GoblinCore64:
A RISC-V Extension for Data Intensive Computing

John D. Leidel, Xi Wang, Yong Chen
June 29, 2015

Data-Intensive Scalable Computing Laboratory (DISCL)
Overview

• Why a RISC-V Extension?

• GC64 Architecture Organization

• GC64 ISA Extension(s)

• Future Development
Motivations & goals behind GoblinCore64

WHY A RISC-V EXTENSION?
GC64 Overview

• Original project
  • Ground up effort to build an architecture and ISA for data intensive computing

• Latest Research
  • Utilize the core ISA/machine architecture from RISC-V
  • Build extensions to core RISC-V architecture for data intensive computing

• Focus on what matters:
  
  **Programmable Data Intensive Computing Performance**
GC64 Requirements

- Data intensive algorithms & applications
  - Sparse data structures
  - Sparse Matrix/SPMV
  - Graph computations
  - Network Theory
- Driving characteristics
  - Non-Unit Stride Memory Access
  - *Scatters/Gathers*
  - Memory Intensive
  - *Cache unfriendly*
  - *Non deterministic*

Architectural Goals

• Simple architectural components
  • Scaling the degree concurrency within an SoC is a manufacturing problem, not a design problem

• Simple ISA extensions conducive to compiler optimizations for concurrent applications
  • NOT writing the world’s latest parallel compiler

• Provide low-level hardware support for mutable concurrency
  • A task lives in HARDWARE

• Provide hardware mechanisms to minimize hardware context switch latency

• Provide well-defined mechanisms on when and how context switch events occur
  • Provide a well-defined mechanism for USER SPACE code to induce a context switch
...and why its important for our ISA extension

GC64 ARCHITECTURE ORGANIZATION
The task processor is permitted to execute instructions from a single task unit on any given cycle. In this manner, each task processor permits full control from a single task unit at any given time. 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 on-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.
The GC64 socket consists of a single GC64 system on chip (SoC) module. This module consists 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 are 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**: 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.
- **On-chip Network Interface**: The on-chip interface handles any on-chip memory requests that utilize the GC64 memory addressing mechanisms.
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.
User-Visible Registers

- **TCTX**
  - 64bit register that holds the address of the current task context

- **TID**
  - 64bit register that holds the current task ID

- **TQ**
  - 64bit register that holds the address of the task queue

- **TE**
  - Task exception register: exceptions specific to task operations

- **GCONST**
  - Constant register defining the locality of the given task unit

- **GARCH**
  - Architecture description register

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Bits</th>
<th>Size [Bits]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TU</td>
<td>[7:0]</td>
<td>8</td>
<td>Task Units ID [hardware]</td>
</tr>
<tr>
<td>TP</td>
<td>[15:8]</td>
<td>8</td>
<td>Task Processors ID</td>
</tr>
<tr>
<td>TG</td>
<td>[23:16]</td>
<td>8</td>
<td>Task Group ID</td>
</tr>
<tr>
<td>SID</td>
<td>[31:24]</td>
<td>8</td>
<td>Socket ID</td>
</tr>
<tr>
<td>NID</td>
<td>[47:32]</td>
<td>16</td>
<td>Node ID</td>
</tr>
<tr>
<td>PID</td>
<td>[63:48]</td>
<td>16</td>
<td>Partition ID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Bits</th>
<th>Size [Bits]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTU</td>
<td>[7:0]</td>
<td>8</td>
<td>Number of task units per processor</td>
</tr>
<tr>
<td>NTP</td>
<td>[15:8]</td>
<td>8</td>
<td>Number of task processors per group</td>
</tr>
<tr>
<td>NTG</td>
<td>[23:16]</td>
<td>8</td>
<td>Number of task groups per socket</td>
</tr>
<tr>
<td>NS</td>
<td>[31:24]</td>
<td>8</td>
<td>Number of sockets per node</td>
</tr>
<tr>
<td>NN</td>
<td>[47:32]</td>
<td>16</td>
<td>Number of nodes per partition</td>
</tr>
<tr>
<td>NP</td>
<td>[63:48]</td>
<td>16</td>
<td>Number of partitions</td>
</tr>
</tbody>
</table>
Supervisor Registers

• **GKEY**
  • The “gkey” register contains a 64-bit key loaded by the kernel
  • The key determines whether a task may spawn and execute work on neighboring task processors
  • Can only be written from privileged code
  • This provides a very rudimentary protection mechanism in order to prevent:
    • *Task resource starvation from other process spaces*
    • *Memory bandwidth utilization from outside process address spaces*
    • *Well defined mechanism to constrain the bounds of highly concurrent applications*

*Currently being revised based upon the latest supervisor specification*
Machine State Registers

• GCOUNT
  • Tracks the current state of a task’s context pressure
  • Every instruction increments this value
  • Once the value reaches the overflow threshold, a context switch event is injected
  • Cannot be read or written directly from any instructions
  • One instruction, ctxsw, implicitly forces an overflow
Current GC64 RISC-V specification

GC64 RISC-V EXTENSION
RISC-V ISA Requirements

• What are we using from the core RISC-V architecture spec?

• ISA:
  • RV64I: 64bit integer arithmetic and addressing support
  • M-Extension: 64-bit integer multiplication and division
  • A-Extension: Atomic instructions

• Optional Support:
  • F-Extension: Single-precision floating point arithmetic support and storage
  • D-Extension: Double-precision floating point arithmetic support and storage
  • RC128I: Extended (scalable) 128-bit addressing support
    • Physical addressing is architected to accept 128-bit extension
### GC64 Instruction Extensions

- Integer load/store instructions
- Single Precision load/store instructions
- Double Precision load/store instructions
- Concurrency instructions
- Task control instructions
- Environment instructions
- Supervisor instructions
- Extended addressing instructions

<table>
<thead>
<tr>
<th></th>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R-type
GC64 Integer Load/Store

- **Gathers (indexed load)**
  - LBGTHR Rd, Rs1, Rs2
  - Rd = Rs1[Rs2]
  - Effective Base Address = \( \text{addr}(\text{rs1} + (\text{rs2} \times \text{size\_in\_bytes})) \)

- **Scatters (indexed store)**
  - SBSCATR Rs1, Rs2, Rs3
  - Rs2[Rs3] = Rs1
  - Effective Base Address = \( \text{addr}(\text{rs1} + (\text{rs2} \times \text{size\_in\_bytes})) \)

\[ \text{Value} = A[B[i]] \]

\[ A[B[i]] = \text{Value} \]
GC64 Concurrency
Instructions

- **IWAIT Rd, Rs1, Rs2**
  - Pend the execution of the next instruction until the register hazard on the register index at Rd has been cleared
  - The pend is upheld while:
    - $rs2 < rs1$

- **CTXSW**
  - Set the $gcount$ register to an overflow state, thus forcing the current task to context switch

---

**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

```plaintext
while rs2 less than or equal to rs2 do
  rs2++;
end
clear iwait state
```
GC64 Task Control Instructions

- **SPAWN Rd, Rs1**
  - Spawn a new task using the context at Rs1

- **JOIN Rd, Rs1**
  - Join a task using the task context at Rs1

- **GETTASK RD, TCTX**
  - Retrieve the task context value for the encountering task unit
  - “Where do I get my new tasks from?”

- **SETTASK TCTX, RD**
  - Set the task context value for the encountering task unit
  - “This is where I get my new tasks from!”

- **GETTID RD, GTID**
  - “What is my {task,thread,etc} id?

- **Modifying Task Queue Values**
  - **GETTQ RD, TQ**
  - **SETTQ TQ, RS1**
  - **GETTE RD, TE**
  - **SETTE TE, RS1**
GC64 Task Queue Structure

- Task control instructions manipulate a well-defined task queuing construct
  - Instructions are implemented via small RISC-V cores and embedded microcode
- Permits deterministic placement of task/thread work via a low-level runtime library
Next steps in GC64 development

FUTURE DEVELOPMENT
Future Development

• The core architectural specification is documented
  • See http://discl.cs.ttu.edu/gitlab/gc64/gc64-doc

• The next steps in hardware design/implementation:
  1. Complete the design and implementation of the memory coalescing unit!
  2. Complete the design of the scratchpad memory layout
  3. Begin design and path-finding efforts on the scalable network architecture
  4. Demonstrate the initial architectural spec on an FPGA!

• The next steps in software design/implementation:
  1. Harden the current simulator implementation(s)
  2. Rework portions of the the software scratchpad runtime (named address spaces)
  3. Complete the low-level runtime library
Questions/Comments?

John Leidel
john.leidel@ttu.edu
http://discl.cs.ttu.edu/gitlab/groups/gc64