GoblinCore64: Architectural Specification Technical Report 2015-001

John D. Leidel, Xi Wang

March 7, 2015

Version:	0.3PRELEASE
Texas Tech University	
Faculty Mentor	Dr. Yong Chen
Data Intensive Scalable Computing Laboratory	

Contents

1 Introduction

1.1 GC64 Overview

The GoblinCore-64 (herein referred to as GC64) that was originally designed to facilitate the construction of a high performance core architecture that was well-suited to executing applications traditionally known as "data intensive." These applications generally refer to algorithms that operate on sparse data structures such as graphs, sparse matrices and/or perform nonlinear combinatorial operations (et.al.). We consider all of the aforementioned target application areas to share the following two general characteristics.

- **Non-Unit Stride**: All of the applications we consider as design targets for GC64 perform a disproportionate number of non-unit stride computations. These computations may simply be non-unit stride, scatters, gathers or completely random. In all cases, the data elements are not generally well-suited to traditional long SIMD or data caching architectures.
- **Memory Intensive**: Given the first characteristic, we also assume a latent characteristic with respect to the memory bandwidth requirements. Given the sparsity or non-linear access requirements, we assume that the design targets operate with a disproportionally high bandwidth to compute ratio. As such, we consider them to be memory intensive rather than computationally intensive.

In addition to the core design requirements, we also sought to build a completely open source architecture and tool chain suitable for architectural research in academia and possible commercial implementations. As such, we sought to build BSD-like licensing around the core ISA, simulation infrastructure, tools and toolchain. Given this, we found that our implementation goals aligned well with the RISC-V project [2]. These include, but are not limited to the following:

- A completely *open* ISA that is freely available to academia and industry.
- An ISA separated into a *small* base integer ISA.
- Support for the revised 2008 IEEE-754 [1] floating-point standard.
- An ISA with *native* support for highly-parallel multicore or many core implementations.

In addition to the core RISC-V goals, we also wanted to achieve the following architectural goals (as related to our target design requirements):

- Provide simple architectural structures that are conducive to constructing highly (MIMD) parallel and concurrent applications
- Provide simple ISA extensions conducive to compiler optimization of concurrent applications

- Provide a low-level, mutable parallel construct in hardware that can be easily mapped to higher level parallel programming models (threads, tasks, etc)
- Provide hardware mechanisms to minimize context switch latency to a very small number of cycles (goal of single cycle context switching events)
- Provide a well-defined mechanism when context switch events occur
- Provide a well-defined mechanism for user applications to explicitly induce context switch events

1.2 RISC-V ISA Requirements

The GC64 RISC-V extension functions as both a core RISC-V [2] architectural extension and an extension to the standard 64-bit RISC-V IMAFD specification. The following minimum architectural extensions are required to utilize GC64.

- **RV64I**: GC64 requires 64-bit addressing at minimum. The RV32I memory model is supported as a result. However, given the use of HMC devices, we require the use of at least 64-bit addressing (and addressing arithmetic).
- **M-Extension**: GC64 requires the integer multiplication and division for the purpose of efficiently performing address manipulation.
- **A-Extension**: GC64 requires the atomic instructions.

The following architectural extensions are supported in the GC64 tasking model. They are herein named as optional extensions.

- **F-Extension**: By default, the GC64 task context definition supports the storage of the single precision floating point register values. If the F-extension is not present, these values are ignored.
- **D-Extension**: By default, the GC64 task context definition supports the storage of the double precision floating point register values. If the D-extension is not present, these values are ignored.
- **RV128I**: By default, the GC64 reuqires 64-bit addressing. However, when building scalable versions of GC64 in large SMP or PGAS modes, it may be desirable to support 128bit addressing.

1.3 GC64 SoC Architecture

1.3.1 Overview

The GC64 SoC architecture consists of six hierarchical units that are explicitly organized to provide high degrees of local and global concurrency without sacrificing performance. The hierarchy of architectural units organized from smallest

(least encompassing) to the largest (most encompassing) is depicted in the table below. Note that each subsequent layer encompasses one or more units from the previous layer. At least one of the lower four architectural layers must be included in all GC64 configurations.

- Task Unit [required]
- Task Processor [required]
- Task Group [required]
- Socket [required]
- Node [optional]
- Partition [optional]

1.3.2 Task Unit

The GC64 *Task Unit* is the smallest unit of divisible concurrency. The task unit consists of the RISC-V integer register file, the RISC-V floating-point register file, the GC64 user-visible registers and the GC64 machine state registers.

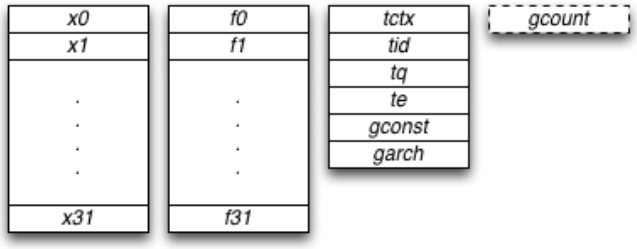


Figure 1: GC64 Task Unit

1.3.3 Task Processor

The GC64 *Task Processor* consists of the basic integer and floating-point arithmetic units, a thread control unit and one or more task units. The thread control unit is essentially an ALU dedicated to managing the state of the task units. This includes the following functions:

- Spawning new tasks
- Joining executing tasks
- Incrementing the task execution pressure [*gcount*]

- Enforcing context switch events

The task processor is permitted to execute instructions from a single task unit on any given cycle. In this manner, each task processor is myopic in its focus. It is the job of the thread control unit to enforce which task unit is in *focus* on any given cycle. Any time the thread control unit switches focus from one task unit to an adjacent task unit, we refer to this event as a *context switch*. The thread control unit may select a new task unit only from the task units that are directly attached. GC64 requires that at least one task unit exist per task processor. GC64 permits a maximum of 256 task units per task processor.

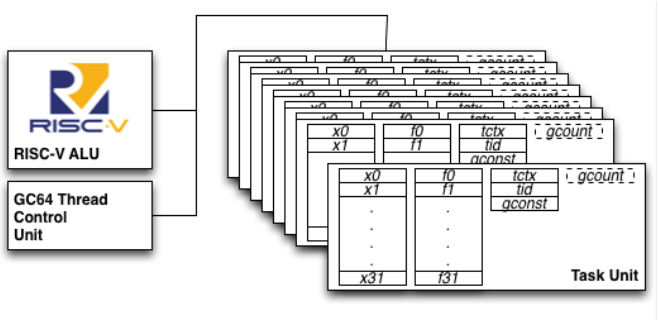


Figure 2: GC64 Task Processor

1.3.4 Task Group

The GC64 *Task Group* consists of one or more task processors interconnected to a local memory management unit (MMU). This MMU serves two purposes. First, it determines whether a request is designated as a local request or a global request. Local requests are serviced by either the on-chip scratchpad memory or one or more HMC interfaces. Global requests are serviced by off-chip resources.

The second major task of the local MMU is request coalescing. Memory requests from multiple tasks across multiple task units and task processors are coalesced into larger request packets in order to optimize channel bandwidth utilization.

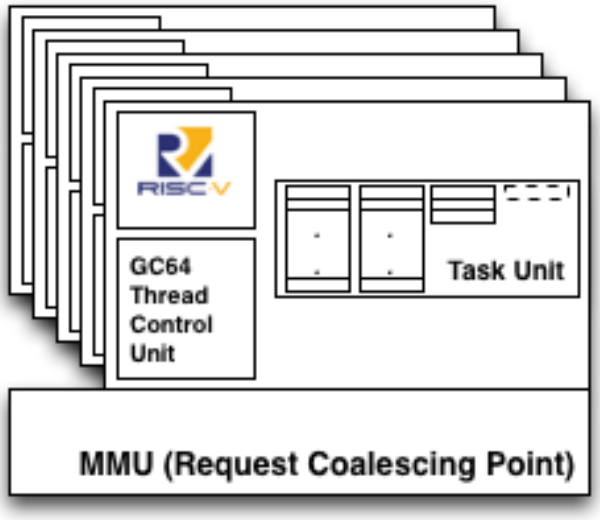


Figure 3: GC64 Task Group

1.3.5 Socket

The GC64 *socket* consists of a single GC64 system on chip (SoC) module. This module consists of of one or more GC64 task groups. These task groups are integrated via a network on chip interface to four shared on-chip components. These components as described as follows:

- **Scratchpad:** The on-chip software-managed scratchpad unit acts as a very high performance, user-mapped storage mechanism for commonly used data
- **Atomic Memory Operation Unit:** [AMO]Controls queuing, ordering and arbitration of atomic memory operations
- **HMC Channel Interface:** One or more HMC channel interfaces handle the protocol interaction to/from one or more HMC devices.
- **Off-Chip Network Interface:** The off-chip interface handles any off-chip memory requests that utilize the GC64 memory addressing mechanisms.

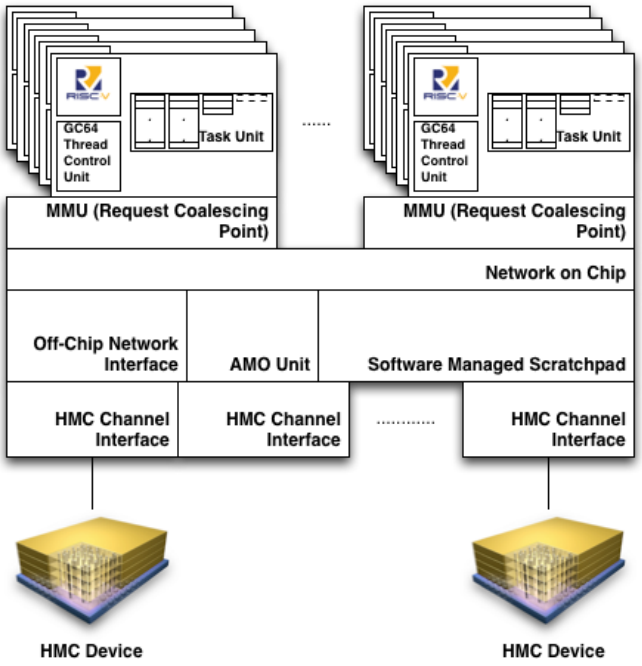


Figure 4: GC64 Socket

1.3.6 Node

The GC64 *node* architecture consists of one or more GC64 socket modules that reside on a single node addressing domain. These modules may be locally interconnected or physically co-located on a singular PCB.

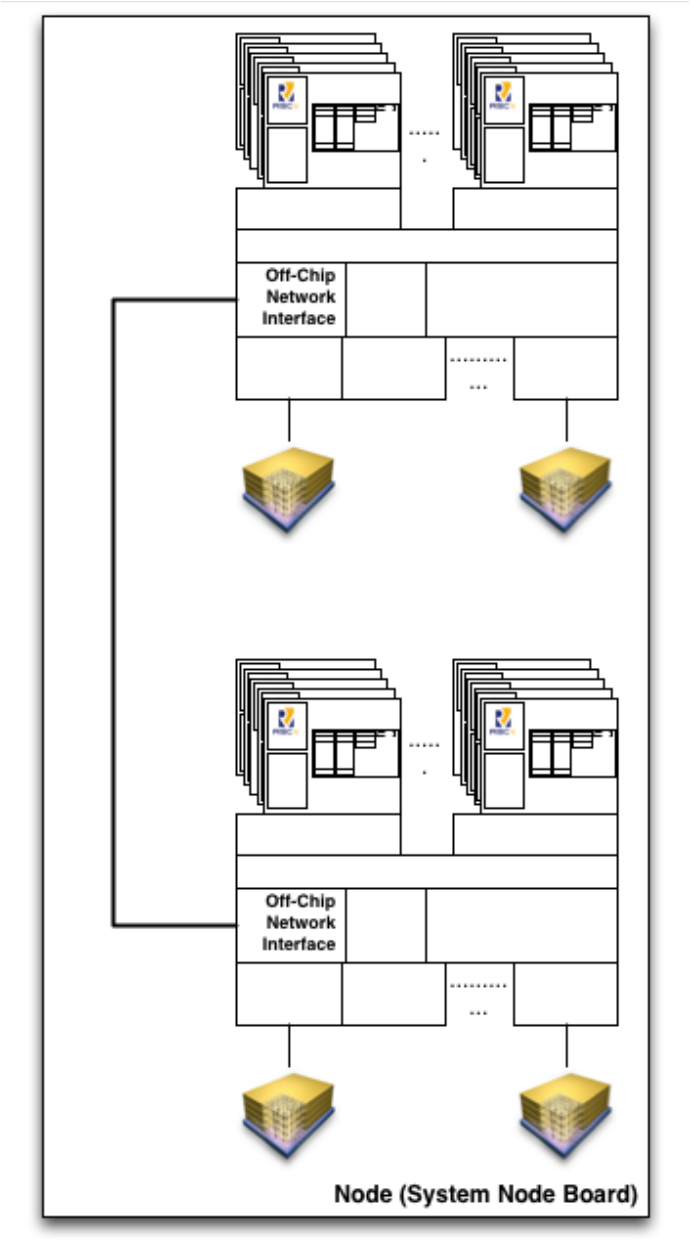


Figure 5: GC64 Node

1.3.7 Partition

The GC64 *partition* consists of one or more GC64 nodes that share a top-level addressing domain. The partition may also contain a link to an adjacent partition or partitions for the purpose of sending and receiving memory traffic.

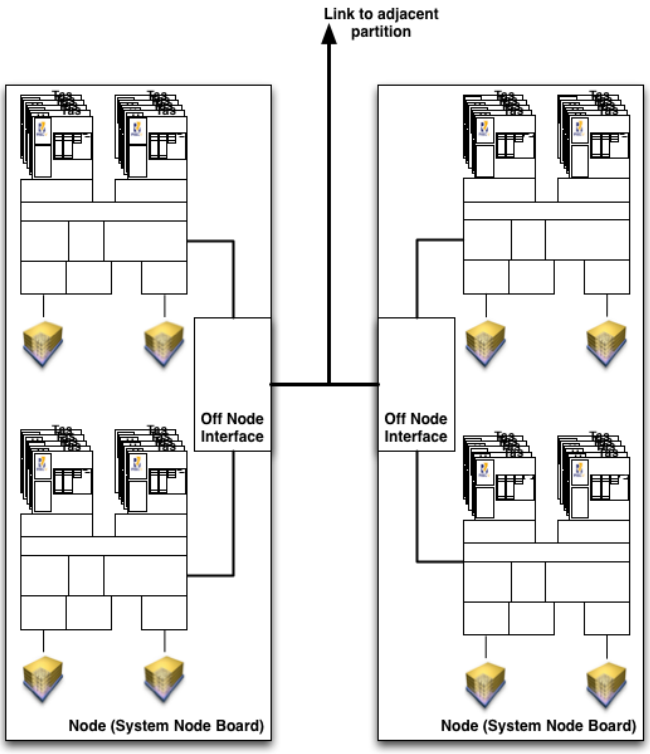


Figure 6: GC64 Partition

1.4 GC64 Memory Organization

1.4.1 Memory Architecture

The GC64 memory architecture and hierarchy is designed to promote efficient utilization of system bandwidth resources for highly concurrent applications that operate in a non-deterministic fashion with respect to memory request traffic. The memory hierarchy consists of three main components with an optional fourth component. The three main components consist of the memory management unit (MMU), the software-managed scratchpad memory and one or more Hybrid Memory Cube (HMC) devices.

The task group MMU serves two main purposes. First, it serves as a memory coalescing point for all tasks in the respective task group. The coalescing logic is designed to build larger request payloads, where applicable, in order to service multiple individual memory requests with a single HMC payload. The MMU also determines whether an individual memory request is designated as *local* or *global*. Local requests are those requests that are serviced by any HMC device or software managed scratchapd that is directly attached to the respective socket. Global requests are those that must be routed off-socket and thus serviced by adjacent sockets, nodes or partitions.

The software managed scratchpad is the second component in the mandatory memory hierarchy. The software managed scratchpad serves a purpose similar to a traditional data cache in that it contains frequently used data items that can be read or written to via any of the tasks in an application's process space. However, it differs in the sense that the actual logic associated with populating the scratchpad is serviced via software mechanisms in the programming model, as opposed to explicit circuit logic. In this manner, the user and programmer have explicit control over what and where certain items fall closer to the task units in terms of the overall memory hierarchy.

The final unit in the memory hierarchy is the HMC device. The HMC devices may be interconnected directly or via routed devices (devices connected via other HMC devices). The GC64 microarchitecture does not prevent any standard HMC configurations.

The fourth [optional] unit in the memory hierarchy consists of a the ability to performance global addressing of system archtiectures that contain multiple nodes and/or partitions. This capability provides shared and partitioned addressing of an entire system via an *extended* 128-bit addressing format [described below].

1.4.2 Physical Addressing

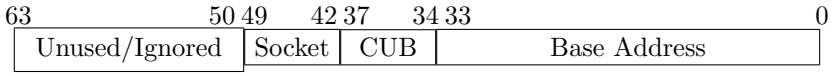
The [local] physical addressing format is designed to support locally scalable (within a node) memory addressing the directly maps to the vault-interleaved nature of the HMC devices. The local physical memory address format consists of four fields as follows:

- **Base Address:** The base address is a 34-bit physical address unit that maps directly to the HMC specification's base physical addressing field.
- **CUB:** The CUB or Cube ID field maps to the HMC request payload's definition of a Cube ID. This is a unique identifier that is local to each GC64 socket. One Cube ID maps to a single HMC device. GC64 reserves the Cube ID *0xF* for the respective socket's scratchpad memory.
- **Reserved:** This field is currently unused, but reserved for future expansion
- **Socket ID:** This 8-bit field identifies the socket ID within the GC64 node. Given that the base physical address and the CUB field are socket-local,

this field will designate addressing into devices beyond that of the local socket.

- **Unused/Ignored:** Bits [63-50] are ignored in the current GC64 microarchitecture. They are neither zero nor sign extended. This field is simply ignored.

GC64 Physical Memory Address Format

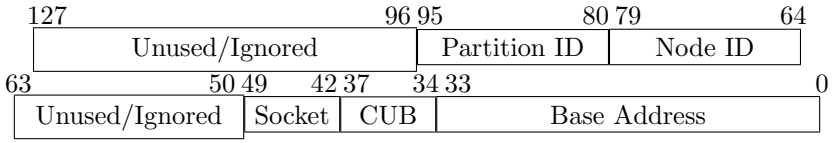


1.4.3 Extended Physical Addressing

The extended physical addressing format requires the optional 128-bit addressing mode (based upon the RISC-V 128-bit addressing mode). In this mode, global addresses are formed using a base 64-bit addressing payload and an extended 64-bit addressing payload. The extended payload consists of two additional fields that permit applications to directly access the physical memory hierarchy of adjacent nodes and/or partitions. The extended physical addressing fields are designated as follows:

- **Node ID:** This 16-bit field designates a remote node ID that may contain one or more GC64 sockets (with attached memory hierarchies).
- **Partition ID:** This 16-bit field designates a remote partition that may contain one or more GC64 nodes (each of which contain one or more sockets).
- **Unused/Ignored:** The upper 32 bits of the extended address are ignored

GC64 Extended Physical Memory Address Format



2 GC64 Instruction Set Extension

The GC64 RISC-V instruction set extension can be classified as a *brownfield extension* with respect to the machine state. In this manner, all the instructions fit within the existing RISC-V encodings. However, the GC64 extension does instantiate additional machine state in terms of user-visible and non user-visible registers.

2.1 GC64 RISC-V Machine Organization

2.2 GC64 Register State

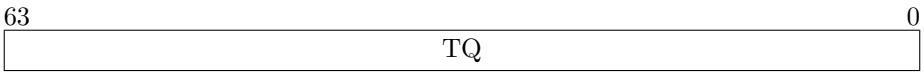
In addition to the registers defined as a part of the base RISC-V IMAFD ISA, we define an additional set of registers for the GC64 extension. The register extension is defined in terms of User-Visible, Supervisor and Machine-State registers. User-Visible registers are those that can be read and written from normal user instructions. Supervisor registers are those that can only be read and written from supervisor-privileged instructions. Machine-State registers cannot be read or written from any instruction space. Rather, these registers are implicitly modified during the normal operation of the core.

2.2.1 User-Visible Registers

The GC64 extension adds a register, *tctx*, to the list of user-visible registers. The *tctx* register is an unsigned 64bit register that exists per task and contains the base address of the task that is currently loaded in the task unit. The *tctx* register can be read from and written to using user-privileged instructions. However, given that it resides outside the base RISC-V IMAFD register set, it can only be read from and written to using the *gettask* and *settask* instructions.

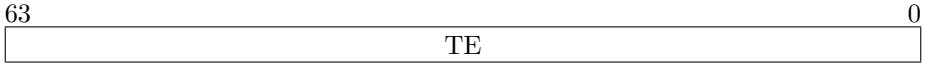
The GC64 extension adds a task ID register, *tid*, to the list of user-visible registers. The *tid* register contains an unsigned 64bit value that represents the task ID currently loaded in the respective task unit. The *tid* registers can be read from a single user-privileged instruction, *gettid*.

The GC64 extension adds a task queue register, *tq*, to the list of user-visible registers. The *tq* register is an unsigned 64bit register that contains the address of the primary task queue. The *task queue* register can only be read from normal user mode using a single instruction, *gettq*. It cannot be written to from normal user mode. It can only be written from supervisor mode using a single instruction, *settq*.



The GC64 extension adds a task exception register, *te*, to the list of user-visible registers. The *te* register is an unsigned 64bit register than contains the address of the task exception queue. The task exception queue contains task contexts that have encountered an exception state or breakpoint (during debugging). The *task exception* register can only be read from normal user

mode using a single instruction, *gette*. It cannot be written to from normal user mode. It can only be written from supervisor mode using a single instruction, *sette*.



The GC64 extension adds a configuration register, *gconst* that is user-visible. The *gconst* register is an unsigned 64bit register that contains the identity parameters for the containing task unit and GC64 processor. The *gconst* register can only be read. It cannot be written to. It can only be read via the *getgconst* instruction. The *gconst* register contains the following fields.

Mnemonic	Bits	Size [Bits]	Description
TU	[7:0]	8	Task Units ID [hardware]
TP	[15:8]	8	Task Processors ID
TG	[23:16]	8	Task Group ID
SID	[31:24]	8	Socket ID
NID	[47:32]	16	Node ID
PID	[63:48]	16	Partition ID

The GC64 extension adds an architecture description register, *garch* that is user-visible. The *garch* register is an unsigned 64bit register that contains the configuration parameters for the containing task unit and GC64 processor. The *garch* register can only be read. It cannot be written to. It can only be read via the *getgarch* instruction. The *garch* register contains the following fields.

Mnemonic	Bits	Size [Bits]	Description
NTU	[7:0]	8	Number of task units per processor
NTP	[15:8]	8	Number of task processors per group
NTG	[23:16]	8	Number of task groups per socket
NS	[31:24]	8	Number of sockets per node
NN	[47:32]	16	Number of nodes per partition
NP	[63:48]	16	Number of partitions

2.2.2 Supervisor Registers

The GC64 extension contains a single supervisor register. This register is contained within each task unit and is referred to as the *gkey* register. The purpose of this register is to contain the security key loaded via the kernel and constructor routines at application launch such that rogue tasks cannot spawn tasks in

the process space of a neighboring process. Given the purpose of this register, it remains a part of the standard context for context save and restore operations. This register is represented using as an unsigned 64bit value. It can be read from and written to using the supervisor instructions *gkey* and *skey*.

2.2.3 Machine-State Registers

The GC64 extension contains a single machine-state register. This register, known as the *gcount* register, is not directly readable or writable from any instructions. Its value is similar in function to a traditional performance counter register. Any time an instruction is retired through the pipeline, this value is incremented. The increment value is dependent upon the cost of the retired instruction. Once this counter reaches its maximum value as defined by the internal architecture state, the encountering task unit is forced to context switch and thus permit another task unit to execute. The only method the user-privileged code has to indirectly access this register state is the *ctxsw* instruction. The *ctxsw* instruction explicitly sets this register to its maximum value, thus forcing a context switch event.

2.2.4 GC64 Register Indexing

A summary of the permissible GC64 register indices is as follows. Note that all register follow a 5-bit encoding schema in order to adhere to the base RISC-V *R-Type* instruction encoding. Note that all supervisor registers appear in the index range above 0b10000.

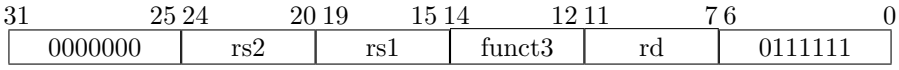
Mnemonic	Index	Access
tctx	0b00000	User,Supervisor
tid	0b00001	User,Supervisor
tq	0b00010	User,Supervisor
te	0b00011	User,Supervisor
gconst	0b00100	User,Supervisor
garch	0b00101	User,Supervisor
gkey	0b10000	Supervisor

2.3 Integer Load/Store Instructions

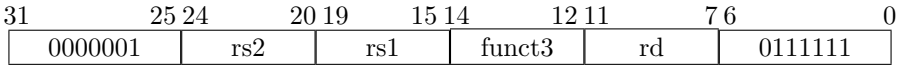
The integer load and store instructions require a full 32-bit instruction encoding space. We utilize the standard RISC-V *R-type* ISA encoding format for each of instructions. In this manner, all the integer load and store instructions support three operands and no bundled immediate values. All explicit index values for scatter/gather store/load operations must be explicitly loaded to an integer register (*x0-x31*).

The following load and store instructions share the same base opcode, but differ in their funct7 field encodings. The instruction encoding space is depicted as follows:

GC64 Integer Load Instructions



GC64 Integer Store Instructions



2.3.1 lbgthr rd, rs1, rs2

Perform a one-byte *gathered load* operation of the address and index *rs1* and *rs2*, respectively, and store the result to the intger register *rd*. The load portion of the operation sign extends to the size of the register. The operation to perform is as follows:

$$rd = rs1[rs2] \tag{1}$$

The effective address to load from is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 1)) \tag{2}$$

2.3.2 lhgthr rd, rs1, rs2

Perform a two-byte *gathered load* operation of the address and index *rs1* and *rs2*, respectively, and store the result to the intger register *rd*. The load portion of the operation sign extends to the size of the register. The operation to perform is as follows:

$$rd = rs1[rs2] \tag{3}$$

The effective address to load from is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 2)) \tag{4}$$

2.3.3 lwgthr rd, rs1, rs2

Perform a four-byte *gathered load* operation of the address and index *rs1* and *rs2*, respectively, and store the result to the intger register *rd*. The load portion of the operation sign extends to the size of the register. The operation to perform is as follows:

$$rd = rs1[rs2] \tag{5}$$

The effective address to load from is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 4)) \tag{6}$$

2.3.4 ldgthr rd, rs1, rs2

Perform an eight-byte *gathered load* operation of the address and index *rs1* and *rs2*, respectively, and store the result to the intger register *rd*. The operation to perform is as follows:

$$rd = rs1[rs2] \tag{7}$$

The effective address to load from is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 8)) \tag{8}$$

2.3.5 lbugthr rd, rs1, rs2

Perform a one-byte *gathered load* operation of the address and index *rs1* and *rs2*, respectively, and store the result to the intger register *rd*. The load portion of the operation zero extends to the size of the register. The operation to perform is as follows:

$$rd = rs1[rs2] \tag{9}$$

The effective address to load from is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 1)) \tag{10}$$

2.3.6 lhugthr rd, rs1, rs2

Perform a two-byte *gathered load* operation of the address and index *rs1* and *rs2*, respectively, and store the result to the intger register *rd*. The load portion of the operation zero extends to the size of the register. The operation to perform is as follows:

$$rd = rs1[rs2] \tag{11}$$

The effective address to load from is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 2)) \tag{12}$$

2.3.7 lwugthr rd, rs1, rs2

Perform a four-byte *gathered load* operation of the address and index *rs1* and *rs2*, respectively, and store the result to the intger register *rd*. The load portion of the operation zero extends to the size of the register. The operation to perform is as follows:

$$rd = rs1[rs2] \tag{13}$$

The effective address to load from is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 4)) \tag{14}$$

2.3.8 sbscatr rs1, rs2, rs3

Perform a one-byte *scattered store* operation using the source value *rs3* and storing to the address and index *rs1* and *rs2*, respectively. The operation to perform is as follows:

$$rs2[rs3] = rs1 \tag{15}$$

The effective address used as the target for the store is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 1)) \tag{16}$$

2.3.9 shscatr rs1, rs2, rs3

Perform a two-byte *scattered store* operation using the source value *rs3* and storing to the address and index *rs1* and *rs2*, respectively. The operation to perform is as follows:

$$rs2[rs3] = rs1 \tag{17}$$

The effective address used as the target for the store is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 2)) \tag{18}$$

2.3.10 swscatr rs1, rs2, rs3

Perform a four-byte *scattered store* operation using the source value *rs3* and storing to the address and index *rs1* and *rs2*, respectively. The operation to perform is as follows:

$$rs2[rs3] = rs1 \tag{19}$$

The effective address used as the target for the store is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 4)) \tag{20}$$

2.3.11 sdscatr rs1, rs2, rs3

Perform an eight-byte *scattered store* operation using the source value *rs3* and storing to the address and index *rs1* and *rs2*, respectively. The operation to perform is as follows:

$$rs2[rs3] = rs1 \tag{21}$$

The effective address used as the target for the store is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 8)) \tag{22}$$

2.4 Single-Precision Load/Store Instructions

The single-precision floating-point load and store instructions require a full 32-bit instruction encoding space. We utilize the standard RISC-V *R-type* ISA encoding format for each of instructions. In this manner, all the SP floating-point load and store instructions support three operands and no bundled immediate values. All explicit index values for scatter/gather store/load operations must be explicitly loaded to an integer register (*x0-x31*). All single-precision floating point values used as source or target registers must be a floating point register (*f0-f31*).

The following load and store instructions share the same base opcode, but differ in their funct7 field encodings. The instruction encoding space is depicted as follows:

GC64 Single-Precision Floating-Point Load Instructions

31 25	24 20	19 15	14 12	11 7	6 0
0000010	rs2	rs1	funct3	rd	0111111

GC64 Single-Precision Floatng-Point Store Instructions

31 25	24 20	19 15	14 12	11 7	6 0
0000011	rs2	rs1	funct3	rd	0111111

2.4.1 flwgthr fd, rs1, rs2

Perform a single-precision floating-point *gathered load* operation of the address and index *rs1* and *rs2*, respectively, and store the result to the intger register *fd*. Note that the source integer registers are utilized for addressing and the target register is a floating point register. The operation to perform is as follows:

$$fd = rs1[rs2] \tag{23}$$

The effective address to load from is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 4)) \tag{24}$$

2.4.2 fswscatr rs1, rs2, fs3

Perform a single-precision floating-point *scattered store* operation using the source value *fs3* and storing to the address and index *rs1* and *rs2*, respectively. The operation to perform is as follows:

$$fs3 = rs2[rs3] \tag{25}$$

The effective address used as the target for the store is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 4)) \tag{26}$$

2.5 Double-Precision Load/Store Instructions

The double-precision floating-point load and store instructions require a full 32-bit instruction encoding space. We utilize the standard RISC-V *R-type* ISA encoding format for each of instructions. In this manner, all the DP floating-point load and store instructions support three operands and no bundled immediate values. All explicit index values for scatter/gather store/load operations must be explicitly loaded to an integer register (*x0-x31*). All double-precision floating point values used as source or target registers must be a floating point register (*f0-f31*).

The following load and store instructions share the same base opcode, but differ in their funct7 field encodings. The instruction encoding space is depicted as follows:

GC64 Double-Precision Floating-Point Load Instructions

31 25	24 20	19 15	14 12	11 7	6 0
0000100	rs2	rs1	funct3	rd	0111111

GC64 Double-Precision Floatng-Point Store Instructions

31 25	24 20	19 15	14 12	11 7	6 0
0000101	rs2	rs1	funct3	rd	0111111

2.5.1 fldgthr fd, rs1, rs2

Perform a double-precision floating-point *gathered load* operation of the address and index *rs1* and *rs2*, respectively, and store the result to the floating-point register *fd*. Note that the source integer registers are utilized for addressing and the target register is a floating point register. The operation to perform is as follows:

$$fd = rs1[rs2] \tag{27}$$

The effective address to load from is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 8)) \tag{28}$$

2.5.2 fsdscatr rs1, rs2, fs3

Perform a double-precision floating-point *scattered store* operation using the source value *fs3* and storing to the address and index *rs1* and *rs2*, respectively. The operation to perform is as follows:

$$fs3 = rs2[rs3] \tag{29}$$

The effective address used as the target for the store is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 8)) \tag{30}$$

2.6 Concurrency Instructions

The concurrency instructions require a full 32-bit encoding space. We utilize the standard *R-type* ISA encoding format for each of the instructions. In this manner, all the concurrency instructions do not accept bundled immediate values. All explicit immediate values must be explicitly loaded to an integer register (*x0-x31*). The instruction encoding space is depicted as follows:

GC64 Concurrency Instructions

31 25	24 20	19 15	14 12	11 7	6 0
0000110	rs2	rs1	funct3	rd	0111111

2.6.1 iwait rd, rs1, rs2

Pend the execution of the encountering task until the hazard on the integer register specified by *rd* has been cleared. The clear register state indicates that no outstanding load operations are in flight and no outstanding arithmetic operations are in the pipeline. The encountering thread pends as long as *rs2* ¡ *rs1*. The maximum number of cycles to pend is *rs1-rs2*. The machine state adheres to the following state:

Data: rd is the target integer register, rs1 is the max unsigned value to be, rs2 is the unsigned start value
Result: The encountering task unit pends until rs2 is greater than or equal to rs1
while *rs2 less than or equal to rs2* **do**
| rs2++;
end
clear iwait state

2.6.2 ctxsw

Implicitly set the *gcount* machine state register to its maximum value. The result being a forcible context switch of the encountering task. If the task control logic does not have any subsequent tasks ready for execution, then the minimum context switch wait time is a single clock cycle.

2.7 Task Control Instructions

The task control instructions require a full 32-bit encoding space. We utilize the standard *R-type* format for each of the instructions. In this manner, all the task control instructions do not accept bundled immediate values. All explicit immediate values must be loaded to an integer register (*x0-x31*). The instruction encoding space is depicted as follows:

GC64 Task Major Control Instructions

31 25	24 20	19 15	14 12	11 7	6 0
0000111	rs2	rs1	funct3	rd	0111111

GC64 Task Context Read (Get) Instructions

31 25	24 20	19 15	14 12	11 7	6 0
0001000	rs2	rs1	funct3	rd	0111111

GC64 Task Context Write (Set) Instructions

31 25	24 20	19 15	14 12	11 7	6 0
0001001	rs2	rs1	funct3	rd	0111111

2.7.1 spawn rd, rs1

Spawn a task using the task context at the address specified by the integer register *rs1*. Return the status of the spawn operation in the integer register

rd. The address is considered to be a 64-bit virtual address at minimum. When 128-bit addressing is enabled, it is permissible for the address to be 128-bits.

2.7.2 join rd, rs1

Perform a task join operation on the task context specified by the integer register *rs1*. Return the status of the spawn operation in the integer register *rd*. The address is considered to be a 64-bit virtual address at minimum. When 128-bit addressing is enabled, it is permissible for the address to be 128-bits.

2.7.3 gettask rd, tctx

Retrieve the task context value for the encountering task unit. The value of the tctx register should be, at minimum, at 64-bit virtual address. Store the value to the integer register specified by *rd*. When 128-bit addressing is enabled, it is permissible for the address to be 128-bits.

2.7.4 settask tctx, rs1

Set the task context value for the current task unit to the address value specified by the integer register *rs1* and store the value to the *tctx* register. This is a 64-bit virtual address at minimum. When 128-bit addressing is enabled, it is permissible for the address to be 128-bits.

2.7.5 gettid rd, gtid

Retrieve the taskid value from the encountered task and write the value to the integer register specified by *rd*.

2.7.6 gettq rd, tq

Retrieve the task queue value for the encountering task unit. The value of the tq register should be, at minimum, at 64-bit virtual address. Store the value to the integer register specified by *rd*. When 128-bit addressing is enabled, it is permissible for the address to be 128-bits.

2.7.7 settq tq, rs1

Set the task queue value for the current task unit to the address value specified by the integer register *rs1* and store the value to the *tq* register. This is a 64-bit virtual address at minimum. When 128-bit addressing is enabled, it is permissible for the address to be 128-bits.

2.7.8 gette rd, te

Set the task exception queue value for the current task unit to the address value specified by the integer register *rs1* and store the value to the *te* register. This

is a 64-bit virtual address at minimum. When 128-bit addressing is enabled, it is permissible for the address to be 128-bits.

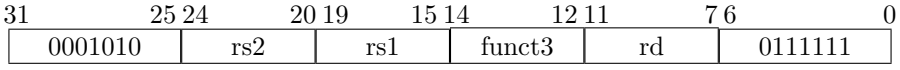
2.7.9 sette te, rs1

Set the task exception queue value for the current task unit to the address value specified by the integer register *rs1* and store the value to the *te* register. This is a 64-bit virtual address at minimum. When 128-bit addressing is enabled, it is permissible for the address to be 128-bits.

2.8 Environment Instructions

The environment instructions require a full 32-bit encoding space. We utilize the standard *R-type* format for each of the instructions. In this manner, all the environment instructions do not accept bundled immediate values. All explicit immediate values must be loaded to an integer register (*x0-x31*). The instruction space is encoded as follows:

GC64 Environment Instructions



2.8.1 getgconst rd, gconst

Retrieve the value of the GC64 constant register *gconst*. Write the value into the integer register specified by *rd*.

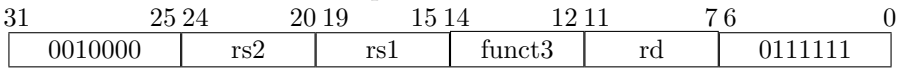
2.8.2 getgarch rd, garch

Retrieve the value of the GC64 constant register *garch*. Write the value to the integer register specified by *rd*.

2.9 Supervisor Instructions

The supervisor instructions require a full 32-bit encoding space. Wed utilize the standard *R-type* format for each of the instructions. In this manner, all the supervisor instructions do not accept bundled immediate values. All explicit immediate values must be loaded to an integer register (*x0-x31*). The instruction space is encoded as follows:

GC64 Supervisor Instructions



2.9.1 sgetkey rd, gkey

Retrieve the value of the current GC64 security key. Write the result to the integer register specified by *rd*. This instruction may only be executed by tasks with supervisor privileges.

2.9.2 ssetkey gkey, rs1

Set the value of the current GC64 security key. The value is written from the integer register specified by *rs1*. This instruction may only be executed by tasks with supervisor privileges.

2.10 [Optional] 128-bit Integer Load/Store Instructions

The 128-bit integer load and store instructions require a full 32-bit instruction encoding space. We utilize the standard RISC-V R-type ISA encoding format for each of instructions. In this manner, all the integer load and store instructions support three operands and no bundled immediate values. All explicit index values for scatter/gather store/load operations must be explicitly loaded to an integer register (x0-x31). The instruction encoding space is depicted as follows:

GC64 128-bit Load/Store Instructions

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	funct3	rd	0111111

2.10.1 ldugthr rd, rs1, rs2

Perform an eight-byte *gathered load* operation of the address and index *rs1* and *rs2*, respectively, and store the result to the intger register *rd*. The load portion of the operation zero extends to the size of the register. The operation to perform is as follows:

$$rd = rs1[rs2] \tag{31}$$

The effective address to load from is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 16)) \tag{32}$$

2.10.2 lqgthr rd, rs1, rs2

Perform a sixteen-byte *gathered load* operation of the address and index *rs1* and *rs2*, respectively, and store the result to the intger register *rd*. The operation to perform is as follows:

$$rd = rs1[rs2] \tag{33}$$

The effective address to load from is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 16)) \tag{34}$$

2.10.3 sqscatr rs1, rs2, rs3

Perform a sixteen-byte *scattered store* operation using the source value *rs3* and storing to the address and index *rs1* and *rs2*, respectively. The operation to perform is as follows:

$$rs2[rs3] = rs1 \tag{35}$$

The effective address used as the target for the store is formed using the following methodology:

$$Effective = addr(rs1 + ((rs2 * 16)) \tag{36}$$

3 GC64 Task Interface Specification

3.1 Overview

Historically, microarchitecture specifications do not specify the method by which the programming model(s) achieves parallelism and concurrency. The limit of the documentation generally stops at instruction level parallelism (ILP) or multi-core connectivity to shared caches. However, the GC64 microarchitecture extension is explicitly designed to foster highly efficient parallel software development. As such, we explicitly outline the mechanisms by which the software and the instruction set may interact with the inherent hardware concurrency mechanisms (eg, *task units*).

We describe these mechanisms in terms of *task contexts* and *task queues*. All local and scalable concurrency mechanisms emanate from these two concepts.

3.2 Task Context

The *task context* is responsible for encapsulating an indivisible task. The encapsulated task may be in one of four states. These states are summarized as follows:

- **Free:** Task contexts in this state have no instructions to execute and no known application state associated with it. They may exist within a queue of other unused task contexts. The purpose of these task contexts is to provide a high performance method to spawn new tasks without unnecessary memory allocation.
- **Queued:** Task contexts in this state have instructions to execute and, potentially, application state. The task contexts are awaiting a task processor to fetch the associated state into a task unit and begin (or continue) execution.
- **Executing:** Task contexts in this state are currently associated with a task unit associated with a task processor. The address of the executing task context is loaded into the *tctx* register of each task unit.
- **Paused:** Task contexts in this state have instruction and application state associated with it, but remain paused and queued on an exception queue. The respective task context may have encountered a break point, a single step, a fatal unmasked exception or an adjacent task context may have encountered one in the same. This is the state utilized for debugging.

Given that the task context contains viable application state and is further utilized in debugging, we explicitly define the size and structure of the context. The task context structure contains all the necessary storage to encapsulate all the register state associated with a single task unit. We define two task context structures. The first shall be associated with the base GC64 microarchitecture specification. The latter serves to contain the register state when the optional 128-bit addressing mode is enabled. We define these using a basic *C* structure.

```
/* Task context: 64-bit addressing mode */
struct task_context{
uint64_t id; /* unique task context id */
struct task_context *next;
uint64_t pc;
uint64_t x_reg[32];
uint64_t f_reg[32];
uint64_t tctx;
uint64_t tid;
uint64_t tq;
uint64_t te;
uint64_t gconst;
uint64_t garch;
uint64_t gkey;
};

/* Task context: 128-bit addressing mode */
struct task_context{
uint64_t id; /* unique task context id */
struct task_context *next;
uint64_t pc[2];
uint64_t x_reg[64];
uint64_t f_reg[32];
uint64_t tctx[2];
uint64_t tid;
uint64_t tq[2];
uint64_t te[2];
uint64_t gconst;
uint64_t garch;
uint64_t gkey;
};
```

3.3 Task Queuing

3.3.1 Task Queue Structure

The GC64 task queue structure provides the basic ability to queue one or more task contexts with or without executable instructions. In addition, task queue structures provide a rudimentary locality mechanism for task groups, sockets, nodes and partitions. While this locality is not explicitly enforced in hardware, the software runtime mechanisms may make explicit use of this inherent locality for spawning or joining tasks. As mentioned in the GC64 register extensions, the *tq* and *te* registers each points to a single task queue instance, respectively. The task queues associated with the *tq* and *te* registers must be different.

Each of the task queues structures contains four relevant items. Pointers are 64bits in the base GC64 architecture. If the system supports the 128bit

addressing mode, the pointers are 128bits. These items are summarized as follows:

- **ID:** A unique identifier for each task queue structure
- **KEYSTONE:** A pointer to a task queue keystone structure. It is permissible (especially in small configurations) for this pointer to be NULL.
- **ENQ:** The enqueue item is a pointer to the first valid task context structure. Valid task contexts contain the state associated with a task that is ready and valid for execution. If this pointer is NULL, then it is assumed that there are no tasks ready to execute on this queue.
- **FQ:** The free queue item is a pointer to the first *free* task context structure. Free task contexts contain the state associated with a task that is ready to be spawned and assigned work. No current valid state or instructions are associated with free task contexts. The goal of free task contexts is to provide applications an ability to spawn tasks without allocating memory.

We may also define these task queue structures in both the 64 and 128bit addressing modes in terms of a basic *C* structure as follows:

```
/* Task Queue Structure */
struct task_queue{
uint64_t id; /* unique task queue id */
struct task_keystone *keystone;
struct task_context *enqueue;
struct task_context *freequeue;
};
```

3.3.2 Task Queue Keystone Structure

The GC64 task queue architecture also provides an explicit and consolidated ability to interconnect the local task queue structures associated with a task group with the task queue structures of adjacent sockets, nodes and partitions. Each keystone structure contains four data values. The values are organized as follows:

- **ID:** The keystone ID value is an unsigned 64bit unique identifier for each keystone structure.
- **SOCKET:** The socket next pointer is, by default, a 64bit pointer to a task queue structure on an adjacent socket. In the optional 128bit addressing mode, this pointer is 128bits. If this pointer value is NULL, then no adjacent task queues are connected.

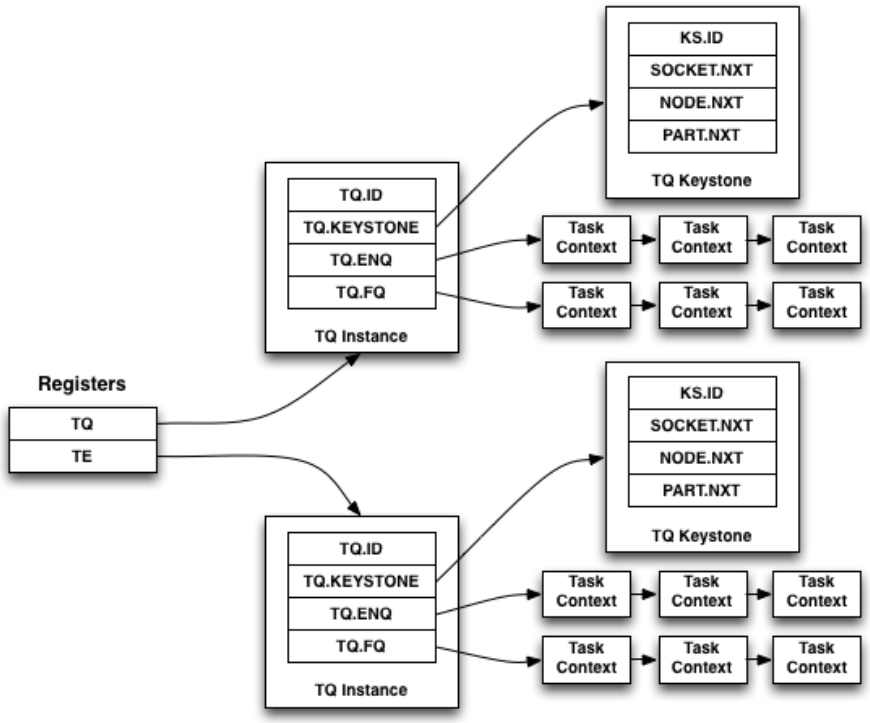


Figure 7: GC64 Task Queue Link Structure

- **NODE:** The node next pointer is, by default, a 64bit pointer to a task queue structure on an adjacent node. In the optional 128bit addressing mode, this pointer is 128bits. If this pointer value is NULL, then no adjacent task queues are connected.
- **PART:** The partition next pointer is, by default, a 64bit pointer to a task queue structure on an adjacent partition. In the optional 128bit addressing mode, this pointer is 128bits. If this pointer value is NULL, then no adjacent task queues are connected.

We may also define these keystone structures in both the 64 and 128bit addressing modes in terms of a basic *C* structure as follows:

```
/* Task Queue Keystone */
struct task_keystone{
uint64_t id; /* unique task keystone id */
struct task_queue *socket;
struct task_queue *node;
```

```
struct task_queue *partition;
};
```

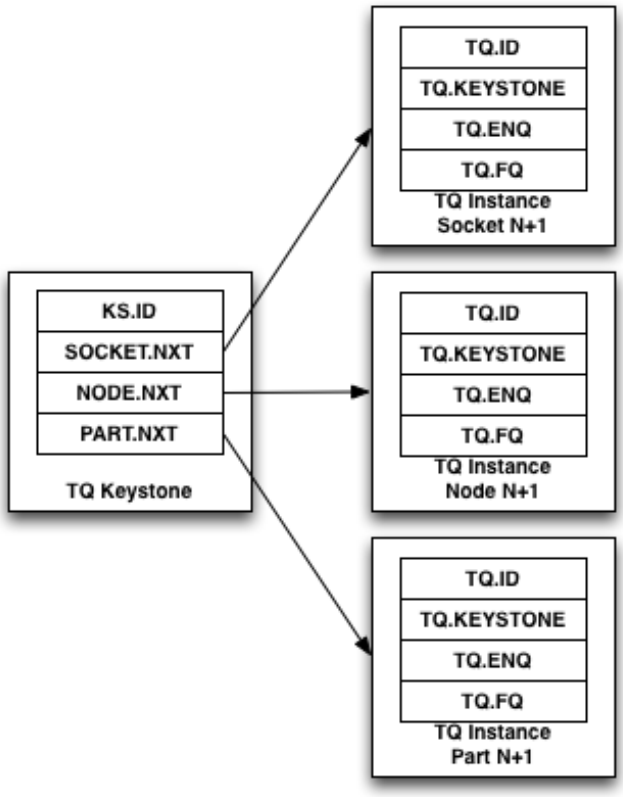


Figure 8: GC64 Task Keystone Link Structure

3.4 Context Save/Restore

Given the aforementioned task context and task queueing architecture, we define the context save and context restore mechanisms in terms of task context and task queues. Each task group must have at least one unique task exception queue instance that is manifested using task queue structure. Any time a *pause* event in encountered and a context save operation is initiated for the given process space, all the associated task units in the executing task group adhere to the following process:

1. Pause event is received (exception, breakpoint, step, et.al)

2. RISC-V pipeline is paused and drained
3. All RISC-V and GC64 register state is saved to the task unit's associated task context at *tctx*
4. The task context *tctx* is inserted into the task exception queue *te*.

The converse of the context save operation is a context restore. The context restore operation performs the reverse of a context save. However, the context restore operation utilizes the value from the *gconst* of each saved task context to determine the physical locality for which the respective task is restored. In this manner, contexts are restored to the same task groups, task processors and task units from which they were originally saved. This process is as follows:

1. Restore event is received (single step, execute signal, et.al.)
2. A task context is fetched from the *te* queue.
3. The respective *gconst* value is read and interpreted for the correct location
4. The task context is restored to the correct physical task unit
5. Execution resumes

3.5 Task Instruction Return Values

The task control instructions *spawn* and *join* attempt to increase and decrease the concurrency of a given application context, respectively. Note the definition of these instructions. Each of the aforementioned instructions returns a value in the target integer register that is essentially a return code that describes the status of the given operation. The return values are signed 64-bit values. This is true in both the base 64-bit addressing mode and the optional extended 128-bit addressing mode. The full list of return values is as follows:

Mnemonic	Value	Description
TFATAL	-1	A fatal error occurred in the operation
TOK	0	No error occurred. The operation was successful
TNULL	1	Task queue was previously null (no queued tasks)

4 GC64 Instruction Set Listings

The following tables contain the full encoding space for GC64 instructions. Note that the encoding tables may contain explicit references to register indices such as *X0* and *TCTX*. Any references to explicit indices will be capitalized in the tables below. Please refer to the RISC-V register indexing and the GC64 register indexing tables.

GC64 Integer Load/Store Instructions[R-Type]

Mnemonic	funct7	funct3	opcode	Encoding
lbgthr rd,rs1,rs	0000000	000	0111111	0000000[rs2][rs1]000[rd]0111111
lhgthr rd,rs1,rs	0000000	001	0111111	0000000[rs2][rs1]001[rd]0111111
lwgthr rd,rs1,rs	0000000	010	0111111	0000000[rs2][rs1]010[rd]0111111
ldgthr rd,rs1,rs	0000000	011	0111111	0000000[rs2][rs1]011[rd]0111111
lbugthr rd,rs1,rs	0000000	100	0111111	0000000[rs2][rs1]100[rd]0111111
lhugthr rd,rs1,rs	0000000	101	0111111	0000000[rs2][rs1]101[rd]0111111
lwugthr rd,rs1,rs	0000000	110	0111111	0000000[rs2][rs1]110[rd]0111111
sbscatr rs1,rs2,rs3	0000001	000	0111111	0000001[rs2][rs1]000[r3]0111111
shscatr rs1,rs2,rs3	0000001	001	0111111	0000001[rs2][rs1]001[r3]0111111
swscatr rs1,rs2,rs3	0000001	010	0111111	0000001[rs2][rs1]010[r3]0111111
sdscatr rs1,rs2,rs3	0000001	011	0111111	0000001[rs2][rs1]011[r3]0111111

GC64 Single Precision Load/Store Instructions[R-Type]

Mnemonic	funct7	funct3	opcode	Encoding
flwgthr fd,rs1,rs	0000010	000	0111111	0000010[rs2][rs1]000[fd]0111111
fswscatr rs1,rs2,fs3	0000011	000	0111111	0000011[rs2][rs1]000[fs3]0111111

GC64 Double Precision Load/Store Instructions[R-Type]

Mnemonic	funct7	funct3	opcode	Encoding
fldgthr fd,rs1,rs	0000100	000	0111111	0000100[rs2][rs1]000[fd]0111111
fsdscatr rs1,rs2,fs3	0000101	000	0111111	0000101[rs2][rs1]000[fs3]0111111

GC64 Concurrency Instructions[R-Type]

Mnemonic	funct7	funct3	opcode	Encoding
iwait rd,rs1,rs2	0000110	000	0111111	0000110[rs2][rs1]000[rd]0111111
ctxsw	0000110	001	0111111	0000110[X0][X0]001[X0]0111111

GC64 Task Control Instructions[R-Type]

Mnemonic	funct7	funct3	opcode	Encoding
spawn rd,rs1	0000111	000	0111111	0000111[X0][rs1]000[rd]0111111
join rd,rs1	0000111	001	0111111	0000111[X0][rs1]001[rd]0111111
gettask rd,tctx	0001000	000	0111111	0001000[X0][TCTX]000[rd]0111111
gettid rd,tid	0001000	001	0111111	0001000[X0][TID]001[rd]0111111
gettq rd,tq	0001000	010	0111111	0001000[X0][TQ]010[rd]0111111
gette rd,te	0001000	011	0111111	0001000[X0][TE]011[rd]0111111
settask tctx,rs1	0001001	000	0111111	0001001[X0][rs1]000[TCTX]0111111
settq tq,rs1	0001001	001	0111111	0001001[X0][rs1]001[TQ]0111111
sette te,rs1	0001001	010	0111111	0001001[X0][rs1]010[TE]0111111

GC64 Environment Instructions[R-Type]

Mnemonic	funct7	funct3	opcode	Encoding
getgconst rd,gconst	0001010	000	0111111	0001010[X0][GCONST]000[rd]0111111
getgarch rd,garch	0001010	001	0111111	0001010[X0][GARCH]001[rd]0111111

GC64 Supervisor Instructions[R-Type]

Mnemonic	funct7	funct3	opcode	Encoding
sgetkey rd,gkey	0010000	000	0111111	0010000[X0][GKEY]000[rd]0111111
ssetkey gkey,rs1	0010000	001	0111111	0010000[X0][rs1]001[GKEY]0111111

GC64 [Optional] 128-bit Load/Store Instructions[R-Type]

Mnemonic	funct7	funct3	opcode	Encoding
ldugthr rd,rs1,rs2	0100000	000	0111111	0010000[rs2][rs1]000[rd]0111111
lqugthr rd,rs1,rs2	0100000	001	0111111	0010000[rs2][rs1]001[rd]0111111
sqscatr rs1,rs2,rs3	0100000	010	0111111	0010000[rs2][rs1]010[rs3]0111111

5 History and Acknowledgements

This project is sponsored by the Data Intensive Scalable Computing Laboratory in the Whitacre School of Engineering at Texas Tech University. The original GoblinCore64 Instruction Set Architecture and concept began in 2013 as a stand-alone ISA without dependence upon any external projects. However, given the emergence of the RISC-V project from the University of California, Berkeley, and the inherent synergies with their design decisions, we decided to migrate the base GC64 ISA to the RISC-V infrastructure. We would like to thank the members and collaborators associated with the RISC-V project for their diligent effort to build, document and distribute the RISC-V ISA.

References

[1] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008.

[2] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovi. The risc-v instruction set manual, volume i: User-level isa, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.