The RoCC Doc V2:

An Introduction to the Rocket Custom Coprocessor Interface

|  |
| --- |
| Anuj Rao  Taylor’s Bespoke Silicon Group & UCSD |

## **Goal**

The RoCC interface enables the integration of custom coprocessors or accelerators to RISC-V cores. The purpose of the document is to understand the RoCC interface signals from the perspective of designing a RISC-V compatible accelerator in Verilog.

## **RoCC Interface Overview**

The RoCC interface has a basic set of signals that are generally necessary for accelerators. We refer to these as the ***default***RoCC interface. However, the RoCC interface also provides some configurable extensions which may be required by accelerators depending on their functionality. We refer to these as the ***extended*** RoCC interface.

The ***default*** RoCC interface signals may be classified into the following groups of signals

1. **Core control (CC):** for co-ordination between an accelerator and Rocket core
2. **Register mode (Core):** for exchange of data between an accelerator and Rocket core
3. **Memory mode (Mem):** for communication between an accelerator and L1-D Cache

The ***extended*** RoCC interface can provide for the following groups of signals

1. **Uncached Tile Link (UTL):** for communication between an accelerator & L2 memory
2. **Floating Point Unit (FPU):** for an accelerator to send and receive data from an FPU
3. **Control Status Register (CSR):** used by Linux on the core to recognize the accelerator
4. **Page Table Walker (PTW):** for address translation from an accelerator

The system-level diagram in *Figure 1* shows the RoCC interface for an accelerator. We prefix acronyms assigned above to subgroups in verilog signal names through rest of the document.



## **RoCC Interface Description**

The signal names for both the ***default*** and the ***extended*** RoCC interfaces are elaborated from the verilog generated from an example Chisel-based accelerator. These verilog interface names would be used by HLS tools to automatically generate the Verilog code for accelerators; optionally a Verilog interface that employs structs could be used as well.

***Please note that signal directions & descriptions are from the accelerator’s perspective.***

### Default RoCC Interface

The default RoCC interface is composed of 3 subgroups, namely the Control, Register and Memory mode signals. A comprehensive list of the signals, with short descriptions and their default values, is provided from a designers’ point of view.

1. Core control (CC)

The core control signals listed in *Table 1* ensure co-ordination between the core and accelerator. All the signal names are prefixed with “cc\_” to denote the group they belong to.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Direction | Signal name | Default value | Description | | |
| output | cc\_busy\_o | 0 | Set during in-flight memory requests | | |
| input | cc\_status\_i | 0 | Set when a privileged process runs on the core | | |
| output | cc\_interrupt\_o | 0 | Set for invalid instructions or to indicate any problem | | |
| input | cc\_exception\_i | 0 | Set by core to trigger exception behavior | | |
| input | cc\_host\_id\_i | host\_id | Used to distinguish between command packets from different hosts.  Note: Bit width = log2(number\_of\_cores) | | |

Table 1: Core control signals

1. Register mode (Core)

The Register mode signals are composed of the RoCC Command and Response subgroups. All the signal names are prefixed with “core\_” to denote their source.

The RoCC Command signals are used by the core to send instructions to the accelerator and are driven directly by the RoCC Instruction of the RISC-V ISA. The RoCC instruction is shown in *Figure 2* which indicates bit widths and positions above and below the respective fields within the instruction. *Table 2* contains a description of signals in the RoCC Command subgroup.

An RoCC response to the command (if expected) is sent by the accelerator using the response interface. The response signals are described in *Table 3.*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 25 | | | | 24 20 | | | 19 15 | | | 14 | 13 | 12 | T | | | 6 0 | | |
| funct7 | | | | rs2 | | | rs1 | | | xd | xs1 | xs2 | rd | | | opcode | | |
| 7 | | | | 5 | | | 5 | | | 1 | 1 | 1 | 5 | | | 7 | | |

Figure 2: RoCC Instruction format

|  |  |  |  |
| --- | --- | --- | --- |
| Direction | Signal name | Default value | Description |
| output | core\_cmd\_ready\_o | 0 | Control lines |
| input | core\_cmd\_valid\_i | 0 |
| input [6:0] | core\_cmd\_inst\_funct\_i | funct7 | For different kinds of accelerator instructions; Value is designers’ choice |
| input [4:0] | core\_cmd\_inst\_rs2\_i | rs2 | Source register IDs |
| input [4:0] | core\_cmd\_inst\_rs1\_i | rs1 |
| input | core\_cmd\_inst\_xd\_i | xd | Set if destination register exists |
| input | core\_cmd\_inst\_xs1\_i | xs1 | Set if source registers exist |
| input | core\_cmd\_inst\_xs2\_i | xs2 |
| input [4:0] | core\_cmd\_inst\_rd\_i | rd | Destination register ID |
| input [6:0] | core\_cmd\_inst\_opcode\_i | 0x1/0x2/0x3/ 0x4 | Custom instruction opcode may be used in case of multiple accelerators |
| input [63:0] | core\_cmd\_rs1\_i | rs1\_data | Source register data |
| input [63:0] | core\_cmd\_rs2\_i | rs2\_data |

Table 2: Register mode RoCC command signals

|  |  |  |  |
| --- | --- | --- | --- |
| Direction | Signal name | Default Value | Description |
| input | core\_resp\_ready\_i | 0 | Control lines |
| output | core\_resp\_valid\_o | 0 |
| output[4:0] | core\_resp\_rd\_o | rd | Destination register ID in the response |
| output[63:0] | core\_resp\_data\_o | rd\_data | Destination register data in the response |

Table 3: Register mode RoCC response signals

3) Memory mode (Mem)

The memory interface is composed of Memory request and Memory response subgroups. All the signal names are prefixed with “mem\_” to denote the group they belong to.

Memory requests from the accelerator are sent out using signal described in *Table 4*.

Memory responses to the accelerator are passed through interface described in *Table 5*. There is a response for both load and store requests. Usually an accelerator may use just the tag and data fields from the response. Please note that there is no ready signal from the accelerator to acknowledge the acceptance of memory responses, which means its is expected that accelerator should be ready to accept data in every cycle.

|  |  |  |  |
| --- | --- | --- | --- |
| Direction | Signal name | Default Value | Description |
| input | mem\_req\_ready\_i | 0 | Control lines |
| output | mem\_req\_valid\_o | 0 |
| output[39:0] | mem\_req\_addr\_o | addr | Memory address corresponds to read/write |
| output[9:0] | mem\_req\_tag\_o | tag | Unique identity assigned to every memory request especially to the same memory location to support out-of-order responses |
| output[4:0] | mem\_req\_cmd\_o | cmd | Memory request opcode [0x0000=load, 0x0001=store] |
| output[2:0] | mem\_req\_typ\_o | typ | Width of response; [0x000=8bits, 0x001=16bits, 0x010=32 bits, 0x011=64 bits] |
| output | mem\_req\_phys\_o | 1 | De-asserted if addresses are virtual and need translation |
| output[63:0] | mem\_req\_data\_o | w\_data | Store data |

Table 4: Memory request signals

|  |  |  |  |
| --- | --- | --- | --- |
| Direction | Signal name | Default Value | Description |
| input | mem\_resp\_valid\_i | 0 | Control |
| input [39:0] | mem\_resp\_addr\_i | addr | Displays load/store request addr |
| input [9:0] | mem\_resp\_tag\_i | tag | To differentiate between responses to multiple in-flight requests |
| input [4:0] | mem\_resp\_cmd\_i | cmd | Returns command code of request |
| input [2:0] | mem\_resp\_typ\_i | typ | Indicates width of the data in response |
| input [63:0] | mem\_resp\_data\_i | data | Contains data response to a load request |
| input | mem\_resp\_nack\_i | 0 | (To be known) |
| input | mem\_resp\_replay\_i | 0 | (To be known) |
| input | mem\_resp\_has\_data\_i | 0 | Set if the data field is valid in response |
| input [63:0] | mem\_resp\_data\_word\_bypass\_i | bypass\_data | A store may be bypassed to a read response in the same cycle |
| input [63:0] | mem\_resp\_store\_data\_i | w\_data | Returns the data stored in store request during the corresponding response |

Table 5: Memory response signals

### Extended RoCC Interface

These signals are included only if RoCC is configured to include them. Please note the below list of signals is not comprehensive but only for representational purposes.

1. Uncached Tile Link (UTL)

The UTL group is composed of 2 subgroups namely *aUTL* which is arbitrated uncached tile link signals and *UTL*signals.

aUTL is arbitrated with the I-cache link to L2 memory system. The aUTL signals composed of AUTL acquire and grant subgroups as listed in *Table 6 and 7* respectively are prefixed with “autl\_” to denote the group they belong to.

The UTLsignals are a vector of signals of the same type as aUTL. The size of the UTL vector is equal to the number of independent memory channels, which is configurable in the rocket core. However these signals are yet to be added in the document.

|  |  |
| --- | --- |
| Direction | Signal name |
| input | autl\_acquire\_ready\_i |
| output | autl\_acquire\_valid\_o |
| output[25:0] | autl\_acquire\_bits\_addr\_block\_o |
| output[2:0] | autl\_acquire\_bits\_client\_xact\_id\_o |
| output[1:0] | autl\_acquire\_bits\_addr\_beat\_o |
| output | autl\_acquire\_bits\_is\_builtin\_type\_o |
| output[2:0] | autl\_acquire\_bits\_a\_type\_o |
| output[16:0] | autl\_acquire\_bits\_union\_o |
| output[127:0] | autl\_acquire\_bits\_data\_o |

Table 6: AUTL acquire signals

|  |  |
| --- | --- |
| Direction | Signal name |
| output | autl\_grant\_ready\_o |
| input | autl\_grant\_valid\_i |
| input [1:0] | autl\_grant\_bits\_addr\_beat\_i |
| input [2:0] | autl\_grant\_bits\_client\_xact\_id\_i |
| input [3:0] | autl\_grant\_bits\_manager\_xact\_i |
| input | autl\_grant\_bits\_is\_builtin\_type\_i |
| input [3:0] | autl\_grant\_bits\_g\_type\_i |
| input [127:0] | autl\_grant\_bits\_data\_i |
| Table 7: AUTL grant signals | |

1. Floating Point Unit (FPU)

This interface may be used by the accelerator if it has a floating point unit attached to it. All the signal names are prefixed with “fpu\_” to denote the group they belong to. The interface is composed of FP request and FP response sub-groups as listed in *Table 8* and *Table 9* respectively.

|  |  |
| --- | --- |
| Direction | Signal name |
| input | fpu\_req\_ready\_i |
| output | fpu\_req\_valid\_o |
| output[4:0] | fpu\_req\_bits\_cmd\_o |
| output | fpu\_req\_bits\_ldst\_o |
| output | fpu\_req\_bits\_wen\_o |
| output | fpu\_req\_bits\_ren1\_o |
| output | fpu\_req\_bits\_ren2\_o |
| output | fpu\_req\_bits\_ren3\_o |
| output | fpu\_req\_bits\_swap12\_o |
| output | fpu\_req\_bits\_swap23\_o |
| output | fpu\_req\_bits\_single\_o |
| output | fpu\_req\_bits\_fromint\_o |
| output | fpu\_req\_bits\_toint\_o |
| output | fpu\_req\_bits\_fastpipe\_o |
| output | fpu\_req\_bits\_fma\_o |
| output | fpu\_req\_bits\_div\_o |
| output | fpu\_req\_bits\_sqrt\_o |
| output | fpu\_req\_bits\_round\_o |
| output | fpu\_req\_bits\_wflags\_o |
| output[2:0] | fpu\_req\_bits\_rm\_o |
| output[1:0] | fpu\_req\_bits\_typ\_o |
| output[64:0] | fpu\_req\_bits\_in1\_o |
| output[64:0] | fpu\_req\_bits\_in2\_o |
| output[64:0] | fpu\_req\_bits\_in3\_o |
| Table 8: FP Request signals | |

|  |  |
| --- | --- |
| Direction | Signal name |
| output | fpu\_resp\_ready\_o |
| input | fpu\_resp\_valid\_i |
| input [64:0] | fpu\_resp\_bits\_data\_i |
| input [4:0] | fpu\_resp\_bits\_exc\_i |
| Table 9: FP Response signals | |

1. Control Status Registers (CSR)

Control status registers may be used optionally within the accelerator. If they exist, the interface signals in *Table 10* can be used by the operating system on the core to memory map the accelerator. Its mechanism is to be added and will be discussed in later versions of the document. All the signal names are prefixed with “csr\_” to denote the group they belong to.

|  |  |
| --- | --- |
| Direction | Signal name |
| input [11:0] | csr\_waddr\_i |
| input [63:0] | csr\_wdata\_i |
| input | csr\_wen\_i |
| output [63:0] | csr\_rdata\_o |
| Table 10: Accelerator CSR signals | |

1. Page Table Walker (PTW)

The page table walker signals are used for address translation from the accelerator (if the core isn’t already handling it). These signals will be added as a part of future work. All the signal names will be prefixed with “ptw\_” to denote the group they belong to. Please stay tuned for updates.

## **Resources**

For more details about any information in the document, please do check some useful resources of the Berkeley RoCC tutorials which were referred while making this document:

* <http://www-inst.eecs.berkeley.edu/~cs250/sp16/disc/Disc02.pdf>
* <http://www-inst.eecs.berkeley.edu/~cs250/sp16/disc/Disc05.pdf>
* <http://www-inst.eecs.berkeley.edu/~cs250/sp16/assignments/lab4.pdf>
* <http://www-inst.eecs.berkeley.edu/~cs250/fa13/handouts/lab3-sumaccel.pdf>

Moreover, there is an example that integrates the Secure Hashing Algorithm (SHA) accelerator with the rocket core and is also tested against sample programs. The SHA repository (with fixes by UCSD) and the rocket implementation are listed below. Follow the instructions in the readme for simulations. We will discuss this in greater detail in the following versions of the document.

* <https://github.com/ucb-bar/rocc-template>
* <https://github.com/ucb-bar/rocket-chip>

## **Conclusion**

This document will be helpful in designing an accelerator that is compatible with the RoCC interface. In upcoming versions of the document, we will provide a tutorial on integrating this accelerator to the core and actually running test programs with custom instructions to use the accelerator.