RISCV-Business: A Configurable, Extensible RISC-V Core

John Skubic
Purdue University
jskubic@purdue.edu

Jacob R. Stevens
Purdue University
steven69@purdue.edu

Chuan Yean Tan
Purdue University
tan56@purdue.edu

Dr. Mark Johnson
Purdue University
mcjohnso@purdue.edu

Dr. Matthew Swabey
Purdue University
maswabey@purdue.edu

***Abstract*—RISCV-Business is a highly configurable core written in an industry standard hardware description language. The top level architecture cleanly splits functionality of the core into sub-components referred to as configurable components. Configurable components have well defined interfaces within the architecture of the core, allowing for components to be swapped out without impacting functionality of other parts of the design. RISCV-Business includes a interface for extending the base integer ISA with both standard and non-standard RISC-V instruction set extensions that will be referred to as RISC-MGMT (RISC Massively Generic Modification Tie-in).**

***Keywords*—ISA, ASIP, RISC-V, extension, System-On-Chip.**

I. Introduction

System-On-Chips use application specific instruction-set processors, or ASIPs, to improve code density and application performance at a low hardware cost.

One available solution to ASIP prototyping and design is Altera's NIOS extensions of the NIOS II processor [1]. The NIOS extension interface presents the user with a simple template for adding register to register ISA extensions. The extended core can be added to an SoC through the use of Altera's Qsys SoC builder tool.

RISCV-Business provides a RISC-V based solution to the creation of synthesizable ASIPs. The rich library of standard extensions and the predefinition of custom instruction set extension opcodes of the RISC-V ISA makes it an ideal target ISA for an extensible core. RISCV-Business' extension interface, RISC-MGMT (RISC Massively Generic Modification Tie-in), is loosely based off of NIOS' extension interface. RISC-MGMT goes further than NIOS by offering access to memory and the ability to change the control flow of the program. These additions are needed to support all currently standard ISA extensions.

II. Architecture

RISCV-Business was designed to be modular, allowing individual components to be swapped with different versions with varying functionality. For example, the standard two stage in-order pipeline can be replaced with a five stage in-order pipeline. Modules that can be configured are referred to as configurable components. A header file is provided to change the version of configurable components in the core. Figure 1 gives an example configuration in YAML. A script is provided to generate a SystemVerilog header file with the specified configuration. The pipeline, branch predictor, privilege block, caches, bus protocol, and ISA extensions are all configurable components. Refer to Figure 2 to see how these components are connected together.

```yaml
# ISA Configurations
isa_params:
  xlen : 32

# Microarchitectural Configurations
microarch_params:
  # Branch/Jump Configurations
  br_predictor_type : "not_taken"

  # Cache configurations
  cache_config   : "separate"
  dcache_type    : "pass_through"
  icache_type    : "pass_through"

  # Bus configurations
  bus_endianness      : "big"
  bus_interface_type  : "generic_bus_if"
```

Fig. 1. Example YAML configuration file

The standard RV32I pipeline can be swapped with versions of differing pipeline depth. If RISCV-Business is to be used in a small microcontroller, a shallower pipeline depth is needed for a lower impact on area and power; deeper pipeline depths would be instantiated when performance is a more important design constraint.

The branch prediction block will contain a BTB and a branch predictor. Configuring the predictor includes the size of the BTB and the policy of the predictor.

The privilege block contains the implementation of the privileged ISA. This is configurable so the ISA version of the pipeline and privilege block can be updated independently as newer ISA revisions are finalized and released.

The caches layer can support various different cache configurations. Figure 3 shows the template of a cache compatible with RISCV-Business. The cache block is comprised of an incoming request interface, an outgoing request interface, and optionally a collection of user defined signals. The incoming and outgoing request interfaces are generic bus interfaces. The generic bus interface is a collection of signals used by the datapath to request instruction and data transactions. The

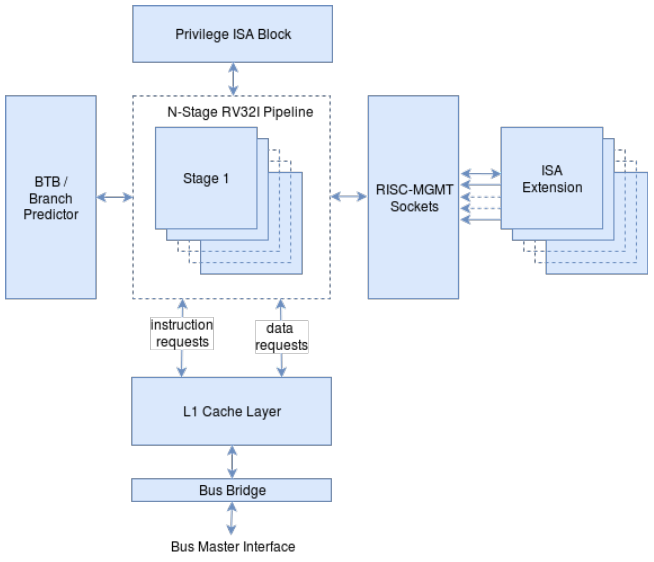

Fig. 2. RISCV-Business Architecture

signals in the generic bus interface match signals one would use to read and write from memory. The signals used for extended functionality are defined by the creator of the cache if needed. An example of a cache requiring extra signals would be a coherent cache.

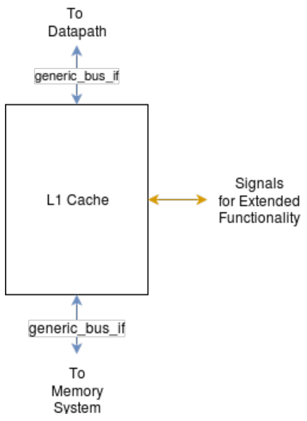

Fig. 3. L1 Cache Template

The bus protocol can be chosen so RISCV-Business has compatibility as a master on any bus system. Bus protocols are implemented as bridges between the generic bus interface and the target bus protocol. The generic bus interface is a pipelined bus that supports both pipelined and nonpipelined buses for maximum throughput.

III. RISC-MGMT

RISC-MGMT, RISC Massively Generic Modification Tie-in, is RISCV-Business' solution to an ISA extension interface. RISC-MGMT will refer to the layer between the standard RV32I pipeline and ISA extensions.

Current RISC-V solutions that allow for ISA extensions, such as RocketCore's Rocket Custom Coprocessor Interface, use a request response mechanism and is limited to register to register and memory instructions [2]. RISC-MGMT can utilize a more tightly coupled connection to the standard core to offer access to the branch and jump unit, load store unit, and read/write access to the register file. By breaking up extensions into separate logical units, RISC-MGMT extensions are compatible with varying pipeline depths.

A. RISC-MGMT Overview

There are four main design goals for RISC-MGMT: the design of a new ISA extension shouldn't be taxing for the implementer, the extension design should be independent of the underlying core's architecture (i.e pipeline depth), all standard RISC-V extensions should be implementable, and custom ISA extensions should be supported. These design goals were the driving factor in partitioning responsibility of RISC-MGMT and extensions.

The goal of supporting all standard RISC-V extensions defined the type of functional unit access given by RISC-MGMT. Register file read and write access is the most obvious functional unit that every extension would need to interact with. Access to branch and jump unit and load store unit is motivated by the compressed ISA extension. Load store unit is also utilized by the atomic ISA extension and may be needed for the place holder transactional memory extension.

At first, it was thought that providing access to the ALU would be useful by reducing the area impact of adding extensions through hardware sharing. There are many problems of adding access to the ALU which outweighed the potential benefits. The main reasons identified are the ALU is likely on the critical path and adding extra multiplexers would reduce the clock speed of the core, the ALU doesn't provide condition codes which may be needed by the extensions, a 32 bit integer ALU may not be sufficient for many extensions, and no concept of a shared ALU exists when connecting the extensions to an out-of-order pipeline.

B. Extension Architecture

ISA extensions are split into three different functional stages analogous to the standard five stage in-order pipeline: decode, execute, and memory. Each stage has a predefined interface that connects to RISC-MGMT. These signals are used to request access to hardware within the standard core. For example, the decode stage sends register read and write information and execute stage will receive the data from the requested registers. Between each stage is a user defined packed bit array of signals being forwarded from one stage to the next.

The decode stage's main responsibility is to parse the instruction and send control signals to the execute and memory stage. The opcode of the instruction should be identified as a parameter so the addition of custom extensions can be automated. Decode notifies RISC-MGMT that the current instruction belongs to the respective extension by asserting an instruction claim signal. The instruction claim signal allows RISC-MGMT to track which extensions are processing instructions and prevents the standard core from throwing an illegal instruction exception. The decode stage may also request bubbles in the core's pipeline. Bubble requests are used when an extension wants to issue multiple instructions from a single instruction fetch; the bubble request would be utilized by the compressed ISA extensions.

The execute stage will perform any needed arithmetic operations. If the write data is produced in this stage, RISC-MGMT will be signaled and the data will be forwarded to the standard core. For multicycle arithmetic operations, a busy signal may be asserted. RISC-MGMT will notify the standard core and the entire pipeline will be stalled. The decision to branch or jump is provided by the execute stage with the target address. The decision to branch will be sent to the branch predictor.

The memory stage has access to the load store unit. A memory busy signal will be asserted until the request has finished. On the completion of a load or store, the fetch stage may trigger a register write. The register write will be forwarded to the standard core.

C. RISC-MGMT Responsibilities

Before going into detail regarding what RISC-MGMT will and won't provide, it is important to note that although individual extensions don't need to know the underlying architecture of the core, the RISC-MGMT layer does. For example, a two stage pipeline and five stage pipeline will each require a different version of RISC-MGMT. This compromise is necessary to abstract the pipeline depth away from extension implementation.

RISC-MGMT's main responsibility is arbitration to functional units inside the standard core. Active instructions will be tracked using tokens in each stage of the pipeline; each token determines which extension owns the instruction. When an active extension attempts to access hardware in the standard core, RISC-MGMT will ensure the request reaches the core and the response returns to the appropriate extension.

Pipeline latches will automatically be inserted between extensions stages depending on the pipeline depth of the standard core. Latches may be placed before the decode stage, between the decode and execute stages, between the execute and memory stages, and after the memory stage. This insertion scheme allows extensions to be compatible with pipeline depths between one and five stages. Figure 4 demonstrates how pipeline latches would be inserted between a two stage and five stage pipeline. The latch between fetch and decode isn't shown because the latch would exist in the standard core and not in RISC-MGMT.

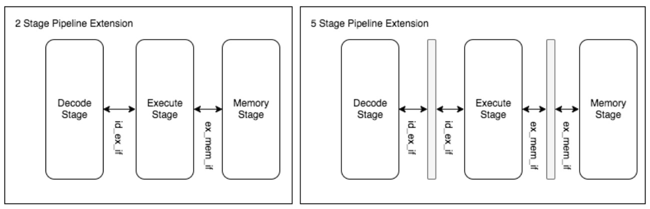

Fig. 4. Insertion of latches between extensions stages

The committing of instructions is handled by RISC-MGMT. When the user provides write data from either the execute or memory stage, the write data is passed through the pipeline by RISC-MGMT. Once the instruction reaches the end of the pipeline, the register file will be updated with the new value. The destination register was identified by the decode stage and is passed through the pipeline in a similar manner. RISC-MGMT was given this responsibility due to different forwarding logic in pipelines of different depths. Register writes by extensions are checked against register reads of both the standard core and the other extensions to ensure correctness without a loss in IPC.

Exceptions may be thrown by either the execute or memory stages. When an exception is encountered, RISC-MGMT will send the exception to the standard core's exception handling. Each extension will be assigned a unique exception ID. The exception ID will be set in the exception cause register so software can determine which extension threw the exception.

To help mitigate the increase in power consumption from added hardware, RISC-MGMT will provide clock gating. Little extra logic is needed to implement clock gating since RISC-MGMT already keeps track of which extensions have active instructions in each stage of the pipeline through tokens. When an extension has no active instructions, its clock will be gated.

IV. Conclusion

RISCV-Business provides a configurable, extensible RISC-V core. The design is provided in the industry standard SystemVerilog hardware description language to make it easily accessible by a wide variety of hardware designers. The core can be extended to use both standard and non-standard ISA extensions through the RISC-MGMT extension interface. These characteristics make RISCV-Business an ideal candidate for ASIP and SoC prototyping and design.

Acknowledgment

The authors would like to thank the entire Purdue SoCet (System-On-Chip Extension Technologies) team for their continued input and feedback on the RISCV-Business project.

References

[1] Altera. (2015) *Nios II Custom Instruction User Guide* [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_nios2_custom_instruction.pdf

[2] Yunsup Lee. (2015) *RISC-V Rocket Chip SoC Generator in Chisel* [Online]. Available: https://riscv.org/wp-content/uploads/2015/02/riscv-rocket-chip-generator-tutorial-hpca2015.pdf