, B < 256$, $C_{\mathtt{op}} = A \ \mathtt{op} \ B$ for $\mathtt{op} \in \{\mathtt{xor}, \mathtt{and}, \mathtt{or}\}$. Using these, $\mathtt{digest}_{\mathtt{op}}[i]$ at row i for

$\mathtt{op} \in \{\mathtt{xor}, \mathtt{and}, \mathtt{or}\}$ satisfies

$$\begin{aligned}\mathtt{digest}_{\mathsf{op}}[i] \quad = \quad & \frac{1}{(\mathsf{triple\text{-}to\text{-}field}(\mathsf{a}[1], \mathsf{b}[1], \mathsf{c}[1]) + \alpha)} - \\ & \frac{\mathtt{m}_{\mathsf{op}}[1]}{(\mathsf{triple\text{-}to\text{-}field}(A[1], B[1], C_{\mathsf{op}}[1]) + \alpha)} + \ldots + \\ & \frac{1}{(\mathsf{triple\text{-}to\text{-}field}(\mathsf{a}[i], \mathsf{b}[i], \mathsf{c}[i]) + \alpha)} - \\ & \frac{\mathtt{m}_{\mathsf{op}}[i]}{(\mathsf{triple\text{-}to\text{-}field}(A[i], B[i], C_{\mathsf{op}}[i]) + \alpha)}\end{aligned}$$

This implies that the difference in the $\mathtt{digest}_{\mathsf{op}}$ column between consecutive rows will be:

$$\begin{aligned}\mathtt{digest}_{\mathsf{op}}[i] - \mathtt{digest}_{\mathsf{op}}[i-1] \quad = \quad & \frac{1}{(\mathsf{triple\text{-}to\text{-}field}(\mathsf{a}[i], \mathsf{b}[i], \mathsf{c}[i]) + \alpha)} - \\ & \frac{\mathtt{m}_{\mathsf{op}}[i]}{(\mathsf{triple\text{-}to\text{-}field}(A[i], B[i], C_{\mathsf{op}}[i]) + \alpha)}\end{aligned}$$

3.4.5 Bitwise logup computations – multiple triples per row

Since we may need to check multiple triples for bitwise operations per row, it is straightforward to extend the logup computation in the previous section to handle that case. Let $(\mathsf{a}_1, \mathsf{b}_1, \mathsf{c}_1), \ldots, (\mathsf{a}_k, \mathsf{b}_k, \mathsf{c}_k)$ denote k triples being checked in the same row for an operation $\mathtt{op} \in \{\mathtt{xor}, \mathtt{and}, \mathtt{or}\}$. Let $\mathtt{m}_{\mathsf{op}}$ be the multiplicity for the entry $(\mathsf{a0}, \ldots, \mathsf{a3}, \mathsf{b0}, \ldots, \mathsf{b3})$. In this case, the difference in the $\mathtt{digest}_{\mathsf{op}}$ column between consecutive rows will be:

$$\begin{aligned}\mathtt{digest}_{\mathsf{op}}[i] - \mathtt{digest}_{\mathsf{op}}[i-1] \quad = \quad & \frac{1}{(\mathsf{triple\text{-}to\text{-}field}(\mathsf{a}_1[i], \mathsf{b}_1[i], \mathsf{c}_1[i]) + \alpha)} + \ldots + \\ & \frac{1}{(\mathsf{triple\text{-}to\text{-}field}(\mathsf{a}_k[i], \mathsf{b}_k[i], \mathsf{c}_k[i]) + \alpha)} - \\ & \frac{\mathtt{m}_{\mathsf{op}}[i]}{(\mathsf{op\text{-}bits\text{-}to\text{-}field}(\mathsf{a0}[i], \ldots, \mathsf{a3}[i], \mathsf{b0}[i], \ldots, \mathsf{b3}[i]) + \alpha)}\end{aligned}$$

Moreover, if we assume the use of a preprocessed trace, the difference in the $\mathtt{digest}_{\mathsf{op}}$ column between consecutive rows will be:

$$\begin{aligned}\mathtt{digest}_{\mathsf{op}}[i] - \mathtt{digest}_{\mathsf{op}}[i-1] \quad = \quad & \frac{1}{(\mathsf{triple\text{-}to\text{-}field}(\mathsf{a}_1[i], \mathsf{b}_1[i], \mathsf{c}_1[i]) + \alpha)} + \ldots + \\ & \frac{1}{(\mathsf{triple\text{-}to\text{-}field}(\mathsf{a}_k[i], \mathsf{b}_k[i], \mathsf{c}_k[i]) + \alpha)} - \\ & \frac{\mathtt{m}_{\mathsf{op}}[i]}{(\mathsf{triple\text{-}to\text{-}field}(A[i], B[i], C_{\mathsf{op}}[i]) + \alpha)}\end{aligned}$$

3.4.6 Bitwise lookup table constraints

Boundary constraints: Let n be the index of the last row, and let k represent the number of triples being looked up for $\mathtt{op} \in \{\mathtt{xor}, \mathtt{and}, \mathtt{or}\}$:

$$\begin{aligned}\mathtt{digest}_{\mathsf{op}}[1] \quad = \quad & \frac{1}{(\mathsf{triple\text{-}to\text{-}field}(\mathsf{a}_1[1], \mathsf{b}_1[1], \mathsf{c}_1[1]) + \alpha)} + \ldots + \\ & \frac{1}{(\mathsf{triple\text{-}to\text{-}field}(\mathsf{a}_k[1], \mathsf{b}_k[1], \mathsf{c}_k[1]) + \alpha)} - \\ & \frac{\mathtt{m}_{\mathsf{op}}[1]}{(\mathsf{op\text{-}bits\text{-}to\text{-}field}(\mathsf{a0}[1], \ldots, \mathsf{a3}[1], \mathsf{b0}[1], \ldots, \mathsf{b3}[1]) + \alpha)}\end{aligned}$$

Transition constraints:

// `Running logup sum`

- $\texttt{digest}_{\texttt{op}}[i] - \texttt{digest}_{\texttt{op}}[i-1] = \frac{1}{(\textsf{triple-to-field}(\texttt{a}_1[i],\texttt{b}_1[i],\texttt{c}_1[i])+\alpha)} + \ldots + \frac{1}{(\textsf{triple-to-field}(\texttt{a}_k[i],\texttt{b}_k[i],\texttt{c}_k[i])+\alpha)} - \frac{\texttt{m}_{\texttt{op}}[i]}{(\textsf{op-bits-to-field}(\texttt{a0}[i],\ldots,\texttt{a3}[i],\texttt{b0}[i],\ldots,\texttt{b3}[i])+\alpha)}$

// `Enforcing` $\texttt{a0},\ldots,\texttt{a3},\texttt{b0},\ldots,\texttt{b3} \in \{0,1\}$

- $((\texttt{a0}[i]) \cdot (1 - \texttt{a0}[i]) = 0$
- $((\texttt{a1}[i]) \cdot (1 - \texttt{a1}[i]) = 0$
- $((\texttt{a2}[i]) \cdot (1 - \texttt{a2}[i]) = 0$
- $((\texttt{a3}[i]) \cdot (1 - \texttt{a3}[i]) = 0$
- $((\texttt{b0}[i]) \cdot (1 - \texttt{b0}[i]) = 0$
- $((\texttt{b1}[i]) \cdot (1 - \texttt{b1}[i]) = 0$
- $((\texttt{b2}[i]) \cdot (1 - \texttt{b2}[i]) = 0$
- $((\texttt{b3}[i]) \cdot (1 - \texttt{b3}[i]) = 0$

Remark 3.2 The constraints above do not assume that lookup table entries are unique. While it suffices for a honest prover to create a separate entry for each value being looked up together with the corresponding multiplicity and pad unused rows with values $(\texttt{a0},\ldots,\texttt{a3},\texttt{b0},\ldots,\texttt{b3},m) = (0,\ldots,0)$, a dishonest prover may create repeated entries in the lookup table with different non-zero multiplicity values.

- For instance, a dishonest prover could create an entry $(\texttt{a0},\ldots,\texttt{a3},\texttt{b0},\ldots,\texttt{b3},m_1)$ in one row and an entry $(\texttt{a0},\ldots,\texttt{a3},\texttt{b0},\ldots,\texttt{b3},m_2)$ in a different row. This is not a problem since the proof will only go through if m_1+m_2 matches the correct multiplicity for the entry $(\texttt{a0},\ldots,\texttt{a3},\texttt{b0},\ldots,\texttt{b3})$. If $m_1 + m_2$ does not match the correct multiplicity for that entry, the logup check will fail.
- Though it is possible to add additional constraints to enforce that each entry can have at most 1 non-zero value for the multiplicity, this does not seem needed.

4 CPU component

The CPU component is responsible for fetching, decoding, and preparing instructions for execution. In particular, this involves:

- Reading the next instruction from the program memory using the program counter
- Decode the instruction and check the correctness of its format
- Read values associated with operands from the register memory
- Execute the instruction using the instruction execution component

While most of the tasks described above will be handled separately by different components of the zkVM, such as the register or program memory, the instruction decoding and format verification task will be handled exclusively by the CPU component. For the remaining tasks, the main job of the CPU component is to specify the values used to call these components.

Remark 4.1 In order to facilitate the understanding of the current specification, we often describe relevant constraints in two steps:

- In a first step, we describe all constraints under the assumption that the underlying finite field used by the arithmetization is sufficiently large to represent 32-bit elements.
- Then, in a second step, we describe the actual constraints used by our implementation assuming small finite fields, such as the `m31` field used by the `Stwo` prover .

During this second step, variables representing large elements will be split into limbs representing their components. When doing so, we often have to introduce additional variables and constraints to capture cases which did not have to be considered in the large field case.

4.1 CPU trace elements

We will create the following set of trace elements for the CPU component:

- `clk`: The current execution time
- `pc`: The current value of the program counter register
- `pc-next`: The next value of the program counter register
- `opcode`: The opcode defining the instruction
- `op-a`: The address of the first operand of the instruction
- `op-b`: The address of the second operand of the instruction
- `op-c`: The address of the third operand of the instruction
- `op-b-flag`: A flag indicating whether operand `op-b` is used
- `imm-c`: A flag indicating whether operand `op-c` is an immediate value
- $\texttt{instr}_{\text{val}}$: A 32-bit word encoding the instruction stored at the `pc` address
- $\texttt{op-a}_{\text{val}}$: the value of operand `op-a`
- $\texttt{op-b}_{\text{val}}$: the value of operand `op-b`
- $\texttt{op-c}_{\text{val}}$: the value of operand `op-c`

In addition to the values defined above, we also define selector flags for each instruction supported by the Nexus zkVM and additional flag used for padding:

- `is-lui`: a selector flag which indicates an `lui` operation
- `is-auipc`: a selector flag which indicates an `auipc` operation
- `is-jal`: a selector flag which indicates an `jal` operation
- `is-jalr`: a selector flag which indicates an `jalr` operation
- `is-ecall`: a selector flag which indicates an `ecall` operation
- `is-ebreak`: a selector flag which indicates an `ebreak` operation
- `is-fence`: a selector flag which indicates an `fence` operation
- `is-unimp`: a selector flag which indicates an `unimp` operation
- `is-beq`: a selector flag which indicates an `beq` operation
- `is-bne`: a selector flag which indicates an `bne` operation
- `is-blt`: a selector flag which indicates an `blt` operation
- `is-bge`: a selector flag which indicates an `bge` operation
- `is-bltu`: a selector flag which indicates an `bltu` operation
- `is-bgeu`: a selector flag which indicates an `bgeu` operation
- `is-lb`: a selector flag which indicates an `lb` operation
- `is-lh`: a selector flag which indicates an `lh` operation
- `is-lw`: a selector flag which indicates an `lw` operation
- `is-lbu`: a selector flag which indicates an `lbu` operation
- `is-lhu`: a selector flag which indicates an `lhu` operation
- `is-sb`: a selector flag which indicates an `sb` operation
- `is-sh`: a selector flag which indicates an `sh` operation
- `is-sw`: a selector flag which indicates an `sw` operation
- `is-add`: a selector flag which indicates an `add` or `addi` operation
- `is-sub`: a selector flag which indicates an `sub` operation
- `is-sll`: a selector flag which indicates an `sll` or `slli` operation
- `is-slt`: a selector flag which indicates an `slt` or `slti` operation
- `is-sltu`: a selector flag which indicates an `sltu` or `sltiu` operation
- `is-xor`: a selector flag which indicates an `xor` or `xori` operation
- `is-srl`: a selector flag which indicates an `srl` or `srli` operation
- `is-sra`: a selector flag which indicates an `sra` or `srai` operation
- `is-or`: a selector flag which indicates an `or` or `ori` operation
- `is-and`: a selector flag which indicates an `and` or `andi` operation

- `is-pad`: a selector flag which is used for padding, not a computational step

While the instruction flags are used to indicate whether the current row in the trace corresponds to a particular instruction, the `is-pad` flag is used when additional padding rows need to be added to the trace to make the total number of rows a power of two (which is required by the `Stwo` prover backend).

The value of each of the flags defined above should be either 0 or 1 and exactly one of these should be set to 1 in each row. Moreover, we also assume that padding rows should only appear at the end of the trace. Hence, once set to 1, the value of the `is-pad` flag should remain 1. We will be adding constraints to enforce these conditions.

Remark 4.2
- We do not define separate flags for ALU instructions with immediate values (such as `addi`) since other flags (such as `is-add` together with `imm-c`) can play the same role.
- We do not currently support `fence` instructions and assume that these are mapped to `nop` ≡ (`addi x0 x0 0`) beforehand.
- `unimp` instructions are also not currently supported since the halting functionality is handled via system calls (see Table 3), though for completeness the decoding constraints do include some handling for it.

4.2 CPU constraints assuming large fields

4.2.1 Range checks

- $\texttt{clk} \in \left[0, 2^{32} - 1\right]$
- $(\texttt{op-b-flag})(1 - \texttt{op-b-flag}) = 0$
- $(\texttt{imm-c})(1 - \texttt{imm-c}) = 0$

// $\texttt{pc} \in \left[0, 2^{32} - 1\right]$ `- guaranteed via program memory checking`
// $\texttt{pc-next} \in \left[0, 2^{32} - 1\right]$ `- implied by arithmetic constraints`
// $\texttt{instr}_{\mathsf{val}} \in \left[0, 2^{32} - 1\right]$ `- performed in the in the program memory component`
// `op-a range check - guaranteed via register memory checking / arithmetic constraints`
// `op-b range check - guaranteed via register memory checking when op-b-flag` $= 1$
// `op-c range check - guaranteed via register memory checking when imm-c` $= 0$
// $\texttt{op-a}_{\mathsf{val}}$ `range check - guaranteed via register memory checking / arithmetic constraints`
// $\texttt{op-b}_{\mathsf{val}}$ `range check - guaranteed via register memory checking when op-b-flag` $= 1$
// $\texttt{op-c}_{\mathsf{val}}$ `range check - guaranteed via register memory checking when imm-c` $= 0$

Remark 4.3
- Flags for instruction types and constraints for specific instructions are specified separately below.
- Range checks when `imm-c` $= 1$ are being specified in the sections for the relevant instructions.
- Range checks for `op-a` and $\texttt{op-a}_{\mathsf{val}}$ are guaranteed via register memory checking except for instructions for which `op-a` is not defined. In such cases, we add constraints to guarantee that `op-a` $= 0$ and $\texttt{op-a}_{\mathsf{val}} = 0$.
- `fence` is not supported and should be mapped to `nop` ≡ (`addi x0 x0 0`).

4.2.2 Instruction flag constraints

Remark 4.4 Since all constraints in the remaining of this section are for the same row, we do not explicitly write down the row index i when describing these constraints.

// `Enforcing instruction flags are either` 0 `or` 1
- $(\texttt{is-lui}) \cdot (1 - \texttt{is-lui}) = 0$ ▷ `lui instruction`
- $(\texttt{is-auipc}) \cdot (1 - \texttt{is-auipc}) = 0$ ▷ `auipc instruction`
- $(\texttt{is-jal}) \cdot (1 - \texttt{is-jal}) = 0$ ▷ `jal instruction`
- $(\texttt{is-jalr}) \cdot (1 - \texttt{is-jalr}) = 0$ ▷ `jalr instruction`
- $(\texttt{is-ecall}) \cdot (1 - \texttt{is-ecall}) = 0$ ▷ `ecall instruction`

- $(\texttt{is-ebreak}) \cdot (1 - \texttt{is-ebreak}) = 0$ ▷ ebreak instruction
- $(\texttt{is-fence}) \cdot (1 - \texttt{is-fence}) = 0$ ▷ fence instruction
- $(\texttt{is-unimp}) \cdot (1 - \texttt{is-unimp}) = 0$ ▷ unimp instruction
- $(\texttt{is-beq}) \cdot (1 - \texttt{is-beq}) = 0$ ▷ beq instruction
- $(\texttt{is-bne}) \cdot (1 - \texttt{is-bne}) = 0$ ▷ bne instruction
- $(\texttt{is-blt}) \cdot (1 - \texttt{is-blt}) = 0$ ▷ blt instruction
- $(\texttt{is-bge}) \cdot (1 - \texttt{is-bge}) = 0$ ▷ bge instruction
- $(\texttt{is-bltu}) \cdot (1 - \texttt{is-bltu}) = 0$ ▷ bltu instruction
- $(\texttt{is-bgeu}) \cdot (1 - \texttt{is-bgeu}) = 0$ ▷ bgeu instruction
- $(\texttt{is-lb}) \cdot (1 - \texttt{is-lb}) = 0$ ▷ lb instruction
- $(\texttt{is-lh}) \cdot (1 - \texttt{is-lh}) = 0$ ▷ lh instruction
- $(\texttt{is-lw}) \cdot (1 - \texttt{is-lw}) = 0$ ▷ lw instruction
- $(\texttt{is-lbu}) \cdot (1 - \texttt{is-lbu}) = 0$ ▷ lbu instruction
- $(\texttt{is-lhu}) \cdot (1 - \texttt{is-lhu}) = 0$ ▷ lhu instruction
- $(\texttt{is-sb}) \cdot (1 - \texttt{is-sb}) = 0$ ▷ sb instruction
- $(\texttt{is-sh}) \cdot (1 - \texttt{is-sh}) = 0$ ▷ sh instruction
- $(\texttt{is-sw}) \cdot (1 - \texttt{is-sw}) = 0$ ▷ sw instruction
- $(\texttt{is-add}) \cdot (1 - \texttt{is-add}) = 0$ ▷ add or addi instruction
- $(\texttt{is-sub}) \cdot (1 - \texttt{is-sub}) = 0$ ▷ sub instruction
- $(\texttt{is-sll}) \cdot (1 - \texttt{is-sll}) = 0$ ▷ sll or slli instruction
- $(\texttt{is-slt}) \cdot (1 - \texttt{is-slt}) = 0$ ▷ slt or slti instruction
- $(\texttt{is-sltu}) \cdot (1 - \texttt{is-sltu}) = 0$ ▷ sltu or sltiu instruction
- $(\texttt{is-xor}) \cdot (1 - \texttt{is-xor}) = 0$ ▷ xor or xori instruction
- $(\texttt{is-srl}) \cdot (1 - \texttt{is-srl}) = 0$ ▷ srl or srli instruction
- $(\texttt{is-sra}) \cdot (1 - \texttt{is-sra}) = 0$ ▷ sra or srai instruction
- $(\texttt{is-or}) \cdot (1 - \texttt{is-or}) = 0$ ▷ or or ori instruction
- $(\texttt{is-and}) \cdot (1 - \texttt{is-and}) = 0$ ▷ and or andi instruction
- $(\texttt{is-pad}) \cdot (1 - \texttt{is-pad}) = 0$ ▷ used for padding

// Enforcing exactly one instruction flag is set to 1

- $\texttt{is-lui} + \texttt{is-auipc} + \texttt{is-jal} + \texttt{is-jalr} + \texttt{is-ecall} + \texttt{is-ebreak} + \texttt{is-unimp} +$
  $\texttt{is-beq} + \texttt{is-bne} + \texttt{is-blt} + \texttt{is-bge} + \texttt{is-bltu} + \texttt{is-bgeu} +$
  $\texttt{is-lb} + \texttt{is-lh} + \texttt{is-lw} + \texttt{is-lbu} + \texttt{is-lhu} + \texttt{is-sb} + \texttt{is-sh} + \texttt{is-sw} +$
  $\texttt{is-add} + \texttt{is-sub} + \texttt{is-sll} + \texttt{is-slt} + \texttt{is-sltu} +$
  $\texttt{is-xor} + \texttt{is-srl} + \texttt{is-sra} + \texttt{is-or} + \texttt{is-and} + \texttt{is-pad} = 1$

// Matching flag with instruction opcode

- $(\texttt{is-lui}) \cdot (\texttt{opcode} - \texttt{LUI}) = 0$ ▷ lui instruction
- $(\texttt{is-auipc}) \cdot (\texttt{opcode} - \texttt{AUIPC}) = 0$ ▷ auipc instruction
- $(\texttt{is-jal}) \cdot (\texttt{opcode} - \texttt{JAL}) = 0$ ▷ jal instruction
- $(\texttt{is-jalr}) \cdot (\texttt{opcode} - \texttt{JALR}) = 0$ ▷ jalr instruction
- $(\texttt{is-ecall}) \cdot (\texttt{opcode} - \texttt{ECALL}) = 0$ ▷ ecall instruction
- $(\texttt{is-ebreak}) \cdot (\texttt{opcode} - \texttt{EBREAK}) = 0$ ▷ ebreak instruction
- $(\texttt{is-fence}) \cdot (\texttt{opcode} - \texttt{FENCE}) = 0$ ▷ fence instruction
- $(\texttt{is-unimp}) \cdot (\texttt{opcode} - \texttt{UNIMP}) = 0$ ▷ unimp instruction
- $(\texttt{is-beq}) \cdot (\texttt{opcode} - \texttt{BEQ}) = 0$ ▷ beq instruction
- $(\texttt{is-bne}) \cdot (\texttt{opcode} - \texttt{BNE}) = 0$ ▷ bne instruction
- $(\texttt{is-blt}) \cdot (\texttt{opcode} - \texttt{BLT}) = 0$ ▷ blt instruction
- $(\texttt{is-bge}) \cdot (\texttt{opcode} - \texttt{BGE}) = 0$ ▷ bge instruction
- $(\texttt{is-bltu}) \cdot (\texttt{opcode} - \texttt{BLTU}) = 0$ ▷ bltu instruction
- $(\texttt{is-bgeu}) \cdot (\texttt{opcode} - \texttt{BGEU}) = 0$ ▷ bgeu instruction
- $(\texttt{is-lb}) \cdot (\texttt{opcode} - \texttt{LB}) = 0$ ▷ lb instruction
- $(\texttt{is-lh}) \cdot (\texttt{opcode} - \texttt{LH}) = 0$ ▷ lh instruction
- $(\texttt{is-lw}) \cdot (\texttt{opcode} - \texttt{LW}) = 0$ ▷ lw instruction
- $(\texttt{is-lbu}) \cdot (\texttt{opcode} - \texttt{LBU}) = 0$ ▷ lbu instruction
- $(\texttt{is-lhu}) \cdot (\texttt{opcode} - \texttt{LHU}) = 0$ ▷ lhu instruction
- $(\texttt{is-sb}) \cdot (\texttt{opcode} - \texttt{SB}) = 0$ ▷ sb instruction
- $(\texttt{is-sh}) \cdot (\texttt{opcode} - \texttt{SH}) = 0$ ▷ sh instruction
- $(\texttt{is-sw}) \cdot (\texttt{opcode} - \texttt{SW}) = 0$ ▷ sw instruction

- $(\texttt{is-add}) \cdot (\texttt{opcode} - \texttt{ADD}) = 0$ ▷ add or addi instruction
- $(\texttt{is-sub}) \cdot (\texttt{opcode} - \texttt{SUB}) = 0$ ▷ sub instruction
- $(\texttt{is-sll}) \cdot (\texttt{opcode} - \texttt{SLL}) = 0$ ▷ sll or slli instruction
- $(\texttt{is-slt}) \cdot (\texttt{opcode} - \texttt{SLT}) = 0$ ▷ slt or slti instruction
- $(\texttt{is-sltu}) \cdot (\texttt{opcode} - \texttt{SLTU}) = 0$ ▷ sltu or sltiu instruction
- $(\texttt{is-xor}) \cdot (\texttt{opcode} - \texttt{XOR}) = 0$ ▷ xor or xori instruction
- $(\texttt{is-srl}) \cdot (\texttt{opcode} - \texttt{SRL}) = 0$ ▷ srl or srli instruction
- $(\texttt{is-sra}) \cdot (\texttt{opcode} - \texttt{SRA}) = 0$ ▷ sra or srai instruction
- $(\texttt{is-or}) \cdot (\texttt{opcode} - \texttt{OR}) = 0$ ▷ or or ori instruction
- $(\texttt{is-and}) \cdot (\texttt{opcode} - \texttt{AND}) = 0$ ▷ and or andi instruction

Remark 4.5 In the constraints defined above:

- $\texttt{LUI}, \texttt{AUIPC}, \dots, \texttt{OR}, \texttt{AND}$ are assumed to be predefined constants for the opcodes.
- The constraints above actually refer to these constants as field elements.
- For instance, if $\texttt{XYZ} = \texttt{0b000100}$, then $\texttt{XYZ} = 4$ in the constraint equation.

// Instruction types

- $\texttt{is-type-u} = \texttt{is-lui} + \texttt{is-auipc}$ ▷ Type U
- $\texttt{is-type-j} = \texttt{is-jal}$ ▷ Type J - JAL instruction
- $\texttt{is-load} = \texttt{is-lb} + \texttt{is-lh} + \texttt{is-lw} + \texttt{is-lbu} + \texttt{is-lhu}$ ▷ Load instructions
- $\texttt{is-type-s} = \texttt{is-sb} + \texttt{is-sh} + \texttt{is-sw}$ ▷ Type S - Store instructions
- $\texttt{is-type-b} = \texttt{is-beq} + \texttt{is-bne} + \texttt{is-blt} + \texttt{is-bge} + \texttt{is-bltu} + \texttt{is-bgeu}$ ▷ Type B - Branch instructions
- $\texttt{is-type-sys} = \texttt{is-ecall} + \texttt{is-ebreak}$ ▷ System calls

// Type R instructions

- $\texttt{is-type-r} = (1 - \texttt{imm-c}) \cdot (\texttt{is-add} + \texttt{is-sub} + \texttt{is-slt} + \texttt{is-sltu} + \texttt{is-xor} + \texttt{is-or} + \texttt{is-and} + \texttt{is-sll} + \texttt{is-srl} + \texttt{is-sra})$

// ALU instructions

- $\texttt{is-alu} = \texttt{is-add} + \texttt{is-sub} + \texttt{is-slt} + \texttt{is-sltu} + \texttt{is-xor} + \texttt{is-or} + \texttt{is-and} + \texttt{is-sll} + \texttt{is-srl} + \texttt{is-sra}$
- $\texttt{is-alu-imm-shift} = \texttt{imm-c} \cdot (\texttt{is-sll} + \texttt{is-srl} + \texttt{is-sra})$
- $\texttt{is-alu-imm-no-shift} = \texttt{imm-c} \cdot (\texttt{is-add} + \texttt{is-slt} + \texttt{is-sltu} + \texttt{is-xor} + \texttt{is-or} + \texttt{is-and})$

// Type I instructions with non-shift immediate values

- $\texttt{is-type-i-no-shift} = \texttt{is-load} + \texttt{is-alu-imm-no-shift} + \texttt{is-jalr}$

// Type I instructions

- $\texttt{is-type-i} = \texttt{is-load} + \texttt{is-alu-imm-no-shift} + \texttt{is-alu-imm-shift} + \texttt{is-jalr}$

4.2.3 Arithmetic constraints

The main checks to be performed by the CPU component are that:

- the instruction word value matches the values in the instruction operands and opcodes;
- the corrected instruction flag being set matches the instruction opcode; and
- only one of the instruction flags is set.

In the following we specify the arithmetic constraints to perform the checks just mentioned. For numbers in binary, we will need to convert them first to a field element, using a standard binary-to-integer conversion (e.g., $\texttt{0b0000011} \equiv 2^1 + 2^0 \equiv 3$, $\texttt{0b1100011} \equiv 2^6 + 2^5 + 2^1 + 2^0 \equiv 99$).

Boundary constraints

// Setting $\texttt{pc}[1] = \texttt{PC-INIT}$ - this is a constant computed during the two-pass phase

- $\texttt{pc}[1] = 0$

// Setting $\texttt{clk}[1] = 1$

- $\texttt{clk}[1] = 1$

Multi-row constraints

// Transition constraints for the program counter $\texttt{pc}[i]$ at row i
- $\texttt{pc}[i+1] = \texttt{pc-next}[i]$ ▷ unless $i+1 \equiv 0 \bmod \texttt{num-rows}$ or $\texttt{is-pad}[i+1] = 1$

// Transition constraints for $\texttt{clk}[i]$ at row i
- $\texttt{clk}[i] = \texttt{clk}[i-1] + 1$

Remark 4.6 We will need to create limbs for `clk` if its maximum value might be greater than the field size.

Common constraints

Remark 4.7 Since all constraints in the remaining of this section are for the same row, we do not explicitly write down the row index i when describing these constraints.

// Ensuring that pc is a multiple of 4
- $\texttt{pc-aux} \cdot 4 - \texttt{pc}$

// Enforcing $\texttt{pc-aux} \in \left[0, 2^{30}-1\right]$
- $\texttt{pc-aux} \in \left[0, 2^{30}-1\right]$

// Ensuring that $\texttt{op-b-flag} = 1$ for all instructions except lui, auipc, jal, unimp
- $(\texttt{is-sb} + \texttt{is-sh} + \texttt{is-sw} + \texttt{is-lb} + \texttt{is-lh} + \texttt{is-lw} + \texttt{is-lbu} + \texttt{is-lhu} + \texttt{is-jalr} +$
  $\texttt{is-add} + \texttt{is-sub} + \texttt{is-slt} + \texttt{is-sltu} + \texttt{is-xor} + \texttt{is-or} + \texttt{is-and} + \texttt{is-sll} + \texttt{is-srl} + \texttt{is-sra} +$
  $\texttt{is-beq} + \texttt{is-bne} + \texttt{is-blt} + \texttt{is-bge} + \texttt{is-bltu} + \texttt{is-bgeu} + \texttt{is-ecall} + \texttt{is-ebreak} - \texttt{op-b-flag}) = 0$

// Ensuring that $\texttt{imm-c} = 1$ for non-ALU instructions
// Notice that imm-c can be 0 or 1 for ALU instructions
- $(\texttt{is-lui} + \texttt{is-auipc} + \texttt{is-jal} + \texttt{is-jalr} + \texttt{is-ecall} + \texttt{is-ebreak} +$
  $\texttt{is-sb} + \texttt{is-sh} + \texttt{is-sw} + \texttt{is-lb} + \texttt{is-lh} + \texttt{is-lw} + \texttt{is-lbu} + \texttt{is-lhu} +$
  $\texttt{is-beq} + \texttt{is-bne} + \texttt{is-blt} + \texttt{is-bge} + \texttt{is-bltu} + \texttt{is-bgeu})(1 - \texttt{imm-c}) = 0$

// Enforcing $\texttt{imm-c} = 0$ for sub
- $(\texttt{is-sub}) \cdot \texttt{imm-c} = 0$

// Determining $\texttt{op-a}_{\texttt{val}}\texttt{-effective}$
// $\texttt{op-a}_{\texttt{val}}\texttt{-effective-flag}$ is an auxiliary variable
// $\texttt{op-a}_{\texttt{val}}\texttt{-effective-flag} = 1$ indicates op-a is non-zero
// $\texttt{op-a}_{\texttt{val}}\texttt{-effective-flag} = 0$ indicates op-a is zero
- $\texttt{op-a} \cdot \texttt{op-a}_{\texttt{val}}\texttt{-effective-flag-aux} = \texttt{op-a}_{\texttt{val}}\texttt{-effective-flag}$

// Ensuring $\texttt{op-a}_{\texttt{val}}\texttt{-effective-flag-aux} \neq 0$
- $\texttt{op-a}_{\texttt{val}}\texttt{-effective-flag-aux} \cdot \texttt{op-a}_{\texttt{val}}\texttt{-effective-flag-aux-inv} = 1$

// Enforcing $\texttt{op-a}_{\texttt{val}}\texttt{-effective-flag} \in \{0, 1\}$
- $(\texttt{op-a}_{\texttt{val}}\texttt{-effective-flag}) \cdot (1 - \texttt{op-a}_{\texttt{val}}\texttt{-effective-flag}) = 0$

// Enforcing relation between $\texttt{op-a}_{\texttt{val}}$ and $\texttt{op-a}_{\texttt{val}}\texttt{-effective}$
- $\texttt{op-a}_{\texttt{val}} \cdot \texttt{op-a}_{\texttt{val}}\texttt{-effective-flag} = \texttt{op-a}_{\texttt{val}}\texttt{-effective}$

Type U – LUI and AUIPC instructions

// Type U instructions - lui and auipc
// op-a is ranged checked via reg memory checking
// op-b is set to 0
// op-c range check follows from other range checks below
// Making sure that op-c is consistent with the immediate parts

- $\texttt{is-type-u} \cdot (\texttt{op-c12-31} - \texttt{op-c}) = 0$

// Setting lower 12 bits to 0 in order to compute $\texttt{op-c}_{\mathsf{val}}$ from op-c
- $\texttt{is-type-u} \cdot (\texttt{op-c12-31} \cdot 2^{12} - \texttt{op-c}_{\mathsf{val}}) = 0$

// Range checking the different immediate parts
- $\texttt{is-type-u} \cdot (\texttt{op-c12-31} \in \Big[0, 2^{20} - 1\Big])$

// Checking instruction format for type U instructions
- $(\texttt{is-lui}) \cdot (\texttt{0b0110111} + \texttt{op-a} \cdot 2^7 + \texttt{op-c} \cdot 2^{12} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-auipc}) \cdot (\texttt{0b0010111} + \texttt{op-a} \cdot 2^7 + \texttt{op-c} \cdot 2^{12} - \texttt{instr}_{\mathsf{val}}) = 0$

// Enforcing $\texttt{op-b} = 0$ for lui, auipc
- $(\texttt{is-type-u}) \cdot (\texttt{op-b}) = 0$

// Enforcing $\texttt{op-b}_{\mathsf{val}} = 0$ for lui, auipc, jal
- $(\texttt{is-type-u}) \cdot (\texttt{op-b}_{\mathsf{val}}) = 0$

Type J – JAL instruction

// Type J instruction - jal
// op-a is ranged checked via reg memory checking
// op-b is set to 0
// op-c range check follows from other range checks below
// Making sure that op-c is consistent with the immediate parts
- $\texttt{is-type-j} \cdot (\texttt{op-c1-10} + \texttt{op-c11} \cdot 2^{10} + \texttt{op-c12-19} \cdot 2^{11} + \texttt{op-c20} \cdot 2^{19} - \texttt{op-c}) = 0$

// Setting lower bit to 0 and performing sign extension to compute $\texttt{op-c}_{\mathsf{val}}$ from op-c
- $\texttt{is-type-j} \cdot (\texttt{op-c1-10} \cdot 2 + \texttt{op-c11} \cdot 2^{11} + \texttt{op-c12-19} \cdot 2^{12} + \texttt{op-c20} \cdot 2^{20} \cdot (2^{12} - 1) - \texttt{op-c}_{\mathsf{val}}) = 0$

// Range checking the different immediate parts
- $\texttt{is-type-j} \cdot (\texttt{op-c1-10} \in \Big[0, 2^{10} - 1\Big])$
- $\texttt{is-type-j} \cdot (\texttt{op-c11}) \cdot (1 - \texttt{op-c11}) = 0$
- $\texttt{is-type-j} \cdot (\texttt{op-c12-19} \in \Big[0, 2^{8} - 1\Big])$
- $\texttt{is-type-j} \cdot (\texttt{op-c20}) \cdot (1 - \texttt{op-c20}) = 0$

// Checking instruction format for type J instructions
- $(\texttt{is-jal}) \cdot (\texttt{0b1101111} + \texttt{op-a} \cdot 2^7 + \texttt{op-c12-19} \cdot 2^{12} + \texttt{op-c11} \cdot 2^{20} + \texttt{op-c1-10} \cdot 2^{21} + \texttt{op-c20} \cdot 2^{31} - \texttt{instr}_{\mathsf{val}}) = 0$

// Enforcing $\texttt{op-b} = 0$ for type J instructions
- $(\texttt{is-type-j}) \cdot (\texttt{op-b}) = 0$

// Enforcing $\texttt{op-b}_{\mathsf{val}} = 0$ for type J instructions
- $(\texttt{is-type-j}) \cdot (\texttt{op-b}_{\mathsf{val}}) = 0$

Type S – Store instructions sb, sh, sw

// Type S - Store instructions sb, sh, sw
// Making sure that op-c is consistent with the immediate parts
- $\texttt{is-type-s} \cdot (\texttt{op-c0-4} + \texttt{op-c5-10} \cdot 2^5 + \texttt{op-c11} \cdot 2^{10} - \texttt{op-c}) = 0$

// Performing sign extension to compute $\texttt{op-c}_{\mathsf{val}}$ from op-c
- $\texttt{is-type-s} \cdot (\texttt{op-c0-4} + \texttt{op-c5-10} \cdot 2^5 + \texttt{op-c11} \cdot 2^{11} \cdot (2^{21} - 1) - \texttt{op-c}_{\mathsf{val}}) = 0$

// Range checking the different immediate parts
- $\texttt{is-type-s} \cdot (\texttt{op-c0-4} \in \Big[0, 2^{5} - 1\Big])$
- $\texttt{is-type-s} \cdot (\texttt{op-c5-10} \in \Big[0, 2^{5} - 1\Big])$
- $\texttt{is-type-s} \cdot (\texttt{op-c11}) \cdot (1 - \texttt{op-c11}) = 0$

Load instructions + JALR + ALU instructions with non-shift immediate values

// Load instructions - lb, lh, lw, lbu, lhu

// ALU instructions with non-shift immediate values - add, slt, sltu, xor, and instructions with imm-c $= 1$
// op-a, op-b are ranged checked via reg memory checking
// Making sure that op-c is consistent with the immediate parts

- $(\texttt{is-load} + \texttt{is-alu-imm-no-shift} + \texttt{is-jalr}) \cdot (\texttt{op-c0-10} + \texttt{op-c11} \cdot 2^{11} - \texttt{op-c}) = 0$

// Performing sign extension to compute op-c$_{\mathsf{val}}$ from op-c

- $(\texttt{is-load} + \texttt{is-alu-imm-no-shift} + \texttt{is-jalr}) \cdot (\texttt{op-c0-10} + \texttt{op-c11} \cdot 2^{11} \cdot (2^{21} - 1) - \texttt{op-c}_{\mathsf{val}}) = 0$

// Range checking the different immediate parts

- $(\texttt{is-load} + \texttt{is-alu-imm-no-shift} + \texttt{is-jalr}) \cdot (\texttt{op-c0-10} \in \left[0, 2^{11} - 1\right])$
- $(\texttt{is-load} + \texttt{is-alu-imm-no-shift} + \texttt{is-jalr}) \cdot (\texttt{op-c11}) \cdot (1 - \texttt{op-c11}) = 0$

// Checking instruction format for load instructions

- $(\texttt{is-lb}) \cdot (\texttt{0b0000011} + \texttt{op-a} \cdot 2^7 + \texttt{0b000} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-lh}) \cdot (\texttt{0b0000011} + \texttt{op-a} \cdot 2^7 + \texttt{0b001} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-lw}) \cdot (\texttt{0b0000011} + \texttt{op-a} \cdot 2^7 + \texttt{0b010} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-lbu}) \cdot (\texttt{0b0000011} + \texttt{op-a} \cdot 2^7 + \texttt{0b100} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-lhu}) \cdot (\texttt{0b0000011} + \texttt{op-a} \cdot 2^7 + \texttt{0b101} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} - \texttt{instr}_{\mathsf{val}}) = 0$

// Checking instruction format for the jalr instruction

- $(\texttt{is-jalr}) \cdot (\texttt{0b1100111} + \texttt{op-a} \cdot 2^7 + \texttt{0b000} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} - \texttt{instr}_{\mathsf{val}}) = 0$

// Checking format for ALU Instructions with non-shift immediate values

- $(\texttt{is-add}) \cdot (\texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b000} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b010} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-sltu}) \cdot (\texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b011} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} + -\texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-xor}) \cdot (\texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b100} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-or}) \cdot (\texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b110} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-and}) \cdot (\texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b111} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} - \texttt{instr}_{\mathsf{val}}) = 0$

ALU instructions with shift immediate values – sll, srl, sra instructions with imm-c $= 1$

// ALU instructions with shift immediate values - sll, srl, sra instructions with imm-c $= 1$
// Making sure that op-c is consistent with the immediate parts

- $(\texttt{is-alu-imm-shift}) \cdot (\texttt{op-c0-4} - \texttt{op-c}) = 0$

// Setting op-c$_{\mathsf{val}}$ to op-c0-4

- $(\texttt{is-alu-imm-shift}) \cdot (\texttt{op-c0-4} - \texttt{op-c}_{\mathsf{val}}) = 0$

// Range checking the different immediate parts

- $(\texttt{is-alu-imm-shift}) \cdot (\texttt{op-c0-4} \in \left[0, 2^5 - 1\right])$

// Checking format for ALU Instructions with shift immediate values

- $(\texttt{is-sll}) \cdot (\texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b001} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} + \texttt{0b0000000} \cdot 2^{25} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b101} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} + \texttt{0b0000000} \cdot 2^{25} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b101} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} + \texttt{0b0100000} \cdot 2^{25} - \texttt{instr}_{\mathsf{val}}) = 0$

Type R - ALU instructions without immediate values – ALU instructions with imm-c $= 0$

// Type R instructions - add, sub, slt, sltu, xor, or, and, sll, srl, sra instructions with imm-c $= 0$
// op-a, op-b, op-c are ranged checked via reg memory checking
// Checking format for type R instructions

- $(\texttt{is-add}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b000} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} + \texttt{0b0000000} \cdot 2^{25} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-sub}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b000} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} + \texttt{0b0100000} \cdot 2^{25} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-sll}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b001} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} + \texttt{0b0000000} \cdot 2^{25} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-slt}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b010} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} + \texttt{0b0000000} \cdot 2^{25} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-sltu}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b011} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} + \texttt{0b0000000} \cdot 2^{25} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-xor}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b100} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} + \texttt{0b0000000} \cdot 2^{25} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-srl}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b101} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} + \texttt{0b0000000} \cdot 2^{25} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-sra}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b101} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} + \texttt{0b0100000} \cdot 2^{25} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-or}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b110} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} + \texttt{0b0000000} \cdot 2^{25} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-and}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{0b0010011} + \texttt{op-a} \cdot 2^7 + \texttt{0b111} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15} + \texttt{op-c} \cdot 2^{20} + \texttt{0b0000000} \cdot 2^{25} - \texttt{instr}_{\mathsf{val}}) = 0$

Type B – Branch instructions beq, bne, blt, bge, bltu, bgeu

// Type B - Branch instructions beq, bne, blt, bge, bltu, bgeu
// op-a, op-b, and op-c are ranged checked via reg memory checking
// Making sure that op-c is consistent with the immediate parts
- $\texttt{is-type-b} \cdot (\texttt{op-c1-4} + \texttt{op-c5-10} \cdot 2^{4} + \texttt{op-c11} \cdot 2^{10} + \texttt{op-c12} \cdot 2^{11} - \texttt{op-c}) = 0$

// Setting lower bit to 0 and performing sign extension to compute $\texttt{op-c}_{\mathsf{val}}$ from op-c
- $\texttt{is-type-b} \cdot (\texttt{op-c1-4} \cdot 2 + \texttt{op-c5-10} \cdot 2^{5} + \texttt{op-c11} \cdot 2^{11} + \texttt{op-c12} \cdot 2^{12} \cdot (2^{20} - 1) - \texttt{op-c}_{\mathsf{val}}) = 0$

// Range checking the different immediate parts
- $\texttt{is-type-b} \cdot (\texttt{op-c1-4} \in \Big[0, 2^{4} - 1\Big])$
- $\texttt{is-type-b} \cdot (\texttt{op-c5-10} \in \Big[0, 2^{5} - 1\Big])$
- $\texttt{is-type-b} \cdot (\texttt{op-c11}) \cdot (1 - \texttt{op-c11}) = 0$
- $\texttt{is-type-b} \cdot (\texttt{op-c12}) \cdot (1 - \texttt{op-c12}) = 0$

// Checking instruction format for branch instructions
- $(\texttt{is-beq}) \cdot (\texttt{0b1100011} + \texttt{op-c11} \cdot 2^{7} + \texttt{op-c1-4} \cdot 2^{8} + \texttt{0b000} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15}$
  $+\texttt{op-a} \cdot 2^{20} + \texttt{op-c5-10} \cdot 2^{25} + \texttt{op-c12} \cdot 2^{31} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-bne}) \cdot (\texttt{0b1100011} + \texttt{op-c11} \cdot 2^{7} + \texttt{op-c1-4} \cdot 2^{8} + \texttt{0b001} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15}$
  $+\texttt{op-a} \cdot 2^{20} + \texttt{op-c5-10} \cdot 2^{25} + \texttt{op-c12} \cdot 2^{31} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{0b1100011} + \texttt{op-c11} \cdot 2^{7} + \texttt{op-c1-4} \cdot 2^{8} + \texttt{0b100} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15}$
  $+\texttt{op-a} \cdot 2^{20} + \texttt{op-c5-10} \cdot 2^{25} + \texttt{op-c12} \cdot 2^{31} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{0b1100011} + \texttt{op-c11} \cdot 2^{7} + \texttt{op-c1-4} \cdot 2^{8} + \texttt{0b101} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15}$
  $+\texttt{op-a} \cdot 2^{20} + \texttt{op-c5-10} \cdot 2^{25} + \texttt{op-c12} \cdot 2^{31} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-bltu}) \cdot (\texttt{0b1100011} + \texttt{op-c11} \cdot 2^{7} + \texttt{op-c1-4} \cdot 2^{8} + \texttt{0b110} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15}$
  $+\texttt{op-a} \cdot 2^{20} + \texttt{op-c5-10} \cdot 2^{25} + \texttt{op-c12} \cdot 2^{31} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-bgeu}) \cdot (\texttt{0b1100011} + \texttt{op-c11} \cdot 2^{7} + \texttt{op-c1-4} \cdot 2^{8} + \texttt{0b111} \cdot 2^{12} + \texttt{op-b} \cdot 2^{15}$
  $+\texttt{op-a} \cdot 2^{20} + \texttt{op-c5-10} \cdot 2^{25} + \texttt{op-c12} \cdot 2^{31} - \texttt{instr}_{\mathsf{val}}) = 0$

System instructions – ecall, ebreak

// System instructions - ecall, ebreak
// op-a value is set by the execution component depending on $R[\texttt{x17}]$
// op-b is set to x17
// op-c is set to 0
- $(\texttt{is-ecall}) \cdot (\texttt{0b1110011} + \texttt{0b00000} \cdot 2^{7} + \texttt{0b000} \cdot 2^{12} + \texttt{0b00000} \cdot 2^{15} + \texttt{0b000000000000} \cdot 2^{20} - \texttt{instr}_{\mathsf{val}}) = 0$
- $(\texttt{is-ebreak}) \cdot (\texttt{0b1110011} + \texttt{0b00000} \cdot 2^{7} + \texttt{0b000} \cdot 2^{12} + \texttt{0b00000} \cdot 2^{15} + \texttt{0b000000000001} \cdot 2^{20} - \texttt{instr}_{\mathsf{val}}) = 0$

// Enforcing $\texttt{op-c} = 0$ for ecall, ebreak
- $(\texttt{is-type-sys}) \cdot (\texttt{op-c}) = 0$

// Enforcing $\texttt{op-c}_{\mathsf{val}} = 0$ for ecall, ebreak
- $(\texttt{is-type-sys}) \cdot (\texttt{op-c}_{\mathsf{val}}) = 0$

// Enforcing $\texttt{op-b} = \texttt{x17}$ for ecall, ebreak
- $(\texttt{is-type-sys}) \cdot (\texttt{0b10001} - \texttt{op-b}) = 0$

UNIMP instruction – unimp

// UNIMP instruction - unimp
// op-a, op-c, op-c are all set to 0
- $(\texttt{is-unimp}) \cdot (\texttt{0b1110011} + \texttt{0b00000} \cdot 2^{7} + \texttt{0b001} \cdot 2^{12} + \texttt{0b00000} \cdot 2^{15} + \texttt{0b000000000011} \cdot 2^{20} - \texttt{instr}_{\mathsf{val}}) = 0$

// Enforcing $\texttt{op-a} = \texttt{op-b} = \texttt{op-c} = 0$ for unimp
- $(\texttt{is-unimp}) \cdot (\texttt{op-a}) = 0$
- $(\texttt{is-unimp}) \cdot (\texttt{op-b}) = 0$
- $(\texttt{is-unimp}) \cdot (\texttt{op-c}) = 0$

// Enforcing $\texttt{op-a}_{\mathsf{val}} = \texttt{op-b}_{\mathsf{val}} = \texttt{op-c}_{\mathsf{val}} = 0$ for unimp
- $(\texttt{is-unimp}) \cdot (\texttt{op-a}_{\mathsf{val}}) = 0$
- $(\texttt{is-unimp}) \cdot (\texttt{op-b}_{\mathsf{val}}) = 0$
- $(\texttt{is-unimp}) \cdot (\texttt{op-c}_{\mathsf{val}}) = 0$

4.2.4 Interactions with other components

As mentioned above, the CPU component needs to interact with a few other components:

- Interaction with program memory: To read the next instruction using the program counter;
- Interaction with register memory: to read values associated with operands;
- Interaction with instruction execution: to enforce correction execution of instructions.

Program memory interaction

- $\texttt{instr}_{\textsf{val}} \leftarrow \text{Read}_{\text{Prog}}(\texttt{pc}, \texttt{clk})$

Register memory interaction

// Type S + B instructions do not have a destination register
// Type S + B instructions instead read from the op-a register
// Hence, we obtain $\texttt{op-a}_{\textsf{val}}$ by reading from the op-a register

- $(\texttt{is-type-s} + \texttt{is-type-b})\text{Read}_{\text{Reg}}(\texttt{op-a}, \texttt{clk}, 3) - \texttt{op-a}_{\textsf{val}}) = 0$

// Type R + I + U + J instructions have a destination register
// Hence, we need to write $\texttt{op-a}_{\textsf{val}}$ to the op-a register

- $(\texttt{is-type-r} + \texttt{is-type-i} + \texttt{is-type-u} + \texttt{is-type-j}) \cdot \text{Write}_{\text{Reg}}(\texttt{op-a}, \texttt{op-a}_{\textsf{val}}, \texttt{clk}, 3)) = 0$

// Type SYS instructions will interact with the register memory directly
// $\texttt{op-a}_{\textsf{val}}$ is an input provided by the environment for Type SYS instructions

// We only read from register op-b to obtain $\texttt{op-b}_{\textsf{val}}$ when $\texttt{op-b-flag} = 1$

- $(\texttt{op-b-flag}) \cdot \text{Read}_{\text{Reg}}(\texttt{op-b}, \texttt{clk}, 1) - \texttt{op-b}_{\textsf{val}}) = 0$

// We only need to read from register op-c to obtain $\texttt{op-c}_{\textsf{val}}$ for Type R instructions

- $(\texttt{is-type-r}) \cdot \text{Read}_{\text{Reg}}(\texttt{op-c}, \texttt{clk}, 2) - \texttt{op-c}_{\textsf{val}}) = 0$

Execution component interaction

- $\textsf{pc-next} \leftarrow \textsf{exec}(\textsf{pc}, \textsf{opcode}, \textsf{op-a}_{\textsf{val}}, \textsf{op-b}_{\textsf{val}}, \textsf{op-c}_{\textsf{val}})$

4.3 CPU constraints assuming small fields

4.3.1 Range checks

// More limbs would be needed if $T \geq 2^{32}$

- $\texttt{clk}^{(i)} \in \left[0, 2^8 - 1\right]$ for $i = 1, 2, 3, 4$

// immediate flags for operations op-b and op-c

- $(\texttt{op-b-flag})(1 - \texttt{op-b-flag}) = 0$
- $(\texttt{imm-c})(1 - \texttt{imm-c}) = 0$

// $\texttt{pc}^{(i)} \in \left[0, 2^8 - 1\right]$ for $i = 1, 2, 3, 4$ - guaranteed via program memory checking
// $\textsf{pc-next}^{(i)} \in \left[0, 2^8 - 1\right]$ for $i = 1, 2, 3, 4$ - implied by arithmetic constraints
// $\texttt{instr}_{\textsf{val}}{}^{(i)} \in \left[0, 2^8 - 1\right]$ for $i = 1, 2, 3, 4$ - performed in the program memory component
// op-a range check - guaranteed via register memory checking / arithmetic constraints
// op-b range check - guaranteed via register memory checking when $\texttt{op-b-flag} = 1$
// op-c range check - guaranteed via register memory checking when $\texttt{imm-c} = 0$
// $\texttt{op-a}_{\textsf{val}}{}^{(i)}$ range check for $i = 1, 2, 3, 4$ - performed in the register memory component
// $\texttt{op-b}_{\textsf{val}}{}^{(i)}$ range check for $i = 1, 2, 3, 4$ - performed in the register memory component when $\texttt{op-b-flag} = 1$
// $\texttt{op-c}_{\textsf{val}}{}^{(i)}$ range check for $i = 1, 2, 3, 4$ - performed in the via register memory checking when $\texttt{imm-c} = 0$

4.3.2 Instruction flag constraints

Remark 4.8 Since all constraints in the remaining of this section are for the same row, we do not explicitly write down the row index i when describing these constraints.

// Enforcing instruction flags are either 0 or 1
- $(\texttt{is-lui}) \cdot (1 - \texttt{is-lui}) = 0$ ▷ lui instruction
- $(\texttt{is-auipc}) \cdot (1 - \texttt{is-auipc}) = 0$ ▷ auipc instruction
- $(\texttt{is-jal}) \cdot (1 - \texttt{is-jal}) = 0$ ▷ jal instruction
- $(\texttt{is-jalr}) \cdot (1 - \texttt{is-jalr}) = 0$ ▷ jalr instruction
- $(\texttt{is-ecall}) \cdot (1 - \texttt{is-ecall}) = 0$ ▷ ecall instruction
- $(\texttt{is-ebreak}) \cdot (1 - \texttt{is-ebreak}) = 0$ ▷ ebreak instruction
- $(\texttt{is-fence}) \cdot (1 - \texttt{is-fence}) = 0$ ▷ fence instruction
- $(\texttt{is-unimp}) \cdot (1 - \texttt{is-unimp}) = 0$ ▷ unimp instruction
- $(\texttt{is-beq}) \cdot (1 - \texttt{is-beq}) = 0$ ▷ beq instruction
- $(\texttt{is-bne}) \cdot (1 - \texttt{is-bne}) = 0$ ▷ bne instruction
- $(\texttt{is-blt}) \cdot (1 - \texttt{is-blt}) = 0$ ▷ blt instruction
- $(\texttt{is-bge}) \cdot (1 - \texttt{is-bge}) = 0$ ▷ bge instruction
- $(\texttt{is-bltu}) \cdot (1 - \texttt{is-bltu}) = 0$ ▷ bltu instruction
- $(\texttt{is-bgeu}) \cdot (1 - \texttt{is-bgeu}) = 0$ ▷ bgeu instruction
- $(\texttt{is-lb}) \cdot (1 - \texttt{is-lb}) = 0$ ▷ lb instruction
- $(\texttt{is-lh}) \cdot (1 - \texttt{is-lh}) = 0$ ▷ lh instruction
- $(\texttt{is-lw}) \cdot (1 - \texttt{is-lw}) = 0$ ▷ lw instruction
- $(\texttt{is-lbu}) \cdot (1 - \texttt{is-lbu}) = 0$ ▷ lbu instruction
- $(\texttt{is-lhu}) \cdot (1 - \texttt{is-lhu}) = 0$ ▷ lhu instruction
- $(\texttt{is-sb}) \cdot (1 - \texttt{is-sb}) = 0$ ▷ sb instruction
- $(\texttt{is-sh}) \cdot (1 - \texttt{is-sh}) = 0$ ▷ sh instruction
- $(\texttt{is-sw}) \cdot (1 - \texttt{is-sw}) = 0$ ▷ sw instruction
- $(\texttt{is-add}) \cdot (1 - \texttt{is-add}) = 0$ ▷ add or addi instruction
- $(\texttt{is-sub}) \cdot (1 - \texttt{is-sub}) = 0$ ▷ sub instruction
- $(\texttt{is-sll}) \cdot (1 - \texttt{is-sll}) = 0$ ▷ sll or slli instruction
- $(\texttt{is-slt}) \cdot (1 - \texttt{is-slt}) = 0$ ▷ slt or slti instruction
- $(\texttt{is-sltu}) \cdot (1 - \texttt{is-sltu}) = 0$ ▷ sltu or sltiu instruction
- $(\texttt{is-xor}) \cdot (1 - \texttt{is-xor}) = 0$ ▷ xor or xori instruction
- $(\texttt{is-srl}) \cdot (1 - \texttt{is-srl}) = 0$ ▷ srl or srli instruction
- $(\texttt{is-sra}) \cdot (1 - \texttt{is-sra}) = 0$ ▷ sra or srai instruction
- $(\texttt{is-or}) \cdot (1 - \texttt{is-or}) = 0$ ▷ or or ori instruction
- $(\texttt{is-and}) \cdot (1 - \texttt{is-and}) = 0$ ▷ and or andi instruction
- $(\texttt{is-pad}) \cdot (1 - \texttt{is-pad}) = 0$ ▷ used for padding

// Enforcing exactly one instruction flag is set to 1
- $\texttt{is-lui} + \texttt{is-auipc} + \texttt{is-jal} + \texttt{is-jalr} + \texttt{is-ecall} + \texttt{is-ebreak} + \texttt{is-unimp} +$
  $\texttt{is-beq} + \texttt{is-bne} + \texttt{is-blt} + \texttt{is-bge} + \texttt{is-bltu} + \texttt{is-bgeu} +$
  $\texttt{is-lb} + \texttt{is-lh} + \texttt{is-lw} + \texttt{is-lbu} + \texttt{is-lhu} + \texttt{is-sb} + \texttt{is-sh} + \texttt{is-sw} +$
  $\texttt{is-add} + \texttt{is-sub} + \texttt{is-sll} + \texttt{is-slt} + \texttt{is-sltu} +$
  $\texttt{is-xor} + \texttt{is-srl} + \texttt{is-sra} + \texttt{is-or} + \texttt{is-and} + \texttt{is-pad} = 1$

// Matching flag with instruction opcode
- $(\texttt{is-lui}) \cdot (\texttt{opcode} - \texttt{LUI}) = 0$ ▷ lui instruction
- $(\texttt{is-auipc}) \cdot (\texttt{opcode} - \texttt{AUIPC}) = 0$ ▷ auipc instruction
- $(\texttt{is-jal}) \cdot (\texttt{opcode} - \texttt{JAL}) = 0$ ▷ jal instruction
- $(\texttt{is-jalr}) \cdot (\texttt{opcode} - \texttt{JALR}) = 0$ ▷ jalr instruction
- $(\texttt{is-ecall}) \cdot (\texttt{opcode} - \texttt{ECALL}) = 0$ ▷ ecall instruction
- $(\texttt{is-ebreak}) \cdot (\texttt{opcode} - \texttt{EBREAK}) = 0$ ▷ ebreak instruction
- $(\texttt{is-fence}) \cdot (\texttt{opcode} - \texttt{FENCE}) = 0$ ▷ fence instruction
- $(\texttt{is-unimp}) \cdot (\texttt{opcode} - \texttt{UNIMP}) = 0$ ▷ unimp instruction
- $(\texttt{is-beq}) \cdot (\texttt{opcode} - \texttt{BEQ}) = 0$ ▷ beq instruction
- $(\texttt{is-bne}) \cdot (\texttt{opcode} - \texttt{BNE}) = 0$ ▷ bne instruction
- $(\texttt{is-blt}) \cdot (\texttt{opcode} - \texttt{BLT}) = 0$ ▷ blt instruction

- $(\text{is-bge}) \cdot (\text{opcode} - \texttt{BGE}) = 0$ ▷ `bge instruction`
- $(\text{is-bltu}) \cdot (\text{opcode} - \texttt{BLTU}) = 0$ ▷ `bltu instruction`
- $(\text{is-bgeu}) \cdot (\text{opcode} - \texttt{BGEU}) = 0$ ▷ `bgeu instruction`
- $(\text{is-lb}) \cdot (\text{opcode} - \texttt{LB}) = 0$ ▷ `lb instruction`
- $(\text{is-lh}) \cdot (\text{opcode} - \texttt{LH}) = 0$ ▷ `lh instruction`
- $(\text{is-lw}) \cdot (\text{opcode} - \texttt{LW}) = 0$ ▷ `lw instruction`
- $(\text{is-lbu}) \cdot (\text{opcode} - \texttt{LBU}) = 0$ ▷ `lbu instruction`
- $(\text{is-lhu}) \cdot (\text{opcode} - \texttt{LHU}) = 0$ ▷ `lhu instruction`
- $(\text{is-sb}) \cdot (\text{opcode} - \texttt{SB}) = 0$ ▷ `sb instruction`
- $(\text{is-sh}) \cdot (\text{opcode} - \texttt{SH}) = 0$ ▷ `sh instruction`
- $(\text{is-sw}) \cdot (\text{opcode} - \texttt{SW}) = 0$ ▷ `sw instruction`
- $(\text{is-add}) \cdot (\text{opcode} - \texttt{ADD}) = 0$ ▷ `add or addi instruction`
- $(\text{is-sub}) \cdot (\text{opcode} - \texttt{SUB}) = 0$ ▷ `sub instruction`
- $(\text{is-sll}) \cdot (\text{opcode} - \texttt{SLL}) = 0$ ▷ `sll or slli instruction`
- $(\text{is-slt}) \cdot (\text{opcode} - \texttt{SLT}) = 0$ ▷ `slt or slti instruction`
- $(\text{is-sltu}) \cdot (\text{opcode} - \texttt{SLTU}) = 0$ ▷ `sltu or sltiu instruction`
- $(\text{is-xor}) \cdot (\text{opcode} - \texttt{XOR}) = 0$ ▷ `xor or xori instruction`
- $(\text{is-srl}) \cdot (\text{opcode} - \texttt{SRL}) = 0$ ▷ `srl or srli instruction`
- $(\text{is-sra}) \cdot (\text{opcode} - \texttt{SRA}) = 0$ ▷ `sra or srai instruction`
- $(\text{is-or}) \cdot (\text{opcode} - \texttt{OR}) = 0$ ▷ `or or ori instruction`
- $(\text{is-and}) \cdot (\text{opcode} - \texttt{AND}) = 0$ ▷ `and or andi instruction`

Remark 4.9 In the constraints defined above:

- $\texttt{LUI}, \texttt{AUIPC}, \ldots, \texttt{OR}, \texttt{AND}$ are assumed to be predefined constants for the opcodes.
- The constraints above actually refer to these constants as field elements.
- For instance, if $\texttt{XYZ} = \texttt{0b000100}$, then $\texttt{XYZ} = 4$ in the constraint equation.

// `Instruction types`

- $\texttt{is-type-u} = \texttt{is-lui} + \texttt{is-auipc}$ ▷ `Type U`
- $\texttt{is-type-j} = \texttt{is-jal}$ ▷ `Type J - JAL instruction`
- $\texttt{is-load} = \texttt{is-lb} + \texttt{is-lh} + \texttt{is-lw} + \texttt{is-lbu} + \texttt{is-lhu}$ ▷ `Load instructions`
- $\texttt{is-type-s} = \texttt{is-sb} + \texttt{is-sh} + \texttt{is-sw}$ ▷ `Type S - Store instructions`
- $\texttt{is-type-b} = \texttt{is-beq} + \texttt{is-bne} + \texttt{is-blt} + \texttt{is-bge} + \texttt{is-bltu} + \texttt{is-bgeu}$ ▷ `Type B - Branch instructions`
- $\texttt{is-type-sys} = \texttt{is-ecall} + \texttt{is-ebreak}$ ▷ `System calls`

// `Type R instructions`

- $\texttt{is-type-r} = (1 - \texttt{imm-c}) \cdot (\texttt{is-add} + \texttt{is-sub} + \texttt{is-slt} + \texttt{is-sltu} + \texttt{is-xor} + \texttt{is-or} + \texttt{is-and} + \texttt{is-sll} + \texttt{is-srl} + \texttt{is-sra})$

// `ALU instructions`

- $\texttt{is-alu} = \texttt{is-add} + \texttt{is-sub} + \texttt{is-slt} + \texttt{is-sltu} + \texttt{is-xor} + \texttt{is-or} + \texttt{is-and} + \texttt{is-sll} + \texttt{is-srl} + \texttt{is-sra}$
- $\texttt{is-alu-imm-shift} = \texttt{imm-c} \cdot (\texttt{is-sll} + \texttt{is-srl} + \texttt{is-sra})$
- $\texttt{is-alu-imm-no-shift} = \texttt{imm-c} \cdot (\texttt{is-add} + \texttt{is-slt} + \texttt{is-sltu} + \texttt{is-xor} + \texttt{is-or} + \texttt{is-and})$

// `Type I instructions with non-shift immediate values`

- $\texttt{is-type-i-no-shift} = \texttt{is-load} + \texttt{is-alu-imm-no-shift} + \texttt{is-jalr}$

// `Type I instructions`

- $\texttt{is-type-i} = \texttt{is-load} + \texttt{is-alu-imm-no-shift} + \texttt{is-alu-imm-shift} + \texttt{is-jalr}$

4.3.3 Arithmetic constraints

Boundary constraints

// `Setting` $\texttt{pc}[1] = \texttt{PC-INIT}$ `- this is a constant set during the two-pass phase`

- $\texttt{pc}[1]^{(1)} = \texttt{PC-INIT}^{(1)}$
- $\texttt{pc}[1]^{(2)} = \texttt{PC-INIT}^{(2)}$
- $\texttt{pc}[1]^{(3)} = \texttt{PC-INIT}^{(3)}$
- $\texttt{pc}[1]^{(4)} = \texttt{PC-INIT}^{(4)}$

// Setting $\texttt{clk}[1] = 1$

- $\texttt{clk}[1]^{(1)} = 1$
- $\texttt{clk}[1]^{(2)} = 0$
- $\texttt{clk}[1]^{(3)} = 0$
- $\texttt{clk}[1]^{(4)} = 0$

Multi-row constraints

// Transition constraints for the program counter $\texttt{pc}[i]$ at row $i > 1$

- $\texttt{pc}[i+1]^{(1)} = \texttt{pc-next}[i]^{(1)}$ $\quad\triangleright$ unless $i+1 \equiv 0 \bmod \texttt{num-rows}$ or $\texttt{is-pad}[i+1] = 1$
- $\texttt{pc}[i+1]^{(2)} = \texttt{pc-next}[i]^{(2)}$ $\quad\triangleright$ unless $i+1 \equiv 0 \bmod \texttt{num-rows}$ or $\texttt{is-pad}[i+1] = 1$
- $\texttt{pc}[i+1]^{(3)} = \texttt{pc-next}[i]^{(3)}$ $\quad\triangleright$ unless $i+1 \equiv 0 \bmod \texttt{num-rows}$ or $\texttt{is-pad}[i+1] = 1$
- $\texttt{pc}[i+1]^{(4)} = \texttt{pc-next}[i]^{(4)}$ $\quad\triangleright$ unless $i+1 \equiv 0 \bmod \texttt{num-rows}$ or $\texttt{is-pad}[i+1] = 1$

// Transition constraints for $\texttt{clk}[i]$ at row $i > 1$
// $\texttt{clk-carry}^{(j)}$ for $j = 1, 2, 3, 4$ used for handling carries

- $\texttt{clk}[i]^{(1)} + \texttt{clk-carry}[i]^{(1)} \cdot 2^8 = \texttt{clk}[i-1]^{(1)} + 1$
- $\texttt{clk}[i]^{(2)} + \texttt{clk-carry}[i]^{(2)} \cdot 2^8 = \texttt{clk}[i-1]^{(2)} + \texttt{clk-carry}[i]^{(1)}$
- $\texttt{clk}[i]^{(3)} + \texttt{clk-carry}[i]^{(3)} \cdot 2^8 = \texttt{clk}[i-1]^{(3)} + \texttt{clk-carry}[i]^{(2)}$
- $\texttt{clk}[i]^{(4)} + \texttt{clk-carry}[i]^{(4)} \cdot 2^8 = \texttt{clk}[i-1]^{(4)} + \texttt{clk-carry}[i]^{(3)}$

// Enforcing $\texttt{clk-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\texttt{clk-carry}[i]^{(1)}) \cdot (1 - \texttt{clk-carry}[i]^{(1)}) = 0$
- $(\texttt{clk-carry}[i]^{(2)}) \cdot (1 - \texttt{clk-carry}[i]^{(2)}) = 0$
- $(\texttt{clk-carry}[i]^{(3)}) \cdot (1 - \texttt{clk-carry}[i]^{(3)}) = 0$
- $(\texttt{clk-carry}[i]^{(4)}) \cdot (1 - \texttt{clk-carry}[i]^{(4)}) = 0$

Remark 4.10 We should raise a clock overflow error if $\texttt{clk-carry}[i]^{(4)} = 1$. In that case, we will need to increase the number of limbs for $\texttt{clk}$ and other timestamps.

Remark 4.11 Since all constraints in the remaining of this section are for the same row, we do not explicitly write down the row index i when describing these constraints.

Common constraints

Remark 4.12 $\texttt{op-a}_{\mathsf{val}}\texttt{-effective}$ is equal to $\texttt{op-a}_{\mathsf{val}}$ (assuming $\texttt{op-a} \neq 0$) or 0 (otherwise).

// Ensuring that pc is a multiple of 4

- $\texttt{pc-aux}^{(1)} \cdot 4 - \texttt{pc}^{(1)} = 0$

// Enforcing $\texttt{pc-aux}^{(1)} \in \left[0, 2^6 - 1\right]$

- $\texttt{pc-aux}^{(1)} \in \left[0, 2^6 - 1\right]$

// Ensuring that $\texttt{op-b-flag} = 1$ for all instructions except lui, auipc, jal, unimp

- $(\texttt{is-sb} + \texttt{is-sh} + \texttt{is-sw} + \texttt{is-lb} + \texttt{is-lh} + \texttt{is-lw} + \texttt{is-lbu} + \texttt{is-lhu} + \texttt{is-jalr} +$
  $\texttt{is-add} + \texttt{is-sub} + \texttt{is-slt} + \texttt{is-sltu} + \texttt{is-xor} + \texttt{is-or} + \texttt{is-and} + \texttt{is-sll} + \texttt{is-srl} + \texttt{is-sra} +$
  $\texttt{is-beq} + \texttt{is-bne} + \texttt{is-blt} + \texttt{is-bge} + \texttt{is-bltu} + \texttt{is-bgeu} + \texttt{is-ecall} + \texttt{is-ebreak} - \texttt{op-b-flag}) = 0$

// Ensuring that $\texttt{imm-c} = 1$ for non-ALU instructions
// Notice that imm-c can be 0 or 1 for ALU instructions

- $(\texttt{is-lui} + \texttt{is-auipc} + \texttt{is-jal} + \texttt{is-jalr} + \texttt{is-ecall} + \texttt{is-ebreak} +$
  $\texttt{is-sb} + \texttt{is-sh} + \texttt{is-sw} + \texttt{is-lb} + \texttt{is-lh} + \texttt{is-lw} + \texttt{is-lbu} + \texttt{is-lhu} +$
  $\texttt{is-beq} + \texttt{is-bne} + \texttt{is-blt} + \texttt{is-bge} + \texttt{is-bltu} + \texttt{is-bgeu})(1 - \texttt{imm-c}) = 0$

// Enforcing $\texttt{imm-c} = 0$ for sub

- $(\texttt{is-sub}) \cdot \texttt{imm-c} = 0$

// Determining op-a$_{\text{val}}$-effective
// op-a$_{\text{val}}$-effective-flag is an auxiliary variable
// op-a$_{\text{val}}$-effective-flag $= 1$ indicates op-a is non-zero
// op-a$_{\text{val}}$-effective-flag $= 0$ indicates op-a is zero

- $\text{op-a} \cdot \text{op-a}_{\text{val}}\text{-effective-flag-aux} = \text{op-a}_{\text{val}}\text{-effective-flag}$

// Ensuring op-a$_{\text{val}}$-effective-flag-aux $\neq 0$

- $\text{op-a}_{\text{val}}\text{-effective-flag-aux} \cdot \text{op-a}_{\text{val}}\text{-effective-flag-aux-inv} = 1$

// Enforcing op-a$_{\text{val}}$-effective-flag $\in \{0, 1\}$

- $(\text{op-a}_{\text{val}}\text{-effective-flag}) \cdot (1 - \text{op-a}_{\text{val}}\text{-effective-flag}) = 0$

// Enforcing relation between op-a$_{\text{val}}$ and op-a$_{\text{val}}$-effective

- ${\text{op-a}_{\text{val}}}^{(1)} \cdot \text{op-a}_{\text{val}}\text{-effective-flag} = \text{op-a}_{\text{val}}\text{-effective}^{(1)}$
- ${\text{op-a}_{\text{val}}}^{(2)} \cdot \text{op-a}_{\text{val}}\text{-effective-flag} = \text{op-a}_{\text{val}}\text{-effective}^{(2)}$
- ${\text{op-a}_{\text{val}}}^{(3)} \cdot \text{op-a}_{\text{val}}\text{-effective-flag} = \text{op-a}_{\text{val}}\text{-effective}^{(3)}$
- ${\text{op-a}_{\text{val}}}^{(4)} \cdot \text{op-a}_{\text{val}}\text{-effective-flag} = \text{op-a}_{\text{val}}\text{-effective}^{(4)}$

Remark 4.13 imm-c can be 0 or 1 for ALU instructions, except for sub instructions for which imm-c must be 0.

Type U – LUI and AUIPC instructions

For Type U instructions (LUI, AUIPC):

- op-a is a destination register selector
- op-a$_{\text{val}}$ is obtained from the instruction execution component
- op-b $= 0$ and ${\text{op-b}_{\text{val}}}^{(j)} = 0$ for $j = 1, 2, 3, 4$
- op-c is a 20-bit U-immediate value (see Section 2.4)

Constraints:

// Type U instructions - lui and auipc
// op-a is ranged checked via reg memory checking
// op-b is set to 0
// op-c range check follows from other range checks below
// Making sure that op-c is consistent with the immediate parts

- $\text{is-type-u} \cdot (\text{op-c12-15} + \text{op-c16-23} \cdot 2^4 + \text{op-c24-31} \cdot 2^{12} - \text{op-c}) = 0$

// Setting lower 12 bits to 0 in order to compute op-c$_{\text{val}}$ from op-c

- $\text{is-type-u} \cdot ({\text{op-c}_{\text{val}}}^{(1)}) = 0$
- $\text{is-type-u} \cdot (\text{op-c12-15} \cdot 2^4 - {\text{op-c}_{\text{val}}}^{(2)}) = 0$
- $\text{is-type-u} \cdot (\text{op-c16-23} - {\text{op-c}_{\text{val}}}^{(3)}) = 0$
- $\text{is-type-u} \cdot (\text{op-c24-31} - {\text{op-c}_{\text{val}}}^{(4)}) = 0$

// Range checking the different op-c immediate parts

- $\text{is-type-u} \cdot (\text{op-c12-15} \in \left[0, 2^4 - 1\right])$
- $\text{is-type-u} \cdot (\text{op-c16-23} \in \left[0, 2^8 - 1\right])$
- $\text{is-type-u} \cdot (\text{op-c24-31} \in \left[0, 2^8 - 1\right])$

// Making sure that op-a is consistent with the immediate parts

- $\text{is-type-u} \cdot (\text{op-a0} + \text{op-a1-4} \cdot 2 - \text{op-a}) = 0$

// Making sure that op-a immediate parts

- $\text{is-type-u} \cdot (\text{op-a1}) \cdot (1 - \text{op-a1}) = 0$
- $\text{is-type-u} \cdot (\text{op-a1-4} \in \left[0, 2^4 - 1\right])$

// Enforcing op-b $= 0$ for lui, auipc

- $\text{is-type-u} \cdot (\text{op-b}) = 0$

// Enforcing op-b$_{\text{val}} = 0$ for lui, auipc

- $(\text{is-type-u}) \cdot ({\text{op-b}_{\text{val}}}^{(1)}) = 0$
- $(\text{is-type-u}) \cdot ({\text{op-b}_{\text{val}}}^{(2)}) = 0$

- $(\texttt{is-type-u}) \cdot (\texttt{op-b}_{\texttt{val}}{}^{(3)}) = 0$
- $(\texttt{is-type-u}) \cdot (\texttt{op-b}_{\texttt{val}}{}^{(4)}) = 0$

// `Checking instruction format for type U instructions`

- $(\texttt{is-lui}) \cdot (\texttt{0b0110111} + \texttt{op-a0} \cdot 2^7 - \texttt{instr}_{\texttt{val}}{}^{(1)}) = 0$ ▷ `limb 1 for lui`
- $(\texttt{is-auipc}) \cdot (\texttt{0b0010111} + \texttt{op-a0} \cdot 2^7 - \texttt{instr}_{\texttt{val}}{}^{(1)}) = 0$ ▷ `limb 1 for auipc`
- $(\texttt{is-type-u}) \cdot (\texttt{op-a1-4} + \texttt{op-c12-15} \cdot 2^4 - \texttt{instr}_{\texttt{val}}{}^{(2)}) = 0$ ▷ `limb 2`
- $(\texttt{is-type-u}) \cdot (\texttt{op-c16-23} - \texttt{instr}_{\texttt{val}}{}^{(3)}) = 0$ ▷ `limb 3`
- $(\texttt{is-type-u}) \cdot (\texttt{op-c24-31} - \texttt{instr}_{\texttt{val}}{}^{(4)}) = 0$ ▷ `limb 4`

Motivational explanations:

- The goal of the above constraints is to use $\texttt{instr}_{\texttt{val}}$ (which is uniquely determined by the program memory checking) to uniquely determine the columns relevant for type U instruction execution.
- Other constraints somewhere else make sure that exactly one of the instruction flags such as `is-lui`, `is-auipc` is set to one and the rest is zero. The above constraints come into play when the prover sets either `is-lui` or `is-auipc`. In those cases, `is-type-u` is also set to one.
- The constraints involving $\texttt{instr}_{\texttt{val}}{}^{(1)}$ make sure that `is-lui` or `is-auipc` can be set only when $\texttt{instr}_{\texttt{val}}$ contains the corresponding opcode.
- The above constraints never force the prover to set `is-lui` or `is-auipc`. The prover sets one of these flags just because the prover needs to set one instruction flag, and they cannot set any other instruction flags (because of the opcode-checking constraints about the other instruction flags).
- The prover can alternatively set `is-pad` to 1, but once the prover does so, the prover needs to keep `is-pad` set to 1 until the end of the trace.
- The above constraints involving $\texttt{instr}_{\texttt{val}}$ (together with the range check) uniquely determine `op-a0`, `op-a1-4`, `op-c12-15`, `op-c16-23`, `op-c24-31`.
- Other constraints above determine `op-a` , `op-c` and $\texttt{op-c}_{\texttt{val}}$ uniquely as a linear combination of above. `op-b` and $\texttt{op-b}_{\texttt{val}}$ are constrained to be zero.
- The register memory checking uses `op-a` to choose the destination register.

Type J – JAL instruction

For Type J instructions (JAL):

- `op-a` is a destination register selector
- $\texttt{op-a}_{\texttt{val}}$ is obtained from the instruction execution component
- $\texttt{op-b} = 0$ and $\texttt{op-b}_{\texttt{val}}{}^{(j)} = 0$ for $j = 1, 2, 3, 4$
- `op-c` is a 20-bit J-immediate value (see Section 2.4)
- $\texttt{op-c}_{\texttt{val}}$ is obtained from `op-c` by setting bit 0 to 0 and, bits 1-20 to `op-c` , and sign extending it

Constraints:

// `Type J instructions - jal`
// `op-a is ranged checked via reg memory checking`
// `op-b is set to 0`
// `op-c range check follows from other range checks below`
// `Making sure that op-c is consistent with the immediate parts`

- $\texttt{is-type-j} \cdot (\texttt{op-c1-3} + \texttt{op-c4-7} \cdot 2^3 + \texttt{op-c8-10} \cdot 2^7 + \texttt{op-c11} \cdot 2^{10} + \texttt{op-c12-15} \cdot 2^{11} + \texttt{op-c16-19} \cdot 2^{15} + \texttt{op-c20} \cdot 2^{19} - \texttt{op-c}) = 0$

// `Computing` $\texttt{op-c}_{\texttt{val}}$ `limbs from op-c and performing sign extension`

- $\texttt{is-type-j} \cdot (\texttt{op-c1-3} \cdot 2 + \texttt{op-c4-7} \cdot 2^4 - \texttt{op-c}_{\texttt{val}}{}^{(1)}) = 0$
- $\texttt{is-type-j} \cdot (\texttt{op-c8-10} + \texttt{op-c11} \cdot 2^3 + \texttt{op-c12-15} \cdot 2^4 - \texttt{op-c}_{\texttt{val}}{}^{(2)}) = 0$
- $\texttt{is-type-j} \cdot (\texttt{op-c16-19} + \texttt{op-c20} \cdot (2^4 - 1) \cdot 2^4 - \texttt{op-c}_{\texttt{val}}{}^{(3)}) = 0$
- $\texttt{is-type-j} \cdot (\texttt{op-c20} \cdot (2^8 - 1) - \texttt{op-c}_{\texttt{val}}{}^{(4)}) = 0$

// `Range checking the different op-c immediate parts`

- $\texttt{is-type-j} \cdot (\texttt{op-c1-3} \in \left[0, 2^3 - 1\right])$

- $\texttt{is-type-j} \cdot (\mathsf{op\text{-}c4\text{-}7} \in \left[0, 2^4 - 1\right])$
- $\texttt{is-type-j} \cdot (\mathsf{op\text{-}c8\text{-}10} \in \left[0, 2^3 - 1\right])$
- $\texttt{is-type-j} \cdot (\mathsf{op\text{-}c11}) \cdot (1 - \mathsf{op\text{-}c11}) = 0$
- $\texttt{is-type-j} \cdot (\mathsf{op\text{-}c12\text{-}15} \in \left[0, 2^4 - 1\right])$
- $\texttt{is-type-j} \cdot (\mathsf{op\text{-}c12\text{-}15} \in \left[0, 2^4 - 1\right])$
- $\texttt{is-type-j} \cdot (\mathsf{op\text{-}c16\text{-}19} \in \left[0, 2^4 - 1\right])$
- $\texttt{is-type-j} \cdot (\mathsf{op\text{-}c20}) \cdot (1 - \mathsf{op\text{-}c20}) = 0$

// Making sure that op-a is consistent with the immediate parts
- $\texttt{is-type-j} \cdot (\mathsf{op\text{-}a0} + \mathsf{op\text{-}a1\text{-}4} \cdot 2 - \mathsf{op\text{-}a}) = 0$

// Making sure that op-a immediate parts
- $\texttt{is-type-j} \cdot (\mathsf{op\text{-}a1}) \cdot (1 - \mathsf{op\text{-}a1}) = 0$
- $\texttt{is-type-j} \cdot (\mathsf{op\text{-}a1\text{-}4} \in \left[0, 2^4 - 1\right])$

// Enforcing op-b $= 0$ for jal
- $\texttt{is-type-j} \cdot (\mathsf{op\text{-}b}) = 0$

// Enforcing op-b$_{\mathsf{val}}$ $= 0$ for jal
- $(\texttt{is-type-j}) \cdot (\mathsf{op\text{-}b_{val}}^{(1)}) = 0$
- $(\texttt{is-type-j}) \cdot (\mathsf{op\text{-}b_{val}}^{(2)}) = 0$
- $(\texttt{is-type-j}) \cdot (\mathsf{op\text{-}b_{val}}^{(3)}) = 0$
- $(\texttt{is-type-j}) \cdot (\mathsf{op\text{-}b_{val}}^{(4)}) = 0$

// Checking instruction format for type J instructions
- $(\texttt{is-lui}) \cdot (\texttt{0b1101111} + \mathsf{op\text{-}a0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(1)}) = 0$ ▷ limb 1
- $(\texttt{is-type-j}) \cdot (\mathsf{op\text{-}a1\text{-}4} + \mathsf{op\text{-}c12\text{-}15} \cdot 2^4 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2
- $(\texttt{is-type-j}) \cdot (\mathsf{op\text{-}c16\text{-}19} + \mathsf{op\text{-}c11} \cdot 2^4 + \mathsf{op\text{-}c1\text{-}3} \cdot 2^5 - \texttt{instr}_{\mathsf{val}}{}^{(3)}) = 0$ ▷ limb 3
- $(\texttt{is-type-j}) \cdot (\mathsf{op\text{-}c4\text{-}7} + \mathsf{op\text{-}c8\text{-}10} \cdot 2^4 + \mathsf{op\text{-}c20} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ limb 4

Type S – Store instructions SB, SH, SW

For Type S instructions (SB, SH, SW):

- op-a is a source register selector
- op-a$_{\mathsf{val}}$ is obtained by reading from the register memory component
- op-b is another source register selector
- op-b$_{\mathsf{val}}$ is obtained by reading from the register memory component
- op-c is a 12-bit S-immediate value (see Section 2.4)
- op-c$_{\mathsf{val}}$ is obtained by sign extending the value of op-c

Constraints:

// Type J instructions - jal
// op-a is ranged checked via reg memory checking
// op-b is ranged checked via reg memory checking
// op-c range check follows from other range checks below
// Making sure that op-c is consistent with the immediate parts
- $\texttt{is-type-s} \cdot (\mathsf{op\text{-}c0} + \mathsf{op\text{-}c1\text{-}4} \cdot 2 + \mathsf{op\text{-}c5\text{-}7} \cdot 2^5 + \mathsf{op\text{-}c8\text{-}10} \cdot 2^8 + \mathsf{op\text{-}c11} \cdot 2^{11} - \mathsf{op\text{-}c}) = 0$

// Computing op-c$_{\mathsf{val}}$ limbs from op-c and performing sign extension
- $\texttt{is-type-s} \cdot (\mathsf{op\text{-}c0} + \mathsf{op\text{-}c1\text{-}4} \cdot 2 + \mathsf{op\text{-}c5\text{-}7} \cdot 2^5 - \mathsf{op\text{-}c_{val}}^{(1)}) = 0$
- $\texttt{is-type-s} \cdot (\mathsf{op\text{-}c8\text{-}10} + \mathsf{op\text{-}c11} \cdot (2^5 - 1) \cdot 2^3 - \mathsf{op\text{-}c_{val}}^{(2)}) = 0$
- $\texttt{is-type-s} \cdot (\mathsf{op\text{-}c11} \cdot (2^8 - 1) - \mathsf{op\text{-}c_{val}}^{(3)}) = 0$
- $\texttt{is-type-s} \cdot (\mathsf{op\text{-}c11} \cdot (2^8 - 1) - \mathsf{op\text{-}c_{val}}^{(4)}) = 0$

// Range checking the different op-c immediate parts
- $\texttt{is-type-s} \cdot (\mathsf{op\text{-}c0}) \cdot (1 - \mathsf{op\text{-}c0}) = 0$
- $\texttt{is-type-s} \cdot (\mathsf{op\text{-}c1\text{-}4} \in \left[0, 2^4 - 1\right])$
- $\texttt{is-type-s} \cdot (\mathsf{op\text{-}c5\text{-}7} \in \left[0, 2^3 - 1\right])$
- $\texttt{is-type-s} \cdot (\mathsf{op\text{-}c8\text{-}10} \in \left[0, 2^3 - 1\right])$
- $\texttt{is-type-s} \cdot (\mathsf{op\text{-}c11}) \cdot (1 - \mathsf{op\text{-}c11}) = 0$

// Making sure that op-a is consistent with the immediate parts

- $\texttt{is-type-s} \cdot (\texttt{op-a0} + \texttt{op-a1-4} \cdot 2 - \texttt{op-a}) = 0$

// Range checking the different op-a immediate parts

- $\texttt{is-type-s} \cdot (\texttt{op-a1}) \cdot (1 - \texttt{op-a1}) = 0$
- $\texttt{is-type-s} \cdot (\texttt{op-a1-4} \in \left[0, 2^4 - 1\right])$

// Making sure that op-b is consistent with the immediate parts

- $\texttt{is-type-s} \cdot (\texttt{op-b0-3} + \texttt{op-b4} \cdot 2^4 - \texttt{op-b}) = 0$

// Range checking the different op-b immediate parts

- $\texttt{is-type-s} \cdot (\texttt{op-b0-3} \in \left[0, 2^4 - 1\right])$
- $\texttt{is-type-s} \cdot (\texttt{op-b4}) \cdot (1 - \texttt{op-b1}) = 0$

// Checking instruction format for type S instructions

- $(\texttt{is-type-s}) \cdot (\texttt{0b0100011} + \texttt{op-c0} \cdot 2^7 - \texttt{instr}_{\texttt{val}}{}^{(1)}) = 0$ ▷ limb 1
- $(\texttt{is-sb}) \cdot (\texttt{op-c1-4} + \texttt{0b000} \cdot 2^4 + \texttt{op-a0} \cdot 2^7 - \texttt{instr}_{\texttt{val}}{}^{(2)}) = 0$ ▷ limb 2 for sb
- $(\texttt{is-sh}) \cdot (\texttt{op-c1-4} + \texttt{0b001} \cdot 2^4 + \texttt{op-a0} \cdot 2^7 - \texttt{instr}_{\texttt{val}}{}^{(2)}) = 0$ ▷ limb 2 for sh
- $(\texttt{is-sw}) \cdot (\texttt{op-c1-4} + \texttt{0b010} \cdot 2^4 + \texttt{op-a0} \cdot 2^7 - \texttt{instr}_{\texttt{val}}{}^{(2)}) = 0$ ▷ limb 2 for sw
- $(\texttt{is-type-s}) \cdot (\texttt{op-a1-4} + \texttt{op-b0-3} \cdot 2^4 - \texttt{instr}_{\texttt{val}}{}^{(3)}) = 0$ ▷ limb 3
- $(\texttt{is-type-s}) \cdot (\texttt{op-b4} + \texttt{op-c5-7} \cdot 2 + \texttt{op-c8-10} \cdot 2^4 + \texttt{op-c11} \cdot 2^7 - \texttt{instr}_{\texttt{val}}{}^{(4)}) = 0$ ▷ limb 4

Type I without shifts – Load + JALR + ALU instructions with non-shift immediates

For Type I instructions without shifts (Load + JALR + ALU with non-shift immediate values):

- op-a is a destination register selector
- $\texttt{op-a}_{\texttt{val}}$ is obtained from the instruction execution component
- $\texttt{op-a}_{\texttt{val}}\texttt{-effective}$ will be written to the op-a register
- op-b is a source register selector
- $\texttt{op-b}_{\texttt{val}}$ is obtained by reading from the register memory component
- op-c is a 12-bit I-immediate value (see Section 2.4)
- $\texttt{op-c}_{\texttt{val}}$ is obtained by sign extending the value of op-c

Constraints:

// Type I instructions with no shifts - Load + JALR + ALU with non-shift immediate values
// Load instructions - lb, lh, lw, lbu, lhu
// ALU instructions with non-shift immediate values - add, slt, sltu, xor, and instructions with imm-c = 1
// op-a is ranged checked via reg memory checking
// op-b is ranged checked via reg memory checking
// op-c range check follows from other range checks below
// Making sure that op-c is consistent with the immediate parts

- $(\texttt{is-type-i-no-shift}) \cdot (\texttt{op-c0-3} + \texttt{op-c4-7} \cdot 2^4 + \texttt{op-c8-10} \cdot 2^8 + \texttt{op-c11} \cdot 2^{11} - \texttt{op-c}) = 0$

// Performing sign extension to compute $\texttt{op-c}_{\texttt{val}}$ from op-c

- $(\texttt{is-type-i-no-shift}) \cdot (\texttt{op-c0-3} + \texttt{op-c4-7} \cdot 2^4 - \texttt{op-c}_{\texttt{val}}{}^{(1)}) = 0$
- $(\texttt{is-type-i-no-shift}) \cdot (\texttt{op-c8-10} + \texttt{op-c11} \cdot (2^5 - 1) \cdot 2^3 - \texttt{op-c}_{\texttt{val}}{}^{(2)}) = 0$
- $(\texttt{is-type-i-no-shift}) \cdot (\texttt{op-c11} \cdot (2^8 - 1) - \texttt{op-c}_{\texttt{val}}{}^{(3)}) = 0$
- $(\texttt{is-type-i-no-shift}) \cdot (\texttt{op-c11} \cdot (2^8 - 1) - \texttt{op-c}_{\texttt{val}}{}^{(4)}) = 0$

// Range checking the different op-c immediate parts

- $\texttt{is-type-i-no-shift} \cdot (\texttt{op-c0-3} \in \left[0, 2^4 - 1\right])$
- $\texttt{is-type-i-no-shift} \cdot (\texttt{op-c4-7} \in \left[0, 2^4 - 1\right])$
- $\texttt{is-type-i-no-shift} \cdot (\texttt{op-c8-10} \in \left[0, 2^3 - 1\right])$
- $\texttt{is-type-i-no-shift} \cdot (\texttt{op-c11}) \cdot (1 - \texttt{op-c11}) = 0$

// Making sure that op-a is consistent with the immediate parts

- $\texttt{is-type-i-no-shift} \cdot (\texttt{op-a0} + \texttt{op-a1-4} \cdot 2 - \texttt{op-a}) = 0$

// Range checking the different op-a immediate parts

- $\texttt{is-type-i-no-shift} \cdot (\texttt{op-a1}) \cdot (1 - \texttt{op-a1}) = 0$

- $\texttt{is-type-i-no-shift} \cdot (\texttt{op-a1-4} \in \left[0, 2^4 - 1\right])$

// Making sure that op-b is consistent with the immediate parts
- $\texttt{is-type-i-no-shift} \cdot (\texttt{op-b0} + \texttt{op-b1-4} \cdot 2 - \texttt{op-b}) = 0$

// Range checking the different op-b immediate parts
- $\texttt{is-type-i-no-shift} \cdot (\texttt{op-b0}) \cdot (1 - \texttt{op-b1}) = 0$
- $\texttt{is-type-i-no-shift} \cdot (\texttt{op-b1-4} \in \left[0, 2^4 - 1\right])$

// Checking instruction format for limb 1
- $(\texttt{is-load}) \cdot (\texttt{0b0000011} + \texttt{op-a0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(1)}) = 0$ ▷ limb 1 for load instructions
- $(\texttt{is-alu-imm-no-shift}) \cdot (\texttt{0b0010011} + \texttt{op-a0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(1)}) = 0$ ▷ limb 1 for ALU with non-shift imm
- $(\texttt{is-jalr}) \cdot (\texttt{0b1100111} + \texttt{op-a0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(1)}) = 0$ ▷ limb 1 for jalr

// Checking instruction format for limb 2
- $(\texttt{is-lb}) \cdot (\texttt{op-a1-4} + \texttt{0b000} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for lb
- $(\texttt{is-lh}) \cdot (\texttt{op-a1-4} + \texttt{0b001} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for lh
- $(\texttt{is-lw}) \cdot (\texttt{op-a1-4} + \texttt{0b010} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for lw
- $(\texttt{is-lbu}) \cdot (\texttt{op-a1-4} + \texttt{0b100} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for lbu
- $(\texttt{is-lhu}) \cdot (\texttt{op-a1-4} + \texttt{0b101} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for lhu
- $(\texttt{is-add}) \cdot (\texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b000} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for addi
- $(\texttt{is-slt}) \cdot (\texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b010} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for slti
- $(\texttt{is-sltu}) \cdot (\texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b011} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for sltiu
- $(\texttt{is-xor}) \cdot (\texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b100} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for xori
- $(\texttt{is-or}) \cdot (\texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b110} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for ori
- $(\texttt{is-and}) \cdot (\texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b111} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for andi
- $(\texttt{is-jalr})(\texttt{op-a1-4} + \texttt{0b000} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for jalr

// Checking instruction format for limb 3
- $(\texttt{is-type-i-no-shift}) \cdot (\texttt{op-b1-4} + \texttt{op-c0-3} \cdot 2^4 - \texttt{instr}_{\mathsf{val}}{}^{(3)}) = 0$ ▷ limb 3

// Checking instruction format for limb 4
- $(\texttt{is-type-i-no-shift}) \cdot (\texttt{op-c4-7} + \texttt{op-c8-10} \cdot 2^4 + \texttt{op-c11} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ limb 4

Type I with shifts – ALU instructions with shift immediate values

For Type I - shifts instructions (ALU instructions with shift immediate values):

- op-a is a destination register selector
- $\mathsf{op\text{-}a}_{\mathsf{val}}$ is obtained from the instruction execution component
- $\mathsf{op\text{-}a}_{\mathsf{val}}$-effective will be written to the op-a register
- op-b is a source register selector
- $\mathsf{op\text{-}b}_{\mathsf{val}}$ is obtained by reading from the register memory component
- op-c is a 5-bit I-immediate value (see Section 2.4)
- $\mathsf{op\text{-}c}_{\mathsf{val}}$ is obtained by setting the 5 lower bits to op-c and the higher bits to 0

CONSTRAINTS:

// ALU instructions with shift immediate values - sll, srl, sra instructions with $\texttt{imm-c} = 1$
// op-a is ranged checked via reg memory checking
// op-b is ranged checked via reg memory checking
// op-c range check follows from other range checks below
// Making sure that op-c is consistent with the immediate parts
- $(\texttt{is-alu-imm-shift}) \cdot (\texttt{op-c0-3} + \texttt{op-c4} \cdot 2^4 - \texttt{op-c}) = 0$

// Computing $\texttt{op-c}_{\mathsf{val}}$ from op-c
- $(\texttt{is-alu-imm-shift}) \cdot (\texttt{op-c0-3} + \texttt{op-c4} \cdot 2^4 - \texttt{op-c}_{\mathsf{val}}{}^{(1)}) = 0$
- $(\texttt{is-alu-imm-shift}) \cdot (\texttt{op-c}_{\mathsf{val}}{}^{(2)}) = 0$
- $(\texttt{is-alu-imm-shift}) \cdot (\texttt{op-c}_{\mathsf{val}}{}^{(3)}) = 0$
- $(\texttt{is-alu-imm-shift}) \cdot (\texttt{op-c}_{\mathsf{val}}{}^{(4)}) = 0$

// Range checking the different op-c immediate parts
- $\texttt{is-alu-imm-shift} \cdot (\texttt{op-c0-3} \in \left[0, 2^4 - 1\right])$

- $\texttt{is-alu-imm-shift} \cdot (\text{op-c4}) \cdot (1 - \text{op-c4}) = 0$

// `Making sure that op-a is consistent with the immediate parts`
- $\texttt{is-alu-imm-shift} \cdot (\text{op-a0} + \text{op-a1-4} \cdot 2 - \text{op-a}) = 0$

// `Range checking the different op-a immediate parts`
- $\texttt{is-alu-imm-shift} \cdot (\text{op-a1}) \cdot (1 - \text{op-a1}) = 0$
- $\texttt{is-alu-imm-shift} \cdot (\text{op-a1-4} \in \left[0, 2^4 - 1\right])$

// `Making sure that op-b is consistent with the immediate parts`
- $\texttt{is-alu-imm-shift} \cdot (\text{op-b0} + \text{op-b1-4} \cdot 2 - \text{op-b}) = 0$

// `Range checking the different op-b immediate parts`
- $\texttt{is-alu-imm-shift} \cdot (\text{op-b0}) \cdot (1 - \text{op-b1}) = 0$
- $\texttt{is-alu-imm-shift} \cdot (\text{op-b1-4} \in \left[0, 2^4 - 1\right])$

// `Checking instruction format for limb 1`
- $(\texttt{is-alu-imm-shift}) \cdot (\texttt{0b0010011} + \text{op-a0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(1)}) = 0$ ▷ `limb 1`

// `Checking instruction format for limb 2`
- $(\texttt{is-sll}) \cdot (\text{imm-c}) \cdot (\text{op-a1-4} + \texttt{0b001} \cdot 2^4 + \text{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ `limb 2 for slli`
- $(\texttt{is-srl}) \cdot (\text{imm-c}) \cdot (\text{op-a1-4} + \texttt{0b101} \cdot 2^4 + \text{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ `limb 2 for srli`
- $(\texttt{is-sra}) \cdot (\text{imm-c}) \cdot (\text{op-a1-4} + \texttt{0b101} \cdot 2^4 + \text{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ `limb 2 for srai`

// `Checking instruction format for limb 3`
- $(\texttt{is-alu-imm-shift}) \cdot (\text{op-b1-4} + \text{op-c0-3} \cdot 2^4 - \texttt{instr}_{\mathsf{val}}{}^{(3)}) = 0$ ▷ `limb 3`

// `Checking instruction format for limb 4`
- $(\texttt{is-sll}) \cdot (\text{imm-c}) \cdot (\text{op-c4} + \texttt{0b0000000} \cdot 2 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ `limb 4 for slli`
- $(\texttt{is-srl}) \cdot (\text{imm-c}) \cdot (\text{op-c4} + \texttt{0b0000000} \cdot 2 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ `limb 4 for srli`
- $(\texttt{is-sra}) \cdot (\text{imm-c}) \cdot (\text{op-c4} + \texttt{0b0100000} \cdot 2 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ `limb 4 for srai`

Type R: ALU instructions without immediate values

For Type R instructions (ALU instructions without immediate values):

- op-a is a destination register selector
- op-a$_{\mathsf{val}}$ is obtained from the instruction execution component
- op-a$_{\mathsf{val}}$`-effective` will be written to the op-a register
- op-b is a source register selector
- op-b$_{\mathsf{val}}$ is obtained by reading from the register memory component
- op-c is another source register selector (see Section 2.4)
- op-c$_{\mathsf{val}}$ is obtained by reading from the register memory component

Constraints:

// `ALU instructions where op-c is a register address - ALU instructions with imm-c` $= 0$
// `op-a is ranged checked via reg memory checking`
// `op-b is ranged checked via reg memory checking`
// `op-c is ranged checked via reg memory checking`
// `Making sure that op-c is consistent with the immediate parts`
- $(\text{is-type-r}) \cdot (\text{op-c0-3} + \text{op-c4} \cdot 2^4 - \text{op-c}) = 0$

// `Range checking the different op-c immediate parts`
- $\text{is-type-r} \cdot (\text{op-c0-3} \in \left[0, 2^4 - 1\right])$
- $\text{is-type-r} \cdot (\text{op-c4}) \cdot (1 - \text{op-c4}) = 0$

// `Making sure that op-a is consistent with the immediate parts`
- $\text{is-type-r} \cdot (\text{op-a0} + \text{op-a1-4} \cdot 2 - \text{op-a}) = 0$

// `Range checking the different op-a immediate parts`
- $\text{is-type-r} \cdot (\text{op-a1}) \cdot (1 - \text{op-a1}) = 0$
- $\text{is-type-r} \cdot (\text{op-a1-4} \in \left[0, 2^4 - 1\right])$

// `Making sure that op-b is consistent with the immediate parts`
- $\text{is-type-r} \cdot (\text{op-b0} + \text{op-b1-4} \cdot 2 - \text{op-b}) = 0$

// Range checking the different op-b immediate parts

- $\texttt{is-type-r} \cdot (\texttt{op-b0}) \cdot (1 - \texttt{op-b1}) = 0$
- $\texttt{is-type-r} \cdot (\texttt{op-b1-4} \in \left[0, 2^4 - 1\right])$

// Checking instruction format for limb 1

- $(\texttt{is-type-r}) \cdot (\texttt{0b0110011} + \texttt{op-a0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(1)}) = 0$ ▷ limb 1

// Checking instruction format for limb 2

- $(\texttt{is-add}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b000} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for add
- $(\texttt{is-sub}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b000} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for sub
- $(\texttt{is-sll}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b001} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for sll
- $(\texttt{is-slt}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b010} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for slt
- $(\texttt{is-sltu}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b011} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for sltu
- $(\texttt{is-xor}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b100} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for xor
- $(\texttt{is-srl}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b101} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for srl
- $(\texttt{is-sra}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b101} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for sra
- $(\texttt{is-or}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b110} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for or
- $(\texttt{is-and}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-a1-4} + \texttt{0b111} \cdot 2^4 + \texttt{op-b0} \cdot 2^7 - \texttt{instr}_{\mathsf{val}}{}^{(2)}) = 0$ ▷ limb 2 for and

// Checking instruction format for limb 3

- $(\texttt{is-type-r}) \cdot (\texttt{op-b1-4} + \texttt{op-c0-3} \cdot 2^4 - \texttt{instr}_{\mathsf{val}}{}^{(3)}) = 0$ ▷ limb 3

// Checking instruction format for limb 4

- $(\texttt{is-add}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-c4} + \texttt{0b0000000} \cdot 2 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ limb 4 for add
- $(\texttt{is-sub}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-c4} + \texttt{0b0100000} \cdot 2 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ limb 4 for sub
- $(\texttt{is-sll}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-c4} + \texttt{0b0000000} \cdot 2 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ limb 4 for sll
- $(\texttt{is-slt}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-c4} + \texttt{0b0000000} \cdot 2 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ limb 4 for slt
- $(\texttt{is-sltu}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-c4} + \texttt{0b0000000} \cdot 2 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ limb 4 for sltu
- $(\texttt{is-xor}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-c4} + \texttt{0b0000000} \cdot 2 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ limb 4 for xor
- $(\texttt{is-srl}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-c4} + \texttt{0b0000000} \cdot 2 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ limb 4 for srl
- $(\texttt{is-sra}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-c4} + \texttt{0b0100000} \cdot 2 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ limb 4 for sra
- $(\texttt{is-or}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-c4} + \texttt{0b0000000} \cdot 2 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ limb 4 for or
- $(\texttt{is-and}) \cdot (1 - \texttt{imm-c}) \cdot (\texttt{op-c4} + \texttt{0b0000000} \cdot 2 - \texttt{instr}_{\mathsf{val}}{}^{(4)}) = 0$ ▷ limb 4 for and

Type B: Branch instructions

For Type B instructions (Branch instructions):

- op-a is a destination register selector
- $\text{op-a}_{\mathsf{val}}$ is obtained from the instruction execution component
- $\text{op-a}_{\mathsf{val}}$-effective will be written to the op-a register
- op-b is a source register selector
- $\text{op-b}_{\mathsf{val}}$ is obtained by reading from the register memory component
- op-c is a 12-bit B-immediate value (see Section 2.4)
- $\text{op-c}_{\mathsf{val}}$ is obtained by setting bit 0 to 0 and, bits 1-12 to op-c, and sign extending it

Constraints:

// Type B instructions - beq, bne, blt, bge, bltu, bgeu
// op-a is ranged checked via reg memory checking
// op-b is ranged checked via reg memory checking
// op-c range check follows from other range checks below
// Making sure that op-c is consistent with the immediate parts

- $(\texttt{is-type-b}) \cdot (\texttt{op-c1-4} + \texttt{op-c5-7} \cdot 2^4 + \texttt{op-c8-10} \cdot 2^7 + \texttt{op-c11} \cdot 2^{10} + \texttt{op-c12} \cdot 2^{11} - \texttt{op-c}) = 0$

// Computing $\text{op-c}_{\mathsf{val}}$ from op-c and performing sign extension

- $(\texttt{is-type-b}) \cdot (\texttt{op-c1-4} \cdot 2 + \texttt{op-c5-7} \cdot 2^5 - \texttt{op-c}_{\mathsf{val}}{}^{(1)}) = 0$
- $(\texttt{is-type-b}) \cdot (\texttt{op-c8-10} + \texttt{op-c11} \cdot 2^3 + \texttt{op-c12} \cdot (2^4 - 1) \cdot 2^4 - \texttt{op-c}_{\mathsf{val}}{}^{(2)}) = 0$
- $(\texttt{is-type-b}) \cdot (\texttt{op-c12} \cdot (2^8 - 1) - \texttt{op-c}_{\mathsf{val}}{}^{(3)}) = 0$
- $(\texttt{is-type-b}) \cdot (\texttt{op-c12} \cdot (2^8 - 1) - \texttt{op-c}_{\mathsf{val}}{}^{(4)}) = 0$

// Range checking the different op-c immediate parts

- $\texttt{is-type-b} \cdot (\text{op-c1-4} \in \left[0, 2^4 - 1\right])$
- $\texttt{is-type-b} \cdot (\text{op-c5-7} \in \left[0, 2^3 - 1\right])$
- $\texttt{is-type-b} \cdot (\text{op-c8-10} \in \left[0, 2^3 - 1\right])$
- $\texttt{is-type-b} \cdot (\text{op-c11}) \cdot (1 - \text{op-c11}) = 0$
- $\texttt{is-type-b} \cdot (\text{op-c12}) \cdot (1 - \text{op-c12}) = 0$

// `Making sure that op-a is consistent with the immediate parts`
- $\texttt{is-type-b} \cdot (\text{op-a0} + \text{op-a1-4} \cdot 2 - \text{op-a}) = 0$

// `Range checking the different op-a immediate parts`
- $\texttt{is-type-b} \cdot (\text{op-a1}) \cdot (1 - \text{op-a1}) = 0$
- $\texttt{is-type-b} \cdot (\text{op-a1-4} \in \left[0, 2^4 - 1\right])$

// `Making sure that op-b is consistent with the immediate parts`
- $\texttt{is-type-b} \cdot (\text{op-b0-3} + \text{op-b4} \cdot 2^4 - \text{op-b}) = 0$

// `Range checking the different op-b immediate parts`
- $\texttt{is-type-b} \cdot (\text{op-b0-3} \in \left[0, 2^4 - 1\right])$
- $\texttt{is-type-b} \cdot (\text{op-b4}) \cdot (1 - \text{op-b4}) = 0$

// `Checking instruction format for limb 1`
- $(\texttt{is-type-b}) \cdot (\texttt{0b1100011} + \text{op-c11} \cdot 2^7 - \texttt{instr}_{\textsf{val}}{}^{(1)}) = 0$ ▷ `limb 1`

// `Checking instruction format for limb 2`
- $(\texttt{is-beq}) \cdot (\text{op-c1-4} + \texttt{0b000} \cdot 2^4 + \text{op-a0} \cdot 2^7 - \texttt{instr}_{\textsf{val}}{}^{(2)}) = 0$ ▷ `limb 2 for beq`
- $(\texttt{is-bne}) \cdot (\text{op-c1-4} + \texttt{0b001} \cdot 2^4 + \text{op-a0} \cdot 2^7 - \texttt{instr}_{\textsf{val}}{}^{(2)}) = 0$ ▷ `limb 2 for bne`
- $(\texttt{is-blt}) \cdot (\text{op-c1-4} + \texttt{0b100} \cdot 2^4 + \text{op-a0} \cdot 2^7 - \texttt{instr}_{\textsf{val}}{}^{(2)}) = 0$ ▷ `limb 2 for blt`
- $(\texttt{is-bge}) \cdot (\text{op-c1-4} + \texttt{0b101} \cdot 2^4 + \text{op-a0} \cdot 2^7 - \texttt{instr}_{\textsf{val}}{}^{(2)}) = 0$ ▷ `limb 2 for bge`
- $(\texttt{is-bltu}) \cdot (\text{op-c1-4} + \texttt{0b110} \cdot 2^4 + \text{op-a0} \cdot 2^7 - \texttt{instr}_{\textsf{val}}{}^{(2)}) = 0$ ▷ `limb 2 for bltu`
- $(\texttt{is-bgeu}) \cdot (\text{op-c1-4} + \texttt{0b111} \cdot 2^4 + \text{op-a0} \cdot 2^7 - \texttt{instr}_{\textsf{val}}{}^{(2)}) = 0$ ▷ `limb 2 for bgeu`

// `Checking instruction format for limb 3`
- $(\texttt{is-type-b}) \cdot (\text{op-a1-4} + \text{op-b0-3} \cdot 2^4 - \texttt{instr}_{\textsf{val}}{}^{(3)}) = 0$ ▷ `limb 3`

// `Checking instruction format for limb 4`
- $(\texttt{is-type-b}) \cdot (\text{op-b4} + \text{op-c5-7} \cdot 2 + \text{op-c8-10} \cdot 2^4 + \text{op-c12} \cdot 2^7 - \texttt{instr}_{\textsf{val}}{}^{(4)}) = 0$ ▷ `limb 4`

Type SYS: System call instructions

For Type SYS instructions (ECALL, EBREAK):

- op-a is computed within the instruction execution component depending on $R[\texttt{x17}]$
- $\text{op-a}_{\textsf{val}}$ is either a private input provided by the prover directly or not set
- op-a range check guaranteed by the register memory component when set
- $\text{op-a}_{\textsf{val}}$`-effective` will be written to the op-a register when set
- op-b is set to `x17`
- $\text{op-b}_{\textsf{val}}$ is obtained by reading from the register memory component
- op-c is set to 0
- $\text{op-c}_{\textsf{val}}$ is set to 0

Constraints:

// `System instructions - ecall, ebreak`
// `op-a value is set by the execution component depending on` $R[\texttt{x17}]$
// `op-b is set to x17`
// `op-c is set to 0`
// `Enforcing op-b` $=$ `x17 for system instructions`
- $\texttt{is-type-sys} \cdot (17 - \text{op-b}) = 0$

// `Enforcing op-c` $= 0$ `for unimp`
- $\texttt{is-type-sys} \cdot (\text{op-c}) = 0$

// `Enforcing op-c`$_{\textsf{val}} = 0$ `for unimp`
- $(\texttt{is-type-sys}) \cdot (\text{op-c}_{\textsf{val}}{}^{(1)}) = 0$

- $(\texttt{is-type-sys}) \cdot (\texttt{op-c}_{\texttt{val}}{}^{(2)}) = 0$
- $(\texttt{is-type-sys}) \cdot (\texttt{op-c}_{\texttt{val}}{}^{(3)}) = 0$
- $(\texttt{is-type-sys}) \cdot (\texttt{op-c}_{\texttt{val}}{}^{(4)}) = 0$

// `Checking instruction format for system instructions`

- $(\texttt{is-type-sys}) \cdot (\texttt{0b01110011} - \texttt{instr}_{\texttt{val}}{}^{(1)}) = 0$ ▷ `limb 1`
- $(\texttt{is-type-sys}) \cdot (\texttt{0b00000000} - \texttt{instr}_{\texttt{val}}{}^{(2)}) = 0$ ▷ `limb 2`
- $(\texttt{is-ecall}) \cdot (\texttt{0b0000} + \texttt{0b0000} \cdot 2^4 - \texttt{instr}_{\texttt{val}}{}^{(3)}) = 0$ ▷ `limb 3 for ecall`
- $(\texttt{is-ebreak} \cdot (\texttt{0b0000} + \texttt{0b1000} \cdot 2^4 - \texttt{instr}_{\texttt{val}}{}^{(3)}) = 0$ ▷ `limb 3 for ebreak`
- $(\texttt{is-type-sys}) \cdot (\texttt{0b00000000} - \texttt{instr}_{\texttt{val}}{}^{(4)}) = 0$ ▷ `limb 4`

UNIMP instruction – `unimp`
For the UNIMP instruction:

- $\mathsf{op\text{-}a} = 0$ $\mathsf{op\text{-}a_{val}}{}^{(j)} = 0$ for $j = 1, 2, 3, 4$
- $\mathsf{op\text{-}b} = 0$ $\mathsf{op\text{-}b_{val}}{}^{(j)} = 0$ for $j = 1, 2, 3, 4$
- $\mathsf{op\text{-}b} = 0$ $\mathsf{op\text{-}c_{val}}{}^{(j)} = 0$ for $j = 1, 2, 3, 4$

Constraints:

// `UNIMP instruction - unimp`
// `op-a, op-c, op-c are all set to 0`
// `Enforcing` $\texttt{op-c} = 0$ `for unimp`

- $\texttt{is-unimp} \cdot (\texttt{op-c}) = 0$

// `Enforcing` $\texttt{op-c}_{\texttt{val}} = 0$ `for unimp`

- $(\texttt{is-unimp}) \cdot (\texttt{op-c}_{\texttt{val}}{}^{(1)}) = 0$
- $(\texttt{is-unimp}) \cdot (\texttt{op-c}_{\texttt{val}}{}^{(2)}) = 0$
- $(\texttt{is-unimp}) \cdot (\texttt{op-c}_{\texttt{val}}{}^{(3)}) = 0$
- $(\texttt{is-unimp}) \cdot (\texttt{op-c}_{\texttt{val}}{}^{(4)}) = 0$

// `Enforcing` $\texttt{op-b} = 0$ `for unimp`

- $\texttt{is-unimp} \cdot (\texttt{op-b}) = 0$

// `Enforcing` $\texttt{op-b}_{\texttt{val}} = 0$ `for unimp`

- $(\texttt{is-unimp}) \cdot (\texttt{op-b}_{\texttt{val}}{}^{(1)}) = 0$
- $(\texttt{is-unimp}) \cdot (\texttt{op-b}_{\texttt{val}}{}^{(2)}) = 0$
- $(\texttt{is-unimp}) \cdot (\texttt{op-b}_{\texttt{val}}{}^{(3)}) = 0$
- $(\texttt{is-unimp}) \cdot (\texttt{op-b}_{\texttt{val}}{}^{(4)}) = 0$

// `Enforcing` $\texttt{op-a} = 0$ `for unimp`

- $\texttt{is-unimp} \cdot (\texttt{op-b}) = 0$

// `Enforcing` $\texttt{op-a}_{\texttt{val}} = 0$ `for unimp`

- $(\texttt{is-unimp}) \cdot (\texttt{op-a}_{\texttt{val}}{}^{(1)}) = 0$
- $(\texttt{is-unimp}) \cdot (\texttt{op-a}_{\texttt{val}}{}^{(2)}) = 0$
- $(\texttt{is-unimp}) \cdot (\texttt{op-a}_{\texttt{val}}{}^{(3)}) = 0$
- $(\texttt{is-unimp}) \cdot (\texttt{op-a}_{\texttt{val}}{}^{(4)}) = 0$

// `Checking instruction format for unimp`

- $(\texttt{is-unimp}) \cdot (\texttt{0b01110011} - \texttt{instr}_{\texttt{val}}{}^{(1)}) = 0$ ▷ `limb 1`
- $(\texttt{is-unimp}) \cdot (\texttt{0b00010000} - \texttt{instr}_{\texttt{val}}{}^{(2)}) = 0$ ▷ `limb 2`
- $(\texttt{is-unimp}) \cdot (\texttt{0b00110000} - \texttt{instr}_{\texttt{val}}{}^{(3)}) = 0$ ▷ `limb 3`
- $(\texttt{is-unimp}) \cdot (\texttt{0b00000000} - \texttt{instr}_{\texttt{val}}{}^{(4)}) = 0$ ▷ `limb 4`

4.3.4 Interactions with other components

As mentioned before, the CPU component needs to interact with a few other components:

- Interaction with program memory: To read the next instruction using the program counter;
- Interaction with register memory: to read values associated with operands;

- Interaction with instruction execution: to enforce correction execution of instructions.

Remark 4.14 Since all constraints in the remaining of this section are for the same row, we do not explicitly write down the row index i when describing these constraints.

Program memory interaction

- $\mathsf{instr}_{\mathsf{val}} \leftarrow \mathrm{Read}_{\mathrm{Prog}}(\texttt{pc}, \texttt{clk})$

Register memory interaction

// Type S + B instructions do not have a destination register
// Type S + B instructions instead read from the op-a register
// Hence, we obtain op-a$_{\mathsf{val}}$ by reading from the op-a register

- $(\mathsf{is\text{-}type\text{-}s} + \mathsf{is\text{-}type\text{-}b})\mathrm{Read}_{\mathrm{Reg}}(\mathsf{op\text{-}a}, \texttt{clk}, 3) - \mathsf{op\text{-}a}_{\mathsf{val}}) = 0$

// Type R + I + U + J instructions have a destination register
// Hence, we need to write op-a$_{\mathsf{val}}$ to the op-a register

- $(\texttt{is-type-r} + \texttt{is-type-i} + \texttt{is-type-u} + \texttt{is-type-j}) \cdot \mathrm{Write}_{\mathrm{Reg}}(\mathsf{op\text{-}a}, \mathsf{op\text{-}a}_{\mathsf{val}}, \texttt{clk}, 3)) = 0$

// Type SYS instructions will interact with the register memory directly
// From the execution component so we ignore SYS instructions above
// op-a$_{\mathsf{val}}$ is an input provided by the environment for Type SYS instructions

// We only read from register op-b to obtain op-b$_{\mathsf{val}}$ when op-b-flag $= 1$

- $(\texttt{op-b-flag}) \cdot \mathrm{Read}_{\mathrm{Reg}}(\mathsf{op\text{-}b}, \texttt{clk}, 1) - \mathsf{op\text{-}b}_{\mathsf{val}}) = 0$

// We only need to read from register op-c to obtain op-c$_{\mathsf{val}}$ for Type R instructions

- $(\texttt{is-type-r}) \cdot \mathrm{Read}_{\mathrm{Reg}}(\mathsf{op\text{-}c}, \texttt{clk}, 2) - \mathsf{op\text{-}c}_{\mathsf{val}}) = 0$

Remark 4.15 Remark on handling the case where the destination register rd = `x0`: Since the contents of register `x0` must remain equal to 0, we need to make sure the value of the destination register does not get updated when rd = Reg0. For this, we added an additional column op-a$_{\mathsf{val}}$-`effective` whose value is op-a$_{\mathsf{val}}$ (if rd $\neq$ `x0`) or 0 (if rd = `x0`). The register memory checking uses op-a$_{\mathsf{val}}$-`effective` instead of op-a$_{\mathsf{val}}$. That is,

- op-a$_{\mathsf{val}}$-`effective` = op-a$_{\mathsf{val}}$ when rd $\neq$ `x0`
- op-a$_{\mathsf{val}}$-`effective` = **0** when rd = `x0`

Execution component interaction

- $\mathsf{pc\text{-}next} \leftarrow \mathsf{exec}(\mathsf{pc}, \mathsf{opcode}, \mathsf{op\text{-}a}_{\mathsf{val}}, \mathsf{op\text{-}b}_{\mathsf{val}}, \mathsf{op\text{-}c}_{\mathsf{val}})$

5 Register memory component

As described above, the read-write register memory component is responsible for managing access to the registers. Since this is a read-write memory, each register will have a timestamp associated with it indicating the last time that that register has been written.

In order to check that the register memory has been updated correctly, we will make use of logups to check the consistency between the read and write sets, where each element of the set has the form (`reg-addr`, `reg-val`, `reg-ts`) and indicates that the value `reg-val` was written to address `reg-addr` at time `reg-ts`.

5.1 Read and write operations

Read operation Suppose we want to read the contents of a register `reg-addr` at time `reg-ts`. Let `reg-val-prev` be the value stored in it and let `reg-ts-prev` be the time in which this value was written. In order to simplify the register memory checking, we make sure that every written value is read only once. Of course a CPU might read the same value again, so each read operation is followed by a write operation where the same value is written to memory. More precisely, we will update the read and write sets as follows:

- $\texttt{reg-read-set} = \texttt{reg-read-set} \cup \{(\texttt{reg-addr}, \texttt{reg-val-prev}, \texttt{reg-ts-prev})\}$
- $\texttt{reg-write-set} = \texttt{reg-write-set} \cup \{(\texttt{reg-addr}, \texttt{reg-val-prev}, \texttt{reg-ts-cur})\}$

Note that the union is disjoint because of the unique timestamps.

Write operation Suppose we want to update the contents of a register `reg-addr` at time `reg-ts-cur` with the `reg-val-cur`. Let `reg-val-prev` be the value stored in this register and let `reg-ts-prev` be the time in which this value was written. In order to simplify the register memory checking, we make sure that every written value is read once. For this reason, although a write operation overwrites the content of the register, each write operation is preceded by a read operation. More precisely, we will update the read and write sets as follows:

- $\texttt{reg-read-set} = \texttt{reg-read-set} \cup \{(\texttt{reg-addr}, \texttt{reg-val-prev}, \texttt{reg-ts-prev})\}$
- $\texttt{reg-write-set} = \texttt{reg-write-set} \cup \{(\texttt{reg-addr}, \texttt{reg-val-cur}, \texttt{reg-ts-cur})\}$

Let α and β be two random values chosen by the verifier after the prover commits to the execution trace of the program. To capture these operations using logups, we will first convert each triple $(\texttt{reg-addr}, \texttt{reg-val}, \texttt{reg-ts})$ in the read and write sets to a field element in the secure extension field, which can be seen as a fingerprint of the triple.

This can be done by picking a random linear combination of the elements in the triple using consecutive powers $\beta^0, \beta^1, \beta^2$ as the coefficients. In other words, the fingerprint for $(\texttt{reg-addr}, \texttt{reg-val}, \texttt{reg-ts})$ will be $\texttt{reg-addr} \cdot \beta^0 + \texttt{reg-val} \cdot \beta^1 + \texttt{reg-ts} \cdot \beta^2$. If we denote by $\mathsf{fp}(\texttt{reg-addr}, \texttt{reg-val}, \texttt{reg-ts})$ this fingerprint function, the logup contribution for the entry $\texttt{reg-addr}, \texttt{reg-val}, \texttt{reg-ts})$ will be $1/(\mathsf{fp}(\texttt{reg-addr}, \texttt{reg-val}, \texttt{reg-ts}) + \alpha)$.

5.2 Register memory trace elements

In the case of the register memory, up to three register addresses can be accessed during an execution cycle. Since each access to the register memory requires us to maintain a set $(\texttt{reg-addr}, \texttt{reg-val-cur}, \texttt{reg-val-prev}, \texttt{reg-ts-prev}, \texttt{reg-ts-cur})$ to properly handle memory updates related to a particular register, we define 3 such sets of values.

As a result, we will have the following set of trace elements:

- `clk`: the current execution time
- `reg1-addr`, `reg2-addr`, `reg3-addr`: the address of the register
- `reg1-val-cur`, `reg2-val-cur`, `reg3-val-cur`: 32-bit values used to update register contents
- `reg1-ts-cur`, `reg2-ts-cur`, `reg3-ts-cur`: current timestamps for the registers
- `reg1-val-prev`, `reg2-val-prev`, `reg3-val-prev`: previous 32-bit values stored at the registers
- `reg1-ts-prev`, `reg2-ts-prev`, `reg3-ts-prev`: previous timestamps for the registers
- `reg-read-digest` (in the interaction trace, not in the execution trace): a digest of the read set, used for logups.
- `reg-write-digest` (in the interaction trace, not in the execution trace): a digest of the write set, used for logups.
- `reg1-accessed`, `reg2-accessed`, `reg3-accessed`: flags indicating whether the set of trace elements $(\texttt{reg}j\texttt{-addr}, \texttt{reg}j\texttt{-val-cur}, \texttt{reg}j\texttt{-ts-cur}, \texttt{reg}j\texttt{-val-prev}, \texttt{reg}j\texttt{-ts-prev})$ for $j = 1, 2, 3$ are being used

5.3 Register memory initialization

Initially, at time 0, we assume that the contents of all registers are initialized to 0. In particular, this means that the initial write set will contain an entry $(\texttt{reg-addr}, 0, 0)$ for each register selector $\texttt{reg-addr}$, where the second and third components correspond to their initial value and timestamp. The corresponding digest for this initial write set will be

$$\texttt{reg-write-init-digest} = \sum_{\texttt{reg-addr} \in \{0,\ldots,31\}} \frac{1}{\mathsf{fp}(\texttt{reg-addr}, 0, 0) + \alpha}.$$

5.4 Register memory interface

In order to clarify the interaction between the register memory and other components, we define here the interface used for reading from and writing to the register memory. Since the register memory allows for up to 3 register read and write operations in a clock cycle and since each of them uses a different time tick, the interface also includes a selector reg-sel $\in \{1, 2, 3\}$ to specify which time tick should be used.

- $\mathrm{Read}_{\mathrm{Reg}}(\texttt{reg-addr}, \texttt{clk}, \text{reg-sel}) \mapsto \mathsf{val}$: the register memory returns the value val stored at register location $\texttt{reg-addr}$, updating timestamps according to the timestamp $\texttt{clk}$ and the register selector reg-sel.
- $\mathrm{Write}_{\mathrm{Reg}}(\texttt{reg-addr}, \mathsf{val}, \texttt{clk}, \text{reg-sel}, \mathsf{val})$: the register memory updates the value stored at register location $\texttt{reg-addr}$ with the value val, updating timestamps according to the timestamp $\texttt{clk}$ and register selector reg-sel.

Remark 5.1 • The interface above considers each input and output as a single element, However, in certain cases, some of these entries will be specified by a set of 8-bit limbs.

- The interface also assumes that registers $\texttt{reg1-addr}$, $\texttt{reg2-addr}$, $\texttt{reg3-addr}$ should be accessed in this order, with $\texttt{reg3-addr}$ being the last one to be updated.
- When a register is not accessed during a clock cycle, the corresponding flag $\texttt{reg}j\texttt{-accessed}$ for $j = 1, 2, 3$ should be set to 0 so that the corresponding entry is not taken into account during the logup computation.
- The value $\texttt{reg}j\texttt{-accessed}$ for $j = 1, 2, 3$ will be set to the flag used by the other components when they call the register memory interface. For instance, since the CPU components only reads the contents of $\texttt{reg2-addr}$ for Type R instructions, $\texttt{reg2-accessed}$ will be set to $\texttt{is-type-r}$.
- When a row is just a padding (after the program execution), the flags $\texttt{reg}j\texttt{-accessed}$ for $j = 1, 2, 3$ should all be set to 0.

5.5 Register memory constraints assuming large fields

5.5.1 Range checks

// $\texttt{reg1-addr} \in \left[0, 2^{32} - 1\right]$ - guaranteed via memory checking

// $\texttt{reg2-addr} \in \left[0, 2^{32} - 1\right]$ - guaranteed via memory checking

// $\texttt{reg3-addr} \in \left[0, 2^{32} - 1\right]$ - guaranteed via memory checking

- $\texttt{reg1-val-cur} \in \left[0, 2^{32} - 1\right]$ ▷ Common to instruction execution
- $\texttt{reg2-val-cur} \in \left[0, 2^{32} - 1\right]$ ▷ Common to instruction execution
- $\texttt{reg3-val-cur} \in \left[0, 2^{32} - 1\right]$ ▷ Common to instruction execution
- $\texttt{reg1-val-prev} \in \left[0, 2^{32} - 1\right]$
- $\texttt{reg2-val-prev} \in \left[0, 2^{32} - 1\right]$
- $\texttt{reg3-val-prev} \in \left[0, 2^{32} - 1\right]$

// $\texttt{reg}i\texttt{-ts-prev} \in \{0, \ldots, \texttt{reg}i\texttt{-ts-cur} - 1\}$ for $i = 1, 2, 3$

- $\texttt{reg1-ts-prev} \in \{0, \ldots, \texttt{reg1-ts-cur} - 1\}$

- $\texttt{reg2-ts-prev} \in \{0, \ldots, \texttt{reg2-ts-cur} - 1\}$
- $\texttt{reg3-ts-prev} \in \{0, \ldots, \texttt{reg3-ts-cur} - 1\}$

Remark 5.2
- Since only one register value gets updated in a time clock, we could assume that $\texttt{reg1-val-cur} = \texttt{reg1-val-prev}$ and $\texttt{reg2-val-cur} = \texttt{reg2-val-prev}$ and not use the values $\texttt{reg1-val-cur}$ and $\texttt{reg2-val-cur}$.
- For $T < 2^{32} - 1$, we can check if an element $a \in \{0, \ldots, T - 1\}$ by performing two ranges: $a \in \left[0, 2^{32} - 1\right]$ and $T - 1 - a \in \left[0, 2^{32} - 1\right]$, as per [GPR21].
- We also assume that $T < 2^{30}$ so that the result of the multiplication by 3 remains a 32-bit value. We can add more limbs for timestamps and the clock if necessary.
- Since the current timestamps for $\texttt{reg}j\texttt{-ts-cur}$ for $j = 1, 2, 3$ are specific functions of the $\texttt{clk}$ value, we added specific arithmetic constraints below for these trace elements.
- Since we are working with a trace that is ordered according to the clock cycle, the range check for $\texttt{clk}$ is not needed. Instead, we require a boundary constraint ($\texttt{clk}[1] = 1$) and a transition constraint ($\texttt{clk}[i + 1] = \texttt{clk}[i] + 1$).

5.5.2 Arithmetic constraints

// Computing $\texttt{reg}j\texttt{-val-cur}$ as a function of $\texttt{clk}$ for $j = 1, 2, 3$
- $\texttt{reg1-val-cur} = 3 \cdot \texttt{clk} - 2$
- $\texttt{reg2-val-cur} = 3 \cdot \texttt{clk} - 1$
- $\texttt{reg3-val-cur} = 3 \cdot \texttt{clk}$

Remark 5.3 The constraints above assume that registers $\texttt{reg1}$, $\texttt{reg2}$, $\texttt{reg3}$ are accessed and updated in this order, with $\texttt{reg3}$ being the last one to be updated.

5.5.3 Logup computations

As we said above, one may access up to 3 registers in a clock cycle depending on the instruction that is being executed. The three flags $\texttt{rf1}$, $\texttt{rf2}$, $\texttt{rf3}$ are $\texttt{reg1-accessed}$, $\texttt{reg2-accessed}$, $\texttt{reg3-accessed}$ respectively.They indicate respectively whether the set of trace elements ($\texttt{reg}j\texttt{-addr}, \texttt{reg}j\texttt{-val-cur}, \texttt{reg}j\texttt{-ts-cur}, \texttt{reg}j\texttt{-val-prev}, \texttt{reg}j\texttt{-ts-prev}$) for $j = 1, 2, 3$ are being used.

In order to compute the difference between the read set and write set digests between rows $i - 1$ and i, we can use these flags as follows:

- $\texttt{reg-read-digest}[i] - \texttt{reg-read-digest}[i - 1] =$
  $\texttt{rf1}[i] / (\mathsf{fp}(\texttt{reg1-addr}[i], \texttt{reg1-val-prev}[i], \texttt{reg1-ts-prev}[i]) + \alpha) +$
  $\texttt{rf2}[i] / (\mathsf{fp}(\texttt{reg2-addr}[i], \texttt{reg2-val-prev}[i], \texttt{reg2-ts-prev}[i]) + \alpha) +$
  $\texttt{rf3}[i] / (\mathsf{fp}(\texttt{reg3-addr}[i], \texttt{reg3-val-prev}[i], \texttt{reg3-ts-prev}[i]) + \alpha)$
- $\texttt{reg-write-digest}[i] - \texttt{reg-write-digest}[i - 1] =$
  $\texttt{rf1}[i] / (\mathsf{fp}(\texttt{reg1-addr}[i], \texttt{reg1-val-cur}[i], \texttt{reg1-ts-cur}[i]) + \alpha) +$
  $\texttt{rf2}[i] / (\mathsf{fp}(\texttt{reg2-addr}[i], \texttt{reg2-val-cur}[i], \texttt{reg2-ts-cur}[i]) + \alpha) +$
  $\texttt{rf3}[i] / (\mathsf{fp}(\texttt{reg3-addr}[i], \texttt{reg3-val-cur}[i], \texttt{reg3-ts-cur}[i]) + \alpha)$,

where $\texttt{rf}j[i] = \texttt{reg}j\texttt{-accessed}[i]$ for $j = 1, 2, 3$ are three flags that indicate whether the set of trace elements ($\texttt{reg}j\texttt{-addr}[i], \texttt{reg}j\texttt{-val-cur}[i], \texttt{reg}j\texttt{-ts-cur}[i], \texttt{reg}j\texttt{-val-prev}[i], \texttt{reg}j\texttt{-ts-prev}[i]$) are being accessed during the i-th clock cycle.

Initial write set digest: Let $\texttt{reg-write-init-digest}$ be the logup sum for the initial state of the register memory (see Section 5.3). That is,

$$\texttt{reg-write-init-digest} = \sum_{\texttt{reg-addr} \in \{0, \ldots, 31\}} \frac{1}{\mathsf{fp}(\texttt{reg-addr}, 0, 0) + \alpha}.$$

Final read set digest: Let reg-val-final(reg-addr) denote the last value written to register reg-addr $\in \{0, \ldots, 31\}$ and let reg-ts-final(reg-addr) denote the corresponding timestamp. Let reg-read-final-digest denote the logup sum for the final state of memory. That is, reg-read-final-digest is equal to

$$\sum_{\texttt{reg-addr} \in \{0,\ldots,31\}} \frac{1}{\texttt{fp}(\texttt{reg-addr}, \texttt{reg-val-final}(\texttt{reg-addr}), \texttt{reg-ts-final}(\texttt{reg-addr})) + \alpha}$$

Boundary constraints: Let reg-write-init-digest and reg-read-final-digest be as defined above. The boundary constraints can then be specified as follows:

- $\texttt{reg-read-digest}[1] =$
  $\texttt{rf1}[1]/(\texttt{fp}(\texttt{reg1-addr}[1], \texttt{reg1-val-prev}[1], \texttt{reg1-ts-prev}[1]) + \alpha) +$
  $\texttt{rf2}[1]/(\texttt{fp}(\texttt{reg2-addr}[1], \texttt{reg2-val-prev}[1], \texttt{reg2-ts-prev}[1]) + \alpha) +$
  $\texttt{rf3}[1]/(\texttt{fp}(\texttt{reg3-addr}[1], \texttt{reg3-val-prev}[1], \texttt{reg3-ts-prev}[1]) + \alpha)$
- $\texttt{reg-write-digest}[1] = \texttt{reg-write-init-digest} +$
  $\texttt{rf1}[1]/(\texttt{fp}(\texttt{reg1-addr}[1], \texttt{reg1-val-cur}[1], \texttt{reg1-ts-cur}[1]) + \alpha) +$
  $\texttt{rf2}[1]/(\texttt{fp}(\texttt{reg2-addr}[1], \texttt{reg2-val-cur}[1], \texttt{reg2-ts-cur}[1]) + \alpha) +$
  $\texttt{rf3}[1]/(\texttt{fp}(\texttt{reg3-addr}[1], \texttt{reg3-val-cur}[1], \texttt{reg3-ts-cur}[1]) + \alpha)$
- $\texttt{reg-read-digest}[n] = \texttt{reg-read-final-digest} + \texttt{reg-write-digest}[n]$

Transition constraints ($1 < i \leq n$): The transition constraints can be specified as follows:

- $\texttt{reg-read-digest}[i] - \texttt{reg-read-digest}[i-1] =$
  $\texttt{rf1}[i]/(\texttt{fp}(\texttt{reg1-addr}[i], \texttt{reg1-val-prev}[i], \texttt{reg1-ts-prev}[i]) + \alpha) +$
  $\texttt{rf2}[i]/(\texttt{fp}(\texttt{reg2-addr}[i], \texttt{reg2-val-prev}[i], \texttt{reg2-ts-prev}[i]) + \alpha) +$
  $\texttt{rf3}[i]/(\texttt{fp}(\texttt{reg3-addr}[i], \texttt{reg3-val-prev}[i], \texttt{reg3-ts-prev}[i]) + \alpha)$
- $\texttt{reg-write-digest}[i] - \texttt{reg-write-digest}[i-1] =$
  $\texttt{rf1}[i]/(\texttt{fp}(\texttt{reg1-addr}[i], \texttt{reg1-val-cur}[i], \texttt{reg1-ts-cur}[i]) + \alpha) +$
  $\texttt{rf2}[i]/(\texttt{fp}(\texttt{reg2-addr}[i], \texttt{reg2-val-cur}[i], \texttt{reg2-ts-cur}[i]) + \alpha) +$
  $\texttt{rf3}[i]/(\texttt{fp}(\texttt{reg3-addr}[i], \texttt{reg3-val-cur}[i], \texttt{reg3-ts-cur}[i]) + \alpha)$

Remark 5.4
- Instead of initializing all register values, it suffices to initialize only those registers which will be accessed during the execution of a program.
- Instead of maintaining separate running sums, the implementation can choose to keep track of the difference $\texttt{reg-read-write-digest}[i] = \texttt{reg-write-digest}[i] - \texttt{reg-read-digest}[i]$ between the two logup sums so that its final value is equal to 0. To make this work, the initialization would need to be changed slightly to also take into account reg-read-final-digest.

5.6 Register memory constraints assuming small fields

5.6.1 Range checks

// $\texttt{reg1-addr}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - guaranteed via memory checking

// $\texttt{reg2-addr}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - guaranteed via memory checking

// $\texttt{reg3-addr}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - guaranteed via memory checking

- $\texttt{reg1-val-cur}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ ▷ Common to instruction execution
- $\texttt{reg2-val-cur}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ ▷ Common to instruction execution
- $\texttt{reg3-val-cur}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ ▷ Common to instruction execution
- $\texttt{reg1-val-prev}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{reg2-val-prev}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{reg3-val-prev}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$

// $\texttt{reg}i\texttt{-ts-prev} \in \{0, \ldots, \texttt{reg}i\texttt{-ts-cur} - 1\}$ for $i = 1, 2, 3$

// $\implies \texttt{reg}i\texttt{-ts-prev} \in \left[0, 2^{32}-1\right]$ $\texttt{for}$ $i = 1, 2, 3$

// $\implies \texttt{reg}i\texttt{-ts-prev-aux} \in \left[0, 2^{32}-1\right]$ $\texttt{for}$ $i = 1, 2, 3$

// $\texttt{where reg}i\texttt{-ts-prev-aux} = \texttt{reg}i\texttt{-ts-cur} - 1 - \texttt{reg}i\texttt{-ts-prev}$ $\texttt{for}$ $i = 1, 2, 3$

- $\texttt{reg1-ts-prev}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{reg2-ts-prev}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{reg3-ts-prev}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{reg1-ts-prev-aux}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{reg2-ts-prev-aux}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{reg3-ts-prev-aux}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$

5.6.2 Arithmetic constraints

As indicated in the large field case, we need to enforce constraints on the limbs associated with `reg1-ts-cur`, `reg2-ts-cur`, and `reg3-ts-cur` so that they satisfy the following conditions:

- $\texttt{reg1-ts-cur} = 3 \cdot \texttt{clk} - 2$
- $\texttt{reg2-ts-cur} = 3 \cdot \texttt{clk} - 1$
- $\texttt{reg3-ts-cur} = 3 \cdot \texttt{clk}$

In order to do so, we need to introduce helper values to handle carries and borrows because, when performing the above computations over the limbs, both positive or negative can occur.

- Let $\texttt{clk} \coloneqq (\texttt{clk}^{(1)}, \texttt{clk}^{(2)}, \texttt{clk}^{(3)}, \texttt{clk}^{(4)})$
- Let $\texttt{reg1-ts-cur} \coloneqq (\texttt{reg1-ts-cur}^{(1)}, \texttt{reg1-ts-cur}^{(2)}, \texttt{reg1-ts-cur}^{(3)}, \texttt{reg1-ts-cur}^{(4)})$
- Let $\texttt{reg2-ts-cur} \coloneqq (\texttt{reg2-ts-cur}^{(1)}, \texttt{reg2-ts-cur}^{(2)}, \texttt{reg2-ts-cur}^{(3)}, \texttt{reg2-ts-cur}^{(4)})$
- Let $\texttt{reg3-ts-cur} \coloneqq (\texttt{reg3-ts-cur}^{(1)}, \texttt{reg3-ts-cur}^{(2)}, \texttt{reg3-ts-cur}^{(3)}, \texttt{reg3-ts-cur}^{(4)})$
- Let $\texttt{h}i\texttt{-carry} \coloneqq (\texttt{h}i\texttt{-carry}^{(1)}, \texttt{h}i\texttt{-carry}^{(2)}, \texttt{h}i\texttt{-carry}^{(3)}, \texttt{h}i\texttt{-carry}^{(4)})$ for $i = 1, 2, 3$ be helper values used to handle carries
- Let $\texttt{h}i\texttt{-borrow} \coloneqq (\texttt{h}i\texttt{-borrow}^{(1)}, \texttt{h}i\texttt{-borrow}^{(2)}, \texttt{h}i\texttt{-borrow}^{(3)}, \texttt{h}i\texttt{-borrow}^{(4)})$ for $i = 1, 2, 3$ be helper values used to handle borrows

Arithmetic constraints for `reg1-ts-cur`, `reg2-ts-cur`, `reg3-ts-cur`

// $\texttt{Carry and borrow handling for reg1-ts-cur} = 3 \cdot \texttt{clk} - 2$

- $\texttt{reg1-ts-cur}^{(1)} + 2 \quad + \texttt{h1-carry}^{(1)} \cdot 2^8 = 3 \cdot \texttt{clk}^{(1)} + \texttt{h1-borrow}^{(1)} \cdot 2^8$
- $\texttt{reg1-ts-cur}^{(2)} + \texttt{h1-borrow}^{(1)} + \texttt{h1-carry}^{(2)} \cdot 2^8 = 3 \cdot \texttt{clk}^{(2)} + \texttt{h1-borrow}^{(2)} \cdot 2^8 + \texttt{h1-carry}^{(1)}$
- $\texttt{reg1-ts-cur}^{(3)} + \texttt{h1-borrow}^{(2)} + \texttt{h1-carry}^{(3)} \cdot 2^8 = 3 \cdot \texttt{clk}^{(3)} + \texttt{h1-borrow}^{(3)} \cdot 2^8 + \texttt{h1-carry}^{(2)}$
- $\texttt{reg1-ts-cur}^{(4)} + \texttt{h1-borrow}^{(3)} + \texttt{h1-carry}^{(4)} \cdot 2^8 = 3 \cdot \texttt{clk}^{(4)} + \texttt{h1-borrow}^{(4)} \cdot 2^8 + \texttt{h1-carry}^{(3)}$

// $\texttt{Carry and borrow handling for reg2-ts-cur} = 3 \cdot \texttt{clk} - 1$

- $\texttt{reg2-ts-cur}^{(1)} + 1 \quad + \texttt{h2-carry}^{(1)} \cdot 2^8 = 3 \cdot \texttt{clk}^{(1)} + \texttt{h2-borrow}^{(1)} \cdot 2^8$
- $\texttt{reg2-ts-cur}^{(2)} + \texttt{h2-borrow}^{(1)} + \texttt{h2-carry}^{(2)} \cdot 2^8 = 3 \cdot \texttt{clk}^{(2)} + \texttt{h2-borrow}^{(2)} \cdot 2^8 + \texttt{h2-carry}^{(1)}$
- $\texttt{reg2-ts-cur}^{(3)} + \texttt{h2-borrow}^{(2)} + \texttt{h2-carry}^{(3)} \cdot 2^8 = 3 \cdot \texttt{clk}^{(3)} + \texttt{h2-borrow}^{(3)} \cdot 2^8 + \texttt{h2-carry}^{(2)}$
- $\texttt{reg2-ts-cur}^{(4)} + \texttt{h2-borrow}^{(3)} + \texttt{h2-carry}^{(4)} \cdot 2^8 = 3 \cdot \texttt{clk}^{(4)} + \texttt{h2-borrow}^{(4)} \cdot 2^8 + \texttt{h2-carry}^{(3)}$

// $\texttt{Carry and borrow handling for reg3-ts-cur} = 3 \cdot \texttt{clk}$

- $\texttt{reg3-ts-cur}^{(1)} \quad + \texttt{h3-carry}^{(1)} \cdot 2^8 = 3 \cdot \texttt{clk}^{(1)} + \texttt{h3-borrow}^{(1)} \cdot 2^8$
- $\texttt{reg3-ts-cur}^{(2)} + \texttt{h3-borrow}^{(1)} + \texttt{h3-carry}^{(2)} \cdot 2^8 = 3 \cdot \texttt{clk}^{(2)} + \texttt{h3-borrow}^{(2)} \cdot 2^8 + \texttt{h3-carry}^{(1)}$
- $\texttt{reg3-ts-cur}^{(3)} + \texttt{h3-borrow}^{(2)} + \texttt{h3-carry}^{(3)} \cdot 2^8 = 3 \cdot \texttt{clk}^{(3)} + \texttt{h3-borrow}^{(3)} \cdot 2^8 + \texttt{h3-carry}^{(2)}$
- $\texttt{reg3-ts-cur}^{(4)} + \texttt{h3-borrow}^{(3)} + \texttt{h3-carry}^{(4)} \cdot 2^8 = 3 \cdot \texttt{clk}^{(4)} + \texttt{h3-borrow}^{(4)} \cdot 2^8 + \texttt{h3-carry}^{(3)}$

// $\texttt{Enforcing ranges for the borrows} \in \{0, 1\}$ $\texttt{- last borrow must be 0}$

- $(\texttt{h1-borrow}^{(j)}) \cdot (1 - \texttt{h1-borrow}^{(j)}) = 0$ for $j = 1, 2, 3$
- $(\texttt{h2-borrow}^{(j)}) \cdot (1 - \texttt{h2-borrow}^{(j)}) = 0$ for $j = 1, 2, 3$
- $(\texttt{h3-borrow}^{(j)}) \cdot (1 - \texttt{h3-borrow}^{(j)}) = 0$ for $j = 1, 2, 3$
- $(\texttt{h1-borrow}^{(4)}) = 0$
- $(\texttt{h2-borrow}^{(4)}) = 0$
- $(\texttt{h3-borrow}^{(4)}) = 0$

// `Enforcing ranges for the carries` $\in \{0,1,2\}$
- $(\texttt{h1-carry}^{(j)}) \cdot (1 - \texttt{h1-carry}^{(j)}) \cdot (2 - \texttt{h1-carry}^{(j)}) = 0$ for $j = 1,2,3,4$
- $(\texttt{h2-carry}^{(j)}) \cdot (1 - \texttt{h2-carry}^{(j)}) \cdot (2 - \texttt{h2-carry}^{(j)}) = 0$ for $j = 1,2,3,4$
- $(\texttt{h3-carry}^{(j)}) \cdot (1 - \texttt{h3-carry}^{(j)}) \cdot (2 - \texttt{h3-carry}^{(j)}) = 0$ for $j = 1,2,3,4$

// `Making sure that borrows and carries cannot both be non-zero`
- $(\texttt{h1-carry}^{(j)}) \cdot (\texttt{h1-borrow}^{(j)}) = 0$ for $j = 1,2,3$
- $(\texttt{h2-carry}^{(j)}) \cdot (\texttt{h2-borrow}^{(j)}) = 0$ for $j = 1,2,3$
- $(\texttt{h3-carry}^{(j)}) \cdot (\texttt{h3-borrow}^{(j)}) = 0$ for $j = 1,2,3$

In addition, we also need to enforce the constraints on the limbs associated with `reg`i`-ts-prev` and `reg`i`-ts-prev-aux` for $i = 1,2,3$ so that they satisfy the following conditions:

- $\texttt{reg1-ts-prev-aux} = \texttt{reg1-val-cur} - 1 - \texttt{reg1-ts-prev}$
- $\texttt{reg2-ts-prev-aux} = \texttt{reg2-val-cur} - 1 - \texttt{reg2-ts-prev}$
- $\texttt{reg3-ts-prev-aux} = \texttt{reg3-val-cur} - 1 - \texttt{reg3-ts-prev}$

Arithmetic constraints for `reg1-ts-prev-aux`, `reg2-ts-prev-aux`, `reg3-ts-prev-aux`

// `Enforcing reg1-ts-prev-aux` $=$ `reg1-ts-cur` $- 1 -$ `reg1-ts-prev`
// $\texttt{h1-aux-borrow}^{(j)}$ `for` $j = 1,2,3,4$ `used for borrow handling`
- $\texttt{reg1-ts-prev-aux}^{(1)} + \texttt{reg1-ts-prev}^{(1)} + 1 = \texttt{reg1-ts-cur}^{(1)} + \texttt{h1-aux-borrow}^{(1)} \cdot 2^8$
- $\texttt{reg1-ts-prev-aux}^{(2)} + \texttt{reg1-ts-prev}^{(2)} + \texttt{h1-aux-borrow}^{(1)} = \texttt{reg1-ts-cur}^{(2)} + \texttt{h1-aux-borrow}^{(2)} \cdot 2^8$
- $\texttt{reg1-ts-prev-aux}^{(2)} + \texttt{reg1-ts-prev}^{(3)} + \texttt{h1-aux-borrow}^{(2)} = \texttt{reg1-ts-cur}^{(3)} + \texttt{h1-aux-borrow}^{(3)} \cdot 2^8$
- $\texttt{reg1-ts-prev-aux}^{(2)} + \texttt{reg1-ts-prev}^{(4)} + \texttt{h1-aux-borrow}^{(3)} = \texttt{reg1-ts-cur}^{(4)} + \texttt{h1-aux-borrow}^{(4)} \cdot 2^8$

// `Enforcing reg2-ts-prev-aux` $=$ `reg2-ts-cur` $- 1 -$ `reg2-ts-prev`
// $\texttt{h2-aux-borrow}^{(j)}$ `for` $j = 1,2,3,4$ `used for borrow handling`
- $\texttt{reg2-ts-prev-aux}^{(1)} + \texttt{reg2-ts-prev}^{(1)} + 1 = \texttt{reg2-ts-cur}^{(1)} + \texttt{h2-aux-borrow}^{(1)} \cdot 2^8$
- $\texttt{reg2-ts-prev-aux}^{(2)} + \texttt{reg2-ts-prev}^{(2)} + \texttt{h2-aux-borrow}^{(1)} = \texttt{reg2-ts-cur}^{(2)} + \texttt{h2-aux-borrow}^{(2)} \cdot 2^8$
- $\texttt{reg2-ts-prev-aux}^{(2)} + \texttt{reg2-ts-prev}^{(3)} + \texttt{h2-aux-borrow}^{(2)} = \texttt{reg2-ts-cur}^{(3)} + \texttt{h2-aux-borrow}^{(3)} \cdot 2^8$
- $\texttt{reg2-ts-prev-aux}^{(2)} + \texttt{reg2-ts-prev}^{(4)} + \texttt{h2-aux-borrow}^{(3)} = \texttt{reg2-ts-cur}^{(4)} + \texttt{h2-aux-borrow}^{(4)} \cdot 2^8$

// `Enforcing reg3-ts-prev-aux` $=$ `reg3-ts-cur` $- 1 -$ `reg3-ts-prev`
// $\texttt{h3-aux-borrow}^{(j)}$ `for` $j = 1,2,3,4$ `used for borrow handling`
- $\texttt{reg3-ts-prev-aux}^{(1)} + \texttt{reg3-ts-prev}^{(1)} + 1 = \texttt{reg3-ts-cur}^{(1)} + \texttt{h3-aux-borrow}^{(1)} \cdot 2^8$
- $\texttt{reg3-ts-prev-aux}^{(2)} + \texttt{reg3-ts-prev}^{(2)} + \texttt{h3-aux-borrow}^{(1)} = \texttt{reg3-ts-cur}^{(2)} + \texttt{h3-aux-borrow}^{(2)} \cdot 2^8$
- $\texttt{reg3-ts-prev-aux}^{(2)} + \texttt{reg3-ts-prev}^{(3)} + \texttt{h3-aux-borrow}^{(2)} = \texttt{reg3-ts-cur}^{(3)} + \texttt{h3-aux-borrow}^{(3)} \cdot 2^8$
- $\texttt{reg3-ts-prev-aux}^{(2)} + \texttt{reg3-ts-prev}^{(4)} + \texttt{h3-aux-borrow}^{(3)} = \texttt{reg3-ts-cur}^{(4)} + \texttt{h3-aux-borrow}^{(4)} \cdot 2^8$

// `Enforcing ranges for the borrows` $\in \{0,1\}$ `- last borrow must be 0`
- $(\texttt{h1-aux-borrow}^{(j)}) \cdot (1 - \texttt{h1-aux-borrow}^{(j)}) = 0$ for $j = 1,2,3$
- $(\texttt{h2-aux-borrow}^{(j)}) \cdot (1 - \texttt{h2-aux-borrow}^{(j)}) = 0$ for $j = 1,2,3$
- $(\texttt{h3-aux-borrow}^{(j)}) \cdot (1 - \texttt{h3-aux-borrow}^{(j)}) = 0$ for $j = 1,2,3$
- $(\texttt{h1-aux-borrow}^{(4)}) = 0$
- $(\texttt{h2-aux-borrow}^{(4)}) = 0$
- $(\texttt{h3-aux-borrow}^{(4)}) = 0$

5.6.3 Logup computations

When adapting the logup argument to small fields, the components `reg-val` and `reg-ts` are each split into four 8-bit limb values $\texttt{reg-val}^{(j)}$ and $\texttt{reg-ts}^{(j)}$ for $j = 1,2,3,4$.

Let `fp` be a fingerprint function which takes as input a tuple $(\texttt{reg-addr}, \texttt{reg-val}, \texttt{reg-ts}) \coloneqq (\texttt{reg-addr}, \texttt{reg-val}^{(1)}, \ldots, \texttt{reg-val}^{(4)}, \texttt{reg-ts}^{(1)}, \ldots, \texttt{reg-ts}^{(4)})$ and returns a field element in the secure extension field `qm31` using a value β chosen by the verifier.

Like in the large field case, one way of achieving this goal is to compute a random linear combination of the elements in the tuple using consecutive powers $\beta^0, \beta^1, \ldots, \beta^8$ as the coefficients. That is, $\mathsf{fp}(\texttt{reg-addr}, \texttt{reg-val}^{(1)}, \ldots, \texttt{reg-val}^{(4)}, \texttt{reg-ts}^{(1)}, \ldots, \texttt{reg-ts}^{(4)})$ simply returns $\texttt{reg-addr} \cdot \beta^0 + \texttt{reg-val}^{(1)} \cdot \beta^1 + \ldots + \texttt{reg-ts}^{(4)} \cdot \beta^1$.

As before, let $\texttt{rf1}[i] = \texttt{reg1-accessed}[i]$, $\texttt{rf2}[i] = \texttt{reg2-accessed}[i]$, $\texttt{rf3}[i] = \texttt{reg3-accessed}[i]$ be flags that indicate respectively whether the set of trace elements $(\texttt{reg}j\texttt{-addr}[i], \texttt{reg}j\texttt{-val-cur}[i], \texttt{reg}j\texttt{-ts-cur}[i], \texttt{reg}j\texttt{-val-prev}[i], \texttt{reg}j\texttt{-ts-prev}[i])$ for $j = 1, 2, 3$ are being accessed during the i-th clock cycle. Using fp and these flags, we can compute the difference between the read set and write set digests between rows $i-1$ and i as follows:

- $\texttt{reg-read-digest}[i] - \texttt{reg-read-digest}[i-1] =$
  $\texttt{rf1}[i]/(\mathsf{fp}(\texttt{reg1-addr}[i], \texttt{reg1-val-prev}[i], \texttt{reg1-ts-prev}[i]) + \alpha) +$
  $\texttt{rf2}[i]/(\mathsf{fp}(\texttt{reg2-addr}[i], \texttt{reg2-val-prev}[i], \texttt{reg2-ts-prev}[i]) + \alpha) +$
  $\texttt{rf3}[i]/(\mathsf{fp}(\texttt{reg3-addr}[i], \texttt{reg3-val-prev}[i], \texttt{reg3-ts-prev}[i]) + \alpha)$
- $\texttt{reg-write-digest}[i] - \texttt{reg-write-digest}[i-1] =$
  $\texttt{rf1}[i]/(\mathsf{fp}(\texttt{reg1-addr}[i], \texttt{reg1-val-cur}[i], \texttt{reg1-ts-cur}[i]) + \alpha) +$
  $\texttt{rf2}[i]/(\mathsf{fp}(\texttt{reg2-addr}[i], \texttt{reg2-val-cur}[i], \texttt{reg2-ts-cur}[i]) + \alpha) +$
  $\texttt{rf3}[i]/(\mathsf{fp}(\texttt{reg3-addr}[i], \texttt{reg3-val-cur}[i], \texttt{reg3-ts-cur}[i]) + \alpha)$.

Initial write set digest: Let $\texttt{reg-write-init-digest}$ be the logup sum for the initial state of the register memory (see Section 5.3). That is,

$$\texttt{reg-write-init-digest} = \sum_{\texttt{reg-addr} \in \{0,\ldots,31\}} \frac{1}{\mathsf{fp}(\texttt{reg-addr}, \mathbf{0}, \mathbf{0}) + \alpha},$$

where $\mathbf{0} = (0, 0, 0, 0)$.

Final read set digest: Let $\texttt{reg-val-final}(\texttt{reg-addr})$ denote the last value written to register $\texttt{reg-addr} \in \{0, \ldots, 31\}$ and let $\texttt{reg-ts-final}(\texttt{reg-addr})$ denote the corresponding timestamp. Let $\texttt{reg-read-final-digest}$ denote the logup sum for the final state of memory. That is, $\texttt{reg-read-final-digest}$ is equal to

$$\sum_{\texttt{reg-addr} \in \{0,\ldots,31\}} \frac{1}{\mathsf{fp}(\texttt{reg-addr}, \texttt{reg-val-final}(\texttt{reg-addr}), \texttt{reg-ts-final}(\texttt{reg-addr})) + \alpha}.$$

Boundary constraints: Let $\texttt{reg-write-init-digest}$ and $\texttt{reg-read-final-digest}$ be be as defined above. The boundary constraints can then be specified as follows:

- $\texttt{reg-read-digest}[1] =$
  $\texttt{rf1}[1]/(\mathsf{fp}(\texttt{reg1-addr}[1], \texttt{reg1-val-prev}[1], \texttt{reg1-ts-prev}[1]) + \alpha) +$
  $\texttt{rf2}[1]/(\mathsf{fp}(\texttt{reg2-addr}[1], \texttt{reg2-val-prev}[1], \texttt{reg2-ts-prev}[1]) + \alpha) +$
  $\texttt{rf3}[1]/(\mathsf{fp}(\texttt{reg3-addr}[1], \texttt{reg3-val-prev}[1], \texttt{reg3-ts-prev}[1]) + \alpha)$.
- $\texttt{reg-write-digest}[1] = \texttt{reg-write-init-digest} +$
  $\texttt{rf1}[1]/(\mathsf{fp}(\texttt{reg1-addr}[1], \texttt{reg1-val-cur}[1], \texttt{reg1-ts-cur}[1]) + \alpha) +$
  $\texttt{rf2}[1]/(\mathsf{fp}(\texttt{reg2-addr}[1], \texttt{reg2-val-cur}[1], \texttt{reg2-ts-cur}[1]) + \alpha) +$
  $\texttt{rf3}[1]/(\mathsf{fp}(\texttt{reg3-addr}[1], \texttt{reg3-val-cur}[1], \texttt{reg3-ts-cur}[1]) + \alpha)$
- $\texttt{reg-read-digest}[n] = \texttt{reg-read-final-digest} + \texttt{reg-write-digest}[n]$

Transition constraints $(1 < i \leq n)$: The transition constraints can be specified as follows:

- $\texttt{reg-read-digest}[i] - \texttt{reg-read-digest}[i-1] =$
  $\texttt{rf1}[i]/(\mathsf{fp}(\texttt{reg1-addr}[i], \texttt{reg1-val-prev}[i], \texttt{reg1-ts-prev}[i]) + \alpha) +$
  $\texttt{rf2}[i]/(\mathsf{fp}(\texttt{reg2-addr}[i], \texttt{reg2-val-prev}[i], \texttt{reg2-ts-prev}[i]) + \alpha) +$
  $\texttt{rf3}[i]/(\mathsf{fp}(\texttt{reg3-addr}[i], \texttt{reg3-val-prev}[i], \texttt{reg3-ts-prev}[i]) + \alpha)$.

- $\texttt{reg-write-digest}[i] - \texttt{reg-write-digest}[i-1] =$
  $\texttt{rf1}[i]/(\texttt{fp}(\texttt{reg1-addr}[i], \texttt{reg1-val-cur}[i], \texttt{reg1-ts-cur}[i]) + \alpha) +$
  $\texttt{rf2}[i]/(\texttt{fp}(\texttt{reg2-addr}[i], \texttt{reg2-val-cur}[i], \texttt{reg2-ts-cur}[i]) + \alpha) +$
  $\texttt{rf3}[i]/(\texttt{fp}(\texttt{reg3-addr}[i], \texttt{reg3-val-cur}[i], \texttt{reg3-ts-cur}[i]) + \alpha)$,

where $\texttt{rf}j[i] = \texttt{reg}j\texttt{-accessed}[i]$ for $j = 1, 2, 3$ are the three flags defined above that indicate whether the set of trace elements $(\texttt{reg}j\texttt{-addr}, \texttt{reg}j\texttt{-val-cur}, \texttt{reg}j\texttt{-ts-cur}, \texttt{reg}j\texttt{-val-prev}, \texttt{reg}j\texttt{-ts-prev})$ are being accessed during the i-th clock cycle.

5.7 Constraints and logup computation for initial write and final read sets

In order to help compute the logup contributions related to the initial write and final read sets mentioned in Section 5.6.3, we define additional trace columns and constraints for the register memory component.

5.7.1 Trace elements for initial write and final read sets

In addition to the trace elements defined in Section 5.2, this section defines additional elements to help the verifier in the calculation of the initial write set and the final read set.

- `reg-init-final-addr`: register addresses used during the execution of the program
- `reg-val-final`: the final value stored at register `reg-init-final-addr` at the end of the execution.
- `reg-ts-final`: the timestamp associated with the last access to register `reg-init-final-addr`.
- `is-reg-addr`: a flag that indicates whether row $i \in \{1, \ldots, 32\}$.
- `row-index`: a column that contains the index of the row.

Remarks

- `reg-init-final-addr` includes all the register addresses accessed during the execution.
- All registers addresses $\{0, \ldots, 31\}$ are initialized with $\mathbf{0} = (0, 0, 0, 0)$.
- Both `is-reg-addr` and `row-index` contain values that are known to the verifier, so we do not need to constrain them.

5.7.2 Constraints for register address values

Since the number of `reg-init-final-addr` addresses is small, the simplest way to account for all the addresses needed for the logup computations is to set the value of `reg-init-final-addr` to be equal to `row-index` $- 1$ and to ignore its value after row 32.

// Enforcing $\texttt{reg-init-final-addr} = \texttt{row-index} - 1$

- $(\texttt{reg-init-final-addr} - \texttt{row-index} - 1) = 0$

// $\texttt{reg-val-final}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by range checks during memory checking

// $\texttt{reg-ts-final}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Does not need to be reinforced

// $\texttt{reg-init-final-addr} \in \{0, \ldots, 31\}$ - Implied by constraints

5.7.3 Logup computation for initial write and final read sets

In order to compute the logup sums for the initial and final states of the register memory, as described in Section 5.6.3, the prover creates two additional columns:

- `reg-write-init-digest`: a digest column used to compute the logup sum for the initial state of the register memory;

- `reg-read-final-digest`: a digest column used to compute the logup sum for the final state of the register memory.

Boundary constraints: Let $\mathbf{0} = (0,0,0,0)$. The boundary constraints are as follows:

- $\texttt{reg-write-init-digest}[1] =$
  $\texttt{is-reg-addr}[1]/(\mathsf{fp}(\texttt{reg-init-final-addr}[1], \mathbf{0}, \mathbf{0}) + \alpha)$
- $\texttt{reg-read-final-digest}[1] =$
  $\texttt{is-reg-addr}[1]/(\mathsf{fp}(\texttt{reg-init-final-addr}[1], \texttt{reg-val-final}[1], \texttt{reg-ts-final}[1]) + \alpha)$

Transition constraints $(1 < i \leq n)$: Let $\mathbf{0} = (0,0,0,0)$. The transition constraints are as follows:

- $\texttt{reg-write-init-digest}[i] - \texttt{reg-write-init-digest}[i-1] =$
  $\texttt{is-reg-addr}[i]/(\mathsf{fp}(\texttt{reg-init-final-addr}[i], \mathbf{0}, \mathbf{0}) + \alpha)$
- $\texttt{reg-write-digest}[i] - \texttt{reg-write-digest}[i-1] =$
  $\texttt{is-reg-addr}[i]/(\mathsf{fp}(\texttt{reg-init-final-addr}[i], \texttt{reg-val-final}[i], \texttt{reg-ts-final}[i]) + \alpha)$

6 Program memory component

The program memory component is responsible for managing access to the read-only program memory. Since this is a read-only memory, this component only requires a simplified version of the offline memory checking technique described in Section 3.2 to guarantee the consistency of read operations. More precisely, the algorithm for performing a read described in Section 3.2 can be simplified so that the timestamp update step is replaced by a counter update step which simply increments the counter at every read.

As a result, instead of keeping a timestamp for each memory cell, each address in the memory will have a counter associated with it indicating the number of times that this memory location has been read.

As for other memory types, we make use of logups to check the consistency between the read and write sets, where each element of the set has the form (`prog-addr`, `prog-val`, `prog-ctr`). This indicates that the value `prog-val` at address `prog-addr` has been read `prog-ctr` times.

Like the data memory component in Section 7, the program memory component is byte-addressable. However, since instructions are 32-bits long, 4 consecutive memory locations are read in each clock cycle. As a result, the program memory is also assumed to be word aligned, with the base address for the first memory location being read always being a multiple of 4. For more details, see Section 6.5.

Convention: The base address of an instruction refers to the address of the first byte of the 32-bit program instruction. Base addresses are word-aligned (i.e., a multiple of 4). The contents of the pc register always point to the base address of an instruction.

6.1 Program memory trace elements

As a result, we will have the following base set of trace elements:

- `pc`: the word-aligned base address associated with a program instruction
- $\texttt{instr}_{\mathsf{val}}{}^{(1)}$: bits 0-7 of the instruction word $\texttt{instr}_{\mathsf{val}}$ stored at address `pc`
- $\texttt{instr}_{\mathsf{val}}{}^{(2)}$: bits 8-15 of the instruction word $\texttt{instr}_{\mathsf{val}}$ stored at address $\texttt{pc} + 1$
- $\texttt{instr}_{\mathsf{val}}{}^{(3)}$: bits 16-23 of the instruction word $\texttt{instr}_{\mathsf{val}}$ stored at address $\texttt{pc} + 2$
- $\texttt{instr}_{\mathsf{val}}{}^{(4)}$: bits 24-31 of the instruction word $\texttt{instr}_{\mathsf{val}}$ stored at address $\texttt{pc} + 3$
- `prog-ctr-prev`: the previous counter value associated with base address `pc`
- `prog-ctr-cur`: the current counter value associated with base address `pc`
- `prog-read-digest`: a digest of the read set, used for logups
- `prog-write-digest`: a digest of the write set, used for logup

Remark 6.1

- $\texttt{instr}_{\mathsf{val}} = (\texttt{instr}_{\mathsf{val}}{}^{(1)}, \texttt{instr}_{\mathsf{val}}{}^{(2)}, \texttt{instr}_{\mathsf{val}}{}^{(3)}, \texttt{instr}_{\mathsf{val}}{}^{(4)})$ refers to the actual 32-bit word of the instruction stored at address `pc`
- `prog-ctr-prev` and `prog-ctr-cur` are shared across the limbs $\texttt{instr}_{\mathsf{val}}{}^{(j)}$ of the instruction $\texttt{instr}_{\mathsf{val}}$, where $j = 1, 2, 3, 4$, because these 4 byte values are always read simultaneously.
- Since the 4 byte values are always read simultaneously, we will use the 32-bit word $\texttt{instr}_{\mathsf{val}}$ when computing the read set digest during the logup computation. That is, each element of the read set is of the form $(\texttt{prog-addr}, \texttt{prog-val}, \texttt{prog-ctr})$ where $\texttt{instr}_{\mathsf{val}} = (\texttt{instr}_{\mathsf{val}}{}^{(1)}, \texttt{instr}_{\mathsf{val}}{}^{(2)}, \texttt{instr}_{\mathsf{val}}{}^{(3)}, \texttt{instr}_{\mathsf{val}}{}^{(4)})$. The prover commits to pairs $(\texttt{pc}, \texttt{instr}_{\mathsf{val}})$ for the whole program in a separate trace, and sends a deterministic commitment of those to the verifier.
- Somewhere in the execution trace, the prover commits to `prog-ctr-final` for every program instruction $\texttt{instr}_{\mathsf{val}}$ accessed during the execution, where `prog-ctr-final` denotes the total number of times the instruction $\texttt{instr}_{\mathsf{val}}$ has been executed. The value `prog-ctr-final` should be committed to the same row as $(\texttt{pc}, \texttt{instr}_{\mathsf{val}})$ on the program memory trace.
- The initial counters can be omitted because they are known to be zero.

Counter update:

- Initially, the counters associated with the base address of each instruction will be 0 and the contents of these addresses will be initialized with the program that is being executed.
- The counter associated with a given base address will be incremented by one every time that base address is accessed.
- The value of the current program counter `prog-ctr-cur` should always be equal to `prog-ctr-prev` $+ 1$ for any given base address.

6.2 Program memory read operations

Let $\texttt{instr}_{\mathsf{val}} = (\texttt{instr}_{\mathsf{val}}{}^{(1)}, \texttt{instr}_{\mathsf{val}}{}^{(2)}, \texttt{instr}_{\mathsf{val}}{}^{(3)}, \texttt{instr}_{\mathsf{val}}{}^{(4)})$ be the instruction word stored at an base address `pc` and let `prog-ctr-prev` be the latest value of the counter associated with this base address (i.e., the number of times the instruction $\texttt{instr}_{\mathsf{val}}$ at base address `pc` has been accessed). Whenever a read operation takes place at an address, the current counter `prog-ctr-cur` associated with the base address `pc` needs to be set to $\texttt{prog-ctr-prev} + 1$.

Read operation Let `pc` be the word-aligned base address used to access the instruction. In order to read the contents of the 4 bytes $(\texttt{instr}_{\mathsf{val}}{}^{(1)}, \ldots, \texttt{instr}_{\mathsf{val}}{}^{(4)})$ of the instruction $\texttt{instr}_{\mathsf{val}}$ stored at addresses $[\texttt{pc}, \texttt{pc} + 3]$, we update the read and write sets as follows:

- $\texttt{prog-read-set} = \texttt{prog-read-set} \cup \{(\texttt{pc}, \texttt{instr}_{\mathsf{val}}, \texttt{prog-ctr-prev})\}$
- $\texttt{prog-write-set} = \texttt{prog-write-set} \cup \{(\texttt{pc}, \texttt{instr}_{\mathsf{val}}, \texttt{prog-ctr-cur})\}$

6.3 Program memory initialization

Initially, the counters associated with each word-aligned base address will be 0 and the memory locations will be initialized with the contents of the program being executed.

Let `fp` be a fingerprint function which takes as input the tuple $(\texttt{prog-addr}, \texttt{prog-val}, \texttt{prog-ctr})$ and returns a field element in the secure extension field `qm31` using a value β chosen by the verifier.

Let $\mathsf{PROG{-}SET}$ denote the set of word-aligned base addresses used by the input program and let $\texttt{prog-val} \coloneqq (\texttt{prog-val}^{(1)}, \ldots, \texttt{prog-val}^{(4)})$ denote the instruction byte values stored respectively at memory locations $[\texttt{prog-addr}, \texttt{prog-addr} + 3]$. The corresponding digest for the initial write set will be

$$\texttt{prog-write-init-digest} = \sum_{\texttt{prog-addr} \in \mathsf{PROG{-}SET}} \frac{1}{(\mathsf{fp}(\texttt{prog-addr}, \texttt{prog-val}, 0) + \alpha)}.$$

In other words, even though the memory is byte-addressable, the computation of the fingerprint function is performed as if the memory was word-addressable since only addresses at 4-byte boundaries are used. This is possible because the base address `prog-addr` is expected to be a multiple of 4 and each access to the program memory will read the 4 consecutive bytes stored at `prog-addr`.

The case where the `prog-addr` and `prog-ctr-prev` are represented by 8-bit limbs is treated internally by the fingerprint function `fp`, as described further below.

6.4 Program memory interface

In order to clarify the interaction between the read-only program memory and other components, we define here the interface used for reading from the program memory.

Currently, we assume that there is exactly 1 program instruction word $\texttt{instr}_{\textsf{val}} = (\texttt{instr}_{\textsf{val}}{}^{(1)}, \ldots, \texttt{instr}_{\textsf{val}}{}^{(4)})$ being accessed in a clock cycle.

- $\text{Read}_{\text{Prog}}(\texttt{prog-addr}) \mapsto \texttt{instr}_{\textsf{val}}$: the program memory returns the 32-bit value $\texttt{instr}_{\textsf{val}} = (\texttt{instr}_{\textsf{val}}{}^{(1)}, \ldots, \texttt{instr}_{\textsf{val}}{}^{(4)})$ stored at base address `prog-addr`, updating the counter associated with `prog-addr` as needed.

Remark 6.2 As in the register memory case, the interface above considers each input as a single element. However, each of these values will be specified by a set of limbs.

- `prog-addr` will be specified by 4 limbs $(\texttt{prog-addr}^{(1)}, \texttt{prog-addr}^{(2)}, \texttt{prog-addr}^{(3)}, \texttt{prog-addr}^{(4)})$
- $\texttt{instr}_{\textsf{val}}$ will be specified by 4 limbs $(\texttt{instr}_{\textsf{val}}{}^{(1)}, \texttt{instr}_{\textsf{val}}{}^{(2)}, \texttt{instr}_{\textsf{val}}{}^{(3)}, \texttt{instr}_{\textsf{val}}{}^{(4)})$

Moreover, the base address `prog-addr` used to access the program memory is assumed to be word-aligned (i.e., a multiple of 4), as discussed in the next subsection. This requirement is enforced by the CPU component.

6.5 Program memory alignment and addressing

The program memory of the Nexus virtual machine follows the RV32I instruction set and is byte addressable. However, since instructions are 4 bytes in length, the base address used to access the program memory has to be a multiple of 4 or else a memory misalignment exception will be raised.

Remark 6.3 Remark about misalignments: According to the RISC-V specification (see https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf, Page 11), "*in the base ISA, there are four core instruction formats (R/I/S/U), as shown in Figure 2.2. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction address misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. No instruction fetch misaligned exception is generated for a conditional branch that is not taken.*"

Given the four-byte alignment requirement, this component presupposes that the base address provided through the interface satisfies this condition. This condition is enforced by the CPU component.

6.6 Program memory constraints assuming large fields

6.6.1 Range checks

// $\texttt{pc} \in \left[0, 2^{32}-1\right]$ `- guaranteed via memory checking`

// `Enforcing` $\texttt{instr}_{\textsf{val}}{}^{(j)} \in \left[0, 2^{8}-1\right]$ `for` $j = 1, 2, 3, 4$

- $\texttt{instr}_{\textsf{val}}{}^{(1)} \in \left[0, 2^{8}-1\right]$
- $\texttt{instr}_{\textsf{val}}{}^{(2)} \in \left[0, 2^{8}-1\right]$
- $\texttt{instr}_{\textsf{val}}{}^{(3)} \in \left[0, 2^{8}-1\right]$
- $\texttt{instr}_{\textsf{val}}{}^{(4)} \in \left[0, 2^{8}-1\right]$

// `Enforcing range check for prog-ctr-prev`

- $\texttt{prog-ctr-prev} \in \left[0, 2^{32} - 1\right]$

// `Enforcing range check for prog-ctr-cur`

- $\texttt{prog-ctr-cur} \in \left[0, 2^{32} - 1\right]$

6.6.2 Arithmetic constraints

// `Enforcing` $\texttt{prog-ctr-cur} = \texttt{prog-ctr-prev} + 1$
// `prog-ctr-carry used for carry handling`

- $\texttt{prog-ctr-prev} + 1 - \texttt{prog-ctr-carry} \cdot 2^{32} - \texttt{prog-ctr-cur}$

// `Enforcing` $\texttt{prog-ctr-carry} \in \{0, 1\}$

- $(\texttt{prog-ctr-carry}) \cdot (1 - \texttt{prog-ctr-carry}) = 0$

6.6.3 Logup computations

Since we assume that only one instruction gets executed in a clock cycle, there will be exactly 4 program memory accesses in each cycle, in order to read the 4 bytes of the program instruction word.

Let fp be a fingerprint function which takes as input the tuple $(\texttt{prog-addr}, \texttt{prog-val}, \texttt{prog-ctr})$ and returns a field element in the secure extension field $\texttt{qm31}$ using a value β chosen by the verifier, where $\texttt{prog-val} \coloneqq (\texttt{prog-val}^{(1)}, \ldots, \texttt{prog-val}^{(4)})$.

Moreover, let PROG–SET denote the set of *word-aligned base addresses* used by the input program.

Initial write set digest: Let `prog-write-init-digest` be the logup sum for the initial state of the program memory (see Section 6.3). That is,

$$\texttt{prog-write-init-digest} = \sum_{\texttt{prog-addr} \in \mathsf{PROG\text{–}SET}} \frac{1}{(\mathsf{fp}(\texttt{prog-addr}, \texttt{prog-val}, 0) + \alpha)},$$

where $\texttt{prog-val} = (\texttt{prog-val}^{(1)}, \ldots, \texttt{prog-val}^{(4)})$ denotes the instruction byte values stored respectively at memory locations $[\texttt{prog-addr}, \texttt{prog-addr} + 3]$.

Final read set digest: Let `prog-read-final-digest` be the logup sum for the final state of the program memory. That is,

$$\texttt{prog-read-final-digest} = \sum_{\texttt{prog-addr} \in \mathsf{PROG\text{–}SET}} \frac{1}{(\mathsf{fp}(\texttt{prog-addr}, \texttt{prog-val}, \texttt{prog-ctr-final}) + \alpha)},$$

where `prog-ctr-final` denotes the final value of the counter associated with the base address `prog-addr` when the latter is last accessed and $\texttt{prog-val} = (\texttt{prog-val}^{(1)}, \ldots, \texttt{prog-val}^{(4)})$ denotes the instruction byte values stored respectively at memory locations $[\texttt{prog-addr}, \texttt{prog-addr} + 3]$.

Difference between read and write set digests: Let $\texttt{prog-addr}[i]$ be the word-aligned base address for the instruction $(\texttt{prog-val}[i]^{(1)}, \texttt{prog-val}[i]^{(2)}, \texttt{prog-val}[i]^{(3)}, \texttt{prog-val}[i]^{(4)})$ being accessed at row i. Let $\texttt{prog-ctr-prev}[i]$ and $\texttt{prog-ctr-cur}[i]$ be the corresponding previous and current counters associated with the base address $\texttt{prog-addr}[i]$. The difference between the read set and write set digests between rows $i - 1$ and i can be computed as follows:

- $\texttt{prog-read-digest}[i] - \texttt{prog-read-digest}[i-1] = \frac{1}{(\mathsf{fp}(\texttt{prog-addr}[i], \texttt{prog-val}[i], \texttt{prog-ctr-prev}[i]) + \alpha)}$
- $\texttt{prog-write-digest}[i] - \texttt{prog-write-digest}[i-1] = \frac{1}{(\mathsf{fp}(\texttt{prog-addr}[i], \texttt{prog-val}[i], \texttt{prog-ctr-cur}[i]) + \alpha)}$

Boundary constraints: Let `prog-write-init-digest` and `prog-read-final-digest` be as defined above. The boundary constraints can then be specified as follows:

- $\texttt{prog-read-digest}[1] = \frac{1}{(\mathsf{fp}(\texttt{prog-addr}[1],\texttt{prog-val}[1],\texttt{prog-ctr-prev}[1])+\alpha)}$
- $\texttt{prog-write-digest}[1] = \texttt{prog-write-init-digest} + \frac{1}{(\mathsf{fp}(\texttt{prog-addr}[1],\texttt{prog-val}[1],\texttt{prog-ctr-cur}[1])+\alpha)}$
- $\texttt{prog-read-digest}[n] = \texttt{prog-read-final-digest} + \texttt{ram-write-digest}[n]$.

Transition constraints $(1 < i \leq n)$: The transition constraints can be specified as follows:

- $\texttt{prog-read-digest}[i] - \texttt{prog-read-digest}[i-1] = \frac{1}{(\mathsf{fp}(\texttt{prog-addr}[i],\texttt{prog-val}[i],\texttt{prog-ctr-prev}[i])+\alpha)}$
- $\texttt{prog-write-digest}[i] - \texttt{prog-write-digest}[i-1] = \frac{1}{(\mathsf{fp}(\texttt{prog-addr}[i],\texttt{prog-val}[i],\texttt{prog-ctr-cur}[i])+\alpha)}$.

6.7 Program memory constraints assuming small fields

6.7.1 Range checks

// $\texttt{pc}^{(1)} \in \left[0, 2^8 - 1\right]$ - guaranteed via memory checking
// $\texttt{pc}^{(2)} \in \left[0, 2^8 - 1\right]$ - guaranteed via memory checking
// $\texttt{pc}^{(3)} \in \left[0, 2^8 - 1\right]$ - guaranteed via memory checking
// $\texttt{pc}^{(4)} \in \left[0, 2^8 - 1\right]$ - guaranteed via memory checking

// Enforcing $\texttt{instr}_{\mathsf{val}}{}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{instr}_{\mathsf{val}}{}^{(1)} \in \left[0, 2^8 - 1\right]$
- $\texttt{instr}_{\mathsf{val}}{}^{(2)} \in \left[0, 2^8 - 1\right]$
- $\texttt{instr}_{\mathsf{val}}{}^{(3)} \in \left[0, 2^8 - 1\right]$
- $\texttt{instr}_{\mathsf{val}}{}^{(4)} \in \left[0, 2^8 - 1\right]$

// Enforcing range check for prog-ctr-prev
- $\texttt{prog-ctr-prev}^{(1)} \in \left[0, 2^8 - 1\right]$
- $\texttt{prog-ctr-prev}^{(2)} \in \left[0, 2^8 - 1\right]$
- $\texttt{prog-ctr-prev}^{(3)} \in \left[0, 2^8 - 1\right]$
- $\texttt{prog-ctr-prev}^{(4)} \in \left[0, 2^8 - 1\right]$

// Enforcing range check for prog-ctr-cur
- $\texttt{prog-ctr-cur}^{(1)} \in \left[0, 2^8 - 1\right]$
- $\texttt{prog-ctr-cur}^{(2)} \in \left[0, 2^8 - 1\right]$
- $\texttt{prog-ctr-cur}^{(3)} \in \left[0, 2^8 - 1\right]$
- $\texttt{prog-ctr-cur}^{(4)} \in \left[0, 2^8 - 1\right]$

6.7.2 Arithmetic constraints

// Enforcing $\texttt{prog-ctr-cur} = \texttt{prog-ctr-prev} + 1$
// prog-ctr-carry used for carry handling
- $\texttt{prog-ctr-prev}^{(1)} + 1 - \texttt{prog-ctr-carry}^{(1)} \cdot 2^8 - \texttt{prog-ctr-cur}^{(1)}$
- $\texttt{prog-ctr-prev}^{(2)} + \texttt{prog-ctr-carry}^{(1)} - \texttt{prog-ctr-carry}^{(2)} \cdot 2^8 - \texttt{prog-ctr-cur}^{(2)}$
- $\texttt{prog-ctr-prev}^{(3)} + \texttt{prog-ctr-carry}^{(2)} - \texttt{prog-ctr-carry}^{(3)} \cdot 2^8 - \texttt{prog-ctr-cur}^{(3)}$
- $\texttt{prog-ctr-prev}^{(4)} + \texttt{prog-ctr-carry}^{(3)} - \texttt{prog-ctr-carry}^{(4)} \cdot 2^8 - \texttt{prog-ctr-cur}^{(4)}$

// Enforcing $\texttt{prog-ctr-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$
- $(\texttt{prog-ctr-carry}^{(1)}) \cdot (1 - \texttt{prog-ctr-carry}^{(1)}) = 0$
- $(\texttt{prog-ctr-carry}^{(2)}) \cdot (1 - \texttt{prog-ctr-carry}^{(2)}) = 0$
- $(\texttt{prog-ctr-carry}^{(3)}) \cdot (1 - \texttt{prog-ctr-carry}^{(3)}) = 0$
- $(\texttt{prog-ctr-carry}^{(4)}) \cdot (1 - \texttt{prog-ctr-carry}^{(4)}) = 0$

6.7.3 Logup computations

In the small field case, the components prog-addr and prog-ctr-prev are specified by four 8-bit limb values $\texttt{prog-addr}^{(j)}$ and $\texttt{prog-ctr-prev}^{(j)}$ for $j = 1, 2, 3, 4$.

Let fp be a fingerprint function which takes as input the tuple ($\texttt{prog-addr}^{(1)}, \dots, \texttt{prog-addr}^{(4)}$, $\texttt{prog-val}^{(1)}, \dots, \texttt{prog-val}^{(4)}, \texttt{prog-ctr}^{(1)}, \dots, \texttt{prog-ctr}^{(4)}$) and returns a field element in the secure extension field $\texttt{qm31}$ using a value β chosen by the verifier. To simplify the notation and avoid explicitly describing each 8-bit value, we use

$$\mathsf{fp}(\texttt{prog-addr}, \texttt{prog-val}, \texttt{prog-ctr})$$

to denote

$$\mathsf{fp}(\texttt{prog-addr}^{(1)}, \dots, \texttt{prog-addr}^{(4)}, \texttt{prog-val}^{(1)}, \dots, \texttt{prog-val}^{(4)}, \texttt{prog-ctr}^{(1)}, \dots, \texttt{prog-ctr}^{(4)}).$$

Using fp, we can compute the difference between the read set and write set digests between rows $i-1$ and i as follows:

- $\texttt{prog-read-digest}[i] - \texttt{prog-read-digest}[i-1] = \frac{1}{(\mathsf{fp}(\texttt{prog-addr}[i], \texttt{prog-val}[i], \texttt{prog-ctr-prev}[i]) + \alpha)}$
- $\texttt{prog-write-digest}[i] - \texttt{prog-write-digest}[i-1] = \frac{1}{(\mathsf{fp}(\texttt{prog-addr}[i], \texttt{prog-val}[i], \texttt{prog-ctr-cur}[i]) + \alpha)}$

Initial write set digest: Let $\texttt{prog-write-init-digest}$ be the logup sum for the initial state of the program memory. That is,

$$\texttt{prog-write-init-digest} = \sum_{\texttt{prog-addr} \in \mathsf{PROG{-}SET}} \frac{1}{(\mathsf{fp}(\texttt{prog-addr}, \texttt{prog-val}, \mathbf{0}) + \alpha)},$$

where

- $\texttt{prog-addr} = (\texttt{prog-addr}^{(1)}, \texttt{prog-addr}^{(2)}, \texttt{prog-addr}^{(3)}, \texttt{prog-addr}^{(4)})$
- $\texttt{prog-val} = (\texttt{prog-val}^{(1)}, \texttt{prog-val}^{(2)}, \texttt{prog-val}^{(3)}, \texttt{prog-val}^{(4)})$ denotes the instruction byte values stored respectively at memory locations $[\texttt{prog-addr}, \texttt{prog-addr} + 3]$
- $\mathbf{0} = (0, 0, 0, 0)$

Final read set digest: Let $\texttt{prog-read-final-digest}$ be the logup sum for the final state of the program memory. That is,

$$\texttt{prog-read-final-digest} = \sum_{\texttt{prog-addr} \in \mathsf{PROG{-}SET}} \frac{1}{(\mathsf{fp}(\texttt{prog-addr}, \texttt{prog-val}, \texttt{prog-ctr-final}) + \alpha)},$$

where

- $\texttt{prog-ctr-final}$ denotes the 4 limbs of the final value of the counter associated with the base address $\texttt{prog-addr}$; and
- $\texttt{prog-val} = (\texttt{prog-val}^{(1)}, \texttt{prog-val}^{(2)}, \texttt{prog-val}^{(3)}, \texttt{prog-val}^{(4)})$ denotes the four-byte instruction value stored at address $\texttt{prog-addr}$.

Difference between read and write set digests: Let $\texttt{prog-addr}[i]$ be the word-aligned base address for the instruction ($\texttt{prog-val}[i]^{(1)}, \texttt{prog-val}[i]^{(2)}, \texttt{prog-val}[i]^{(3)}, \texttt{prog-val}[i]^{(4)}$) being accessed at row i. Let $\texttt{prog-ctr-prev}[i]$ and $\texttt{prog-ctr-cur}[i]$ be the corresponding previous and current counters associated with the base address $\texttt{prog-addr}[i]$. The difference between the read set and write set digests between rows $i-1$ and i can be computed as follows:

- $\texttt{prog-read-digest}[i] - \texttt{prog-read-digest}[i-1] =$ $1/(\mathsf{fp}(\texttt{prog-addr}[i], \texttt{prog-val}[i], \texttt{prog-ctr-prev}[i]) + \alpha)$
- $\texttt{prog-write-digest}[i] - \texttt{prog-write-digest}[i-1] =$ $1/(\mathsf{fp}(\texttt{prog-addr}[i], \texttt{prog-val}[i], \texttt{prog-ctr-cur}[i]) + \alpha)$

Boundary constraints: Let `prog-write-init-digest` and `prog-read-final-digest` be as defined above. The boundary constraints can then be specified as follows:

- $\texttt{prog-read-digest}[1] = \frac{1}{(\mathsf{fp}(\texttt{prog-addr}[1],\texttt{prog-val}[1],\texttt{prog-ctr-prev}[1])+\alpha)}$
- $\texttt{prog-write-digest}[1] = \texttt{prog-write-init-digest} + \frac{1}{(\mathsf{fp}(\texttt{prog-addr}[1],\texttt{prog-val}[1],\texttt{prog-ctr-cur}[1])+\alpha)}$
- $\texttt{prog-read-digest}[n] = \texttt{prog-read-final-digest} + \texttt{ram-write-digest}[n]$.

Transition constraints $(1 < i \leq n)$: The transition constraints can be specified as follows:

- $\texttt{prog-read-digest}[i] - \texttt{prog-read-digest}[i-1] = \frac{1}{(\mathsf{fp}(\texttt{prog-addr}[i],\texttt{prog-val}[i],\texttt{prog-ctr-prev}[i])+\alpha)}$
- $\texttt{prog-write-digest}[i] - \texttt{prog-write-digest}[i-1] = \frac{1}{(\mathsf{fp}(\texttt{prog-addr}[i],\texttt{prog-val}[i],\texttt{prog-ctr-cur}[i])+\alpha)}$.

6.8 Constraints and logup computation for initial write and final read sets

In order to help compute the logup contributions related to the initial write set and the final read set mentioned in Section 6.7.3, we define additional trace columns and constraints for the program memory component.

6.8.1 Trace elements for initial write and final read sets

In addition to the trace elements defined in Section 6.1, this section defines additional elements to help with the computation of the initial write set and the final read set.

- `prog-init-base-addr`: the memory address given for each 4-byte instruction in the program memory ever touched.
- `prog-val-init`: the 4-byte instruction word stored at address `prog-init-base-addr`.
- `prog-ctr-final`: the counter associated with the last access to address `prog-init-base-addr`.
- `prog-init-flag`: a flag indicating whether `prog-val-init`, `prog-ctr-final` columns on the current row are being used.

Remarks

- `prog-init-base-addr` should include all the program memory base addresses accessed during execution.
- `prog-init-base-addr` must be a multiple of 4 due to memory alignment requirements.

6.8.2 Constraints for ensuring uniqueness of base address values

To ensure that there are no duplicates, we constrain the relevant values of `prog-init-base-addr` to increase strictly monotonically and remain a multiple of 4, although gaps in the address range are allowed.

Let i be the index of the current row. The precise guaranteed condition is the following:

- For all rows i, $\texttt{prog-init-base-addr}[i]$ must be a multiple of 4.
- If row i is the last row in the trace (that is, $\texttt{is-last}[i] = 1$), no additional constraint is enforced.
- If row i is not the last row (that is, $\texttt{is-last}[i] = 0$), then the following must also be enforced:
  - If the value of $\texttt{prog-init-base-addr}[i+1]$ is being used (that is, $\texttt{prog-init-flag}[i+1] = 1$), then $\texttt{prog-init-base-addr}[i+1]$ should be strictly larger than $\texttt{prog-init-base-addr}[i]$ (that is, $\texttt{prog-init-base-addr}[i+1] > \texttt{prog-init-base-addr}[i]$)
  - If the value of $\texttt{prog-init-base-addr}[i+1]$ is not being used (that is, $\texttt{prog-init-flag}[i+1] = 0$), then $\texttt{prog-init-base-addr}[i+1] = \texttt{prog-init-base-addr}[i]$

In order to enforce the above conditions:

- let `prog-addr-cur` denote the value of `prog-init-base-addr` in row i;
- let `prog-addr-next` denote the value of `prog-init-base-addr` in row $i + 1$;
- let `is-last` be a flag that indicates whether row i is the last row in the trace;
- let `prog-addr-diff` be a helper variable used to compute the difference (`prog-addr-cur` − `prog-addr-next`);
- let `prog-addr-borrow` be a helper borrow variable; and
- let `prog-init-flag-next` denote the value of `prog-init-flag` in row $i + 1$.

// `Setting prog-addr-borrow to the borrow value for prog-addr-cur` $-$ `prog-addr-next`
- $(\texttt{prog-addr-cur}^{(1)} + \texttt{prog-addr-borrow}^{(1)} \cdot 2^8 - \texttt{prog-addr-next}^{(1)} - \texttt{prog-addr-diff}^{(1)}) = 0$
- $(\texttt{prog-addr-cur}^{(2)} + \texttt{prog-addr-borrow}^{(2)} \cdot 2^8 - \texttt{prog-addr-next}^{(2)} - \texttt{prog-addr-diff}^{(2)} - \texttt{prog-addr-borrow}^{(1)}) = 0$
- $(\texttt{prog-addr-cur}^{(3)} + \texttt{prog-addr-borrow}^{(3)} \cdot 2^8 - \texttt{prog-addr-next}^{(3)} - \texttt{prog-addr-diff}^{(3)} - \texttt{prog-addr-borrow}^{(2)}) = 0$
- $(\texttt{prog-addr-cur}^{(4)} + \texttt{prog-addr-borrow}^{(4)} \cdot 2^8 - \texttt{prog-addr-next}^{(4)} - \texttt{prog-addr-diff}^{(4)} - \texttt{prog-addr-borrow}^{(3)}) = 0$

// `Enforcing increment if the value prog-init-base-addr in the next row is being used`
- $(1 - \texttt{is-last})(\texttt{prog-init-flag-next}) \cdot (1 - \texttt{prog-addr-borrow}^{(4)}) = 0$

// `Enforcing no change if the value prog-init-base-addr in the next row is not being used`
- $(1 - \texttt{is-last})(1 - \texttt{prog-init-flag-next}) \cdot (\texttt{prog-addr-next}^{(1)} - \texttt{prog-addr-cur}^{(1)}) = 0$
- $(1 - \texttt{is-last})(1 - \texttt{prog-init-flag-next}) \cdot (\texttt{prog-addr-next}^{(2)} - \texttt{prog-addr-cur}^{(2)}) = 0$
- $(1 - \texttt{is-last})(1 - \texttt{prog-init-flag-next}) \cdot (\texttt{prog-addr-next}^{(3)} - \texttt{prog-addr-cur}^{(3)}) = 0$
- $(1 - \texttt{is-last})(1 - \texttt{prog-init-flag-next}) \cdot (\texttt{prog-addr-next}^{(4)} - \texttt{prog-addr-cur}^{(4)}) = 0$

// `Enforcing prog-addr-borrow`$^{(j)} \in \{0, 1\}$ `for` $j = 1, 2, 3, 4$
- $(\texttt{prog-addr-borrow}^{(1)}) \cdot (1 - \texttt{prog-addr-borrow}^{(1)}) = 0$
- $(\texttt{prog-addr-borrow}^{(2)}) \cdot (1 - \texttt{prog-addr-borrow}^{(2)}) = 0$
- $(\texttt{prog-addr-borrow}^{(3)}) \cdot (1 - \texttt{prog-addr-borrow}^{(3)}) = 0$
- $(\texttt{prog-addr-borrow}^{(4)}) \cdot (1 - \texttt{prog-addr-borrow}^{(4)}) = 0$

// `Ensuring that prog-addr-cur is a multiple of 4`
- $\texttt{prog-addr-aux} \cdot 4 - \texttt{prog-addr-cur}^{(1)} = 0$

6.8.3 Range checks

// `Enforcing ranges for additional program memory variables`
// `prog-val-init`$^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Implied by range checks during memory checking`
// `prog-ctr-final`$^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Does not need to be reinforced`
- $\texttt{prog-init-base-addr}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{prog-addr-diff}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{prog-addr-aux} \in \left[0, 2^6 - 1\right]$
- $(\texttt{prog-init-flag}) \cdot (1 - \texttt{prog-init-flag}) = 0$

6.8.4 Logup computation for initial write and final read sets

In order to compute the logup sums for the initial and final states of the program memory, as described in Section 6.7.3, the prover creates two additional columns:

- `prog-write-init-digest`: a digest column used to compute the logup sum for the initial state of the program memory;
- `prog-read-final-digest`: a digest column used to compute the logup sum for the final state of the program memory.

Boundary constraints: Let $\mathbf{0} = (0, 0, 0, 0)$. The boundary constraints are as follows:

- $\texttt{prog-write-init-digest}[1] =$
  $\texttt{prog-init-flag}[1] / (\mathsf{fp}(\texttt{prog-init-base-addr}[1], \texttt{prog-val-init}[1], \mathbf{0}) + \alpha)$

- $\texttt{prog-read-final-digest}[1] =$
  $\texttt{prog-init-flag}[1]/(\texttt{fp}(\texttt{prog-init-base-addr}[1], \texttt{prog-val-init}[1], \texttt{prog-ctr-final}[1]) + \alpha)$

Transition constraints ($1 < i \leq n$): Let $\mathbf{0} = (0, 0, 0, 0)$. The transition constraints are as follows:

- $\texttt{prog-write-init-digest}[i] - \texttt{prog-write-init-digest}[i-1] =$
  $\texttt{prog-init-flag}[i]/(\texttt{fp}(\texttt{prog-init-base-addr}[i], \texttt{prog-val-init}[i], \mathbf{0}) + \alpha)$
- $\texttt{prog-write-digest}[i] - \texttt{prog-write-digest}[i-1] =$
  $\texttt{prog-init-flag}[i]/(\texttt{fp}(\texttt{prog-init-base-addr}[i], \texttt{prog-val-init}[i], \texttt{prog-ctr-final}[i]) + \alpha)$

7 Data memory component

The read-write data memory is responsible for managing access to the RAM. Since this is a read-write memory, each address will have a timestamp associated with it indicating the last time that an address location has been accessed.

As in the case of the register memory, we will treat each memory access as first a read of the previous value, and then a write of the current value. When the operation does not change the value, e.g., a load from memory, we write back the same value as was read, but update the timestamp. We will make use of logups to check the consistency between the read and write sets, where each element of the set has the form $(\texttt{ram-addr}, \texttt{ram-val}, \texttt{ram-ts})$. This indicates that the value `ram-val` was written to address `ram-addr` at time `ram-ts`.

7.1 Read and write operations

Let `ram-val-prev` be the value stored at an address `ram-addr` and let `ram-ts-prev` be the timestamp when the value was written to it.

Read operation In order to read the contents of a memory address `ram-addr` at the current time `ram-ts-cur`, we will update the read and write sets as follows:

- $\texttt{ram-read-set} = \texttt{ram-read-set} \cup \{(\texttt{ram-addr}, \texttt{ram-val-prev}, \texttt{ram-ts-prev})\}$
- $\texttt{ram-write-set} = \texttt{ram-write-set} \cup \{(\texttt{ram-addr}, \texttt{ram-val-prev}, \texttt{ram-ts-cur})\}$

Write operation In order to update the contents of a memory address `ram-addr` at the current time `ram-ts-cur` with the value `ram-val-cur`, we will update the read and write sets as follows:

- $\texttt{ram-read-set} = \texttt{ram-read-set} \cup \{(\texttt{ram-addr}, \texttt{ram-val-prev}, \texttt{ram-ts-prev})\}$
- $\texttt{ram-write-set} = \texttt{ram-write-set} \cup \{(\texttt{ram-addr}, \texttt{ram-val-cur}, \texttt{ram-ts-cur})\}$

Let α be a random value chosen by the verifier after the prover commits to the execution trace of the program. In order to convert each triple $(\texttt{ram-addr}, \texttt{ram-val}, \texttt{ram-ts})$ to a field element so that we can use it in the logup computation, we will use the fingerprint function $\texttt{fp}(\texttt{ram-addr}, \texttt{ram-val}, \texttt{ram-ts})$. As a result, the logup contribution for the entry $(\texttt{ram-addr}, \texttt{ram-val}, \texttt{ram-ts})$ will be

$$1/(\texttt{fp}(\texttt{ram-addr}, \texttt{ram-val}, \texttt{ram-ts}) + \alpha).$$

7.2 Data memory trace elements

In the case of the data memory, up to four consecutive addresses can be accessed during an execution cycle (for instance, when executing `lw` or `sw`). Since each access to the data memory requires us to maintain a set $(\texttt{ram-addr}, \texttt{ram-val-cur}, \texttt{ram-val-prev}, \texttt{ram-ts-prev})$ to properly handle memory updates related to a particular address, we will have 4 such sets of values.

As a result, we will have the following base set of trace elements:

- `clk`: the current execution time
- `ram-base-addr`: memory base address
- `ram1-val-cur`, . . . , `ram4-val-cur`: 8-bit values used to update memory locations
- `ram1-val-prev`, . . . , `ram4-val-prev`: previous 8-bit values stored at each location
- `ram1-ts-prev`, . . . , `ram4-ts-prev`: previous timestamps for each location
- `ram1-accessed`, . . . , `ram4-accessed`: flags indicating whether each address in [`ram-base-addr`, `ram-base-addr` + 3] is accessed.
- `ram-read-digest`: a digest of the read set, used for logups.
- `ram-write-digest`: a digest of the write set, used for logups.

Remarks

- Since the data memory addresses being accessed in a row are consecutive, only the value associated with the first of the four memory locations (the base address `ram-base-addr`) needs to be maintained. The remaining addresses can be computed from it by adding 1, 2, or 3 to `ram-base-addr`.
- Due to memory alignment restrictions, adding 1, 2, or 3 to the base address will not cause an overflow so we do not need to worry about carries.
  - For instance, when the function `lh` calls the data memory component interface to read two bytes, the base address `ram-base-addr` will be a multiple of 2 and therefore at most equal to 254. Hence, the address for the second memory location (i.e., `ram-base-addr` + 1) would not overflow.
  - Although the values of `ram-base-addr` + i for $i = 2, 3$ could overflow (meaning could be larger than 255) when used within the `lh` instruction, their logup contributions would be ignored since `ram3-accessed` and `ram4-accessed` would be 0 in this case.
  - The same reasoning used for `lh` applies to `lhu` and `sh` instructions.
  - For `lb`, `lbu`, `sb` instructions, `ram-base-addr`+i for $i = 1, 2, 3$ could overflow but their logup contributions would be ignored since `ram2-accessed`, `ram3-accessed`, and `ram4-accessed` would be 0.
  - For `lw` and `sw` instructions, `ram-base-addr` is always 4-aligned, hence `ram-base-addr` + i for $i = 1, 2, 3$ would not overflow.
- When updating the write set for the data memory, it suffices to use the current execution time `clk` as the current timestamp for the memory addresses being accessed. Hence, we do need to explicitly define `ram-ts-cur` and we can use `clk` in its place.

7.3 Data memory initialization

We assume that the contents of memory locations accessed during the execution of the program are initialized to 0 at time 0, except for public input locations which may contain other values. In particular, this means that the initial write set will contain an entry (`ram-addr`, 0, 0) for each memory address `ram-addr` accessed during the execution of the program that is not a public input address. Similarly, for each public input address `pub-in-addr` being initialized to a value `pub-in-val`, there will be an entry (`pub-in-addr`, `pub-in-val`, 0) in the initial write set.

Let $\mathsf{MEM{-}SET}$ denote the set of locations accessed during the execution of the program and let $\mathsf{PUB{-}IN{-}SET} \subseteq \mathsf{MEM{-}SET}$ denote the subset of public input locations. Moreover, let $\mathsf{init\text{-}pub\text{-}val}(\texttt{pub-in-addr})$ denote the initial value stored at the public input address `pub-in-addr`. The corresponding digest for this initial write set will be

$$\begin{aligned}\texttt{ram-write-init-digest} \quad &= \sum_{\texttt{ram-addr} \in \mathsf{MEM{-}SET} \setminus \mathsf{PUB{-}IN{-}SET}} \frac{1}{\mathsf{fp}(\texttt{ram-addr}, 0, 0) + \alpha} \\ &+ \sum_{\texttt{pub-in-addr} \in \mathsf{PUB{-}IN{-}SET}} \frac{1}{\mathsf{fp}(\texttt{pub-in-addr}, \mathsf{init\text{-}pub\text{-}val}(\texttt{pub-in-addr}), 0) + \alpha}.\end{aligned}$$

7.4 Data memory interface

In order to clarify the interaction between the data memory and other components, we define here the interface used for reading from and writing to the data memory.

Currently, we assume that there might be at most 4 memory locations being accessed in a clock cycle. For this reason, the interface also includes an input ram-shift $\in \{0, 1, 2, 3\}$ that specifies the memory location being accessed.

- $\text{Read}_{\text{RAM}}(\texttt{clk}, \text{base-addr}, \text{ram-shift}) \mapsto \mathsf{val}$: the data memory returns the value val stored at memory location base-addr + ram-shift, updating timestamps according to the timestamp $\texttt{clk}$ and shift amount ram-shift.
- $\text{Write}_{\text{RAM}}(\texttt{clk}, \mathsf{val}, \text{base-addr}, \text{ram-shift})$: the data memory updates the value stored at memory location base-addr + ram-shift with the value val, updating timestamps according to the timestamp $\texttt{clk}$ and shift amount ram-shift.

Remark 7.1 As in the register memory case, the interface above considers each input as a single element, However, in the actual implementation, the values $\texttt{clk}$ and base-addr will be specified by a set of 8-bit limbs.

7.5 Data memory constraints assuming large fields

In order to appropriately implement memory checking for the memory locations being accessed at clock cycle $\texttt{clk}$, it is important to enforce that the timestamps $\texttt{ram}j\texttt{-ts-prev}$ associated with the previous values $\texttt{ram}j\texttt{-val-prev}$ stored at addresses base-addr $+\, j$ for $j = 1, 2, 3, 4$ fall in the range $\{0, \ldots, \texttt{clk} - 1\}$.

To achieve this goal using 32-bit range checks, we split each range check of the form $\texttt{ram}j\texttt{-ts-prev} \in \{0, \ldots, \texttt{clk} - 1\}$ for $j = 1, 2, 3, 4$ into two range checks $\texttt{ram}j\texttt{-ts-prev} \in \left[0, 2^{32} - 1\right]$ and $\texttt{clk} - 1 - \texttt{ram}j\texttt{-ts-prev} \in \left[0, 2^{32} - 1\right]$. To implement this idea, we first define the auxiliary variables $\texttt{ram}j\texttt{-ts-prev-aux}$ for $j = 1, 2, 3, 4$ and then create the following set of constraints:

- $\texttt{ram}j\texttt{-ts-prev} \in \left[0, 2^{32} - 1\right]$
- $\texttt{ram}j\texttt{-ts-prev-aux} \in \left[0, 2^{32} - 1\right]$
- $\texttt{ram}j\texttt{-ts-prev-aux} = \texttt{clk} - 1 - \texttt{ram}j\texttt{-ts-prev}$

7.5.1 Range checks

// $\texttt{ram-base-addr} \in \left[0, 2^{32} - 1\right]$ - guaranteed via memory checking

- $\texttt{ram-val1} \in \left[0, 2^{8} - 1\right]$ ▷ Common to instruction execution
- $\texttt{ram-val2} \in \left[0, 2^{8} - 1\right]$ ▷ Common to instruction execution
- $\texttt{ram-val3} \in \left[0, 2^{8} - 1\right]$ ▷ Common to instruction execution
- $\texttt{ram-val4} \in \left[0, 2^{8} - 1\right]$ ▷ Common to instruction execution
- $\texttt{ram1-val-prev} \in \left[0, 2^{8} - 1\right]$
- $\texttt{ram2-val-prev} \in \left[0, 2^{8} - 1\right]$
- $\texttt{ram3-val-prev} \in \left[0, 2^{8} - 1\right]$
- $\texttt{ram4-val-prev} \in \left[0, 2^{8} - 1\right]$

// $\texttt{ram}j\texttt{-ts-prev} \in \{0, \ldots, \texttt{clk} - 1\}$ for $j = 1, 2, 3, 4$

- $\texttt{ram1-ts-prev} \in \left[0, 2^{32} - 1\right]$
- $\texttt{ram2-ts-prev} \in \left[0, 2^{32} - 1\right]$
- $\texttt{ram3-ts-prev} \in \left[0, 2^{32} - 1\right]$
- $\texttt{ram4-ts-prev} \in \left[0, 2^{32} - 1\right]$
- $\texttt{ram1-ts-prev-aux} \in \left[0, 2^{32} - 1\right]$
- $\texttt{ram2-ts-prev-aux} \in \left[0, 2^{32} - 1\right]$
- $\texttt{ram3-ts-prev-aux} \in \left[0, 2^{32} - 1\right]$

- $\texttt{ram4-ts-prev-aux} \in \left[0, 2^{32}-1\right]$

7.5.2 Arithmetic constraints

// Computing $\texttt{ram}j\texttt{-ts-prev-aux} = \texttt{clk} - 1 - \texttt{ram}j\texttt{-ts-prev}$ for $j = 1, 2, 3, 4$

- $\texttt{ram1-ts-prev-aux} = \texttt{clk} - 1 - \texttt{ram1-ts-prev}$
- $\texttt{ram2-ts-prev-aux} = \texttt{clk} - 1 - \texttt{ram2-ts-prev}$
- $\texttt{ram3-ts-prev-aux} = \texttt{clk} - 1 - \texttt{ram3-ts-prev}$
- $\texttt{ram4-ts-prev-aux} = \texttt{clk} - 1 - \texttt{ram4-ts-prev}$

7.5.3 Logup computations

As stated above, one may access up to 4 consecutive memory locations in a clock cycle depending on the instruction that is being executed. For this reason, we define four flags $\texttt{ram}j\texttt{-flag} = \texttt{ram}j\texttt{-accessed}$ for $j = 1, 2, 3, 4$ to indicate whether the set of trace elements $(\texttt{ram}j\texttt{-val-cur}, \texttt{ram}j\texttt{-val-prev}, \texttt{ram}j\texttt{-ts-prev})$ are being accessed during the current clock cycle.

In order to compute the difference between the read set and write set digests between rows $i-1$ and i, we can use these flags as follows:

- $\texttt{ram-read-digest}[i] - \texttt{ram-read-digest}[i-1] =$
  $\texttt{ram1-flag}[i]/(\mathsf{fp}(\texttt{ram-base-addr}[i], \texttt{ram1-val-prev}[i], \texttt{ram1-ts-prev}[i]) + \alpha) +$
  $\texttt{ram2-flag}[i]/(\mathsf{fp}(\texttt{ram-base-addr}[i] + 1, \texttt{ram2-val-prev}[i], \texttt{ram2-ts-prev}[i]) + \alpha) +$
  $\texttt{ram3-flag}[i]/(\mathsf{fp}(\texttt{ram-base-addr}[i] + 2, \texttt{ram3-val-prev}[i], \texttt{ram3-ts-prev}[i]) + \alpha) +$
  $\texttt{ram4-flag}[i]/(\mathsf{fp}(\texttt{ram-base-addr}[i] + 3, \texttt{ram4-val-prev}[i], \texttt{ram4-ts-prev}[i]) + \alpha)$
- $\texttt{ram-write-digest}[i] - \texttt{ram-write-digest}[i-1] =$
  $\texttt{ram1-flag}[i]/(\mathsf{fp}(\texttt{ram-base-addr}[i], \texttt{ram1-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram2-flag}[i]/(\mathsf{fp}(\texttt{ram-base-addr}[i] + 1, \texttt{ram2-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram3-flag}[i]/(\mathsf{fp}(\texttt{ram-base-addr}[i] + 2, \texttt{ram3-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram4-flag}[i]/(\mathsf{fp}(\texttt{ram-base-addr}[i] + 3, \texttt{ram4-val-cur}[i], \texttt{clk}[i]) + \alpha)$

Let fp be a fingerprint function which takes as input the tuple $(\texttt{ram-addr}, \texttt{ram-val}, \texttt{ram-ts})$ and returns a field element in the secure extension field $\texttt{qm31}$ using a random value β chosen by the verifier. Moreover, let $\mathsf{MEM{-}SET}$ denote the set of data memory addresses used by the program over the course of the execution and let $\mathsf{PUB{-}IN{-}SET} \subseteq \mathsf{MEM{-}SET}$ denote its subset of public input locations. Finally, let $\mathsf{init\text{-}pub\text{-}val}(\texttt{pub-in-addr})$ denote the initial value stored at the public input address $\texttt{pub-in-addr}$.

Initial write set digest: Let $\texttt{ram-write-init-digest}$ be the logup sum for the initial state of the data memory (see Section 7.3). That is,

$$\begin{aligned}\texttt{ram-write-init-digest} &= \sum_{\texttt{ram-addr} \in \mathsf{MEM{-}SET} \setminus \mathsf{PUB{-}IN{-}SET}} \frac{1}{\mathsf{fp}(\texttt{ram-addr}, 0, 0) + \alpha} \\ &+ \sum_{\texttt{pub-in-addr} \in \mathsf{PUB{-}IN{-}SET}} \frac{1}{\mathsf{fp}(\texttt{pub-in-addr}, \mathsf{init\text{-}pub\text{-}val}(\texttt{pub-in-addr}), 0) + \alpha}.\end{aligned}$$

Final read set digest: Let $\texttt{ram-read-final-digest}$ be the logup sum for the final state of the data memory. That is,

$$\texttt{ram-read-final-digest} = \sum_{\texttt{ram-addr} \in \mathsf{MEM{-}SET}} \frac{1}{\mathsf{fp}(\texttt{ram-addr}, \texttt{ram-val}, \texttt{ram-ts-final}) + \alpha},$$

where $\texttt{ram-ts-final}$ denotes the timestamp associated with the last access to address $\texttt{ram-addr}$ and $\texttt{ram-val}$ denotes the corresponding byte value stored at this location.

Difference between read and write set digests: Let $\texttt{ram-base-addr}[i]$ be the base address for the memory locations being accessed at row i. Let $\texttt{clk}[i]$ denote the value of $\texttt{clk}$ at row i and let $\texttt{ram}j\texttt{-val-prev}[i]$ and $\texttt{ram}j\texttt{-ts-prev}[i]$ for $j = 1, 2, 3, 4$ denote respectively the previous values and the previous timestamps associated with addresses $\texttt{ram-base-addr}[i], \ldots, \texttt{ram-base-addr}[i] + 3$. The difference between the read set and write set digests between row $i-1$ and row i can be computed as follows:

- $\texttt{ram-read-digest}[i] - \texttt{ram-read-digest}[i-1] =$
  $\texttt{ram1-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i], \texttt{ram1-val-prev}[i], \texttt{ram1-ts-prev}[i]) + \alpha) +$
  $\texttt{ram2-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 1, \texttt{ram2-val-prev}[i], \texttt{ram2-ts-prev}[i]) + \alpha) +$
  $\texttt{ram3-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 2, \texttt{ram3-val-prev}[i], \texttt{ram3-ts-prev}[i]) + \alpha) +$
  $\texttt{ram4-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 3, \texttt{ram4-val-prev}[i], \texttt{ram4-ts-prev}[i]) + \alpha)$
- $\texttt{ram-write-digest}[i] - \texttt{ram-write-digest}[i-1] =$
  $\texttt{ram1-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i], \texttt{ram1-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram2-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 1, \texttt{ram2-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram3-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 2, \texttt{ram3-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram4-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 3, \texttt{ram4-val-cur}[i], \texttt{clk}[i]) + \alpha),$

where $\texttt{ram1-flag}[i], \ldots, \texttt{ram4-flag}[i]$ are the flags indicating whether these addresses are being accessed in row i.

Boundary constraints: Let $\texttt{ram-write-init-digest}$ and $\texttt{ram-read-final-digest}$ be as defined above. The boundary constraints can then be specified as follows:

- $\texttt{ram-read-digest}[1] =$
  $\texttt{ram1-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1], \texttt{ram1-val-prev}[1], \texttt{ram1-ts-prev}[1]) + \alpha) +$
  $\texttt{ram2-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1] + 1, \texttt{ram2-val-prev}[1], \texttt{ram2-ts-prev}[1]) + \alpha) +$
  $\texttt{ram3-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1] + 2, \texttt{ram3-val-prev}[1], \texttt{ram3-ts-prev}[1]) + \alpha) +$
  $\texttt{ram4-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1] + 3, \texttt{ram4-val-prev}[1], \texttt{ram4-ts-prev}[1]) + \alpha)$
- $\texttt{ram-write-digest}[1] = \texttt{ram-write-init-digest} +$
  $\texttt{ram1-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1], \texttt{ram1-val-cur}[1], \texttt{clk}[1]) + \alpha) +$
  $\texttt{ram2-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1] + 1, \texttt{ram2-val-cur}[1], \texttt{clk}[1]) + \alpha) +$
  $\texttt{ram3-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1] + 2, \texttt{ram3-val-cur}[1], \texttt{clk}[1]) + \alpha) +$
  $\texttt{ram4-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1] + 3, \texttt{ram4-val-cur}[1], \texttt{clk}[1]) + \alpha),$
- $\texttt{ram-read-digest}[n] = \texttt{ram-read-final-digest} + \texttt{ram-write-digest}[n].$

Transition constraints $(1 < i \leq n)$: The transition constraints can be specified as follows:

- $\texttt{ram-read-digest}[i] - \texttt{ram-read-digest}[i-1] =$
  $\texttt{ram1-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i], \texttt{ram1-val-prev}[i], \texttt{ram1-ts-prev}[i]) + \alpha) +$
  $\texttt{ram2-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 1, \texttt{ram2-val-prev}[i], \texttt{ram2-ts-prev}[i]) + \alpha) +$
  $\texttt{ram3-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 2, \texttt{ram3-val-prev}[i], \texttt{ram3-ts-prev}[i]) + \alpha) +$
  $\texttt{ram4-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 3, \texttt{ram4-val-prev}[i], \texttt{ram4-ts-prev}[i]) + \alpha)$
- $\texttt{ram-write-digest}[i] - \texttt{ram-write-digest}[i-1] =$
  $\texttt{ram1-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i], \texttt{ram1-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram2-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 1, \texttt{ram2-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram3-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 2, \texttt{ram3-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram4-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 3, \texttt{ram4-val-cur}[i], \texttt{clk}[i]) + \alpha),$

where $\texttt{ram1-flag}[i], \ldots, \texttt{ram4-flag}[i]$ are the flags indicating whether these addresses are being accessed in row i.

7.6 Data memory constraints assuming small fields

7.6.1 Range checks

// $\texttt{ram-base-addr}^{(1)} \in \left[0, 2^8 - 1\right]$ - guaranteed via memory checking

// $\texttt{ram-base-addr}^{(2)} \in \left[0, 2^8 - 1\right]$ - guaranteed via memory checking

// $\texttt{ram-base-addr}^{(3)} \in \left[0, 2^8 - 1\right]$ - guaranteed via memory checking

// $\texttt{ram-base-addr}^{(4)} \in \left[0, 2^8 - 1\right]$ - guaranteed via memory checking

- $\texttt{ram-val1} \in \left[0, 2^8 - 1\right]$ ▷ Common to instruction execution
- $\texttt{ram-val2} \in \left[0, 2^8 - 1\right]$ ▷ Common to instruction execution
- $\texttt{ram-val3} \in \left[0, 2^8 - 1\right]$ ▷ Common to instruction execution
- $\texttt{ram-val4} \in \left[0, 2^8 - 1\right]$ ▷ Common to instruction execution
- $\texttt{ram1-val-prev} \in \left[0, 2^8 - 1\right]$
- $\texttt{ram2-val-prev} \in \left[0, 2^8 - 1\right]$
- $\texttt{ram3-val-prev} \in \left[0, 2^8 - 1\right]$
- $\texttt{ram4-val-prev} \in \left[0, 2^8 - 1\right]$

// $\texttt{ram}i\texttt{-ts-prev} \in \{0, \ldots, \texttt{clk} - 1\}$ for $i = 1, 2, 3, 4$

- $\texttt{ram1-ts-prev}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{ram2-ts-prev}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{ram3-ts-prev}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{ram4-ts-prev}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{ram1-ts-prev-aux}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{ram2-ts-prev-aux}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{ram3-ts-prev-aux}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{ram4-ts-prev-aux}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$

7.6.2 Arithmetic constraints

In order to compute the range checks for $\texttt{ram}i\texttt{-ts-prev}^{(j)}$ for $i = 1, 2, 3, 4$ and $j = 1, 2, 3, 4$ using 8-bit range checks, we need to define auxiliary variables $\texttt{ram}i\texttt{-ts-prev-aux}^{(j)}$ to help with the computation of the expressions in the auxiliary range check. More precisely, in the large field case, we had to define the following constraints for these auxiliary variables:

- $\texttt{ram1-ts-prev-aux} = \texttt{clk} - 1 - \texttt{ram1-ts-prev}$
- $\texttt{ram2-ts-prev-aux} = \texttt{clk} - 1 - \texttt{ram2-ts-prev}$
- $\texttt{ram3-ts-prev-aux} = \texttt{clk} - 1 - \texttt{ram3-ts-prev}$
- $\texttt{ram4-ts-prev-aux} = \texttt{clk} - 1 - \texttt{ram4-ts-prev}$

To enforce these constraints in the small field case, we also need to take borrows into account when enforcing these constraints one limb at a time. As a result, we also define helper values $\texttt{ram1-ts-prev-borrow}^{(j)}$, $\texttt{ram2-ts-prev-borrow}^{(j)}$, $\texttt{ram3-ts-prev-borrow}^{(j)}$, $\texttt{ram4-ts-prev-borrow}^{(j)}$ for $j = 1, 2, 3, 4$ to handle these borrows.

// Computing $\texttt{ram1-ts-prev-aux} = \texttt{clk} - 1 - \texttt{ram1-ts-prev}$

// $\texttt{ram1-ts-prev-borrow}^{(j)}$ for $j = 1, 2, 3, 4$ used for borrow handling

- $\texttt{ram1-ts-prev-aux}^{(1)} + 1 + \texttt{ram1-ts-prev}^{(1)} = \texttt{clk}^{(1)} + \texttt{ram1-ts-prev-borrow}^{(1)} \cdot 2^8$
- $\texttt{ram1-ts-prev-aux}^{(2)} + \texttt{ram1-ts-prev-borrow}^{(1)} + \texttt{ram1-ts-prev}^{(2)} = \texttt{clk}^{(2)} + \texttt{ram1-ts-prev-borrow}^{(2)} \cdot 2^8$
- $\texttt{ram1-ts-prev-aux}^{(3)} + \texttt{ram1-ts-prev-borrow}^{(2)} + \texttt{ram1-ts-prev}^{(3)} = \texttt{clk}^{(3)} + \texttt{ram1-ts-prev-borrow}^{(3)} \cdot 2^8$
- $\texttt{ram1-ts-prev-aux}^{(4)} + \texttt{ram1-ts-prev-borrow}^{(3)} + \texttt{ram1-ts-prev}^{(4)} = \texttt{clk}^{(4)} + \texttt{ram1-ts-prev-borrow}^{(4)} \cdot 2^8$

// Computing $\texttt{ram2-ts-prev-aux} = \texttt{clk} - 1 - \texttt{ram2-ts-prev}$

// $\texttt{ram2-ts-prev-borrow}^{(j)}$ for $j = 1, 2, 3, 4$ used for borrow handling

- $\texttt{ram2-ts-prev-aux}^{(1)} + 1 + \texttt{ram2-ts-prev}^{(1)} = \texttt{clk}^{(1)} + \texttt{ram2-ts-prev-borrow}^{(1)} \cdot 2^8$
- $\texttt{ram2-ts-prev-aux}^{(2)} + \texttt{ram2-ts-prev-borrow}^{(1)} + \texttt{ram2-ts-prev}^{(2)} = \texttt{clk}^{(2)} + \texttt{ram2-ts-prev-borrow}^{(2)} \cdot 2^8$
- $\texttt{ram2-ts-prev-aux}^{(3)} + \texttt{ram2-ts-prev-borrow}^{(2)} + \texttt{ram2-ts-prev}^{(3)} = \texttt{clk}^{(3)} + \texttt{ram2-ts-prev-borrow}^{(3)} \cdot 2^8$

- $\texttt{ram2-ts-prev-aux}^{(4)} + \texttt{ram2-ts-prev-borrow}^{(3)} + \texttt{ram2-ts-prev}^{(4)} = \texttt{clk}^{(4)} + \texttt{ram2-ts-prev-borrow}^{(4)} \cdot 2^8$

// Computing $\texttt{ram3-ts-prev-aux} = \texttt{clk} - 1 - \texttt{ram3-ts-prev}$
// $\texttt{ram3-ts-prev-borrow}^{(j)}$ for $j = 1, 2, 3, 4$ used for borrow handling

- $\texttt{ram3-ts-prev-aux}^{(1)} + 1 + \texttt{ram3-ts-prev}^{(1)} = \texttt{clk}^{(1)} + \texttt{ram3-ts-prev-borrow}^{(1)} \cdot 2^8$
- $\texttt{ram3-ts-prev-aux}^{(2)} + \texttt{ram3-ts-prev-borrow}^{(1)} + \texttt{ram3-ts-prev}^{(2)} = \texttt{clk}^{(2)} + \texttt{ram3-ts-prev-borrow}^{(2)} \cdot 2^8$
- $\texttt{ram3-ts-prev-aux}^{(3)} + \texttt{ram3-ts-prev-borrow}^{(2)} + \texttt{ram3-ts-prev}^{(3)} = \texttt{clk}^{(3)} + \texttt{ram3-ts-prev-borrow}^{(3)} \cdot 2^8$
- $\texttt{ram3-ts-prev-aux}^{(4)} + \texttt{ram3-ts-prev-borrow}^{(3)} + \texttt{ram3-ts-prev}^{(4)} = \texttt{clk}^{(4)} + \texttt{ram3-ts-prev-borrow}^{(4)} \cdot 2^8$

// Computing $\texttt{ram4-ts-prev-aux} = \texttt{clk} - 1 - \texttt{ram4-ts-prev}$
// $\texttt{ram4-ts-prev-borrow}^{(j)}$ for $j = 1, 2, 3, 4$ used for borrow handling

- $\texttt{ram4-ts-prev-aux}^{(1)} + 1 + \texttt{ram4-ts-prev}^{(1)} = \texttt{clk}^{(1)} + \texttt{ram4-ts-prev-borrow}^{(1)} \cdot 2^8$
- $\texttt{ram4-ts-prev-aux}^{(2)} + \texttt{ram4-ts-prev-borrow}^{(1)} + \texttt{ram4-ts-prev}^{(2)} = \texttt{clk}^{(2)} + \texttt{ram4-ts-prev-borrow}^{(2)} \cdot 2^8$
- $\texttt{ram4-ts-prev-aux}^{(3)} + \texttt{ram4-ts-prev-borrow}^{(2)} + \texttt{ram4-ts-prev}^{(3)} = \texttt{clk}^{(3)} + \texttt{ram4-ts-prev-borrow}^{(3)} \cdot 2^8$
- $\texttt{ram4-ts-prev-aux}^{(4)} + \texttt{ram4-ts-prev-borrow}^{(3)} + \texttt{ram4-ts-prev}^{(4)} = \texttt{clk}^{(4)} + \texttt{ram4-ts-prev-borrow}^{(4)} \cdot 2^8$

// Enforcing $\texttt{ram}i\texttt{-ts-prev-borrow}^{(j)} \in \{0, 1\}$ for $i = 1, 2, 3, 4$ and $j = 1, 2, 3$

- $\texttt{ram1-ts-prev-borrow}^{(j)} \cdot (1 - \texttt{ram1-ts-prev-borrow}^{(j)}) = 0$ for $j = 1, 2, 3$
- $\texttt{ram2-ts-prev-borrow}^{(j)} \cdot (1 - \texttt{ram2-ts-prev-borrow}^{(j)}) = 0$ for $j = 1, 2, 3$
- $\texttt{ram3-ts-prev-borrow}^{(j)} \cdot (1 - \texttt{ram3-ts-prev-borrow}^{(j)}) = 0$ for $j = 1, 2, 3$
- $\texttt{ram4-ts-prev-borrow}^{(j)} \cdot (1 - \texttt{ram4-ts-prev-borrow}^{(j)}) = 0$ for $j = 1, 2, 3$

// Enforcing $\texttt{ram}i\texttt{-ts-prev-borrow}^{(4)} = 0$ for $i = 1, 2, 3, 4$

- $\texttt{ram1-ts-prev-borrow}^{(4)} = 0$
- $\texttt{ram2-ts-prev-borrow}^{(4)} = 0$
- $\texttt{ram3-ts-prev-borrow}^{(4)} = 0$
- $\texttt{ram4-ts-prev-borrow}^{(4)} = 0$

7.6.3 Logup computations

In the small field case, the components $\texttt{ram-base-addr}$ and $\texttt{ram}i\texttt{-ts-prev}$ for $i = 1, 2, 3, 4$ are each specified by four field elements representing the 8-bit limb values of the base address and time stamps.

Let fp be a fingerprint function which takes as input the tuple $(\texttt{ram-addr}^{(1)}, \ldots, \texttt{ram-addr}^{(4)}, \texttt{ram-val}, \texttt{ram-ts}^{(1)}, \ldots, \texttt{ram-ts}^{(4)})$ and returns a field element in the secure extension field $\texttt{qm31}$ using a value β chosen by the verifier. To simplify the notation and avoid explicitly describing each limb for the address and timestamp values, we use

$$\mathsf{fp}(\texttt{ram-addr}, \texttt{ram-val}, \texttt{ram-ts})$$

to denote

$$\mathsf{fp}(\texttt{ram-addr}^{(1)}, \ldots, \texttt{ram-addr}^{(4)}, \texttt{ram-val}, \texttt{ram-ts}^{(1)}, \ldots, \texttt{ram-ts}^{(4)})$$

Using fp, we can compute the difference between the read set and write set digests between rows i and $i-1$ as follows:

- $\texttt{ram-read-digest}[i] - \texttt{ram-read-digest}[i-1] =$
  $\texttt{ram1-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i], \texttt{ram1-val-prev}[i], \texttt{ram1-ts-prev}[i]) + \alpha) +$
  $\texttt{ram2-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i] + 1, \texttt{ram2-val-prev}[i], \texttt{ram2-ts-prev}[i]) + \alpha) +$
  $\texttt{ram3-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i] + 2, \texttt{ram3-val-prev}[i], \texttt{ram3-ts-prev}[i]) + \alpha) +$
  $\texttt{ram4-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i] + 3, \texttt{ram4-val-prev}[i], \texttt{ram4-ts-prev}[i]) + \alpha)$
- $\texttt{ram-write-digest}[i] - \texttt{ram-write-digest}[i-1] =$
  $\texttt{ram1-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i], \texttt{ram1-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram2-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i] + 1, \texttt{ram2-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram3-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i] + 2, \texttt{ram3-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram4-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i] + 3, \texttt{ram4-val-cur}[i], \texttt{clk}[i]) + \alpha)$,

where $\texttt{ram-base-addr}[i] + j \coloneqq (\texttt{ram-base-addr}[i]^{(1)} + j, \texttt{ram-base-addr}[i]^{(2)}, \texttt{ram-base-addr}[i]^{(3)}, \texttt{ram-base-addr}[i]^{(4)})$ for $j = 1, 2, 3$. As before, $\texttt{ram}j\texttt{-flag}[i] = \texttt{ram}j\texttt{-accessed}[i]$ for $j = 1, 2, 3, 4$

are four flags that indicate whether the set of trace elements ($\texttt{ram}j\texttt{-val-cur}[i], \texttt{ram}j\texttt{-val-prev}[i]$, $\texttt{ram}j\texttt{-ts-prev}[i]$) are being accessed during the i-th clock cycle.

Initial write set digest: Let `ram-write-init-digest` be the logup sum for the initial state of the data memory (see Section 7.3). That is,

$$\begin{aligned}\texttt{ram-write-init-digest} \quad &= \sum_{\texttt{ram-addr} \in \mathsf{MEM\text{--}SET} \setminus \mathsf{PUB\text{--}IN\text{--}SET}} \frac{1}{\mathsf{fp}(\texttt{ram-addr}, 0, \mathbf{0}) + \alpha} \\ &+ \sum_{\texttt{pub-in-addr} \in \mathsf{PUB\text{--}IN\text{--}SET}} \frac{1}{\mathsf{fp}(\texttt{pub-in-addr}, \mathsf{init\text{-}pub\text{-}val}(\texttt{pub-in-addr}), \mathbf{0}) + \alpha},\end{aligned}$$

where

- $\texttt{ram-addr} = (\texttt{ram-addr}^{(1)}, \texttt{ram-addr}^{(2)}, \texttt{ram-addr}^{(3)}, \texttt{ram-addr}^{(4)})$,
- $\texttt{pub-in-addr} = (\texttt{pub-in-addr}^{(1)}, \texttt{pub-in-addr}^{(2)}, \texttt{pub-in-addr}^{(3)}, \texttt{pub-in-addr}^{(4)})$, and
- $\mathbf{0} = (0, 0, 0, 0)$.

Final read set digest: Let `ram-read-final-digest` be the logup sum for the final state of the data memory. That is,

$$\texttt{ram-read-final-digest} \quad = \sum_{\texttt{ram-addr} \in \mathsf{MEM\text{--}SET}} \frac{1}{\mathsf{fp}(\texttt{ram-addr}, \texttt{ram-val}, \texttt{ram-ts-final}) + \alpha},$$

where $\texttt{ram-ts-final} = (\texttt{ram-ts-final}^{(1)}, \ldots, \texttt{ram-ts-final}^{(4)})$ denotes the timestamp associated with the last access to address $\texttt{ram-addr} = (\texttt{ram-addr}^{(1)}, \ldots, \texttt{ram-addr}^{(4)})$ and `ram-val` corresponds to the byte value stored at this location.

Remark 7.2 The values `ram-addr`, `ram-val`, and `ram-ts-final` get committed to the trace. Especially, when there is public output (including exit status code), the values in the public output can be placed in a separate trace or in the same trace as the program memory.

Difference between read and write set digests: Let

- $\texttt{ram-base-addr}[i] = (\texttt{ram-base-addr}[i]^{(1)}, \ldots, \texttt{ram-base-addr}[i]^{(4)})$ be the base address for the memory locations being accessed at row i.
- $\texttt{clk}[i] = (\texttt{clk}[i]^{(1)}, \ldots, \texttt{clk}[i]^{(4)})$ denote the 4 limbs of the value of `clk` at row i.
- $\texttt{ram}j\texttt{-val-prev}[i]$ and $\texttt{ram}j\texttt{-ts-prev}[i]$ for $j = 1, 2, 3, 4$ denote respectively the previous values and the previous timestamps associated with addresses $\texttt{ram-base-addr}[i], \ldots, \texttt{ram-base-addr}[i] + 3$, where $\texttt{ram-base-addr}[i] + j := (\texttt{ram-base-addr}[i]^{(1)} + j, \texttt{ram-base-addr}[i]^{(2)}, \texttt{ram-base-addr}[i]^{(3)}, \texttt{ram-base-addr}[i]^{(4)})$ for $j = 1, 2, 3$.

The difference between the read set and write set digests between row $i-1$ and row i can be computed as follows:

- $\texttt{ram-read-digest}[i] - \texttt{ram-read-digest}[i-1] =$
  $\texttt{ram1-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i], \texttt{ram1-val-prev}[i], \texttt{ram1-ts-prev}[i]) + \alpha) +$
  $\texttt{ram2-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i] + 1, \texttt{ram2-val-prev}[i], \texttt{ram2-ts-prev}[i]) + \alpha) +$
  $\texttt{ram3-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i] + 2, \texttt{ram3-val-prev}[i], \texttt{ram3-ts-prev}[i]) + \alpha) +$
  $\texttt{ram4-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i] + 3, \texttt{ram4-val-prev}[i], \texttt{ram4-ts-prev}[i]) + \alpha)$
- $\texttt{ram-write-digest}[i] - \texttt{ram-write-digest}[i-1] =$
  $\texttt{ram1-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i], \texttt{ram1-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram2-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i] + 1, \texttt{ram2-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram3-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i] + 2, \texttt{ram3-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram4-flag}[i] / (\mathsf{fp}(\texttt{ram-base-addr}[i] + 3, \texttt{ram4-val-cur}[i], \texttt{clk}[i]) + \alpha)$,

where $\texttt{ram1-flag}[i], \ldots, \texttt{ram4-flag}[i]$ are the flags indicating whether these addresses are being accessed in row i.

Boundary constraints: Let `ram-write-init-digest` and `ram-read-final-digest` be as defined above. The boundary constraints can then be specified as follows:

- $\texttt{ram-read-digest}[1] =$
  $\texttt{ram1-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1], \texttt{ram1-val-prev}[1], \texttt{ram1-ts-prev}[1]) + \alpha) +$
  $\texttt{ram2-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1] + 1, \texttt{ram2-val-prev}[1], \texttt{ram2-ts-prev}[1]) + \alpha) +$
  $\texttt{ram3-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1] + 2, \texttt{ram3-val-prev}[1], \texttt{ram3-ts-prev}[1]) + \alpha) +$
  $\texttt{ram4-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1] + 3, \texttt{ram4-val-prev}[1], \texttt{ram4-ts-prev}[1]) + \alpha)$
- $\texttt{ram-write-digest}[1] = \texttt{ram-write-init-digest} +$
  $\texttt{ram1-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1], \texttt{ram1-val-cur}[1], \texttt{clk}[1]) + \alpha) +$
  $\texttt{ram2-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1] + 1, \texttt{ram2-val-cur}[1], \texttt{clk}[1]) + \alpha) +$
  $\texttt{ram3-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1] + 2, \texttt{ram3-val-cur}[1], \texttt{clk}[1]) + \alpha) +$
  $\texttt{ram4-flag}[1]/(\texttt{fp}(\texttt{ram-base-addr}[1] + 3, \texttt{ram4-val-cur}[1], \texttt{clk}[1]) + \alpha),$
- $\texttt{ram-read-digest}[n] = \texttt{ram-read-final-digest} + \texttt{ram-write-digest}[n],$

where $\texttt{ram1-flag}[i], \ldots, \texttt{ram4-flag}[i]$ are the flags indicating whether these addresses are being accessed in row i.

Transition constraints $(1 < i \leq n)$: The transition constraints can be specified as follows:

- $\texttt{ram-read-digest}[i] - \texttt{ram-read-digest}[i-1] =$
  $\texttt{ram1-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i], \texttt{ram1-val-prev}[i], \texttt{ram1-ts-prev}[i]) + \alpha) +$
  $\texttt{ram2-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 1, \texttt{ram2-val-prev}[i], \texttt{ram2-ts-prev}[i]) + \alpha) +$
  $\texttt{ram3-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 2, \texttt{ram3-val-prev}[i], \texttt{ram3-ts-prev}[i]) + \alpha) +$
  $\texttt{ram4-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 3, \texttt{ram4-val-prev}[i], \texttt{ram4-ts-prev}[i]) + \alpha)$
- $\texttt{ram-write-digest}[i] - \texttt{ram-write-digest}[i-1] =$
  $\texttt{ram1-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i], \texttt{ram1-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram2-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 1, \texttt{ram2-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram3-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 2, \texttt{ram3-val-cur}[i], \texttt{clk}[i]) + \alpha) +$
  $\texttt{ram4-flag}[i]/(\texttt{fp}(\texttt{ram-base-addr}[i] + 3, \texttt{ram4-val-cur}[i], \texttt{clk}[i]) + \alpha),$

where $\texttt{ram1-flag}[i], \ldots, \texttt{ram4-flag}[i]$ are the flags indicating whether these addresses are being accessed in row i.

7.7 Constraints and logup computation for initial write and final read sets

In order to help compute the logup contributions related to the initial write set and the final read set mentioned in Section 7.3, we define additional trace columns and constraints for the data memory component. When doing so, we will make a distinction between public and private parts of the execution trace, where the public part will contain elements which are known to the verifier and whose commitments the verifier can efficiently check.

7.7.1 Trace elements for initial write and final read sets

In addition to the trace elements defined in Section 7.2, this section defines additional elements to help with the computation of the initial write set (which depends on the public I/O values) and the final read set.

- `ram-init-final-addr`: the memory address given for each byte in the RAM ever touched or relevant for public I/O (located in private part of the execution trace and *not* known to the verifier)
- `ram-val-final`: 8-bit values of the final RAM, given byte-wise (located in a private part of the execution trace and *not* known to verifier)

- `ram-val-init`: a helper column containing 8-bit values of the initial RAM (located in a private part of the execution trace and *not* known to verifier)
- `ram-ts-final`: the timestamp associated with the last access to address `ram-init-final-addr` (located in a private part of the execution trace and *not* known to verifier)
- `ram-init-final-flag`: a flag indicating whether `ram-final`, `ram-init` columns on the current row are being used (located in a private part of the execution trace and *not* known to verifier).
- `pub-in-flag`: a flag indicating whether (`pub-io-addr`, `pub-in-val`) on the current row is considered as public input (located in a public part of the execution trace). If `pub-in-flag` is set, `ram-init-final-flag` also needs to be set.
- `pub-io-addr`: the same value as in `ram-init-final-addr` but only for public input and output (located in a public part of the execution trace). Needs to be equal to `ram-init-final-addr` if either `pub-in-flag` or `pub-out-flag` is set.
- `pub-in-val`: 8-bit values of the public input, given byte-wise (located in a public part of the execution trace).
- `pub-out-val`: 8-bit values of the public output, given byte-wise (located in a public part of the execution trace). Needs to be equal to `ram-val-final` on the same row when `pub-out-flag` is set
- `pub-out-flag`: flag indicating whether (`pub-io-addr`, `pub-out-val`) on the current row is considered as public output (located in a public part of the execution trace). If `pub-out-flag` is set, `ram-init-final-flag` needs to be also set.

Remarks

- `ram-init-final-addr` should include all the read-write memory addresses ever accessed during the execution and all the addresses in the public input.
- The RAM is initialized with zero if `pub-in-flag` is false.
- The initial digest computation determines the initial memory content as follows:
  - For every `ram-init-final-addr` with `ram-init-final-flag` set, the initial value of the RAM `ram-val-init` is equal to `pub-in-flag` · `pub-in-val`. In particular, this latter value is zero when the address is not part of the public input.

7.7.2 Arithmetic constraints

Constraints for data memory public I/O consistency

// Enforcing $\texttt{ram-init-final-addr}^{(i)} = \texttt{pub-io-addr}^{(i)}$ for $i = 1, 2, 3, 4$ for public I/O addresses
// The sum of the flags might be two, but any non-zero value

- $(\texttt{pub-in-flag} + \texttt{pub-out-flag}) \cdot (\texttt{ram-init-final-addr}^{(1)} - \texttt{pub-io-addr}^{(1)}) = 0$
- $(\texttt{pub-in-flag} + \texttt{pub-out-flag}) \cdot (\texttt{ram-init-final-addr}^{(2)} - \texttt{pub-io-addr}^{(2)}) = 0$
- $(\texttt{pub-in-flag} + \texttt{pub-out-flag}) \cdot (\texttt{ram-init-final-addr}^{(3)} - \texttt{pub-io-addr}^{(3)}) = 0$
- $(\texttt{pub-in-flag} + \texttt{pub-out-flag}) \cdot (\texttt{ram-init-final-addr}^{(4)} - \texttt{pub-io-addr}^{(4)}) = 0$

// Enforcing $\texttt{ram-val-final} = \texttt{pub-out-val}$ when $\texttt{pub-out-flag} = 1$

- $(\texttt{pub-out-flag}) \cdot (\texttt{ram-val-final} - \texttt{pub-out-val}) = 0$

// Enforcing $\texttt{ram-val-init} = \texttt{pub-in-flag} \cdot \texttt{pub-in-val}$

- $(\texttt{pub-in-flag} \cdot \texttt{pub-in-val} - \texttt{ram-val-final}) = 0$

Constraints for ensuring uniqueness of `ram-init-final-addr` values

Since `ram-init-final-addr` values are private witnesses, we need to ensure that there are no duplicates. For this reason, we constrain the relevant `ram-init-final-addr` values to be strictly monotonically increasing, though gaps in the address range are allowed.

Let i be the index of the current row. The precise guaranteed condition is the following:

- If row i is the last row in the trace (i.e., $\texttt{is-last}[i] = 1$), nothing is enforced.

- If row i is not the last row (i.e., $\texttt{is-last}[i] = 0$), then the following must be enforced:
  - If the value of $\texttt{ram-init-final-addr}[i+1]$ is being used (i.e., $\texttt{ram-init-final-flag}[i+1] = 1$), then $\texttt{ram-init-final-addr}[i+1]$ should be strictly larger than $\texttt{ram-init-final-addr}[i]$ (i.e., $\texttt{ram-init-final-addr}[i+1] > \texttt{ram-init-final-addr}[i]$)
  - If the value of $\texttt{ram-init-final-addr}[i+1]$ is not being used (i.e., $\texttt{ram-init-final-flag}[i+1] = 0$), then $\texttt{ram-init-final-addr}[i+1] = \texttt{ram-init-final-addr}[i]$

In order to enforce the above conditions:

- let ram-addr-cur denote the value of ram-init-final-addr in row i;
- let ram-addr-next denote the value of ram-init-final-addr in row $i+1$;
- let is-last be a flag that indicates whether row i is the last row in the trace;
- let ram-addr-diff be a helper variable used to compute the difference between ram-addr-cur andram-addr-next;
- let ram-addr-borrow be a helper borrow variable; and
- let ram-init-final-flag-next denote the value of ram-init-final-flag in row $i+1$.

// Setting ram-addr-borrow to the borrow value for ram-addr-cur $-$ ram-addr-next

- $(\texttt{ram-addr-cur}^{(1)} + \texttt{ram-addr-borrow}^{(1)} \cdot 2^8 - \texttt{ram-addr-next}^{(1)} - \texttt{ram-addr-diff}^{(1)}) = 0$
- $(\texttt{ram-addr-cur}^{(2)} + \texttt{ram-addr-borrow}^{(2)} \cdot 2^8 - \texttt{ram-addr-next}^{(2)} - \texttt{ram-addr-diff}^{(2)} - \texttt{ram-addr-borrow}^{(1)}) = 0$
- $(\texttt{ram-addr-cur}^{(3)} + \texttt{ram-addr-borrow}^{(3)} \cdot 2^8 - \texttt{ram-addr-next}^{(3)} - \texttt{ram-addr-diff}^{(3)} - \texttt{ram-addr-borrow}^{(2)}) = 0$
- $(\texttt{ram-addr-cur}^{(4)} + \texttt{ram-addr-borrow}^{(4)} \cdot 2^8 - \texttt{ram-addr-next}^{(4)} - \texttt{ram-addr-diff}^{(4)} - \texttt{ram-addr-borrow}^{(3)}) = 0$

// Enforcing increment if the value ram-init-final-addr in the next row is being used

- $(1 - \texttt{is-last})(\texttt{ram-init-final-flag-next}) \cdot (1 - \texttt{ram-addr-borrow}^{(4)}) = 0$

// Enforcing no change if the value ram-init-final-addr in the next row is not being used

- $(1 - \texttt{is-last})(1 - \texttt{ram-init-final-flag-next}) \cdot (\texttt{ram-addr-next}^{(1)} - \texttt{ram-addr-cur}^{(1)}) = 0$
- $(1 - \texttt{is-last})(1 - \texttt{ram-init-final-flag-next}) \cdot (\texttt{ram-addr-next}^{(2)} - \texttt{ram-addr-cur}^{(2)}) = 0$
- $(1 - \texttt{is-last})(1 - \texttt{ram-init-final-flag-next}) \cdot (\texttt{ram-addr-next}^{(3)} - \texttt{ram-addr-cur}^{(3)}) = 0$
- $(1 - \texttt{is-last})(1 - \texttt{ram-init-final-flag-next}) \cdot (\texttt{ram-addr-next}^{(4)} - \texttt{ram-addr-cur}^{(4)}) = 0$

// Enforcing $\texttt{ram-addr-borrow}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\texttt{ram-addr-borrow}^{(1)}) \cdot (1 - \texttt{ram-addr-borrow}^{(1)}) = 0$
- $(\texttt{ram-addr-borrow}^{(2)}) \cdot (1 - \texttt{ram-addr-borrow}^{(2)}) = 0$
- $(\texttt{ram-addr-borrow}^{(3)}) \cdot (1 - \texttt{ram-addr-borrow}^{(3)}) = 0$
- $(\texttt{ram-addr-borrow}^{(4)}) \cdot (1 - \texttt{ram-addr-borrow}^{(4)}) = 0$

7.7.3 Range checks

// Enforcing ranges for additional data memory variables

// $\texttt{ram-val-final}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by range checks during memory checking

// $\texttt{ram-ts-final}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Does not need to be reinforced

- $\texttt{ram-init-final-addr}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $\texttt{ram-addr-diff}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $(\texttt{ram-init-final-flag}) \cdot (1 - \texttt{ram-init-final-flag}) = 0$

// Enforcing ranges for public I/O variables

// $\texttt{pub-out-val} \in \left[0, 2^8 - 1\right]$ - Implied by constraints

// $\texttt{pub-io-addr}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by constraints

- $\texttt{pub-in-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $(\texttt{pub-out-flag}) \cdot (1 - \texttt{pub-out-flag}) = 0$
- $(\texttt{pub-in-flag}) \cdot (1 - \texttt{pub-in-flag}) = 0$

7.7.4 Logup computation for initial write and final read sets

In order to compute the logup sums for the initial and final states of the data memory, as described in Section 7.6.3, the prover creates two additional columns:

- `ram-write-init-digest`: a digest column used to compute the logup sum for the initial state of the data memory;
- `ram-read-final-digest`: a digest column used to compute the logup sum for the final state of the data memory.

Boundary constraints: Let $\mathbf{0} = (0, 0, 0, 0)$. The boundary constraints are as follows:

- $\texttt{ram-write-init-digest}[1] =$ $\texttt{ram-init-final-flag}[1]/(\mathsf{fp}(\texttt{ram-init-final-addr}[1], \texttt{ram-val-init}[1], \mathbf{0}) + \alpha)$
- $\texttt{ram-read-final-digest}[1] =$ $\texttt{ram-init-final-flag}[1]/(\mathsf{fp}(\texttt{ram-init-final-addr}[1], \texttt{ram-val-final}[1], \texttt{ram-ts-final}[1]) + \alpha)$

Transition constraints $(1 < i \leq n)$: Let $\mathbf{0} = (0, 0, 0, 0)$. The transition constraints are as follows:

- $\texttt{ram-write-init-digest}[i] - \texttt{ram-write-init-digest}[i-1] =$ $\texttt{ram-init-final-flag}[i]/(\mathsf{fp}(\texttt{ram-init-final-addr}[i], \texttt{ram-val-init}[i], \mathbf{0}) + \alpha)$
- $\texttt{ram-write-digest}[i] - \texttt{ram-write-digest}[i-1] =$ $\texttt{ram-init-final-flag}[i]/(\mathsf{fp}(\texttt{ram-init-final-addr}[i], \texttt{ram-val-final}[i], \texttt{ram-ts-final}[i]) + \alpha)$

8 Execution component

The instruction execution component deals with the actual execution of instructions by enforcing constraints that guarantee the correct execution of these instructions over their operands. In some cases, such as for load and store instructions, this may require interactions with other components. In this section, we describe constraints for all instructions supported by the Nexus Virtual Machine.

8.1 Instruction execution trace elements

In order to enforce instruction execution constraints, we keep track of the set of operands used by virtual machine instructions as well as some helper values. Moreover, each instruction has a flag associated with it, described in the CPU component, in order to help with the update of the machine state. Since some of the instructions may require interaction with the data memory component, we also require the trace column `clk`.

The following set of trace elements will be needed for the instruction executor component:

- `clk`: the current execution time
- $\mathsf{a}_{\mathsf{val}}$: a 32-bit word specifying the value of operand op-a
- $\mathsf{b}_{\mathsf{val}}$: a 32-bit word specifying the value of operand op-b
- $\mathsf{c}_{\mathsf{val}}$: a 32-bit word specifying the value of operand op-c
- `pc`: the current value of the program counter register
- `pc-next`: the next value of the program counter register after the execution
- $\texttt{is-lui}, \ldots, \texttt{is-and}$: boolean flags for supported instructions (see Table 5)
- $\texttt{h1}, \ldots, \texttt{hn}$: helper elements (defined as needed)

8.2 Instruction execution interface

In order to clarify the interaction with other components, we now define the interface that these other components can use to call the instruction execution component. For this, we assume that the instruction opcode as defined in the instruction encoding will be passed as a parameter together with the three operands, denoted a-val, b-val, and c-val, and the value of the current program counter pc.

Table 5: List of instructions flags for the Nexus Virtual Machine Instruction Set.

Instruction flag	Description
`is-lui`	indicates an `lui` operation
`is-auipc`	indicates an `auipc` operation
`is-jal`	indicates an `jal` operation
`is-jalr`	indicates an `jalr` operation
`is-ecall`	indicates an `ecall` operation
`is-ebreak`	indicates an `ebreak` operation
`is-fence`	indicates an `fence` operation
`is-unimp`	indicates an `unimp` operation
`is-beq`	indicates an `beq` operation
`is-bne`	indicates an `bne` operation
`is-blt`	indicates an `blt` operation
`is-bge`	indicates an `bge` operation
`is-bltu`	indicates an `bltu` operation
`is-bgeu`	indicates an `bgeu` operation
`is-lb`	indicates an `lb` operation
`is-lh`	indicates an `lh` operation
`is-lw`	indicates an `lw` operation
`is-lbu`	indicates an `lbu` operation
`is-lhu`	indicates an `lhu` operation
`is-sb`	indicates an `sb` operation
`is-sh`	indicates an `sh` operation
`is-sw`	indicates an `sw` operation
`is-add`	indicates an `add` or `addi` operation
`is-sub`	indicates an `sub` operation
`is-sll`	indicates an `sll` or `slli` operation
`is-slt`	indicates an `slt` or `slti` operation
`is-sltu`	indicates an `sltu` or `sltiu` operation
`is-xor`	indicates an `xor` or `xori` operation
`is-srl`	indicates an `srl` or `srli` operation
`is-sra`	indicates an `sra` or `srai` operation
`is-or`	indicates an `or` or `ori` operation
`is-and`	indicates an `and` or `andi` operation
`is-pad`	used for padding, not a computational step

- $\mathsf{exec}(\mathrm{pc}, \mathrm{opcode}, \mathrm{a}\text{-}\mathsf{val}, \mathrm{b}\text{-}\mathsf{val}, \mathrm{c}\text{-}\mathsf{val}) \mapsto \mathrm{pc}\text{-}\mathsf{next}$: the instruction execution component performs the operation over the values $(\mathrm{a}\text{-}\mathsf{val}, \mathrm{b}\text{-}\mathsf{val}, \mathrm{c}\text{-}\mathsf{val})$ according to the opcode value opcode. It also sets pc-next to the updated value of the program counter value pc.

Remark 8.1 The interface above considers each value as a single element, However, in certain cases, some of these entries will be specified by a set of 8-bit limbs.

Remark 8.2 To make use of the common structure of certain instructions such as ADD and ADDI, operations such as sign extension of immediate values are performed before calling the interface to the instruction execution component.

Notation: When introducing limbs for a variable val, we use the notation "$\mathsf{val}^{(j)}$" to indicate the j-th limb for a given variable val.

8.3 Basic Instruction Set: flags

// `is-alu includes instructions with and without immediate values`
- $(\mathtt{is\text{-}add} + \mathtt{is\text{-}sub} + \mathtt{is\text{-}slt} + \mathtt{is\text{-}sltu} + \mathtt{is\text{-}xor} + \mathtt{is\text{-}or} + \mathtt{is\text{-}and} + \mathtt{is\text{-}sll} + \mathtt{is\text{-}srl} + \mathtt{is\text{-}sra} - \mathtt{is\text{-}alu}) = 0$

// `is-load includes load instructions`
- $(\mathtt{is\text{-}lb} + \mathtt{is\text{-}lh} + \mathtt{is\text{-}lw} + \mathtt{is\text{-}lbu} + \mathtt{is\text{-}lhu} - \mathtt{is\text{-}load}) = 0$

// `is-type-s includes store instructions`
- $(\mathtt{is\text{-}sb} + \mathtt{is\text{-}sh} + \mathtt{is\text{-}sw} - \mathtt{is\text{-}type\text{-}s}) = 0$

// `is-type-b includes branch instructions`
- $(\mathtt{is\text{-}beq} + \mathtt{is\text{-}bne} + \mathtt{is\text{-}blt} + \mathtt{is\text{-}bge} + \mathtt{is\text{-}bltu} + \mathtt{is\text{-}bgeu} - \mathtt{is\text{-}type\text{-}b}) = 0$

// `is-type-u includes lui and auipc instructions`
- $(\mathtt{is\text{-}lui} + \mathtt{is\text{-}auipc} - \mathtt{is\text{-}type\text{-}u}) = 0$

// `is-type-sys includes ebreak and ecall instructions`
- $(\mathtt{is\text{-}ecall} + \mathtt{is\text{-}ebreak} - \mathtt{is\text{-}type\text{-}sys}) = 0$

// `is-type-j includes the jal instruction`
- $(\mathtt{is\text{-}jal} - \mathtt{is\text{-}type\text{-}j}) = 0$

8.4 Basic Instruction Set: Common constraints

8.4.1 Constraints assuming large fields

// `Instructions for which PC always increments by 4,`
// `Except when the next row is the first row`
- $(\mathtt{is\text{-}alu} + \mathtt{is\text{-}load} + \mathtt{is\text{-}type\text{-}s} + \mathtt{is\text{-}type\text{-}u} - \mathtt{is\text{-}pc\text{-}inc\text{-}std}) = 0$

// `Incrementing PC by 4`
- $(\mathtt{is\text{-}pc\text{-}inc\text{-}std}) \cdot (\mathtt{pc}\text{-}\mathsf{next} + \mathtt{pc\text{-}carry} \cdot 2^{32} - \mathtt{pc} - 4) = 0$

// `Enforcing pc-carry` $\in \{0, 1\}$
- $(\mathtt{is\text{-}pc\text{-}inc\text{-}std}) \cdot (\mathtt{pc\text{-}carry}) \cdot (1 - \mathtt{pc\text{-}carry}) = 0$

8.4.2 Constraints assuming small fields

// `Instructions for which PC always increments by 4,`
// `Except when the next row is the first row`
- $(\mathtt{is\text{-}alu} + \mathtt{is\text{-}load} + \mathtt{is\text{-}type\text{-}s} + \mathtt{is\text{-}type\text{-}u} - \mathtt{is\text{-}pc\text{-}inc\text{-}std}) = 0$

// `Incrementing PC by 4`
// `pc-carry used for carry handling`
- $(\mathtt{is\text{-}pc\text{-}inc\text{-}std}) \cdot (\mathtt{pc}\text{-}\mathsf{next}^{(1)} + \mathtt{pc\text{-}carry}^{(1)} \cdot 2^{8} - \mathtt{pc}^{(1)} - 4) = 0$
- $(\mathtt{is\text{-}pc\text{-}inc\text{-}std}) \cdot (\mathtt{pc}\text{-}\mathsf{next}^{(2)} + \mathtt{pc\text{-}carry}^{(2)} \cdot 2^{8} - \mathtt{pc}^{(2)} - \mathtt{pc\text{-}carry}^{(1)}) = 0$

- $(\texttt{is-pc-inc-std}) \cdot (\text{pc-next}^{(3)} + \texttt{pc-carry}^{(3)} \cdot 2^8 - \texttt{pc}^{(3)} - \texttt{pc-carry}^{(2)}) = 0$
- $(\texttt{is-pc-inc-std}) \cdot (\text{pc-next}^{(4)} + \texttt{pc-carry}^{(4)} \cdot 2^8 - \texttt{pc}^{(4)} - \texttt{pc-carry}^{(3)}) = 0$

// Enforcing pc-carry$^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$
- $(\texttt{is-pc-inc-std}) \cdot (\texttt{pc-carry}^{(1)}) \cdot (1 - \texttt{pc-carry}^{(1)}) = 0$
- $(\texttt{is-pc-inc-std}) \cdot (\texttt{pc-carry}^{(2)}) \cdot (1 - \texttt{pc-carry}^{(2)}) = 0$
- $(\texttt{is-pc-inc-std}) \cdot (\texttt{pc-carry}^{(3)}) \cdot (1 - \texttt{pc-carry}^{(3)}) = 0$
- $(\texttt{is-pc-inc-std}) \cdot (\texttt{pc-carry}^{(4)}) \cdot (1 - \texttt{pc-carry}^{(4)}) = 0$

8.5 Basic Instruction Set: ALU Instructions

8.5.1 ADD Instruction

The parameters and functionality for the ADD instruction are as follows:

- opcode: ADD
- Parameters: $(\text{a-val}, \text{b-val}, \text{c-val})$
- Instruction selector: $\texttt{is-add} = 1$
- Functionality: $\text{a-val} \leftarrow \text{b-val} + \text{c-val} \bmod 2^{32}$

The mapping from the add and addi instructions in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
add	rd	rs1	rs2	0	R[rd]	R[rs1]	R[rs2]
addi	rd	rs1	rs2	1	R[rd]	R[rs1]	sext(i)

where sext is the sign extension function and i is a 12-bit immediate value.

Constraints assuming large fields

// Carry handling
- $(\texttt{is-add}) \cdot (\text{a-val} + \texttt{h-carry} \cdot 2^{32} - \text{b-val} - \text{c-val}) = 0$

// Enforcing h-carry $\in \{0, 1\}$
- $(\texttt{is-add}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// Range check a-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check b-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check c-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check is-add $\in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Carry handling
- $(\texttt{is-add}) \cdot (\text{a-val}^{(1)} + \texttt{h-carry}^{(1)} \cdot 2^8 - \text{b-val}^{(1)} - \text{c-val}^{(1)}) = 0$
- $(\texttt{is-add}) \cdot (\text{a-val}^{(2)} + \texttt{h-carry}^{(2)} \cdot 2^8 - \text{b-val}^{(2)} - \text{c-val}^{(2)} - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-add}) \cdot (\text{a-val}^{(3)} + \texttt{h-carry}^{(3)} \cdot 2^8 - \text{b-val}^{(3)} - \text{c-val}^{(3)} - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-add}) \cdot (\text{a-val}^{(4)} + \texttt{h-carry}^{(4)} \cdot 2^8 - \text{b-val}^{(4)} - \text{c-val}^{(4)} - \texttt{h-carry}^{(3)}) = 0$

// Enforcing h-carry$^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$
- $(\texttt{is-add}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-add}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-add}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$

- $(\texttt{is-add}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// Range check a-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check b-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check c-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check is-add $\in \{0, 1\}$ - Performed in the CPU component

8.5.2 SUB Instruction

The parameters and functionality for the SUB instruction are as follows:

- opcode: SUB
- Parameters: $(\textsf{a-val}, \textsf{b-val}, \textsf{c-val})$
- Instruction selector: $\texttt{is-sub} = 1$
- Functionality: $\textsf{a-val} \leftarrow \textsf{b-val} - \textsf{c-val} \bmod 2^{32}$

The mapping from the sub instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
sub	rd	rs1	rs2	0	R[rd]	R[rs1]	R[rs2]

Constraints assuming large fields

// Carry handling
- $(\texttt{is-sub}) \cdot (\textsf{b-val} + \texttt{h-borrow} \cdot 2^{32} - \textsf{a-val} - \textsf{c-val}) = 0$

// Enforcing h-borrow $\in \{0, 1\}$
- $(\texttt{is-sub}) \cdot (\texttt{h-borrow}) \cdot (1 - \texttt{h-borrow}) = 0$

// Range check a-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check b-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check c-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check is-sub $\in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Borrow handling
- $(\texttt{is-sub}) \cdot (\textsf{b-val}^{(1)} + \texttt{h-borrow}^{(1)} \cdot 2^8 - \textsf{a-val}^{(1)} - \textsf{c-val}^{(1)}) = 0$
- $(\texttt{is-sub}) \cdot (\textsf{b-val}^{(2)} + \texttt{h-borrow}^{(2)} \cdot 2^8 - \textsf{a-val}^{(2)} - \textsf{c-val}^{(2)} - \texttt{h-borrow}^{(1)}) = 0$
- $(\texttt{is-sub}) \cdot (\textsf{b-val}^{(3)} + \texttt{h-borrow}^{(3)} \cdot 2^8 - \textsf{a-val}^{(3)} - \textsf{c-val}^{(3)} - \texttt{h-borrow}^{(2)}) = 0$
- $(\texttt{is-sub}) \cdot (\textsf{b-val}^{(4)} + \texttt{h-borrow}^{(4)} \cdot 2^8 - \textsf{a-val}^{(4)} - \textsf{c-val}^{(4)} - \texttt{h-borrow}^{(3)}) = 0$

// Enforcing h-borrow$^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$
- $(\texttt{is-sub}) \cdot (\texttt{h-borrow}^{(1)}) \cdot (1 - \texttt{h-borrow}^{(1)}) = 0$
- $(\texttt{is-sub}) \cdot (\texttt{h-borrow}^{(2)}) \cdot (1 - \texttt{h-borrow}^{(2)}) = 0$
- $(\texttt{is-sub}) \cdot (\texttt{h-borrow}^{(3)}) \cdot (1 - \texttt{h-borrow}^{(3)}) = 0$
- $(\texttt{is-sub}) \cdot (\texttt{h-borrow}^{(4)}) \cdot (1 - \texttt{h-borrow}^{(4)}) = 0$

// Range check a-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check b-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check c-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check is-sub $\in \{0, 1\}$ - Performed in the CPU component

8.5.3 SLTU Instruction

The parameters and functionality for the SLTU instruction are as follows:

- opcode: SLTU
- Parameters: $(\mathsf{a\text{-}val}, \mathsf{b\text{-}val}, \mathsf{c\text{-}val})$
- Instruction selector: $\texttt{is-sltu} = 1$
- Functionality:
  - $\mathsf{a\text{-}val} \leftarrow 1$ if $\mathsf{b\text{-}val} < \mathsf{c\text{-}val}$ (*unsigned* comparison)
  - $\mathsf{a\text{-}val} \leftarrow 0$ if $\mathsf{b\text{-}val} \geq \mathsf{c\text{-}val}$ (*unsigned* comparison)

The mapping from the **sltu** and **sltiu** instructions in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
sltu	rd	rs1	rs2	0	R[rd]	R[rs1]	R[rs2]
sltiu	rd	rs1	rs2	1	R[rd]	R[rs1]	sext(i)

where sext is the sign extension function and i is a 12-bit immediate value.

Constraints assuming large fields

// Performing unsigned comparison between b-val and c-val using SUB borrow bit
- $(\texttt{is-sltu}) \cdot (\mathsf{b\text{-}val} + \texttt{h-borrow} \cdot 2^{32} - \mathsf{c\text{-}val} - \texttt{h-rem}) = 0$

// Enforcing $\texttt{h-borrow} \in \{0, 1\}$
- $(\texttt{is-sltu}) \cdot (\texttt{h-borrow}) \cdot (1 - \texttt{h-borrow}) = 0$

// Enforcing $\texttt{h-rem} \in \left[0, 2^{32} - 1\right]$
- $(\texttt{is-sltu}) \cdot (\texttt{h-rem} \in \left[0, 2^{32} - 1\right]) = 0$

// Setting $\mathsf{a\text{-}val} = \texttt{h-borrow}$
- $(\texttt{is-sltu}) \cdot (\texttt{h-borrow} - \mathsf{a\text{-}val}) = 0$

// Range check $\mathsf{a\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\mathsf{b\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\mathsf{c\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\texttt{is-sltu} \in \{0, 1\}$ - Performed in the CPU component

The set of constraints above implements the unsigned comparison between b-val and c-val using the borrow bit of the subtraction b-val − c-val. More precisely, if the result of subtracting c-val from b-val results in a borrow ($\texttt{h-borrow} = 1$ and $\texttt{h-rem} \equiv \mathsf{b\text{-}val} - \mathsf{c\text{-}val} \bmod 2^{32}$), then we know that $\mathsf{c\text{-}val} > \mathsf{b\text{-}val}$ and a-val must be equal to 1. If there is no borrow ($\texttt{h-borrow} = 0$ and $\texttt{h-rem} = \mathsf{b\text{-}val} - \mathsf{c\text{-}val}$), then $\mathsf{b\text{-}val} \geq \mathsf{c\text{-}val}$ and a-val must be equal to 1. Hence, it suffices to set $\mathsf{a\text{-}val} = \texttt{h-borrow}$.

Constraints assuming small fields

// Borrow handling
- $(\texttt{is-sltu}) \cdot (\mathsf{b\text{-}val}^{(1)} + \texttt{h-borrow}^{(1)} \cdot 2^8 - \mathsf{c\text{-}val}^{(1)} - \texttt{h-rem}^{(1)}) = 0$
- $(\texttt{is-sltu}) \cdot (\mathsf{b\text{-}val}^{(2)} + \texttt{h-borrow}^{(2)} \cdot 2^8 - \mathsf{c\text{-}val}^{(2)} - \texttt{h-rem}^{(2)} - \texttt{h-borrow}^{(1)}) = 0$
- $(\texttt{is-sltu}) \cdot (\mathsf{b\text{-}val}^{(3)} + \texttt{h-borrow}^{(3)} \cdot 2^8 - \mathsf{c\text{-}val}^{(3)} - \texttt{h-rem}^{(3)} - \texttt{h-borrow}^{(2)}) = 0$
- $(\texttt{is-sltu}) \cdot (\mathsf{b\text{-}val}^{(4)} + \texttt{h-borrow}^{(4)} \cdot 2^8 - \mathsf{c\text{-}val}^{(4)} - \texttt{h-rem}^{(4)} - \texttt{h-borrow}^{(3)}) = 0$

// Enforcing $\texttt{h-borrow}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$
- $(\texttt{is-sltu}) \cdot (\texttt{h-borrow}^{(1)}) \cdot (1 - \texttt{h-borrow}^{(1)}) = 0$
- $(\texttt{is-sltu}) \cdot (\texttt{h-borrow}^{(2)}) \cdot (1 - \texttt{h-borrow}^{(2)}) = 0$

- $(\texttt{is-sltu}) \cdot (\texttt{h-borrow}^{(3)}) \cdot (1 - \texttt{h-borrow}^{(3)}) = 0$
- $(\texttt{is-sltu}) \cdot (\texttt{h-borrow}^{(4)}) \cdot (1 - \texttt{h-borrow}^{(4)}) = 0$

// `Enforcing` $\texttt{h-rem}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$
- $(\texttt{is-sltu}) \cdot (\texttt{h-rem}^{(1)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-sltu}) \cdot (\texttt{h-rem}^{(2)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-sltu}) \cdot (\texttt{h-rem}^{(3)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-sltu}) \cdot (\texttt{h-rem}^{(4)} \in \left[0, 2^8 - 1\right]) = 0$

// `Setting` $\textsf{a-val}^{(1)} = \texttt{h-borrow}^{(4)}$
- $(\texttt{is-sltu}) \cdot (\texttt{h-borrow}^{(4)} - \textsf{a-val}^{(1)}) = 0$

// `Setting` $\textsf{a-val}^{(j)} = 0$ `for` $j = 2, 3, 4$
- $(\texttt{is-sltu}) \cdot (\textsf{a-val}^{(2)}) = 0$
- $(\texttt{is-sltu}) \cdot (\textsf{a-val}^{(3)}) = 0$
- $(\texttt{is-sltu}) \cdot (\textsf{a-val}^{(4)}) = 0$

// `Range check` $\textsf{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`

// `Range check` $\textsf{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`

// `Range check` $\textsf{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`

// `Range check` $\texttt{is-sltu} \in \{0, 1\}$ `- Performed in the CPU component`

The set of constraints above is similar to the large field case, implementing the unsigned comparison between b-val and c-val using the borrow bit of the subtraction b-val − c-val and setting the value of a-val accordingly. The main difference is that we only need to ensure that the first limb of a-val is equal to the last limb of the borrow bit `h-borrow` and that all the remaining limbs of a-val are 0.

More precisely, if the result of subtracting c-val from b-val limb by limb results in a borrow ($\texttt{h-borrow}^{(4)} = 1$), then we know that c-val > b-val and a-val must be equal to 1. If there is no borrow ($\texttt{h-borrow}^{(4)} = 0$), then b-val ≥ c-val and a-val must be equal to 1. Therefore, it suffices to set $\textsf{a-val}^{(1)} = \texttt{h-borrow}^{(4)}$ and $\textsf{a-val}^{(2)} = \textsf{a-val}^{(3)} = \textsf{a-val}^{(4)} = 0$.

8.5.4 `SLT` Instruction

The parameters and functionality for the `SLT` instruction are as follows:

- opcode: `SLT`
- Parameters: (a-val, b-val, c-val)
- Instruction selector: `is-slt` = 1
- Functionality:
  - a-val ← 1 if b-val < c-val (<u>*signed*</u> comparison)
  - a-val ← 0 if b-val ≥ c-val (<u>*signed*</u> comparison)

The mapping from the `slt` and `slti` instructions in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
`slt`	rd	rs1	rs2	0	R[rd]	R[rs1]	R[rs2]
`slti`	rd	rs1	rs2	1	R[rd]	R[rs1]	sext(i)

where sext is the sign extension function and i is a 12-bit immediate value.

Constraints assuming large fields

// `Performing unsigned comparison between` b−val `and` c−val `using SUB borrow bit`

- $(\texttt{is-slt}) \cdot (\text{b-val} - \text{c-val} + \texttt{h-borrow} \cdot 2^{32} - \texttt{h-rem}) = 0$

// Enforcing $\texttt{h-rem} \in \left[0, 2^{32} - 1\right]$
- $(\texttt{is-slt}) \cdot (\texttt{h-rem} \in \left[0, 2^{32} - 1\right]) = 0$

// Enforcing $\texttt{h-borrow} \in \{0, 1\}$
- $(\texttt{is-slt}) \cdot (\texttt{h-borrow}) \cdot (1 - \texttt{h-borrow}) = 0$

// Setting $\texttt{h-ltu-flag} = \texttt{h-borrow}$
- $(\texttt{is-slt}) \cdot (\texttt{h-borrow} - \texttt{h-ltu-flag}) = 0$

// Extracting sign bits from b-val and c-val
- $(\texttt{is-slt}) \cdot (\texttt{h-rem-b} + \texttt{h-sgn-b} \cdot 2^{31} - \text{b-val}) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-rem-c} + \texttt{h-sgn-c} \cdot 2^{31} - \text{c-val}) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-rem-b} \in \left[0, 2^{31} - 1\right]) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-rem-c} \in \left[0, 2^{31} - 1\right]) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-sgn-b}) \cdot (1 - \texttt{h-sgn-b}) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-sgn-c}) \cdot (1 - \texttt{h-sgn-c}) = 0$

// Computing a-val from h-ltu-flag and sign bits h-sgn-b and h-sgn-c
- $(\texttt{is-slt}) \cdot ((\texttt{h-sgn-b})(1 - \texttt{h-sgn-c}) + \texttt{h-ltu-flag}((\texttt{h-sgn-b})(\texttt{h-sgn-c}) + (1 - \texttt{h-sgn-b})(1 - \texttt{h-sgn-c})) - \text{a-val}) = 0$

// Range check $\text{a-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\text{b-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\text{c-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\texttt{is-slt} \in \{0, 1\}$ - Performed in the CPU component

The set of constraints above implements the signed comparison between b-val and c-val in three main steps. First, it computes the unsigned comparison between b-val and c-val and stores the result in h-ltu-flag. Second, it extracts the sign bits of b-val and c-val and stores these values into h-sgn-b and h-sgn-c. Finally, it sets the value of a-val based on the value of h-ltu-flag and the sign bits h-sgn-b and h-sgn-c. In particular, a-val $= 1$ whenever

1. $(\texttt{h-sgn-b}, \texttt{h-sgn-c}) = (1, 0)$ since b-val $<$ c-val;
2. $(\texttt{h-sgn-b}, \texttt{h-sgn-c}) = (0, 0)$ and $\texttt{h-ltu-flag} = 1$ since b-val and c-val are positive values in this case and a-val should match the value of h-ltu-flag
3. $(\texttt{h-sgn-b}, \texttt{h-sgn-c}) = (1, 1)$ and $\texttt{h-ltu-flag} = 1$ since b-val and c-val are negative values in this case and a-val should also match the value of h-ltu-flag.

Constraints assuming small fields

// Borrow handling
- $(\texttt{is-slt}) \cdot (\text{b-val}^{(1)} + \texttt{h-borrow}^{(1)} \cdot 2^8 - \text{c-val}^{(1)} - \texttt{h-rem}^{(1)}) = 0$
- $(\texttt{is-slt}) \cdot (\text{b-val}^{(2)} + \texttt{h-borrow}^{(2)} \cdot 2^8 - \text{c-val}^{(2)} - \texttt{h-rem}^{(2)} - \texttt{h-borrow}^{(1)}) = 0$
- $(\texttt{is-slt}) \cdot (\text{b-val}^{(3)} + \texttt{h-borrow}^{(3)} \cdot 2^8 - \text{c-val}^{(3)} - \texttt{h-rem}^{(3)} - \texttt{h-borrow}^{(2)}) = 0$
- $(\texttt{is-slt}) \cdot (\text{b-val}^{(4)} + \texttt{h-borrow}^{(4)} \cdot 2^8 - \text{c-val}^{(4)} - \texttt{h-rem}^{(4)} - \texttt{h-borrow}^{(3)}) = 0$

// Enforcing $\texttt{h-borrow}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$
- $(\texttt{is-slt}) \cdot (\texttt{h-borrow}^{(1)}) \cdot (1 - \texttt{h-borrow}^{(1)}) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-borrow}^{(2)}) \cdot (1 - \texttt{h-borrow}^{(2)}) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-borrow}^{(3)}) \cdot (1 - \texttt{h-borrow}^{(3)}) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-borrow}^{(4)}) \cdot (1 - \texttt{h-borrow}^{(4)}) = 0$

// Enforcing $\texttt{h-rem}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$

- $(\texttt{is-slt}) \cdot (\texttt{h-rem}^{(1)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-rem}^{(2)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-rem}^{(3)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-rem}^{(4)} \in \left[0, 2^8 - 1\right]) = 0$

// `Setting` $\texttt{h-ltu-flag} = \texttt{h-borrow}^{(4)}$
- $(\texttt{is-slt}) \cdot (\texttt{h-borrow}^{(4)} - \texttt{h-ltu-flag}) = 0$

// `Extracting sign bits from` b-val `and` c-val
- $(\texttt{is-slt}) \cdot (\texttt{h-rem-b} + \texttt{h-sgn-b} \cdot 2^7 - \mathsf{b\text{-}val}^{(4)}) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-rem-c} + \texttt{h-sgn-c} \cdot 2^7 - \mathsf{c\text{-}val}^{(4)}) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-rem-b} \in \left[0, 2^7 - 1\right]) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-rem-c} \in \left[0, 2^7 - 1\right]) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-sgn-b}) \cdot (1 - \texttt{h-sgn-b}) = 0$
- $(\texttt{is-slt}) \cdot (\texttt{h-sgn-c}) \cdot (1 - \texttt{h-sgn-c}) = 0$

// `Computing` $\mathsf{a\text{-}val}^{(1)}$ `from h-ltu-flag and sign bits h-sgn-b and h-sgn-c`
- $(\texttt{is-slt}) \cdot ((\texttt{h-sgn-b})(1 - \texttt{h-sgn-c}) + \texttt{h-ltu-flag}((\texttt{h-sgn-b})(\texttt{h-sgn-c}) + (1 - \texttt{h-sgn-b})(1 - \texttt{h-sgn-c})) - \mathsf{a\text{-}val}^{(1)}) = 0$

// `Setting` $\mathsf{a\text{-}val}^{(j)} = 0$ `for` $j = 2, 3, 4$
- $(\texttt{is-slt}) \cdot (\mathsf{a\text{-}val}^{(2)}) = 0$
- $(\texttt{is-slt}) \cdot (\mathsf{a\text{-}val}^{(3)}) = 0$
- $(\texttt{is-slt}) \cdot (\mathsf{a\text{-}val}^{(4)}) = 0$

// `Range check` $\mathsf{a\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check` $\mathsf{b\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check` $\mathsf{c\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check` $\texttt{is-slt} \in \{0, 1\}$ `- Performed in the CPU component`

The set of constraints above is similar to the large field case, first computing the unsigned comparison flag `h-ltu-flag` between b-val and c-val, then computing the sign bits `h-sgn-b` and `h-sgn-c` of b-val and c-val, and finally setting the value of a-val based on the value of `h-ltu-flag` and the sign bits `h-sgn-b` and `h-sgn-c`.

8.5.5 `SLL` Instruction

The parameters and functionality for the `SLL` instruction are as follows:

- opcode: `SLL`
- Parameters: (a-val, b-val, c-val)
- Instruction selector: $\texttt{is-sll} = 1$
- Functionality: $\mathsf{a\text{-}val} \leftarrow \mathsf{b\text{-}val} \ll (\mathsf{c\text{-}val}\ \&\ \texttt{0x0000001F})$

The mapping from the `sll` and `slli` instructions in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
`sll`	rd	rs1	rs2	0	R[rd]	R[rs1]	R[rs2]
`slli`	rd	rs1	rs2	1	R[rd]	R[rs1]	i

where i is a 5-bit immediate value.

Constraints assuming large fields

// Extracting shift bits from c-val
- $(\texttt{is-sll}) \cdot (\texttt{sh1} + \texttt{sh2} \cdot 2 + \texttt{sh3} \cdot 2^2 + \texttt{sh4} \cdot 2^3 + \texttt{sh5} \cdot 2^4 + \texttt{h-rem} \cdot 2^5 - \text{c-}\mathsf{val}) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{h-rem} \in \left[0, 2^{27} - 1\right]) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{sh1}) \cdot (1 - \texttt{sh1}) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{sh2}) \cdot (1 - \texttt{sh2}) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{sh3}) \cdot (1 - \texttt{sh3}) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{sh4}) \cdot (1 - \texttt{sh4}) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{sh5}) \cdot (1 - \texttt{sh5}) = 0$

// Computing auxiliary amount exp5 from shift bits to help with the left shift operation
- $(\texttt{is-sll}) \cdot ((\texttt{sh1} + 1) \cdot ((2^2 - 1)\texttt{sh2} + 1) \cdot ((2^4 - 1)\texttt{sh3} + 1) \cdot ((2^8 - 1)\texttt{sh4} + 1) \cdot ((2^{16} - 1)\texttt{sh5} + 1) - \texttt{exp5}) = 0$

// Performing the left shift and storing the result in rem1
- $(\texttt{is-sll}) \cdot (\texttt{rem1} + \texttt{qt1} \cdot 2^{32} - \text{b-}\mathsf{val} \cdot \texttt{exp5}) = 0$

// Range check qt1 $\in \left[0, 2^{32} - 1\right]$
- $(\texttt{is-sll}) \cdot (\texttt{qt1} \in \left[0, 2^{32} - 1\right]) = 0$

// Range check rem1 $\in [0, \texttt{exp5} - 1]$
- $(\texttt{is-sll}) \cdot (\texttt{exp5} - 1 - \texttt{rem1} - \texttt{rem1-aux}) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{rem1} \in \left[0, 2^{32} - 1\right]) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{rem1-aux} \in \left[0, 2^{32} - 1\right]) = 0$

// Setting rem1 $=$ a-val
- $(\texttt{is-sll}) \cdot (\texttt{rem1} - \text{a-}\mathsf{val}) = 0$

// Range check a-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check b-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check c-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check is-sll $\in \{0, 1\}$ - Performed in the CPU component

The set of constraints above implements the shift left operation in three steps. First, it extracts the shift bits $\texttt{sh1}, \ldots, \texttt{sh5}$ from c-val. Next, it computes an auxiliary variable exp5 from the shift bits so that $\texttt{exp5} = 2^{\text{shamt}}$ whenever the value b-val needs to be shifted by shamt bits. In particular, one can observe that $\text{shamt} = \texttt{sh1} + 2 \cdot \texttt{sh2} + 4 \cdot \texttt{sh3} + 8 \cdot \texttt{sh4} + 16 \cdot \texttt{sh5}$. Finally, the last constraint performs the shift operation by setting a-val to be the remainder of the division of b-val $\cdot$ exp5 by 2^{32}, which is equivalent to shifting left the contents of the 32-bit value representing b-val by shamt positions and filling the lower shamt bits with zeros.

To see why the last step computes the left shift operation correctly, let b-val $= \sum_{i=0}^{31} b_i \cdot 2^i$ where $(b_{31}, \ldots, b_0)$ corresponds to the bit representation of the field element b-val. Since $\texttt{exp5} = 2^{\text{shamt}}$, it follows that

$$\begin{aligned}
\text{b-}\mathsf{val} \cdot \texttt{exp5} &= \sum_{i=0}^{31} b_i \cdot 2^i \cdot 2^{\text{shamt}} = \sum_{i=0}^{31} b_i \cdot 2^{i+\text{shamt}} \\
&= \sum_{i=0}^{31-\text{shamt}} b_i \cdot 2^{i+\text{shamt}} + \sum_{i=31-\text{shamt}+1}^{31} b_i \cdot 2^{i+\text{shamt}} \\
&= \sum_{i=\text{shamt}}^{31} b_{i-\text{shamt}} \cdot 2^i + \sum_{i=32}^{31+\text{shamt}} b_{i-\text{shamt}} \cdot 2^i \\
&= \sum_{i=\text{shamt}}^{31} b_{i-\text{shamt}} \cdot 2^i + \sum_{i=0}^{\text{shamt}-1} b_{i+32-\text{shamt}} \cdot 2^{i+32} \\
&= \sum_{i=\text{shamt}}^{31} b_{i-\text{shamt}} \cdot 2^i + 2^{32} \cdot \sum_{j=0}^{\text{shamt}-1} b_{i+32-\text{shamt}} \cdot 2^i
\end{aligned}$$

As a result, the remainder of the division of b-$\mathsf{val} \cdot \texttt{exp5}$ by 2^{32} is equal to $\sum_{i=\text{shamt}}^{31} b_{i-\text{shamt}} \cdot 2^i$, which is equivalent to shifting left the contents of the 32-bit value representing b-val by shamt positions and filling the lower shamt bits with zeros, as desired.

Also note that the quotient of the division of b-$\mathsf{val} \cdot \texttt{exp5}$ by 2^{32} is $\sum_{j=0}^{\text{shamt}-1} b_{i+32-\text{shamt}} \cdot 2^i$ and corresponds to the bits b-val which will get discarded after the left shift operation.

Constraints assuming small fields

// `Extracting shift bits from c-val`$^{(1)}$

- $(\texttt{is-sll}) \cdot (\texttt{sh1} + \texttt{sh2} \cdot 2 + \texttt{sh3} \cdot 2^2 + \texttt{sh4} \cdot 2^3 + \texttt{sh5} \cdot 2^4 + \texttt{h-rem} \cdot 2^5 - \text{c-}\mathsf{val}^{(1)}) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{h-rem} \in \left[0, 2^3 - 1\right]) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{sh1}) \cdot (1 - \texttt{sh1}) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{sh2}) \cdot (1 - \texttt{sh2}) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{sh3}) \cdot (1 - \texttt{sh3}) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{sh4}) \cdot (1 - \texttt{sh4}) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{sh5}) \cdot (1 - \texttt{sh5}) = 0$

// `Computing auxiliary amount exp3 from shift bits sh1, sh2, sh3 for a partial left shift operation`

- $(\texttt{is-sll}) \cdot ((\texttt{sh1} + 1) \cdot ((2^2 - 1)\texttt{sh2} + 1) \cdot ((2^4 - 1)\texttt{sh3} + 1) - \texttt{exp3}) = 0$

// `Performing a partial left shift operation using shift bits sh1, sh2, sh3`

- $(\texttt{is-sll}) \cdot (\texttt{rem1} + \texttt{qt1} \cdot 2^8 - \text{b-}\mathsf{val}^{(1)} \cdot \texttt{exp3}) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{rem2} + \texttt{qt2} \cdot 2^8 - \texttt{qt1} - \text{b-}\mathsf{val}^{(2)} \cdot \texttt{exp3}) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{rem3} + \texttt{qt3} \cdot 2^8 - \texttt{qt2} - \text{b-}\mathsf{val}^{(3)} \cdot \texttt{exp3}) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{rem4} + \texttt{qt4} \cdot 2^8 - \texttt{qt3} - \text{b-}\mathsf{val}^{(4)} \cdot \texttt{exp3}) = 0$

// `Range check qt`$j \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$

- $(\texttt{is-sll}) \cdot (\texttt{qt1} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{qt2} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{qt3} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{qt4} \in \left[0, 2^8 - 1\right]) = 0$

// `Range check rem`$j \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$

- $(\texttt{is-sll}) \cdot (\texttt{rem1} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{rem2} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{rem3} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-sll}) \cdot (\texttt{rem4} \in \left[0, 2^8 - 1\right]) = 0$

// `Computing final left shift using remaining bits of the shift amount`
// $\texttt{sh4} = 1$ `implies an additional 1-byte left shift`
// $\texttt{sh5} = 1$ `implies an additional 2-byte left shift`

- $(\texttt{is-sll}) \cdot (\text{a-}\mathsf{val}^{(1)} - \texttt{rem1} \cdot (1 - \texttt{sh4}) \cdot (1 - \texttt{sh5})) = 0$
- $(\texttt{is-sll}) \cdot (\text{a-}\mathsf{val}^{(2)} - \texttt{rem2} \cdot (1 - \texttt{sh4}) \cdot (1 - \texttt{sh5}) - \texttt{rem1} \cdot (\texttt{sh4}) \cdot (1 - \texttt{sh5})) = 0$
- $(\texttt{is-sll}) \cdot (\text{a-}\mathsf{val}^{(3)} - \texttt{rem3} \cdot (1 - \texttt{sh4}) \cdot (1 - \texttt{sh5}) - \texttt{rem2} \cdot (\texttt{sh4}) \cdot (1 - \texttt{sh5}) - \texttt{rem1} \cdot (1 - \texttt{sh4}) \cdot (\texttt{sh5})) = 0$
- $(\texttt{is-sll}) \cdot (\text{a-}\mathsf{val}^{(4)} - \texttt{rem4} \cdot (1 - \texttt{sh4}) \cdot (1 - \texttt{sh5}) - \texttt{rem3} \cdot (\texttt{sh4}) \cdot (1 - \texttt{sh5}) - \texttt{rem2} \cdot (1 - \texttt{sh4}) \cdot (\texttt{sh5}) - \texttt{rem1} \cdot (\texttt{sh4}) \cdot (\texttt{sh5})) = 0$

// `Range check a-val`$^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check b-val`$^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check c-val`$^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check is-sll` $\in \{0, 1\}$ `- Performed in the CPU component`

The set of constraints above implements the shift left operation in three main stages.

In a first stage, we define an initial set of constraints to extract the shift bits $\texttt{sh1}, \ldots, \texttt{sh5}$ from c-val.

In a second stage, we define an additional set of constraints to perform a temporary left shift of b-val based on the values of the shift bits $\texttt{sh1}, \texttt{sh2}, \texttt{sh3}$. In this stage, the variable b-$\mathsf{val} :=$ (b-$\mathsf{val}^{(1)}$, $\ldots$, b-$\mathsf{val}^{(4)}$) is left shifted by shamt bits and the result is stored in the variables $(\texttt{rem1}, \ldots, \texttt{rem4})$, where shamt $= \texttt{sh1} + 2 \cdot \texttt{sh2} + 4 \cdot \texttt{sh3}$. This is done as follows:

- First, we define an auxiliary variable $\texttt{exp3}$ which enforces that $\texttt{exp3} = 2^{\text{shamt}}$, thus avoiding the need to explicitly define the variable shamt.
- Second, we compute the left shift of $\textsf{b-val}^{(1)}$ by shamt bits by setting $\texttt{rem1}$ to be the remainder of the division of $\textsf{b-val}^{(1)} \cdot \texttt{exp3}$ by 2^8. The quotient $\texttt{qt1}$ of this division is used to store the bits from $\textsf{b-val}^{(1)}$ that carry over into the second byte $\texttt{rem2}$.
- Third, we compute the left shift of $\textsf{b-val}^{(2)}$ by shamt bits by setting $\texttt{rem2}$ to be the remainder of the division of $\texttt{qt1} + \textsf{b-val}^{(2)} \cdot \texttt{exp3}$ by 2^8. The quotient $\texttt{qt2}$ of this division is used to store the bits from $\textsf{b-val}^{(2)}$ that carry over into the third byte $\texttt{rem3}$.
- Fourth, we compute the left shift of $\textsf{b-val}^{(3)}$ by shamt bits by setting $\texttt{rem3}$ to be the remainder of the division of $\texttt{qt2} + \textsf{b-val}^{(3)} \cdot \texttt{exp3}$ by 2^8. The quotient $\texttt{qt3}$ of this division is used to store the bits from $\textsf{b-val}^{(3)}$ that carry over into the fourth byte $\texttt{rem4}$.
- Fifth, we compute the left shift of $\textsf{b-val}^{(4)}$ by shamt bits by setting $\texttt{rem4}$ to be the remainder of the division of $\texttt{qt3} + \textsf{b-val}^{(4)} \cdot \texttt{exp3}$ by 2^8. The quotient $\texttt{qt4}$ of this division contains the bits from $\textsf{b-val}^{(4)}$ that gets discarded after this shamt-bit left shift.
- Finally, in order to guarantee that the divisions above are computed correctly, we implement range checks for $\texttt{qt}j$ and $\texttt{rem}j$ for $j = 1, 2, 3, 4$ that guarantee that these values lie in the proper range (i.e., $\texttt{rem}j \in [0, \texttt{exp3} - 1]$ and $\texttt{qt}j \in \left[0, 2^8 - 1\right]$). In particular, in order to guarantee that $\texttt{rem}j \in [0, \texttt{exp3} - 1]$, we also introduce auxiliary variables $\texttt{rem}j\texttt{-aux}$ for $j = 1, 2, 3, 4$ and range check these as well.

in a final stage, we use the values of the shift bits $\texttt{sh4}, \texttt{sh5}$ to complete the left shift operation and compute the final value of $\textsf{a-val} = (\textsf{a-val}^{(1)}, \ldots, \textsf{a-val}^{(4)})$ from the temporary values $(\texttt{rem1}, \ldots, \texttt{rem4})$. This is done as follows:

- if $(\texttt{sh4}, \texttt{sh5}) = (0, 0)$, then no additional left shift is needed and we simply set $\textsf{a-val}^{(j)} = \texttt{rem}j$ for $j = 1, 2, 3, 4$;
- if $(\texttt{sh4}, \texttt{sh5}) = (1, 0)$, then we need to left shift the temporary values $(\texttt{rem1}, \ldots, \texttt{rem4})$ by one byte. In this case, $\textsf{a-val}^{(1)} = 0$ and $\textsf{a-val}^{(j+1)} = \texttt{rem}j$ for $j = 1, 2, 3$;
- if $(\texttt{sh4}, \texttt{sh5}) = (0, 1)$, then we need to left shift the temporary values $(\texttt{rem1}, \ldots, \texttt{rem4})$ by two bytes. In this case, $\textsf{a-val}^{(1)} = \textsf{a-val}^{(2)} = 0$ and $\textsf{a-val}^{(j+2)} = \texttt{rem}j$ for $j = 1, 2$;
- if $(\texttt{sh4}, \texttt{sh5}) = (1, 1)$, then we need to left shift the temporary values $(\texttt{rem1}, \ldots, \texttt{rem4})$ by three bytes. In this case, $\textsf{a-val}^{(1)} = \textsf{a-val}^{(2)} = \textsf{a-val}^{(3)} = 0$ and $\textsf{a-val}^{(4)} = \texttt{rem1}$.

8.5.6 SRL Instruction

The parameters and functionality for the SRL instruction are as follows:

- opcode: SRL
- Parameters: $(\textsf{a-val}, \textsf{b-val}, \textsf{c-val})$
- Instruction selector: $\texttt{is-srl} = 1$
- Functionality: $\textsf{a-val} \leftarrow \textsf{b-val} \gg (\textsf{c-val} \,\&\, \texttt{0x0000001F})$

The mapping from the srl and srli instructions in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
srl	rd	rs1	rs2	0	R[rd]	R[rs1]	R[rs2]
srli	rd	rs1	rs2	1	R[rd]	R[rs1]	i

where i is a 5-bit immediate value.

Constraints assuming large fields

// Extracting shift bits from c-val

- $(\texttt{is-srl}) \cdot (\texttt{sh1} + \texttt{sh2} \cdot 2 + \texttt{sh3} \cdot 2^2 + \texttt{sh4} \cdot 2^3 + \texttt{sh5} \cdot 2^4 + \texttt{h-rem} \cdot 2^5 - \textsf{c-val}) = 0$

- $(\texttt{is-srl}) \cdot (\texttt{h-rem} \in \left[0, 2^{27} - 1\right]) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{sh1}) \cdot (1 - \texttt{sh1}) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{sh2}) \cdot (1 - \texttt{sh2}) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{sh3}) \cdot (1 - \texttt{sh3}) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{sh4}) \cdot (1 - \texttt{sh4}) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{sh5}) \cdot (1 - \texttt{sh5}) = 0$

// Computing auxiliary amount exp5 from shift bits to help with the right shift operation
- $(\texttt{is-srl}) \cdot ((\texttt{sh1} + 1) \cdot ((2^2 - 1)\texttt{sh2} + 1) \cdot ((2^4 - 1)\texttt{sh3} + 1) \cdot ((2^8 - 1)\texttt{sh4} + 1) \cdot ((2^{16} - 1)\texttt{sh5} + 1) - \texttt{exp5}) = 0$

// Performing the right shift and storing the result in qt1
- $(\texttt{is-srl}) \cdot (\text{b-val} - \texttt{rem1} - \texttt{qt1} \cdot \texttt{exp5}) = 0$

// Range check qt1 $\in \left[0, 2^{32} - 1\right]$
- $(\texttt{is-srl}) \cdot (\texttt{qt1} \in \left[0, 2^{32} - 1\right]) = 0$

// Range check rem1 $\in [0, \texttt{exp5} - 1]$
- $(\texttt{is-srl}) \cdot (\texttt{exp5} - 1 - \texttt{rem1} - \texttt{rem1-aux}) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{rem1} \in \left[0, 2^{32} - 1\right]) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{rem1-aux} \in \left[0, 2^{32} - 1\right]) = 0$

// Setting qt1 $= \text{a-val}$
- $(\texttt{is-srl}) \cdot (\texttt{qt1} - \text{a-val}) = 0$

// Range check a-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check b-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check c-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check is-srl $\in \{0, 1\}$ - Performed in the CPU component

The set of constraints above implements the shift right operation in three steps. First, it extracts the shift bits $\texttt{sh1}, \ldots, \texttt{sh5}$ from c-val. Next, it computes an auxiliary variable $\texttt{exp5}$ from the shift bits so that $\texttt{exp5} = 2^{\text{shamt}}$ whenever the value b-val needs to be shifted by shamt bits. Finally, the last constraint performs the shift right operation by setting a-val to be the quotient of the division of b-val by $\texttt{exp5}$.

Constraints assuming small fields

// Extracting shift bits from c-val
- $(\texttt{is-srl}) \cdot (\texttt{sh1} + \texttt{sh2} \cdot 2 + \texttt{sh3} \cdot 2^2 + \texttt{sh4} \cdot 2^3 + \texttt{sh5} \cdot 2^4 + \texttt{h-rem} \cdot 2^5 - \text{c-val}) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{h-rem} \in \left[0, 2^{27} - 1\right]) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{sh1}) \cdot (1 - \texttt{sh1}) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{sh2}) \cdot (1 - \texttt{sh2}) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{sh3}) \cdot (1 - \texttt{sh3}) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{sh4}) \cdot (1 - \texttt{sh4}) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{sh5}) \cdot (1 - \texttt{sh5}) = 0$

// Computing auxiliary amount exp3 from shift bits sh1, sh2, sh3 for a partial right shift operation
- $(\texttt{is-srl}) \cdot ((\texttt{sh1} + 1) \cdot ((2^2 - 1)\texttt{sh2} + 1) \cdot ((2^4 - 1)\texttt{sh3} + 1) - \texttt{exp3}) = 0$

// Performing a partial right shift operation using shift bits sh1, sh2, sh3
- $(\texttt{is-srl}) \cdot (\text{b-val}^{(4)} - \texttt{rem4} - \texttt{qt4} \cdot \texttt{exp3}) = 0$
- $(\texttt{is-srl}) \cdot (\text{b-val}^{(3)} + \texttt{rem4} \cdot 2^8 - \texttt{rem3} - \texttt{qt3} \cdot \texttt{exp3}) = 0$
- $(\texttt{is-srl}) \cdot (\text{b-val}^{(2)} + \texttt{rem3} \cdot 2^8 - \texttt{rem2} - \texttt{qt2} \cdot \texttt{exp3}) = 0$
- $(\texttt{is-srl}) \cdot (\text{b-val}^{(1)} + \texttt{rem2} \cdot 2^8 - \texttt{rem1} - \texttt{qt1} \cdot \texttt{exp3}) = 0$

// Range check qt$j \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $(\texttt{is-srl}) \cdot (\texttt{qt1} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{qt2} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{qt3} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{qt4} \in \left[0, 2^8 - 1\right]) = 0$

// Range check rem$j \in [0, \texttt{exp3} - 1]$ for $j = 1, 2, 3, 4$
- $(\texttt{is-srl}) \cdot (\texttt{exp3} - 1 - \texttt{rem1} - \texttt{rem1-aux}) = 0$

- $(\texttt{is-srl}) \cdot (\texttt{exp3} - 1 - \texttt{rem2} - \texttt{rem2-aux}) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{exp3} - 1 - \texttt{rem3} - \texttt{rem3-aux}) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{exp3} - 1 - \texttt{rem4} - \texttt{rem4-aux}) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{rem1} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{rem2} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{rem3} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{rem4} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{rem1-aux} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{rem2-aux} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{rem3-aux} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-srl}) \cdot (\texttt{rem4-aux} \in \left[0, 2^8 - 1\right]) = 0$

// `Computing final right shift using remaining bits of the shift amount`
// $\texttt{sh4} = 1$ `implies an additional 1-byte right shift`
// $\texttt{sh5} = 1$ `implies an additional 2-byte right shift`

- $(\texttt{is-srl}) \cdot (\text{a-val}^{(4)} - \texttt{qt4} \cdot (1 - \texttt{sh4}) \cdot (1 - \texttt{sh5})) = 0$
- $(\texttt{is-srl}) \cdot (\text{a-val}^{(3)} - \texttt{qt3} \cdot (1 - \texttt{sh4}) \cdot (1 - \texttt{sh5}) - \texttt{qt4} \cdot (\texttt{sh4}) \cdot (1 - \texttt{sh5})) = 0$
- $(\texttt{is-srl}) \cdot (\text{a-val}^{(2)} - \texttt{qt2} \cdot (1 - \texttt{sh4}) \cdot (1 - \texttt{sh5}) - \texttt{qt3} \cdot (\texttt{sh4}) \cdot (1 - \texttt{sh5}) - \texttt{qt4} \cdot (1 - \texttt{sh4}) \cdot (\texttt{sh5})) = 0$
- $(\texttt{is-srl}) \cdot (\text{a-val}^{(1)} - \texttt{qt1} \cdot (1 - \texttt{sh4}) \cdot (1 - \texttt{sh5}) - \texttt{qt2} \cdot (\texttt{sh4}) \cdot (1 - \texttt{sh5}) - \texttt{qt3} \cdot (1 - \texttt{sh4}) \cdot (\texttt{sh5}) - \texttt{qt4} \cdot (\texttt{sh4}) \cdot (\texttt{sh5})) = 0$

// `Range check` $\text{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check` $\text{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check` $\text{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check` $\texttt{is-srl} \in \{0, 1\}$ `- Performed in the CPU component`

The set of constraints above implements the shift right operation in three main stages.

In a first stage, we define an initial set of constraints to extract the shift bits $\texttt{sh1}, \ldots, \texttt{sh5}$ from c-val.

In a second stage, we define an additional set of constraints to perform a temporary right shift of b-val based on the values of the shift bits $\texttt{sh1}, \texttt{sh2}, \texttt{sh3}$. In this stage, the variable $\text{b-val} := (\text{b-val}^{(1)}, \ldots, \text{b-val}^{(4)})$ is shifted to the right by shamt bits and the result is stored in the variables $(\texttt{qt1}, \ldots, \texttt{qt4})$, where $\text{shamt} = \texttt{sh1} + 2 \cdot \texttt{sh2} + 4 \cdot \texttt{sh3}$. This is done as follows:

- First, we define an auxiliary variable $\texttt{exp3}$ which enforces that $\texttt{exp3} = 2^{\text{shamt}}$, thus avoiding the need to explicitly define the variable shamt.
- Second, we compute the right shift of $\text{b-val}^{(4)}$ by shamt bits by setting $\texttt{qt4}$ to be the quotient of the division of $\text{b-val}^{(4)}$ by $\texttt{exp3}$. The remainder $\texttt{rem4}$ of this division is used to store the bits from $\text{b-val}^{(4)}$ that carry over into the third byte $\texttt{qt3}$.
- Third, we compute the right shift of $\text{b-val}^{(3)}$ by shamt bits by setting $\texttt{qt3}$ to be the quotient of the division of $\texttt{rem4} \cdot 2^8 + \text{b-val}^{(3)}$ by $\texttt{exp3}$. The remainder $\texttt{rem3}$ of this division is used to store the bits from $\text{b-val}^{(3)}$ that carry over into the second byte $\texttt{qt2}$.
- Fourth, we compute the right shift of $\text{b-val}^{(2)}$ by shamt bits by setting $\texttt{qt2}$ to be the quotient of the division of $\texttt{rem3} \cdot 2^8 + \text{b-val}^{(2)}$ by $\texttt{exp3}$. The remainder $\texttt{rem2}$ of this division is used to store the bits from $\text{b-val}^{(2)}$ that carry over into the first byte $\texttt{qt1}$.
- Fifth, we compute the right shift of $\text{b-val}^{(1)}$ by shamt bits by setting $\texttt{qt1}$ to be the quotient of the division of $\texttt{rem2} \cdot 2^8 + \text{b-val}^{(1)}$ by $\texttt{exp3}$. The remainder $\texttt{rem1}$ of this division contains the bits from $\text{b-val}^{(1)}$ that gets discarded after this shamt-bit right shift.
- Finally, in order to guarantee that the divisions above are computed correctly, we implement range checks for $\texttt{qt}j$ and $\texttt{rem}j$ for $j = 1, 2, 3, 4$ that guarantee that these values lie in the proper range (i.e., $\texttt{rem}j \in [0, \texttt{exp3} - 1]$ and $\texttt{qt}j \in \left[0, 2^8 - 1\right]$). In particular, in order to guarantee that $\texttt{rem}j \in [0, \texttt{exp3} - 1]$, we also introduce auxiliary variables $\texttt{rem}j\texttt{-aux}$ for $j = 1, 2, 3, 4$ and range check these as well.

In a final stage, we use the values of the shift bits $\texttt{sh4}, \texttt{sh5}$ to complete the right shift operation and compute the final value of $\textsf{a-val} = (\textsf{a-val}^{(1)}, \ldots, \textsf{a-val}^{(4)})$ from the temporary values $(\texttt{qt1}, \ldots, \texttt{qt4})$. This is done as follows:

- if $(\texttt{sh4}, \texttt{sh5}) = (0, 0)$, then no additional right shift is needed and we simply set $\textsf{a-val}^{(j)} = \texttt{qt}j$ for $j = 1, 2, 3, 4$;
- if $(\texttt{sh4}, \texttt{sh5}) = (1, 0)$, then we need to right shift the temporary values $(\texttt{qt1}, \ldots, \texttt{qt4})$ by one byte. In this case, $\textsf{a-val}^{(4)} = 0$ and $\textsf{a-val}^{(j-1)} = \texttt{qt}j$ for $j = 2, 3, 4$;
- if $(\texttt{sh4}, \texttt{sh5}) = (0, 1)$, then we need to right shift the temporary values $(\texttt{qt1}, \ldots, \texttt{qt4})$ by two bytes. In this case, $\textsf{a-val}^{(4)} = \textsf{a-val}^{(3)} = 0$ and $\textsf{a-val}^{(j-2)} = \texttt{qt}j$ for $j = 3, 4$;
- if $(\texttt{sh4}, \texttt{sh5}) = (1, 1)$, then we need to right shift the temporary values $(\texttt{qt1}, \ldots, \texttt{qt4})$ by three bytes. In this case, $\textsf{a-val}^{(4)} = \textsf{a-val}^{(3)} == \textsf{a-val}^{(3)} = 0$ and $\textsf{a-val}^{(1)} = \texttt{qt4}$.

8.5.7 SRA Instruction

The parameters and functionality for the SRA instruction are as follows:

- opcode: SRA
- Parameters: $(\textsf{a-val}, \textsf{b-val}, \textsf{c-val})$
- Instruction selector: $\texttt{is-sra} = 1$
- Functionality: $\textsf{a-val} \leftarrow \textsf{b-val} \gg (\textsf{c-val} \,\&\, \textsf{0x0000001F})$ (*sign preserving*)
- Observation: Sign preserving means that vacated positions are filled with the sign bit and not necessarily with 0s as it was done in the SRL instruction Section 8.5.6.

The mapping from the sra and srai instructions in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
sra	rd	rs1	rs2	0	R[rd]	R[rs1]	R[rs2]
srai	rd	rs1	rs2	1	R[rd]	R[rs1]	i

where i is a 5-bit immediate value.

Constraints assuming large fields

// Extracting shift bits from c-val

- $(\texttt{is-sra}) \cdot (\texttt{sh1} + \texttt{sh2} \cdot 2 + \texttt{sh3} \cdot 2^2 + \texttt{sh4} \cdot 2^3 + \texttt{sh5} \cdot 2^4 + \texttt{h-rem} \cdot 2^5 - \textsf{c-val}) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{h-rem} \in \left[0, 2^{27} - 1\right]) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{sh1}) \cdot (1 - \texttt{sh1}) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{sh2}) \cdot (1 - \texttt{sh2}) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{sh3}) \cdot (1 - \texttt{sh3}) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{sh4}) \cdot (1 - \texttt{sh4}) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{sh5}) \cdot (1 - \texttt{sh5}) = 0$

// Extracting sign bit from b-val for arithmetic right shift

- $(\texttt{is-sra}) \cdot (\texttt{h-rem-b} + \texttt{h-sgn-b} \cdot 2^{31} - \textsf{b-val}) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{h-rem-b} \in \left[0, 2^{31} - 1\right]) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{h-sgn-b}) \cdot (1 - \texttt{h-sgn-b}) = 0$

// Computing auxiliary amounts exp5 and exp5-aux from shift bits
// exp5 and exp5-aux are used for logical and arithmetic right shift operations
// Note that $\texttt{exp5} \cdot \texttt{exp5-aux} = 2^{32}$

- $(\texttt{is-sra}) \cdot ((\texttt{sh1} + 1) \cdot ((2^2 - 1)\texttt{sh2} + 1) \cdot ((2^4 - 1)\texttt{sh3} + 1) \cdot ((2^8 - 1)\texttt{sh4} + 1) \cdot ((2^{16} - 1)\texttt{sh5} + 1) - \texttt{exp5}) = 0$
- $(\texttt{is-sra}) \cdot (2 \cdot (2 - \texttt{sh1}) \cdot (2^2 - (2^2 - 1)\texttt{sh2}) \cdot (2^4 - (2^4 - 1)\texttt{sh3}) \cdot (2^8 - (2^8 - 1)\texttt{sh4}) \cdot (2^{16} - (2^{16} - 1)\texttt{sh5}) - \texttt{exp5-aux}) = 0$

// Performing the logical right shift and storing the result in qt1

- $(\texttt{is-sra}) \cdot (\textsf{b-val} - \texttt{rem1} - \texttt{qt1} \cdot \texttt{exp5}) = 0$

// Range check $\texttt{qt1} \in \left[0, 2^{32} - 1\right]$

- $(\texttt{is-sra}) \cdot (\texttt{qt1} \in \left[0, 2^{32} - 1\right]) = 0$

// Range check $\texttt{rem1} \in [0, \texttt{exp5} - 1]$

- $(\texttt{is-sra}) \cdot (\texttt{exp5} - 1 - \texttt{rem1} - \texttt{rem1-aux}) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{rem1} \in \left[0, 2^{32} - 1\right]) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{rem1-aux} \in \left[0, 2^{32} - 1\right]) = 0$

// Computing arithmetic right shift from logical right shift
- $(\texttt{is-sra}) \cdot (\texttt{qt1} + \texttt{h-sgn-b} \cdot (\texttt{exp5} - 1) \cdot \texttt{exp5-aux} - \text{a-val}) = 0$

// Range check $\text{a-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\text{b-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\text{c-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\texttt{is-sra} \in \{0, 1\}$ - Performed in the CPU component

The set of constraints above is similar to those for the SRL instruction in Section 8.5.6. The main difference is that we need to additionally account for the sign bit $\texttt{h-sgn-b}$ of $\texttt{b-val}$ when performing the right shift operation. To achieve this goal, the set of constraints first stores the result of a logical right shift into $\texttt{qt1}$ and then replicates the sign bit $\texttt{h-sgn-b}$ into the vacated positions by adding $\texttt{h-sgn-b} \cdot (\texttt{exp5} - 1) \cdot \texttt{exp5-aux}$ to $\texttt{qt1}$.

To see why the last step works as desired, please note that, by construction, $\texttt{exp5} = 2^{\text{shamt}}$ and $\texttt{exp5} \cdot \texttt{exp5-aux} = 2^{32}$, where $\text{shamt} = \texttt{sh1} + 2 \cdot \texttt{sh2} + 4 \cdot \texttt{sh3} + 8 \cdot \texttt{sh4} + 16 \cdot \texttt{sh5}$ indicates the number of bits being right shifted. Hence, the field element $(\texttt{exp5} - 1) \cdot \texttt{exp5-aux}$ corresponds to the 32-bit string containing shamt one bits at the high-order bit positions and $32 - \text{shamt}$ zero bits at the low-order bit positions. Thus, by multiplying the latter quantity by $\texttt{h-sgn-b}$ and adding the result to $\texttt{qt1}$, we achieve the desired functionality of replicating the sign bit into the vacated positions.

Constraints assuming small fields

// Extracting shift bits from c-val
- $(\texttt{is-sra}) \cdot (\texttt{sh1} + \texttt{sh2} \cdot 2 + \texttt{sh3} \cdot 2^2 + \texttt{sh4} \cdot 2^3 + \texttt{sh5} \cdot 2^4 + \texttt{h-rem} \cdot 2^5 - \text{c-val}) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{h-rem} \in \left[0, 2^{27} - 1\right]) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{sh1}) \cdot (1 - \texttt{sh1}) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{sh2}) \cdot (1 - \texttt{sh2}) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{sh3}) \cdot (1 - \texttt{sh3}) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{sh4}) \cdot (1 - \texttt{sh4}) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{sh5}) \cdot (1 - \texttt{sh5}) = 0$

// Extracting sign bit from b-val for arithmetic right shift
- $(\texttt{is-sra}) \cdot (\texttt{h-rem-b} + \texttt{h-sgn-b} \cdot 2^7 - \text{b-val}^{(4)}) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{h-rem-b} \in \left[0, 2^7 - 1\right]) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{h-sgn-b}) \cdot (1 - \texttt{h-sgn-b}) = 0$

// Computing auxiliary amounts exp3 and exp3-aux from $\texttt{sh1}, \texttt{sh2}, \texttt{sh3}$

// exp3 and exp3-aux are used for partial logical and arithmetic right shift operations

// Note that $\texttt{exp3} \cdot \texttt{exp3-aux} = 2^8$
- $(\texttt{is-sra}) \cdot ((\texttt{sh1} + 1) \cdot ((2^2 - 1)\texttt{sh2} + 1) \cdot ((2^4 - 1)\texttt{sh3} + 1) - \texttt{exp3}) = 0$
- $(\texttt{is-sra}) \cdot (2 \cdot (2 - \texttt{sh1}) \cdot (2^2 - (2^2 - 1)\texttt{sh2}) \cdot (2^4 - (2^4 - 1)\texttt{sh3}) - \texttt{exp3-aux}) = 0$

// Performing a partial logical right shift operation using shift bits $\texttt{sh1}, \texttt{sh2}, \texttt{sh3}$
- $(\texttt{is-sra}) \cdot (\text{b-val}^{(4)} - \texttt{rem4} - \texttt{qt4} \cdot \texttt{exp3}) = 0$
- $(\texttt{is-sra}) \cdot (\text{b-val}^{(3)} + \texttt{rem4} \cdot 2^8 - \texttt{rem3} - \texttt{qt3} \cdot \texttt{exp3}) = 0$
- $(\texttt{is-sra}) \cdot (\text{b-val}^{(2)} + \texttt{rem3} \cdot 2^8 - \texttt{rem2} - \texttt{qt2} \cdot \texttt{exp3}) = 0$
- $(\texttt{is-sra}) \cdot (\text{b-val}^{(1)} + \texttt{rem2} \cdot 2^8 - \texttt{rem1} - \texttt{qt1} \cdot \texttt{exp3}) = 0$

// Range check $\texttt{qt}j \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $(\texttt{is-sra}) \cdot (\texttt{qt1} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{qt2} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{qt3} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-sra}) \cdot (\texttt{qt4} \in \left[0, 2^8 - 1\right]) = 0$

// Range check $\texttt{rem}j \in [0, \texttt{exp3} - 1]$ for $j = 1, 2, 3, 4$
- $(\texttt{is-sra}) \cdot (\texttt{exp3} - 1 - \texttt{rem1} - \texttt{rem1-aux}) = 0$

- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{exp3} - 1 - \mathtt{rem2} - \mathtt{rem2\text{-}aux}) = 0$
- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{exp3} - 1 - \mathtt{rem3} - \mathtt{rem3\text{-}aux}) = 0$
- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{exp3} - 1 - \mathtt{rem4} - \mathtt{rem4\text{-}aux}) = 0$
- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{rem1} \in \left[0, 2^8 - 1\right]) = 0$
- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{rem2} \in \left[0, 2^8 - 1\right]) = 0$
- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{rem3} \in \left[0, 2^8 - 1\right]) = 0$
- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{rem4} \in \left[0, 2^8 - 1\right]) = 0$
- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{rem1\text{-}aux} \in \left[0, 2^8 - 1\right]) = 0$
- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{rem2\text{-}aux} \in \left[0, 2^8 - 1\right]) = 0$
- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{rem3\text{-}aux} \in \left[0, 2^8 - 1\right]) = 0$
- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{rem4\text{-}aux} \in \left[0, 2^8 - 1\right]) = 0$

// `Computing final logical right shift using remaining bits of the shift amount`

// $\mathtt{sh4} = 1$ `implies an additional 1-byte logical right shift`

// $\mathtt{sh5} = 1$ `implies an additional 2-byte logical right shift`

- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{srl4} - \mathtt{qt4} \cdot (1 - \mathtt{sh4}) \cdot (1 - \mathtt{sh5})) = 0$
- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{srl3} - \mathtt{qt3} \cdot (1 - \mathtt{sh4}) \cdot (1 - \mathtt{sh5}) - \mathtt{qt4} \cdot (\mathtt{sh4}) \cdot (1 - \mathtt{sh5})) = 0$
- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{srl2} - \mathtt{qt2} \cdot (1 - \mathtt{sh4}) \cdot (1 - \mathtt{sh5}) - \mathtt{qt3} \cdot (\mathtt{sh4}) \cdot (1 - \mathtt{sh5}) - \mathtt{qt4} \cdot (1 - \mathtt{sh4}) \cdot (\mathtt{sh5})) = 0$
- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{srl1} - \mathtt{qt1} \cdot (1 - \mathtt{sh4}) \cdot (1 - \mathtt{sh5}) - \mathtt{qt2} \cdot (\mathtt{sh4}) \cdot (1 - \mathtt{sh5}) - \mathtt{qt3} \cdot (1 - \mathtt{sh4}) \cdot (\mathtt{sh5}) - \mathtt{qt4} \cdot (\mathtt{sh4}) \cdot (\mathtt{sh5})) = 0$

// `Computing auxiliary mask for the replication of the sign bit`

- $(\mathtt{is\text{-}sra}) \cdot (\mathtt{h\text{-}sgn\text{-}b} \cdot (\mathtt{exp3} - 1) \cdot \mathtt{exp3\text{-}aux} - \mathtt{sra\text{-}mask}) = 0$

// `Replicating sign bit into vacated positions during logical right shift`

// $\text{a-}\mathsf{val}^{(4)} = \mathtt{srl4} + \mathtt{h\text{-}sgn\text{-}b} \cdot (\mathtt{exp3} - 1) \cdot \mathtt{exp3\text{-}aux})$ `when additionally shifting 0 bytes (`$\mathtt{sh4} = \mathtt{sh5} = 0)$

// $\text{a-}\mathsf{val}^{(4)} = \mathtt{h\text{-}sgn\text{-}b} \cdot (2^8 - 1)$ `when additionally shifting 1, 2 or 3 bytes (`$\mathtt{sh4} + \mathtt{sh5} - \mathtt{sh4} \cdot \mathtt{sh5} = 1)$

- $(\mathtt{is\text{-}sra}) \cdot (\text{a-}\mathsf{val}^{(4)} - (1 - \mathtt{sh4}) \cdot (1 - \mathtt{sh5}) \cdot (\mathtt{srl4} + \mathtt{sra\text{-}mask}) - (\mathtt{sh4} + \mathtt{sh5} - \mathtt{sh4} \cdot \mathtt{sh5}) \cdot \mathtt{h\text{-}sgn\text{-}b} \cdot (2^8 - 1)) = 0$

// $\text{a-}\mathsf{val}^{(3)} = \mathtt{srl3}$ `when additionally shifting 0 bytes (`$\mathtt{sh4} = \mathtt{sh5} = 0)$

// $\text{a-}\mathsf{val}^{(3)} = \mathtt{srl3} + \mathtt{h\text{-}sgn\text{-}b} \cdot (\mathtt{exp3} - 1) \cdot \mathtt{exp3\text{-}aux})$ `when additionally shifting 1 byte (`$\mathtt{sh4} = 1, \mathtt{sh5} = 0)$

// $\text{a-}\mathsf{val}^{(3)} = \mathtt{h\text{-}sgn\text{-}b} \cdot (2^8 - 1)$ `when additionally shifting 2 or 3 bytes (`$\mathtt{sh5} = 1)$

- $(\mathtt{is\text{-}sra}) \cdot (\text{a-}\mathsf{val}^{(3)} - (1 - \mathtt{sh4}) \cdot (1 - \mathtt{sh5}) \cdot (\mathtt{srl3}) - (\mathtt{sh4}) \cdot (1 - \mathtt{sh5}) \cdot (\mathtt{srl3} + \mathtt{sra\text{-}mask}) - (\mathtt{sh5}) \cdot \mathtt{h\text{-}sgn\text{-}b} \cdot (2^8 - 1)) = 0$

// $\text{a-}\mathsf{val}^{(2)} = \mathtt{srl2}$ `when additionally shifting 0 or 1 bytes (`$\mathtt{sh5} = 0)$

// $\text{a-}\mathsf{val}^{(2)} = \mathtt{srl2} + \mathtt{h\text{-}sgn\text{-}b} \cdot (\mathtt{exp3} - 1) \cdot \mathtt{exp3\text{-}aux})$ `when additionally shifting 2 bytes (`$\mathtt{sh4} = 0, \mathtt{sh5} = 1)$

// $\text{a-}\mathsf{val}^{(2)} = \mathtt{h\text{-}sgn\text{-}b} \cdot (2^8 - 1)$ `when additionally shifting 3 bytes (`$\mathtt{sh4} = \mathtt{sh5} = 1)$

- $(\mathtt{is\text{-}sra}) \cdot (\text{a-}\mathsf{val}^{(2)} - (1 - \mathtt{sh5}) \cdot (\mathtt{srl2}) - (1 - \mathtt{sh4}) \cdot (\mathtt{sh5}) \cdot (\mathtt{srl2} + \mathtt{sra\text{-}mask}) - (\mathtt{sh4}) \cdot (\mathtt{sh5}) \cdot \mathtt{h\text{-}sgn\text{-}b} \cdot (2^8 - 1)) = 0$

// $\text{a-}\mathsf{val}^{(1)} = \mathtt{srl1}$ `when additionally shifting 0, 1, or 2 bytes (`$\mathtt{sh4} \cdot \mathtt{sh5} = 0)$

// $\text{a-}\mathsf{val}^{(1)} = \mathtt{srl1} + \mathtt{h\text{-}sgn\text{-}b} \cdot (\mathtt{exp3} - 1) \cdot \mathtt{exp3\text{-}aux})$ `when additionally shifting 3 bytes (`$\mathtt{sh4} = \mathtt{sh5} = 1)$

- $(\mathtt{is\text{-}sra}) \cdot (\text{a-}\mathsf{val}^{(1)} - (1 - \mathtt{sh4} \cdot \mathtt{sh5}) \cdot (\mathtt{srl1}) - (\mathtt{sh4}) \cdot (\mathtt{sh5}) \cdot (\mathtt{srl1} + \mathtt{sra\text{-}mask})) = 0$

// `Range check` $\text{a-}\mathsf{val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`

// `Range check` $\text{b-}\mathsf{val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`

// `Range check` $\text{c-}\mathsf{val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`

// `Range check` $\mathtt{is\text{-}sra} \in \{0, 1\}$ `- Performed in the CPU component`

The set of constraints above is similar to those for the SRL instruction in Section 8.5.6. As in the large field case described above, the main difference with respect to the SRL constraints is that we need to additionally account for the sign bit $\mathtt{h\text{-}sgn\text{-}b}$ of b-val when performing the right shift operation. To achieve this goal, the set of constraints first stores the result of a logical right shift into $(\mathtt{qt1}, \mathtt{qt2}, \mathtt{qt3}, \mathtt{qt4})$ and then replicates the sign bit $\mathtt{h\text{-}sgn\text{-}b}$ into the vacated positions.

To see why the last step works as desired, first note that, by construction, $\mathtt{exp3} = 2^{\text{shamt}}$ and $\mathtt{exp3} \cdot \mathtt{exp3\text{-}aux} = 2^8$, where shamt $= \mathtt{sh1} + 2 \cdot \mathtt{sh2} + 4 \cdot \mathtt{sh3}$ indicates the number of bits being temporarily right shifted. Hence, the field element $(\mathtt{exp3} - 1) \cdot \mathtt{exp3\text{-}aux}$ corresponds to the 8-bit string containing shamt one bits at the high-order bit positions and $8 -$ shamt zero bits at the low-order bit positions.

The final value of the arithmetic right shift operation is then obtained by adding $\mathtt{h\text{-}sgn\text{-}b} \cdot (\mathtt{exp3} - 1) \cdot \mathtt{exp3\text{-}aux}$ to the appropriate limb of a-val by considering the values of the shift bits $\mathtt{sh4}, \mathtt{sh5}$. More precisely,

- if $(\texttt{sh4}, \texttt{sh5}) = (0, 0)$, then the amount $\texttt{h-sgn-b} \cdot (\texttt{exp3} - 1) \cdot \texttt{exp3-aux}$ is added to $\texttt{srl4}$ when computing the value of the 4-th limb $\mathsf{a\text{-}val}^{(4)}$. No additional changes are needed for the other limbs and we simply set $\mathsf{a\text{-}val}^{(j)} = \texttt{qt}j$ for $j = 1, 2, 3$.
- if $(\texttt{sh4}, \texttt{sh5}) = (1, 0)$, then $\mathsf{a\text{-}val}^{(4)}$ is set to 255 (corresponding to the all-1s byte) and the amount $\texttt{h-sgn-b} \cdot (\texttt{exp3} - 1) \cdot \texttt{exp3-aux}$ is added to $\texttt{srl3}$ when computing the value of the 3-rd limb $\mathsf{a\text{-}val}^{(3)}$. No additional changes are needed for the other limbs and we simply set $\mathsf{a\text{-}val}^{(j)} = \texttt{qt}j$ for $j = 1, 2$.
- if $(\texttt{sh4}, \texttt{sh5}) = (0, 1)$, then $\mathsf{a\text{-}val}^{(3)}$ and $\mathsf{a\text{-}val}^{(4)}$ are set to 255 and the amount $\texttt{h-sgn-b} \cdot (\texttt{exp3} - 1) \cdot \texttt{exp3-aux}$ is added to $\texttt{srl2}$ when computing the value of the 2-nd limb $\mathsf{a\text{-}val}^{(2)}$. No additional changes are needed for the first limb and we simply set $\mathsf{a\text{-}val}^{(1)} = \texttt{qt1}$.
- if $(\texttt{sh4}, \texttt{sh5}) = (1, 1)$, then $\mathsf{a\text{-}val}^{(2)}$, $\mathsf{a\text{-}val}^{(3)}$, and $\mathsf{a\text{-}val}^{(4)}$ are set to 255 and the amount $\texttt{h-sgn-b} \cdot (\texttt{exp3} - 1) \cdot \texttt{exp3-aux}$ is added to $\texttt{srl1}$ when computing the value of the 1-st limb $\mathsf{a\text{-}val}^{(1)}$.

8.5.8 XOR Instruction

The parameters and functionality for the XOR instruction are as follows:

- opcode: XOR
- Parameters: (a-val, b-val, c-val)
- Instruction selector: $\texttt{is-xor} = 1$
- Functionality: a-val ← b-val ⊕ c-val

The mapping from the xor and xori instructions in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
xor	rd	rs1	rs2	0	R[rd]	R[rs1]	R[rs2]
xori	rd	rs1	rs2	1	R[rd]	R[rs1]	sext(i)

where sext is the sign extension function and i is a 12-bit immediate value.

Constraints assuming large fields

// Extracting 4-bit limbs from a-val, b-val, and c-val

- $(\texttt{is-xor}) \cdot (\texttt{a-val-low}^{(1)} + \texttt{a-val-high}^{(1)} \cdot 2^4 + \texttt{a-val-low}^{(2)} \cdot 2^8 + \texttt{a-val-high}^{(2)} \cdot 2^{12} + \texttt{a-val-low}^{(3)} \cdot 2^{16} + \texttt{a-val-high}^{(3)} \cdot 2^{20} + \texttt{a-val-low}^{(4)} \cdot 2^{24} + \texttt{a-val-high}^{(4)} \cdot 2^{28} - \mathsf{a\text{-}val}) = 0$
- $(\texttt{is-xor}) \cdot (\texttt{b-val-low}^{(1)} + \texttt{b-val-high}^{(1)} \cdot 2^4 + \texttt{b-val-low}^{(2)} \cdot 2^8 + \texttt{b-val-high}^{(2)} \cdot 2^{12} + \texttt{b-val-low}^{(3)} \cdot 2^{16} + \texttt{b-val-high}^{(3)} \cdot 2^{20} + \texttt{b-val-low}^{(4)} \cdot 2^{24} + \texttt{b-val-high}^{(4)} \cdot 2^{28} - \mathsf{b\text{-}val}) = 0$
- $(\texttt{is-xor}) \cdot (\texttt{c-val-low}^{(1)} + \texttt{c-val-high}^{(1)} \cdot 2^4 + \texttt{c-val-low}^{(2)} \cdot 2^8 + \texttt{c-val-high}^{(2)} \cdot 2^{12} + \texttt{c-val-low}^{(3)} \cdot 2^{16} + \texttt{c-val-high}^{(3)} \cdot 2^{20} + \texttt{c-val-low}^{(4)} \cdot 2^{24} + \texttt{c-val-high}^{(4)} \cdot 2^{28} - \mathsf{c\text{-}val}) = 0$

// Performing xor lookup queries

- $(\texttt{is-xor}) \cdot ((\texttt{a-val-low}^{(1)}, \texttt{b-val-low}^{(1)}, \texttt{c-val-low}^{(1)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$
- $(\texttt{is-xor}) \cdot ((\texttt{a-val-high}^{(1)}, \texttt{b-val-high}^{(1)}, \texttt{c-val-high}^{(1)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$
- $(\texttt{is-xor}) \cdot ((\texttt{a-val-low}^{(2)}, \texttt{b-val-low}^{(2)}, \texttt{c-val-low}^{(2)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$
- $(\texttt{is-xor}) \cdot ((\texttt{a-val-high}^{(2)}, \texttt{b-val-high}^{(2)}, \texttt{c-val-high}^{(2)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$
- $(\texttt{is-xor}) \cdot ((\texttt{a-val-low}^{(3)}, \texttt{b-val-low}^{(3)}, \texttt{c-val-low}^{(3)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$
- $(\texttt{is-xor}) \cdot ((\texttt{a-val-high}^{(3)}, \texttt{b-val-high}^{(3)}, \texttt{c-val-high}^{(3)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$
- $(\texttt{is-xor}) \cdot ((\texttt{a-val-low}^{(4)}, \texttt{b-val-low}^{(4)}, \texttt{c-val-low}^{(4)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$
- $(\texttt{is-xor}) \cdot ((\texttt{a-val-high}^{(4)}, \texttt{b-val-high}^{(4)}, \texttt{c-val-high}^{(4)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$

// Range check $\texttt{a-val-low}^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by xor lookup query

// Range check $\texttt{a-val-high}^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by xor lookup query

// Range check $\texttt{b-val-low}^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by xor lookup query

// Range check $\texttt{b-val-high}^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by xor lookup query

// Range check $\texttt{c-val-low}^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by xor lookup query

// Range check $\texttt{c-val-high}^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by xor lookup query

// Range check $\mathsf{a\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check b-val $\in \left[0, 2^{32}-1\right]$ - Performed in the CPU component

// Range check c-val $\in \left[0, 2^{32}-1\right]$ - Performed in the CPU component

// Range check is-xor $\in \{0,1\}$ - Performed in the CPU component

Constraints assuming small fields

// Extracting 4-bit limbs from a-val, b-val, and c-val

- $(\texttt{is-xor}) \cdot (\text{a-val-}\texttt{low}^{(1)} + \text{a-val-}\texttt{high}^{(1)} \cdot 2^4 - \text{a-val}^{(1)}) = 0$
- $(\texttt{is-xor}) \cdot (\text{a-val-}\texttt{low}^{(2)} + \text{a-val-}\texttt{high}^{(2)} \cdot 2^4 - \text{a-val}^{(2)}) = 0$
- $(\texttt{is-xor}) \cdot (\text{a-val-}\texttt{low}^{(3)} + \text{a-val-}\texttt{high}^{(3)} \cdot 2^4 - \text{a-val}^{(3)}) = 0$
- $(\texttt{is-xor}) \cdot (\text{a-val-}\texttt{low}^{(4)} + \text{a-val-}\texttt{high}^{(4)} \cdot 2^4 - \text{a-val}^{(4)}) = 0$
- $(\texttt{is-xor}) \cdot (\text{b-val-}\texttt{low}^{(1)} + \text{b-val-}\texttt{high}^{(1)} \cdot 2^4 - \text{b-val}^{(1)}) = 0$
- $(\texttt{is-xor}) \cdot (\text{b-val-}\texttt{low}^{(2)} + \text{b-val-}\texttt{high}^{(2)} \cdot 2^4 - \text{b-val}^{(2)}) = 0$
- $(\texttt{is-xor}) \cdot (\text{b-val-}\texttt{low}^{(3)} + \text{b-val-}\texttt{high}^{(3)} \cdot 2^4 - \text{b-val}^{(3)}) = 0$
- $(\texttt{is-xor}) \cdot (\text{b-val-}\texttt{low}^{(4)} + \text{b-val-}\texttt{high}^{(4)} \cdot 2^4 - \text{b-val}^{(4)}) = 0$
- $(\texttt{is-xor}) \cdot (\text{c-val-}\texttt{low}^{(1)} + \text{c-val-}\texttt{high}^{(1)} \cdot 2^4 - \text{c-val}^{(1)}) = 0$
- $(\texttt{is-xor}) \cdot (\text{c-val-}\texttt{low}^{(2)} + \text{c-val-}\texttt{high}^{(2)} \cdot 2^4 - \text{c-val}^{(2)}) = 0$
- $(\texttt{is-xor}) \cdot (\text{c-val-}\texttt{low}^{(3)} + \text{c-val-}\texttt{high}^{(3)} \cdot 2^4 - \text{c-val}^{(3)}) = 0$
- $(\texttt{is-xor}) \cdot (\text{c-val-}\texttt{low}^{(4)} + \text{c-val-}\texttt{high}^{(4)} \cdot 2^4 - \text{c-val}^{(4)}) = 0$

// Performing xor lookup queries

- $(\texttt{is-xor}) \cdot ((\text{a-val-}\texttt{low}^{(1)}, \text{b-val-}\texttt{low}^{(1)}, \text{c-val-}\texttt{low}^{(1)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$
- $(\texttt{is-xor}) \cdot ((\text{a-val-}\texttt{high}^{(1)}, \text{b-val-}\texttt{high}^{(1)}, \text{c-val-}\texttt{high}^{(1)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$
- $(\texttt{is-xor}) \cdot ((\text{a-val-}\texttt{low}^{(2)}, \text{b-val-}\texttt{low}^{(2)}, \text{c-val-}\texttt{low}^{(2)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$
- $(\texttt{is-xor}) \cdot ((\text{a-val-}\texttt{high}^{(2)}, \text{b-val-}\texttt{high}^{(2)}, \text{c-val-}\texttt{high}^{(2)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$
- $(\texttt{is-xor}) \cdot ((\text{a-val-}\texttt{low}^{(3)}, \text{b-val-}\texttt{low}^{(3)}, \text{c-val-}\texttt{low}^{(3)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$
- $(\texttt{is-xor}) \cdot ((\text{a-val-}\texttt{high}^{(3)}, \text{b-val-}\texttt{high}^{(3)}, \text{c-val-}\texttt{high}^{(3)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$
- $(\texttt{is-xor}) \cdot ((\text{a-val-}\texttt{low}^{(4)}, \text{b-val-}\texttt{low}^{(4)}, \text{c-val-}\texttt{low}^{(4)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$
- $(\texttt{is-xor}) \cdot ((\text{a-val-}\texttt{high}^{(4)}, \text{b-val-}\texttt{high}^{(4)}, \text{c-val-}\texttt{high}^{(4)}) \in \texttt{lookup}_{\texttt{xor}}) = 0$

// Range check a-val-low$^{(j)} \in \left[0, 2^4-1\right]$ for $j = 1,2,3,4$ - Implied by xor lookup query

// Range check a-val-high$^{(j)} \in \left[0, 2^4-1\right]$ for $j = 1,2,3,4$ - Implied by xor lookup query

// Range check b-val-low$^{(j)} \in \left[0, 2^4-1\right]$ for $j = 1,2,3,4$ - Implied by xor lookup query

// Range check b-val-high$^{(j)} \in \left[0, 2^4-1\right]$ for $j = 1,2,3,4$ - Implied by xor lookup query

// Range check c-val-low$^{(j)} \in \left[0, 2^4-1\right]$ for $j = 1,2,3,4$ - Implied by xor lookup query

// Range check c-val-high$^{(j)} \in \left[0, 2^4-1\right]$ for $j = 1,2,3,4$ - Implied by xor lookup query

// Range check a-val$^{(j)} \in \left[0, 2^8-1\right]$ for $j = 1,2,3,4$ - Performed in the CPU component

// Range check b-val$^{(j)} \in \left[0, 2^8-1\right]$ for $j = 1,2,3,4$ - Performed in the CPU component

// Range check c-val$^{(j)} \in \left[0, 2^8-1\right]$ for $j = 1,2,3,4$ - Performed in the CPU component

// Range check is-xor $\in \{0,1\}$ - Performed in the CPU component

8.5.9 AND Instruction

The parameters and functionality for the AND instruction are as follows:

- opcode: AND
- Parameters: (a-val, b-val, c-val)
- Instruction selector: is-and $= 1$
- Functionality: a-val $\leftarrow$ b-val & c-val

The mapping from the and and andi instructions in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
and	rd	rs1	rs2	0	R[rd]	R[rs1]	R[rs2]
andi	rd	rs1	rs2	1	R[rd]	R[rs1]	sext(i)

where sext is the sign extension function and i is a 12-bit immediate value.

Constraints assuming large fields

// Extracting 4-bit limbs from a-val, b-val, and c-val

- $(\texttt{is-and})\cdot(\texttt{a-val-low}^{(1)} + \texttt{a-val-high}^{(1)}\cdot 2^4 + \texttt{a-val-low}^{(2)}\cdot 2^8 + \texttt{a-val-high}^{(2)}\cdot 2^{12} + \texttt{a-val-low}^{(3)}\cdot 2^{16} + \texttt{a-val-high}^{(3)}\cdot 2^{20} + \texttt{a-val-low}^{(4)}\cdot 2^{24} + \texttt{a-val-high}^{(4)}\cdot 2^{28} - \text{a-val}) = 0$
- $(\texttt{is-and})\cdot(\texttt{b-val-low}^{(1)} + \texttt{b-val-high}^{(1)}\cdot 2^4 + \texttt{b-val-low}^{(2)}\cdot 2^8 + \texttt{b-val-high}^{(2)}\cdot 2^{12} + \texttt{b-val-low}^{(3)}\cdot 2^{16} + \texttt{b-val-high}^{(3)}\cdot 2^{20} + \texttt{b-val-low}^{(4)}\cdot 2^{24} + \texttt{b-val-high}^{(4)}\cdot 2^{28} - \text{b-val}) = 0$
- $(\texttt{is-and})\cdot(\texttt{c-val-low}^{(1)} + \texttt{c-val-high}^{(1)}\cdot 2^4 + \texttt{c-val-low}^{(2)}\cdot 2^8 + \texttt{c-val-high}^{(2)}\cdot 2^{12} + \texttt{c-val-low}^{(3)}\cdot 2^{16} + \texttt{c-val-high}^{(3)}\cdot 2^{20} + \texttt{c-val-low}^{(4)}\cdot 2^{24} + \texttt{c-val-high}^{(4)}\cdot 2^{28} - \text{c-val}) = 0$

// Performing and lookup queries

- $(\texttt{is-and})\cdot((\texttt{a-val-low}^{(1)}, \texttt{b-val-low}^{(1)}, \texttt{c-val-low}^{(1)}) \in \texttt{lookup}_{\texttt{and}}) = 0$
- $(\texttt{is-and})\cdot((\texttt{a-val-high}^{(1)}, \texttt{b-val-high}^{(1)}, \texttt{c-val-high}^{(1)}) \in \texttt{lookup}_{\texttt{and}}) = 0$
- $(\texttt{is-and})\cdot((\texttt{a-val-low}^{(2)}, \texttt{b-val-low}^{(2)}, \texttt{c-val-low}^{(2)}) \in \texttt{lookup}_{\texttt{and}}) = 0$
- $(\texttt{is-and})\cdot((\texttt{a-val-high}^{(2)}, \texttt{b-val-high}^{(2)}, \texttt{c-val-high}^{(2)}) \in \texttt{lookup}_{\texttt{and}}) = 0$
- $(\texttt{is-and})\cdot((\texttt{a-val-low}^{(3)}, \texttt{b-val-low}^{(3)}, \texttt{c-val-low}^{(3)}) \in \texttt{lookup}_{\texttt{and}}) = 0$
- $(\texttt{is-and})\cdot((\texttt{a-val-high}^{(3)}, \texttt{b-val-high}^{(3)}, \texttt{c-val-high}^{(3)}) \in \texttt{lookup}_{\texttt{and}}) = 0$
- $(\texttt{is-and})\cdot((\texttt{a-val-low}^{(4)}, \texttt{b-val-low}^{(4)}, \texttt{c-val-low}^{(4)}) \in \texttt{lookup}_{\texttt{and}}) = 0$
- $(\texttt{is-and})\cdot((\texttt{a-val-high}^{(4)}, \texttt{b-val-high}^{(4)}, \texttt{c-val-high}^{(4)}) \in \texttt{lookup}_{\texttt{and}}) = 0$

// Range check $\texttt{a-val-low}^{(j)} \in \left[0, 2^4-1\right]$ for $j = 1,2,3,4$ - Implied by and lookup query

// Range check $\texttt{a-val-high}^{(j)} \in \left[0, 2^4-1\right]$ for $j = 1,2,3,4$ - Implied by and lookup query

// Range check $\texttt{b-val-low}^{(j)} \in \left[0, 2^4-1\right]$ for $j = 1,2,3,4$ - Implied by and lookup query

// Range check $\texttt{b-val-high}^{(j)} \in \left[0, 2^4-1\right]$ for $j = 1,2,3,4$ - Implied by and lookup query

// Range check $\texttt{c-val-low}^{(j)} \in \left[0, 2^4-1\right]$ for $j = 1,2,3,4$ - Implied by and lookup query

// Range check $\texttt{c-val-high}^{(j)} \in \left[0, 2^4-1\right]$ for $j = 1,2,3,4$ - Implied by and lookup query

// Range check $\text{a-val} \in \left[0, 2^{32}-1\right]$ - Performed in the CPU component

// Range check $\text{b-val} \in \left[0, 2^{32}-1\right]$ - Performed in the CPU component

// Range check $\text{c-val} \in \left[0, 2^{32}-1\right]$ - Performed in the CPU component

// Range check is-and $\in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Extracting 4-bit limbs from a-val, b-val, and c-val

- $(\texttt{is-and})\cdot(\texttt{a-val-low}^{(1)} + \texttt{a-val-high}^{(1)}\cdot 2^4 - \texttt{a-val}^{(1)}) = 0$
- $(\texttt{is-and})\cdot(\texttt{a-val-low}^{(2)} + \texttt{a-val-high}^{(2)}\cdot 2^4 - \texttt{a-val}^{(2)}) = 0$
- $(\texttt{is-and})\cdot(\texttt{a-val-low}^{(3)} + \texttt{a-val-high}^{(3)}\cdot 2^4 - \texttt{a-val}^{(3)}) = 0$
- $(\texttt{is-and})\cdot(\texttt{a-val-low}^{(4)} + \texttt{a-val-high}^{(4)}\cdot 2^4 - \texttt{a-val}^{(4)}) = 0$
- $(\texttt{is-and})\cdot(\texttt{b-val-low}^{(1)} + \texttt{b-val-high}^{(1)}\cdot 2^4 - \texttt{b-val}^{(1)}) = 0$
- $(\texttt{is-and})\cdot(\texttt{b-val-low}^{(2)} + \texttt{b-val-high}^{(2)}\cdot 2^4 - \texttt{b-val}^{(2)}) = 0$
- $(\texttt{is-and})\cdot(\texttt{b-val-low}^{(3)} + \texttt{b-val-high}^{(3)}\cdot 2^4 - \texttt{b-val}^{(3)}) = 0$
- $(\texttt{is-and})\cdot(\texttt{b-val-low}^{(4)} + \texttt{b-val-high}^{(4)}\cdot 2^4 - \texttt{b-val}^{(4)}) = 0$
- $(\texttt{is-and})\cdot(\texttt{c-val-low}^{(1)} + \texttt{c-val-high}^{(1)}\cdot 2^4 - \texttt{c-val}^{(1)}) = 0$
- $(\texttt{is-and})\cdot(\texttt{c-val-low}^{(2)} + \texttt{c-val-high}^{(2)}\cdot 2^4 - \texttt{c-val}^{(2)}) = 0$
- $(\texttt{is-and})\cdot(\texttt{c-val-low}^{(3)} + \texttt{c-val-high}^{(3)}\cdot 2^4 - \texttt{c-val}^{(3)}) = 0$
- $(\texttt{is-and})\cdot(\texttt{c-val-low}^{(4)} + \texttt{c-val-high}^{(4)}\cdot 2^4 - \texttt{c-val}^{(4)}) = 0$

// Performing and lookup queries

- $(\texttt{is-and})\cdot((\texttt{a-val-low}^{(1)}, \texttt{b-val-low}^{(1)}, \texttt{c-val-low}^{(1)}) \in \texttt{lookup}_{\texttt{and}}) = 0$
- $(\texttt{is-and})\cdot((\texttt{a-val-high}^{(1)}, \texttt{b-val-high}^{(1)}, \texttt{c-val-high}^{(1)}) \in \texttt{lookup}_{\texttt{and}}) = 0$
- $(\texttt{is-and})\cdot((\texttt{a-val-low}^{(2)}, \texttt{b-val-low}^{(2)}, \texttt{c-val-low}^{(2)}) \in \texttt{lookup}_{\texttt{and}}) = 0$
- $(\texttt{is-and})\cdot((\texttt{a-val-high}^{(2)}, \texttt{b-val-high}^{(2)}, \texttt{c-val-high}^{(2)}) \in \texttt{lookup}_{\texttt{and}}) = 0$
- $(\texttt{is-and})\cdot((\texttt{a-val-low}^{(3)}, \texttt{b-val-low}^{(3)}, \texttt{c-val-low}^{(3)}) \in \texttt{lookup}_{\texttt{and}}) = 0$
- $(\texttt{is-and})\cdot((\texttt{a-val-high}^{(3)}, \texttt{b-val-high}^{(3)}, \texttt{c-val-high}^{(3)}) \in \texttt{lookup}_{\texttt{and}}) = 0$
- $(\texttt{is-and})\cdot((\texttt{a-val-low}^{(4)}, \texttt{b-val-low}^{(4)}, \texttt{c-val-low}^{(4)}) \in \texttt{lookup}_{\texttt{and}}) = 0$

- $(\texttt{is-and}) \cdot ((\texttt{a-val-high}^{(4)}, \texttt{b-val-high}^{(4)}, \texttt{c-val-high}^{(4)}) \in \texttt{lookup}_{\texttt{and}}) = 0$

// Range check a-val-low$^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by and lookup query
// Range check a-val-high$^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by and lookup query
// Range check b-val-low$^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by and lookup query
// Range check b-val-high$^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by and lookup query
// Range check c-val-low$^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by and lookup query
// Range check c-val-high$^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by and lookup query

// Range check a-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check b-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check c-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check is-and $\in \{0, 1\}$ - Performed in the CPU component

8.5.10 OR Instruction

The parameters and functionality for the OR instruction are as follows:

- opcode: OR
- Parameters: (a-val, b-val, c-val)
- Instruction selector: is-or $= 1$
- Functionality: a-val $=$ b-val $|$ c-val

The mapping from the or and ori instructions in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
or	rd	rs1	rs2	0	R[rd]	R[rs1]	R[rs2]
ori	rd	rs1	rs2	1	R[rd]	R[rs1]	sext(i)

where sext is the sign extension function and i is a 12-bit immediate value.

Constraints assuming large fields

// Extracting 4-bit limbs from a-val, b-val, and c-val

- $$\begin{aligned}(\texttt{is-or}) \cdot (&\texttt{a-val-low}^{(1)} + \texttt{a-val-high}^{(1)} \cdot 2^4 + \texttt{a-val-low}^{(2)} \cdot 2^8 + \texttt{a-val-high}^{(2)} \cdot 2^{12} + \\ &\texttt{a-val-low}^{(3)} \cdot 2^{16} + \texttt{a-val-high}^{(3)} \cdot 2^{20} + \texttt{a-val-low}^{(4)} \cdot 2^{24} + \texttt{a-val-high}^{(4)} \cdot 2^{28} - \text{a-val}) = 0\end{aligned}$$
- $$\begin{aligned}(\texttt{is-or}) \cdot (&\texttt{b-val-low}^{(1)} + \texttt{b-val-high}^{(1)} \cdot 2^4 + \texttt{b-val-low}^{(2)} \cdot 2^8 + \texttt{b-val-high}^{(2)} \cdot 2^{12} + \\ &\texttt{b-val-low}^{(3)} \cdot 2^{16} + \texttt{b-val-high}^{(3)} \cdot 2^{20} + \texttt{b-val-low}^{(4)} \cdot 2^{24} + \texttt{b-val-high}^{(4)} \cdot 2^{28} - \text{b-val}) = 0\end{aligned}$$
- $$\begin{aligned}(\texttt{is-or}) \cdot (&\texttt{c-val-low}^{(1)} + \texttt{c-val-high}^{(1)} \cdot 2^4 + \texttt{c-val-low}^{(2)} \cdot 2^8 + \texttt{c-val-high}^{(2)} \cdot 2^{12} + \\ &\texttt{c-val-low}^{(3)} \cdot 2^{16} + \texttt{c-val-high}^{(3)} \cdot 2^{20} + \texttt{c-val-low}^{(4)} \cdot 2^{24} + \texttt{c-val-high}^{(4)} \cdot 2^{28} - \text{c-val}) = 0\end{aligned}$$

// Performing or lookup queries

- $(\texttt{is-or}) \cdot ((\texttt{a-val-low}^{(1)}, \texttt{b-val-low}^{(1)}, \texttt{c-val-low}^{(1)}) \in \texttt{lookup}_{\texttt{or}}) = 0$
- $(\texttt{is-or}) \cdot ((\texttt{a-val-high}^{(1)}, \texttt{b-val-high}^{(1)}, \texttt{c-val-high}^{(1)}) \in \texttt{lookup}_{\texttt{or}}) = 0$
- $(\texttt{is-or}) \cdot ((\texttt{a-val-low}^{(2)}, \texttt{b-val-low}^{(2)}, \texttt{c-val-low}^{(2)}) \in \texttt{lookup}_{\texttt{or}}) = 0$
- $(\texttt{is-or}) \cdot ((\texttt{a-val-high}^{(2)}, \texttt{b-val-high}^{(2)}, \texttt{c-val-high}^{(2)}) \in \texttt{lookup}_{\texttt{or}}) = 0$
- $(\texttt{is-or}) \cdot ((\texttt{a-val-low}^{(3)}, \texttt{b-val-low}^{(3)}, \texttt{c-val-low}^{(3)}) \in \texttt{lookup}_{\texttt{or}}) = 0$
- $(\texttt{is-or}) \cdot ((\texttt{a-val-high}^{(3)}, \texttt{b-val-high}^{(3)}, \texttt{c-val-high}^{(3)}) \in \texttt{lookup}_{\texttt{or}}) = 0$
- $(\texttt{is-or}) \cdot ((\texttt{a-val-low}^{(4)}, \texttt{b-val-low}^{(4)}, \texttt{c-val-low}^{(4)}) \in \texttt{lookup}_{\texttt{or}}) = 0$
- $(\texttt{is-or}) \cdot ((\texttt{a-val-high}^{(4)}, \texttt{b-val-high}^{(4)}, \texttt{c-val-high}^{(4)}) \in \texttt{lookup}_{\texttt{or}}) = 0$

// Range check a-val-low$^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by or lookup query
// Range check a-val-high$^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by or lookup query
// Range check b-val-low$^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by or lookup query
// Range check b-val-high$^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by or lookup query
// Range check c-val-low$^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by or lookup query
// Range check c-val-high$^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by or lookup query

// Range check $\text{a-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\text{b-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\text{c-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\texttt{is-or} \in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Extracting 4-bit limbs from a-val, b-val, and c-val

- $(\texttt{is-or}) \cdot (\text{a-val-low}^{(1)} + \text{a-val-high}^{(1)} \cdot 2^4 - \text{a-val}^{(1)}) = 0$
- $(\texttt{is-or}) \cdot (\text{a-val-low}^{(2)} + \text{a-val-high}^{(2)} \cdot 2^4 - \text{a-val}^{(2)}) = 0$
- $(\texttt{is-or}) \cdot (\text{a-val-low}^{(3)} + \text{a-val-high}^{(3)} \cdot 2^4 - \text{a-val}^{(3)}) = 0$
- $(\texttt{is-or}) \cdot (\text{a-val-low}^{(4)} + \text{a-val-high}^{(4)} \cdot 2^4 - \text{a-val}^{(4)}) = 0$
- $(\texttt{is-or}) \cdot (\text{b-val-low}^{(1)} + \text{b-val-high}^{(1)} \cdot 2^4 - \text{b-val}^{(1)}) = 0$
- $(\texttt{is-or}) \cdot (\text{b-val-low}^{(2)} + \text{b-val-high}^{(2)} \cdot 2^4 - \text{b-val}^{(2)}) = 0$
- $(\texttt{is-or}) \cdot (\text{b-val-low}^{(3)} + \text{b-val-high}^{(3)} \cdot 2^4 - \text{b-val}^{(3)}) = 0$
- $(\texttt{is-or}) \cdot (\text{b-val-low}^{(4)} + \text{b-val-high}^{(4)} \cdot 2^4 - \text{b-val}^{(4)}) = 0$
- $(\texttt{is-or}) \cdot (\text{c-val-low}^{(1)} + \text{c-val-high}^{(1)} \cdot 2^4 - \text{c-val}^{(1)}) = 0$
- $(\texttt{is-or}) \cdot (\text{c-val-low}^{(2)} + \text{c-val-high}^{(2)} \cdot 2^4 - \text{c-val}^{(2)}) = 0$
- $(\texttt{is-or}) \cdot (\text{c-val-low}^{(3)} + \text{c-val-high}^{(3)} \cdot 2^4 - \text{c-val}^{(3)}) = 0$
- $(\texttt{is-or}) \cdot (\text{c-val-low}^{(4)} + \text{c-val-high}^{(4)} \cdot 2^4 - \text{c-val}^{(4)}) = 0$

// Performing or lookup queries

- $(\texttt{is-or}) \cdot ((\text{a-val-low}^{(1)}, \text{b-val-low}^{(1)}, \text{c-val-low}^{(1)}) \in \texttt{lookup}_{\texttt{or}}) = 0$
- $(\texttt{is-or}) \cdot ((\text{a-val-high}^{(1)}, \text{b-val-high}^{(1)}, \text{c-val-high}^{(1)}) \in \texttt{lookup}_{\texttt{or}}) = 0$
- $(\texttt{is-or}) \cdot ((\text{a-val-low}^{(2)}, \text{b-val-low}^{(2)}, \text{c-val-low}^{(2)}) \in \texttt{lookup}_{\texttt{or}}) = 0$
- $(\texttt{is-or}) \cdot ((\text{a-val-high}^{(2)}, \text{b-val-high}^{(2)}, \text{c-val-high}^{(2)}) \in \texttt{lookup}_{\texttt{or}}) = 0$
- $(\texttt{is-or}) \cdot ((\text{a-val-low}^{(3)}, \text{b-val-low}^{(3)}, \text{c-val-low}^{(3)}) \in \texttt{lookup}_{\texttt{or}}) = 0$
- $(\texttt{is-or}) \cdot ((\text{a-val-high}^{(3)}, \text{b-val-high}^{(3)}, \text{c-val-high}^{(3)}) \in \texttt{lookup}_{\texttt{or}}) = 0$
- $(\texttt{is-or}) \cdot ((\text{a-val-low}^{(4)}, \text{b-val-low}^{(4)}, \text{c-val-low}^{(4)}) \in \texttt{lookup}_{\texttt{or}}) = 0$
- $(\texttt{is-or}) \cdot ((\text{a-val-high}^{(4)}, \text{b-val-high}^{(4)}, \text{c-val-high}^{(4)}) \in \texttt{lookup}_{\texttt{or}}) = 0$

// Range check $\text{a-val-low}^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by or lookup query
// Range check $\text{a-val-high}^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by or lookup query
// Range check $\text{b-val-low}^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by or lookup query
// Range check $\text{b-val-high}^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by or lookup query
// Range check $\text{c-val-low}^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by or lookup query
// Range check $\text{c-val-high}^{(j)} \in \left[0, 2^4 - 1\right]$ for $j = 1, 2, 3, 4$ - Implied by or lookup query

// Range check $\text{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\text{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\text{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\texttt{is-or} \in \{0, 1\}$ - Performed in the CPU component

8.6 Basic Instruction Set: Branch Instructions

8.6.1 BEQ Instruction

The parameters and functionality for the BEQ instruction are as follows:

- opcode: BEQ
- Parameters: $(\text{a-val}, \text{b-val}, \text{c-val})$
- Instruction selector: $\texttt{is-beq} = 1$
- Functionality:
  - $\text{pc-next} \leftarrow \text{pc} + \text{c-val}$ if $\text{a-val} = \text{b-val}$

– pc-next $\leftarrow$ pc + 4 if a-val $\neq$ b-val

The mapping from the `beq` instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
beq	rs1	rs2	i	1	R[rs1]	R[rs2]	sext(i)

where sext is the sign extension function, i is a 12-bit value specifying bits 1-12 of the immediate value, and bit 0 of the immediate value is equal to 0.

Constraints assuming large fields

// Comparing a-val, b-val
// h-neq-flag $= 0$ indicates a-val $=$ b-val
// h-neq-flag $= 1$ indicates a-val $\neq$ b-val
- (is-beq) $\cdot$ ((a-val $-$ b-val) $\cdot$ h-neq-flag-aux $-$ h-neq-flag) $= 0$
- (is-beq) $\cdot$ (h-neq-flag) $\cdot$ ($1 -$ h-neq-flag) $= 0$

// Enforcing h-neq-flag-aux $\neq 0$
- (is-beq) $\cdot$ (h-neq-flag-aux $\cdot$ h-neq-flag-aux-inv $- 1$) $= 0$

// Setting pc-next based on comparison result, unless the next row is the first row
// pc-next $=$ pc $+$ c-val if h-neq-flag $= 0$
// pc-next $=$ pc $+ 4$ if h-neq-flag $= 1$
- (is-beq) $\cdot$ (($1 -$ h-neq-flag) $\cdot$ c-val $+$ (h-neq-flag) $\cdot 4 +$ pc $-$ pc-next $-$ h-carry $\cdot 2^{32}$) $= 0$

// Enforcing h-carry $\in \{0, 1\}$
- (is-beq) $\cdot$ (h-carry) $\cdot$ ($1 -$ h-carry) $= 0$

// Range check pc $\in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking
// Range check pc-next $\in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking
// Range check a-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check b-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check c-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check is-beq $\in \{0, 1\}$ - Performed in the CPU component

In the description above, the goal of the first three constraints is to ensure that h-neq-flag $= 0$ whenever a-val $=$ b-val and h-neq-flag $= 1$ if a-val $\neq$ b-val. To see why this is the case, first notice that the second and third constraints guarantee that h-neq-flag $\in \{0, 1\}$ and h-neq-flag-aux $\neq 0$. Now there two cases to consider:

- If a-val $=$ b-val, then the first constraint can only be satisfied if the term (a-val $-$ b-val) $\cdot$ h-neq-flag-aux $-$ h-neq-flag $= 0$, which implies h-neq-flag $= 0$. Note that h-neq-flag-aux can be set to any non-zero value in this case.
- If a-val $\neq$ b-val, then the term (a-val $-$ b-val) $\cdot$ h-neq-flag-aux $-$ h-neq-flag in the first constraint can only be satisfied if h-neq-flag-aux $= 1/$(a-val $-$ b-val) and h-neq-flag $= 1$.

The remaining constraints then simply enforce the correct increment to the program counter pc when computing pc-next by taking into account the value of the flag h-neq-flag. The value h-carry is simply introduce to help handle carries during the addition operation.

Constraints assuming small fields

// Comparing a-val, b-val two limbs at a time
// h-neq12-flag $= 0$ indicates (a-val$^{(1)}$, a-val$^{(2)}$) $=$ (b-val$^{(1)}$, b-val$^{(2)}$)
// h-neq12-flag $= 1$ indicates (a-val$^{(1)}$, a-val$^{(2)}$) $\neq$ (b-val$^{(1)}$, b-val$^{(2)}$)
// h-neq34-flag $= 0$ indicates (a-val$^{(3)}$, a-val$^{(4)}$) $=$ (b-val$^{(3)}$, b-val$^{(4)}$)
// h-neq34-flag $= 1$ indicates (a-val$^{(3)}$, a-val$^{(4)}$) $\neq$ (b-val$^{(3)}$, b-val$^{(4)}$)

- $(\texttt{is-beq}) \cdot ((\mathsf{a\text{-}val}^{(1)} + 2^8 \cdot \mathsf{a\text{-}val}^{(2)} - \mathsf{b\text{-}val}^{(1)} - 2^8 \cdot \mathsf{b\text{-}val}^{(2)}) \cdot \texttt{h-neq12-flag-aux} - \texttt{h-neq12-flag}) = 0$
- $(\texttt{is-beq}) \cdot ((\mathsf{a\text{-}val}^{(3)} + 2^8 \cdot \mathsf{a\text{-}val}^{(4)} - \mathsf{b\text{-}val}^{(3)} - 2^8 \cdot \mathsf{b\text{-}val}^{(4)}) \cdot \texttt{h-neq34-flag-aux} - \texttt{h-neq34-flag}) = 0$
- $(\texttt{is-beq}) \cdot (\texttt{h-neq12-flag}) \cdot (1 - \texttt{h-neq12-flag}) = 0$
- $(\texttt{is-beq}) \cdot (\texttt{h-neq34-flag}) \cdot (1 - \texttt{h-neq34-flag}) = 0$

// `Enforcing` $\texttt{h-neq12-flag-aux} \neq 0$, $\texttt{h-neq34-flag-aux} \neq 0$
- $(\texttt{is-beq}) \cdot (\texttt{h-neq12-flag-aux} \cdot \texttt{h-neq12-flag-aux-inv} - 1) = 0$
- $(\texttt{is-beq}) \cdot (\texttt{h-neq34-flag-aux} \cdot \texttt{h-neq34-flag-aux-inv} - 1) = 0$

// $\texttt{h-neq-flag} = 0$ `indicates` $\mathsf{a\text{-}val} = \mathsf{b\text{-}val}$
// $\texttt{h-neq-flag} = 1$ `indicates` $\mathsf{a\text{-}val} \neq \mathsf{b\text{-}val}$
- $(\texttt{is-beq}) \cdot ((1 - \texttt{h-neq12-flag}) \cdot (1 - \texttt{h-neq34-flag}) - (1 - \texttt{h-neq-flag})) = 0$

// `Setting pc-next based on comparison result, unless the next row is the first row`
// $\mathrm{pc\text{-}next} = \mathrm{pc} + \mathsf{c\text{-}val}$ `if` $\texttt{h-neq-flag} = 0$
// $\mathrm{pc\text{-}next} = \mathrm{pc} + 4$ `if` $\texttt{h-neq-flag} = 1$
// $\texttt{h-carry}^{(j)} \in \{0, 1\}$ `for` $j = 1, 2, 3, 4$ `used for carry handling`
- $(\texttt{is-beq}) \cdot ((1 - \texttt{h-neq-flag}) \cdot \mathsf{c\text{-}val}^{(1)} + (\texttt{h-neq-flag}) \cdot 4 + \mathrm{pc}^{(1)} - \mathrm{pc\text{-}next}^{(1)} - \texttt{h-carry}^{(1)} \cdot 2^8) = 0$
- $(\texttt{is-beq}) \cdot ((1 - \texttt{h-neq-flag}) \cdot \mathsf{c\text{-}val}^{(2)} + \mathrm{pc}^{(2)} + \texttt{h-carry}^{(1)} - \mathrm{pc\text{-}next}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^8) = 0$
- $(\texttt{is-beq}) \cdot ((1 - \texttt{h-neq-flag}) \cdot \mathsf{c\text{-}val}^{(3)} + \mathrm{pc}^{(3)} + \texttt{h-carry}^{(2)} - \mathrm{pc\text{-}next}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^8) = 0$
- $(\texttt{is-beq}) \cdot ((1 - \texttt{h-neq-flag}) \cdot \mathsf{c\text{-}val}^{(4)} + \mathrm{pc}^{(4)} + \texttt{h-carry}^{(3)} - \mathrm{pc\text{-}next}^{(4)} - \texttt{h-carry}^{(4)} \cdot 2^8) = 0$

// `Enforcing` $\texttt{h-carry}^{(j)} \in \{0, 1\}$ `for` $j = 1, 2, 3, 4$
- $(\texttt{is-beq}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-beq}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-beq}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-beq}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// `Range check` $\mathrm{pc}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Guaranteed by the program memory checking`
// `Range check` $\mathrm{pc\text{-}next}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Guaranteed by the program memory checking`
// `Range check` $\mathsf{a\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check` $\mathsf{b\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check` $\mathsf{c\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check` $\texttt{is-beq} \in \{0, 1\}$ `- Performed in the CPU component`
// `Range check` $\texttt{h-neq-flag} \in \{0, 1\}$ `- Implied by h-neq12-flag and h-neq34-flag range checks`

The set of constraints is similar to the ones used in large field case in Section 8.6.1, but introduces additional variables to help perform the comparison of the limbs of $\mathsf{a\text{-}val}$ and $\mathsf{b\text{-}val}$. More precisely, while the flag `h-neq12-flag` will contain the result of the comparison of the first two limbs of $\mathsf{a\text{-}val}$ and $\mathsf{b\text{-}val}$, `h-neq34-flag` will contain the result of the comparison of the last two limbs. The flag `h-neq-flag` can then easily be derived from `h-neq12-flag` and `h-neq34-flag` via the term $((1 - \texttt{h-neq12-flag}) \cdot (1 - \texttt{h-neq34-flag}) - (1 - \texttt{h-neq-flag}))$, which guarantees that $\texttt{h-neq-flag} = 0$ only if $\texttt{h-neq12-flag} = \texttt{h-neq34-flag} = 0$ as desired.

8.6.2 `BNE` Instruction

The parameters and functionality for the `BNE` instruction are as follows:

- opcode: `BNE`
- Parameters: $(\mathsf{a\text{-}val}, \mathsf{b\text{-}val}, \mathsf{c\text{-}val})$
- Instruction selector: $\texttt{is-bne} = 1$
- Functionality:
  - $\mathrm{pc\text{-}next} \leftarrow \mathrm{pc} + \mathsf{c\text{-}val}$ if $\mathsf{a\text{-}val} \neq \mathsf{b\text{-}val}$
  - $\mathrm{pc\text{-}next} \leftarrow \mathrm{pc} + 4$ if $\mathsf{a\text{-}val} = \mathsf{b\text{-}val}$

The mapping from the `bne` instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
`bne`	rs1	rs2	i	1	$R[\mathrm{rs1}]$	$R[\mathrm{rs2}]$	$\mathrm{sext}(i)$

where sext is the sign extension function, i is a 12-bit value specifying bits 1-12 of the immediate value, and bit 0 of the immediate value is equal to 0.

Constraints assuming large fields

// Comparing a-val, b-val
// h-neq-flag $= 0$ indicates a-val $=$ b-val
// h-neq-flag $= 1$ indicates a-val $\neq$ b-val

- $(\text{is-bne}) \cdot (\text{a-val} - \text{b-val}) \cdot \text{h-neq-flag-aux} - \text{h-neq-flag}) = 0$
- $(\text{is-bne}) \cdot (\text{h-neq-flag}) \cdot (1 - \text{h-neq-flag}) = 0$

// Enforcing h-neq-flag-aux $\neq 0$

- $(\text{is-bne}) \cdot (\text{h-neq-flag-aux} \cdot \text{h-neq-flag-aux-inv} - 1) = 0$

// Setting pc-next based on comparison result, unless the next row is the first row
// pc-next $=$ pc $+$ c-val if h-neq-flag $= 1$
// pc-next $=$ pc $+ 4$ if h-neq-flag $= 0$

- $(\text{is-bne}) \cdot ((\text{h-neq-flag}) \cdot \text{c-val} + (1 - \text{h-neq-flag}) \cdot 4 + \text{pc} - \text{pc-next} - \text{h-carry} \cdot 2^{32}) = 0$

// Enforcing h-carry $\in \{0, 1\}$

- $(\text{is-bne}) \cdot (\text{h-carry}) \cdot (1 - \text{h-carry}) = 0$

// Range check pc $\in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking
// Range check pc-next $\in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking
// Range check a-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check b-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check c-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check is-bne $\in \{0, 1\}$ - Performed in the CPU component

The set of constraints above is similar to one for the beq instruction in Section 8.6.1, except that pc-next is computed slightly differently.

Constraints assuming small fields

// Comparing a-val, b-val two limbs at a time
// h-neq12-flag $= 0$ indicates $(\text{a-val}^{(1)}, \text{a-val}^{(2)}) = (\text{b-val}^{(1)}, \text{b-val}^{(2)})$
// h-neq12-flag $= 1$ indicates $(\text{a-val}^{(1)}, \text{a-val}^{(2)}) \neq (\text{b-val}^{(1)}, \text{b-val}^{(2)})$
// h-neq34-flag $= 0$ indicates $(\text{a-val}^{(3)}, \text{a-val}^{(4)}) = (\text{b-val}^{(3)}, \text{b-val}^{(4)})$
// h-neq34-flag $= 1$ indicates $(\text{a-val}^{(3)}, \text{a-val}^{(4)}) \neq (\text{b-val}^{(3)}, \text{b-val}^{(4)})$

- $(\text{is-bne}) \cdot ((\text{a-val}^{(1)} + 2^8 \cdot \text{a-val}^{(2)} - \text{b-val}^{(1)} - 2^8 \cdot \text{b-val}^{(2)}) \cdot \text{h-neq12-flag-aux} - \text{h-neq12-flag}) = 0$
- $(\text{is-bne}) \cdot ((\text{a-val}^{(3)} + 2^8 \cdot \text{a-val}^{(4)} - \text{b-val}^{(3)} - 2^8 \cdot \text{b-val}^{(4)}) \cdot \text{h-neq34-flag-aux} - \text{h-neq34-flag}) = 0$
- $(\text{is-bne}) \cdot (\text{h-neq12-flag}) \cdot (1 - \text{h-neq12-flag}) = 0$
- $(\text{is-bne}) \cdot (\text{h-neq34-flag}) \cdot (1 - \text{h-neq34-flag}) = 0$

// Enforcing h-neq12-flag-aux $\neq 0$, h-neq34-flag-aux $\neq 0$

- $(\text{is-bne}) \cdot (\text{h-neq12-flag-aux} \cdot \text{h-neq12-flag-aux-inv} - 1) = 0$
- $(\text{is-bne}) \cdot (\text{h-neq34-flag-aux} \cdot \text{h-neq34-flag-aux-inv} - 1) = 0$

// h-neq-flag $= 0$ indicates a-val $=$ b-val
// h-neq-flag $= 1$ indicates a-val $\neq$ b-val

- $(\text{is-bne}) \cdot (1 - \text{h-neq12-flag}) \cdot (1 - \text{h-neq34-flag}) - (1 - \text{h-neq-flag}) = 0$

// Setting pc-next based on comparison result, unless the next row is the first row
// pc-next $=$ pc $+$ c-val if h-neq-flag $= 1$
// pc-next $=$ pc $+ 4$ if h-neq-flag $= 0$
// h-carry$^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$ used for carry handling

- $(\text{is-bne}) \cdot ((\text{h-neq-flag}) \cdot \text{c-val}^{(1)} + (1 - \text{h-neq-flag}) \cdot 4 + \text{pc}^{(1)} - \text{pc-next}^{(1)} - \text{h-carry}^{(1)} \cdot 2^8) = 0$
- $(\text{is-bne}) \cdot ((\text{h-neq-flag}) \cdot \text{c-val}^{(2)} + \text{pc}^{(2)} + \text{h-carry}^{(1)} - \text{pc-next}^{(2)} - \text{h-carry}^{(2)} \cdot 2^8) = 0$
- $(\text{is-bne}) \cdot ((\text{h-neq-flag}) \cdot \text{c-val}^{(3)} + \text{pc}^{(3)} + \text{h-carry}^{(2)} - \text{pc-next}^{(3)} - \text{h-carry}^{(3)} \cdot 2^8) = 0$
- $(\text{is-bne}) \cdot ((\text{h-neq-flag}) \cdot \text{c-val}^{(4)} + \text{pc}^{(4)} + \text{h-carry}^{(3)} - \text{pc-next}^{(4)} - \text{h-carry}^{(4)} \cdot 2^8) = 0$

// Enforcing h-carry$^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\text{is-bne}) \cdot (\text{h-carry}^{(1)}) \cdot (1 - \text{h-carry}^{(1)}) = 0$

- $(\texttt{is-bne}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-bne}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-bne}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// Range check $\mathrm{pc}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\text{pc-next}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\text{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\text{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\text{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\texttt{is-bne} \in \{0, 1\}$ - Performed in the CPU component
// Range check $\texttt{h-neq-flag} \in \{0, 1\}$ - Implied by h-neq12-flag and h-neq34-flag range checks

The set of constraints above is similar to one for the beq instruction in Section 8.6.1, except that pc-next is computed slightly differently.

8.6.3 BLTU Instruction

The parameters and functionality for the BLTU instruction are as follows:

- opcode: BLTU
- Parameters: (a-val, b-val, c-val)
- Instruction selector: $\texttt{is-bltu} = 1$
- Functionality:
  - pc-next ← pc + c-val if a-val < b-val (*unsigned* comparison)
  - pc-next ← pc + 4 if a-val ≥ b-val (*unsigned* comparison)

The mapping from the bltu instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
bltu	rs1	rs2	i	1	R[rs1]	R[rs2]	sext(i)

where sext is the sign extension function, i is a 12-bit value specifying bits 1-12 of the immediate value, and bit 0 of the immediate value is equal to 0.

Constraints assuming large fields

// Performing unsigned comparison between a-val and b-val using SUB borrow bit
- $(\texttt{is-bltu}) \cdot (\text{a-val} - \text{b-val} - \texttt{h-rem} + \texttt{h-ltu-flag} \cdot 2^{32}) = 0$

// Enforcing $\texttt{h-ltu-flag} \in \{0, 1\}$
- $(\texttt{is-bltu}) \cdot (\texttt{h-ltu-flag}) \cdot (1 - \texttt{h-ltu-flag}) = 0$

// Enforcing $\texttt{h-rem} \in \left[0, 2^{32} - 1\right]$
- $(\texttt{is-bltu}) \cdot (\texttt{h-rem} \in \left[0, 2^{32} - 1\right]) = 0$

// Setting pc-next based on comparison result, unless the next row is the first row in the STARK trace
// $\text{pc-next} = \mathrm{pc} + \text{c-val}$ if $\texttt{h-ltu-flag} = 1$
// $\text{pc-next} = \mathrm{pc} + 4$ if $\texttt{h-ltu-flag} = 0$
- $(\texttt{is-bltu}) \cdot ((\texttt{h-ltu-flag}) \cdot \text{c-val} + (1 - \texttt{h-ltu-flag}) \cdot 4 + \mathrm{pc} - \text{pc-next} - \texttt{h-carry} \cdot 2^{32}) = 0$

// Enforcing $\texttt{h-carry} \in \{0, 1\}$
- $(\texttt{is-bltu}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// Range check $\mathrm{pc} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking
// Range check $\text{pc-next} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking
// Range check $\text{a-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\text{b-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\text{c-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\texttt{is-bltu} \in \{0, 1\}$ - Performed in the CPU component

The set of constraints above first performs an unsigned comparison between a-val and b-val by computing the subtraction a-val − b-val and setting the comparison flag `h-ltu-flag` to the borrow bit. To see why this works, please notice that the borrow bit `h-ltu-flag` will always be equal to 1 when a-val $\leq$ b-val and equal to 0 if a-val $>$ b-val, as desired.

The remaining constraints then simply enforce the correct increment to the program counter pc when computing pc-next by taking into account the value of the flag `h-ltu-flag`. The value `h-carry` is simply introduce to help handle carries during the addition operation.

Constraints assuming small fields

// Borrow handling

- $(\texttt{is-bltu}) \cdot (\text{b-val}^{(1)} + \texttt{h-borrow}^{(1)} \cdot 2^8 - \text{c-val}^{(1)} - \texttt{h-rem}^{(1)}) = 0$
- $(\texttt{is-bltu}) \cdot (\text{b-val}^{(2)} + \texttt{h-borrow}^{(2)} \cdot 2^8 - \text{c-val}^{(2)} - \texttt{h-rem}^{(2)} - \texttt{h-borrow}^{(1)}) = 0$
- $(\texttt{is-bltu}) \cdot (\text{b-val}^{(3)} + \texttt{h-borrow}^{(3)} \cdot 2^8 - \text{c-val}^{(3)} - \texttt{h-rem}^{(3)} - \texttt{h-borrow}^{(2)}) = 0$
- $(\texttt{is-bltu}) \cdot (\text{b-val}^{(4)} + \texttt{h-borrow}^{(4)} \cdot 2^8 - \text{c-val}^{(4)} - \texttt{h-rem}^{(4)} - \texttt{h-borrow}^{(3)}) = 0$

// Enforcing h-borrow$^{(j)} \in \{0,1\}$ for $j = 1,2,3,4$

- $(\texttt{is-bltu}) \cdot (\texttt{h-borrow}^{(1)}) \cdot (1 - \texttt{h-borrow}^{(1)}) = 0$
- $(\texttt{is-bltu}) \cdot (\texttt{h-borrow}^{(2)}) \cdot (1 - \texttt{h-borrow}^{(2)}) = 0$
- $(\texttt{is-bltu}) \cdot (\texttt{h-borrow}^{(3)}) \cdot (1 - \texttt{h-borrow}^{(3)}) = 0$
- $(\texttt{is-bltu}) \cdot (\texttt{h-borrow}^{(4)}) \cdot (1 - \texttt{h-borrow}^{(4)}) = 0$

// Enforcing h-rem$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1,2,3,4$

- $(\texttt{is-bltu}) \cdot (\texttt{h-rem}^{(1)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-bltu}) \cdot (\texttt{h-rem}^{(2)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-bltu}) \cdot (\texttt{h-rem}^{(3)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-bltu}) \cdot (\texttt{h-rem}^{(4)} \in \left[0, 2^8 - 1\right]) = 0$

// Setting h-ltu-flag $=$ h-borrow$^{(4)}$

- $(\texttt{is-bltu}) \cdot (\texttt{h-borrow}^{(4)} - \texttt{h-ltu-flag}) = 0$

// Setting pc-next based on comparison result, unless the next row is the first row
// pc-next $=$ pc $+$ c-val if h-ltu-flag $= 1$
// pc-next $=$ pc $+ 4$ if h-ltu-flag $= 0$
// h-carry$^{(j)} \in \{0,1\}$ for $j = 1,2,3,4$ used for carry handling

- $(\texttt{is-bltu}) \cdot ((\texttt{h-ltu-flag}) \cdot \text{c-val}^{(1)} + (1 - \texttt{h-ltu-flag}) \cdot 4 + \text{pc}^{(1)} - \text{pc-next}^{(1)} - \texttt{h-carry}^{(1)} \cdot 2^8) = 0$
- $(\texttt{is-bltu}) \cdot ((\texttt{h-ltu-flag}) \cdot \text{c-val}^{(2)} + \text{pc}^{(2)} + \texttt{h-carry}^{(1)} - \text{pc-next}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^8) = 0$
- $(\texttt{is-bltu}) \cdot ((\texttt{h-ltu-flag}) \cdot \text{c-val}^{(3)} + \text{pc}^{(3)} + \texttt{h-carry}^{(2)} - \text{pc-next}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^8) = 0$
- $(\texttt{is-bltu}) \cdot ((\texttt{h-ltu-flag}) \cdot \text{c-val}^{(4)} + \text{pc}^{(4)} + \texttt{h-carry}^{(3)} - \text{pc-next}^{(4)} - \texttt{h-carry}^{(4)} \cdot 2^8) = 0$

// Enforcing h-carry$^{(j)} \in \{0,1\}$ for $j = 1,2,3,4$

- $(\texttt{is-bltu}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-bltu}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-bltu}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-bltu}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// Range check pc$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1,2,3,4$ - Guaranteed by the program memory checking
// Range check pc-next$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1,2,3,4$ - Guaranteed by the program memory checking
// Range check a-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1,2,3,4$ - Performed in the CPU component
// Range check b-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1,2,3,4$ - Performed in the CPU component
// Range check c-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1,2,3,4$ - Performed in the CPU component
// Range check is-bltu $\in \{0,1\}$ - Performed in the CPU component

The set of constraints above is similar to the large field case in Section 8.6.3, but introduces limbs for the carry and borrow values to be able to properly perform the subtractions and additions.

8.6.4 BLT Instruction

The parameters and functionality for the BLT instruction are as follows:

- opcode: BLT
- Parameters: $(\textsf{a-val}, \textsf{b-val}, \textsf{c-val})$
- Instruction selector: $\texttt{is-blt} = 1$
- Functionality:
  - pc-next ← pc + c-val if a-val < b-val (*signed* comparison)
  - pc-next ← pc + 4 if a-val ≥ b-val (*signed* comparison)

The mapping from the blt instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
blt	rs1	rs2	i	1	$R[\text{rs1}]$	$R[\text{rs2}]$	$\text{sext}(i)$

where sext is the sign extension function, i is a 12-bit value specifying bits 1-12 of the immediate value, and bit 0 of the immediate value is equal to 0.

Constraints assuming large fields

// Performing unsigned comparison between a-val and b-val using SUB borrow bit
- $(\texttt{is-blt}) \cdot (\textsf{a-val} - \textsf{b-val} - \texttt{h-rem} + \texttt{h-borrow} \cdot 2^{32}) = 0$

// Enforcing $\texttt{h-borrow} \in \{0, 1\}$
- $(\texttt{is-blt}) \cdot (\texttt{h-borrow}) \cdot (1 - \texttt{h-borrow}) = 0$

// Enforcing $\texttt{h-rem} \in \left[0, 2^{32} - 1\right]$
- $(\texttt{is-blt}) \cdot (\texttt{h-rem} \in \left[0, 2^{32} - 1\right]) = 0$

// Setting $\texttt{h-ltu-flag} = \texttt{h-borrow}$
- $(\texttt{is-blt}) \cdot (\texttt{h-borrow} - \texttt{h-ltu-flag}) = 0$

// Extracting sign bits from a-val and b-val
- $(\texttt{is-blt}) \cdot (\texttt{h-rem-a} + \texttt{h-sgn-a} \cdot 2^{31} - \textsf{a-val}) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-rem-b} + \texttt{h-sgn-b} \cdot 2^{31} - \textsf{b-val}) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-rem-a} \in \left[0, 2^{31} - 1\right]) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-rem-b} \in \left[0, 2^{31} - 1\right]) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-sgn-a}) \cdot (1 - \texttt{h-sgn-a}) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-sgn-a}) \cdot (1 - \texttt{h-sgn-b}) = 0$

// Computing h-lt-flag from h-ltu-flag and sign bits h-sgn-a and h-sgn-b
- $(\texttt{is-blt}) \cdot ((\texttt{h-sgn-a})(1 - \texttt{h-sgn-b}) + \texttt{h-ltu-flag}((\texttt{h-sgn-a})(\texttt{h-sgn-b}) + (1 - \texttt{h-sgn-a})(1 - \texttt{h-sgn-b})) - \texttt{h-lt-flag}) = 0$

// Setting pc-next based on comparison result, unless the next row is the first row in the STARK trace
// $\text{pc-next} = \text{pc} + \textsf{c-val}$ if $\texttt{h-lt-flag} = 1$
// $\text{pc-next} = \text{pc} + 4$ if $\texttt{h-lt-flag} = 0$
- $(\texttt{is-blt}) \cdot ((\texttt{h-lt-flag}) \cdot \textsf{c-val} + (1 - \texttt{h-lt-flag}) \cdot 4 + \text{pc} - \text{pc-next} - \texttt{h-carry} \cdot 2^{32}) = 0$

// Enforcing $\texttt{h-carry} \in \{0, 1\}$
- $(\texttt{is-blt}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// Range check $\text{pc} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking
// Range check $\text{pc-next} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking
// Range check $\textsf{a-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\textsf{b-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\textsf{c-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\texttt{is-blt} \in \{0, 1\}$ - Performed in the CPU component

The set of constraints above is similar to ones for the bltu instruction in Section 8.6.3, except that it takes the signs of a-val and b-val into account when enforcing the way pc-next is computed..

Constraints assuming small fields

// Computing unsigned comparison flag h-ltu-flag using SUB borrow bit

- $(\texttt{is-blt}) \cdot (\text{b-val}^{(1)} + \texttt{h-borrow}^{(1)} \cdot 2^8 - \text{c-val}^{(1)} - \texttt{h-rem}^{(1)}) = 0$
- $(\texttt{is-blt}) \cdot (\text{b-val}^{(2)} + \texttt{h-borrow}^{(2)} \cdot 2^8 - \text{c-val}^{(2)} - \texttt{h-rem}^{(2)} - \texttt{h-borrow}^{(1)}) = 0$
- $(\texttt{is-blt}) \cdot (\text{b-val}^{(3)} + \texttt{h-borrow}^{(3)} \cdot 2^8 - \text{c-val}^{(3)} - \texttt{h-rem}^{(3)} - \texttt{h-borrow}^{(2)}) = 0$
- $(\texttt{is-blt}) \cdot (\text{b-val}^{(4)} + \texttt{h-borrow}^{(4)} \cdot 2^8 - \text{c-val}^{(4)} - \texttt{h-rem}^{(4)} - \texttt{h-borrow}^{(3)}) = 0$

// Enforcing $\texttt{h-borrow}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\texttt{is-blt}) \cdot (\texttt{h-borrow}^{(1)}) \cdot (1 - \texttt{h-borrow}^{(1)}) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-borrow}^{(2)}) \cdot (1 - \texttt{h-borrow}^{(2)}) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-borrow}^{(3)}) \cdot (1 - \texttt{h-borrow}^{(3)}) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-borrow}^{(4)}) \cdot (1 - \texttt{h-borrow}^{(4)}) = 0$

// Enforcing $\texttt{h-rem}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$

- $(\texttt{is-blt}) \cdot (\texttt{h-rem}^{(1)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-rem}^{(2)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-rem}^{(3)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-rem}^{(4)} \in \left[0, 2^8 - 1\right]) = 0$

// Setting $\texttt{h-ltu-flag} = \texttt{h-borrow}^{(4)}$

- $(\texttt{is-blt}) \cdot (\texttt{h-borrow}^{(4)} - \texttt{h-ltu-flag}) = 0$

// Extracting sign bits from b-val and c-val

- $(\texttt{is-blt}) \cdot (\texttt{h-rem-b} + \texttt{h-sgn-b} \cdot 2^7 - \text{b-val}^{(4)}) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-rem-c} + \texttt{h-sgn-c} \cdot 2^7 - \text{c-val}^{(4)}) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-rem-b} \in \left[0, 2^7 - 1\right]) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-rem-c} \in \left[0, 2^7 - 1\right]) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-sgn-b}) \cdot (1 - \texttt{h-sgn-b}) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-sgn-c}) \cdot (1 - \texttt{h-sgn-c}) = 0$

// Computing h-lt-flag from h-ltu-flag and sign bits h-sgn-a and h-sgn-b

- $(\texttt{is-blt}) \cdot ((\texttt{h-sgn-a})(1 - \texttt{h-sgn-b}) + \texttt{h-ltu-flag}((\texttt{h-sgn-a})(\texttt{h-sgn-b}) + (1 - \texttt{h-sgn-a})(1 - \texttt{h-sgn-b})) - \texttt{h-lt-flag}) = 0$

// Setting pc-next based on comparison result, unless the next row is the first row
// $\text{pc-next} = \text{pc} + \text{c-val}$ if $\texttt{h-lt-flag} = 1$
// $\text{pc-next} = \text{pc} + 4$ if $\texttt{h-lt-flag} = 0$
// $\texttt{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$ used for carry handling

- $(\texttt{is-blt}) \cdot ((\texttt{h-lt-flag}) \cdot \text{c-val}^{(1)} + (1 - \texttt{h-lt-flag}) \cdot 4 + \text{pc}^{(1)} - \text{pc-next}^{(1)} - \texttt{h-carry}^{(1)} \cdot 2^8) = 0$
- $(\texttt{is-blt}) \cdot ((\texttt{h-lt-flag}) \cdot \text{c-val}^{(2)} + \text{pc}^{(2)} + \texttt{h-carry}^{(1)} - \text{pc-next}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^8) = 0$
- $(\texttt{is-blt}) \cdot ((\texttt{h-lt-flag}) \cdot \text{c-val}^{(3)} + \text{pc}^{(3)} + \texttt{h-carry}^{(2)} - \text{pc-next}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^8) = 0$
- $(\texttt{is-blt}) \cdot ((\texttt{h-lt-flag}) \cdot \text{c-val}^{(4)} + \text{pc}^{(4)} + \texttt{h-carry}^{(3)} - \text{pc-next}^{(4)} - \texttt{h-carry}^{(4)} \cdot 2^8) = 0$

// Enforcing $\texttt{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\texttt{is-blt}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-blt}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// Range check $\text{pc}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\text{pc-next}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\text{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\text{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\text{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\texttt{is-blt} \in \{0, 1\}$ - Performed in the CPU component

The set of constraints above is similar to ones for the bltu instruction in Section 8.6.3, except that it takes the signs of a-val and b-val into account when enforcing the way pc-next is computed..

8.6.5 `BGEU` Instruction

The parameters and functionality for the `BGEU` instruction are as follows:

- opcode: `BGEU`
- Parameters: $(\mathsf{a\text{-}val}, \mathsf{b\text{-}val}, \mathsf{c\text{-}val})$
- Instruction selector: $\texttt{is-bgeu} = 1$
- Functionality:
  - $\mathsf{pc\text{-}next} \leftarrow \mathrm{pc} + \mathsf{c\text{-}val}$ if $\mathsf{a\text{-}val} \geq \mathsf{b\text{-}val}$ (*unsigned* comparison)
  - $\mathsf{pc\text{-}next} \leftarrow \mathrm{pc} + 4$ if $\mathsf{a\text{-}val} < \mathsf{b\text{-}val}$ (*unsigned* comparison)

The mapping from the `bgeu` instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
`bgeu`	rs1	rs2	i	1	R[rs1]	R[rs2]	sext(i)

where sext is the sign extension function, i is a 12-bit value specifying bits 1-12 of the immediate value, and bit 0 of the immediate value is equal to 0.

Constraints assuming large fields

// `Performing unsigned comparison between a-val and b-val using SUB borrow bit`
- $(\texttt{is-bgeu}) \cdot (\mathsf{a\text{-}val} - \mathsf{b\text{-}val} - \texttt{h-rem} + \texttt{h-ltu-flag} \cdot 2^{32}) = 0$

// `Enforcing` $\texttt{h-ltu-flag} \in \{0, 1\}$
- $(\texttt{is-bgeu}) \cdot (\texttt{h-ltu-flag}) \cdot (1 - \texttt{h-ltu-flag}) = 0$

// `Enforcing` $\texttt{h-rem} \in \left[0, 2^{32} - 1\right]$
- $(\texttt{is-bgeu}) \cdot \left(\texttt{h-rem} \in \left[0, 2^{32} - 1\right]\right) = 0$

// `Setting pc-next based on comparison result, unless the next row is the first row in the STARK trace`
// $\mathsf{pc\text{-}next} = \mathrm{pc} + \mathsf{c\text{-}val}$ `if` $\texttt{h-ltu-flag} = 0$
// $\mathsf{pc\text{-}next} = \mathrm{pc} + 4$ `if` $\texttt{h-ltu-flag} = 1$
- $(\texttt{is-bgeu}) \cdot ((1 - \texttt{h-ltu-flag}) \cdot \mathsf{c\text{-}val} + (\texttt{h-ltu-flag}) \cdot 4 + \mathrm{pc} - \mathsf{pc\text{-}next} - \texttt{h-carry} \cdot 2^{32}) = 0$

// `Enforcing` $\texttt{h-carry} \in \{0, 1\}$
- $(\texttt{is-bgeu}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// `Range check` $\mathrm{pc} \in \left[0, 2^{32} - 1\right]$ `- Guaranteed by the program memory checking`
// `Range check` $\mathsf{pc\text{-}next} \in \left[0, 2^{32} - 1\right]$ `- Guaranteed by the program memory checking`
// `Range check` $\mathsf{a\text{-}val} \in \left[0, 2^{32} - 1\right]$ `- Performed in the CPU component`
// `Range check` $\mathsf{b\text{-}val} \in \left[0, 2^{32} - 1\right]$ `- Performed in the CPU component`
// `Range check` $\mathsf{c\text{-}val} \in \left[0, 2^{32} - 1\right]$ `- Performed in the CPU component`
// `Range check` $\texttt{is-bgeu} \in \{0, 1\}$ `- Performed in the CPU component`

The set of constraints above is similar to one for the `bltu` instruction in Section 8.6.3, except that $\mathsf{pc\text{-}next}$ is computed slightly differently based on the results of the `h-ltu-flag` flag.

Constraints assuming small fields

// `Borrow handling`
- $(\texttt{is-bgeu}) \cdot (\mathsf{b\text{-}val}^{(1)} + \texttt{h-borrow}^{(1)} \cdot 2^8 - \mathsf{c\text{-}val}^{(1)} - \texttt{h-rem}^{(1)}) = 0$
- $(\texttt{is-bgeu}) \cdot (\mathsf{b\text{-}val}^{(2)} + \texttt{h-borrow}^{(2)} \cdot 2^8 - \mathsf{c\text{-}val}^{(2)} - \texttt{h-rem}^{(2)} - \texttt{h-borrow}^{(1)}) = 0$
- $(\texttt{is-bgeu}) \cdot (\mathsf{b\text{-}val}^{(3)} + \texttt{h-borrow}^{(3)} \cdot 2^8 - \mathsf{c\text{-}val}^{(3)} - \texttt{h-rem}^{(3)} - \texttt{h-borrow}^{(2)}) = 0$
- $(\texttt{is-bgeu}) \cdot (\mathsf{b\text{-}val}^{(4)} + \texttt{h-borrow}^{(4)} \cdot 2^8 - \mathsf{c\text{-}val}^{(4)} - \texttt{h-rem}^{(4)} - \texttt{h-borrow}^{(3)}) = 0$

// `Enforcing` $\texttt{h-borrow}^{(j)} \in \{0, 1\}$ `for` $j = 1, 2, 3, 4$
- $(\texttt{is-bgeu}) \cdot (\texttt{h-borrow}^{(1)}) \cdot (1 - \texttt{h-borrow}^{(1)}) = 0$
- $(\texttt{is-bgeu}) \cdot (\texttt{h-borrow}^{(2)}) \cdot (1 - \texttt{h-borrow}^{(2)}) = 0$

- $(\texttt{is-bgeu}) \cdot (\texttt{h-borrow}^{(3)}) \cdot (1 - \texttt{h-borrow}^{(3)}) = 0$
- $(\texttt{is-bgeu}) \cdot (\texttt{h-borrow}^{(4)}) \cdot (1 - \texttt{h-borrow}^{(4)}) = 0$

// Enforcing $\texttt{h-rem}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$
- $(\texttt{is-bgeu}) \cdot (\texttt{h-rem}^{(1)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-bgeu}) \cdot (\texttt{h-rem}^{(2)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-bgeu}) \cdot (\texttt{h-rem}^{(3)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-bgeu}) \cdot (\texttt{h-rem}^{(4)} \in \left[0, 2^8 - 1\right]) = 0$

// Setting $\texttt{h-ltu-flag} = \texttt{h-borrow}^{(4)}$
- $(\texttt{is-bgeu}) \cdot (\texttt{h-borrow}^{(4)} - \texttt{h-ltu-flag}) = 0$

// Setting pc-next based on comparison result, unless the next row is the first row
// $\text{pc-next} = \text{pc} + \text{c-val}$ if $\texttt{h-ltu-flag} = 0$
// $\text{pc-next} = \text{pc} + 4$ if $\texttt{h-ltu-flag} = 1$
// $\texttt{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$ used for carry handling
- $(\texttt{is-bgeu}) \cdot ((1 - \texttt{h-ltu-flag}) \cdot \text{c-val}^{(1)} + (\texttt{h-ltu-flag}) \cdot 4 + \text{pc}^{(1)} - \text{pc-next}^{(1)} - \texttt{h-carry}^{(1)} \cdot 2^8) = 0$
- $(\texttt{is-bgeu}) \cdot ((1 - \texttt{h-ltu-flag}) \cdot \text{c-val}^{(2)} + \text{pc}^{(2)} + \texttt{h-carry}^{(1)} - \text{pc-next}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^8) = 0$
- $(\texttt{is-bgeu}) \cdot ((1 - \texttt{h-ltu-flag}) \cdot \text{c-val}^{(3)} + \text{pc}^{(3)} + \texttt{h-carry}^{(2)} - \text{pc-next}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^8) = 0$
- $(\texttt{is-bgeu}) \cdot ((1 - \texttt{h-ltu-flag}) \cdot \text{c-val}^{(4)} + \text{pc}^{(4)} + \texttt{h-carry}^{(3)} - \text{pc-next}^{(4)} - \texttt{h-carry}^{(4)} \cdot 2^8) = 0$

// Enforcing $\texttt{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$
- $(\texttt{is-bgeu}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-bgeu}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-bgeu}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-bgeu}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// Range check $\text{pc}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\text{pc-next}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\text{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\text{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\text{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\texttt{is-bgeu} \in \{0, 1\}$ - Performed in the CPU component

The set of constraints above is similar to one for the bltu instruction in Section 8.6.3, except that pc-next is computed slightly differently based on the results of the h-ltu-flag flag.

8.6.6 BGE Instruction

The parameters and functionality for the BGE instruction are as follows:

- opcode: BGE
- Parameters: (a-val, b-val, c-val)
- Instruction selector: is-bge $= 1$
- Functionality:
  - pc-next $\leftarrow$ pc + c-val if a-val $\geq$ b-val (*signed* comparison)
  - pc-next $\leftarrow$ pc + 4 if a-val $<$ b-val (*signed* comparison)

The mapping from the bge instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
bge	rs1	rs2	i	1	R[rs1]	R[rs2]	sext(i)

where sext is the sign extension function, i is a 12-bit value specifying bits 1-12 of the immediate value, and bit 0 of the immediate value is equal to 0.

Constraints assuming large fields

// Performing unsigned comparison between a-val and b-val using SUB borrow bit

- $(\texttt{is-bge}) \cdot (\text{a-val} - \text{b-val} - \texttt{h-rem} + \texttt{h-borrow} \cdot 2^{32}) = 0$

// Enforcing $\texttt{h-borrow} \in \{0, 1\}$

- $(\texttt{is-bge}) \cdot (\texttt{h-borrow}) \cdot (1 - \texttt{h-borrow}) = 0$

// Enforcing $\texttt{h-rem} \in \left[0, 2^{32} - 1\right]$

- $(\texttt{is-bge}) \cdot (\texttt{h-rem} \in \left[0, 2^{32} - 1\right]) = 0$

// Setting $\texttt{h-ltu-flag} = \texttt{h-borrow}$

- $(\texttt{is-bge}) \cdot (\texttt{h-borrow} - \texttt{h-ltu-flag}) = 0$

// Extracting sign bits from a-val and b-val

- $(\texttt{is-bge}) \cdot (\texttt{h-rem-a} + \texttt{h-sgn-a} \cdot 2^{31} - \text{a-val}) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-rem-b} + \texttt{h-sgn-b} \cdot 2^{31} - \text{b-val}) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-rem-a} \in \left[0, 2^{31} - 1\right]) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-rem-b} \in \left[0, 2^{31} - 1\right]) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-sgn-a}) \cdot (1 - \texttt{h-sgn-a}) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-sgn-a}) \cdot (1 - \texttt{h-sgn-b}) = 0$

// Computing h-lt-flag from h-ltu-flag and sign bits h-sgn-a and h-sgn-b

- $(\texttt{is-bge}) \cdot ((\texttt{h-sgn-a})(1 - \texttt{h-sgn-b}) + \texttt{h-ltu-flag}((\texttt{h-sgn-a})(\texttt{h-sgn-b}) + (1 - \texttt{h-sgn-a})(1 - \texttt{h-sgn-b})) - \texttt{h-lt-flag}) = 0$

// Setting pc-next based on comparison result, unless the next row is the first row in the STARK trace

// $\text{pc-next} = \text{pc} + \text{c-val}$ if $\texttt{h-lt-flag} = 0$

// $\text{pc-next} = \text{pc} + 4$ if $\texttt{h-lt-flag} = 1$

- $(\texttt{is-bge}) \cdot ((1 - \texttt{h-lt-flag}) \cdot \text{c-val} + (\texttt{h-lt-flag}) \cdot 4 + \text{pc} - \text{pc-next} - \texttt{h-carry} \cdot 2^{32}) = 0$

// Enforcing $\texttt{h-carry} \in \{0, 1\}$

- $(\texttt{is-blt}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// Range check $\text{pc} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking

// Range check $\text{pc-next} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking

// Range check $\text{a-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\text{b-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\text{c-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\texttt{is-bge} \in \{0, 1\}$ - Performed in the CPU component

The set of constraints above is similar to ones for the bgeu instruction in Section 8.6.5, except that it takes the signs of a-val and b-val into account when enforcing the way pc-next is computed.

Constraints assuming small fields

// Computing unsigned comparison flag h-ltu-flag using SUB borrow bit

- $(\texttt{is-bge}) \cdot (\text{b-val}^{(1)} + \texttt{h-borrow}^{(1)} \cdot 2^8 - \text{c-val}^{(1)} - \texttt{h-rem}^{(1)}) = 0$
- $(\texttt{is-bge}) \cdot (\text{b-val}^{(2)} + \texttt{h-borrow}^{(2)} \cdot 2^8 - \text{c-val}^{(2)} - \texttt{h-rem}^{(2)} - \texttt{h-borrow}^{(1)}) = 0$
- $(\texttt{is-bge}) \cdot (\text{b-val}^{(3)} + \texttt{h-borrow}^{(3)} \cdot 2^8 - \text{c-val}^{(3)} - \texttt{h-rem}^{(3)} - \texttt{h-borrow}^{(2)}) = 0$
- $(\texttt{is-bge}) \cdot (\text{b-val}^{(4)} + \texttt{h-borrow}^{(4)} \cdot 2^8 - \text{c-val}^{(4)} - \texttt{h-rem}^{(4)} - \texttt{h-borrow}^{(3)}) = 0$

// Enforcing $\texttt{h-borrow}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\texttt{is-bge}) \cdot (\texttt{h-borrow}^{(1)}) \cdot (1 - \texttt{h-borrow}^{(1)}) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-borrow}^{(2)}) \cdot (1 - \texttt{h-borrow}^{(2)}) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-borrow}^{(3)}) \cdot (1 - \texttt{h-borrow}^{(3)}) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-borrow}^{(4)}) \cdot (1 - \texttt{h-borrow}^{(4)}) = 0$

// Enforcing $\texttt{h-rem}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$

- $(\texttt{is-bge}) \cdot (\texttt{h-rem}^{(1)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-rem}^{(2)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-rem}^{(3)} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-rem}^{(4)} \in \left[0, 2^8 - 1\right]) = 0$

// Setting $\texttt{h-ltu-flag} = \texttt{h-borrow}^{(4)}$

- $(\texttt{is-bge}) \cdot (\texttt{h-borrow}^{(4)} - \texttt{h-ltu-flag}) = 0$

// Extracting sign bits from b-val and c-val
- $(\texttt{is-bge}) \cdot (\texttt{h-rem-b} + \texttt{h-sgn-b} \cdot 2^7 - \mathsf{b\text{-}val}^{(4)}) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-rem-c} + \texttt{h-sgn-c} \cdot 2^7 - \mathsf{c\text{-}val}^{(4)}) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-rem-b} \in \left[0, 2^7 - 1\right]) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-rem-c} \in \left[0, 2^7 - 1\right]) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-sgn-b}) \cdot (1 - \texttt{h-sgn-b}) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-sgn-c}) \cdot (1 - \texttt{h-sgn-c}) = 0$

// Computing h-lt-flag from h-ltu-flag and sign bits h-sgn-a and h-sgn-b
- $(\texttt{is-bge}) \cdot ((\texttt{h-sgn-a})(1 - \texttt{h-sgn-b}) + \texttt{h-ltu-flag}((\texttt{h-sgn-a})(\texttt{h-sgn-b}) + (1 - \texttt{h-sgn-a})(1 - \texttt{h-sgn-b})) - \texttt{h-lt-flag}) = 0$

// Setting pc-next based on comparison result, unless the next row is the first row
// $\mathsf{pc\text{-}next} = \mathsf{pc} + \texttt{c-val}$ if $\texttt{h-lt-flag} = 0$
// $\mathsf{pc\text{-}next} = \mathsf{pc} + 4$ if $\texttt{h-lt-flag} = 1$
// $\texttt{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$ used for carry handling
- $(\texttt{is-bge}) \cdot ((1 - \texttt{h-lt-flag}) \cdot \mathsf{c\text{-}val}^{(1)} + (\texttt{h-lt-flag}) \cdot 4 + \mathsf{pc}^{(1)} - \mathsf{pc\text{-}next}^{(1)} - \texttt{h-carry}^{(1)} \cdot 2^8) = 0$
- $(\texttt{is-bge}) \cdot ((1 - \texttt{h-lt-flag}) \cdot \mathsf{c\text{-}val}^{(2)} + \mathsf{pc}^{(2)} + \texttt{h-carry}^{(1)} - \mathsf{pc\text{-}next}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^8) = 0$
- $(\texttt{is-bge}) \cdot ((1 - \texttt{h-lt-flag}) \cdot \mathsf{c\text{-}val}^{(3)} + \mathsf{pc}^{(3)} + \texttt{h-carry}^{(2)} - \mathsf{pc\text{-}next}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^8) = 0$
- $(\texttt{is-bge}) \cdot ((1 - \texttt{h-lt-flag}) \cdot \mathsf{c\text{-}val}^{(4)} + \mathsf{pc}^{(4)} + \texttt{h-carry}^{(3)} - \mathsf{pc\text{-}next}^{(4)} - \texttt{h-carry}^{(4)} \cdot 2^8) = 0$

// Enforcing $\texttt{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$
- $(\texttt{is-bge}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-bge}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// Range check $\mathsf{pc}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\mathsf{pc\text{-}next}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\mathsf{a\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\mathsf{b\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\mathsf{c\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\texttt{is-bge} \in \{0, 1\}$ - Performed in the CPU component

The set of constraints above is similar to ones for the bgeu instruction in Section 8.6.5, except that it takes the signs of a-val and b-val into account when enforcing the way pc-next is computed.

8.7 Basic Instruction Set: Load Instructions

8.7.1 LB Instruction

The parameters and functionality for the LB instruction are as follows:

- opcode: LB
- Parameters: (a-val, b-val, c-val)
- Instruction selector: $\texttt{is-lb} = 1$
- Functionality:
  - base-addr := b-val + c-val mod 2^{32}
  - a-val ← sext(M[base-addr])

The mapping from the lb instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
lb	rd	rs1	i	1	R[rd]	R[rs1]	sext(i)

where sext is the sign extension function, i is a 12-bit value specifying bits 0-11 of the immediate value.

Constraints assuming large fields

// Computing memory read address $\mathsf{b\text{-}val} + \mathsf{c\text{-}val}$
- $(\texttt{is-lb}) \cdot (\mathsf{b\text{-}val} + \mathsf{c\text{-}val} - \texttt{h-ram-base-addr} - \texttt{h-carry} \cdot 2^{32}) = 0$

// Enforcing $\texttt{h-carry} \in \{0, 1\}$
- $(\texttt{is-lb}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// Reading byte from memory address h-ram-base-addr
- $(\texttt{is-lb}) \cdot (\mathrm{Read}_{\mathrm{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 0) - \texttt{ram-val1}) = 0$

// Extracting sign bit from ram-val1
- $(\texttt{is-lb}) \cdot (\texttt{h-ram-val-rem} + \texttt{h-ram-val-sgn} \cdot 2^{7} - \texttt{ram-val1}) = 0$
- $(\texttt{is-lb}) \cdot (\texttt{h-ram-val-rem} \in \left[0, 2^{7} - 1\right]) = 0$
- $(\texttt{is-lb}) \cdot (\texttt{h-ram-val-sgn}) \cdot (1 - \texttt{h-ram-val-sgn}) = 0$

// Performing sign extension of ram-val1
- $(\texttt{is-lb}) \cdot (\texttt{ram-val1} + \texttt{h-ram-val-sgn} \cdot (2^{24} - 1) \cdot (2^{8}) - \texttt{h-ram-sext-val}) = 0$

// Setting output to h-ram-sext-val
- $(\texttt{is-lb}) \cdot (\mathsf{a\text{-}val} - \texttt{h-ram-sext-val}) = 0$

// Range check $\texttt{h-ram-base-addr} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the RAM memory checking

// Range check $\texttt{ram-val1} \in \left[0, 2^{8} - 1\right]$ - Performed in the RAM component

// Range check $\mathsf{a\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\mathsf{b\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\mathsf{c\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\texttt{is-lb} \in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Computing memory read address $\mathsf{b\text{-}val} + \mathsf{c\text{-}val}$

// $\texttt{h-carry}^{(j)}$ for $j = 1, 2, 3, 4$ used for carry handling
- $(\texttt{is-lb}) \cdot (\mathsf{b\text{-}val}^{(1)} + \mathsf{c\text{-}val}^{(1)} - \texttt{h-ram-base-addr}^{(1)} - \texttt{h-carry}^{(1)} \cdot 2^{8}) = 0$
- $(\texttt{is-lb}) \cdot (\mathsf{b\text{-}val}^{(2)} + \mathsf{c\text{-}val}^{(2)} + \texttt{h-carry}^{(1)} - \texttt{h-ram-base-addr}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^{8}) = 0$
- $(\texttt{is-lb}) \cdot (\mathsf{b\text{-}val}^{(3)} + \mathsf{c\text{-}val}^{(3)} + \texttt{h-carry}^{(2)} - \texttt{h-ram-base-addr}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^{8}) = 0$
- $(\texttt{is-lb}) \cdot (\mathsf{b\text{-}val}^{(4)} + \mathsf{c\text{-}val}^{(4)} + \texttt{h-carry}^{(3)} - \texttt{h-ram-base-addr}^{(4)} - \texttt{h-carry}^{(4)} \cdot 2^{8}) = 0$

// Enforcing $\texttt{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$
- $(\texttt{is-lb}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-lb}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-lb}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-lb}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// $\texttt{h-ram-base-addr} = (\texttt{h-ram-base-addr}^{(1)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(4)})$

// Reading byte ram-val1 from memory address h-ram-base-addr
- $(\texttt{is-lb}) \cdot (\mathrm{Read}_{\mathrm{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 0) - \texttt{ram-val1}) = 0$

// Extracting sign bit from ram-val1
- $(\texttt{is-lb}) \cdot (\texttt{h-ram-val-rem} + \texttt{h-ram-val-sgn} \cdot 2^{7} - \texttt{ram-val1}) = 0$
- $(\texttt{is-lb}) \cdot (\texttt{h-ram-val-rem} \in \left[0, 2^{7} - 1\right]) = 0$
- $(\texttt{is-lb}) \cdot (\texttt{h-ram-val-sgn}) \cdot (1 - \texttt{h-ram-val-sgn}) = 0$

// Performing sign extension of ram-val1
- $(\texttt{is-lb}) \cdot (\texttt{ram-val1} - \texttt{h-ram-sext-val}^{(1)}) = 0$
- $(\texttt{is-lb}) \cdot (\texttt{h-ram-val-sgn} \cdot (2^{8} - 1) - \texttt{h-ram-sext-val}^{(2)}) = 0$
- $(\texttt{is-lb}) \cdot (\texttt{h-ram-val-sgn} \cdot (2^{8} - 1) - \texttt{h-ram-sext-val}^{(3)}) = 0$
- $(\texttt{is-lb}) \cdot (\texttt{h-ram-val-sgn} \cdot (2^{8} - 1) - \texttt{h-ram-sext-val}^{(4)}) = 0$

// Setting output to h-ram-sext-val
- $(\texttt{is-lb}) \cdot (\mathsf{a\text{-}val}^{(1)} - \texttt{h-ram-sext-val}^{(1)}) = 0$

- $(\texttt{is-lb}) \cdot (\textsf{a-val}^{(2)} - \texttt{h-ram-sext-val}^{(2)}) = 0$
- $(\texttt{is-lb}) \cdot (\textsf{a-val}^{(3)} - \texttt{h-ram-sext-val}^{(3)}) = 0$
- $(\texttt{is-lb}) \cdot (\textsf{a-val}^{(4)} - \texttt{h-ram-sext-val}^{(4)}) = 0$

// Range check $\texttt{h-ram-base-addr}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the RAM memory checking
// Range check $\texttt{ram-val1} \in \left[0, 2^8 - 1\right]$ - Performed in the RAM component
// Range check $\textsf{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\textsf{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\textsf{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\texttt{is-lb} \in \{0, 1\}$ - Performed in the CPU component

8.7.2 LH Instruction

The parameters and functionality for the LH instruction are as follows:

- opcode: LH
- Parameters: $(\textsf{a-val}, \textsf{b-val}, \textsf{c-val})$
- Instruction selector: $\texttt{is-lh} = 1$
- Functionality:
  - base-addr $:=$ $\textsf{b-val} + \textsf{c-val} \bmod 2^{32}$
  - $\textsf{a-val} \leftarrow \text{sext}(M[\text{base-addr} + 1] \parallel M[\text{base-addr}])$
- Requirements: base-addr needs to be a *multiple of 2* for memory alignment

The mapping from the lh instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
lh	rd	rs1	i	1	R[rd]	R[rs1]	sext(i)

where sext is the sign extension function, i is a 12-bit value specifying bits 0-11 of the immediate value.

Constraints assuming large fields

// Computing memory read address $\textsf{b-val} + \textsf{c-val} \equiv \texttt{h-ram-base-addr} \cdot 2$
// h-ram-base-addr used to enforce memory alignment
- $(\texttt{is-lh}) \cdot (\textsf{b-val} + \textsf{c-val} - 2 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-carry} \cdot 2^{32}) = 0$
- $(\texttt{is-lh}) \cdot (2 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-ram-base-addr}) = 0$

// Enforcing $\texttt{h-carry} \in \{0, 1\}$
- $(\texttt{is-lh}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// Enforcing $\texttt{h-ram-base-addr-aux} \in \left[0, 2^{31} - 1\right]$
- $(\texttt{is-lh}) \cdot (\texttt{h-ram-base-addr-aux} \in \left[0, 2^{31} - 1\right]) = 0$

// Reading byte ram-val1 from memory address h-ram-base-addr
// Reading byte ram-val2 from memory address $\texttt{h-ram-base-addr} + 1$
- $(\texttt{is-lh}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 0) - \texttt{ram-val1}) = 0$
- $(\texttt{is-lh}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 1) - \texttt{ram-val2}) = 0$

// Extracting sign bit from ram-val2
- $(\texttt{is-lh}) \cdot (\texttt{h-ram-val-rem} + \texttt{h-ram-val-sgn} \cdot 2^7 - \texttt{ram-val2}) = 0$
- $(\texttt{is-lh}) \cdot (\texttt{h-ram-val-rem} \in \left[0, 2^7 - 1\right]) = 0$
- $(\texttt{is-lh}) \cdot (\texttt{h-ram-val-sgn}) \cdot (1 - \texttt{h-ram-val-sgn}) = 0$

// Performing sign extension of $(\texttt{ram-val1}, \texttt{ram-val2})$
- $(\texttt{is-lh}) \cdot (\texttt{ram-val1} + \texttt{ram-val2} \cdot 2^8 + \texttt{h-ram-val-sgn} \cdot (2^{16} - 1) \cdot (2^{16}) - \texttt{h-ram-sext-val}) = 0$

// Setting output to h-ram-sext-val
- $(\texttt{is-lh}) \cdot (\textsf{a-val} - \texttt{h-ram-sext-val}) = 0$

// Range check $\texttt{h-ram-base-addr} \in \left[0, 2^{32}-1\right]$ - Guaranteed by the RAM memory checking
// Range check $\texttt{ram-val}i \in \left[0, 2^{8}-1\right]$ for $i = 1, 2$ - Performed in the RAM component
// Range check $\text{a-val} \in \left[0, 2^{32}-1\right]$ - Performed in the CPU component
// Range check $\text{b-val} \in \left[0, 2^{32}-1\right]$ - Performed in the CPU component
// Range check $\text{c-val} \in \left[0, 2^{32}-1\right]$ - Performed in the CPU component
// Range check $\texttt{is-lh} \in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Computing memory read address $\text{b-val} + \text{c-val} \equiv \texttt{h-ram-base-addr} \cdot 2$
// $\texttt{h-ram-base-addr}$ used to enforce memory alignment
// $\texttt{h-carry}^{(j)}$ for $j = 1, 2, 3, 4$ used for carry handling

- $(\texttt{is-lh}) \cdot (\text{b-val}^{(1)} + \text{c-val}^{(1)} - 2 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-carry}^{(1)} \cdot 2^8) = 0$
- $(\texttt{is-lh}) \cdot (2 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-ram-base-addr}^{(1)}) = 0$
- $(\texttt{is-lh}) \cdot (\text{b-val}^{(2)} + \text{c-val}^{(2)} + \texttt{h-carry}^{(1)} - \texttt{h-ram-base-addr}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^8) = 0$
- $(\texttt{is-lh}) \cdot (\text{b-val}^{(3)} + \text{c-val}^{(3)} + \texttt{h-carry}^{(2)} - \texttt{h-ram-base-addr}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^8) = 0$
- $(\texttt{is-lh}) \cdot (\text{b-val}^{(4)} + \text{c-val}^{(4)} + \texttt{h-carry}^{(3)} - \texttt{h-ram-base-addr}^{(4)} - \texttt{h-carry}^{(4)} \cdot 2^8) = 0$

// Enforcing $\texttt{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\texttt{is-lh}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-lh}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-lh}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-lh}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// Enforcing $\texttt{h-ram-base-addr-aux} \in \left[0, 2^7 - 1\right]$

- $(\texttt{is-lh}) \cdot (\texttt{h-ram-base-addr-aux} \in \left[0, 2^7 - 1\right]) = 0$

// $\texttt{h-ram-base-addr} = (\texttt{h-ram-base-addr}^{(1)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(4)})$
// Reading byte $\texttt{ram-val1}$ from memory address $\texttt{h-ram-base-addr}$
// Reading byte $\texttt{ram-val2}$ from memory address $\texttt{h-ram-base-addr} + 1$

- $(\texttt{is-lh}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 0) - \texttt{ram-val1}) = 0$
- $(\texttt{is-lh}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 1) - \texttt{ram-val2}) = 0$

// Extracting sign bit from $\texttt{ram-val2}$

- $(\texttt{is-lh}) \cdot (\texttt{h-ram-val-rem} + \texttt{h-ram-val-sgn} \cdot 2^7 - \texttt{ram-val2}) = 0$
- $(\texttt{is-lh}) \cdot (\texttt{h-ram-val-rem} \in \left[0, 2^7 - 1\right]) = 0$
- $(\texttt{is-lh}) \cdot (\texttt{h-ram-val-sgn}) \cdot (1 - \texttt{h-ram-val-sgn}) = 0$

// Performing sign extension of $(\texttt{ram-val1}, \texttt{ram-val2})$

- $(\texttt{is-lh}) \cdot (\texttt{ram-val1} - \texttt{h-ram-sext-val}^{(1)}) = 0$
- $(\texttt{is-lh}) \cdot (\texttt{ram-val2} - \texttt{h-ram-sext-val}^{(2)}) = 0$
- $(\texttt{is-lh}) \cdot (\texttt{h-ram-val-sgn} \cdot (2^8 - 1) - \texttt{h-ram-sext-val}^{(3)}) = 0$
- $(\texttt{is-lh}) \cdot (\texttt{h-ram-val-sgn} \cdot (2^8 - 1) - \texttt{h-ram-sext-val}^{(4)}) = 0$

// Setting output to $\texttt{h-ram-sext-val}$

- $(\texttt{is-lh}) \cdot (\text{a-val}^{(1)} - \texttt{h-ram-sext-val}^{(1)}) = 0$
- $(\texttt{is-lh}) \cdot (\text{a-val}^{(2)} - \texttt{h-ram-sext-val}^{(2)}) = 0$
- $(\texttt{is-lh}) \cdot (\text{a-val}^{(3)} - \texttt{h-ram-sext-val}^{(3)}) = 0$
- $(\texttt{is-lh}) \cdot (\text{a-val}^{(4)} - \texttt{h-ram-sext-val}^{(4)}) = 0$

// Range check $\texttt{h-ram-base-addr}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the RAM memory checking
// Range check $\texttt{ram-val}j \in \left[0, 2^8 - 1\right]$ for $j = 1, 2$ - Performed in the RAM component
// Range check $\text{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\text{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\text{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\texttt{is-lh} \in \{0, 1\}$ - Performed in the CPU component

8.7.3 LW Instruction

The parameters and functionality for the LW instruction are as follows:

- opcode: LW
- Parameters: (a-val, b-val, c-val)
- Instruction selector: $\texttt{is-lw} = 1$
- Functionality:
  - base-addr := b-val + c-val mod 2^{32}
  - a-val ← M[base-addr + 3] ∥ M[base-addr + 2] ∥ M[base-addr + 1] ∥ M[base-addr]
- Requirements: base-addr needs to be a *multiple of 4* for memory alignment

The mapping from the lw instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
lw	rd	rs1	i	1	R[rd]	R[rs1]	sext(i)

where sext is the sign extension function, i is a 12-bit value specifying bits 0-11 of the immediate value.

Constraints assuming large fields

// Computing memory read address $\text{b-val} + \text{c-val} \equiv \texttt{h-ram-base-addr} \cdot 4$
// h-ram-base-addr used to enforce memory alignment
- $(\texttt{is-lw}) \cdot (\text{b-val} + \text{c-val} - 4 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-carry} \cdot 2^{32}) = 0$
- $(\texttt{is-lw}) \cdot (4 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-ram-base-addr}) = 0$

// Enforcing $\texttt{h-carry} \in \{0, 1\}$
- $(\texttt{is-lw}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// Enforcing $\texttt{h-ram-base-addr-aux} \in \left[0, 2^{30} - 1\right]$
- $(\texttt{is-lw}) \cdot (\texttt{h-ram-base-addr-aux} \in \left[0, 2^{30} - 1\right]) = 0$

// Reading byte ram-val1 from memory address h-ram-base-addr
// Reading byte ram-val2 from memory address $\texttt{h-ram-base-addr} + 1$
// Reading byte ram-val3 from memory address $\texttt{h-ram-base-addr} + 2$
// Reading byte ram-val4 from memory address $\texttt{h-ram-base-addr} + 3$
- $(\texttt{is-lw}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 0) - \texttt{ram-val1}) = 0$
- $(\texttt{is-lw}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 1) - \texttt{ram-val2}) = 0$
- $(\texttt{is-lw}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 2) - \texttt{ram-val3}) = 0$
- $(\texttt{is-lw}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 3) - \texttt{ram-val4}) = 0$

// Computing h-ram-sext-val from ram-val$i$
- $(\texttt{is-lw}) \cdot (\texttt{ram-val1} + \texttt{ram-val2} \cdot 2^{8} + \texttt{ram-val3} \cdot 2^{16} + \texttt{ram-val4} \cdot 2^{24} - \texttt{h-ram-sext-val}) = 0$

// Setting output to h-ram-sext-val
- $(\texttt{is-lw}) \cdot (\text{a-val} - \texttt{h-ram-sext-val}) = 0$

// Range check $\texttt{h-ram-base-addr} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the RAM memory checking
// Range check $\texttt{ram-val}j \in \left[0, 2^{8} - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the RAM component
// Range check $\text{a-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\text{b-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\text{c-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\texttt{is-lw} \in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Computing memory read address $\text{b-val} + \text{c-val} \equiv \texttt{h-ram-base-addr} \cdot 4$
// h-ram-base-addr used to enforce memory alignment

// $\texttt{h-carry}^{(j)}$ `for` $j = 1, 2, 3, 4$ `used for carry handling`

- $(\texttt{is-lw}) \cdot (\textsf{b-val}^{(1)} + \textsf{c-val}^{(1)} - 4 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-carry}^{(1)} \cdot 2^8) = 0$
- $(\texttt{is-lw}) \cdot (4 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-ram-base-addr}^{(1)}) = 0$
- $(\texttt{is-lw}) \cdot (\textsf{b-val}^{(2)} + \textsf{c-val}^{(2)} + \texttt{h-carry}^{(1)} - \texttt{h-ram-base-addr}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^8) = 0$
- $(\texttt{is-lw}) \cdot (\textsf{b-val}^{(3)} + \textsf{c-val}^{(3)} + \texttt{h-carry}^{(2)} - \texttt{h-ram-base-addr}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^8) = 0$
- $(\texttt{is-lw}) \cdot (\textsf{b-val}^{(4)} + \textsf{c-val}^{(4)} + \texttt{h-carry}^{(3)} - \texttt{h-ram-base-addr}^{(4)} - \texttt{h-carry}^{(4)} \cdot 2^8) = 0$

// `Enforcing` $\texttt{h-ram-base-addr-aux} \in \left[0, 2^6 - 1\right]$

- $(\texttt{is-lw}) \cdot (\texttt{h-ram-base-addr-aux} \in \left[0, 2^6 - 1\right]) = 0$

// `Enforcing` $\texttt{h-carry}^{(j)} \in \{0, 1\}$ `for` $j = 1, 2, 3, 4$

- $(\texttt{is-lw}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-lw}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-lw}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-lw}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// $\texttt{h-ram-base-addr} = (\texttt{h-ram-base-addr}^{(1)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(4)})$
// `Reading byte ram-val1 from memory address h-ram-base-addr`
// `Reading byte ram-val2 from memory address` $\texttt{h-ram-base-addr} + 1$
// `Reading byte ram-val3 from memory address` $\texttt{h-ram-base-addr} + 2$
// `Reading byte ram-val4 from memory address` $\texttt{h-ram-base-addr} + 3$

- $(\texttt{is-lw}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 0) - \texttt{ram-val1}) = 0$
- $(\texttt{is-lw}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 1) - \texttt{ram-val2}) = 0$
- $(\texttt{is-lw}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 2) - \texttt{ram-val3}) = 0$
- $(\texttt{is-lw}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 3) - \texttt{ram-val4}) = 0$

// `Setting` $\textsf{a-val}^{(j)}$ `to` $\texttt{ram-val}j$ `for` $j = 1, 2, 3, 4$

- $(\texttt{is-lw}) \cdot (\textsf{a-val}^{(1)} - \texttt{ram-val1}) = 0$
- $(\texttt{is-lw}) \cdot (\textsf{a-val}^{(2)} - \texttt{ram-val2}) = 0$
- $(\texttt{is-lw}) \cdot (\textsf{a-val}^{(3)} - \texttt{ram-val3}) = 0$
- $(\texttt{is-lw}) \cdot (\textsf{a-val}^{(4)} - \texttt{ram-val4}) = 0$

// `Range check` $\texttt{h-ram-base-addr}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Guaranteed by the RAM memory checking`
// `Range check` $\texttt{ram-val}j \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the RAM component`
// `Range check` $\textsf{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check` $\textsf{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check` $\textsf{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ `for` $j = 1, 2, 3, 4$ `- Performed in the CPU component`
// `Range check` $\texttt{is-lw} \in \{0, 1\}$ `- Performed in the CPU component`

8.7.4 LBU Instruction

The parameters and functionality for the LBU instruction are as follows:

- opcode: LBU
- Parameters: $(\textsf{a-val}, \textsf{b-val}, \textsf{c-val})$
- Instruction selector: $\texttt{is-lbu} = 1$
- Functionality:
  - $\text{base-addr} := \textsf{b-val} + \textsf{c-val} \bmod 2^{32}$
  - $\textsf{a-val} \leftarrow \text{zext}(M[\text{base-addr}])$

  where zext is the zero extension function

The mapping from the **lbu** instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
lbu	rd	rs1	i	1	R[rd]	R[rs1]	sext(i)

where sext is the sign extension function, i is a 12-bit value specifying bits 0-11 of the immediate value.

Constraints assuming large fields

// Computing memory read address $\mathsf{b\text{-}val} + \mathsf{c\text{-}val}$

- $(\texttt{is-lbu}) \cdot (\mathsf{b\text{-}val} + \mathsf{c\text{-}val} - \texttt{h-ram-base-addr} - \texttt{h-carry} \cdot 2^{32}) = 0$

// Enforcing $\texttt{h-carry} \in \{0, 1\}$

- $(\texttt{is-lbu}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// Reading byte from memory address h-ram-base-addr

- $(\texttt{is-lbu}) \cdot (\mathrm{Read}_{\mathrm{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 0) - \texttt{ram-val1}) = 0$

// Performing zero extension of ram-val1

- $(\texttt{is-lbu}) \cdot (\texttt{ram-val1} - \texttt{h-ram-zext-val}) = 0$

// Setting $\mathsf{a\text{-}val}$ to h-ram-zext-val

- $(\texttt{is-lbu}) \cdot (\mathsf{a\text{-}val} - \texttt{h-ram-zext-val}) = 0$

// Range check $\texttt{h-ram-base-addr} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the RAM memory checking

// Range check $\texttt{ram-val1} \in \left[0, 2^{8} - 1\right]$ - Performed in the RAM component

// Range check $\mathsf{a\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\mathsf{b\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\mathsf{c\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\texttt{is-lbu} \in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Computing memory read address $\mathsf{b\text{-}val} + \mathsf{c\text{-}val}$

// $\texttt{h-carry}^{(j)}$ for $j = 1, 2, 3, 4$ used for carry handling

- $(\texttt{is-lbu}) \cdot (\mathsf{b\text{-}val}^{(1)} + \mathsf{c\text{-}val}^{(1)} - \texttt{h-ram-base-addr}^{(1)} - \texttt{h-carry}^{(1)} \cdot 2^{8}) = 0$
- $(\texttt{is-lbu}) \cdot (\mathsf{b\text{-}val}^{(2)} + \mathsf{c\text{-}val}^{(2)} + \texttt{h-carry}^{(1)} - \texttt{h-ram-base-addr}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^{8}) = 0$
- $(\texttt{is-lbu}) \cdot (\mathsf{b\text{-}val}^{(3)} + \mathsf{c\text{-}val}^{(3)} + \texttt{h-carry}^{(2)} - \texttt{h-ram-base-addr}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^{8}) = 0$
- $(\texttt{is-lbu}) \cdot (\mathsf{b\text{-}val}^{(4)} + \mathsf{c\text{-}val}^{(4)} + \texttt{h-carry}^{(3)} - \texttt{h-ram-base-addr}^{(4)} - \texttt{h-carry}^{(4)} \cdot 2^{8}) = 0$

// Enforcing $\texttt{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\texttt{is-lbu}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-lbu}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-lbu}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-lbu}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// $\texttt{h-ram-base-addr} = (\texttt{h-ram-base-addr}^{(1)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(4)})$

// Reading byte ram-val1 from memory address h-ram-base-addr

- $(\texttt{is-lbu}) \cdot (\mathrm{Read}_{\mathrm{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 0) - \texttt{ram-val1}) = 0$

// Performing zero extension of ram-val1

- $(\texttt{is-lbu}) \cdot (\texttt{ram-val1} - \texttt{h-ram-zext-val}^{(1)}) = 0$
- $(\texttt{is-lbu}) \cdot (\texttt{h-ram-zext-val}^{(2)}) = 0$
- $(\texttt{is-lbu}) \cdot (\texttt{h-ram-zext-val}^{(3)}) = 0$
- $(\texttt{is-lbu}) \cdot (\texttt{h-ram-zext-val}^{(4)}) = 0$

// Setting output to h-ram-zext-val

- $(\texttt{is-lbu}) \cdot (\mathsf{a\text{-}val}^{(1)} - \texttt{h-ram-zext-val}^{(1)}) = 0$
- $(\texttt{is-lbu}) \cdot (\mathsf{a\text{-}val}^{(2)} - \texttt{h-ram-zext-val}^{(2)}) = 0$
- $(\texttt{is-lbu}) \cdot (\mathsf{a\text{-}val}^{(3)} - \texttt{h-ram-zext-val}^{(3)}) = 0$
- $(\texttt{is-lbu}) \cdot (\mathsf{a\text{-}val}^{(4)} - \texttt{h-ram-zext-val}^{(4)}) = 0$

// Range check $\texttt{h-ram-base-addr}^{(j)} \in \left[0, 2^{8} - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the RAM memory checking

// Range check $\texttt{ram-val1} \in \left[0, 2^{8} - 1\right]$ - Performed in the RAM component

// Range check $\mathsf{a\text{-}val}^{(j)} \in \left[0, 2^{8} - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check $\mathsf{b\text{-}val}^{(j)} \in \left[0, 2^{8} - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check $\mathsf{c\text{-}val}^{(j)} \in \left[0, 2^{8} - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check $\texttt{is-lbu} \in \{0, 1\}$ - Performed in the CPU component

8.7.5 `LHU` Instruction

The parameters and functionality for the `LHU` instruction are as follows:

- opcode: `LHU`
- Parameters: $(\textsf{a-val}, \textsf{b-val}, \textsf{c-val})$
- Instruction selector: $\texttt{is-lhu} = 1$
- Functionality:
  - base-addr $:= \textsf{b-val} + \textsf{c-val} \bmod 2^{32}$
  - $\textsf{a-val} \leftarrow M[\text{base-addr} + 1] \,\|\, M[\text{base-addr}]$

  where zext is the zero extension function
- Requirements: base-addr needs to be a *multiple of 2* for memory alignment

The mapping from the `lhu` instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
`lhu`	rd	rs1	i	1	R[rd]	R[rs1]	sext(i)

where sext is the sign extension function, i is a 12-bit value specifying bits 0-11 of the immediate value.

Constraints assuming large fields

// `Computing memory read address` $\textsf{b-val} + \textsf{c-val} \equiv \texttt{h-ram-base-addr} \cdot 2$
// `h-ram-base-addr used to enforce memory alignment`
- $(\texttt{is-lhu}) \cdot (\textsf{b-val} + \textsf{c-val} - 2 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-carry} \cdot 2^{32}) = 0$
- $(\texttt{is-lhu}) \cdot (2 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-ram-base-addr}) = 0$

// `Enforcing h-carry` $\in \{0, 1\}$
- $(\texttt{is-lhu}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// `Enforcing h-ram-base-addr-aux` $\in \left[0, 2^{31} - 1\right]$
- $(\texttt{is-lhu}) \cdot (\texttt{h-ram-base-addr-aux} \in \left[0, 2^{31} - 1\right]) = 0$

// `Reading byte ram-val1 from memory address h-ram-base-addr`
// `Reading byte ram-val2 from memory address` $\texttt{h-ram-base-addr} + 1$
- $(\texttt{is-lhu}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 0) - \texttt{ram-val1}) = 0$
- $(\texttt{is-lhu}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 1) - \texttt{ram-val2}) = 0$

// `Performing zero extension of` $(\texttt{ram-val1}, \texttt{ram-val2})$
- $(\texttt{is-lhu}) \cdot (\texttt{ram-val1} + \texttt{ram-val2} \cdot 2^8 - \texttt{h-ram-zext-val}) = 0$

// `Setting output to h-ram-zext-val`
- $(\texttt{is-lhu}) \cdot (\textsf{a-val} - \texttt{h-ram-zext-val}) = 0$

// `Range check h-ram-base-addr` $\in \left[0, 2^{32} - 1\right]$ `- Guaranteed by the RAM memory checking`
// `Range check ram-val`$i \in \left[0, 2^{8} - 1\right]$ `for` $i = 1, 2$ `- Performed in the RAM component`
// `Range check a-val` $\in \left[0, 2^{32} - 1\right]$ `- Performed in the CPU component`
// `Range check b-val` $\in \left[0, 2^{32} - 1\right]$ `- Performed in the CPU component`
// `Range check c-val` $\in \left[0, 2^{32} - 1\right]$ `- Performed in the CPU component`
// `Range check is-lhu` $\in \{0, 1\}$ `- Performed in the CPU component`

Constraints assuming small fields

// `Computing memory read address` $\textsf{b-val} + \textsf{c-val} \equiv \texttt{h-ram-base-addr} \cdot 2$
// `h-ram-base-addr used to enforce memory alignment`
// $\texttt{h-carry}^{(j)}$ `for` $j = 1, 2, 3, 4$ `used for carry handling`
- $(\texttt{is-lhu}) \cdot (\textsf{b-val}^{(1)} + \textsf{c-val}^{(1)} - 2 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-carry}^{(1)} \cdot 2^8) = 0$

- $(\texttt{is-lhu}) \cdot (2 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-ram-base-addr}^{(1)}) = 0$
- $(\texttt{is-lhu}) \cdot (\textsf{b-val}^{(2)} + \textsf{c-val}^{(2)} + \texttt{h-carry}^{(1)} - \texttt{h-ram-base-addr}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^8) = 0$
- $(\texttt{is-lhu}) \cdot (\textsf{b-val}^{(3)} + \textsf{c-val}^{(3)} + \texttt{h-carry}^{(2)} - \texttt{h-ram-base-addr}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^8) = 0$
- $(\texttt{is-lhu}) \cdot (\textsf{b-val}^{(4)} + \textsf{c-val}^{(4)} + \texttt{h-carry}^{(3)} - \texttt{h-ram-base-addr}^{(4)} - \texttt{h-carry}^{(4)} \cdot 2^8) = 0$

// Enforcing $\texttt{h-carry}^{(j)} \in \{0,1\}$ for $j = 1,2,3,4$
- $(\texttt{is-lhu}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-lhu}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-lhu}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-lhu}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// Enforcing $\texttt{h-ram-base-addr-aux} \in \left[0, 2^7 - 1\right]$
- $(\texttt{is-lhu}) \cdot (\texttt{h-ram-base-addr-aux} \in \left[0, 2^7 - 1\right]) = 0$

// $\texttt{h-ram-base-addr} = (\texttt{h-ram-base-addr}^{(1)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(4)})$
// Reading byte ram-val1 from memory address h-ram-base-addr
// Reading byte ram-val2 from memory address $\texttt{h-ram-base-addr} + 1$
- $(\texttt{is-lhu}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 0) - \texttt{ram-val1}) = 0$
- $(\texttt{is-lhu}) \cdot (\text{Read}_{\text{RAM}}(\texttt{clk}, \texttt{h-ram-base-addr}, 1) - \texttt{ram-val2}) = 0$

// Performing zero extension of $(\texttt{ram-val1}, \texttt{ram-val2})$
- $(\texttt{is-lhu}) \cdot (\texttt{ram-val1} - \texttt{h-ram-zext-val}^{(1)}) = 0$
- $(\texttt{is-lhu}) \cdot (\texttt{ram-val2} - \texttt{h-ram-zext-val}^{(2)}) = 0$
- $(\texttt{is-lhu}) \cdot (\texttt{h-ram-sext-val}^{(3)}) = 0$
- $(\texttt{is-lhu}) \cdot (\texttt{h-ram-sext-val}^{(4)}) = 0$

// Setting output to h-ram-zext-val
- $(\texttt{is-lhu}) \cdot (\textsf{a-val}^{(1)} - \texttt{h-ram-zext-val}^{(1)}) = 0$
- $(\texttt{is-lhu}) \cdot (\textsf{a-val}^{(2)} - \texttt{h-ram-zext-val}^{(2)}) = 0$
- $(\texttt{is-lhu}) \cdot (\textsf{a-val}^{(3)} - \texttt{h-ram-zext-val}^{(3)}) = 0$
- $(\texttt{is-lhu}) \cdot (\textsf{a-val}^{(4)} - \texttt{h-ram-zext-val}^{(4)}) = 0$

// Range check $\texttt{h-ram-base-addr}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1,2,3,4$ - Guaranteed by the RAM memory checking
// Range check $\texttt{ram-val}j \in \left[0, 2^8 - 1\right]$ for $j = 1,2$ - Performed in the RAM component
// Range check $\textsf{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1,2,3,4$ - Performed in the CPU component
// Range check $\textsf{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1,2,3,4$ - Performed in the CPU component
// Range check $\textsf{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1,2,3,4$ - Performed in the CPU component
// Range check $\texttt{is-lhu} \in \{0,1\}$ - Performed in the CPU component

8.8 Basic Instruction Set: Store Instructions

8.8.1 SB Instruction

The parameters and functionality for the SB instruction are as follows:

- opcode: SB
- Parameters: $(\textsf{a-val}, \textsf{b-val}, \textsf{c-val})$
- Instruction selector: $\texttt{is-sb} = 1$
- Functionality:
  - base-addr $:= \textsf{b-val} + \textsf{c-val} \bmod 2^{32}$
  - $M[\text{base-addr}] \leftarrow \textsf{a-val}\ \&\ \texttt{0x000000FF}$

The mapping from the sb instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
sb	rs1	rs2	i	1	R[rs1]	R[rs2]	sext(i)

where sext is the sign extension function, i is a 12-bit value specifying bits 0-11 of the immediate value.

Constraints assuming large fields

// Computing memory write address $\mathsf{b\text{-}val} + \mathsf{c\text{-}val} \equiv \texttt{h-ram-base-addr}$

- $(\texttt{is-sb}) \cdot (\mathsf{b\text{-}val} + \mathsf{c\text{-}val} - \texttt{h-ram-base-addr} - \texttt{h-carry} \cdot 2^{32}) = 0$

// Enforcing $\texttt{h-carry} \in \{0, 1\}$

- $(\texttt{is-sb}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// Extracting ram-val1 from bits 0-7 of $\mathsf{a\text{-}val}$

- $(\texttt{is-sb}) \cdot (\mathsf{a\text{-}val} - \texttt{h-ram-val-rem} \cdot 2^8 - \texttt{ram-val1}) = 0$
- $(\texttt{is-sb}) \cdot (\texttt{ram-val1} \in \left[0, 2^8 - 1\right]) = 0$
- $(\texttt{is-sb}) \cdot (\texttt{h-ram-val-rem} \in \left[0, 2^{24} - 1\right]) = 0$

// Writing byte ram-val1 at memory address h-ram-base-addr

- $(\texttt{is-sb}) \cdot (\mathrm{Write}_{\mathrm{RAM}}(\texttt{clk}, \texttt{ram-val1}, \texttt{h-ram-base-addr}, 0)) = 0$

// Range check $\texttt{h-ram-base-addr} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the RAM memory checking

// Range check $\texttt{ram-val1} \in \left[0, 2^8 - 1\right]$ - Also performed in the RAM component

// Range check $\mathsf{a\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\mathsf{b\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\mathsf{c\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\texttt{is-sb} \in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Computing memory write address $\mathsf{b\text{-}val} + \mathsf{c\text{-}val} \equiv \texttt{h-ram-base-addr}$

// $\texttt{h-carry}^{(j)}$ for $j = 1, 2, 3, 4$ used for carry handling

- $(\texttt{is-sb}) \cdot (\mathsf{b\text{-}val}^{(1)} + \mathsf{c\text{-}val}^{(1)} - \texttt{h-ram-base-addr}^{(1)} - \texttt{h-carry}^{(1)} \cdot 2^8) = 0$
- $(\texttt{is-sb}) \cdot (\mathsf{b\text{-}val}^{(2)} + \mathsf{c\text{-}val}^{(2)} + \texttt{h-carry}^{(1)} - \texttt{h-ram-base-addr}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^8) = 0$
- $(\texttt{is-sb}) \cdot (\mathsf{b\text{-}val}^{(3)} + \mathsf{c\text{-}val}^{(3)} + \texttt{h-carry}^{(2)} - \texttt{h-ram-base-addr}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^8) = 0$
- $(\texttt{is-sb}) \cdot (\mathsf{b\text{-}val}^{(4)} + \mathsf{c\text{-}val}^{(4)} + \texttt{h-carry}^{(3)} - \texttt{h-ram-base-addr}^{(4)} - \texttt{h-carry}^{(4)} \cdot 2^8) = 0$

// Enforcing $\texttt{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\texttt{is-sb}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-sb}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-sb}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-sb}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// Extracting ram-val1 from bits 0-7 of $\mathsf{a\text{-}val}$

- $(\texttt{is-sb}) \cdot (\mathsf{a\text{-}val}^{(1)} - \texttt{ram-val1}) = 0$

// $\texttt{h-ram-base-addr} = (\texttt{h-ram-base-addr}^{(1)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(4)})$

// Writing byte ram-val1 at memory address h-ram-base-addr

- $(\texttt{is-sb}) \cdot (\mathrm{Write}_{\mathrm{RAM}}(\texttt{clk}, \texttt{ram-val1}, \texttt{h-ram-base-addr}, 0)) = 0$

// Range check $\texttt{h-ram-base-addr}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the RAM memory checking

// Range check $\texttt{ram-val1} \in \left[0, 2^8 - 1\right]$ - Follows from $\mathsf{a\text{-}val}^{(1)}$ constraint

// Range check $\mathsf{a\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check $\mathsf{b\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check $\mathsf{c\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check $\texttt{is-sb} \in \{0, 1\}$ - Performed in the CPU component

8.8.2 SH Instruction

The parameters and functionality for the SH instruction are as follows:

- opcode: SH
- Parameters: $(\mathsf{a\text{-}val}, \mathsf{b\text{-}val}, \mathsf{c\text{-}val})$
- Instruction selector: $\texttt{is-sh} = 1$

- Functionality:
  - base-addr $:=$ $\mathsf{b\text{-}val} + \mathsf{c\text{-}val} \bmod 2^{32}$
  - $M[\text{base-addr}] \leftarrow \mathsf{a\text{-}val} \;\&\; \mathtt{0x000000FF}$
  - $M[\text{base-addr} + 1] \leftarrow \mathsf{a\text{-}val} \;\&\; \mathtt{0x0000FF00}$
- Requirements: base-addr needs to be a *multiple of 2* for memory alignment

The mapping from the **sh** instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
sh	rs1	rs2	i	1	R[rs1]	R[rs2]	sext(i)

where sext is the sign extension function, i is a 12-bit value specifying bits 0-11 of the immediate value.

Constraints assuming large fields

// Computing memory write address $\mathsf{b\text{-}val} + \mathsf{c\text{-}val} \equiv \texttt{h-ram-base-addr} \cdot 2$
// h-ram-base-addr used to enforce memory alignment

- $(\texttt{is-sh}) \cdot (\mathsf{b\text{-}val} + \mathsf{c\text{-}val} - 2 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-carry} \cdot 2^{32}) = 0$
- $(\texttt{is-sh}) \cdot (2 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-ram-base-addr}) = 0$

// Enforcing $\texttt{h-carry} \in \{0, 1\}$

- $(\texttt{is-sh}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// Enforcing $\texttt{h-ram-base-addr-aux} \in \left[0, 2^{31} - 1\right]$

- $(\texttt{is-sh}) \cdot (\texttt{h-ram-base-addr-aux} \in \left[0, 2^{31} - 1\right]) = 0$

// Extracting $(\texttt{ram-val1}, \texttt{ram-val2})$ from bits 0-15 of $\mathsf{a\text{-}val}$

- $(\texttt{is-sh}) \cdot (\mathsf{a\text{-}val} - \texttt{h-ram-val-rem} \cdot 2^{16} - \texttt{ram-val2} \cdot 2^{8} - \texttt{ram-val1}) = 0$
- $(\texttt{is-sh}) \cdot (\texttt{ram-val1} \in \left[0, 2^{8} - 1\right]) = 0$
- $(\texttt{is-sh}) \cdot (\texttt{ram-val2} \in \left[0, 2^{8} - 1\right]) = 0$
- $(\texttt{is-sh}) \cdot (\texttt{h-ram-val-rem} \in \left[0, 2^{16} - 1\right]) = 0$

// $\texttt{h-ram-base-addr} = (\texttt{h-ram-base-addr}^{(1)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(4)})$
// Writing byte ram-val1 at memory address h-ram-base-addr
// Writing byte ram-val2 at memory address $\texttt{h-ram-base-addr} + 1$

- $(\texttt{is-sh}) \cdot (\text{Write}_{\text{RAM}}(\texttt{clk}, \texttt{ram-val1}, \texttt{h-ram-base-addr}, 0)) = 0$
- $(\texttt{is-sh}) \cdot (\text{Write}_{\text{RAM}}(\texttt{clk}, \texttt{ram-val2}, \texttt{h-ram-base-addr}, 1)) = 0$

// Range check $\texttt{h-ram-base-addr} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the RAM memory checking
// Range check $\texttt{ram-val}i \in \left[0, 2^{8} - 1\right]$ for $i = 1, 2$ - Also performed in the RAM component
// Range check $\mathsf{a\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\mathsf{b\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\mathsf{c\text{-}val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\texttt{is-sh} \in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Computing memory write address $\mathsf{b\text{-}val} + \mathsf{c\text{-}val} \equiv \texttt{h-ram-base-addr} \cdot 2$
// h-ram-base-addr used to enforce memory alignment
// $\texttt{h-carry}^{(j)}$ for $j = 1, 2, 3, 4$ used for carry handling

- $(\texttt{is-sh}) \cdot (\mathsf{b\text{-}val}^{(1)} + \mathsf{c\text{-}val}^{(1)} - 2 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-carry}^{(1)} \cdot 2^{8}) = 0$
- $(\texttt{is-sh}) \cdot (2 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-ram-base-addr}^{(1)}) = 0$
- $(\texttt{is-sh}) \cdot (\mathsf{b\text{-}val}^{(2)} + \mathsf{c\text{-}val}^{(2)} + \texttt{h-carry}^{(1)} - \texttt{h-ram-base-addr}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^{8}) = 0$
- $(\texttt{is-sh}) \cdot (\mathsf{b\text{-}val}^{(3)} + \mathsf{c\text{-}val}^{(3)} + \texttt{h-carry}^{(2)} - \texttt{h-ram-base-addr}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^{8}) = 0$
- $(\texttt{is-sh}) \cdot (\mathsf{b\text{-}val}^{(4)} + \mathsf{c\text{-}val}^{(4)} + \texttt{h-carry}^{(3)} - \texttt{h-ram-base-addr}^{(4)} - \texttt{h-carry}^{(3)} \cdot 2^{8}) = 0$

// Enforcing $\texttt{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\texttt{is-sh}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-sh}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-sh}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-sh}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// Enforcing $\texttt{h-ram-base-addr-aux} \in \left[0, 2^7 - 1\right]$

- $(\texttt{is-sh}) \cdot (\texttt{h-ram-base-addr-aux} \in \left[0, 2^7 - 1\right]) = 0$

// Extracting $(\texttt{ram-val1}, \texttt{ram-val2})$ from bits 0-15 of $\mathsf{a\text{-}val}$

- $(\texttt{is-sh}) \cdot (\mathsf{a\text{-}val}^{(1)} - \texttt{ram-val1}) = 0$
- $(\texttt{is-sh}) \cdot (\mathsf{a\text{-}val}^{(2)} - \texttt{ram-val2}) = 0$

// $\texttt{h-ram-base-addr} = (\texttt{h-ram-base-addr}^{(1)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(4)})$
// Writing byte ram-val1 at memory address h-ram-base-addr
// Writing byte ram-val2 at memory address $\texttt{h-ram-base-addr} + 1$

- $(\texttt{is-sh}) \cdot (\mathrm{Write}_{\mathrm{RAM}}(\texttt{clk}, \texttt{ram-val1}, \texttt{h-ram-base-addr}, 0)) = 0$
- $(\texttt{is-sh}) \cdot (\mathrm{Write}_{\mathrm{RAM}}(\texttt{clk}, \texttt{ram-val1}, \texttt{h-ram-base-addr}, 1)) = 0$

// Range check $\texttt{h-ram-base-addr}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the RAM memory checking

// Range check $\texttt{ram-val}j \in \left[0, 2^8 - 1\right]$ for $j = 1, 2$ - Performed in the RAM component

// Range check $\mathsf{a\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check $\mathsf{b\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check $\mathsf{c\text{-}val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check $\texttt{is-sh} \in \{0, 1\}$ - Performed in the CPU component

8.8.3 SW Instruction

The parameters and functionality for the SW instruction are as follows:

- opcode: SW
- Parameters: $(\mathsf{a\text{-}val}, \mathsf{b\text{-}val}, \mathsf{c\text{-}val})$
- Instruction selector: $\texttt{is-sw} = 1$
- Functionality:
  - $\text{base-addr} := \mathsf{b\text{-}val} + \mathsf{c\text{-}val} \bmod 2^{32}$
  - $M[\text{base-addr}] \leftarrow \mathsf{a\text{-}val}\ \&\ \mathsf{0x000000FF}$
  - $M[\text{base-addr} + 1] \leftarrow \mathsf{a\text{-}val}\ \&\ \mathsf{0x0000FF00}$
  - $M[\text{base-addr} + 2] \leftarrow \mathsf{a\text{-}val}\ \&\ \mathsf{0x00FF0000}$
  - $M[\text{base-addr} + 3] \leftarrow \mathsf{a\text{-}val}\ \&\ \mathsf{0xFF000000}$
- Requirements: base-addr needs to be a *multiple of 4* for memory alignment

The mapping from the sw instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
sw	rs1	rs2	i	1	$R[\text{rs1}]$	$R[\text{rs2}]$	$\text{sext}(i)$

where sext is the sign extension function, i is a 12-bit value specifying bits 0-11 of the immediate value.

Constraints assuming large fields

// Computing memory write address $\mathsf{b\text{-}val} + \mathsf{c\text{-}val} \equiv \texttt{h-ram-base-addr} \cdot 4$
// h-ram-base-addr used to enforce memory alignment

- $(\texttt{is-sw}) \cdot (\mathsf{b\text{-}val} + \mathsf{c\text{-}val} - 4 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-carry} \cdot 2^{32}) = 0$
- $(\texttt{is-sw}) \cdot (4 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-ram-base-addr}) = 0$

// Enforcing $\texttt{h-carry} \in \{0, 1\}$

- $(\texttt{is-sw}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// Enforcing $\texttt{h-ram-base-addr-aux} \in \left[0, 2^{30} - 1\right]$

- $(\texttt{is-sw}) \cdot (\texttt{h-ram-base-addr-aux} \in \left[0, 2^{30} - 1\right]) = 0$

// Extracting $(\texttt{ram-val1}, \texttt{ram-val2}, \texttt{ram-val3}, \texttt{ram-val4})$ from $\textsf{a-val}$

- $(\texttt{is-sw}) \cdot (\textsf{a-val} - \texttt{ram-val4} \cdot 2^{24} - \texttt{ram-val3} \cdot 2^{16} - \texttt{ram-val2} \cdot 2^{8} - \texttt{ram-val1}) = 0$
- $(\texttt{is-sw}) \cdot (\texttt{ram-val1} \in \left[0, 2^{8} - 1\right]) = 0$
- $(\texttt{is-sw}) \cdot (\texttt{ram-val2} \in \left[0, 2^{8} - 1\right]) = 0$
- $(\texttt{is-sw}) \cdot (\texttt{ram-val3} \in \left[0, 2^{8} - 1\right]) = 0$
- $(\texttt{is-sw}) \cdot (\texttt{ram-val4} \in \left[0, 2^{8} - 1\right]) = 0$

// $\texttt{h-ram-base-addr} = (\texttt{h-ram-base-addr}^{(1)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(4)})$
// Writing byte ram-val1 at memory address h-ram-base-addr
// Writing byte ram-val2 at memory address $\texttt{h-ram-base-addr} + 1$
// Writing byte ram-val3 at memory address $\texttt{h-ram-base-addr} + 2$
// Writing byte ram-val4 at memory address $\texttt{h-ram-base-addr} + 3$

- $(\texttt{is-sw}) \cdot (\text{Write}_{\text{RAM}}(\texttt{clk}, \texttt{ram-val1}, \texttt{h-ram-base-addr}, 0)) = 0$
- $(\texttt{is-sw}) \cdot (\text{Write}_{\text{RAM}}(\texttt{clk}, \texttt{ram-val2}, \texttt{h-ram-base-addr}, 1)) = 0$
- $(\texttt{is-sw}) \cdot (\text{Write}_{\text{RAM}}(\texttt{clk}, \texttt{ram-val3}, \texttt{h-ram-base-addr}, 2)) = 0$
- $(\texttt{is-sw}) \cdot (\text{Write}_{\text{RAM}}(\texttt{clk}, \texttt{ram-val4}, \texttt{h-ram-base-addr}, 3)) = 0$

// Range check $\texttt{h-ram-base-addr} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the RAM memory checking

// Range check $\texttt{ram-val}j \in \left[0, 2^{8} - 1\right]$ for $j = 1, 2, 3, 4$ - Also performed in the RAM component

// Range check $\textsf{a-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\textsf{b-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\textsf{c-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\texttt{is-sw} \in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Computing memory write address $\textsf{b-val} + \textsf{c-val} \equiv \texttt{h-ram-base-addr} \cdot 4$
// h-ram-base-addr used to enforce memory alignment
// $\texttt{h-carry}^{(j)}$ for $j = 1, 2, 3, 4$ used for carry handling

- $(\texttt{is-sw}) \cdot (\textsf{b-val}^{(1)} + \textsf{c-val}^{(1)} - 4 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-carry}^{(1)} \cdot 2^{8}) = 0$
- $(\texttt{is-sw}) \cdot (4 \cdot \texttt{h-ram-base-addr-aux} - \texttt{h-ram-base-addr}^{(1)}) = 0$
- $(\texttt{is-sw}) \cdot (\textsf{b-val}^{(2)} + \textsf{c-val}^{(2)} + \texttt{h-carry}^{(1)} - \texttt{h-ram-base-addr}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^{8}) = 0$
- $(\texttt{is-sw}) \cdot (\textsf{b-val}^{(3)} + \textsf{c-val}^{(3)} + \texttt{h-carry}^{(2)} - \texttt{h-ram-base-addr}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^{8}) = 0$
- $(\texttt{is-sw}) \cdot (\textsf{b-val}^{(4)} + \textsf{c-val}^{(4)} + \texttt{h-carry}^{(3)} - \texttt{h-ram-base-addr}^{(4)} - \texttt{h-carry}^{(3)} \cdot 2^{8}) = 0$

// Enforcing $\texttt{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\texttt{is-sw}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-sw}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-sw}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-sw}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// Enforcing $\texttt{h-ram-base-addr-aux} \in \left[0, 2^{6} - 1\right]$

- $(\texttt{is-sw}) \cdot (\texttt{h-ram-base-addr-aux} \in \left[0, 2^{6} - 1\right]) = 0$

// Extracting $(\texttt{ram-val1}, \texttt{ram-val2}, \texttt{ram-val3}, \texttt{ram-val4})$ from $\textsf{a-val}$

- $(\texttt{is-sw}) \cdot (\textsf{a-val}^{(1)} - \texttt{ram-val1}) = 0$
- $(\texttt{is-sw}) \cdot (\textsf{a-val}^{(2)} - \texttt{ram-val2}) = 0$
- $(\texttt{is-sw}) \cdot (\textsf{a-val}^{(3)} - \texttt{ram-val3}) = 0$
- $(\texttt{is-sw}) \cdot (\textsf{a-val}^{(4)} - \texttt{ram-val4}) = 0$

// $\texttt{h-ram-base-addr} = (\texttt{h-ram-base-addr}^{(1)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(3)}, \texttt{h-ram-base-addr}^{(4)})$
// Writing byte ram-val1 at memory address h-ram-base-addr
// Writing byte ram-val2 at memory address $\texttt{h-ram-base-addr} + 1$
// Writing byte ram-val3 at memory address $\texttt{h-ram-base-addr} + 2$
// Writing byte ram-val4 at memory address $\texttt{h-ram-base-addr} + 3$

- $(\texttt{is-sw}) \cdot (\text{Write}_{\text{RAM}}(\texttt{clk}, \texttt{ram-val1}, \texttt{h-ram-base-addr}, 0)) = 0$
- $(\texttt{is-sw}) \cdot (\text{Write}_{\text{RAM}}(\texttt{clk}, \texttt{ram-val1}, \texttt{h-ram-base-addr}, 1)) = 0$

- $(\texttt{is-sw}) \cdot (\text{Write}_{\text{RAM}}(\texttt{clk}, \texttt{ram-val3}, \texttt{h-ram-base-addr}, 2)) = 0$
- $(\texttt{is-sw}) \cdot (\text{Write}_{\text{RAM}}(\texttt{clk}, \texttt{ram-val4}, \texttt{h-ram-base-addr}, 3)) = 0$

// Range check h-ram-base-addr$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the RAM memory checking

// Range check ram-val$j \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the RAM component

// Range check a-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check b-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check c-val$^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check is-sw $\in \{0, 1\}$ - Performed in the CPU component

8.9 Basic Instruction Set: Jump Instructions

8.9.1 JAL Instruction

The parameters and functionality for the JAL instruction are as follows:

- opcode: JAL
- Parameters: $(\textsf{a-val}, \textsf{b-val}, \textsf{c-val})$
- Instruction selector: $\texttt{is-jal} = 1$
- Functionality:
  - $\text{pc-}\textsf{next} \leftarrow \text{pc} + \textsf{c-val} \bmod 2^{32}$
  - $\textsf{a-val} \leftarrow \text{pc} + \textsf{c-val} \bmod 2^{32}$

The mapping from the jal instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
jal	rd	0	i	1	R[rd]	0	sext(i)

where sext is the sign extension function, i is a 20-bit value specifying bits 1-20 of the immediate value, and bit 0 of the immediate value is equal to 0.

Constraints assuming large fields

// Setting a-val
- $(\texttt{is-jal}) \cdot (4 + \text{pc} - \textsf{a-val} - \texttt{h-carry} \cdot 2^{32}) = 0$

// Enforcing h-carry $\in \{0, 1\}$
- $(\texttt{is-jal}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// Setting pc-next
// $\text{pc-}\textsf{next} = \text{pc} + \textsf{c-val}$
- $(\texttt{is-jal}) \cdot (\textsf{c-val} + \text{pc} - \text{pc-}\textsf{next} - \texttt{pc-carry} \cdot 2^{32}) = 0$

// Enforcing pc-carry $\in \{0, 1\}$
- $(\texttt{is-jal}) \cdot (\texttt{pc-carry}) \cdot (1 - \texttt{pc-carry}) = 0$

// Range check pc $\in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking

// Range check pc-next $\in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking

// Range check a-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check b-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check c-val $\in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check is-jal $\in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Setting $\texttt{a-val} = \mathrm{pc} + 4$
// $\texttt{h-carry}^{(j)}$ for $j = 1, 2, 3, 4$ used for carry handling

- $(\texttt{is-jal}) \cdot (\mathrm{pc}^{(1)} + 4 - \texttt{a-val}^{(1)} - \texttt{h-carry}^{(1)} \cdot 2^8) = 0$
- $(\texttt{is-jal}) \cdot (\mathrm{pc}^{(2)} + \texttt{h-carry}^{(1)} - \texttt{a-val}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^8) = 0$
- $(\texttt{is-jal}) \cdot (\mathrm{pc}^{(3)} + \texttt{h-carry}^{(2)} - \texttt{a-val}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^8) = 0$
- $(\texttt{is-jal}) \cdot (\mathrm{pc}^{(4)} + \texttt{h-carry}^{(3)} - \texttt{a-val}^{(4)} - \texttt{h-carry}^{(4)} \cdot 2^8) = 0$

// Enforcing $\texttt{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\texttt{is-jal}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-jal}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-jal}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-jal}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// Setting $\texttt{pc-next} = \mathrm{pc} + \texttt{c-val}$
// $\texttt{pc-carry}^{(j)}$ for $j = 1, 2, 3, 4$ used for carry handling

- $(\texttt{is-jal}) \cdot (\mathrm{pc}^{(1)} + \texttt{c-val}^{(1)} - \texttt{pc-next}^{(1)} - \texttt{pc-carry}^{(1)} \cdot 2^8) = 0$
- $(\texttt{is-jal}) \cdot (\mathrm{pc}^{(2)} + \texttt{c-val}^{(2)} + \texttt{pc-carry}^{(1)} - \texttt{pc-next}^{(2)} - \texttt{pc-carry}^{(2)} \cdot 2^8) = 0$
- $(\texttt{is-jal}) \cdot (\mathrm{pc}^{(3)} + \texttt{c-val}^{(3)} + \texttt{pc-carry}^{(2)} - \texttt{pc-next}^{(3)} - \texttt{pc-carry}^{(3)} \cdot 2^8) = 0$
- $(\texttt{is-jal}) \cdot (\mathrm{pc}^{(4)} + \texttt{c-val}^{(4)} + \texttt{pc-carry}^{(3)} - \texttt{pc-next}^{(4)} - \texttt{pc-carry}^{(4)} \cdot 2^8) = 0$

// Enforcing $\texttt{pc-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\texttt{is-jal}) \cdot (\texttt{pc-carry}^{(1)}) \cdot (1 - \texttt{pc-carry}^{(1)}) = 0$
- $(\texttt{is-jal}) \cdot (\texttt{pc-carry}^{(2)}) \cdot (1 - \texttt{pc-carry}^{(2)}) = 0$
- $(\texttt{is-jal}) \cdot (\texttt{pc-carry}^{(3)}) \cdot (1 - \texttt{pc-carry}^{(3)}) = 0$
- $(\texttt{is-jal}) \cdot (\texttt{pc-carry}^{(4)}) \cdot (1 - \texttt{pc-carry}^{(4)}) = 0$

// Range check $\mathrm{pc}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\texttt{pc-next}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\texttt{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\texttt{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\texttt{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\texttt{is-jal} \in \{0, 1\}$ - Performed in the CPU component

8.9.2 JALR Instruction

The parameters and functionality for the JALR instruction are as follows:

- opcode: JALR
- Parameters: $(\texttt{a-val}, \texttt{b-val}, \texttt{c-val})$
- Instruction selector: $\texttt{is-jalr} = 1$
- Functionality:
  - $\texttt{pc-next} \leftarrow (\texttt{b-val} + \texttt{c-val} \bmod 2^{32})\ \&\ \texttt{0xFFFFFFFE}$
  - $\texttt{a-val} \leftarrow \mathrm{pc} + 4 \bmod 2^{32}$

The mapping from the jalr instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
jalr	rd	rs1	i	1	R[rd]	R[rs1]	sext(i)

where sext is the sign extension function, i is a 12-bit value specifying bits 0-11 of the immediate value.

Constraints assuming large fields

// Setting $\texttt{a-val} = \mathrm{pc} + 4$

- $(\texttt{is-jalr}) \cdot (4 + \mathrm{pc} - \texttt{a-val} - \texttt{h-carry} \cdot 2^{32}) = 0$

// Enforcing $\texttt{h-carry} \in \{0, 1\}$

- $(\texttt{is-jalr}) \cdot (\texttt{h-carry}) \cdot (1 - \texttt{h-carry}) = 0$

// Setting pc-next-aux

// $\text{pc-next-aux} = \text{b-val} + \text{c-val}$

- $(\texttt{is-jalr}) \cdot (\text{b-val} + \text{c-val} - \text{pc-next-aux} - \texttt{pc-carry} \cdot 2^{32}) = 0$

// Enforcing $\texttt{pc-carry} \in \{0, 1\}$

- $(\texttt{is-jalr}) \cdot (\texttt{pc-carry}) \cdot (1 - \texttt{pc-carry}) = 0$

// Setting $\text{pc-next} = \text{pc-next-aux} \,\&\, \texttt{0xFFFFFFFE}$

// $\texttt{pc-rem-aux}$ and $\texttt{pc-qt-aux}$ used to set bit 0 of pc-next-aux to 0

- $(\texttt{is-jalr}) \cdot (\text{pc-next-aux} - \texttt{pc-rem-aux} - \texttt{pc-qt-aux} \cdot 2) = 0$
- $(\texttt{is-jalr}) \cdot (\texttt{pc-qt-aux} \cdot 2 - \text{pc-next}) = 0$

// Enforcing $\texttt{pc-rem-aux} \in \{0, 1\}$

- $(\texttt{is-jalr}) \cdot (\texttt{pc-rem-aux}) \cdot (1 - \texttt{pc-rem-aux}) = 0$

// Range check for pc-next-aux and $\texttt{pc-qt-aux}$

- $(\texttt{is-jalr}) \cdot (\text{pc-next-aux} \in \left[0, 2^{32} - 1\right]) = 0$
- $(\texttt{is-jalr}) \cdot (\texttt{pc-qt-aux} \in \left[0, 2^{31} - 1\right]) = 0$

// Range check $\text{pc} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking

// Range check $\text{pc-next} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking

// Range check $\text{a-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\text{b-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\text{c-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\texttt{is-jalr} \in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Setting $\text{a-val} = \text{pc} + 4$

// $\texttt{h-carry}^{(j)}$ for $j = 1, 2, 3, 4$ used for carry handling

- $(\texttt{is-jalr}) \cdot (\text{pc}^{(1)} + 4 \qquad\qquad - \text{a-val}^{(1)} - \texttt{h-carry}^{(1)} \cdot 2^8) = 0$
- $(\texttt{is-jalr}) \cdot (\text{pc}^{(2)} + \texttt{h-carry}^{(1)} - \text{a-val}^{(2)} - \texttt{h-carry}^{(2)} \cdot 2^8) = 0$
- $(\texttt{is-jalr}) \cdot (\text{pc}^{(3)} + \texttt{h-carry}^{(2)} - \text{a-val}^{(3)} - \texttt{h-carry}^{(3)} \cdot 2^8) = 0$
- $(\texttt{is-jalr}) \cdot (\text{pc}^{(4)} + \texttt{h-carry}^{(3)} - \text{a-val}^{(4)} - \texttt{h-carry}^{(4)} \cdot 2^8) = 0$

// Enforcing $\texttt{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\texttt{is-jalr}) \cdot (\texttt{h-carry}^{(1)}) \cdot (1 - \texttt{h-carry}^{(1)}) = 0$
- $(\texttt{is-jalr}) \cdot (\texttt{h-carry}^{(2)}) \cdot (1 - \texttt{h-carry}^{(2)}) = 0$
- $(\texttt{is-jalr}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-jalr}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// Setting $\text{pc-next-aux} = \text{b-val} + \text{c-val}$

// $\texttt{pc-carry}^{(j)}$ for $j = 1, 2, 3, 4$ used for carry handling

- $(\texttt{is-jalr}) \cdot (\text{b-val}^{(1)} + \text{c-val}^{(1)} - \text{pc-next-aux}^{(1)} - \texttt{pc-carry}^{(1)} \cdot 2^8) = 0$
- $(\texttt{is-jalr}) \cdot (\text{b-val}^{(2)} + \text{c-val}^{(2)} + \texttt{pc-carry}^{(1)} - \text{pc-next-aux}^{(2)} - \texttt{pc-carry}^{(2)} \cdot 2^8) = 0$
- $(\texttt{is-jalr}) \cdot (\text{b-val}^{(3)} + \text{c-val}^{(3)} + \texttt{pc-carry}^{(2)} - \text{pc-next-aux}^{(3)} - \texttt{pc-carry}^{(3)} \cdot 2^8) = 0$
- $(\texttt{is-jalr}) \cdot (\text{b-val}^{(4)} + \text{c-val}^{(4)} + \texttt{pc-carry}^{(3)} - \text{pc-next-aux}^{(4)} - \texttt{pc-carry}^{(4)} \cdot 2^8) = 0$

// Enforcing $\texttt{pc-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$

- $(\texttt{is-jalr}) \cdot (\texttt{pc-carry}^{(1)}) \cdot (1 - \texttt{pc-carry}^{(1)}) = 0$
- $(\texttt{is-jalr}) \cdot (\texttt{pc-carry}^{(2)}) \cdot (1 - \texttt{pc-carry}^{(2)}) = 0$
- $(\texttt{is-jalr}) \cdot (\texttt{pc-carry}^{(3)}) \cdot (1 - \texttt{pc-carry}^{(3)}) = 0$
- $(\texttt{is-jalr}) \cdot (\texttt{pc-carry}^{(4)}) \cdot (1 - \texttt{pc-carry}^{(4)}) = 0$

// Setting $\text{pc-next} = \text{pc-next-aux} \,\&\, \texttt{0xFFFFFFFE}$

// $\texttt{pc-rem-aux}$ and $\texttt{pc-qt-aux}$ used to set bit 0 of $\text{pc-next-aux}^{(1)}$ to 0

- $(\texttt{is-jalr}) \cdot (\text{pc-next-aux}^{(1)} - \texttt{pc-rem-aux} - \texttt{pc-qt-aux} \cdot 2) = 0$
- $(\texttt{is-jalr}) \cdot (\texttt{pc-qt-aux} \cdot 2 - \text{pc-next}^{(1)}) = 0$
- $(\texttt{is-jalr}) \cdot (\text{pc-next-aux}^{(2)} - \text{pc-next}^{(2)}) = 0$
- $(\texttt{is-jalr}) \cdot (\text{pc-next-aux}^{(3)} - \text{pc-next}^{(3)}) = 0$
- $(\texttt{is-jalr}) \cdot (\text{pc-next-aux}^{(4)} - \text{pc-next}^{(4)}) = 0$

// Enforcing $\texttt{pc-rem-aux} \in \{0, 1\}$

- $(\texttt{is-jalr}) \cdot (\texttt{pc-rem-aux}) \cdot (1 - \texttt{pc-rem-aux}) = 0$

// Range check for pc-next-aux and pc-qt-aux
- $(\texttt{is-jalr}) \cdot (\text{pc-next-aux}^{(j)} \in \left[0, 2^8 - 1\right]) = 0$ for $j = 1, 2, 3, 4$
- $(\texttt{is-jalr}) \cdot (\texttt{pc-qt-aux} \in \left[0, 2^8 - 1\right]) = 0$

// Range check $\text{pc}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking

// Range check $\text{pc-next}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking

// Range check $\text{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check $\text{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check $\text{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component

// Range check $\texttt{is-jalr} \in \{0, 1\}$ - Performed in the CPU component

8.10 Basic Instruction Set: LUI and AUIPC Instructions

8.10.1 LUI Instruction

The parameters and functionality for the LUI instruction are as follows:

- opcode: LUI
- Parameters: (a-val, b-val, c-val)
- Instruction selector: $\texttt{is-lui} = 1$
- Functionality:
  - $\text{pc-next} \leftarrow \text{pc} + 4 \bmod 2^{32}$
  - $\text{a-val} \leftarrow \text{c-val}$

The mapping from the lui instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
lui	rd	0	i	1	R[rd]	0	sext(i)

where sext is the sign extension function, i is a 20-bit value specifying bits 12-31 of the immediate value, and bits 0-11 of the immediate value are equal to 0.

Constraints assuming large fields

// Setting a-val = c-val
- $(\texttt{is-lui}) \cdot (\text{c-val} - \text{a-val}) = 0$

// Setting $\text{pc-next} = \text{pc} + 4$ is performed in Section 8.4.1 under common constraints

// Range check $\text{pc} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking

// Range check $\text{pc-next} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking

// Range check $\text{a-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\text{b-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\text{c-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\texttt{is-lui} \in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Setting a-val = c-val
- $(\texttt{is-lui}) \cdot (\text{c-val}^{(1)} - \text{a-val}^{(1)}) = 0$
- $(\texttt{is-lui}) \cdot (\text{c-val}^{(2)} - \text{a-val}^{(2)}) = 0$
- $(\texttt{is-lui}) \cdot (\text{c-val}^{(3)} - \text{a-val}^{(3)}) = 0$
- $(\texttt{is-lui}) \cdot (\text{c-val}^{(4)} - \text{a-val}^{(4)}) = 0$

// Setting $\text{pc-next} = \text{pc} + 4$ is performed in Section 8.4.2 under common constraints
// Range check $\text{pc}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\text{pc-next}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\text{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\text{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\text{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check is-lui $\in \{0, 1\}$ - Performed in the CPU component

8.10.2 AUIPC Instruction

The parameters and functionality for the AUIPC instruction are as follows:

- opcode: AUIPC
- Parameters: (a-val, b-val, c-val)
- Instruction selector: is-auipc $= 1$
- Functionality:
  - pc-next $\leftarrow \text{pc} + 4 \bmod 2^{32}$
  - a-val $\leftarrow \text{pc} + \text{c-val} \bmod 2^{32}$

The mapping from the auipc instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
auipc	rd	0	i	1	$R[\text{rd}]$	0	$\text{sext}(i)$

where sext is the sign extension function, i is a 20-bit value specifying bits 12-31 of the immediate value, and bits 0-11 of the immediate value are equal to 0.

Constraints assuming large fields

// Setting $\text{a-val} = \text{pc} + \text{c-val}$
- $(\text{is-auipc}) \cdot (\text{c-val} + \text{pc} - \text{a-val} - \text{h-carry} \cdot 2^{32}) = 0$

// Enforcing h-carry $\in \{0, 1\}$
- $(\text{is-auipc}) \cdot (\text{h-carry}) \cdot (1 - \text{h-carry}) = 0$

// Setting $\text{pc-next} = \text{pc} + 4$ is performed in Section 8.4.1 under common constraints
// Range check $\text{pc} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking
// Range check $\text{pc-next} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking
// Range check $\text{a-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\text{b-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check $\text{c-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component
// Range check is-auipc $\in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Setting $\text{a-val} = \text{pc} + \text{c-val}$
// $\text{h-carry}^{(j)}$ for $j = 1, 2, 3, 4$ used for carry handling
- $(\text{is-auipc}) \cdot (\text{pc}^{(1)} + \text{c-val}^{(1)} - \text{a-val}^{(1)} - \text{h-carry}^{(1)} \cdot 2^8) = 0$
- $(\text{is-auipc}) \cdot (\text{pc}^{(2)} + \text{c-val}^{(2)} + \text{h-carry}^{(1)} - \text{a-val}^{(2)} - \text{h-carry}^{(2)} \cdot 2^8) = 0$
- $(\text{is-auipc}) \cdot (\text{pc}^{(3)} + \text{c-val}^{(3)} + \text{h-carry}^{(2)} - \text{a-val}^{(3)} - \text{h-carry}^{(3)} \cdot 2^8) = 0$
- $(\text{is-auipc}) \cdot (\text{pc}^{(4)} + \text{c-val}^{(4)} + \text{h-carry}^{(3)} - \text{a-val}^{(4)} - \text{h-carry}^{(4)} \cdot 2^8) = 0$

// Enforcing $\text{h-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$
- $(\text{is-auipc}) \cdot (\text{h-carry}^{(1)}) \cdot (1 - \text{h-carry}^{(1)}) = 0$
- $(\text{is-auipc}) \cdot (\text{h-carry}^{(2)}) \cdot (1 - \text{h-carry}^{(2)}) = 0$

- $(\texttt{is-auipc}) \cdot (\texttt{h-carry}^{(3)}) \cdot (1 - \texttt{h-carry}^{(3)}) = 0$
- $(\texttt{is-auipc}) \cdot (\texttt{h-carry}^{(4)}) \cdot (1 - \texttt{h-carry}^{(4)}) = 0$

// Setting $\text{pc-next} = \text{pc} + 4$ is performed in Section 8.4.2 under common constraints
// Range check $\text{pc}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\text{pc-next}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\text{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\text{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\text{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check is-auipc $\in \{0, 1\}$ - Performed in the CPU component

8.11 Basic Instruction Set: System Instructions

8.11.1 ECALL Instruction

The parameters and functionality for the ECALL instruction are as follows:

- opcode: ECALL
- Parameters: (a-val, b-val, c-val)
- Instruction selector: is-ecall = 1
- Functionality: see Table 3
- Remark: The ECALL behavior depends on the value of R[x17].

The mapping from the ecall instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
ecall	0 | x2 | x10	x17	0	1	R[op-a]	R[x17]	0

Constraints assuming large fields

// Setting flags depending on the value of b-val
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-debug})(\text{b-val} - \text{0x200}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-halt})(\text{b-val} - \text{0x201}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-priv-input})(\text{b-val} - \text{0x400}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-cycle-count})(\text{b-val} - \text{0x401}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-stack-reset})(\text{b-val} - \text{0x402}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-heap-reset})(\text{b-val} - \text{0x403}) = 0$

// Enforcing flags in $\{0, 1\}$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-debug}) \cdot (1 - \texttt{is-sys-debug}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-halt}) \cdot (1 - \texttt{is-sys-halt}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-priv-input}) \cdot (1 - \texttt{is-sys-priv-input}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-cycle-count}) \cdot (1 - \texttt{is-sys-cycle-count}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-stack-reset}) \cdot (1 - \texttt{is-sys-stack-reset}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-heap-reset}) \cdot (1 - \texttt{is-sys-heap-reset}) = 0$

// Enforcing that only one of these flags is set
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-debug} + \texttt{is-sys-halt} + \texttt{is-sys-priv-input} + \texttt{is-sys-cycle-count} + \texttt{is-sys-stack-reset} + \texttt{is-sys-heap-reset} - 1) = 0$

// Enforcing values for op-a
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-debug} + \texttt{is-sys-halt} + \texttt{is-sys-cycle-count}) \cdot (\text{op-a}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-priv-input} + \texttt{is-sys-heap-reset}) \cdot (10 - \text{op-a}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-stack-reset}) \cdot (2 - \text{op-a}) = 0$

// Enforcing ranges for a-val
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-debug} + \texttt{is-sys-halt} + \texttt{is-sys-cycle-count}) \cdot (\text{a-val}) = 0$

// The following range checks are being performed by the register memory component
// $(\texttt{is-type-s}) \cdot (\texttt{is-sys-priv-input} + \texttt{is-sys-heap-reset} + \texttt{is-sys-stack-reset}) \cdot (\text{a-val} \in \left[0, 2^{32} - 1\right]) = 0$

// Setting $\text{pc-next} = \text{pc} + 4$ when $\texttt{is-sys-halt} = 1$ or $\text{pc-next} = \text{pc} + 4$ for other flags

- $(\texttt{is-type-s}) \cdot (4 \cdot (1 - \texttt{is-sys-halt}) + \mathrm{pc} - \text{pc-next} - \text{pc-carry} \cdot 2^{32}) = 0$

// Enforcing $\text{pc-carry} \in \{0, 1\}$
- $(\texttt{is-type-s}) \cdot (\text{pc-carry}) \cdot (1 - \text{pc-carry}) = 0$

// Range check $\mathrm{pc} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking

// Range check $\text{pc-next} \in \left[0, 2^{32} - 1\right]$ - Guaranteed by the program memory checking

// Range check $\text{a-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\text{b-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\text{c-val} \in \left[0, 2^{32} - 1\right]$ - Performed in the CPU component

// Range check $\text{is-auipc} \in \{0, 1\}$ - Performed in the CPU component

Constraints assuming small fields

// Setting flags depending on the value of b-val
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-debug})(\text{b-val}^{(1)} - \texttt{0x00}) = 0$ ▷ $\text{b-val} = \texttt{0x200}$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-debug})(\text{b-val}^{(2)} - \texttt{0x02}) = 0$ ▷ $\text{b-val} = \texttt{0x200}$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-halt})(\text{b-val}^{(1)} - \texttt{0x01}) = 0$ ▷ $\text{b-val} = \texttt{0x201}$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-halt})(\text{b-val}^{(2)} - \texttt{0x02}) = 0$ ▷ $\text{b-val} = \texttt{0x201}$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-priv-input})(\text{b-val}^{(1)} - \texttt{0x00}) = 0$ ▷ $\text{b-val} = \texttt{0x400}$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-priv-input})(\text{b-val}^{(2)} - \texttt{0x04}) = 0$ ▷ $\text{b-val} = \texttt{0x400}$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-cycle-count})(\text{b-val}^{(1)} - \texttt{0x01}) = 0$ ▷ $\text{b-val} = \texttt{0x401}$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-cycle-count})(\text{b-val}^{(2)} - \texttt{0x04}) = 0$ ▷ $\text{b-val} = \texttt{0x401}$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-stack-reset})(\text{b-val}^{(1)} - \texttt{0x02}) = 0$ ▷ $\text{b-val} = \texttt{0x402}$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-stack-reset})(\text{b-val}^{(2)} - \texttt{0x04}) = 0$ ▷ $\text{b-val} = \texttt{0x402}$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-heap-reset})(\text{b-val}^{(1)} - \texttt{0x03}) = 0$ ▷ $\text{b-val} = \texttt{0x403}$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-heap-reset})(\text{b-val}^{(2)} - \texttt{0x04}) = 0$ ▷ $\text{b-val} = \texttt{0x403}$
- $(\texttt{is-type-s}) \cdot (\text{b-val}^{(3)}) = 0$
- $(\texttt{is-type-s}) \cdot (\text{b-val}^{(4)}) = 0$

// Enforcing flags in $\{0, 1\}$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-debug}) \cdot (1 - \texttt{is-sys-debug}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-halt}) \cdot (1 - \texttt{is-sys-halt}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-priv-input}) \cdot (1 - \texttt{is-sys-priv-input}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-cycle-count}) \cdot (1 - \texttt{is-sys-cycle-count}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-stack-reset}) \cdot (1 - \texttt{is-sys-stack-reset}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-heap-reset}) \cdot (1 - \texttt{is-sys-heap-reset}) = 0$

// Enforcing that only one of these flags is set
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-debug} + \texttt{is-sys-halt} + \texttt{is-sys-priv-input} +$
  $\texttt{is-sys-cycle-count} + \texttt{is-sys-stack-reset} + \texttt{is-sys-heap-reset} - 1) = 0$

// Enforcing values for op-a
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-debug} + \texttt{is-sys-halt} + \texttt{is-sys-cycle-count}) \cdot (\text{op-a}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-priv-input} + \texttt{is-sys-heap-reset}) \cdot (10 - \text{op-a}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-stack-reset}) \cdot (2 - \text{op-a}) = 0$

// Enforcing ranges for a-val
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-debug} + \texttt{is-sys-halt} + \texttt{is-sys-cycle-count}) \cdot (\text{a-val}^{(1)}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-debug} + \texttt{is-sys-halt} + \texttt{is-sys-cycle-count}) \cdot (\text{a-val}^{(2)}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-debug} + \texttt{is-sys-halt} + \texttt{is-sys-cycle-count}) \cdot (\text{a-val}^{(3)}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-debug} + \texttt{is-sys-halt} + \texttt{is-sys-cycle-count}) \cdot (\text{a-val}^{(4)}) = 0$

// The following range checks are being performed by the register memory component

// $(\texttt{is-type-s}) \cdot (\texttt{is-sys-priv-input} + \texttt{is-sys-heap-reset} + \texttt{is-sys-stack-reset}) \cdot (\text{a-val}^{(1)} \in \left[0, 2^8 - 1\right]) = 0$

// $(\texttt{is-type-s}) \cdot (\texttt{is-sys-priv-input} + \texttt{is-sys-heap-reset} + \texttt{is-sys-stack-reset}) \cdot (\text{a-val}^{(2)} \in \left[0, 2^8 - 1\right]) = 0$

// $(\texttt{is-type-s}) \cdot (\texttt{is-sys-priv-input} + \texttt{is-sys-heap-reset} + \texttt{is-sys-stack-reset}) \cdot (\text{a-val}^{(3)} \in \left[0, 2^8 - 1\right]) = 0$

// $(\texttt{is-type-s}) \cdot (\texttt{is-sys-priv-input} + \texttt{is-sys-heap-reset} + \texttt{is-sys-stack-reset}) \cdot (\text{a-val}^{(4)} \in \left[0, 2^8 - 1\right]) = 0$

// Setting $\text{pc-next} = \mathrm{pc} + 4$ when $\texttt{is-sys-halt} = 1$ or $\text{pc-next} = \mathrm{pc} + 4$ for other flags

// $\text{pc-carry}^{(j)}$ for $j = 1, 2, 3, 4$ used for carry handling
- $(\texttt{is-type-s}) \cdot (\mathrm{pc}^{(1)} + 4 \cdot (1 - \texttt{is-sys-halt}) - \text{pc-next}^{(1)} - \text{pc-carry}^{(1)} \cdot 2^8) = 0$

- $(\texttt{is-type-s}) \cdot (\mathrm{pc}^{(2)} + \texttt{pc-carry}^{(1)} - \texttt{pc-next}^{(2)} - \texttt{pc-carry}^{(2)} \cdot 2^8) = 0$
- $(\texttt{is-type-s}) \cdot (\mathrm{pc}^{(3)} + \texttt{pc-carry}^{(2)} - \texttt{pc-next}^{(3)} - \texttt{pc-carry}^{(3)} \cdot 2^8) = 0$
- $(\texttt{is-type-s}) \cdot (\mathrm{pc}^{(4)} + \texttt{pc-carry}^{(3)} - \texttt{pc-next}^{(4)} - \texttt{pc-carry}^{(4)} \cdot 2^8) = 0$

// Enforcing $\texttt{pc-carry}^{(j)} \in \{0, 1\}$ for $j = 1, 2, 3, 4$
- $(\texttt{is-type-s}) \cdot (\texttt{pc-carry}^{(1)}) \cdot (1 - \texttt{pc-carry}^{(1)}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{pc-carry}^{(2)}) \cdot (1 - \texttt{pc-carry}^{(2)}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{pc-carry}^{(3)}) \cdot (1 - \texttt{pc-carry}^{(3)}) = 0$
- $(\texttt{is-type-s}) \cdot (\texttt{pc-carry}^{(4)}) \cdot (1 - \texttt{pc-carry}^{(4)}) = 0$

// Range check $\mathrm{pc}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\texttt{pc-next}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Guaranteed by the program memory checking
// Range check $\texttt{a-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\texttt{b-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\texttt{c-val}^{(j)} \in \left[0, 2^8 - 1\right]$ for $j = 1, 2, 3, 4$ - Performed in the CPU component
// Range check $\texttt{is-type-s} \in \{0, 1\}$ - Performed in the CPU component

8.11.2 EBREAK Instruction

The parameters and functionality for the EBREAK instruction are as follows:

- opcode: EBREAK
- Parameters: (a-val, b-val, c-val)
- Instruction selector: is-ebreak = 1
- Functionality: see Table 3
- Remark: The EBREAK behavior depends on the value of R[x17].

The mapping from the ebreak instruction in the Nexus Virtual Machine Instruction Set Table (see Table 1) is as follows:

NVM opcode	op-a	op-b	op-c	imm-c	a-val	b-val	c-val
ebreak	0 | x2 | x10	x17	0	1	R[op-a]	R[x17]	0

Remark 8.3 Since the current version of the Nexus Virtual Machine behaves similarly for EBREAK and ECALL instructions, we describe these constraints together in the ECALL section (Section 8.11.1) using the flag is-type-sys instead of is-ecall or is-ebreak.

8.12 Interactions with other components

As mentioned in Section 4.2.4, in the case of system instructions, the execution component may end up writing to the register memory component depending on the contents of register R[x17]. More precisely, as stated in Table 3,

- When R[x17] = 0x400, the system call reads from the private input and loads a 32-bit value onto R[x10];
- When R[x17] = 0x400, the system call overwrites the stack pointer and loads a 32-bit value onto R[x2]; and
- When R[x17] = 0x400, the system call overwrites the heap pointer and loads a 32-bit value onto R[x10].

For this reason, we specify in the interaction with the register memory component explicitly here.

Register memory interaction

// Writing to register op-a = x2 for TYPE SYS instructions when is-sys-stack-reset = 1
// Writing to register op-a = x10 for TYPE SYS instructions when is-sys-priv-input = 1
// Writing to register op-a = x10 for TYPE SYS instructions when is-sys-heap-reset = 1

- $(\texttt{is-type-s}) \cdot (\texttt{is-sys-priv-input} + \texttt{is-sys-heap-reset} + \texttt{is-sys-stack-reset}) \cdot \mathrm{Write}_{\mathrm{Reg}}(\texttt{op-a}, \texttt{op-a}_{\texttt{val}}, \texttt{clk}, 1)) = 0$

References

[AST24] Arasu Arun, Srinath T. V. Setty, and Justin Thaler. Jolt: SNARKs for virtual machines via lookups. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part VI*, volume 14656 of *LNCS*, pages 3–33. Springer, Cham, May 2024.

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018.

[BEG$^+$94] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

[EKRN24] Liam Eagen, Sanket Kanjalkar, Tim Ruffing, and Jonas Nick. Bulletproofs++: Next generation confidential transactions via reciprocal set membership arguments. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part V*, volume 14655 of *LNCS*, pages 249–279. Springer, Cham, May 2024.

[GPR21] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a Turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Report 2021/1063, 2021.

[Hab22] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Report 2022/1530, 2022.

[HLP24] Ulrich Haböck, David Levit, and Shahar Papini. Circle STARKs. Cryptology ePrint Archive, Report 2024/278, 2024.

[Mid] Polygon Miden VM. https://0xpolygonmiden.github.io/miden-vm/design/range.html.

[Nex24] Nexus Laboratories, Inc. *Nexus 1.0: Enabling Verifiable Computation*, January 2024. Authors D. Marin, M. Abdalla, P. Govereau, J. Groth, S. Judson, K. Sosnin, G.-V. Policharla, and Yinuo Zhang. Available at https://whitepaper.nexus.xyz.

[RIS19] RISC-V Foundation. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, Document Version 20191213*, December 2019. Editors Andrew Waterman and Krste Asanovic. Available at https://drive.google.com/file/d/1s0lZxUZaa7eV_O0_WsZzaurFLLww7ou5/view.

[SIE21] Sieve: Securing information for encrypted verification and evaluation. DARPA Program. https://www.darpa.mil/research/programs/securing-information-for-encrypted-verification-and-evaluation, 2021.

[STW24] Stwo Prover: The next-gen of STARK scaling is here. https://starkware.co/blog/stwo-prover-the-next-gen-of-stark-scaling-is-here/, February 2024.