Ibex Documentation

Release 0.1.dev76+g056cb44.d20220830

Aug 30, 2022
# CONTENTS

1 **Introduction to Ibex** 3  
   1.1 Standards Compliance .................................................. 3  
   1.2 Synthesis Targets ..................................................... 4  
   1.3 Licensing ................................................................. 4  

2 **Ibex User Guide** 5  
   2.1 System and Tool Requirements ........................................ 5  
   2.2 Getting Started with Ibex ............................................. 6  
   2.3 Core Integration ....................................................... 6  
   2.4 Examples ................................................................. 10  

3 **Ibex Reference Guide** 11  
   3.1 Pipeline Details ....................................................... 11  
   3.2 Instruction Cache .................................................... 13  
   3.3 Instruction Fetch ..................................................... 18  
   3.4 Instruction Decode and Execute .................................... 19  
   3.5 Load-Store Unit ....................................................... 22  
   3.6 Register File ........................................................... 24  
   3.7 Control and Status Registers ...................................... 27  
   3.8 Performance Counters ............................................... 36  
   3.9 Exceptions and Interrupts ......................................... 38  
   3.10 Physical Memory Protection (PMP) ............................... 41  
   3.11 Security Features .................................................. 42  
   3.12 Debug Support ...................................................... 45  
   3.13 Tracer ................................................................. 45  
   3.14 Verification .......................................................... 47  
   3.15 Co-simulation System .............................................. 51  
   3.16 Test Plan ............................................................. 54  
   3.17 Coverage Plan ...................................................... 56  
   3.18 RISC-V Formal Interface ......................................... 64  
   3.19 History ............................................................... 64  

4 **Ibex Developer Guide** 67  
   4.1 The Ibex Concierge .................................................. 67
Ibex is a production-quality open source 32 bit RISC-V CPU core written in SystemVerilog. The CPU core is heavily parametrizable and well suited for embedded control applications. Ibex is being extensively verified and has seen multiple tape-outs.

You are now reading the Ibex documentation. The documentation is split into four parts.

The **Overview documentation** looks at Ibex from high up. It answers questions like what are the high-level properties of Ibex, which standards is Ibex following, and where is it typically used.

The **User Guide** provides all necessary information to use Ibex. It is aimed at hardware developers integrating Ibex into a design, and software developers writing software running on Ibex.

The **Reference Guide** provides background information. It describes the design in detail, discusses the verification approach and the resulting testbench structures, and generally helps to understand Ibex in depth.

The **Developer Guide** is aimed at people making changes to Ibex itself. Since Ibex is open source, every user of Ibex is encouraged to learn how to adapt Ibex to their use case, and be part of the open development process.
Ibex is a production-quality open source 32-bit RISC-V CPU core written in SystemVerilog. The CPU core is heavily parametrizable and well suited for embedded control applications. Ibex is being extensively verified and has seen multiple tape-outs.

Read on for more information Ibex in general: what standards it implements, what synthesis targets it supports, and what legal and financial obligations you have when using Ibex.

1.1 Standards Compliance

Ibex is a standards-compliant 32 bit RISC-V processor. It follows these specifications:

- RISC-V External Debug Support, version 0.13.2
- RISC-V Bit-Manipulation Extension, version 1.0.0 and version 0.93 (draft from January 10, 2021)
- PMP Enhancements for memory access and execution prevention on Machine mode (smpmp) version 0.9.3

Many features in the RISC-V specification are optional, and Ibex can be parametrized to enable or disable some of them.

Ibex can be parametrized to support either of the following two instruction sets.

- The RV32I Base Integer Instruction Set, version 2.1
- The RV32E Base Integer Instruction Set, version 1.9 (draft from June 8, 2019)

In addition, the following instruction set extensions are available.

<table>
<thead>
<tr>
<th>Extension</th>
<th>Version</th>
<th>Configurability</th>
</tr>
</thead>
<tbody>
<tr>
<td>C: Standard Extension for Compressed Instructions</td>
<td>2.0</td>
<td>always enabled</td>
</tr>
<tr>
<td>M: Standard Extension for Integer Multiplication and Division</td>
<td>2.0</td>
<td>optional</td>
</tr>
<tr>
<td>B: Standard Extension for Bit-Manipulation Instructions</td>
<td>1.0.0 + 0.93</td>
<td>optional</td>
</tr>
<tr>
<td>Zicsr: Control and Status Register Instructions</td>
<td>2.0</td>
<td>always enabled</td>
</tr>
<tr>
<td>Zifencei: Instruction-Fetch Fence</td>
<td>2.0</td>
<td>always enabled</td>
</tr>
</tbody>
</table>
Most content of the RISC-V privileged specification is optional. Ibex currently supports the following features according to the RISC-V Privileged Specification, version 1.11:

- M-Mode and U-Mode
- All CSRs listed in *Control and Status Registers*
- Performance counters as described in *Performance Counters*
- Vectorized trap handling as described at *Exceptions and Interrupts*

See *PMP Enhancements* for more information on Ibex’s experimental and optional support for the PMP Enhancement proposal from the Trusted Execution Environment (TEE) working group.

### 1.2 Synthesis Targets

#### 1.2.1 ASIC Synthesis

ASIC synthesis is supported for Ibex. The whole design is completely synchronous and uses positive-edge triggered flip-flops, except for the register file, which can be implemented either with latches or with flip-flops. See *Register File* for more details. The core occupies an area of roughly 24 kGE when using the flip-flop-based register file and implementing the RV32IMC ISA, or 15 kGE when implementing the RV32EC ISA.

#### 1.2.2 FPGA Synthesis

FPGA Synthesis is supported for Ibex. The FPGA-optimized register file implementation should be used. The flip-flop-based register file is also compatible with FPGA synthesis, however it may result in significantly higher resource utilization. Since latches are not well supported on FPGAs, the latch-based register file should not be used.

### 1.3 Licensing

Ibex is released under the Apache license, version 2.0.

Ibex can be used, modified, and distributed for any purpose (including commercial) and without any royalties. There are some requirements on including copyright notices and the original license.

Please see the LICENSE file in the source code for the full (and legally binding) license text.

Even though the license doesn’t require it, we appreciate feedback and contributions to make Ibex work better for everyone. Please open an issue for bug reports, questions, or suggested improvements, or a pull request if you’d like to contribute code.

---

**Note:** Commercial support for Ibex is available from lowRISC.
The Ibex User Guide provides all necessary information to use Ibex. It is aimed at hardware developers integrating Ibex into a design, and software developers writing software running on Ibex.

2.1 System and Tool Requirements

The Ibex CPU core is written in SystemVerilog. We try to achieve a balance between the used language features (as described in our style guide) and reasonably wide tool support.

The following tools are known to work with the RTL code of Ibex. Please file an issue if you experience problems with any of the listed tools, or if you have successfully used a tool with Ibex which is not listed here.

- Synopsys Design Compiler
- Xilinx Vivado, version 2020.2 and up.
- Verilator, version 4.028 and up.
- Synopsys VCS, version at least 2020.03-SP2.
- Cadence Incisive/Xcelium
- Mentor Questa
- Aldec Riviera Pro

To run the UVM testbench a RTL simulator which supports SystemVerilog and UVM 1.2 is required. The documentation of riscv-dv contains a list of supported simulators.

To compile code that runs on Ibex, you’ll need a RISC-V toolchain. This isn’t part of the core as such, but is necessary for verification. See the Verification section of the Reference Guide for more details about which toolchains the project currently uses for testing.

2.1.1 Tools with known issues

Not all EDA tools have enough SystemVerilog support to be able to work with the Ibex code base. Users of such tools are encouraged to file issues with the vendor. As a workaround, tools like sv2v can pre-process the source code to an older version of Verilog.

- Intel (Altera) Quartus Prime Lite and Standard are not supported due to insufficient SystemVerilog support (issue #117).
- Yosys cannot be used directly due to insufficient SystemVerilog support (issue #60). The syn folder in the Ibex repository contains scripts to use sv2v together with Yosys.
- Icarus Verilog is not supported due to insufficient SystemVerilog support.
2.2 Getting Started with Ibex

This page discusses initial steps and requirements to start using Ibex in your design.

2.2.1 Register File

Ibex comes with three different register file implementations that can be selected using the enumerated parameter RegFile defined in rtl/ibex_pkg.sv. Depending on the target technology, either the flip-flop-based (“ibex_pkg::RegFileFF”, default), the latch-based (“ibex_pkg::RegFileLatch”) or an FPGA-targeted (“ibex_pkg::RegFileFPGA”) implementation should be selected. For more information about the three register file implementations and their trade-offs, check out Register File.

2.2.2 Identification CSRs

The RISC-V Privileged Architecture specifies several read-only CSRs that identify the vendor and micro-architecture of a CPU. These are mvendorid, marchid and mimpid. The fixed, read-only values for these CSRs are defined in rtl/ibex_pkg.sv. Implementers should carefully consider appropriate values for these registers. Ibex, as an open source implementation, has an assigned architecture ID (marchid) of 22. (Allocations are specified in marchid.md of the riscv-isa-manual repository.) If significant changes are made to the micro-architecture a different architecture ID should be used. The vendor ID and implementation ID (mvendorid and mimpid) both read as 0 by default, meaning non-implemented. Implementers may wish to use other values here. Please see the RISC-V Privileged Architecture specification for more details on what these IDs represent and how they should be chosen.

2.3 Core Integration

The main module is named ibex_top and can be found in ibex_top.sv. Note that the core logic is split-out from the register file and RAMs under ibex_top. This is to facilitate a dual-core lockstep implementation (see Security Features).

Below, the instantiation template is given and the parameters and interfaces are described.

2.3.1 Instantiation Template

```verbatim
ibex_top #(  .PMPEnable ( 0 ),  .PMPGranularity ( 0 ),  .PMPNumRegions ( 4 ),  .MHPMCounterNum ( 0 ),  .MHPMCounterWidth ( 40 ),  .RV32E ( 0 ),  .RV32M ( ibex_pkg::RV32MFast ),  .RV32B ( ibex_pkg::RV32BNone ),  .RegFile ( ibex_pkg::RegFileFF ),  .ICache ( 0 ),  .ICacheECC ( 0 ),  .ICacheScramble ( 0 ),  .BranchPrediction ( 0 ),  .SecureIbex ( 0 ),  .RndCnstLfsrSeed ( ibex_pkg::RndCnstLfsrSeedDefault ),  .RndCnstLfsrPerm ( ibex_pkg::RndCnstLfsrPermDefault ),
```

(continues on next page)
.DbgTriggerEn ( 0 ),
.DmHaltAddr ( 32'h1A110800 ),
.DmExceptionAddr ( 32'h1A110808 )
)

u_top(

// Clock and reset
.clk_i (),
.rst_ni (),
.test_en_i (),
.scan_rst_ni (),
.ram_cfg_i ()

// Configuration
.hart_id_i (),
.boot_addr_i ()

// Instruction memory interface
.instr_req_o (),
.instr_gnt_i (),
.instr_rvalid_i (),
.instr_addr_o (),
.instr_rdata_i (),
.instr_rdata_intg_i (),
.instr_err_i ()

// Data memory interface
.data_req_o (),
.data_gnt_i (),
.data_rvalid_i (),
.data_we_o (),
.data_be_o (),
.data_addr_o (),
.data_wdata_o (),
.data_wdata_intg_o (),
.data_rdata_i (),
.data_rdata_intg_i (),
.data_err_i ()

// Interrupt inputs
.irq_software_i (),
.irq_timer_i (),
.irq_external_i (),
.irq_fast_i (),
.irq_nm_i ()

// Debug interface
.debug_req_i (),
.crash_dump_o ()

// Special control signals
.fetch_enable_i (),
.alert_minor_o (),
.alert_major_internal_o (),
.alert_major_bus_o (),
.core_sleep_o ()
);
## 2.3.2 Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type/Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMPEnable</td>
<td>bit</td>
<td>0</td>
<td>Enable PMP support</td>
</tr>
<tr>
<td>PMPGranularity</td>
<td>int (0..31)</td>
<td>0</td>
<td>Minimum granularity of PMP address matching</td>
</tr>
<tr>
<td>PMPNumRegions</td>
<td>int (1..16)</td>
<td>4</td>
<td>Number implemented PMP regions (ignored if PMPEnable == 0)</td>
</tr>
<tr>
<td>MHPMCounterNum</td>
<td>int (0..10)</td>
<td>0</td>
<td>Number of performance monitor event counters</td>
</tr>
<tr>
<td>MHPMCounterWd</td>
<td>int (64..1)</td>
<td>40</td>
<td>Bit width of performance monitor event counters</td>
</tr>
<tr>
<td>RV32E</td>
<td>bit</td>
<td>0</td>
<td>RV32E mode enable (16 integer registers only)</td>
</tr>
<tr>
<td>RV32M</td>
<td>ibex_pkg::rv32m_e</td>
<td></td>
<td>RV32M (multiply) extension select:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“ibex_pkg::RV32MNone”: No M-extension</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“ibex_pkg::RV32MSlow”: Slow multi-cycle multiplier, iterative divider</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“ibex_pkg::RV32MFast”: 3-4 cycle multiplier, iterative divider</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“ibex_pkg::RV32MSingleCycle”: 1-2 cycle multiplier, iterative divider</td>
</tr>
<tr>
<td>RV32B</td>
<td>ibex_pkg::rv32b_e</td>
<td></td>
<td>RV32B (manipulation) extension select:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“ibex_pkg::RV32BBalanced”: Sub-extensions Zba, Zbb, Zbs, Zbf and Zbt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“ibex_pkg::RV32BOTEarlGrey”: All sub-extensions except Zbe</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“ibex_pkg::RV32BFull”: All sub-extensions</td>
</tr>
<tr>
<td>RegFile</td>
<td>ibex_pkg::RegFile_e</td>
<td></td>
<td>Register file implementation select:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“ibex_pkg::RegFileFF”: Generic flip-flop-based register file</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“ibex_pkg::RegFileFPGA”: Register file for FPGA targets</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“ibex_pkg::RegFileLatch”: Latch-based register file for ASIC targets</td>
</tr>
<tr>
<td>BranchTargetALU</td>
<td>bit</td>
<td>0</td>
<td>EXPERIMENTAL - Enables branch target ALU removing a stall cycle from taken branches</td>
</tr>
<tr>
<td>WritebackStage</td>
<td>bit</td>
<td>0</td>
<td>EXPERIMENTAL - Enables third pipeline stage (writeback) improving performance of loads and stores</td>
</tr>
<tr>
<td>ICache</td>
<td>bit</td>
<td>0</td>
<td>EXPERIMENTAL Enable instruction cache instead of prefetch buffer</td>
</tr>
<tr>
<td>ICacheECC</td>
<td>bit</td>
<td>0</td>
<td>EXPERIMENTAL Enable SECDED ECC protection in ICache (if ICache == 1)</td>
</tr>
<tr>
<td>ICacheScramble</td>
<td>bit</td>
<td>0</td>
<td>EXPERIMENTAL Enabling this parameter replaces tag and data RAMs of ICache with scrambling RAM primitives.</td>
</tr>
<tr>
<td>BranchPred</td>
<td>bit</td>
<td>0</td>
<td>EXPERIMENTAL Enable Static branch prediction</td>
</tr>
<tr>
<td>SecureIbex</td>
<td>bit</td>
<td>0</td>
<td>EXPERIMENTAL Enable various additional features targeting secure code execution. Note: SecureIbex == 1'b1 and RV32M == ibex_pkg::RV32MNone is an illegal combination.</td>
</tr>
<tr>
<td>RndCnstSeed</td>
<td>lfsr_seed_t</td>
<td>above</td>
<td>Set the starting seed of the LFSR used to generate dummy instructions (only relevant when SecureIbex == 1'b1)</td>
</tr>
<tr>
<td>RndCnstPerm</td>
<td>lfsr_perm_t</td>
<td>above</td>
<td>Set the permutation applied to the output of the LFSR used to generate dummy instructions (only relevant when SecureIbex == 1'b1)</td>
</tr>
<tr>
<td>DBGTriggerEn</td>
<td>bit</td>
<td>0</td>
<td>Enable debug trigger support (one trigger only)</td>
</tr>
<tr>
<td>DmHaltAddr</td>
<td>int</td>
<td>0x1A1</td>
<td>Address to jump to when entering Debug Mode</td>
</tr>
<tr>
<td>DmExceptionAddr</td>
<td>int</td>
<td>0x1A1</td>
<td>Address to jump to when an exception occurs while in Debug Mode</td>
</tr>
</tbody>
</table>

Any parameter marked EXPERIMENTAL when enabled is not verified to the same standard as the rest of the Ibex core.

Note that Ibex uses SystemVerilog enum parameters e.g. for RV32M and RV32B. This is well supported by most tools but some care is needed when overriding these parameters at the top level:

- Synopsys VCS does not support overriding enum and string parameters at the top level via command line. As a workaround, SystemVerilog defines are used in Ibex top level files simulated with VCS. These defines can be set via command line.
• Yosys does not support overriding enum parameters at the top level by setting enum names. Instead, the enum values need to be used.

## 2.3.3 Interfaces

<table>
<thead>
<tr>
<th>Signal(s)</th>
<th>Width</th>
<th>Dir</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk_i</td>
<td>1</td>
<td>in</td>
<td>Clock signal</td>
</tr>
<tr>
<td>rst_ni</td>
<td>1</td>
<td>in</td>
<td>Active-low asynchronous reset</td>
</tr>
<tr>
<td>test_en_i</td>
<td>1</td>
<td>in</td>
<td>Test input, enables clock and allows test control of reset.</td>
</tr>
<tr>
<td>scan_rst_ni</td>
<td>in</td>
<td></td>
<td>Test controlled reset. If DFT not used, tie off to 1.</td>
</tr>
<tr>
<td>ram_cfg_i</td>
<td>10</td>
<td>in</td>
<td>RAM configuration inputs, routed to the icache RAMs</td>
</tr>
<tr>
<td>hart_id_i</td>
<td>32</td>
<td>in</td>
<td>Hart ID, usually static, can be read from Hardware Thread ID (mhartid) CSR</td>
</tr>
<tr>
<td>boot_addr_i</td>
<td>32</td>
<td>in</td>
<td>First program counter after reset = boot_addr_i + 0x80, see Exceptions and Interrupts</td>
</tr>
<tr>
<td>instr_*</td>
<td></td>
<td></td>
<td>Instruction fetch interface, see Instruction Fetch</td>
</tr>
<tr>
<td>data_*</td>
<td></td>
<td></td>
<td>Load-store unit interface, see Load-Store Unit</td>
</tr>
<tr>
<td>irq_*</td>
<td></td>
<td></td>
<td>Interrupt inputs, see Exceptions and Interrupts</td>
</tr>
<tr>
<td>scramble_*</td>
<td></td>
<td></td>
<td>Scrambling key interface, see Instruction Cache</td>
</tr>
<tr>
<td>debug_*</td>
<td></td>
<td></td>
<td>Debug interface, see Debug Support</td>
</tr>
<tr>
<td>crash_dump_o</td>
<td></td>
<td></td>
<td>A set of signals that can be captured on reset to aid crash debugging.</td>
</tr>
<tr>
<td>double_fault_seen_o</td>
<td></td>
<td></td>
<td>Double fault was observed, see Double Fault Detection</td>
</tr>
<tr>
<td>fetch_enable_i</td>
<td>4</td>
<td>in</td>
<td>Allow the core to fetch instructions. If this bit is set low, the core will pause fetching new instructions. A multi-bit encoding scheme is used. See FetchEnableOn / FetchEnableOff in rtl/ibex_pkg.sv</td>
</tr>
<tr>
<td>core_sleep_o</td>
<td></td>
<td>out</td>
<td>Core in WFI with no outstanding data or instruction accesses. Deasserts if an external event (interrupt or debug req) wakes the core up</td>
</tr>
<tr>
<td>alert_minor_o</td>
<td></td>
<td>out</td>
<td>Core has detected a fault which it can safely recover from. Can be used by a system to log errors over time and detect tampering / attack. This signal is a pulse, one cycle per alert.</td>
</tr>
<tr>
<td>alert_major_internal_o</td>
<td></td>
<td>out</td>
<td>Core has detected an internal fault which cannot be recovered from. Can be used by a system to reset the core and possibly take other remedial action. This signal is a pulse, but might be set for multiple cycles per alert.</td>
</tr>
<tr>
<td>alert_major_bus_o</td>
<td></td>
<td>out</td>
<td>Core has detected a bus fault which cannot be recovered from. Can be used by a system to reset the core and possibly take other remedial action. This signal is a pulse, but might be set for multiple cycles per alert.</td>
</tr>
</tbody>
</table>
2.4 Examples

To make use of Ibex it has to be integrated as described in Core Integration.

2.4.1 FPGA

A minimal example for the Artyst A7 FPGA Development board is provided. In this example Ibex is directly linked to a SRAM memory instance. Four LEDs from the board are connected to the data bus and are updated each time when a word is written. The memory is separated into an instruction and data section. The instructions memory is initialized at synthesis time by reading the output from the software build. The software writes to the data section the complementary lower for bits of a word every second resulting in blinking LEDs.

Find the description of how to build and program the Arty board in examples/fpga/artya7/README.md.
The Ibex Reference Guide provides background information. It describes the design in detail, discusses the verification approach and the resulting testbench structures, and generally helps to understand Ibex in depth.

Figure 3.1: Ibex Pipeline

### 3.1 Pipeline Details

Ibex has a 2-stage pipeline, the 2 stages are:

- **Instruction Fetch (IF)** Fetches instructions from memory via a prefetch buffer, capable of fetching 1 instruction per cycle if the instruction side memory system allows. See *Instruction Fetch* for details.

- **Instruction Decode and Execute (ID/EX)** Decodes fetched instruction and immediately executes it, register read and write all occurs in this stage. Multi-cycle instructions will stall this stage until they are complete See *Instruction Decode and Execute* for details.

All instructions require two cycles minimum to pass down the pipeline. One cycle in the IF stage and one in the ID/EX stage. Not all instructions can complete in the ID/EX stage in one cycle so will stall there until they complete. This means the maximum IPC (Instructions per Cycle) Ibex can achieve is 1 when multi-cycle instructions aren’t used. See Multi- and Single-Cycle Instructions below for the details.

#### 3.1.1 Third Pipeline Stage

Ibex can be configured to have a third pipeline stage (Writeback) which has major effects on performance and instruction behaviour. This feature is *EXPERIMENTAL* and the details of its impact are not yet documented here. All of the information presented below applies only to the two stage pipeline provided in the default configurations.

#### 3.1.2 Multi- and Single-Cycle Instructions

In the table below when an instruction stalls for X cycles X + 1 cycles pass before a new instruction enters the ID/EX stage. Some instructions stall for a variable time, this is indicated as a range e.g. 1 - N means the instruction stalls a minimum of 1 cycle with an indeterminate maximum cycles. Read the description for more information.
<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Stall Cycles</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Computational</td>
<td>0</td>
<td>Integer Computational Instructions are defined in the RISCV-V RV32I Base Integer Instruction Set.</td>
</tr>
<tr>
<td>CSR Access</td>
<td>0</td>
<td>CSR Access Instruction are defined in “Zicsr” of the RISC-V specification.</td>
</tr>
<tr>
<td>Load/Store</td>
<td>Stor(N)</td>
<td>Both loads and stores stall for at least one cycle to await a response. For loads this response is the load data (which is written directly to the register file the same cycle it is received). For stores this is whether an error was seen or not. The longer the data side memory interface takes to receive a response the longer loads and stores will stall.</td>
</tr>
<tr>
<td>Multiplication</td>
<td>0/1 (Single-Cycle Multiplier) 2/3 (Fast Multi-Cycle Multiplier) clog2(op(_b))/32 (Slow Multi-Cycle Multiplier)</td>
<td>0 for MUL, 1 for MULH. 2 for MUL, 3 for MULH. clog2(op(_b)) for MUL, 32 for MULH. See details in Multiplier/Divider Block (MULT/DIV),</td>
</tr>
<tr>
<td>Division Remainder</td>
<td>1 or 37</td>
<td>1 stall cycle if divide by 0, otherwise full long division. See details in Multiplier/Divider Block (MULT/DIV)</td>
</tr>
<tr>
<td>Jump</td>
<td>1 - N</td>
<td>Minimum one cycle stall to flush the prefetch counter and begin fetching from the new Program Counter (PC). The new PC request will appear on the instruction-side memory interface the same cycle the jump instruction enters ID/EX. The longer the instruction-side memory interface takes to receive data the longer the jump will stall.</td>
</tr>
<tr>
<td>Branch</td>
<td>0</td>
<td>Any branch where the condition is not met will not stall.</td>
</tr>
<tr>
<td>Branch (Not-Taken)</td>
<td>1</td>
<td>Any branch where the condition is met will stall for 2 cycles as in the first cycle the branch is in ID/EX the ALU is used to calculate the branch condition. The following cycle the ALU is used again to calculate the branch target where it proceeds as Jump does above (Flush IF stage and prefetch buffer, new PC on instruction-side memory interface the same cycle it is calculated). The longer the instruction-side memory interface takes to receive data the longer the branch will stall. With the parameter BranchTargetALU set to 1 a separate ALU calculates the branch target simultaneously to calculating the branch condition with the main ALU so 1 less stall cycle is required.</td>
</tr>
<tr>
<td>Instruction Fence</td>
<td>1 - N</td>
<td>The FENCE.I instruction as defined in “Zifencei” of the RISC-V specification. Internally it is implemented as a jump (which does the required flushing) so it has the same stall characteristics (see above).</td>
</tr>
</tbody>
</table>
3.2 Instruction Cache

rtl/ibex_icache.sv.

NOTE - This module is currently DRAFT

The optional Instruction Cache (I$) is designed to improve CPU performance in systems with high instruction memory latency. The I$ integrates into the CPU by replacing the prefetch buffer, interfacing directly between the bus and IF stage.

3.2.1 High-level operation

The I$ passes instructions to the core using a ready / valid interface. Inside the cache is an address counter, which increments for every instruction fetched (by 2 or 4 bytes, depending on whether the instruction contents show it to be compressed). When the core takes a branch, it resets the counter to a new address by raising the `branch_i` signal and supplying the new address on `addr_i`. The next instruction returned by the cache will be the instruction at this new address.

The I$ communicates with instruction memory using an interface that matches the IF stage (allowing the cache to be enabled or disabled without needing to change the Ibex toplevel’s interface). For more details of this interface, see Instruction Fetch.

To avoid the cache fetching needlessly when the core is asleep (after a `wfi` instruction), it has a `req_i` input. Shortly after this goes low, the cache will stop making memory transactions.

If the `icache_enable_i` input is low, the cache operates in pass-through mode, where every requested instruction is fetched from memory and no results are cached.

In order to invalidate the cache, the core can raise the `icache_inval_i` line for one or more cycles, which will start an internal cache invalidation. No fetches are cached while the invalidation is taking place (behaving as if `icache_enable_i` is low).

While the I$ is busy, either (pre)fetched data or invalidating its memory, it raises the `busy_o` signal. This can be used to avoid the cache’s clock being gated when it is doing something.

3.2.2 Configuration options

The following table describes the available configuration parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusWidth</td>
<td>32</td>
<td>Width of instruction bus. Note, this is fixed at 32 for Ibex at the moment.</td>
</tr>
<tr>
<td>CacheSize</td>
<td>4kB</td>
<td>Size of cache in bytes.</td>
</tr>
<tr>
<td>CacheECC</td>
<td>1'b0</td>
<td>Enable SECDED ECC protection in tag and data RAMs.</td>
</tr>
<tr>
<td>LineSize</td>
<td>64</td>
<td>The width of one cache line in bits. Line sizes smaller than 64 bits may give compilation errors.</td>
</tr>
<tr>
<td>NumWays</td>
<td>2</td>
<td>The number of ways.</td>
</tr>
<tr>
<td>BranchCache</td>
<td>1'b0</td>
<td>When set, the cache will only allocate the targets of branches + two subsequent cache lines. This gives improved performance in systems with moderate latency by not polluting the cache with data that can be prefetched instead. When not set, all misses are allocated.</td>
</tr>
</tbody>
</table>
3.2.3 Performance notes

Note that although larger cache line sizes allow for better area efficiency (lower tagram area overhead), there is a performance penalty. When the core branches to an address that is not aligned to the bottom of a cache line (and the request misses in the cache), the IS will attempt to fetch this address first from the bus. The IS will then fetch the rest of the remaining beats of data in wrapping address order to complete the cache line (in order to allocate it to the cache). While these lower addresses are being fetched, the core is starved of data. Based on current experimental results, a line size of 64 bits gives the best performance.

In cases where the core branches to addresses currently being prefetched, the same line can end up allocated to the cache in multiple ways. This causes a minor performance inefficiency, but should not happen often in practice.

3.2.4 RAM Arrangement

The data RAMs are arranged as \texttt{NumWays} banks of \texttt{LineSize} width. If ECC is configured, the tag and data banks will be wider to accommodate the extra checkbits.

Indicative RAM sizes for common configurations are given in the table below:

<table>
<thead>
<tr>
<th>Cache config</th>
<th>Tag RAMs</th>
<th>Data RAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>4kB, 2 way, 64bit line</td>
<td>2 x 256 x 22bit</td>
<td>2 x 256 x 64bit</td>
</tr>
<tr>
<td>4kB, 2 way, 64bit line w/ECC</td>
<td>2 x 256 x 28bit</td>
<td>2 x 256 x 72bit</td>
</tr>
<tr>
<td>4kB, 2 way, 128bit line</td>
<td>2 x 128 x 22bit</td>
<td>2 x 128 x 128bit</td>
</tr>
<tr>
<td>4kB, 4 way, 64bit line</td>
<td>4 x 128 x 22bit</td>
<td>4 x 128 x 64bit</td>
</tr>
</tbody>
</table>

If \texttt{ICacheScramble} parameter is enabled, all RAM primitives are replaced with scrambling RAM primitive. For more information about how scrambling works internally (see \texttt{vendor/lowrisc_ip/ip/prim/doc/prim_ram_1p_scr.md}). Interface for receiving scrambling key follows \texttt{req / ack} protocol. Ibex first requests a new ephemeral key by asserting the request (\texttt{scramble_req_o}) and when a fresh valid key is indicated by \texttt{scramble_key_valid_i}, it deasserts the request. Note that in current implementation, it is assumed req/ack protocol is synchronized before arriving to Ibex top level.

3.2.5 Sub Unit Description

Figure 3.2: Instruction Cache Block Diagram

Prefetch Address

The prefetch address is updated to the branch target on every branch. This address is then updated in cache-line increments each time a cache lookup is issued to the cache pipeline.
Cache Pipeline

The cache pipeline consists of two stages, IC0 and IC1.

In IC0, lookup requests are arbitrated against cache allocation requests. Lookup requests have highest priority since they naturally throttle themselves as fill buffer resources run out. The arbitrated request is made to the RAMs in IC0.

In IC1, data from the RAMs are available and the cache hit status is determined. Hit data is multiplexed from the data RAMs based on the hitting way. If there was a cache miss, the victim way is chosen pseudo-randomly using a counter.

Fill buffers

The fill buffers perform several functions in the I$ and constitute most of its complexity.

- Since external requests can be made speculatively in parallel with the cache lookup, a fill buffer must be allocated in IC0 to track the request.
- The fill buffers are used as data storage for hitting requests as well as for miss tracking so all lookup requests require a fill buffer.
- A fill buffer makes multiple external requests to memory to fetch the required data to fill a cache line (tracked via fill_ext_cnt_q).
- Returning data is tracked via fill_rvd_cnt_q. Not all requests will fetch all their data, since requests can be cancelled due to a cache hit or an intervening branch.
- If a fill buffer has not made any external requests it will be cancelled by an intervening branch, if it has made requests then the requests will be completed and the line allocated.
- Beats of data are supplied to the IF stage, tracked via fill_out_cnt_q.
- If the line is due to be allocated into the cache, it will request for arbitration once all data has been received.
- Once all required actions are complete, the fill buffer releases and becomes available for a new request.

Since requests can perform actions out of order (cache hit in the shadow of an outstanding miss), and multiple requests can complete at the same time, the fill buffers are not a simple FIFO. Each fill buffer maintains a matrix of which requests are older than it, and this is used for arbitrating between the fill buffers.

Data output

Figure 3.3: Instruction Cache Data Multiplexing

Data supplied to the IF stage are multiplexed between cache-hit data, fill buffer data, and incoming memory data. The fill buffers track which request should supply data, and where that data should come from. Data from the cache and the fill buffers are of cache line width, which is multiplexed down to 32 bits and then multiplexed against data from the bus.

The fill buffers attempt to supply the relevant word of data to the IF stage as soon as possible. Hitting requests will supply the first word directly from the RAMs in IC1 while demand misses will supply data directly from the bus. The remaining data from hits is buffered in the fill buffer data storage and supplied to the IF stage as-required.

To deal with misalignment caused by compressed instructions, there is a 16bit skid buffer to store the upper halfword.
Cache ECC protection

When ECC protection is enabled, extra checkbits are appended to the top of the tag and data RAM write data as follows:

For the Tag RAMs (4kB cache):

<table>
<thead>
<tr>
<th>ECC checkbits</th>
<th>Valid bit</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>[27:22]</td>
<td>[21]</td>
<td>[20:0]</td>
</tr>
</tbody>
</table>

For the Data RAMs (64bit line):

<table>
<thead>
<tr>
<th>ECC checkbits</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>[71:64]</td>
<td>[63:0]</td>
</tr>
</tbody>
</table>

The checkbits are generated by dedicated modules in IC0 before the RAMs are written. In IC1, the RAM read data and checkbits are fed into dedicated modules which output whether there was an error. Although the modules used have the required outputs to allow inline correction of single bit errors, the IS does not make use of them since it never performs corrections.

Any error (single or double bit) in any RAM will effectively cancel a cache hit in IC1. The request which observed an error will fetch it’s data from the main instruction memory as normal for a cache miss. The cache index and way (or ways) with errors are stored in IC1, and a cache write is forced the next cycle to invalidate that line. Lookup requests will be blocked in IC0 while the invalidation write is performed. If an ECC error is seen a minor alert will be signaled.

Cache invalidation

After reset, and when requested by the core (due to a FENCE.I instruction), the whole cache is invalidated. Requests are inserted to invalidate the tag RAM for all ways in each cache line in sequence. While the invalidation is in-progress, lookups and instruction fetches can proceed, but nothing will be allocated to the cache.

Detailed behaviour

This section describes the expected behaviour of the cache, in order to allow functional verification. This isn’t an attempt to describe the cache’s performance characteristics.

The IS has a single clock (clk_i) and asynchronous reset (rst_ni).

Data is requested from the instruction memory with the ports prefixed by instr_. These work as described in Instruction Fetch. Note that there’s one extra port on the IS, which doesn’t appear at the ibex_top top-level. This is instr_pmp_err_i. If the PMP block disallows a fetch for a certain address, it will squash the outgoing memory request entirely and set instr_pmp_err_i. If that happens, the cache drops instr_req_o and stops making any further requests for that cache line. Note that it is possible for instr_gnt_i and instr_pmp_err_i to be high on the same cycle. In that case, the error signal takes precedence.

Fetched instructions are returned to the core using ports ready_i, valid_o, rdata_o, addr_o, err_o and err_plus2_o. This interface uses a form of ready/valid handshaking. A transaction is signalled by ready and valid being high. If valid goes high, it will remain high and the other output signals will remain stable until the transaction goes through or is cancelled by branch_i being asserted. The only exception is after an error is passed to the core. Once that has happened, there is no constraint on the values of valid_o, rdata_o, addr_o, err_o and err_plus2_o until the next time branch_i is asserted. There is no constraint on the behaviour of ready_i.

The 32-bit wide rdata_o signal contains instruction data fetched from addr_o. An instruction is either 16 or 32 bits wide (called compressed or uncompressed, respectively). The width of an instruction can be calculated from
Ibex Documentation, Release 0.1.dev76+g056cb44.d20220830

its bottom two bits: an instruction is uncompressed if they equal 2'b11 and compressed otherwise. If there is a compressed instruction in the lower 16 bits, the upper 16 bits are unconstrained (and may change even after valid has been asserted). The err_o signal will be high if the instruction fetch failed (either with instr_pmp_err_i or instr_err_i); in this case rdata_o is not specified.

The req_i signal tells the cache that the core is awake and will start requesting instructions soon. As well as the main cache memory, the I$ contains a prefetch buffer. The cache fills this buffer by issuing fetches when req_i is high. If req_i becomes false, the cache may do a few more instruction fetches to fill a cache line, but will stop fetching when that is done. The cache will not do any instruction fetches after this until req_i goes high again. A correctly behaving core should not not assert ready_i when req_i is low.

Inside the cache is an address counter. If branch_i is asserted then the address counter will be set to addr_i and the next instruction that is passed to the core will be the one fetched from that address. The address is required to be halfword aligned, so addr_i[0] must be zero. The cache will also start reading into a new prefetch buffer, storing the current contents into the main cache memory or discarding it (see icache_enable_i below). On cycles where branch_i is not asserted, the address counter will be incremented when an instruction is passed to the core. This increment depends on the instruction data (visible at rdata_o): it will be 2 if the instruction is compressed and 4 otherwise. Since the contents of rdata_o are not specified if an instruction fetch has caused an error, the core must signal a branch before accepting another instruction after it sees err_o.

There is an additional branch signal branch_spec_i which is a speculative version of the actual branch signal. Internally, branch_spec_i is used to setup address multiplexing as it is available earlier in the cycle. In cases where branch_spec_i is high, but branch_i is low, any lookup that might have been made that cycle is suppressed. Note that if branch_i is high, branch_spec_i must also be high.

Because a single instruction can span two 32bit memory addresses, an extra signal (err_plus2_o) indicates when an error is caused by the second half of an unaligned uncompressed instruction. This signal is only valid when valid_o and err_o are set, and will only be set for uncompressed instructions. The core uses this signal to record the correct address in the mtval CSR upon an error.

Since the address counter is not initialised on reset, the behaviour of the I$ is unspecified unless branch_i is asserted on or before the first cycle that req_i is asserted after reset. If that is not true, there’s nothing to stop the cache fetching from random addresses.

The icache_enable_i signal controls whether the cache copies fetched data from the prefetch buffer to the main cache memory. If the signal is false, fetched data will be discarded on a branch or after enough instructions have been consumed by the core. On reset, or whenever icache_inval_i goes high, the cache will invalidate its stored data. While doing this, the cache behaves as if icache_enable_i is false and will not store any fetched data.

Note: The rules for icache_enable_i and icache_inval_i mean that, in order to be completely sure of executing newly fetched code, the core should raise the icache_inval_i line for at least a cycle and then should branch. The Ibex core does this in response to a FENCE.I instruction, branching explicitly to the next PC.

The busy_o signal is guaranteed to be high while the cache is invalidating its internal memories or whenever it has a pending fetch on the instruction bus. When the busy_o signal is low, it is safe to clock gate the cache.

The cache doesn’t have circuitry to avoid inconsistent multi-way hits. As such, the core must never fetch from an address with the cache enabled after modifying the data at that address, without first starting a cache invalidation.

Note: This is a constraint on software, not just on the core.

3.2. Instruction Cache
3.3 Instruction Fetch

Figure 3.4: Instruction Fetch (IF) stage

The Instruction Fetch (IF) stage of the core is able to supply one instruction to the Instruction-Decode (ID) stage per cycle if the instruction cache or the instruction memory is able to serve one instruction per cycle.

Instructions are fetched into a prefetch buffer (`rtl/ibex_prefetch_buffer.sv`) for optimal performance and timing closure reasons. This buffer simply fetches instructions linearly until it is full. The instructions themselves are stored along with the Program Counter (PC) they came from in the fetch FIFO (`rtl/ibex_fetch_fifo.sv`). The fetch FIFO has a feedthrough path so when empty a new instruction entering the FIFO is immediately made available on the FIFO output. A localparam `DEPTH` gives a configurable depth which is set to 3 by default.

The top-level of the instruction fetch controls the prefetch buffer (in particular flushing it on branches/jumps/exception and beginning prefetching from the appropriate new PC) and supplies new instructions to the ID/EX stage along with their PC. Compressed instructions are expanded by the IF stage so the decoder can always deal with uncompressed instructions (the ID stage still receives the compressed instruction for placing into `mtval` on an illegal instruction exception).

If Ibex has been configured with an instruction cache (parameter `ICache == 1`), then the prefetch buffer is replaced by the icache module (`Instruction Cache`). The interfaces of the icache module are the same as the prefetch buffer with two additions. Firstly, a signal to enable the cache which is driven from a custom CSR. Secondly a signal to the flush the cache which is set every time a `fence.i` instruction is executed.

3.3.1 Branch Prediction

Ibex can be configured to use static branch prediction by setting the `BranchPrediction` parameter to 1. This improves performance by predicting that any branch with a negative offset is taken and that any branch with a positive offset is not. When successful, the prediction removes a stall cycle from a taken branch. However, there is a mis-predict penalty if a branch is wrongly predicted to be taken. This penalty is at least one cycle, or at least two cycles if the instruction following the branch is uncompressed and not aligned. This feature is `EXPERIMENTAL` and its effects are not yet fully documented.

3.3.2 Instruction-Side Memory Interface

The following table describes the signals that are used to fetch instructions. This interface is a simplified version of the interface used on the data interface as described in `Load-Store Unit`. The main difference is that the instruction interface does not allow for write transactions and thus needs less signals.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instr_req_o</td>
<td>output</td>
<td>Request valid, must stay high until <code>instr_gnt_i</code> is high for one cycle</td>
</tr>
<tr>
<td>instr_addr_o[31:0]</td>
<td>output</td>
<td>Address, word aligned</td>
</tr>
<tr>
<td>instr_gnt_i</td>
<td>input</td>
<td>The other side accepted the request. <code>instr_req_o</code> may be deasserted in the next cycle.</td>
</tr>
<tr>
<td>instr_rvalid_i</td>
<td>input</td>
<td><code>instr_rdata_i</code> holds valid data when <code>instr_rvalid_i</code> is high. This signal will be high for exactly one cycle per request.</td>
</tr>
<tr>
<td>instr_rdata_i[31:0]</td>
<td>input</td>
<td>Data read from memory</td>
</tr>
<tr>
<td>instr_rdata_intg_i</td>
<td>input</td>
<td>Data integrity bits from memory</td>
</tr>
<tr>
<td>instr_err_i</td>
<td>input</td>
<td>Memory access error</td>
</tr>
</tbody>
</table>
3.3.3 Misaligned Accesses

Externally, the IF interface performs word-aligned instruction fetches only. Misaligned instruction fetches are handled by performing two separate word-aligned instruction fetches. Internally, the core can deal with both word- and half-word-aligned instruction addresses to support compressed instructions. The LSB of the instruction address is ignored internally.

3.3.4 Protocol

The protocol used to communicate with the instruction cache or the instruction memory is very similar to the protocol used by the LSU on the data interface of Ibex. See the description of the LSU in LSU Protocol for details about this protocol.

3.4 Instruction Decode and Execute

The Instruction Decode and Execute stage takes instruction data from the instruction fetch stage (which has been converted to the uncompressed representation in the compressed instruction case). The instructions are decoded and executed all within one cycle including the register read and write. The stage is made up of multiple sub-blocks which are described below.

3.4.1 Instruction Decode Block (ID)

Source File: rtl/ibex_id_stage.sv

The Instruction Decode (ID) controls the overall decode/execution process. It contains the muxes to choose what is sent to the ALU inputs and where the write data for the register file comes from. A small state machine is used to control multi-cycle instructions (see Ibex Pipeline for more details), which stalls the whole stage whilst a multi-cycle instruction is executing.

3.4.2 Controller

Source File: rtl/ibex_controller.sv

The Controller contains the state machine that controls the overall execution of the processor. It is responsible for:

- Handling core startup from reset
- Setting the PC for the IF stage on jump/branch
- Dealing with exceptions/interrupts (jump to appropriate PC, set relevant CSR values)
- Controlling sleep/wakeup on WFI
- Debugging control
3.4.3 Decoder

Source File: rtl/ibex_decoder.sv

The decoder takes uncompressed instruction data and issues appropriate control signals to the other blocks to execute the instruction.

3.4.4 Register File


See Register File for more details.

3.4.5 Execute Block

Source File: rtl/ibex_ex_block.sv

The execute block contains the ALU and the multiplier/divider blocks, it does little beyond wiring and instantiating these blocks.

3.4.6 Arithmetic Logic Unit (ALU)

Source File: rtl/ibex_alu.sv

The Arithmetic Logic Unit (ALU) is a purely combinational block that implements operations required for the Integer Computational Instructions and the comparison operations required for the Control Transfer Instructions in the RV32I RISC-V Specification. Other blocks use the ALU for the following tasks:

- Mult/Div uses it to perform addition as part of the multiplication and division algorithms
- It computes branch targets with a PC + Imm calculation
- It computes memory addresses for loads and stores with a Reg + Imm calculation
- The LSU uses it to increment addresses when performing two accesses to handle an unaligned access

Bit-Manipulation Extension Support for the RISC-V Bit-Manipulation Extension version 1.0.0 and draft version 0.93 from January 10, 2021 is optional.1 It can be enabled via the enumerated parameter RV32B defined in rtl/ibex_pkg.sv. By default, this parameter is set to “ibex_pkg::RV32BNone” to disable the bit-manipulation extension.

There are three versions of the bit-manipulation extension available: The balanced version comprises a set of sub-extensions aiming for good benefits at a reasonable area overhead. It can be selected by setting the RV32B parameter to “ibex_pkg::RV32BBalanced”. The OTEarlGrey version comprises all sub-extensions except for the Zbe. This version can be selected by setting the RV32B parameter to “ibex_pkg::RV32BOTEarlGrey”. The full version comprises all sub-extensions and can be selected by setting the RV32B parameter to “ibex_pkg::RV32BFull”. The following table gives an overview of which sub-extensions are implemented in each version and of which instructions are implemented as multi-cycle instructions. Multi-cycle instructions are completed in 2 cycles. All remaining instructions complete in a single cycle.

---

1 Ibex fully implements the ratified version 1.0.0 of the RISC-V Bit-Manipulation Extension including the Zba, Zbb, Zbc and Zbs sub-extensions. In addition, Ibex also supports the remaining Zbe, Zbf, Zbp, Zbr and Zbt sub-extensions as defined in draft version 0.93 of the RISC-V Bit-Manipulation Extension. Note that the latter sub-extensions may change before being ratified as a standard by the RISC-V Foundation. Ibex will be updated to match future versions of the specification. Prior to ratification this may involve backwards incompatible changes. Additionally, neither GCC or Clang have committed to maintaining support upstream for unratified versions of the specification.
The implementation of the Bit-Manipulation Extension comes with an area overhead of 2.7 kGE for the balanced version, 6.1 kGE for the OTEarlGrey version, and 7.5 kGE for the full version. These numbers were obtained by synthesizing the design with Yosys and relaxed timing constraints.

3.4.7 Multiplier/Divider Block (MULT/DIV)

Source Files: rtl/ibex_multdiv_slow.sv rtl/ibex_multdiv_fast.sv

The Multiplier/Divider (MULT/DIV) is a state machine driven block to perform multiplication and division. The fast and slow versions differ in multiplier only. All versions implement the same form of long division algorithm. The ALU block is used by the long division algorithm in all versions.

**Multiplier** The multiplier can be implemented in three variants controlled via the enumerated parameter RV32M defined in rtl/ibex_pkg.sv.

**Single-Cycle Multiplier** This implementation is chosen by setting the RV32M parameter to “ibex_pkg::RV32MSingleCycle”. The single-cycle multiplier makes use of three parallel multiplier units, designed to be mapped to hardware multiplier primitives on FPGAs. It is therefore the first choice for FPGA synthesis.

- Using three parallel 17-bit x 17-bit multiplication units and a 34-bit accumulator, it completes a MUL instruction in 1 cycle. MULH is completed in 2 cycles.
- This MAC is internal to the mult/div block (no external ALU use).
- Beware it is simply implemented with the * and + operators so results heavily depend upon the synthesis tool used.
- ASIC synthesis has not yet been tested but is expected to consume 3-4x the area of the fast multiplier for ASIC.

**Fast Multi-Cycle Multiplier** This implementation is chosen by setting the RV32M parameter to “ibex_pkg::RV32MFast”. The fast multi-cycle multiplier provides a reasonable trade-off between area and performance. It is the first choice for ASIC synthesis.

- Completes multiply in 3-4 cycles using a MAC (multiply accumulate) which is capable of a 17-bit x 17-bit multiplication with a 34-bit accumulator.
- A MUL instruction takes 3 cycles, MULH takes 4.
- This MAC is internal to the mult/div block (no external ALU use).
- Beware it is simply implemented with the * and + operators so results heavily depend upon the synthesis tool used.
- In some cases it may be desirable to replace this with a specific implementation such as an explicit gate level implementation.
**Slow Multi-Cycle Multiplier** To select the slow multi-cycle multiplier, set the `RV32M` parameter to “ibex_pkg::RV32MSlow”.

- Completes multiply in $clog2(op_b) + 1$ cycles (for MUL) or 33 cycles (for MULH) using a Baugh-Wooley multiplier.
- The ALU block is used to compute additions.

**Divider** Both the fast and slow blocks use the same long division algorithm, it takes 37 cycles to compute (though only requires 2 cycles when there is a divide by 0) and proceeds as follows:

- Cycle 0: Check for divide by 0
- Cycle 1: Compute absolute value of operand A (or return result on divide by 0)
- Cycle 2: Compute absolute value of operand B

By setting the `RV32M` parameter to “ibex_pkg::RV32MNone”, the M-extension can be disabled completely.

### 3.4.8 Control and Status Register Block (CSR)

**Source File:** `rtl/ibex_cs_registers.sv`

The CSR contains all of the CSRs (control/status registers). Any CSR read/write is handled through this block. Performance counters are held in this block and incremented when appropriate (this includes `mcycle` and `minstret`). Read data from a CSR is available the same cycle it is requested. Further detail on the implemented CSRs can be found in [Control and Status Registers](#).

### 3.4.9 Load-Store Unit (LSU)

**Source File:** `rtl/ibex_load_store_unit.sv`

The Load-Store Unit (LSU) interfaces with main memory to perform load and store operations. See [Load-Store Unit](#) for more details.

### 3.5 Load-Store Unit

**Source File:** `rtl/ibex_load_store_unit.sv`

The Load-Store Unit (LSU) of the core takes care of accessing the data memory. Loads and stores of words (32 bit), half words (16 bit) and bytes (8 bit) are supported.

Any load or store will stall the ID/EX stage for at least a cycle to await the response (whether that is awaiting load data or a response indicating whether an error has been seen for a store).
3.5.1 Data-Side Memory Interface

Signals that are used by the LSU:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_req_o</td>
<td>output</td>
<td>Request valid, must stay high until data_gnt_i is high for one cycle</td>
</tr>
<tr>
<td>data_addr_o[31:0]</td>
<td>output</td>
<td>Address, word aligned</td>
</tr>
<tr>
<td>data_we_o</td>
<td>output</td>
<td>Write Enable, high for writes, low for reads. Sent together with data_req_o</td>
</tr>
<tr>
<td>data_be_o[3:0]</td>
<td>output</td>
<td>Byte Enable. Is set for the bytes to write/read, sent together with data_req_o</td>
</tr>
<tr>
<td>data_wdata_o[31:0]</td>
<td>output</td>
<td>Data to be written to memory, sent together with data_req_o</td>
</tr>
<tr>
<td>data_wdata_intg_o[6:0]</td>
<td>output</td>
<td>Integrity bits to be written to memory, sent together with data_req_o (not used unless the SecureIbex parameter is set)</td>
</tr>
<tr>
<td>data_gnt_i</td>
<td>input</td>
<td>The other side accepted the request. Outputs may change in the next cycle.</td>
</tr>
<tr>
<td>data_rvalid_i</td>
<td>input</td>
<td>data_err_i and data_rdata_i hold valid data when data_rvalid_i is high. This signal will be high for exactly one cycle per request.</td>
</tr>
<tr>
<td>data_err_i</td>
<td>input</td>
<td>Error response from the bus or the memory: request cannot be handled. High in case of an error.</td>
</tr>
<tr>
<td>data_rdata_i[31:0]</td>
<td>input</td>
<td>Data read from memory</td>
</tr>
<tr>
<td>data_rdata_intg_i[6:0]</td>
<td>input</td>
<td>Integrity bits read from memory (ignored unless the SecureIbex parameter is set)</td>
</tr>
</tbody>
</table>

3.5.2 Bus Integrity Checking

The core can optionally generate and verify check bits sent alongside the data for memory accesses. Checkbits are generated and checked using an inverted 39/32 Hsai code (see vendor/lowrisc_ip/ip/prim/rtl/prim_seceded_inv_39_32_enc.sv). An internal interrupt will be generated and a bus major alert signalled if there is a mismatch. Where load data has bad checkbits the write to the load’s destination register will be suppressed.

This feature is only used if the core is configured with the SecureIbex parameter set. For all other configurations, the integrity signals can be ignored.

3.5.3 Misaligned Accesses

The LSU is able to handle misaligned memory accesses, meaning accesses that are not aligned on natural word boundaries. However, it does so by performing two separate word-aligned accesses. This means that at least two cycles are needed for misaligned loads and stores.

If an error response is received for the first transaction, the second transaction will still be issued. The second transaction will then follow the normal bus protocol, but its response/data will be ignored. If a new load/store request is received while waiting for an abandoned second part to complete, it will not be serviced until the state machine returns to IDLE.
3.5.4 Protocol

The protocol that is used by the LSU to communicate with a memory works as follows:

1. The LSU provides a valid address in `data_addr_o` and sets `data_req_o` high. In the case of a store, the LSU also sets `data_we_o` high and configures `data_be_o` and `data_wdata_o`. The memory then answers with a `data_gnt_i` set high as soon as it is ready to serve the request. This may happen in the same cycle as the request was sent or any number of cycles later.

2. After receiving a grant, the address may be changed in the next cycle by the LSU. In addition, the `data_wdata_o`, `data_we_o` and `data_be_o` signals may be changed as it is assumed that the memory has already processed and stored that information.

3. The memory answers with a `data_rvalid_i` set high for exactly one cycle to signal the response from the bus or the memory using `data_err_i` and `data_rdata_i` (during the very same cycle). This may happen one or more cycles after the grant has been received. If `data_err_i` is low, the request could successfully be handled at the destination and in the case of a load, `data_rdata_i` contains valid data. If `data_err_i` is high, an error occurred in the memory system and the core will raise an exception.

4. When multiple granted requests are outstanding, it is assumed that the memory requests will be kept in-order and one `data_rvalid_i` will be signalled for each of them, in the order they were issued.

Figure 3.6, Figure 3.7 and Figure 3.8 show example-timing diagrams of the protocol.

3.6 Register File


Ibex has either 31 or 15 32-bit registers if the RV32E extension is disabled or enabled, respectively. Register x0 is statically bound to 0 and can only be read, it does not contain any sequential logic.

The register file has two read ports and one write port, register file data is available the same cycle a read is requested. There is no write to read forwarding path so if one register is being both read and written the read will return the current value rather than the value being written.
3.6. Register File

Figure 3.7: Back-to-back Memory Transaction

Figure 3.8: Slow Response Memory Transaction
There are three flavors of register file available, each having their own benefits and trade-offs. The register file flavor is selected via the enumerated parameter `RegFile` defined in `rtl/ibex_pkg.sv`.

### 3.6.1 Flip-Flop-Based Register File

The flip-flop-based register file uses regular, positive-edge-triggered flip-flops to implement the registers. This makes it the first choice when simulating the design using Verilator.

This implementation can be selected by setting the `RegFile` parameter to "ibex_pkg::RegFileFF". It is the default selection.

### 3.6.2 FPGA Register File

The FPGA register file leverages synchronous-write / asynchronous-read RAM design elements, where available on FPGA targets.

For Xilinx FPGAs, synthesis results in an implementation using RAM32M primitives. Using this design with a Xilinx Artya7-100 FPGA conserves around 600 Logic LUTs and 1000 flip-flops at the expense of 48 LUTRAMs for the 31-entry register file as compared to the flip-flop-based register file.

This makes it the first choice for FPGA synthesis.

To select the FPGA register file, set the `RegFile` parameter to "ibex_pkg::RegFileFPGA".

### 3.6.3 Latch-Based Register File

The latch-based register file uses level-sensitive latches to implement the registers.

This allows for significant area savings compared to an implementation using regular flip-flops and thus makes the latch-based register file the first choice for ASIC implementations. Simulation of the latch-based register file is possible using commercial tools.

**Note:** The latch-based register file cannot be simulated using Verilator.

The latch-based register file can also be used for FPGA synthesis, but this is not recommended as FPGAs usually do not well support latches.

To select the latch-based register file, set the `RegFile` parameter to "ibex_pkg::RegFileLatch". In addition, a technology-specific clock gating cell must be provided to keep the clock inactive when the latches are not written. This cell must be wrapped in a module called `prim_clock_gating`. For more information regarding the clock gating cell, checkout *Getting Started with Ibex*.

**Note:** The latch-based register file requires the gated clock to be enabled in the cycle after the write enable `we_a_i` signal was set high. This can be achieved by latching `we_a_i` in the clock gating cell during the low phase of `clk_i`.

The resulting behavior of the latch-based register file is visualized in Figure 3.9. The input data `wdata_a_i` is sampled into a flip-flop-based register `wdata_a_q` using `clk_int`. The actual latch-based registers `mem[1]` and `mem[2]` are transparent during high phases of `mem_clk[1]` and `mem_clk[2]`, respectively. Their content is sampled from `wdata_a_q` on falling edges of these clocks.
### 3.7 Control and Status Registers

Ibex implements all the Control and Status Registers (CSRs) listed in the following table according to the RISC-V Privileged Specification, version 1.11.

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
<td>mstatus</td>
<td>WARL</td>
<td>Machine Status</td>
</tr>
<tr>
<td>0x0301</td>
<td>misa</td>
<td>WARL</td>
<td>Machine ISA and Extensions</td>
</tr>
<tr>
<td>0x0304</td>
<td>mie</td>
<td>WARL</td>
<td>Machine Interrupt Enable Register</td>
</tr>
<tr>
<td>0x0305</td>
<td>mtvec</td>
<td>WARL</td>
<td>Machine Trap-Vector Base Address</td>
</tr>
<tr>
<td>0x0320</td>
<td>mcountinhibit</td>
<td>RW</td>
<td>Machine Counter-Inhibit Register</td>
</tr>
<tr>
<td>0x0323</td>
<td>mhpmevent3</td>
<td>WARL</td>
<td>Machine Performance-Monitoring Event Selector</td>
</tr>
<tr>
<td>0x033F</td>
<td>mhpmevent31</td>
<td>WARL</td>
<td>Machine Performance-Monitoring Event Selector</td>
</tr>
<tr>
<td>0x0340</td>
<td>mscratch</td>
<td>RW</td>
<td>Machine Scratch Register</td>
</tr>
<tr>
<td>0x0341</td>
<td>mepc</td>
<td>WARL</td>
<td>Machine Exception Program Counter</td>
</tr>
<tr>
<td>0x0342</td>
<td>mcause</td>
<td>WLRL</td>
<td>Machine Cause Register</td>
</tr>
<tr>
<td>0x0343</td>
<td>mtval</td>
<td>WARL</td>
<td>Machine Trap Value Register</td>
</tr>
<tr>
<td>0x0344</td>
<td>mip</td>
<td>R</td>
<td>Machine Interrupt Pending Register</td>
</tr>
<tr>
<td>0x03A0</td>
<td>pmpcfg0</td>
<td>WARL</td>
<td>PMP Configuration Register</td>
</tr>
<tr>
<td>0x03A3</td>
<td>pmpcfg3</td>
<td>WARL</td>
<td>PMP Configuration Register</td>
</tr>
<tr>
<td>0x03B0</td>
<td>pmpadr0</td>
<td>WARL</td>
<td>PMP Address Register</td>
</tr>
<tr>
<td>0x03BF</td>
<td>pmpadr15</td>
<td>WARL</td>
<td>PMP Address Register</td>
</tr>
<tr>
<td>0x0747</td>
<td>mseccfg</td>
<td>WARL</td>
<td>Machine Security Configuration</td>
</tr>
<tr>
<td>0x0757</td>
<td>mseccfg0</td>
<td>WARL</td>
<td>Upper 32 bits of mseccfg</td>
</tr>
<tr>
<td>0x07A0</td>
<td>tselect</td>
<td>WARL</td>
<td>Trigger Select Register</td>
</tr>
<tr>
<td>0x07A1</td>
<td>tdata1</td>
<td>WARL</td>
<td>Trigger Data Register 1</td>
</tr>
<tr>
<td>0x07A2</td>
<td>tdata2</td>
<td>WARL</td>
<td>Trigger Data Register 2</td>
</tr>
</tbody>
</table>

Figure 3.9: Latch-based register file operation

3.7. Control and Status Registers  27
Table 3.1 – continued from previous page

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7A3</td>
<td>tdata3</td>
<td>W ARL</td>
<td>Trigger Data Register 3</td>
</tr>
<tr>
<td>0x7A8</td>
<td>mcontext</td>
<td>W ARL</td>
<td>Machine Context Register</td>
</tr>
<tr>
<td>0x7AA</td>
<td>scontext</td>
<td>W ARL</td>
<td>Supervisor Context Register</td>
</tr>
<tr>
<td>0x7B0</td>
<td>dcsr</td>
<td>WARL</td>
<td>Debug Control and Status Register</td>
</tr>
<tr>
<td>0x7B1</td>
<td>dpc</td>
<td>RW</td>
<td>Debug PC</td>
</tr>
<tr>
<td>0x7B2</td>
<td>dscratch0</td>
<td>RW</td>
<td>Debug Scratch Register 0</td>
</tr>
<tr>
<td>0x7B3</td>
<td>dscratch1</td>
<td>RW</td>
<td>Debug Scratch Register 1</td>
</tr>
<tr>
<td>0x7C0</td>
<td>cpuctrl</td>
<td>WARL</td>
<td>CPU Control Register (Custom CSR)</td>
</tr>
<tr>
<td>0x7C1</td>
<td>secureseed</td>
<td>WARL</td>
<td>Security feature random seed (Custom CSR)</td>
</tr>
<tr>
<td>0xB00</td>
<td>mcycle</td>
<td>RW</td>
<td>Machine Cycle Counter</td>
</tr>
<tr>
<td>0xB02</td>
<td>minstret</td>
<td>RW</td>
<td>Machine Instructions-Retired Counter</td>
</tr>
<tr>
<td>0xB03</td>
<td>mhpmcounter3</td>
<td>WARL</td>
<td>Machine Performance-Monitoring Counter</td>
</tr>
</tbody>
</table>

....

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xB1F</td>
<td>mhpmcounter31</td>
<td>WARL</td>
<td>Machine Performance-Monitoring Counter</td>
</tr>
<tr>
<td>0xB80</td>
<td>mcycleh</td>
<td>RW</td>
<td>Upper 32 bits of mcycle</td>
</tr>
<tr>
<td>0xB82</td>
<td>minstreth</td>
<td>RW</td>
<td>Upper 32 bits of minstret</td>
</tr>
<tr>
<td>0xB83</td>
<td>mhpmcounter3h</td>
<td>WARL</td>
<td>Upper 32 bits of mhpmcounter3</td>
</tr>
</tbody>
</table>

....

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xB9F</td>
<td>mhpmcounter31h</td>
<td>WARL</td>
<td>Upper 32 bits of mhpmcounter31</td>
</tr>
<tr>
<td>0xF11</td>
<td>mvendorid</td>
<td>R</td>
<td>Machine Vendor ID</td>
</tr>
<tr>
<td>0xF12</td>
<td>marchid</td>
<td>R</td>
<td>Machine Architecture ID</td>
</tr>
<tr>
<td>0xF13</td>
<td>mimpid</td>
<td>R</td>
<td>Machine Implementation ID</td>
</tr>
<tr>
<td>0xF14</td>
<td>mhartid</td>
<td>R</td>
<td>Hardware Thread ID</td>
</tr>
</tbody>
</table>

See the Performance Counters documentation for a description of the counter registers.

### 3.7.1 Machine Status (mstatus)

CSR Address: 0x300

Reset Value: 0x0000_1800

<table>
<thead>
<tr>
<th>Bit#</th>
<th>R/W</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>RW</td>
<td>TW: Timeout Wait (WFI executed in User Mode will trap to Machine Mode).</td>
</tr>
<tr>
<td>17</td>
<td>RW</td>
<td>MPRV: Modify Privilege (Loads and stores use MPP for privilege checking).</td>
</tr>
<tr>
<td>12:11</td>
<td>RW</td>
<td>MPP: Machine Previous Privilege mode.</td>
</tr>
<tr>
<td>7</td>
<td>RW</td>
<td>Previous Interrupt Enable (MPIE), i.e., before entering exception handling.</td>
</tr>
<tr>
<td>3</td>
<td>RW</td>
<td>Interrupt Enable (MIE): If set to 1'b1, interrupts are globally enabled.</td>
</tr>
</tbody>
</table>

When an exception is encountered, mstatus.MPIE will be set to mstatus.MIE, and mstatus.MPP will be set to the current privilege mode. When the MRET instruction is executed, the value of MPIE will be stored back to mstatus.MIE, and the privilege mode will be restored from mstatus.MPP.

If you want to enable interrupt handling in your exception handler, set mstatus.MIE to 1'b1 inside your handler code.

Only Machine Mode and User Mode are supported. Any write to mstatus.MPP of an unsupported value will be interpreted as Machine Mode.
3.7.2 Machine ISA Register (misa)

CSR Address: 0x301

misa is a WARL register which describes the ISA supported by the hart. On Ibex, misa is hard-wired, i.e. it will remain unchanged after any write.

3.7.3 Machine Interrupt Enable Register (mie)

CSR Address: 0x304
Reset Value: 0x0000_0000

mie is a WARL register which allows to individually enable/disable local interrupts. After reset, all interrupts are disabled.

<table>
<thead>
<tr>
<th>Bit#</th>
<th>Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>30:16</td>
<td>Machine Fast Interrupt Enables: Set bit x+16 to enable fast interrupt irq_fast_i[x].</td>
</tr>
<tr>
<td>11</td>
<td>Machine External Interrupt Enable (MEIE): If set, irq_external_i is enabled.</td>
</tr>
<tr>
<td>7</td>
<td>Machine Timer Interrupt Enable (MTIE): If set, irq_timer_i is enabled.</td>
</tr>
<tr>
<td>3</td>
<td>Machine Software Interrupt Enable (MSIE): if set, irq_software_i is enabled.</td>
</tr>
</tbody>
</table>

3.7.4 Machine Trap-Vector Base Address (mtvec)

CSR Address: 0x305
Reset Value: 0x0000_0001

mtvec is a WARL register which contains the machine trap-vector base address.

<table>
<thead>
<tr>
<th>Bit#</th>
<th>Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:2</td>
<td>BASE: The trap-vector base address, always aligned to 256 bytes, i.e., mtvec[7:2] is always set to 6'b0.</td>
</tr>
<tr>
<td>1:0</td>
<td>MODE: Always set to 2'b01 to indicate vectored interrupt handling (read-only).</td>
</tr>
</tbody>
</table>

3.7.5 Machine Exception PC (mepc)

CSR Address: 0x341
Reset Value: 0x0000_0000

When an exception is encountered, the current program counter is saved in mepc, and the core jumps to the exception address. When an MRET instruction is executed, the value from mepc replaces the current program counter.

3.7.6 Machine Cause (mcause)

CSR Address: 0x342
Reset Value: 0x0000_0000

<table>
<thead>
<tr>
<th>Bit#</th>
<th>R/W</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>R</td>
<td>Interrupt: This bit is set when the exception was triggered by an interrupt.</td>
</tr>
<tr>
<td>4:0</td>
<td>R</td>
<td>Exception Code</td>
</tr>
</tbody>
</table>
When an exception is encountered, the corresponding exception code is stored in this register.

### 3.7.7 Machine Trap Value (mtval)

**CSR Address:** 0x343  
**Reset Value:** 0x0000_0000

When an exception is encountered, this register can hold exception-specific information to assist software in handling the trap.

- In the case of errors in the load-store unit, mtval holds the address of the transaction causing the error.
- If this transaction is misaligned, mtval holds the address of the missing transaction part.
- In the case of illegal instruction exceptions, mtval holds the actual faulting instruction.

For all other exceptions, mtval is 0.

### 3.7.8 Machine Interrupt Pending Register (mip)

**CSR Address:** 0x344  
**Reset Value:** 0x0000_0000

mip is a read-only register indicating pending interrupt requests. A particular bit in the register reads as one if the corresponding interrupt input signal is high and if the interrupt is enabled in the mie CSR.

<table>
<thead>
<tr>
<th>Bit#</th>
<th>Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>30:16</td>
<td>Machine Fast Interrupts Pending: If bit x+16 is set, fast interrupt irq_fast_i[x] is pending.</td>
</tr>
<tr>
<td>11</td>
<td>Machine External Interrupt Pending (MEIP): If set, irq_external_i is pending.</td>
</tr>
<tr>
<td>7</td>
<td>Machine Timer Interrupt Pending (MTIP): If set, irq_timer_i is pending.</td>
</tr>
<tr>
<td>3</td>
<td>Machine Software Interrupt Pending (MSIP): If set, irq_software_i is pending.</td>
</tr>
</tbody>
</table>

### 3.7.9 PMP Configuration Register (pmpcfgx)

**CSR Address:** 0x3A0 - 0x3A3  
**Reset Value:** 0x0000_0000

pmpcfgx are registers to configure PMP regions. Each register configures 4 PMP regions.

<table>
<thead>
<tr>
<th>31:24</th>
<th>23:16</th>
<th>15:8</th>
<th>7:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>pmp3cfg</td>
<td>pmp2cfg</td>
<td>pmp1cfg</td>
<td>pmp0cfg</td>
</tr>
</tbody>
</table>

The configuration fields for each region are as follows:

<table>
<thead>
<tr>
<th>Bit#</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Lock</td>
</tr>
<tr>
<td>6:5</td>
<td>Reserved (Read as zero)</td>
</tr>
<tr>
<td>4:3</td>
<td>Mode</td>
</tr>
<tr>
<td>2</td>
<td>Execute permission</td>
</tr>
<tr>
<td>1</td>
<td>Write permission</td>
</tr>
<tr>
<td>0</td>
<td>Read permission</td>
</tr>
</tbody>
</table>
Details of these configuration bits can be found in the RISC-V Privileged Specification, version 1.11 (see Physical Memory Protection CSRs, Section 3.6.1).

Note that the combination of Write permission = 1, Read permission = 0 is reserved, and will be treated by the core as Read/Write permission = 0.

### 3.7.10 PMP Address Register (pmpaddrx)

**CSR Address:** 0x3B0 - 0x3BF  
**Reset Value:** 0x0000_0000  

pmpaddrx are registers to set address matching for PMP regions.

![Address Table](image)

### 3.7.11 Machine Security Configuration (mseccfg/mseccfgh)

**CSR Address:** mseccfg: 0x747 mseccfg: 0x757  
**Reset Value:** 0x0000_0000_0000_0000  

<table>
<thead>
<tr>
<th>Bit#</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td><strong>Rule Locking Bypass (RLB):</strong> If set locked PMP entries can be modified</td>
</tr>
<tr>
<td>1</td>
<td><strong>Machine Mode Whitelist Policy (MMWP):</strong> If set default policy for PMP is deny for M-Mode accesses that don’t match a PMP region</td>
</tr>
<tr>
<td>0</td>
<td><strong>Machine Mode Lockdown (MML):</strong> Alters behaviour of pmpcfgX bits</td>
</tr>
</tbody>
</table>

mseccfg is specified in the Trusted Execution Environment (TEE) working group proposal PMP Enhancements for memory access and execution prevention on Machine mode (Smepmp) version 0.9.3, which gives the full details of it’s functionality including the new PMP behaviour when mseccfg.MML is set. Note that the reset value means PMP behavior out of reset matches the RISC-V Privileged Architecture. A write to mseccfg is required to change it. Note mseccfgh reads as all 0s and ignores all writes. Any access to mseccfg or mseccfgh when using an Ibex configuration without PMP (PMPEnable is 0) will trigger an illegal instruction exception.

### 3.7.12 Trigger Select Register (tselect)

**CSR Address:** 0x7A0  
**Reset Value:** 0x0000_0000  

Accessible in Debug Mode or M-Mode when trigger support is enabled (using the DbgTriggerEn parameter).

Number of the currently selected trigger starting at 0. The number of triggers is configured by the DbgHwNumLen parameter.

Writing a value larger than or equal to the number of supported triggers will write the highest valid index. This allows a debugger to detect the allowed number of triggers by reading back the value.
3.7.13 Trigger Data Register 1 (tdata1)

CSR Address: 0x7A1
Reset Value: 0x2800_1000

Accessible in Debug Mode or M-Mode when trigger support is enabled (using the DbgTriggerEn parameter). Since native triggers are not supported, writes to this register from M-Mode will be ignored.

Ibex only implements one type of trigger, instruction address match. Most fields of this register will read as a fixed value to reflect the mode that is supported.

<table>
<thead>
<tr>
<th>Bit#</th>
<th>R/W</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:28</td>
<td>R</td>
<td>type: 2 = Address/Data match trigger type.</td>
</tr>
<tr>
<td>27</td>
<td>R</td>
<td>dmode: 1 = Only debug mode can write tdata registers</td>
</tr>
<tr>
<td>26:21</td>
<td>R</td>
<td>maskmax: 0 = Only exact matching supported.</td>
</tr>
<tr>
<td>20</td>
<td>R</td>
<td>hit: 0 = Hit indication not supported.</td>
</tr>
<tr>
<td>19</td>
<td>R</td>
<td>select: 0 = Only address matching is supported.</td>
</tr>
<tr>
<td>18</td>
<td>R</td>
<td>timing: 0 = Break before the instruction at the specified address.</td>
</tr>
<tr>
<td>17:16</td>
<td>R</td>
<td>sizelo: 0 = Match accesses of any size.</td>
</tr>
<tr>
<td>15:12</td>
<td>R</td>
<td>action: 1 = Enter debug mode on match.</td>
</tr>
<tr>
<td>11</td>
<td>R</td>
<td>chain: 0 = Chaining not supported.</td>
</tr>
<tr>
<td>10:7</td>
<td>R</td>
<td>match: 0 = Match the whole address.</td>
</tr>
<tr>
<td>6</td>
<td>R</td>
<td>m: 1 = Match in M-Mode.</td>
</tr>
<tr>
<td>5</td>
<td>R</td>
<td>s: 0 = S-Mode not supported.</td>
</tr>
<tr>
<td>4</td>
<td>R</td>
<td>u: 1 = Match in U-Mode.</td>
</tr>
<tr>
<td>3</td>
<td>R</td>
<td>execute: Enable matching on instruction address.</td>
</tr>
<tr>
<td>2</td>
<td>RW</td>
<td>load: 0 = Load address / data matching not supported.</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>store: 0 = Store address / data matching not supported.</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
<td>load: 0 = Load address / data matching not supported.</td>
</tr>
</tbody>
</table>

Details of these configuration bits can be found in the RISC-V Debug Specification, version 0.13.2 (see Trigger Registers, Section 5.2).

3.7.14 Trigger Data Register 2 (tdata2)

CSR Address: 0x7A2
Reset Value: 0x0000_0000

Accessible in Debug Mode or M-Mode when trigger support is enabled (using the DbgTriggerEn parameter). Since native triggers are not supported, writes to this register from M-Mode will be ignored.

This register stores the instruction address to match against for a breakpoint trigger.
3.7.15 Trigger Data Register 3 (tdata3)

CSR Address: 0x7A3
Reset Value: 0x0000_0000

Accessible in Debug Mode or M-Mode when trigger support is enabled (using the DbgTriggerEn parameter). Ibex does not support the features requiring this register, so writes are ignored and it will always read as zero.

3.7.16 Machine Context Register (mcontext)

CSR Address: 0x7A8
Reset Value: 0x0000_0000

Accessible in Debug Mode or M-Mode when trigger support is enabled (using the DbgTriggerEn parameter). Ibex does not support the features requiring this register, so writes are ignored and it will always read as zero.

3.7.17 Supervisor Context Register (scontext)

CSR Address: 0x7AA
Reset Value: 0x0000_0000

Accessible in Debug Mode or M-Mode when trigger support is enabled (using the DbgTriggerEn parameter). Ibex does not support the features requiring this register, so writes are ignored and it will always read as zero.

3.7.18 Debug Control and Status Register (dcsr)

CSR Address: 0x7B0
Reset Value: 0x4000_0003

Accessible in Debug Mode only. Ibex implements the following bit fields. Other bit fields read as zero.

<table>
<thead>
<tr>
<th>Bit#</th>
<th>R/W</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:28</td>
<td>R</td>
<td><strong>xdebugver:</strong> 4 = External spec-compliant debug support exists.</td>
</tr>
<tr>
<td>15</td>
<td>RW</td>
<td><strong>ebreakm:</strong> EBREAK in M-Mode behaves as described in Privileged Spec (0), or enters Debug Mode (1).</td>
</tr>
<tr>
<td>12</td>
<td>WAR</td>
<td><strong>ebreaku:</strong> EBREAK in U-Mode behaves as described in Privileged Spec (0), or enters Debug Mode (1).</td>
</tr>
<tr>
<td>8:6</td>
<td>R</td>
<td><strong>cause:</strong> 1 = EBREAK, 2 = trigger, 3 = halt request, 4 = step</td>
</tr>
<tr>
<td>2</td>
<td>RW</td>
<td><strong>step:</strong> When set and not in Debug Mode, execute a single instruction and enter Debug Mode.</td>
</tr>
<tr>
<td>1:0</td>
<td>WAR</td>
<td><strong>prv:</strong> Privilege level the core was operating in when Debug Mode was entered. May be modified by debugger to change privilege level. Ibex allows transitions to all supported modes. (M- and U-Mode).</td>
</tr>
</tbody>
</table>

Details of these configuration bits can be found in the RISC-V Debug Specification, version 0.13.2 (see Core Debug Registers, Section 4.8). Note that **ebreaku** and **prv** are accidentally specified as RW in version 0.13.2 of the RISC-V Debug Specification. More recent versions of the specification define these fields correctly as WARL.
### 3.7.19 Debug PC Register (dpc)

CSR Address: 0x7B1  
Reset Value: 0x0000_0000

When entering Debug Mode, dpc is updated with the address of the next instruction that would be executed (if Debug Mode would not have been entered). When resuming, the PC is set to the address stored in dpc. The debug module may modify dpc. Accessible in Debug Mode only.

### 3.7.20 Debug Scratch Register 0 (dscratch0)

CSR Address: 0x7B2  
Reset Value: 0x0000_0000

Scratch register to be used by the debug module. Accessible in Debug Mode only.

### 3.7.21 Debug Scratch Register 1 (dscratch1)

CSR Address: 0x7B3  
Reset Value: 0x0000_0000

Scratch register to be used by the debug module. Accessible in Debug Mode only.

### 3.7.22 CPU Control Register (cpuctrl)

CSR Address: 0x7C0  
Reset Value: 0x0000_0000

Custom CSR to control runtime configuration of CPU components. Accessible in Machine Mode only. Ibex implements the following bit fields. Other bit fields read as zero.

<table>
<thead>
<tr>
<th>Bit#</th>
<th>R/W</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>RW</td>
<td><strong>double_fault_seen:</strong> A synchronous exception was observed when the sync_exc_seen field was set. This field must be manually cleared, hardware only sets it (see Double Fault Detection).</td>
</tr>
<tr>
<td>6</td>
<td>RW</td>
<td><strong>sync_exc_seen:</strong> A synchronous exception has been observed. This flag is cleared when mret is executed. (see Double Fault Detection).</td>
</tr>
<tr>
<td>5:3</td>
<td>W ARL</td>
<td><strong>dummy_instr_mask:</strong> Mask to control frequency of dummy instruction insertion. If the core has not been configured with security features (SecureIbex parameter == 0), this field will always read as zero (see Security Features).</td>
</tr>
<tr>
<td>2</td>
<td>W ARL</td>
<td><strong>dummy_instr_en:</strong> Enable (1) or disable (0) dummy instruction insertion features. If the core has not been configured with security features (SecureIbex parameter == 0), this field will always read as zero (see Security Features).</td>
</tr>
<tr>
<td>1</td>
<td>W ARL</td>
<td><strong>data_ind_timing:</strong> Enable (1) or disable (0) data-independent timing features. If the core has not been configured with security features (SecureIbex parameter == 0), this field will always read as zero.</td>
</tr>
<tr>
<td>0</td>
<td>W ARL</td>
<td><strong>icache_enable:</strong> Enable (1) or disable (0) the instruction cache. If the instruction cache has not been configured (ICache parameter == 0), this field will always read as zero.</td>
</tr>
</tbody>
</table>
3.7.23 Security Feature Seed Register (secureseed)

CSR Address: 0x7C1
Reset Value: 0x0000_0000
Accessible in Machine Mode only.

Custom CSR to allow re-seeding of security-related pseudo-random number generators. A write to this register will update the seeding of pseudo-random number generators inside the design. This allows software to improve the randomness, and therefore security, of certain features by periodically reading from a true random number generator peripheral. Seed values are not actually stored in a register and so reads to this register will always return zero.

3.7.24 Time Registers (time(h))

CSR Address: 0xC01 / 0xC81

The User Mode time(h) registers are not implemented in Ibex. Any access to these registers will trap. It is recommended that trap handler software provides a means of accessing platform-defined mtime(h) timers where available.

3.7.25 Machine Vendor ID (mvendorid)

CSR Address: 0xF11
Reset Value: 0x0000_0000

Use the CSR_MVENDORID_VALUE parameter in rtl/ibex_pkg.sv to change the fixed value. Details of what the ID represents can be found in the RISC-V Privileged Specification.

3.7.26 Machine Architecture ID (marchid)

CSR Address: 0xF12
Reset Value: 0x0000_0016

Use the CSR_MARCHID_VALUE parameter in rtl/ibex_pkg.sv to change the fixed value. The value used is allocated specifically to Ibex. If significant changes are made a different ID should be used. Details of what the ID represents can be found in the RISC-V Privileged Specification.

3.7.27 Machine Implementation ID (mimpid)

CSR Address: 0xF13
Reset Value: 0x0000_0000

Use the CSR_MIMPID_VALUE parameter in rtl/ibex_pkg.sv to change the fixed value. Details of what the ID represents can be found in the RISC-V Privileged Specification.
### 3.7.28 Hardware Thread ID (mhartid)

**CSR Address:** 0xF14

Reads directly return the value of the `hart_id_i` input signal. See also *Core Integration*.

### 3.8 Performance Counters

Ibex implements performance counters according to the RISC-V Privileged Specification, version 1.11 (see Hardware Performance Monitor, Section 3.1.11). The performance counters are placed inside the Control and Status Registers (CSRs) and can be accessed with the `CSRRW(I)` and `CSRRS/C(I)` instructions.

Ibex implements the clock cycle counter `mcycle(h)`, the retired instruction counter `minstret(h)`, as well as the 29 event counters `mhpmcounter3(h) - mhpmcounter31(h)` and the corresponding event selector CSRs `mhpmevent3 - mhpmevent31`, and the `mcountinhibit` CSR to individually enable/disable the counters. `mcycle(h)` and `minstret(h)` are always available and 64 bit wide. The `mhpmcounter` performance counters are optional (unavailable by default) and parametrizable in width.

#### 3.8.1 Event Selector

The following events can be monitored using the performance counters of Ibex.

<table>
<thead>
<tr>
<th>Event ID/Bit</th>
<th>Event Name</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NumCycles</td>
<td>Number of cycles</td>
</tr>
<tr>
<td>2</td>
<td>NumInstrRet</td>
<td>Number of instructions retired</td>
</tr>
<tr>
<td>3</td>
<td>NumCyclesLSU</td>
<td>Number of cycles waiting for data memory</td>
</tr>
<tr>
<td>4</td>
<td>NumCyclesIF</td>
<td>Cycles waiting for instruction fetches, i.e., number of instructions wasted due to non-ideal caching</td>
</tr>
<tr>
<td>5</td>
<td>NumLoads</td>
<td>Number of data memory loads. Misaligned accesses are counted as two accesses</td>
</tr>
<tr>
<td>6</td>
<td>NumStores</td>
<td>Number of data memory stores. Misaligned accesses are counted as two accesses</td>
</tr>
<tr>
<td>7</td>
<td>NumJumps</td>
<td>Number of unconditional jumps (j, jal, jr, jalr)</td>
</tr>
<tr>
<td>8</td>
<td>NumBranches</td>
<td>Number of branches (conditional)</td>
</tr>
<tr>
<td>9</td>
<td>NumBranches-Taken</td>
<td>Number of taken branches (conditional)</td>
</tr>
<tr>
<td>10</td>
<td>NumInstrRetC</td>
<td>Number of compressed instructions retired</td>
</tr>
<tr>
<td>11</td>
<td>NumCyclesMul-Wait</td>
<td>Cycles waiting for multiply to complete</td>
</tr>
<tr>
<td>12</td>
<td>NumCyclesDiv-Wait</td>
<td>Cycles waiting for divide to complete</td>
</tr>
</tbody>
</table>

The event selector CSRs `mhpmevent3 - mhpmevent31` define which of these events are counted by the event counters `mhpmcounter3(h) - mhpmcounter31(h)`. If a specific bit in an event selector CSR is set to 1, this means that events with this ID are being counted by the counter associated with that selector CSR. If an event selector CSR is 0, this means that the corresponding counter is not counting any event.
### 3.8.2 Controlling the counters from software

By default, all available counters are enabled after reset. They can be individually enabled/disabled by overwriting the corresponding bit in the `mcountinhibit` CSR at address `0x320` as described in the RISC-V Privileged Specification, version 1.11 (see Machine Counter-Inhibit CSR, Section 3.1.13). In particular, to enable/disable `mcycle(h)`, bit 0 must be written. For `minstret(h)`, it is bit 2. For event counter `mhpmcounterX(h)`, it is bit X.

The lower 32 bits of all counters can be accessed through the base register, whereas the upper 32 bits are accessed through the h-register. Reads to all these registers are non-destructive.

### 3.8.3 Parametrization at synthesis time

The `mcycle(h)` and `minstret(h)` counters are always available and 64 bit wide.

The event counters `mhpmcounter3(h) - mhpmcounter31(h)` are parametrizable. Their width can be parametrized between 1 and 64 bit through the `WidthMHPMCounters` parameter, which defaults to 40 bit wide counters.

The number of available event counters `mhpmcounterX(h)` can be controlled via the `NumMHPMCounters` parameter. By default (`NumMHPMCounters set to 0`), no counters are available to software. Set `NumMHPMCounters` to a value between 1 and 8 to make the counters `mhpmcounter3(h) - mhpmcounter10(h)` available as listed below. Setting `NumMHPMCounters` to values larger than 8 does not result in any more performance counters.

Unavailable counters always read 0.

The association of events with the `mpmcnter` registers is hardwired as listed in the following table.

<table>
<thead>
<tr>
<th>Event Counter</th>
<th>CSR Address</th>
<th>Event ID/Bit</th>
<th>Event Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mcycle(h)</code></td>
<td>0xB00 (0xB80)</td>
<td>0</td>
<td><code>NumCycles</code></td>
</tr>
<tr>
<td><code>minstret(h)</code></td>
<td>0xB02 (0xB82)</td>
<td>2</td>
<td><code>NumInstrRet</code></td>
</tr>
<tr>
<td><code>mhpmcounter3(h)</code></td>
<td>0xB03 (0xB83)</td>
<td>3</td>
<td><code>NumCyclesLSU</code></td>
</tr>
<tr>
<td><code>mhpmcounter4(h)</code></td>
<td>0xB04 (0xB84)</td>
<td>4</td>
<td><code>NumCyclesIF</code></td>
</tr>
<tr>
<td><code>mhpmcounter5(h)</code></td>
<td>0xB05 (0xB85)</td>
<td>5</td>
<td><code>NumLoads</code></td>
</tr>
<tr>
<td><code>mhpmcounter6(h)</code></td>
<td>0xB06 (0xB86)</td>
<td>6</td>
<td><code>NumStores</code></td>
</tr>
<tr>
<td><code>mhpmcounter7(h)</code></td>
<td>0xB07 (0xB87)</td>
<td>7</td>
<td><code>NumJumps</code></td>
</tr>
<tr>
<td><code>mhpmcounter8(h)</code></td>
<td>0xB08 (0xB88)</td>
<td>8</td>
<td><code>NumBranches</code></td>
</tr>
<tr>
<td><code>mhpmcounter9(h)</code></td>
<td>0xB09 (0xB89)</td>
<td>9</td>
<td><code>NumBranchesTaken</code></td>
</tr>
<tr>
<td><code>mhpmcounter10(h)</code></td>
<td>0xB0A (0xB8A)</td>
<td>10</td>
<td><code>NumInstrRetC</code></td>
</tr>
<tr>
<td><code>mhpmcounter11(h)</code></td>
<td>0xB0B (0xB8B)</td>
<td>11</td>
<td><code>NumCyclesMulWait</code></td>
</tr>
<tr>
<td><code>mhpmcounter12(h)</code></td>
<td>0xB0C (0xB8C)</td>
<td>12</td>
<td><code>NumCyclesDivWait</code></td>
</tr>
</tbody>
</table>

Similarly, the event selector CSRs are hardwired as follows. The remaining event selector CSRs are tied to 0, i.e., no events are counted by the corresponding counters.
### 3.8.4 FPGA Targets

For FPGA targets the performance counters constitute a particularly large structure. Implementing the maximum 29 event counters 32, 48 and 64 bit wide results in relative logic utilizations of the core of 100%, 111% and 129% respectively. The relative numbers of flip-flops are 100%, 125% and 150%. It is recommended to implement event counters of 32 bit width where possible.

For Xilinx FPGA devices featuring the *DSP48E1* DSP slice or similar, counter logic can be absorbed into the DSP slice for widths up to 48 bits. The resulting relative logic utilizations with respect to the non-DSP 32 bit counter implementation are 83% and 89% respectively for 32 and 48 bit DSP counters. This comes at the expense of 1 DSP slice per counter. For 32 bit counters only, the corresponding flip-flops can be incorporated into the DSP’s output pipeline register, resulting in a reduction of the number of flip-flops to 50%. In order to infer DSP slices for performance counters, define the preprocessor variable `FPGA_XILINX`.

### 3.9 Exceptions and Interrupts

Ibex implements trap handling for interrupts and exceptions according to the RISC-V Privileged Specification, version 1.11.

When entering an interrupt/exception handler, the core sets the `mepc` CSR to the current program counter and saves `mstatus.MIE` to `mstatus.MPIE`. All exceptions cause the core to jump to the base address of the vector table in the `mtvec` CSR. Interrupts are handled in vectored mode, i.e., the core jumps to the base address plus four times the interrupt ID. Upon executing an `MRET` instruction, the core jumps to the program counter previously saved in the `mepc` CSR and restores `mstatus.MPIE` to `mstatus.MIE`.

The base address of the vector table is initialized to the boot address (must be aligned to 256 bytes, i.e., its least significant byte must be 0x00) when the core is booting. The base address can be changed after bootup by writing to the `mtvec` CSR. For more information, see the *Control and Status Registers* documentation.

The core starts fetching at the address made by concatenating the most significant 3 bytes of the boot address and the reset value (0x80) as the least significant byte. It is assumed that the boot address is supplied via a register to avoid long paths to the instruction fetch unit.
3.9.1 Privilege Modes

Ibex supports operation in Machine Mode (M-Mode) and User Mode (U-Mode). The core resets into M-Mode and will jump to M-Mode on any interrupt or exception. On execution of an MRET instruction, the core will return to the Privilege Mode stored in \texttt{mstatus.MPP}.

3.9.2 Interrupts

Ibex supports the following interrupts.

<table>
<thead>
<tr>
<th>Interrupt Input Signal</th>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>irq_nm_i</td>
<td>31</td>
<td>Non-maskable interrupt (NMI)</td>
</tr>
<tr>
<td>irq_fast_i[14:0]</td>
<td>30:16</td>
<td>15 fast, local interrupts</td>
</tr>
<tr>
<td>irq_external_i</td>
<td>11</td>
<td>Connected to platform-level interrupt controller</td>
</tr>
<tr>
<td>irq_timer_i</td>
<td>7</td>
<td>Connected to timer module</td>
</tr>
<tr>
<td>irq_software_i</td>
<td>3</td>
<td>Connected to memory-mapped (inter-processor) interrupt register</td>
</tr>
</tbody>
</table>

All interrupts except for the non-maskable interrupt (NMI) are controlled via the \texttt{mstatus}, \texttt{mie} and \texttt{mip} CSRs. After reset, all interrupts are disabled. To enable interrupts, both the global interrupt enable (MIE) bit in the \texttt{mstatus} CSR and the corresponding individual interrupt enable bit in the \texttt{mie} CSR need to be set. For more information, see the \textit{Control and Status Registers} documentation.

If multiple interrupts are pending, they are handled in the priority order defined by the RISC-V Privileged Specification, version 1.11 (see Machine Interrupt Registers, Section 3.1.9). The fast interrupts have a platform defined priority. In Ibex they take priority over all other interrupts and between fast interrupts the highest priority is given to the interrupt with the lowest ID.

The NMI is enabled independent of the values in the \texttt{mstatus} and \texttt{mie} CSRs, and it is not visible through the \texttt{mip} CSR. It has interrupt ID 31, i.e., it has the highest priority of all interrupts and the core jumps to the trap-handler base address (in \texttt{mtvec}) plus 0x7C to handle the NMI. When handling the NMI, all interrupts including the NMI are ignored. Nested NMIs are not supported.

All interrupt lines are level-sensitive. It is assumed that the interrupt handler signals completion of the handling routine to the interrupt source, e.g., through a memory-mapped register, which then deasserts the corresponding interrupt line.

In Debug Mode, all interrupts including the NMI are ignored independent of \texttt{mstatus.MIE} and the content of the \texttt{mie} CSR.

3.9.3 Internal Interrupts

Some events produce an ‘internal interrupt’. An internal interrupt produces an NMI (using the same vector as the external NMI) with \texttt{mcause} and \texttt{mtval} being set to indicate the cause of the internal interrupt. The external NMI takes priority over all internal interrupts. Entering the handler for an internal interrupt automatically clears the internal interrupt. Internal interrupts are considered to be non-recoverable in general. Specific details of how an internal interrupt relates to the event that triggers it are listed below. Given these details it may be possible for software to recover from an internal interrupt under specific circumstances.

The possible \texttt{mcause} values for an internal interrupt as listed below:

<table>
<thead>
<tr>
<th>\texttt{mcause}</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFFFE0</td>
<td>Load integrity error internal interrupt. Only generated when SecureIbex == 0. \texttt{mtval} gives the faulting address. The interrupt will be taken at most one instruction after the faulting load. In particular a load or store immediately after a faulting load may execute before the interrupt is taken.</td>
</tr>
<tr>
<td>0x8000</td>
<td>External NMI</td>
</tr>
</tbody>
</table>

3.9. Exceptions and Interrupts
3.9.4 Recoverable Non-Maskable Interrupt

To support recovering from an NMI happening during a trap handling routine, Ibex features additional CSRs for backing up `mstatus.MPP`, `mstatus.MPIE`, `mepc` and `mcause`. These CSRs are not accessible by software running on the core.

These CSRs are nonstandard. For more information, see the corresponding proposal.

3.9.5 Exceptions

Ibex can trigger an exception due to the following exception causes:

<table>
<thead>
<tr>
<th>Exception Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Instruction access fault</td>
</tr>
<tr>
<td>2</td>
<td>Illegal instruction</td>
</tr>
<tr>
<td>3</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>5</td>
<td>Load access fault</td>
</tr>
<tr>
<td>7</td>
<td>Store access fault</td>
</tr>
<tr>
<td>8</td>
<td>Environment call from U-Mode (ECALL)</td>
</tr>
<tr>
<td>11</td>
<td>Environment call from M-Mode (ECALL)</td>
</tr>
</tbody>
</table>

The illegal instruction exception, instruction access fault, LSU error exceptions and ECALL instruction exceptions cannot be disabled and are always active.

3.9.6 Nested Interrupt/Exception Handling

Ibex does support nested interrupt/exception handling in software. The hardware automatically disables interrupts upon entering an interrupt/exception handler. Otherwise, interrupts/exceptions during the critical part of the handler, i.e. before software has saved the `mepc` and `mstatus` CSRs, would cause those CSRs to be overwritten. If desired, software can explicitly enable interrupts by setting `mstatus.MIE` to 1'b1 from within the handler. However, software should only do this after saving `mepc` and `mstatus`. There is no limit on the maximum number of nested interrupts. Note that, after enabling interrupts by setting `mstatus.MIE` to 1'b1, the current handler will be interrupted also by lower priority interrupts. To allow higher priority interrupts only, the handler must configure `mie` accordingly.

The following pseudo-code snippet visualizes how to perform nested interrupt handling in software.

```c
1  isr_handle_nested_interrupts(id) {
2      // Save mepc and mstatus to stack
3      mepc_bak = mepc;
4      mstatus_bak = mstatus;
5      // Save mie to stack (optional)
6      mie_bak = mie;
7      // Keep lower-priority interrupts disabled (optional)
8      mie = ~((1 << (id + 1)) - 1);  
9      // Re-enable interrupts
10     mstatus.MIE = 1;
11    // Handle interrupt
12    // This code block can be interrupted by other interrupts.
13
14}  (continues on next page)
```
Nesting of interrupts/exceptions in hardware is not supported. The purpose of the nonstandard mstack CSRs in Ibex is only to support recoverable NMIs. These CSRs are not accessible by software. While handling an NMI, all interrupts are ignored independent of mstatus.MIE. Nested NMIs are not supported.

### 3.9.7 Double Fault Detection

Ibex has a mechanism to detect when a double fault has occurred. A double fault is defined as a synchronous exception occurring whilst handling a previous synchronous exception. The cpuctrl custom CSR has fields to provide software visibility and access to this mechanism.

When a synchronous exception occurs, Ibex sets cpuctrl.sync_exception_seen. Ibex clears cpuctrl.sync_exception_seen when mret is executed. If a synchronous exception occurs whilst cpuctrl.sync_exception_seen is set, a double fault has been detected.

When a double fault is detected, the double_fault_seen_o output is asserted for one cycle and cpuctrl.double_fault_seen is set. Note that writing the cpuctrl.double_fault_seen field has no effect on the double_fault_seen_o output.

### 3.10 Physical Memory Protection (PMP)

The Physical Memory Protection (PMP) unit implements region-based memory access checking in-accordance with the RISC-V Privileged Specification, version 1.11 and includes the Trusted Execution Environment (TEE) working group proposal PMP Enhancements for memory access and execution prevention on Machine mode (Smeppmp) version 0.9.3. The following configuration parameters are available to control PMP checking:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMPEnable</td>
<td>0</td>
<td>PMP support enabled</td>
</tr>
<tr>
<td>PMPNumRegions</td>
<td>4</td>
<td>Number of implemented regions (1 - 16)</td>
</tr>
<tr>
<td>PMPGranularity</td>
<td>0</td>
<td>Minimum match granularity $2^{G+2}$ bytes (0 - 31)</td>
</tr>
</tbody>
</table>

When PMPEnable is zero, the PMP module is not instantiated and all PMP registers read as zero (regardless of the value of PMPNumRegions)
3.10.1 PMP Integration

Addresses from the instruction fetch unit and load-store unit are passed to the PMP module for checking, and the output of the PMP check is used to gate the external request. To maintain consistency with external errors, the instruction fetch unit and load-store unit progress with their request as if it was granted externally. The PMP error is registered and consumed by the core when the data would have been consumed.

3.10.2 PMP Granularity

The PMP granularity parameter is used to reduce the size of the address matching comparators by increasing the minimum region size. When the granularity is greater than zero, NA4 mode is not available and will be treated as OFF mode.

3.10.3 PMP Enhancements

These are described in more detail in PMP Enhancements for memory access and execution prevention on Machine mode (Smepmp) version 0.9.3. If Ibex is configured to include PMP (PMPEnable is not zero) the PMP enhancements are always included. Use of the enhanced behavior is optional, if no writes to mseccfg occur PMP behavior will remain exactly as specified in the RISC-V privileged specification. The enhancements add:

- A new CSR mseccfg providing functionality to allow locked regions to be modified and to implement default deny for M-mode accesses.
- New PMP region configurations which are U-Mode or M-Mode accessible only with varying read/write/execute settings along with some shared U and M mode accessible configurations. These new configurations supersede the original ones and are enabled via mseccfg.

3.10.4 Custom Reset Values

By default all PMP CSRs (include mseccfg) are reset to 0. Some applications may want other reset values. Default reset values are defined in ibex_pmp_reset_default.svh. An implementation can either modify this file or define IBEX_CUSTOM_PMP_RESET_VALUES and place a copy of ibex_pmp_result_default.svh in a new file, ibex_pmp_reset.svh, changing the values as required and adding the new file to the include path of whatever build flow is being used.

3.11 Security Features

Ibex implements a set of extra features (when the SecureIbex parameter is set) to support security-critical applications. All features are runtime configurable via bits in the cpuctrl custom CSR.

3.11.1 Outputs

Ibex has three alert outputs for signalling security issues. The internal major alert (alert_major_internal_o) indicates a critical security issue from which the core cannot recover which was detected internally in ibex_top. The bus major alert (alert_major_internal_o) indicates a critical security issue from which the core cannot recover which was detected on incoming bus data. The minor alert (alert_minor_o) indicates potential security issues which can be monitored over time by a system.
3.11.2 Data Independent Timing

When enabled (via the `data_ind_timing` control bit in the `cpuctrl` register), execution times and power consumption of all instructions shall be independent of input data. This makes it more difficult for an external observer to infer secret data by observing power lines or exploiting timing side-channels.

In Ibex, most instructions already execute independent of their input operands. When data-independent timing is enabled:

- Branches execute identically regardless of their taken/not-taken status
- Early completion of multiplication by zero/one is removed
- Early completion of divide by zero is removed

Note that data memory operations to unaligned addresses might result in multiple bus accesses being made. This in turn could expose information about the address as a timing side-channel. It is therefore recommended to stick to aligned memory accesses when using this feature for critical code regions.

When Ibex is configured to use an instruction cache, stalls on instruction fetch can see variable latency (depending on whether or not they hit in the cache). Software that has need of data independent timing may wish to disable the instruction cache to avoid this or to carefully analyse execution to determine if variable latency introduced by the cache causes unacceptable leakage. The instruction cache is controlled by the `icache_enable` bit in the `cpuctrl` register. Precise details of fetch timing will depend upon the memory system Ibex is connected to.

3.11.3 Dummy Instruction Insertion

When enabled (via the `dummy_instr_en` control bit in the `cpuctrl` register), dummy instructions will be inserted at random intervals into the execution pipeline. The dummy instructions have no functional impact on processor state, but add some difficult-to-predict timing and power disruption to executed code. This disruption makes it more difficult for an attacker to infer what is being executed, and also makes it more difficult to execute precisely timed fault injection attacks.

The frequency of injected instructions can be tuned via the `dummy_instr_mask` bits in the `cpuctrl` register.

<table>
<thead>
<tr>
<th><code>dummy_instr_mask</code></th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Dummy instruction every 0 - 4 real instructions</td>
</tr>
<tr>
<td>001</td>
<td>Dummy instruction every 0 - 8 real instructions</td>
</tr>
<tr>
<td>011</td>
<td>Dummy instruction every 0 - 16 real instructions</td>
</tr>
<tr>
<td>111</td>
<td>Dummy instruction every 0 - 32 real instructions</td>
</tr>
</tbody>
</table>

Other values of `dummy_instr_mask` are legal, but will have a less predictable impact.

The interval between instruction insertion is randomized in the core using an LFSR. The initial seed and output permutation for this LFSR can be set using parameters from the top-level of Ibex. Software can periodically re-seed this LFSR with true random numbers (if available) via the `secureseed` CSR. This will make the insertion interval of dummy instructions much harder for an attacker to predict.

Note that the dummy instruction feature inserts multiply and divide instructions. The core must be configured with a multiplier (`RV32M != ibex_pkg::RV32MNone`) or errors will occur using this feature.
3.11.4 Bus integrity checking

Extra signals are available alongside the instruction and data side memory channels to support bus integrity checking. When the SecureIbex parameter is set, incoming data will be checked against the supplied checkbits. An **internal interrupt** will be generated and a bus major alert signalled if there is a mismatch. Where load data has bad checkbits the write to the load’s destination register will be suppressed. Write data can be checked against the supplied checkbits at its destination to confirm integrity.

3.11.5 Register file ECC

When Ibex is configured with the SecureIbex parameter, ECC checking is added to all reads of the register file. This can be useful to detect fault injection attacks since the register file covers a reasonably large area. No attempt is made to correct detected errors, but an internal major alert is signaled for the system to take action.

3.11.6 ICache ECC

The ICache can be configured with ECC protection. When an ECC error is detected a minor alert is signaled. See *Cache ECC protection* for more information.

3.11.7 Hardened PC

This adds a check that the PC driven from the IF stage has not been modified. A check is asserted that the current IF stage PC equals the previous PC plus the correct increment. The check is disabled after branches and after reset. If a mismatch is detected, an internal major alert is signaled.

3.11.8 Shadow CSRs

Certain critical CSRs (*mstatus*, *mtvec*, *cpucrtl*, *pmpcfg* and *pmpaddr*) have extra glitch detection enabled. This creates a second copy of the register which stores a complemented version of the main CSR data. A constant check is made that the two copies are consistent, and an internal major alert is signalled if not. Note that this feature is not currently used when the SecureIbex parameter is set due to overlap with dual core lockstep.

3.11.9 Dual core lockstep

This configuration option instantiates a second copy of the core logic, referred to as the shadow core. The shadow core executes using a delayed version of all inputs supplied to the main core. All outputs of the shadow core are compared against a delayed version of the outputs of the main core. Any mismatch between the two sets of outputs will trigger an internal major alert.

Note that the register file and icache RAMs are not duplicated since these units are covered by ECC protection.
3.12 Debug Support

Ibex offers support for execution-based debug according to the RISC-V Debug Specification, version 0.13.

**Note:** Debug support in Ibex is only one of the components needed to build a System on Chip design with run-control debug support (think “the ability to attach GDB to a core over JTAG”). Additionally, a Debug Module and a Debug Transport Module, compliant with the RISC-V Debug Specification, are needed.

A supported open source implementation of these building blocks can be found in the RISC-V Debug Support for PULP Cores IP block.

The OpenTitan project can serve as an example of how to integrate the two components in a toplevel design.

### 3.12.1 Interface

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug_req_i</td>
<td>input</td>
<td>Request to enter Debug Mode</td>
</tr>
</tbody>
</table>

**debug_req_i** is the “debug interrupt”, issued by the debug module when the core should enter Debug Mode.

### 3.12.2 Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DmHaltAddr</td>
<td>Address to jump to when entering Debug Mode</td>
</tr>
<tr>
<td>DmExceptionAddr</td>
<td>Address to jump to when an exception occurs while in Debug Mode</td>
</tr>
<tr>
<td>DbgTriggerEn</td>
<td>Enable support for debug triggers</td>
</tr>
<tr>
<td>DbgHwBreakNum</td>
<td>Number of debug triggers</td>
</tr>
</tbody>
</table>

### 3.12.3 Core Debug Registers

Ibex implements four core debug registers, namely **Debug Control and Status Register (dcsr)**, **Debug PC Register (dpc)**, and two debug scratch registers. If the DbgTriggerEn parameter is set, debug trigger registers are available. See **Trigger Select Register (tselect)**, **Trigger Data Register 1 (tdata1)** and **Trigger Data Register 2 (tdata2)** for details. All those registers are accessible from Debug Mode only. If software tries to access them without the core being in Debug Mode, an illegal instruction exception is triggered.

### 3.13 Tracer

The module **ibex_tracer** can be used to create a log of the executed instructions. It is used by **ibex_top_tracing** which forwards the RVFI signals to the tracer (see also **RISC-V Formal Interface**).
3.13.1 Output file

All traced instructions are written to a log file. By default, the log file is named `trace_core_<HARTID>.log`, with `<HARTID>` being the 8 digit hart ID of the core being traced.

The file name base, defaulting to `trace_core` can be set using the `ibex_tracer_file_base` plusarg passed to the simulation. For example, `+ibex_tracer_file_base=ibex_my_trace` will produce log files named `ibex_my_trace_<HARTID>.log`. The exact syntax of passing plusargs to a simulation depends on the simulator.

3.13.2 Disabling the tracer

If the instruction log is not needed for a specific simulation run, the tracer can be disabled.

The plusarg `ibex_tracer_enable` controls the tracer. The tracer is enabled by default. To disable the tracer use `ibex_tracer_enable=0` with the correct plusarg syntax of the simulator.

3.13.3 Trace output format

The trace output is in tab-separated columns.

1. **Time**: The current simulation time.
2. **Cycle**: The number of cycles since the last reset.
3. **PC**: The program counter
4. **Instr**: The executed instruction (base 16). 32 bit wide instructions (8 hex digits) are uncompressed instructions, 16 bit wide instructions (4 hex digits) are compressed instructions.
5. **Decoded instruction**: The decoded (disassembled) instruction in a format equal to what objdump produces when calling it like `objdump -Mnumeric -Mno-aliases -D`.
   - Unsigned numbers are given in hex (prefixed with `0x`), signed numbers are given as decimal numbers.
   - Numeric register names are used (e.g. `x1`).
   - Symbolic CSR names are used.
   - Jump/branch targets are given as absolute address if possible (PC + immediate).
6. **Register and memory contents**: For all accessed registers, the value before and after the instruction execution is given. Writes to registers are indicated as `registername=value`, reads as `registername:value`. For memory accesses, the address and the loaded and stored data are given.

<table>
<thead>
<tr>
<th>Time</th>
<th>Cycle</th>
<th>PC</th>
<th>Instr</th>
<th>Decoded instruction</th>
<th>Register and memory contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>130</td>
<td>61</td>
<td>00000150</td>
<td>4481</td>
<td>c.li x9,0</td>
<td>x9=0x00000000</td>
</tr>
<tr>
<td>132</td>
<td>62</td>
<td>00000152</td>
<td>0008437</td>
<td>lui x8,0x8</td>
<td>x8=0x00008000</td>
</tr>
<tr>
<td>134</td>
<td>63</td>
<td>00000156</td>
<td>fff40413</td>
<td>addi x8,x8,-1</td>
<td>x8,x8,-1 x8:0x0008000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x8:0x0008000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>136</td>
<td>64</td>
<td>0000015a</td>
<td>8c65</td>
<td>c.and x8,x9</td>
<td>x8:0x00007fff</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>142</td>
<td>67</td>
<td>0000015c</td>
<td>c622</td>
<td>c.swsp x8,12(x2)</td>
<td>x2:0x00002000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>142</td>
<td>67</td>
<td>0000015c</td>
<td>c622</td>
<td>c.swsp x8,12(x2)</td>
<td>x2:0x00002000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Chapter 3. Ibex Reference Guide
3.14 Verification

Todo: This section needs to be split into a HOWTO-style user/developer guide, and reference information on the testbench structure.

3.14.1 Ibex Core

Overview

This is a SV/UVM testbench for verification of the Ibex core, located in dv/uvm/core_ibex. At a high level, this testbench uses the open source RISCV-DV random instruction generator to generate compiled instruction binaries, loads them into a simple memory model, stimulates the Ibex core to run this program in memory, and then compares the core trace log against a golden model ISS trace log to check for correctness of execution.

Testbench Architecture

As previously mentioned, this testbench has been constructed based on its usage of the RISC-V-DV random instruction generator developed by Google. A block diagram of the testbench is below.

![Figure 3.10: Architecture of the UVM testbench for Ibex core](image)

Memory Interface Agents

The code can be found in the dv/uvm/core_ibex/common/ibex_mem_intf_agent directory. Two of these agents are instantiated within the testbench, one for the instruction fetch interface, and the second for the LSU interface. These agents run slave sequences that wait for memory requests from the core, and then grant the requests for instructions or for data.

Interrupt Interface Agent

The code can be found in the dv/uvm/core_ibex/common/irq_agent directory. This agent is used to drive stimulus onto the Ibex core’s interrupt pins randomly during test execution.

Memory Model

The code is vendored from OpenTitan and can be found in the vendor/lowrisc_ip/dv/sv/mem_model directory. The testbench instantiates a single instance of this memory model that it loads the compiled assembly test program into at the beginning of each test. This acts as a unified instruction/data memory that serves all requests from both of the memory interface agents.
Test and Sequence Library

The code can be found in the dv/uvm/core_ibex/tests directory. The tests here are the main sources of external stimulus generation and checking for this testbench, as the memory interface slave sequences simply serve the core’s memory requests. The tests here are all extended from core_ibex_base_test, and coordinate the entire flow for a single test, from loading the compiled assembly binary program into the testbench memory model, to checking the Ibex core status during the test and dealing with test timeouts. The sequences here are used to drive interrupt and debug stimulus into the core.

Testplan

The goal of this bench is to fully verify the Ibex core with 100% coverage. This includes testing all RV32IMCB instructions, privileged spec compliance, exception and interrupt testing, Debug Mode operation etc. The complete test list can be found in the file dv/uvm/core_ibex/riscv_dv_extension/testlist.yaml. For details on coverage see the Coverage Plan.

Please note that verification is still a work in progress.

Getting Started

Prerequisites & Environment Setup

In order to run the co-simulation flow, you’ll need:

- A SystemVerilog simulator that supports UVM. The flow is currently tested with VCS.
- The Spike RISC-V instruction set simulator
  lowRISC maintains a lowRISC-specific Spike fork, needed to model: + Cosimulation (needed for verification) + Some custom CSRs + Custom NMI behavior
  Ibex verification should work with the Spike version that is tagged as ibex-cosim-v0.3. Other, later, versions called ibex-cosim-v* may also work but there’s no guarantee of backwards compatibility.
  Spike must be built with the --enable-commitlog and --enable-misaligned options. --enable-commitlog is needed to produce log output to track the instructions that were executed. --enable-misaligned tells Spike to simulate a core that handles misaligned accesses in hardware (rather than jumping to a trap handler).
  Note that Ibex used to support the commercial OVPsim simulator. This is not currently possible because OVPsim doesn’t support the co-simulation approach that we use.
- A working RISC-V toolchain (to compile / assemble the generated programs before simulating them).
  Either download a pre-built toolchain (quicker) or download and build the RISC-V GNU compiler toolchain. For the latter, the Bitmanip patches have to be manually installed to enable support for the Bitmanip draft extension. For further information, checkout the Bitmanip Extension on GitHub and how we create the pre-built toolchains.

Once these are installed, you need to set some environment variables to tell the RISCV-DV code where to find them:

```
export RISCV_TOOLCHAIN=/path/to/riscv
export RISCV_GCC="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-gcc"
export RISCV_OBJCOPY="$RISCV_TOOLCHAIN/bin/riscv32-unknown-elf-objcopy"
export SPIKE_PATH=/path/to/spike/bin
export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:/path/to/spike/lib/pkgconfig
```
End-to-end RTL/ISS co-simulation flow

The last stage in this flow handles log comparisons to determine correctness of a given simulation. To do this, both the trace log produced by the core and the trace log produced by the chosen golden model ISS are parsed to collect information about all register writebacks that occur. These two sets of register writeback data are then compared to verify that the core is writing the correct data to the correct registers in the correct order.

However, this checking model quickly falls apart once situations involving external stimulus (such as interrupts and debug requests) start being tested, as while ISS models can simulate traps due to exceptions, they cannot model traps due to external stimulus. In order to provide support for these sorts of scenarios to verify if the core has entered the proper interrupt handler, entered Debug Mode properly, updated any CSRs correctly, and so on, the handshake mechanism provided by the RISCV-DV instruction generator is heavily used, which effectively allows the core to send status information to the testbench during program execution for any analysis that is required to increase verification effectiveness. This mechanism is explained in detail at https://github.com/google/riscv-dv/blob/master/docs/source/handshake.rst. As a sidenote, the signature address that this testbench uses for the handshaking is 0x8fffffff. Additionally, as is mentioned in the RISCV-DV documentation of this handshake, a small set of API tasks are provided in dv/uvm/core_ibex/tests/core_ibex_base_test.sv to enable easy and efficient integration and usage of this mechanism in this test environment. To see how this handshake is used during real simulations, look in dv/uvm/core_ibex/tests/core_ibex_test_lib.sv. As can be seen, this mechanism is extensively used to provide runtime verification for situations involving external debug requests, interrupt assertions, and memory faults. To add another layer of correctness checking to the checking already provided by the handshake mechanism, a modified version of the trace log comparison is used, as comparing every register write performed during the entire simulation will lead to an incorrect result since the ISS trace log will not contain any execution information in the debug ROM or in any interrupt handler code. As a result, only the final values contained in every register at the end of the test are compared against each other, since any code executed in the debug ROM and trap handlers should not corrupt register state in the rest of the program.

The entirety of this flow is controlled by the Makefile found at dv/uvm/core_ibex/Makefile; here is a list of frequently used commands:

```
cd dv/uvm/core_ibex
make
```

Figure 3.11: RTL/ISS co-simulation flow chart

(continues on next page)
# Run a full regression, redirect the output directory
make OUT=xxx

# Run a single test
make TEST=riscv_machine_mode_rand_test ITERATIONS=1

# Run a test with a specific seed, dump waveform
make TEST=riscv_machine_mode_rand_test ITERATIONS=1 SEED=123 WAVES=1

# Verbose logging
make ... VERBOSE=1

# Run multiple tests in parallel through LSF
make ... LSF_CMD="bsub -Is"

# Get command reference of the simulation script
python3 sim.py --help

# Generate the assembly tests only
make gen

# Compile and run RTL simulation
make TEST=xxx compile,rtl_sim

# Run a full regression with coverage
make COV=1

**Run with a different RTL simulator**

You can add any compile/runtime options in `dv/uvm/core_ibex/yaml/simulator.yaml`.

```bash
# Use the new RTL simulator to run
make ... SIMULATOR=xxx
```

### 3.14.2 Instruction Cache

#### Overview

Due to the complexity of the instruction cache, a separate testbench is used to ensure that full verification and coverage closure is performed on this module. This testbench is located at `dv/uvm/icache/dv`.

As icache verification is being carried out as part of the OpenTitan open-source project, the testbench derives from the `dv_lib` UVM class library, which is a set of extended UVM classes that provides basic UVM testbench functionality and components.

This DV environment will be compiled and simulated using the `dvsim` simulation tool. The master `.hjson` file that controls simulation with `dvsim` can be found at `dv/uvm/icache/dv/ibex_icache_sim_cfg.hjson`. The associated testplan `.hjson` file is located at `dv/uvm/icache/data/ibex_icache_testplan.hjson`. As this testbench is still in its infancy, it is currently only able to be compiled, as no tests or sequences are implemented, nor are there any entries in the testplan file. To build the testbench locally using the VCS simulator, run the following command from the root of the Ibex repository:
Specify the intended output directory using either the --sr or --scratch-root option. The --skip-ral option is mandatory for building/simulating the Icache testbench, as it does not have any CSRs, excluding this option will lead to build errors. --purge directs the tool to rm -rf the output directory before running the tool, this can be removed if not desired.

3.15 Co-simulation System

3.15.1 Overview

A co-simulation system is provided that can run in either the Ibex UVM DV environment or with Simple System. This system runs a RISC-V ISS (currently only Spike is supported) in lockstep with an Ibex core. All instructions executed by Ibex and memory transactions generated are checked against the behaviour of the ISS. This system supports memory errors, interrupt and debug requests which are observed in the RTL simulation and forwarded to the ISS so the ISS and RTL remain in sync. The system uses a generic interface to allow support of multiple ISSes. Only VCS is supported as a simulator, though no VCS specific functionality is required so adding support for another simulator should be straightforward.

To run the co-simulation system, a particular version of Spike is required (see the Setup and Usage section, below).

The RISC-V Formal Interface (RVFI) is used to provide information about retired instructions and instructions that produce synchronous traps for checking. The RVFI has been extended to provide interrupt and debug information and the value of the mcycle CSR. These extended signals have the prefix rvfi_ext.

The co-simulation system is EXPERIMENTAL. It is disabled by default in the UVM DV environment currently, however it is intended to become the primary checking method for the UVM testbench.

3.15.2 Setup and Usage

Clone the lowRISC fork of Spike and check out the ibex-cosim-v0.3 tag. Other, later, versions called ibex-cosim-v* may also work but there’s no guarantee of backwards compatibility. Follow the Spike build instructions to build and install Spike. The --enable-commitlog and --enable-misaligned options must be passed to configure. We recommend using a custom install location (using --prefix=<path> with configure) to avoid cluttering system directories. Note that, if you do this, you will also need to add an entry to PKG_CONFIG_PATH so that pkg-config can tell us how to build against the installed Spike libraries.

To build/run the UVM DV environment with the co-simulator, add the COSIM=1 argument to the make command. To build Simple System with the co-simulator, build the lowrisc:ibex:ibex_simple_system_cosim core.

Quick Build and Run Instructions

Build and install the co-simulator

```bash
# Get the Ibex co-simulation spike branch
git clone -b ibex_cosim https://github.com/lowRISC/riscv-isa-sim.git riscv-isa-sim-cosim

# Setup build directory
cd riscv-isa-sim-cosim
```

(continues on next page)
mkdir build
cd build

# Configure and build spike
../configure --enable-commitlog --enable-misaligned --prefix=/opt/spike-cosim
sudo make -j8 install

Run the UVM DV regression with co-simulation enabled

# Run regression with co-simulation enabled
cd <ibex_area>/dv/uvm/core_ibex
make COSIM=1

Build and run Simple System with the co-simulation enabled

# Build simulator
fusesoc --cores-root=. run --target=sim --setup --build lowrisc:ibex:ibex_simple_system_cosim --system_cosim --RV32E=0 --RV32M=ibex_pkg::RV32MFast

# Build coremark test binary, with performance counter dump disabled. The co-simulator system doesn't produce matching performance counters in spike so any read of those CSRs results in a mismatch and a failure.
make -C ./examples/sw/benchmarks/coremark SUPPRESS_PCOUNT_DUMP=1

# Run coremark binary with co-simulation checking
build/lowrisc_ibex_ibex_simple_system_cosim_0/sim-verilator/Vibex_simple_system --meminit=ram,examples/sw/benchmarks/coremark SUPPRESS_PCOUNT_DUMP=1

3.15.3 Co-simulation details

The co-simulation system uses DPI calls to link the DV and ISS sides together. A C++ interface is defined in `dv/cosim/cosim.h` with a DPI wrapper provided by `dv/cosim/cosim_dpi.cc` and `dv/cosim/cosim_dpi.h`. A handle, which points to some class instance that implements the interface, must be provided by the DV environment. All the co-simulation DPI calls take this handle as a first argument.

The details below discuss the C++ interface. The DPI version of the interface is almost identical, with all functions prefaced with `riscv_cosim` and taking a handle of the co-simulation instance to use.

The core function of the co-simulation interface is the `step` function:

```cpp
virtual bool step(uint32_t write_reg, uint32_t write_reg_data, uint32_t pc, bool sync_, --trap);
```

`step` takes arguments giving the PC of the most recently retired or synchronously trapping instruction in the DUT along with details of any register write that occurred.

Where `step` is provided with a retired (successfully executed) instruction it steps the ISS by one instruction and checks it executed the same instruction, with the same register write result, as the DUT.

When `step` is provided with an instruction that produces a synchronous trap, it checks the ISS also traps on the same instruction but does not step to the next executed instruction. That instruction will be the first instruction of the trap handler and will be checked/stepped by the next call to `step` when it retires from the DUT.

Any data memory accesses that the ISS produces during the `step` are checked against observed DUT memory accesses.
step returns false if any checks have failed. If any errors occur during the step they can be accessed via get_errors which returns a vector of error messages. For the DPI interface errors are accessed using riscv_cosim_get_num_errors and riscv_cosim_get_error. When errors have been checked they can be cleared with clear_errors.

Trap Handling

Traps are separated into two categories, synchronous and asynchronous. Synchronous traps are caused by a particular instruction’s execution (e.g. an illegal instruction). Asynchronous traps are caused by external interrupts. Note that in Ibex error responses to both loads and store produce a synchronous trap so the co-simulation system has the same behaviour.

A synchronous trap is associated with a particular instruction and prevents that instruction from completing its execution. That instruction doesn’t retire, but is still made visible on the RVFI. The rvfi_trap signal is asserted for an instruction that causes a synchronous trap. As described above step should be called for any instruction that causes a synchronous trap to check the trap is also seen by the ISS.

An asynchronous trap can be seen as occurring between instructions and as such doesn’t have an associated instruction, nothing will be seen on RVFI with rvfi_trap set. The co-simulation system will immediately take any pending asynchronous trap when step is called, expecting the instruction checked with step to be the first instruction of the trap handler.

While a debug request is not strictly an asynchronous trap (it doesn’t use the same exception handling mechanism), they work identically to asynchronous traps for the co-simulation system. When a debug request is pending when step is called the co-simulation will expect the instruction checked by step to be the first instruction of the debug handler.

Interrupts and Debug Requests

The DV environment must observe any incoming interrupts and debug requests generated by the testbench and notify the co-simulation system of them using set_mip, set_debug_req and set_nmi. An interrupt or debug request will take immediate effect at the next step (if architecturally required to do so). The DV environment is responsible for determining when to call set_mip, set_debug_req and set_nmi to ensure a RTL and co-simulation match.

The state of the incoming interrupts and debug request is sampled when an instruction moves from IF to ID/EX. The sampled state is tracked with the rest of the RVFI pipeline and used to call set_mip, set_debug_req and set_nmi when the instruction is output by the RVFI. See the comments in rtl/ibex_core.sv, around the new_debug_req, new_nmi and new_irq signals for further details.

Memory Access Checking and Bus Errors

The co-simulation system must be informed of all Dside accesses performed by the RTL using notify_dside_access. See dv/cosim/cosim.h for further details. As Ibex doesn’t perform speculative Dside memory accesses, all notified accesses are expected to match with accesses performed by the ISS in the same order they are notified.

Accesses notified via notify_dside_access can specify they saw an error response, the co-simulation system will produce the appropriate trap when the ISS attempts to access the address that saw the error.

Accesses must be notified before they occur in the ISS for the access matching and trapping on errors to work.

Iside accesses from Ibex can be speculative, so there is no simple link between accesses produced by the RTL and the accesses performed by the ISS for the Iside. This means no direct checking of Iside accesses is done, however errors on the Iside accesses that result in an instruction fault trap need to be notified to the co-simulation system.
set_iside_error does this, it is provided with the address that saw the bus error and it should be called immediately before the step that will process the trap. The co-simulation system will produce an instruction fault trap if it attempts to access the provided error address in the step call following the set_iside_error call.

Two methods are available for dealing with bus errors on the Iside, they differ in where they probe. One probes on the external instr_X memory interface, the other probes internally within the IF stage. The probe used is selected by the probe_imem_for_err field of the core_ibex_cosim_cfg structure. When set external probing is used, otherwise internal probing is used.

Both probe points look for addresses that have seen bus errors. If an instruction entering ID/EX fetches from an address that has seen a bus error (as recorded by one of the probing methods) its rvfi_order_id is recorded. When a faulting instruction is reported on the RVFI and its rvfi_order_id matches a recorded faulting one set_iside_error is called with the faulting address before the next step.

The external interface probe should be used when it is guaranteed that a bus error to address A on the external interface results in a fetch error the next time an instruction with address A is observed entering the ID/EX stage (providing no successful access to A has occurred in the mean time). Otherwise the internal probe should be used. When Ibex is used with the prefetch buffer this guarantee holds and the external probe can be used. When Ibex is used with the instruction cache this guarantee does not hold and the internal probe must be used.

Care should be taken when using the internal probe as it will miss any bug that causes instruction faults to be ignored by the prefetch buffer or ICache (or whatever else has been used in place of these by a custom implementation). In the case of the Ibex ICache a separate testbench ensures instruction faults are dealt with appropriately within the ICache.

Todo: Add detail about security hardening verification.

Note: This testplan is a work in progress still being implemented so this document may not match the implemented verification in the repository.

3.16 Test Plan

3.16.1 Goals

- Verify compliance with all the RISC-V specifications Ibex supports.
- Verify Ibex’s security hardening features.
- Ensure correct functionality is maintained across all possible behaviours of external interfaces (interrupts, memory responses, debug requests etc).
- Hit all functional coverage points, described in Coverage Plan.
3.16.2 Testbench Architecture

Ibex utilises a co-simulation checking approach described in detail in Co-simulation System. With the co-simulation system all instructions Ibex executes and all external events such as an interrupts or memory errors are fed to a golden model. The results of every instruction execution and every memory access are crossed checked against the golden model with any mismatches resulting in a test failure. The aim is to check all possible externally observable behaviours of \texttt{ibex\_top} against the golden model. The golden model used is the Spike RISC-V ISS.

The testbench uses UVM. It consists of 3 agents:

Co-simulation Agent: This has multiple monitors. One monitors the RVFI interface which provides details of retired instructions. The other monitors relate to fetched instructions and instruction memory errors; more details are provided in Coverage Plan. Additionally it connects to the monitor of the Memory Interface Agent for the instruction and data side via analysis ports. The monitored transactions are used by a scoreboard to provide information to the co-simulation system allowing it to step the golden model and check its execution and memory activity against Ibex’s behaviour.

Memory Interface Agent: This provides a driver and a monitor for the Ibex Memory Interface Protocol. The driver provides fully randomised and configurable timings for responses and randomisation of error responses. Two agents are instantiated; one for the data memory interface the other for the instruction memory interface. Read data for memory responses is provided from a backing memory; write requests update the contents of the backing memory. This is separate from the memory used by the golden model in the co-simulation agent. The contents of these two memories will be identical unless there is a mismatch resulting in a failure. The backing memory is held in a memory model as a separate UVM component. The two agents use the same backing memory so they have a coherent view of memory.

IRQ Agent: This provides a driver and a monitor for the IRQ interface. It provides randomised interrupt stimulus to Ibex when a test requests it. Constraints can be used to control types of interrupts generated (e.g. NMI or not) and whether multiple interrupts should be raised together.

Debug and reset signals are a single wire each so do not have a dedicated agent. Instead any sequence that wishes to use them will directly manipulate them via a virtual interface.

The testbench instantiates the agents described above along with the memory model used by both the data and instruction side memory agents. A test consists of executing a pre-built binary (which is loaded into the memory model at the start of the test via backdoor accesses) along with configuring agents to provide appropriate stimulus for the test. Some tests may use the agents to generate stimulus at particular times (e.g. interrupts). A test may perform additional checking on top of the co-simulation golden model comparison where appropriate (e.g. ensuring a raised interrupt has caused an exception).

3.16.3 Stimulus Strategy

Stimulus falls into two categories:

- Instructions to execute: These are generated by the RISC-V DV random instruction generator and provided to the testbench via a raw binary file.
- Activity on external interfaces.

Instructions are generated ahead of time so the test has no control over them at run time. All external interfaces have their stimulus generated at run time so can be controlled by the test. It is the responsibility of the regression run environment to ensure generated instructions are matched with appropriate tests (e.g. ensuring an exception handler is present where interrupts are expected).
Stimulus generation will use a coverage based approach. Stimulus is developed based upon the Coverage Plan. Where coverage is not being hit stimulus will be added to hit it.

3.16.4 Tests

As with stimulus, test sequence development uses a coverage based approach. Tests will be added such that all coverage in the Coverage Plan can be hit. Not all the details of specific tests will be documented here. The test list (dv/uvm/core_ibex/riscv_dv_extension/testlist.yaml), provides an exhaustive list of all tests along with a brief description of what the test does.

A test will execute a binary whilst running zero or more sequences that provide stimulus to external interfaces of ibex_top. As the memory interfaces are all driven by Ibex, with any testbench generated activity in response to a request from Ibex, they do not require explicit sequences run by the test. Instead the test can configure the randomisation of memory delays as it wishes. Memory errors can be configured to always occur in statically defined areas of the memory map or a sequence can be used to inject them via the memory interface agent.

The following sequences are available for tests to use. Each sequence is derived from a base sequence which provides controls to repeat the sequence at fixed or random internals, forever or after a random number of repeats. Full details can be found in dv/uvm/core_ibex/tests/core_ibex_seq_lib.sv.

- **irq_raise_seq** - Raises one or more interrupts. The testbench binary can write to a special memory location to acknowledge the interrupt and cause it to drop. Alternatively the testbench can drop it after a given amount of time.
- **debug_seq** - Raises the external debug request. The testbench binary can write to a special memory location to acknowledge the request and cause it to drop. Alternatively the testbench can drop it after a given amount of time.
- **mem_error_seq** - Injects a memory error in either the instruction side or data side, so the next access results in an error response.
- **reset_seq** - Resets the core.

3.17 Coverage Plan

Todo: Branch prediction hasn’t yet been considered, this will add more coverage points and alter some others

3.17.1 Introduction

Ibex functional coverage is split into two major categories:

- Architectural coverage - which is concerned with instructions being executed and exercising various features of the RISC-V architecture (e.g. PMP) and does not consider the details of how this execution occurs.
- Microarchitectural coverage - which is concerned with the specifics of the RTL operation, ensuring interesting corner cases are seen along with various micro-architectural events (e.g. the different kinds of stall) and combinations of them.

Architectural coverage is not Ibex specific. It can be determined directly from a trace of executed instructions and is handled by RISCV-DV, details can be found in the RISCV-DV documentation.

Microarchitectural coverage will probe the Ibex RTL directly and is described here. There is some inevitable overlap between architectural and microarchitectural coverage but we aim to minimise it.
3.17.2 Coverage Implementation

All coverpoints and cross coverage defined below is associated with a name `cp_name`. This is the name of the coverpoint or cross that implements the described coverage. Coverage is implemented in two files; `dv/uvm/core_ibex/fcov/core_ibex_pmp_fcov_if.sv` for PMP related coverage and `dv/uvm/core_ibex/fcov/core_ibex_fcov_if.sv` for everything else.

3.17.3 Microarchitectural Events and Behaviour

Below are lists of specific things from the microarchitecture that will be included in functional coverage. Each of the points listed below must be covered. This will be further combined using cross coverage which is described in the section below.

**Instructions**

**Categories**

`cp_id_instr_category`

Instructions can be grouped into a number of categories. Each category exercises different data and control paths in the core. For example the `ADD` and `SUB` instructions are in the same category as they are almost identical for the microarchitecture (both read two registers and write to one, both feed operands to the ALU and take their result from it, both have the same response to interrupts etc; the only difference is the ALU operation).

Instructions can be compressed or uncompressed but that isn’t factored into the instruction categories below (excepting for illegal instructions). The decompression occurs in the IF stage and is invisible to the ID/EX stage so isn’t relevant for instruction execution. A separate set of category-agnostic compressed instruction behaviour is considered instead.

An instruction category is sampled at the ID/EX stage (which is where all the varying behaviours actually occur). Some categories are just a single instruction, which is named without further description.

- **ALU** - All of the reg/reg reg/imm instructions that use the ALU. This is any RISC-V instruction with an opcode of `7'b0010011` or `7'b0110011` (`ibex_pkg::OPCODE_OP` and `ibex_pkg::OPCODE_OP_IMM`) other than the `MUL*` and `DIV*` family of instructions (from RV32M).
- **Mul** - Any `MUL*` instruction (from RV32M).
- **Div** - Any `DIV*` instruction (from RV32M).
- **Branch** - Any `B*` family branch instruction.
- **Jump** - `JAL/JALR`
- **Load** - Any `L*` family load instruction.
- **Store** - Any `S*` family load instruction.
- **CSRAccess** - Any instruction from Zicsr.
- **EBreakDbg/EBreakExc** - An `EBREAK` instruction that either enters debug mode (Dbg) or causes an exception (Exc). Which occurs depends upon the setting of `dcsr.ebreakm/dcsr.ebreaku` combined with the privilege level of executed instruction.
  - **ECALL**
  - **MRET**
  - **DRET**
  - **WFI**
• FENCE
• FENCE.I
• FetchError - Any instruction that saw a fetch error.
  • CompressedIllegal - Any compressed instruction with an illegal encoding.
  • UncompressedIllegal - Any uncompressed instruction with an illegal encoding.
  • CSRIllegal - Any instruction attempting a CSR access that is not allowed.
  • PrivIllegal - Illegal due to privilege level or being in/out of debug mode.
  • OtherIllegal - Any other instruction that raises an Illegal instruction exception that isn’t in the other categories.
  • None - No instruction in ID/EX stage.

Stalls

```c
cp_stall_type_id
```

Not all instructions can see all kinds of stalls. A stall category is sampled at the ID/EX stage only (as stalls in IF and WB don’t break down into categories).

• Instr - A stall caused by a multi-cycle instruction. This can be seen by instructions from categories:
  – MUL
  – DIV
  – Branch
  – Jump

• LdHz - A load hazard, the instruction in ID/EX depends upon the result of a load that is awaiting a response in writeback. This can be seen by instructions from categories:
  – ALU
  – Mul
  – Div
  – Branch
  – Jump
  – Load
  – Store
  – CSRAccess

• Mem - Memory stall, the instruction in ID/EX is awaiting a prior memory request to complete before it can begin (to allow precise interrupts on a memory error response). This can be seen for all instruction categories.
Privilege Level

Ibex can operate at either the M (machine) or U (user) privilege levels. Different aspects of the Ibex microarchitecture can be using different privilege levels at once.

- `cp_priv_mode_id` - Privilege level of ID/EX stage instruction.
- `cp_priv_mode_lsu` - Privilege level of LSU operation (ID/EX privilege level modified by `mstatus.mprv` and `mstatus.mpp` settings).

Note that the privilege level of the instruction in WB isn’t retained by the microarchitecture and is not relevant to coverage. The privilege level of the IF instruction is effectively unknown. The instruction is checked when moving from IF to ID/EX against the ID stage privilege level to check if execution is permitted by PMP. Any instruction that reaches WB can be considered bound to retire and any relevant checks and functionality altered by the privilege mode is dealt with at an earlier stage.

Hazards

Ibex hazards all occur in the interaction between the ID and EX stage.

- RAW Reg - Read after write hazard, instruction in ID/EX reads a register that writeback is writing. Split into two versions:
  - RAW load - Instruction in ID/EX reading from destination of load in writeback. Produces a stall (Category LdHz) and shouldn’t forward data. Covered by `cp_stall_type_id`
  - `cp_wb_reg_no_load_hz` - Instruction in writeback isn’t a load. Handled with data forwarding and no stall.
- RAW Load/Store bytes - Load with bytes overlapping a store immediately before it. Covered by `cp_mem_raw_hz`

State Specific Behaviour

Some instructions will behave differently depending upon the state of the processor (e.g. the privilege level the instruction executes at, CSR settings or whether the processor is in debug mode).

- Instruction illegal in U Mode.
  - `cp_mret_in_umode` - MRET
  - `cp_wfi_in_umode` - WFI
  - Read and write to M-mode CSR - Covered by crosses `csr_write_priv_cross` and `csr_read_only_priv_cross`.
- Debug mode instructions (cover execution in and out of debug mode).
  - `DRET`
  - `csr_read_only_debug_cross, csr_write_debug_cross` - Access to debug CSRs.
    * `dcsr`
    * `dpc`
    * `dscratch0`
    * `dscratch1`
– Access to trigger CSRs (also possible in M mode: cover execution in M mode, debug mode and U mode). Covered by csr_read_only_debug_cross, csr_write_debug_cross, csr_read_only_priv_cross, csr_write_priv_cross.

* tselect
* tdata1
* tdata2
* tdata3

• Loads/stores with mstatus.mprv set and unset. Covered by mprv_effect_cross

• EBreak behaviour in U/M mode with different dcsr.ebreakm / dcsr.ebreaku settings. Covered by priv_mode_instr_cross

• cp_single_step_instr - Single step over every instruction category

**Pipeline State**

Each pipeline stage has some associated state.

• **cp_if_stage_state** - IF stage full and fetching, full and idle, empty and fetching, or empty and idle. General IF stage full and stalled uninteresting as will only occur when ID stage is full and stalled.

• **cp_wb_stage_state** - WB stage full and stalled, full and unstalled, or empty

• **cp_id_stage_state** - ID stage full and stalled, full and unstalled, or empty.

• Controller (within ID stage) state machine states

  - **cp_controller_fsm** - Possible transitions between these states. Those marked with a ‘*’ are of particular interest and should be crossed with instruction categories and other coverpoints as appropriate to fully explore the transitions.

    * RESET -> BOOT_SET
    * BOOT_SET -> FIRST_FETCH
    * FIRST_FETCH -> DECODE
    * FIRST_FETCH -> IRQ_TAKEN
    * FIRST_FETCH -> DBG_TAKEN_IF
    * DECODE -> FLUSH *
    * DECODE -> DBG_TAKEN_IF *
    * DECODE -> IRQ_TAKEN *
    * IRQ_TAKEN -> DECODE
    * DBG_TAKEN_IF -> DECODE
    * DBG_TAKEN_ID -> DECODE
    * FLUSH -> DECODE *
    * FLUSH -> DBG_TAKEN_ID
    * FLUSH -> WAIT_SLEEP
    * FLUSH -> IRQ_TAKEN *
    * FLUSH -> DBG_TAKEN_IF *
Exceptions/Interrupts/Debug

Exceptions, interrupts and debug entry can all cause control flow changes combined with CSR writes and privilege level changes and work quite similarly within the controller but not identically. Furthermore they can all occur together and must be appropriately prioritised (consider a instruction with hardware trigger point matching it, that causes some exception and an interrupt is raised the cycle it enters the ID/EX stage)

- Exception from instruction fetch error (covered by the FetchError instruction category).
- pmp_iside_mode_cross - Exception from instruction PMP violation.
- Exception from illegal instruction (covered by the illegal instruction categories).
- cp_ls_error_exception - Exception from memory fetch error.
- pmp_dside_mode_cross - Exception from memory access PMP violation.
- Unaligned memory access
  - misaligned_insn_bus_err_cross, misaligned_data_bus_err_cross - Cover all error and no error scenarios for memory fetch error; first access saw error, second access saw error, neither access saw error
- Interrupt raised/taken.
  - cp_interrupt_taken - Interrupt raised/taken for each available interrupt line. For cross coverage, the precise interrupt that’s raised/taken is not relevant and it only needs to be grouped by NMI vs non-NMI. This is done by using cp_nmi_taken coverpoint in the crosses.
  - interrupt_taken_instr_cross - Interrupt raised/taken the first cycle an instruction is in ID/EX or some other cycle the instruction is in ID/EX.
- cp_debug_req - External debug request.
- cp_single_step_taken - Instruction executed when debug single step enabled.
- cpInsn_trigger_enter_debug - Instruction matches hardware trigger point.
  - cpInsn_trigger_exception - Instruction matching trigger point causes exception
- cp_debug_mode - Ibex operating in debug mode.
- irq_wfi_cross, debug_wfi_cross - Debug and Interrupt whilst sleeping with WFI
  - Cover with global interrupts enabled and disabled
  - Cover with specific interrupt enabled and disabled (Should exit sleep when interrupt is enabled but global interrupts set to disabled, should continue sleeping when both are disabled). Continuing to sleep in the case explained above is covered by cp_irq_continue_sleep, otherwise the behaviour is captured in irq_wfi_cross
- Debug and interrupt occurring whilst entering WFI
  - Covering period between WFI entering ID/EX stage and going into sleep Covered by bin enter_sleep of cp_controller_fsm_sleep that is used by irq_wfi_cross and debug_wfi_cross.
- cp_double_fault - Double fault
PMP

- **cp_region_mode** - Each region configured with different matching modes.
  - Off
  - TOR
  - NA4
  - NAPOT

- **cp_region_priv_bits** - Each region configured with all possible permissions including locked/unlocked.
  - Different permissions with MML enabled and disabled, separate cover points for R/W/X/L values with and without MML.

- **Access fail & pass.**
  - **misaligned_lsu_access_cross** - All combinations of unaligned access split across a boundary, both halves pass, neither pass, just the first passes, just the second passes.
    * Two possible boundary splits; across a 32-bit boundary within a region or a boundary between PMP regions.
  - **cp_pmp_iside_region_override, cp_pmp_iside2_region_override, cp_pmp_dside_region_override** - Higher priority entry allows access that lower priority entry prevents.
  - **pmp_instr_edge_cross** - Compressed instruction access (16-bit) passes PMP but 32-bit access at same address crosses PMP region boundary.

- **Each field of mssecfg enabled/disabled with relevant functionality tested.**
  - **RLB** - rule locking bypass.
    * **rlb_csr_cross** - Try to enable RLB when RLB is disabled and locked regions present.
  - **MMWP** - machine mode whitelist policy.
    * **pmp_dside/iside/iside2_nomatch_cross** - M-mode access fail due to not matching any PMP regions.
    * **mmwp_csr_cross** - Try to disable when enabled.
  - **MML** - machine mode lockdown policy.
    * **rlb_csr_cross** - Try to disable when enabled.

- **Access close to PMP region modification that allows/disallows that access.**

CSRs

Basic read/write functionality must be tested on all implemented CSRs.

- **cp_csr_read_only** - Read from CSR.
- **cp_csr_write** - Write to CSR.
  - Write to read only CSR. Covered by ensuring **cp_csr_write** is seen for read-only CSRs
- **cp_warl_check_CSRNAME** - Write illegal/unsupported value to WARL field for CSR named CSRNAME.
• *csr_read_only_priv_cross*, *csr_write_priv_cross*, *csr_read_only_debug_cross*, *csr_write_debug_cross* - Crosses of reads and writes to CSRs from different privilege levels/debug mode.
  – Access to CSR disallowed due to privilege levels/debug mode Covered by ensuring within the crosses
• *cp_ignored_csrs_ro*, *cp_ignored_csrs_w* - Read and write from/to an unimplemented CSR

CSRs addresses do not need to be crossed with the variety of CSR instructions as these all use the same basic read & write interface into *ibex_cs_registers*. Coverage of the above points will be sampled at the *ibex_cs_registers* interface (as opposed to sampling CSR instructions).

**Miscellaneous**

Various points of interest do not fit into the categories above.

• *instr_unstalled* - Instruction unstalled - Cover the cycle an instruction is unstalled having just been stalled.
• *cp_icache_enable* - Enabling/Disabling ICache.

### 3.17.4 Cross Coverage

Much of the more complex behaviour lies at the combination of the individual microarchitectural behaviours above. Cross coverage is used to capture that. Crosses listed below are ones that don’t already fit into the above categories. There are some broad crosses containing many bins aiming to capture all combinations of some generalised behaviours as well as some more specific ones to capture all combinations of behaviours focused on a particular area.

Cross coverage will be intentionally broad. Where it is proving hard to hit particular bins they will be reviewed in more detail to determine if they’re impossible to hit or if simply hard to hit and whether hitting them provides meaningful gains to verification quality.

Excluded bins will either become illegal bins (where they are impossible to hit, so a failure will be seen if they are hit) or ignore bins (where they don’t factor into coverage statistics). There must be a documented reason a particular bin is added to the illegal or ignore bins.

• *pipe_cross* - Instruction Categories x Pipeline stage states across IF, ID/EX and WB
  – Covers all possibilities of instruction combinations that could fill the pipeline. State only for IF/WB suffices to cover this as all the interesting per instruction behaviour occurs in ID/EX.
  – All bins containing instruction categories other than *None* ignored when ID/EX stage is empty.
• *priv_mode_instr_cross* - Instructions Categories x ID/EX Privilege level
• *stall_cross* - Instruction Categories x Stall Categories
  – Illegal bins will be used to exclude instruction and stall categories that cannot occur.
• *wb_reg_no_load_hz_instr_cross* - Instruction Categories x Hazards
  – *stall_cross* covers the RAW load hazard (as it produces a LdHz stall).
  – RAW hazard between load/store requires no cross coverage as it’s only seen for load and store instructions so the single coverpoint suffices.
• *debug_instruction_cross* - Instruction Categories x Debug Mode
• *controller_instr_cross* - Instruction Categories x Controller state transitions of interest
• `interrupt_takenInstr_cross`, `debug_entry_if_instr_cross`, `pipe_flush_instr_cross` - Interrupt taken/Debug mode entry/Pipe flush x instruction unstalled x instruction category
  
  – Three separate cross coverage groups: one for interrupt, debug and pipe flush.
  
  – Covers all instruction categories being interrupted/entering debug mode/flushing the pipeline both where this occurs during a stall and when it occurs just when they’ve unstalled.

• `exception_stallInstr_cross` - PMP exception x load/store error exception x instruction category x stall type x stalled x irq pending x debug req
  
  – Large cross to cover all possibilities of combinations between interrupt, debug and exceptions for all instruction categories across all stall behaviours.

• `pmp_iside_priv_bits_cross`, `pmp_iside2_priv_bits_cross`, `pmp_dside_priv_bits_cross` - PMP regions x permissions x access fail/pass x privilege level
  
  – Three crosses, one for each PMP channel (instruction, instruction 2 and data).

### 3.18 RISC-V Formal Interface

Ibex supports the [RISC-V Formal Interface](https://riscv.org) (RVFI). This interface basically decodes the current instruction and provides additional insight into the core state thereby enabling formal verification. Examples of such information include opcode, source and destination registers, program counter, as well as address and data for memory operations.

#### 3.18.1 Formal Verification

The signals provided by RVFI can be used to formally verify compliance of Ibex with the [RISC-V specification](https://riscv.org). Currently, the implementation is restricted to support the “I” and “C” extensions, and Ibex is not yet formally verified. It predecessor “Zero-riscy” had been tested, but this required changes to the core as well as to the tool used in the process ([yosys](https://yosys-silicon-engine.org)). The formal verification of the Ibex core is work in progress.

### 3.19 History

Ibex development started in 2015 under the name “Zero-riscy” as part of the PULP platform for energy-efficient computing. Much of the code was developed by simplifying the RV32 CPU core “RI5CY” to demonstrate how small a RISC-V CPU core could actually be [1]. To make it even smaller, support for the “E” extension was added under the code name “Micro-riscy”. In the PULP ecosystem, the core is used as the control core for PULP, PULPino and PULPissimo.

In December 2018 lowRISC took over the development of Zero-riscy and renamed it to Ibex.
3.19.1 References

Ibex is an open source project and invites everyone to contribute. The Ibex Developer Guide documents how Ibex is developed, both in terms of process and tools.

Read on if you would like to work with the Ibex code base to fix a bug, add a feature, or reproduce the verification.

**Todo:** Describe how to set up development environment, how to make changes, etc. Use content from various READMEs and the CONTRIBUTING guide in the repo.

### 4.1 The Ibex Concierge

The Ibex Concierge is the friendly caretaker of the Ibex project. It’s a rotating duty shared by experienced contributors to help newcomers find their way around the project, and to stay on top of the various small tasks necessary to keep the project going.
The Ibex CPU project is a reasonably large open source project. Like all projects we experience two challenges: we want to lend a helping hand to new developers, answering their questions or helping them with code contributions. And we need to stay on top of our “caretaker” tasks, like fixing problems with our continuous integration setup, triaging issues and pull requests, etc. The Ibex Concierge combines these two duties in one person.

Please reach out to the Ibex Concierge if you have trouble finding your way around the Ibex project. You can find today’s Ibex Concierge in the calendar below.

4.1.1 Who is Ibex Concierge today?

The concierge duties rotate between several core developers on a weekly basis. You can find today’s concierge on duty in a public calendar.

- Greg Chadwick (@GregAC)
- Tom Roberts (@tomroberts-lowrisc)
- Rupert Swarbrick (@rswarbrick)
- Pirmin Vogel (@vogelpi)
- Philipp Wagner (@imphil)

You can be Ibex Concierge, too. Please talk to any of the current concierges to discuss!

4.1.2 Ibex Concierge duties

The Ibex Concierge is aware of what’s happening in the Ibex project, and helps to ensure that everyone feels welcome and is able to work productively. The list of duties includes, but isn’t strictly limited to the following tasks.

- Triage incoming issues and pull requests.
  - Assign labels to them.
  - Give initial feedback with an indication of what the next steps are.
  - Answer questions if possible.
  - Ask for clarifications where necessary.
  - Redirect to the right developers as needed.

- Track progress of open issues and pull requests. Ensure contributors always know what’s going on, and are informed if things take longer.

- Welcome new contributors, and provide (hands-on) help to get them up to speed. For example, help them get their commits into good shape, etc.

- Fix or coordinate fixes to necessary infrastructure, such as the continuous integration setup in a timely manner.

- Go through the list of open pull requests: ping developers if information or action is needed, close abandoned pull requests, etc.

- Assist with the review and update of open issues.

- At the end of the week, hand over to the next Ibex Concierge on the rota.

Note the obvious: it is not the job of the Ibex Concierge to fix all bugs, implement all incoming feature requests, or be available 24/7.