Synogate Anti-Replay Core

IP User Guide - v1.0

Contents

Connections [#]	Window [bits]	Pipelining	Storage	Device	Fmax [MHz]	ALM [#]	FF [#]	M20K [#]
512	3968	moderate	on chip	Arria 10	270	1932	2997	132
				Agilex-F	470	2541	3497	108
2048	3584	moderate	on chip	Arria 10	270	2747	4475	521
				Agilex-F	470	4034	5380	425
16384	3584	moderate	external	Arria 10	220	3683	4451	129
				Agilex-F	460	4455	6357	128

Table 1: Resource consumption for common configurations.

1 Introduction

This IP-Core implements an Anti-Replay protection for use in IPSec network pipelines. The core is configurable wrt. the maximal number of connections, the window size, and the number of pipeline stages. In typical configurations, the core can operate at well over 200 MHz on Arria 10 devices with a throughput of one replay check per clock cycle.

The architecture was carefully chosen such that the main storage can be offloaded to QDR SRAM without impacting performance. Typical resource consumptions and timings can be seen in Table 1. Sample vhdl code generations have been tested with recent versions of Intel Quartus, Xilinx Vivado, and GHDL.

2 Functional Overview

2.1 Functionality

On a functional level, the Anti-Replay Core keeps track of a number of connections, each of which is independent from the others. For each connection, it checks whether a given packet, identified by its sequence number, has already been received. To this end, it stores the highest so far received sequence number as well as a limited history of lower sequence numbers that have already been received. This history is implemented as a window of fixed size that moves with the highest so far received sequence number. The sequence numbers have sufficient bits and typically are initiated to low values (e.g. 0) so that no wrap-around needs to be considered.

The windows are physically implemented in memory and thus have a fixed physical size which can be configured upon generation of the core. During runtime, the apparent window size can be altered by changing a *virtual* window size. If this virtual window size is set to a smaller value than the physical window size, the core simply behaves as if it had smaller windows. The virtual window size may, however, also be set to a larger value than the physical window size. In this case, sequence numbers that fall in between the physical and virtual window sizes are considered unseen. Figure 1 illustrates this case. Upon reset, the virtual window size defaults to the physical window size.

More specifically, for each replay check the core will test the sequence number and output a code that indicates whether that sequence number had already been received or not as well as why. Reasons for passing this test (being classified as *unseen*) are:

- The check for the connection is disabled.
- The sequence number is larger than any received so far.
- The sequence number falls into the physical window but has not yet been received.

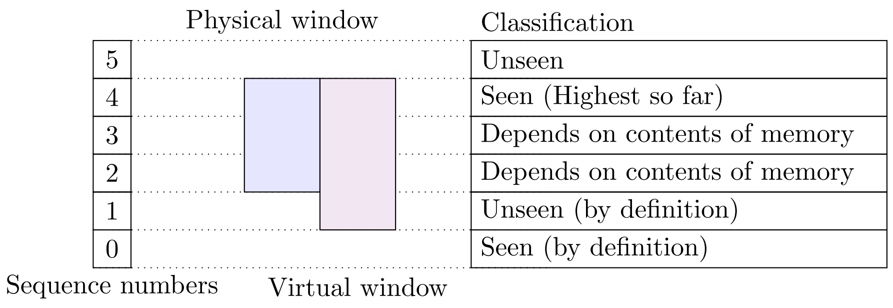

Figure 1: Physical and virtual windows in the Anti-Replay Core, and which areas are deemed seen or unseen. Note that the highest sequence number is always deemed part of the window.

- The sequence number falls outside the physical but still into the virtual window and is considered to be new by definition.

Reasons for failing this test (being classified as *seen*) are:

- The sequence number falls into the physical window but has been received before.
- The sequence number is older than the bottom end of the virtual window and is thus considered to have already been received.

In addition to returning the code, the core updates its internal state accordingly.

For runtime configuration and control, the core accepts commands to:

- Resize the virtual window.
- Activate or deactivate the checks for individual connections.
- Reset a connection and set its highest so far seen sequence number to a specific value.

Details of the commands and returned opcodes can be found in the Interface Documentation (see 3.7).

2.2 Structure

2.2.1 Anti-Replay Core Proper

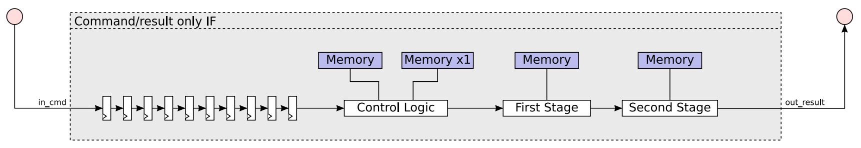

Figure 2: High level block diagram of the inner part of the Anti-Replay Core.

The inner part of the Anti-Replay Core is purely stream based (see Figure 2). An input stream of commands is ingested which can be any of the commands to runtime-configure the core or a command to process (check) a sequence number. All commands are processed in order at a rate of one command per clock cycle. For replay check commands, an output stream returns result codes that indicate the result of the operation with a latency that is fixed at runtime but depends on various configuration options of the core. The result codes are returned in the same order as the replay check commands.

The core needs memory to store its internal state, the largest of which is the storage for the physical windows, the bitmaps of which sequence numbers for each connection have already been seen. The latter is split into two parts, a small first stage and a full sized second stage. Since the second stage can be quite large for large window sizes and number of connections, it can be configured to not use internal memory (e.g. BRAMs) but expose a memory interface to which external fixed-latency memory, such as external sRAM, can be connected.

2.2.2 Packet IF

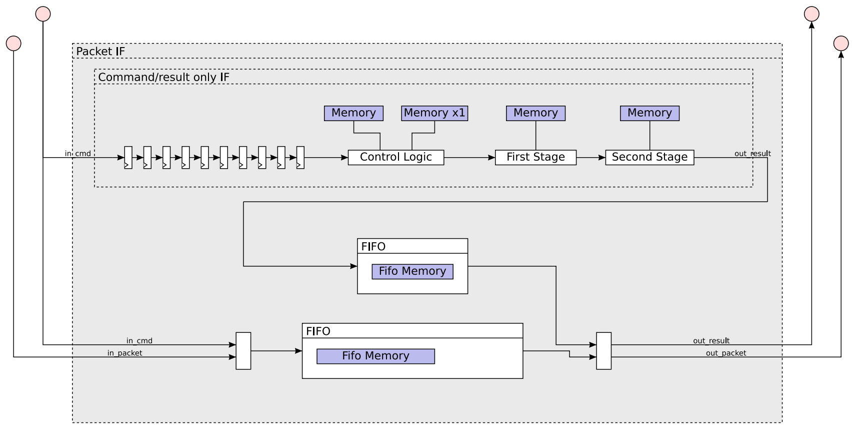

Figure 3: High level block diagram of the core in the packet IF configuration.

The “Packet IF” is a wrapper around the aforementioned Anti-Replay Core Proper that exposes the input stream of commands, but features additional input and output packet streams (see Figure 3). All packets are forwarded from the input to the output in order.

Requests (configuration commands or sequence number checks) can be issued during a frame/packet or outside a packet. A request is considered to be during a packet if it occurs on any cycle between start of packet and end of packet (including the SOP and EOP cycles). It is also considered to be during a packet, if it is transmitted during bubbles of a packet (between SOP and EOP but in cycles where the packet stream is not valid).

If a replay check request is issued during a packet, the packet stream is delayed with an internal fifo to bridge the latency of the Anti-Replay Core, such that the result of the request can be output at the latest with the last beat of the packet. More specifically, if a request is issued with the EOP of the input packet stream, its result is guaranteed to arrive together with the EOP of the same packet on the output stream.

If no request is issued during a packet on the command stream, the packet is forwarded with minimal delay, but in order with other packets. Details about the interface and example waveforms can be found in the Interface Documentation (see 3.7).

2.3 Reference Implementation

A reference implementation of the core’s functionality is included in the c++ source code in `source/tests/AntiReplayCoreModel.{h/cpp}` in the class `AntiReplayCoreModel`. Note that this is an implementation of the inner Anti-Replay Core Proper.

For brevity, only the most relevant parts are shown here. The structs `Cmd` and `Result` mirror the input and output signals and as described in the Interface Documentation (see 3.7) where a description of the commands and their bit representation can be found as well. Note that the reference implementation has no limit on the number of connections, while in the actual core a limit is imposed by the configuration options.

```
class AntiReplayCoreModel {
  public:
    struct Cmd {
      uint64_t opcode;
      uint64_t connectionIndex;
      uint64_t sequenceNumber;
      uint64_t windowSize;
    };

    struct Result {
      bool valid;
      uint64_t code;
    };

    Result execute(const Cmd& cmd);
  protected:
    struct SaState {
      bool active = false;
      std::set<uint64_t> seen;
    };

    uint64_t m_physicalWindowSize = 64;
    uint64_t m_virtualWindowSize = 128;

    std::map<uint64_t, SaState> m_state;

    uint64_t update(SaState& sa, uint64_t sequenceNumber);
};

AntiReplayCoreModel::Result AntiReplayCoreModel::execute(const Cmd& cmd)
{
  Result result = { .valid = false };
  switch (cmd.opcode)
  {
  case 0: // OpUpdate
    result.code = update(m_state[cmd.connectionIndex], cmd.sequenceNumber);
    result.valid = true;
    break;
  case 1: // OpDeactivate
    m_state[cmd.connectionIndex].active = false;
    break;
  case 5: // OpActivate
    m_state[cmd.connectionIndex].active = true;
    break;
  case 2: // OpSetSeqNr
    m_state[cmd.connectionIndex].seen.clear();
    m_state[cmd.connectionIndex].seen.insert(cmd.sequenceNumber);
    break;
  case 3: // OpSetWindowSize
    m_virtualWindowSize = cmd.windowSize;
    break;
  default:
    assert(!"unknown opcode");
  }
  return result;
}

uint64_t AntiReplayCoreModel::update(SaState& sa, uint64_t sequenceNumber)
{
  if (!sa.active)
  {
    return 3; // PassCheckDisabled
  }
  else if (sa.seen.empty())
  {
    assert(!"sa not initialized");
    return -1; // undefined
  }
  else if (uint64_t maxSeqNbr = *sa.seen.rbegin(); maxSeqNbr < sequenceNumber)
  {
    sa.seen.insert(sequenceNumber);
    return 1; // PassAboveWindow
  }
  else if (maxSeqNbr - sequenceNumber < std::min(m_physicalWindowSize,
     m_virtualWindowSize))
  {
    if (sa.seen.contains(sequenceNumber))
    {
      return 2; // FailReplay
    }
    else
    {
      sa.seen.insert(sequenceNumber);
      return 7; // PassInPhysicalWindow
    }
  }
  else if (maxSeqNbr - sequenceNumber < m_virtualWindowSize)
  {
    return 5; // PassInVirtualWindow
  }
  else
  {
    return 0; // FailBelowWindow
  }
}
```

3 Core Generation

3.1 Overview

This IP-Core is written in *Gatery*, a C++ based Hardware Construction Library that allows IP-Cores to be build in a highly flexible and customizable way. The purpose of this section is to describe common concepts of this approach and how to make use of them in your workflow. If you are already familiar with the structure of Synogate products, you can skip ahead to Section 4 where product specific options and parameters are discussed.

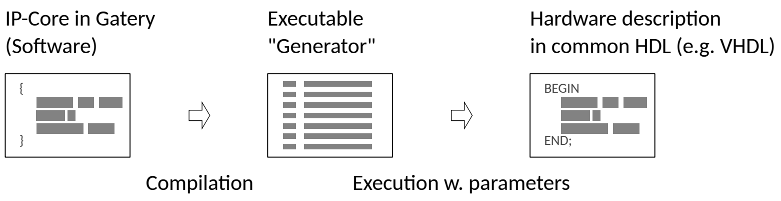

Figure 4: The three levels of the IP-Core. See text for details.

This documentation is bundled with:

- a full copy of the original source code (`source/` subfolder),
- precompiled binary executables called the generator (in the package's root folder),
- and sample exports of the IP-Core for a couple of the many possible parameter choices (`sample/` subfolder).

The generator arises from the compilation of the bundled C++ source code. Its purpose is to generate hardware description files for specific parametrizations of the IP-Core (see Figure 4).

Your license for this IP-Core states exactly what you are allowed to do with the code and generators and is the binding document in this regard. Usually, however, you are allowed to make modifications on any of those three levels: You may directly modify the generated outputs although it is not advisable. You may reinvoke the generator with different parameters if e.g. your requirements change. But you may also make modifications in the C++ source code if you require changes that can't be achieved with the exposed set of configuration options. Since invoking the generator with different parameters is the most common mode of customization, this document focuses on this aspect. Details on how to use *Gatery* can be found on the gatery website[1].

The generator is a command line program that is controlled through a human readable YAML configuration file. Sample configuration files can be found with the sample exports in the subfolders of `sample/` that can serve as a template for further modifications. The configuration file is passed to the generator as a command line argument:

```
# Generate core
./anti-replay-core custom_core_config.yaml
```

Usually (subject to the configuration file), the generator performs the following operations when generating an IP-Core:

1. Build internal representation of the IP-Core according to the given or default parameters.
2. Export the IP-Core as vhdl.
3. Run a top-level fuzzing testbench on the internal representation with an internal simulator.
4. Run a top-level demonstration testbench (for generating documentation waveforms) on the internal representation with an internal simulator.
5. Export the top-level fuzzing testbench to vhdl.
6. Export the top-level demonstration testbench to vhdl.
7. Export the Interface Documentation.
8. Export a vhdl package called `interface_package` that contains all relevant constants, codes, and parameters derived directly or indirectly from configuration options.

Section 4 discusses in detail the configuration file parameters that are unique (or of specific interest) to the Anti-Replay Core. The following sections instead focus on the more general parameters.

3.2 Configuration File Basics

The configuration file is a yaml file that controls aspects of the generation (such as which directory to export to and which steps to perform) as well as configuration options of the IP-Core. However, all configuration options are optional. If the generator is invoked with an empty file, it will use default values for all aspects of the generation. The chosen configuration options for the IP-Core (default or explicit) are listed in the Interface Documentation (if its generation wasn't disabled).

Whenever filesystem paths are specified in the configuration file (e.g. for where to export the vhdl code or Interface Documentation to), the paths allow injecting environment variables by bracketing them in `${ }`. For example, in a path of the form `${customEnvVar}/file.vhd`

[1] `https://www.synogate.com/gatery.html`

the first part would be replaced with the environment variable `customEnvVar`. In addition, `${BinaryPath}` is replaced by the path to the generator executable, `${ConfigPath}` by the path to the config file, and `${CurrentPath}` by the current working directory.

3.3 Controlling HDL Generation

By default, the generator exports the IP-Core as vhdl (2008). The exported code and all related files are written into the `vhdl/` subdirectory relative to the current working directory.

The defaults can be overriden with the following options (or subsets thereof):

```
vhdl:
  # If set to yes, will disable vhdl output
  disable: no
  # Path to export vhdl code to
  path: vhdl/
  # Path to export testbenches to
  testbench_path: vhdl_tb/
  # Library name to use in case import scripts or project files are generated
  library: AntiReplayCore
  # Whether and where to create an instantiation template
  instantiation_template_vhdl: vhdl/ip_core_inst.vhd
```

disable Disables the vhdl export. Defaults to `disable: no`.

path Where on the filesystem to export the vhdl code to. The `path` can point to a directory, as in above example, or to a file, e.g. `path: vhdl/AntiReplayCore.vhd`. In the latter case, all entities and packages are concatenated into a single vhdl file. Defaults to `path: vhdl/`.

testbench_path Where on the filesystem to export the testbenches and associated files to. If `testbench_path` is not explicitely given, the default is the same directory as `path`.

library If project files are generated, the default is to place all entites and packages into a library of a suitable name. This library name can be overriden with the `library` option.

instantiation_template_vhdl Where on the filesystem to place the VHDL instantiation template. Use No to disable generation of the VHDL instantiation template. If `instantiation_template_vhdl` is not explicitely given, the default is the same directory as `path`.

3.4 Targeting Synthesis Tools

The export can be attuned to various tools as described in the following sections. These may generate stand-alone project files, .tcl scripts for easy integration into existing project files, clock/constraint files, and so on. They may also alter the generated vhdl code by adapting attributes to the vendor specific style. The synthesis tool can be chosen with the `synthesis_tool` parameter.

The default is:

```
synthesis_tool: none
```

3.4.1 Quartus Integration

The export can be attuned to Intel Quartus by choosing `synthesis_tool: intel_quartus` with the following effects:

- Generation of a standalone project file with virtual pins suitable for resource consumption and timing analysis.
- Generation of a `.tcl` script for integration into existing Intel Quartus projects.
- Generation of Modelsim `.do` project files for the exported fuzzing and demonstration testbenches.
- Generation of an Intel Quartus compatible `.sdc` file for clocks for stand alone timing analysis.
- Generation of an Intel Quartus compatible `.sdc` file with constraints and path attributes.
- Adaptation of attributes inside the vhdl code to the Intel Quartus style.

3.4.2 Vivado Integration

The export can be attuned to Xilinx Vivado by choosing `synthesis_tool: xilinx_vivado` with the following effects:

- Generation of a `.tcl` script for integration into existing Xilinx Vivado projects.
- Generation of an Xilinx Vivado compatible `.xdc` file for clocks for stand alone timing analysis.
- Generation of an Xilinx Vivado compatible `.xdc` file with constraints and path attributes.
- Adaptation of attributes inside the vhdl code to the Xilinx Vivado style.

3.4.3 GHDL Simulation

The export can also export support files for simulation with ghdl by choosing `synthesis_tool: ghdl` with the following effects:

- Generation of a bash/shell script that processes all files and runs the fuzzing and demonstration testbenches.

3.5 Targeting Specific Technologies

While the choice of a synthesis tool affects how synthesis attributes are translated (which may be target device specific), the generator will not automatically target a specific technology or device family based on the synthesis tool. Such a target technology must be specified explicitely.

By default, the generator does not target any specific technology or fpga device. Instead, it produces plain vhdl code and makes reasonable assumptions about technology capabilities such as memory latencies.

To target a specifc fpga device, the vendor and device family must be specified:

```
target_technology:
  vendor: intel
  family: Arria 10
  device: 10AX115N2F40I1SG
```

If the `family` is optional and will be inferred if a `device` is specified. Vice versa, if no `device` is specified, a reasonable choice will be made from the `family`.

So far, only choices regarding memory pipelining and memory macro instantiations are based on the selected target technology. However, future versions may also finetune pipelining aspects based on speedgrades, so selecting the correct device may be beneficial.

For `vendor: intel`, the generator supports the families:

- `family: MAX 10`
- `family: Cyclone 10`
- `family: Stratix 10`
- `family: Arria 10`
- `family: Agilex`

For `vendor: xilinx`, the generator supports the families:

- `family: Kintex Ultrascale`
- `family: Virtex Ultrascale`

3.6 Controlling Tests and Waveform Export

The generator contains an internal simulator. It is used to simulate a small demonstration testbench but can also run a larger fuzzing test. The results of these simulator runs are used to create vhdl testbenches, but can also be exported as waveforms.

```
waveforms:
  disable: No
  path: vhdl/

fuzzing_test:
  num_batches: 1
  num_cycles_per_batch: 10000
  report_progress: true
```

The waveforms can be disabled and their output path can be changed, similarly to the vhdl output. The fuzzing test can be disabled by setting `num_batches: 0`. Otherwise it runs multiple batches of `num_cycles_per_batch` cycles each. Since simulation can take significant processing time, the generator can report its progress on the command line with `report_progress: true`.

Keep in mind that for large fuzzing tests, the waveform files and the testbench stimuli files can grow to significant file sizes.

3.7 Interface and Signal Descriptions

Changing the configuration of the IP Core can have significant effects on the interface of the IP-Core. Signal widths, latencies, but also the registers of memory mapped interfaces can change. To prevent confusion, the generator produces an *Interface Documentation* for the specific choice of IP-Core settings. By default, the generator exports this documentation into the `doc/` subdirectory relative to the current working directory. It can be disabled with the `disable` parameter. The location of the documentation output can be changed with the `doc` parameter. The target directory is created if it doesn't exist.

```
doc:
  disable: no
  path: doc/
```

The generated Interface Documentation is html based and can be opened in any common browser through the entry file `doc/index.html`. Detailed descriptions of the interface signals, block diagrams, register tables (if applicable), and signal waveforms can be found there.

4 Parameterizing the Anti-Replay Core

The following describes configuration settings that are unique to the Anti-Replay Core or of specific interest. The Anti-Replay Core is distributed with three sample configuration files. It is recommended to use these as the basis for further modifications.

4.1 Clock and Reset

The core only has a single clock that can configured as follows:

```
clocks:
  clock:
    period: 100 MHz
    reset_type: synchronous
    reset_active: high
    initialize_registers: Yes
    initialize_memories: Yes
```

period The intended speed of the clock. It can be specified as a period in `s`, `ms`, `us`, `ns`, or `ps`. It can also be specified as a frequency in `Hz`, `KHz`, `MHz`, `GHz`, or `THz`. The units are not case sensitive. This only affects the generated clock .xdc file and the timescale of waveforms in the Interface Documentation but is expected to affect e.g. pipelining decisions in future versions. The setting defaults to `period: 100 MHz`.

reset_type The reset mode to implement for the registers. Can be `synchronous` for clock synchronous resets or `none`. The setting `synchronous` also triggers the generation of initialization logic for memories that need to be reset to specific values. In this case, the required duration of the reset can be found in the Interface Documentation and in the vhdl Interface Package. If the core is configured with `reset_type: none`, registers must be initialized by default values (see below). The reset type defaults to `reset_type: synchronous`.

reset_active Can be `high` or `low` depending on whether the reset is active high or active low (negated). The default is `reset_active: high`.

initialize_registers Whether to initialized registers with their reset values as default values in vhdl. This results in the registers already having their reset value after power-on on fpga devices. This setting must be `initialize_registers: Yes` if `reset_type: none`. The default is `initialize_registers: Yes`.

initialize_memories Whether to initialized memories with their reset values as default values in vhdl. This results in the memories already having their reset value after power-on on fpga devices. This setting only affects RAMs, not ROMs. The default is `initialize_memories: Yes`.

4.2 Packet IF Wrapper

The Packet IF wrapper mates the stream based Anti-Replay Core to an Avalon stream of potential network packets. The width of the Avalon stream as well as its auxiliary signals can be configured:

```
instance:
  /:
    frame_stream:
      data_width: 16
      dataBitsPerSymbol: 8
      # Remove setting to remove channel signals
      channel_width: 1
      # Remove setting to remove error signals
      error_width: 1
      # Remove setting to remove userdata signals
      userdata_width: 0
```

data_width Controls the width of the `data` and, by extension, `empty` signals. Defaults to `data_width: 16`.

dataBitsPerSymbol Indirectly controls the width of the `empty` signal. Defaults to `dataBitsPerSymbol: 8`.

channel_width The width of the `channel` signal in bits. Can be zero, in which case the `channel` signal is present in the port map, but with a zero bit width. Defaults to the `channel` signal not being present in the port map.

error_width The width of the `error` signal in bits. Can be zero, in which case the `error` signal is present in the port map, but with a zero bit width. Defaults to the `error` signal not being present in the port map.

userdata_width The width of the `userdata` signal in bits. Can be zero, in which case the `userdata` signal is present in the port map, but with a zero bit width. Defaults to the `userdata` signal not being present in the port map.

4.3 Anti-Replay Core Proper

The inner Anti-Replay Core can be configured wrt. to its limits (windows sizes, connections) as well as memory and pipelining tradeoffs:

```
instance:
  /:
    connection_addr_width: 9
    virtual_window_width: 17
    sequence_number_width: 64

  AntiReplay0:
    window_addr_width: 12
    memory_word_width: 128
    pipelining: moderate
```

connection_addr_width Controls the number of supported connections, which is 2^c with c being `connection_addr_width`. The required memory of the core scales roughly linear with the number of supported connections (2^c). Defaults to `connection_addr_width: 9`.

virtual_window_width Controls the maximal virtual window size, which is 2^v with v being `virtual_window_width`. This should be at least `window_addr_width` and is often the same. Note that the highest sequence number received is always considered part of the windows (physical and virtual). Defaults to `window_addr_width: 17`.

sequence_number_width Sets the sequence number range from 0 to $2^s - 1$ with s being `sequence_number_width`. For large number of connections (large `connection_addr_width`), this setting has some influence on required internal memory but is usually dwarfed by the second stage memory. However, it is often dictated by the application scenario. Defaults to `sequence_number_width: 64`.

window_addr_width Controls the size of the physical window. The total physical window size is $2^n - m$ with n being `window_addr_width` and m being `memory_word_width`. The required memory of the core scales roughly linear with physical window size. Note that the highest sequence number received is always considered part of the windows (physical and virtual). Defaults to `window_addr_width: 12`.

memory_word_width Width of the second stage memory data interface. The memory requirements of the first stage memory scale reciprocally with `memory_word_width`. If the second stage memory is external, a wider interface will thus reduce the on-chip memory requirement. For configurations without external memory, this parameter should be tuned so that the available sram block address bits are not exceeded. Defaults to `memory_word_width: 128`.

pipelining Gives corse control over how many pipeline stages are used to trade latency and area with clock rates. The setting supports three levels:

- `pipelining: minimum` places the minimal amount of registers only, which is registers for memories and registers for the output of the core.
- `pipelining: moderate` places extra registers for good clock rates.
- `pipelining: aggressive` splits large comparators and multiplexers for extra register stages. Useful for high clock rates, but significantly increases the core's latency.

Increasing the amount of pipelining improves Fmax, but also increases the register consumption and, because of the core's latency, increases the fifo sizes. For the actual number of used pipeline stages refer to the generated Interface Documentation or the generated interface package.

The Generator allows further, more fine grained customization of memory and logic pipelining with which it can be pushed to very high frequencies. If the core hits a timing limit, contact Synogate with the critical path that failed and we will try to propose further configuration settings for the specific problem.

4.4 Second Stage Memory

The second stage memory is the storage for the physical windows, the bitmaps of which sequence numbers for each connection have already been seen. It is typically the largest memory of the core and consumes significant resources. While potentially all memories can be configured to be external memories, the second stage memory is the best choice if memory is to be offloaded.

To turn the second stage memory into external memory, use the following template:

```
instance:
  SecondLookupStage0/scl_memory0:
    type: EXTERNAL
    readLatency: 5
    prefix: bitmap
```

Specifically, the generator will not use internal memory resources to represent the second stage memory, but will create signals in the top level entity of the core to which an appropriate memory must be connected. For the second stage memory, the external memory must be a simple dual port memory with fixed (but configurable) read latency. If chosen, details of the memory signals can be found in the generated Interface Documentation.

type Controls the type of the memory. Choosing `type: EXTERNAL` turns the memory into external memory.

readLatency Specifies the read latency of the external memory which must be at least 1 cycle. The generator will automatically build the necessary bypass logic to bridge the read latency and may instantiate small on-chip ring buffer memories for this purpose.

prefix The prefix to use for the memory signal names in the top level entity port map. For example, choosing `prefix: bitmap` will result in signal names of the form `bitmap_rd_addr`, `bitmap_wr_addr`, etc. For details about the signals, see the generated Interface Documentation.