Ibex Documentation

Release 0.1.dev80+g3d29e51.d20200707

lowRISC

Jul 07, 2020

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Figure 1.1: Block Diagram

Ibex is a 2-stage in-order 32b RISC-V processor core. Ibex has been designed to be small and efficient. Via two parameters, the core is configurable to support four ISA configurations. Figure 1.1 shows a block diagram of the core.

1.1 Standards Compliance

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

- RISC-V Instruction Set Manual, Volume I: User-Level ISA, document version 20190608-Base-Ratified (June 8, 2019)
- RISC-V Instruction Set Manual, Volume II: Privileged Architecture, document version 20190608-Base-Ratified (June 8, 2019). Ibex implements the Machine ISA version 1.11.
- RISC-V External Debug Support, version 0.13.2
- RISC-V Bit Manipulation Extension, version 0.92 (draft from November 8, 2019)

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 1.1: Ibex Instruction Set Extensions

Extension	Version	Configurability
C: Standard Extension for Compressed Instructions	2.0	always enabled
M: Standard Extension for Integer Multiplication and Division	2.0	optional
B: Draft Extension for Bit Manipulation Instructions	0.92[1]	optional
Zicsr: Control and Status Register Instructions	2.0	always enabled
Zifencei: Instruction-Fetch Fence	2.0	always enabled

[1] Note that while Ibex fully implements draft version 0.92 of the RISC-V Bit Manipulation Extension, this extension 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.

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*

1.2 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 18.9 kGE when using the latch-based register file and implementing the RV32IMC ISA, or 11.6 kGE when implementing the RV32EC ISA.

1.3 FPGA Synthesis

FPGA synthesis is supported for Ibex when the flip-flop based register file is used. Since latches are not well supported on FPGAs, it is crucial to select the flip-flop based register file.

1.4 Contents

- *Getting Started with Ibex* discusses the requirements and initial steps to start using Ibex.
- *Core Integration* provides the instantiation template and gives descriptions of the design parameters as well as the input and output ports.
- *Ibex Pipeline* described the overal pipeline structure.
- *Instruction Decode and Execute* describes how the Instruction Decode and Execute stage works.
- The instruction and data interfaces of Ibex are explained in *Instruction Fetch* and *Load-Store Unit*, respectively.
- *Instruction Cache* describes the optional Instruction Cache.
- The two register-file flavors are described in *Register File*.
- The control and status registers are explained in *Control and Status Registers*.
- *Performance Counters* gives an overview of the performance monitors and event counters available in Ibex.
- *Exceptions and Interrupts* deals with the infrastructure for handling exceptions and interrupts,
- *Physical Memory Protection (PMP)* gives a brief overview of PMP support.
- *Debug Support* gives a brief overview on the debug infrastructure.
- *Tracer* gives a brief overview of the tracer module.
- For information regarding formal verification support, check out *RISC-V Formal Interface*.
- *Examples* gives an overview of how Ibex can be used.

1.5 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.

1.6 References

1. Schiavone, Pasquale Davide, et al. "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications." 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS 2017)

CHAPTER

TWO

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
- Verilator, version 4.028 and up.
- Synopsys VCS
- 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.

2.1 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.

CHAPTER

THREE

GETTING STARTED WITH IBEX

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

3.1 Register File

Ibex comes with two different register file implementations. Depending on the target technology, either the implementation in `ibex_register_file_ff.sv` or the one in `ibex_register_file_latch.sv` should be selected. For more information about the two register file implementations and their trade-offs, check out *Register File*.

CHAPTER

FOUR

CORE INTEGRATION

The main module is named `ibex_core` and can be found in `ibex_core.sv`. Below, the instantiation template is given and the parameters and interfaces are described.

4.1 Instantiation Template

```
ibex_core #(
    .PMPEnable                ( 0                    ),
    .PMPGranularity           ( 0                    ),
    .PMPNumRegions            ( 4                    ),
    .MHPMCounterNum           ( 0                    ),
    .MHPMCounterWidth         ( 40                   ),
    .RV32E                    ( 0                    ),
    .RV32M                    ( 1                    ),
    .RV32B                    ( ibex_pkg::RV32BNone  ),
    .MultiplierImplementation ( "fast"               ),
    .ICache                   ( 0                    ),
    .ICacheECC                ( 0                    ),
    .SecureIbex               ( 0                    ),
    .DbgTriggerEn             ( 0                    ),
    .DmHaltAddr               ( 32'h1A110800         ),
    .DmExceptionAddr          ( 32'h1A110808         )
) u_core (
    // Clock and reset
    .clk_i          (),
    .rst_ni         (),
    .test_en_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_err_i    (),

    // Data memory interface
    .data_req_o     (),
```

(continues on next page)

(continued from previous page)

```
    .data_gnt_i     (),
    .data_rvalid_i  (),
    .data_we_o      (),
    .data_be_o      (),
    .data_addr_o    (),
    .data_wdata_o   (),
    .data_rdata_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    (),

    // Special control signals
    .fetch_enable_i (),
    .core_sleep_o   ()
);
```

4.2 Parameters

Name	Type/Range	Default	Description
PMPEnable	bit	0	Enable PMP support
PMPGranularity	int (0..31)	0	Minimum granularity of PMP address matching
PMPNumRegions	int (1..16)	4	Number implemented PMP regions (ignored if PMPEnable == 0)
MHPMCounterNum	int (0..10)	0	Number of performance monitor event counters
MHPMCounterWidth	int (64..1)	40	Bit width of performance monitor event counters
RV32E	bit	0	RV32E mode enable (16 integer registers only)
RV32M	bit	1	M(ultiply) extension enable
RV32B	ibex_pkg::rv32b_e	RV32BNone	*EXPERIMENTAL* - B(itmanipulation) extension select: "RV32BNone": No B-extension "RV32BBalanced": Sub-extensions Zbb, Zbs, Zbf and Zbt "RV32Full": All sub-extensions
BranchTargetALU	bit	0	*EXPERIMENTAL* - Enables branch target ALU removing a stall cycle from taken branches
WritebackStage	bit	0	*EXPERIMENTAL* - Enables third pipeline stage (writeback) improving performance of loads and stores
MultiplierImplementation	string	"fast"	Multiplicator type: "slow": multi-cycle slow, "fast": multi-cycle fast, "single-cycle": single-cycle
ICache	bit	0	*EXPERIMENTAL* Enable instruction cache instead of prefetch buffer
ICacheECC	bit	0	*EXPERIMENTAL* Enable SECDED ECC protection in ICache (if ICache == 1)
SecureIbex	bit	0	*EXPERIMENTAL* Enable various additional features targeting secure code execution.
DbgTriggerEn	bit	0	Enable debug trigger support (one trigger only)
DmHaltAddr	int	0x1A110800	Address to jump to when entering Debug Mode
DmExceptionAddr	int	0x1A110808	Address to jump to when an exception occurs while in Debug Mode

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

4.3 Interfaces

Signal(s)	Width	Dir	Description
`clk_i`	1	in	Clock signal
`rst_ni`	1	in	Active-low asynchronous reset
`test_en_i`	1	in	Test input, enables clock
`hart_id_i`	32	in	Hart ID, usually static, can be read from *Hardware Thread ID (mhartid)* CSR
`boot_addr_i`	32	in	First program counter after reset = `boot_addr_i` + 0x80, see *Exceptions and Interrupts*
`instr_*`	Instruction fetch interface, see *Instruction Fetch*		
`data_*`	Load-store unit interface, see *Load-Store Unit*		
`irq_*`	Interrupt inputs, see *Exceptions and Interrupts*		
`debug_*`	Debug interface, see *Debug Support*		
`fetch_enable_i`	1	in	When it comes out of reset, the core will not start fetching and executing instructions until it sees this pin set to 1'b1. Once started, it will continue until the next reset, regardless of the value of this pin.
`core_sleep_o`	1	out	Core in WFI with no outstanding data or instruction accesses. Deasserts if an external event (interrupt or debug req) wakes the core up

Figure 4.1: Ibex Pipeline

CHAPTER

FIVE

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.

5.1 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.

5.2 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.

Instruction Type	Stall Cycles	Description
Integer Computational	0	Integer Computational Instructions are defined in the RISCV-V RV32I Base Integer Instruction Set.
CSR Access	0	CSR Access Instruction are defined in 'Zicsr' of the RISC-V specification.
Load/Store	1 - N	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.
Multiplication	0/1 (Single-Cycle Multiplier) 2/3 (Fast Multi-Cycle Multiplier) clog2(`op_b`)/32 (Slow Multi-Cycle Multiplier)	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)*.
Division Remainder	1 or 37	1 stall cycle if divide by 0, otherwise full long division. See details in *Multiplier/Divider Block (MULT/DIV)*
Jump	1 - N	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.
Branch (Not-Taken)	0	Any branch where the condition is not met will not stall.
Branch (Taken)	2 - N 1 - N (Branch Target ALU enabled)	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.
Instruction Fence	1 - N	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).

CHAPTER

SIX

INSTRUCTION FETCH

rtl/ibex_if_stage.sv.

Figure 6.1: 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.

6.1 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.

Signal	Direction	Description
`instr_req_o`	output	Request valid, must stay high until `instr_gnt_i` is high for one cycle
`instr_addr_o[31:0]`	output	Address, word aligned
`instr_gnt_i`	input	The other side accepted the request. `instr_req_o` may be deasserted in the next cycle.
`instr_rvalid_i`	input	`instr_rdata_i` holds valid data when `instr_rvalid_i` is high. This signal will be high for exactly one cycle per request.
`instr_rdata_i[31:0]`	input	Data read from memory
`instr_err_i`	input	Memory access error

6.2 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.

6.3 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.

CHAPTER

SEVEN

INSTRUCTION DECODE AND EXECUTE

Figure 7.1: 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.

7.1 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.

7.2 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

7.3 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.

7.4 Register File

Source Files: `rtl/ibex_register_file_ff.sv` `rtl/ibex_register_file_latch.sv`

See *Register File* for more details.

7.5 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.

7.6 Arithmetic Logic Unit (ALU)

Source File: `rtl/ibex_alu.sv`

The Arithmetic Logic Logic (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 (draft version 0.92 from November 8, 2019) is optional.[1] It can be enabled via the enumerated parameter `RV32B` defined in `rtl/ibex_pkg.sv`.

There are two versions of the bit manipulation extension available: The balanced implementation comprises a set of sub-extensions aiming for good benefits at a reasonable area overhead. The full implementation comprises all 32 bit instructions defined in the extension. The following table lists the implemented instructions in each version. Multi-cycle instructions are completed in 2 cycles. All remaining instructions complete in a single cycle.

[1] Ibex fully implements draft version 0.92 of the RISC-V Bit Manipulation Extension. This extension 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.

Z-Extension	Version	Multi-Cycle Instructions
Zbb (Base)	Balanced/Full	rol, ror[i]
Zbs (Single-bit)	Balanced/Full	None
Zbp (Permutation)	Full	None
Zbp (Bit extract/deposit)	Full	All
Zbf (Bit-field place)	Balanced/Full	All
Zbc (Carry-less multiply)	Full	None
Zbr (CRC)	Full	All
Zbt (Ternary)	Balanced/Full	All
Zb_tmp (Temporary)[2]	Balanced/Full	None

The implementation of the B-extension comes with an area overhead of 1.8 to 3.0 kGE for the balanced version and 6.0 to 8.7 kGE for the full version. That corresponds to an approximate percentage increase in area of 9 to 14 % and 25 to 30 % for the balanced and full versions respectively. The ranges correspond to synthesis results generated using relaxed and maximum frequency targets respectively. The designs have been synthesized using Synopsys Design Compiler targeting TSMC 65 nm technology.

7.7 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 parameter `MultiplierImplementation`.

Single-Cycle Multiplier This implementation is chosen by setting the `MultiplierImplementation` parameter to "single-cycle". 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 $\star$ 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 `MultiplierImplementation` parameter to "fast". 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.

[2] The sign-extend instructions *sext.b/sext.h* are defined but not unambiguously categorized in draft version 0.92 of the extension. Temporarily, they have been assigned a separate Z-extension (Zb_tmp) both in Ibex and the RISCV-DV random instruction generator used to verify the bit manipulation instructions in Ibex.

- 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 `MultiplierImplementation` parameter to "slow".

- 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
- Cycles 4 - 36: Perform long division as described here: https://en.wikipedia.org/wiki/Division_algorithm#Integer_division_(unsigned)_with_remainder.

7.8 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*

7.9 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.

CHAPTER

EIGHT

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.

8.1 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)fetching 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.

8.2 Configuration options

The following table describes the available configuration parameters.

Parameter	Default	Description
BusWidth	32	Width of instruction bus. Note, this is fixed at 32 for Ibex at the moment.
CacheSizeKB	4	Size of cache in bytes.
CacheECC	1'b0	Enable SECDED ECC protection in tag and data RAMs.
LineSize	64	The width of one cache line in bits. Line sizes smaller than 64 bits may give compilation errors.
NumWays	2	The number of ways.
SpecRequest	1'b0	When set, the system will attempt to speculatively request data from memory in parallel with the cache lookup. This can give improved performance for workloads which cache poorly (at the expense of power). When not set, only branches will make speculative requests.
BranchCache	1'b0	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.

8.3 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 I$ will attempt to fetch this address first from the bus. The I$ 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.

8.4 RAM Arrangement

The data RAMs are arranged as `NumWays` banks of `LineSize` width. If ECC is configured, the tag and data banks will be wider to accomodate the extra checkbits.

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

Cache config	Tag RAMs	Data RAMs
4kB, 2 way, 64bit line	2 x 256 x 22bit	2 x 256 x 64bit
4kB, 2 way, 64bit line w/ECC	2 x 256 x 28bit	2 x 256 x 72bit
4kB, 2 way, 128bit line	2 x 128 x 22bit	2 x 128 x 128bit
4kB, 4 way, 64bit line	4 x 128 x 22bit	4 x 128 x 64bit

8.5 Sub Unit Description

Figure 8.1: Instruction Cache Block Diagram

8.5.1 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.

8.5.2 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.

8.5.3 Fill buffers

The fill buffers perform several functions in the I$ and constitute most of it's 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.

8.5.4 Data output

Figure 8.2: 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.

8.5.5 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):

ECC checkbits	Valid bit	Tag
[27:22]	[21]	[20:0]

For the Data RAMs (64bit line):

ECC checkbits	Data
[71:64]	[63:0]

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 I$ 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.

8.5.6 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.

8.5.7 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 I$ 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 I$, which doesn't appear at the `ibex_core` 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 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.

CHAPTER

NINE

LOAD-STORE UNIT

`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).

9.1 Data-Side Memory Interface

Signals that are used by the LSU:

Signal	Direction	Description
`data_req_o`	output	Request valid, must stay high until `data_gnt_i` is high for one cycle
`data_addr_o[31:0]`	output	Address, word aligned
`data_we_o`	output	Write Enable, high for writes, low for reads. Sent together with `data_req_o`
`data_be_o[3:0]`	output	Byte Enable. Is set for the bytes to write/read, sent together with `data_req_o`
`data_wdata_o[31:0]`	output	Data to be written to memory, sent together with `data_req_o`
`data_gnt_i`	input	The other side accepted the request. Outputs may change in the next cycle.
`data_rvalid_i`	input	`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.
`data_err_i`	input	Error response from the bus or the memory: request cannot be handled. High in case of an error.
`data_rdata_i[31:0]`	input	Data read from memory

9.2 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.

9.3 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 9.1, Figure 9.2 and Figure 9.3 show example-timing diagrams of the protocol.

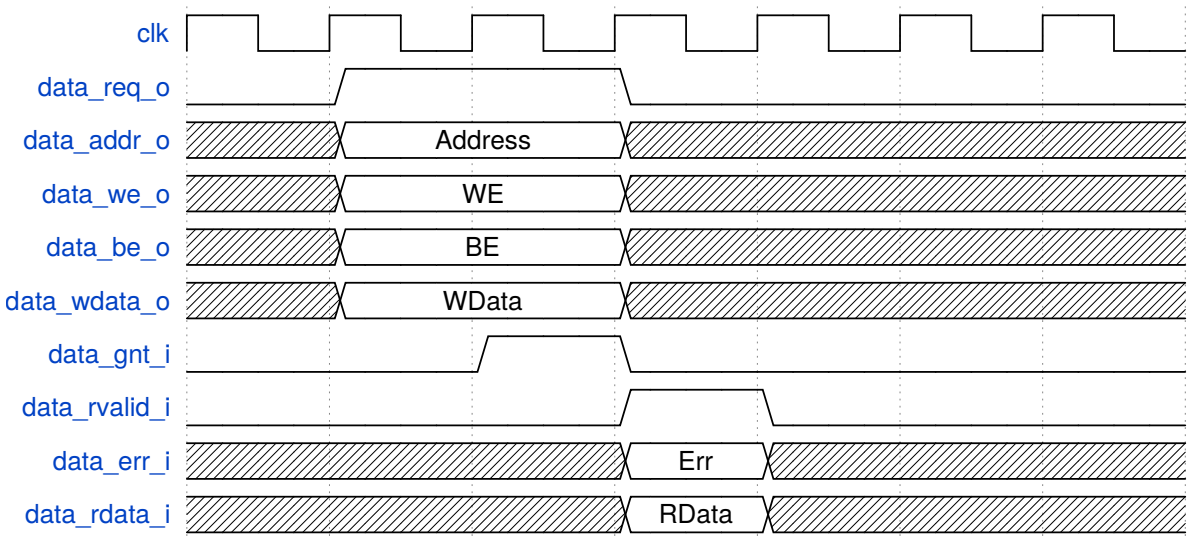


Figure 9.1: Basic Memory Transaction

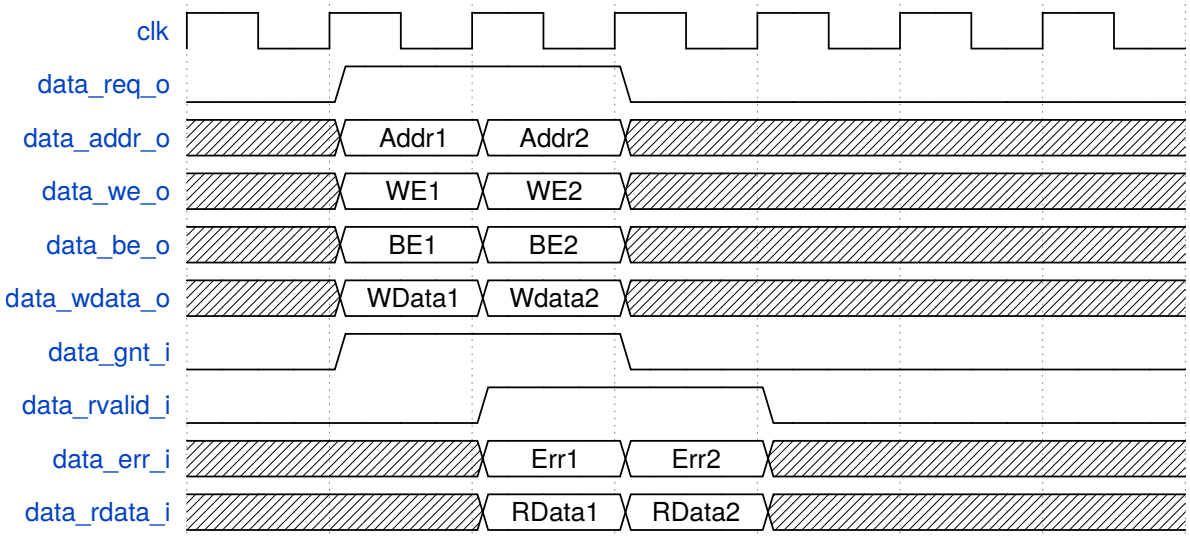


Figure 9.2: Back-to-back Memory Transaction

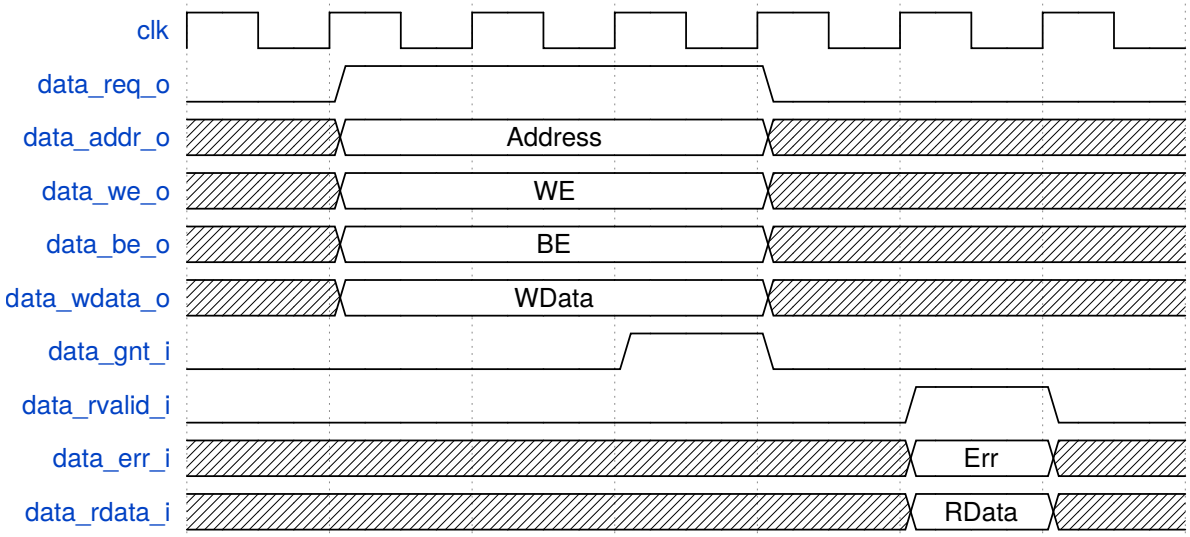


Figure 9.3: Slow Response Memory Transaction

CHAPTER

TEN

REGISTER FILE

Source FIles: `rtl/ibex_register_file_ff.sv` `rtl/ibex_register_file_latch.sv`

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.

There are two flavors of register file available, both having their own benefits and trade-offs.

10.1 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**.

To select the flip-flop-based register file, make sure to use the source file `ibex_register_file_ff.sv` in your project.

10.2 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, make sure to use the source file `ibex_register_file_fpga.sv` in your project.

10.3 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, make sure to use the source file `ibex_register_file_latch.sv` in your project. 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 10.1. 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.

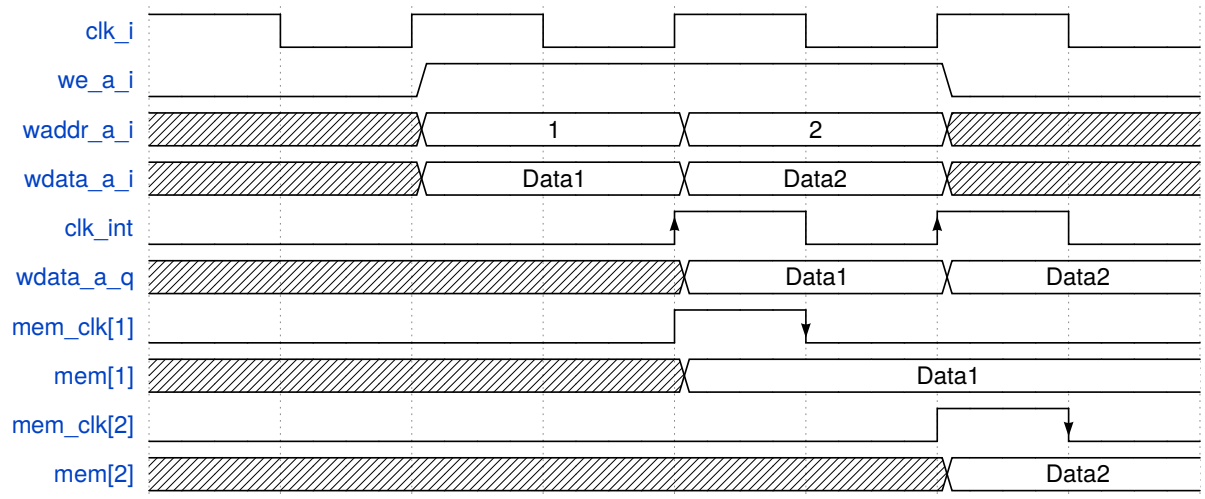


Figure 10.1: Latch-based register file operation

CHAPTER

ELEVEN

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.

Address	Name	Access	Description
0x300	mstatus	WARL	Machine Status
0x301	misa	WARL	Machine ISA and Extensions
0x304	mie	WARL	Machine Interrupt Enable Register
0x305	mtvec	WARL	Machine Trap-Vector Base Address
0x320	mcountinhibit	RW	Machine Counter-Inhibit Register
0x323	mhpmevent3	WARL	Machine Performance-Monitoring Event Sele
....			
0x33F	mhpmevent31	WARL	Machine Performance-Monitoring Event Sele
0x340	mscratch	RW	Machine Scratch Register
0x341	mepc	WARL	Machine Exception Program Counter
0x342	mcause	WLRL	Machine Cause Register
0x343	mtval	WARL	Machine Trap Value Register
0x344	mip	R	Machine Interrupt Pending Register
0x3A0	pmpcfg0	WARL	PMP Configuration Register
....			
0x3A3	pmpcfg3	WARL	PMP Configuration Register
0x3B0	pmpaddr0	WARL	PMP Address Register
....			
0x3BF	pmpaddr15	WARL	PMP Address Register
0x7A0	tselect	WARL	Trigger Select Register
0x7A1	tdata1	WARL	Trigger Data Register 1
0x7A2	tdata2	WARL	Trigger Data Register 2
0x7A3	tdata3	WARL	Trigger Data Register 3
0x7A8	mcontext	WARL	Machine Context Register
0x7AA	scontext	WARL	Supervisor Context Register
0x7B0	dcsr	WARL	Debug Control and Status Register
0x7B1	dpc	RW	Debug PC
0x7B2	dscratch0	RW	Debug Scratch Register 0
0x7B3	dscratch1	RW	Debug Scratch Register 1
0x7C0	cpuctrl	WARL	CPU Control Register (Custom CSR)
0x7C1	secureseed	WARL	Security feature random seed (Custom CSR)
0xB00	mcycle	RW	Machine Cycle Counter
0xB02	minstret	RW	Machine Instructions-Retired Counter
0xB03	mhpmcounter3	WARL	Machine Performance-Monitoring Counter

continues on next pa

Table 11.1 – continued from previous page

Address	Name	Access	Description
....			
0xB1F	mhpmcounter31	WARL	Machine Performance-Monitoring Counter
0xB80	mcycleh	RW	Upper 32 bits of mcycle
0xB82	minstreth	RW	Upper 32 bits of minstret
0xB83	mhpmcounter3h	WARL	Upper 32 bits of mhmpcounter3
....			
0xB9F	mhpmcounter31h	WARL	Upper 32 bits of mhmpcounter31
0xF14	mhartid	R	Hardware Thread ID

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

11.1 Machine Status (mstatus)

CSR Address: 0x300

Reset Value: 0x0000_1800

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

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.

11.2 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.

11.3 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.

Bit#	Interrupt
30:16	Machine Fast Interrupt Enables: Set bit x+16 to enable fast interrupt `irq_fast_i[x]`.
11	**Machine External Interrupt Enable (MEIE):** If set, `irq_external_i` is enabled.
7	**Machine Timer Interrupt Enable (MTIE):** If set, `irq_timer_i` is enabled.
3	**Machine Software Interrupt Enable (MSIE):** if set, `irq_software_i` is enabled.

11.4 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.

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

11.5 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.

11.6 Machine Cause (mcause)

CSR Address: `0x342`

Reset Value: `0x0000_0000`

Bit#	R/W	Description
31	R	**Interrupt:** This bit is set when the exception was triggered by an interrupt.
4:0	R	**Exception Code**

When an exception is encountered, the corresponding exception code is stored in this register.

11.7 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.

11.8 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.

Bit#	Interrupt
30:16	Machine Fast Interrupts Pending: If bit x+16 is set, fast interrupt `irq_fast_i[x]` is pending.
11	**Machine External Interrupt Pending (MEIP):** If set, `irq_external_i` is pending.
7	**Machine Timer Interrupt Pending (MTIP):** If set, `irq_timer_i` is pending.
3	**Machine Software Interrupt Pending (MSIP):** if set, `irq_software_i` is pending.

11.9 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.

31:24	23:16	15:8	7:0
pmp3cfg	pmp2cfg	pmp1cfg	pmp0cfg

The configuration fields for each region are as follows:

Bit#	Definition
7	Lock
6:5	Reserved (Read as zero)
4:3	Mode
2	Execute permission
1	Write permission
0	Read permission

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.

11.10 PMP Address Register (pmpaddrx)

CSR Address: `0x3B0 - 0x3BF`

Reset Value: `0x0000_0000`

`pmpaddrx` are registers to set address matching for PMP regions.

31:0
address[33:2]

11.11 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).

Ibex implements a single trigger, therefore this register will always read as zero.

11.12 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.

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

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

11.13 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.

11.14 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.

11.15 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.

11.16 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.

11.17 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.

Bit#	R/W	Description
31:28	R	**xdebugver:** 4 = External spec-compliant debug support exists.
15	RW	**ebreakm:** EBREAK in M-Mode behaves as described in Privileged Spec (0), or enters Debug Mode (1).
12	WARL	**ebreaku:** EBREAK in U-Mode behaves as described in Privileged Spec (0), or enters Debug Mode (1).
8:6	R	**cause:** 1 = EBREAK, 2 = trigger, 3 = halt request, 4 = step
2	RW	**step:** When set and not in Debug Mode, execute a single instruction and enter Debug Mode.
1:0	WARL	**prv:** 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).

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.

11.18 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.

11.19 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.

11.20 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.

11.21 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.

Bit#	R/W	Description
5:3	WARL	**dummy_instr_mask:** 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*).
2	WARL	**dummy_instr_en:** 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*).
1	WARL	**data_ind_timing:** 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.
0	WARL	**icache_enable:** 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.

11.22 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.

11.23 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.

11.24 Hardware Thread ID (mhartid)

CSR Address: `0xF14`

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

CHAPTER

TWELVE

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.

12.1 Event Selector

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

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

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.

12.2 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.

12.3 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 `mphmcounter` registers is hardwired as listed in the following table.

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

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.

Event Selector	CSR Address	Reset Value	Event ID/Bit
`mhpmevent3(h)`	0x323	0x0000_0008	3
`mhpmevent4(h)`	0x324	0x0000_0010	4
`mhpmevent5(h)`	0x325	0x0000_0020	5
`mhpmevent6(h)`	0x326	0x0000_0040	6
`mhpmevent7(h)`	0x327	0x0000_0080	7
`mhpmevent8(h)`	0x328	0x0000_0100	8
`mhpmevent9(h)`	0x329	0x0000_0200	9
`mhpmevent10(h)`	0x32A	0x0000_0400	10
`mhpmevent11(h)`	0x32B	0x0000_0800	11
`mhpmevent12(h)`	0x32C	0x0000_1000	12

12.4 FPGA Targets

For FPGA targets the performance counters constitute a particularily 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`.

CHAPTER

THIRTEEN

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.

13.1 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 `mstatus`.MPP.

13.2 Interrupts

Ibex supports the following interrupts.

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

All interrupts except for the non-maskable interrupt (NMI) are controlled via the `mstatus`, `mie` and `mip` CSRs. After reset, all interrupts are disabled. To enable interrupts, both the global interrupt enable (MIE) bit in the `mstatus` CSR and the corresponding individual interrupt enable bit in the `mie` CSR need to be set. For more information, see the *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 highest priority is given to the interrupt with the highest ID, except for timer interrupts, which have the lowest priority.

The NMI is enabled independent of the values in the `mstatus` and `mie` CSRs, and it is not visible through the `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 `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 `mstatus`.MIE and the content of the `mie` CSR.

13.3 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.

13.4 Exceptions

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

Exception Code	Description
1	Instruction access fault
2	Illegal instruction
3	Breakpoint
5	Load access fault
7	Store access fault
8	Environment call from U-Mode (ECALL)
11	Environment call from M-Mode (ECALL)

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

13.5 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.

```
isr_handle_nested_interrupts(id) {
  // Save mpec and mstatus to stack
  mepc_bak = mepc;
  mstatus_bak = mstatus;

  // Save mie to stack (optional)
  mie_bak = mie;

  // Keep lower-priority interrupts disabled (optional)
  mie = ~((1 << (id + 1)) - 1);

  // Re-enable interrupts
  mstatus.MIE = 1;

  // Handle interrupt
  // This code block can be interrupted by other interrupts.
  // ...

  // Restore mstatus (this disables interrupts) and mepc
  mstatus = mstatus_bak;
  mepc = mepc_bak;

  // Restore mie (optional)
  mie = mie_bak;
}
```

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.

CHAPTER

FOURTEEN

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. The following configuration parameters are available to control PMP checking:

Parameter	Default value	Description
PMPEnable	0	PMP support enabled
PMPNumRegions	4	Number of implemented regions (1 - 16)
PMPGranularity	0	Minimum match granularity 2^{G+2} bytes (0 - 31)

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

14.1 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.

14.2 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.

CHAPTER

FIFTEEN

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.

15.1 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

15.2 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.

dummy_instr_mask	Interval
000	Dummy instruction every 0 - 4 real instructions
001	Dummy instruction every 0 - 8 real instructions
011	Dummy instruction every 0 - 16 real instructions
111	Dummy instruction every 0 - 32 real instructions

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. Sofware 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.

CHAPTER

SIXTEEN

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.

16.1 Interface

Signal	Direction	Description
`debug_req_i`	input	Request to enter Debug Mode

`debug_req_i` is the "debug interrupt", issued by the debug module when the core should enter Debug Mode.

16.2 Parameters

Parameter	Description
`DmHaltAddr`	Address to jump to when entering Debug Mode
`DmExceptionAddr`	Address to jump to when an exception occurs while in Debug Mode
`DbgTriggerEn`	Enable support for debug triggers

16.3 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.

CHAPTER

SEVENTEEN

TRACER

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

17.1 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.

17.2 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.

```
Time          Cycle      PC       Instr    Decoded instruction Register and memory␣
↪contents
         130         61 00000150 4481     c.li    x9,0        x9=0x00000000
         132         62 00000152 00008437 lui     x8,0x8      x8=0x00008000
         134         63 00000156 fff40413 addi    x8,x8,-1    x8:0x00008000 ␣
↪x8=0x00007fff
         136         64 0000015a 8c65     c.and   x8,x9       x8:0x00007fff ␣
↪x9:0x00000000  x8=0x00000000
         142         67 0000015c c622     c.swsp  x8,12(x2)   x2:0x00002000 ␣
↪x8:0x00000000 PA:0x0000200c store:0x00000000  load:0xffffffff
```

CHAPTER

EIGHTEEN

RISC-V FORMAL INTERFACE

Ibex supports the RISC-V Formal Interface (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.

18.1 Formal Verification

The signals provided by RVFI can be used to formally verify compliance of Ibex with the RISC-V specification.

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). The formal verification of the Ibex core is work in progress.

CHAPTER

NINETEEN

VERIFICATION

19.1 Ibex Core

19.1.1 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.

19.1.2 Testbench Architecture

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

Figure 19.1: Architecture of the UVM testbench for Ibex core

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 can be found in the dv/uvm/core_ibex/common/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 RV32IMC 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.

Please note that verification is still a work in progress.

19.1.3 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.
- A RISC-V instruction set simulator. For example, Spike or OVPsim. Note that Spike must be configured with `--enable-commitlog` and `--enable-misaligned`. The commit log is needed to track the instructions that were executed and `--enable-misaligned` tells Spike to simulate a core that handles misaligned accesses in hardware (rather than jumping to a trap handler). In addition, Spike does not support the RISC-V Bit Manipulation Extension (Bitmanip) by default. To support this draft extension implemented in Ibex, the riscv-bitmanip branch of Spike needs to be used.
- 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 OVPSIM_PATH=/path/to/ovpsim/bin
```

(Obviously, you only need to set `SPIKE_PATH` or `OVPSIM_PATH` if you have installed the corresponding instruction set simulator)

End-to-end RTL/ISS co-simulation flow

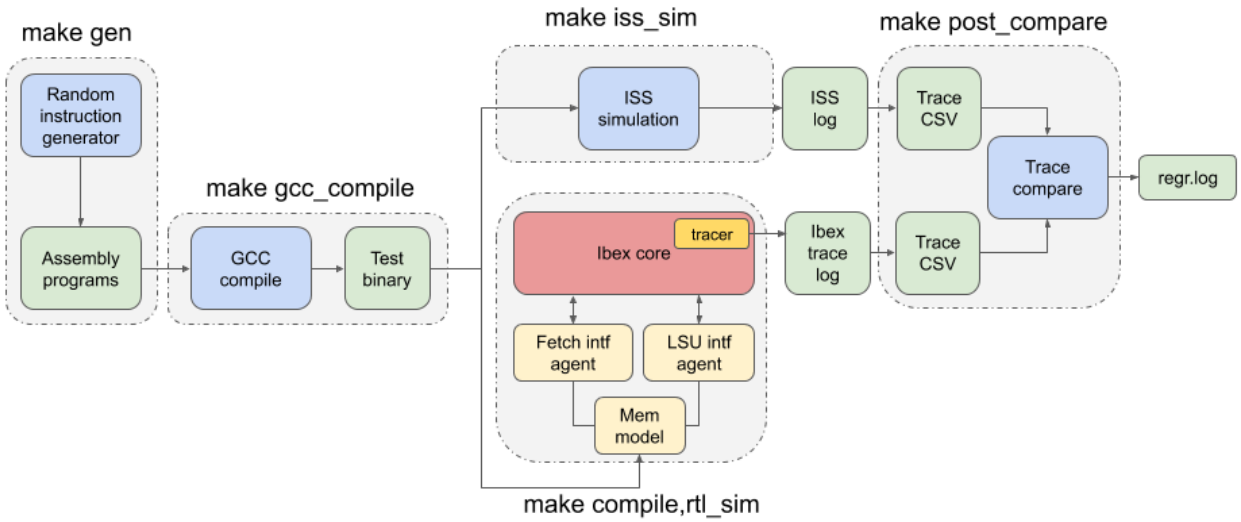


Figure 19.2: RTL/ISS co-simulation flow chart

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 handshaking 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/HANDSHAKE.md. As a sidenote, the signature address that this testbench uses for the handshaking is `0x8ffffffc`. 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

# Run a full regression
make
```

(continues on next page)

(continued from previous 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

# Pass addtional options to the generator
make GEN_OPTS="xxxx"  ...

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

# Use a different ISS (default is spike)
make ... ISS=ovpsim

# 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.

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

19.2 Instruction Cache

19.2.1 Overview

NOTE: Icache verification, as well as documentation, is still in very early stages.

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:

```
./vendor/lowrisc_ip/dvsim/dvsim.py dv/uvm/icache/dv/ibex_icache_sim_cfg.hjson --build-
↪only
--skip-ral --purge --sr sim_out
```

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.

CHAPTER

TWENTY

EXAMPLES

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

20.1 FPGA

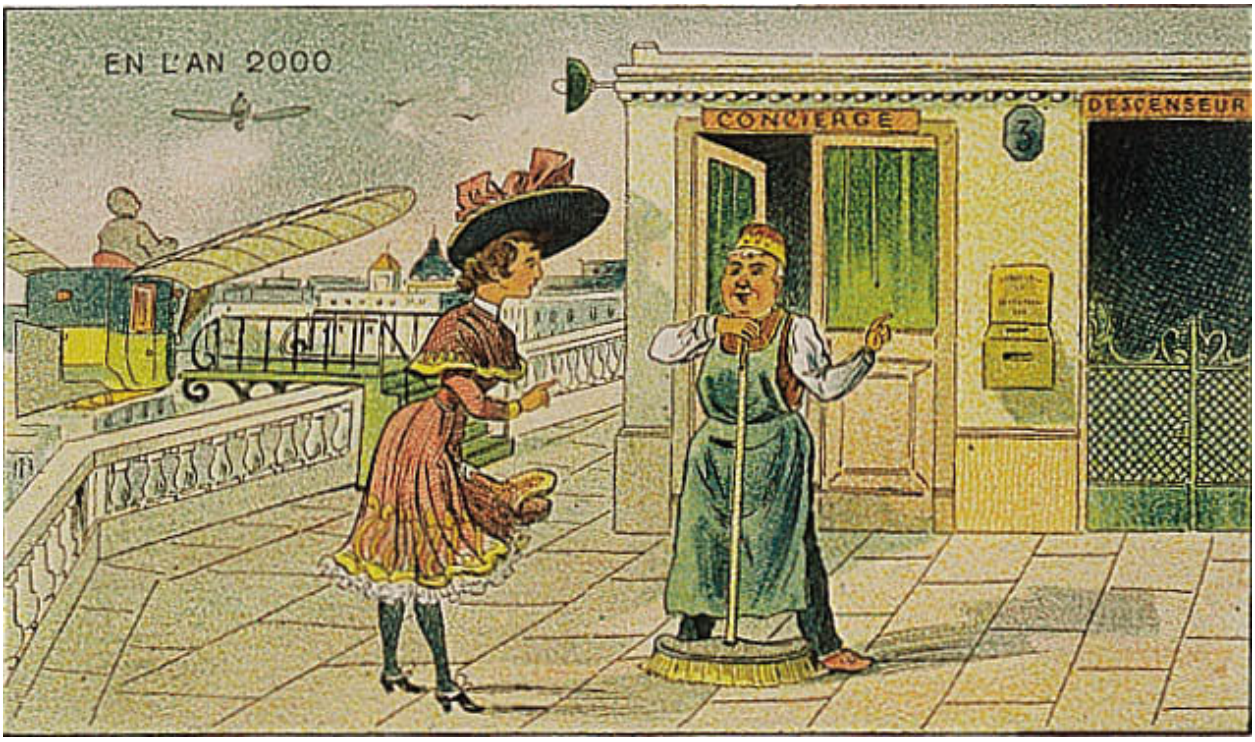
A minimal example for the Arty A7 FPGA Development board is provided. In this example Ibex is directly linked to a SRAM memory instance. Four LEDs from the board are connected to the data bus and are updated each time when a word is written. The memory is separated into a instruction and data section. 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`.

CHAPTER

TWENTYONE

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.

21.1 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!

21.2 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.

CHAPTER

TWENTYTWO

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.