**Overview**

This document specifies the Cryptographically Secure Random Number Generator (CSRNG) hardware IP functionality. Due to the importance of secure random number generation (RNG), it is a topic which is extensively covered in security standards. This IP targets compliance with both [BSI's AIS31 recommendations for Common Criteria (CC)](https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_31_Functionality_classes_for_random_number_generators_e.pdf), as well as [NIST's SP 800-90A](https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf) and [NIST's SP 800-90C (Second Draft)](https://csrc.nist.gov/CSRC/media/Publications/sp/800-90c/draft/documents/sp800_90c_second_draft.pdf), both of which are referenced in [FIPS 140-3](https://csrc.nist.gov/publications/detail/fips/140/3/final). The CSRNG IP supports both of these standards for both deterministic (DRNG) and true random number generation (TRNG). In NIST terms, it works with the [Entropy Source IP](https://github.com/lowRISC/opentitan/blob/master/hw/ip/entropy_src/README.md) to satisfy the requirements as a deterministic random bit generator (DRBG) or non-deterministic random bit generator (NRBG). In AIS31 language, this same implementation can be used to satisfy either the DRG.3 requirements for deterministic generation, or the PTG.3 requirements for cryptographically processed physical generation.

In this document the terms "DRNG" and "TRNG" are used most generally to refer to deterministic or true random number generation functionalities implemented to this specification. However, the terms "DRBG" or "NRBG" are specifically used when respectively referring to SP 800-90A or SP 800-90C requirements. Meanwhile, when addressing requirements which originate from AIS31 we refer to the specific DRG.3 or PTG.3 classes of RNGs.

This IP block is attached to the chip interconnect bus as a peripheral module conforming to the [comportability definition and specification](https://github.com/lowRISC/opentitan/blob/master/doc/contributing/hw/comportability/README.md), but also has direct hardware links to other IPs for secure and software-inaccessible transmission of random numbers. The bus connections to peripheral modules are done using the CSRNG application interface. This interface allows peripherals to manage CSRNG instances, and request the CSRNG module to return obfuscated entropy.

**Features**

* Provides support for both deterministic (DRNG) and true random number generation (TRNG), when combined with a secure entropy source (i.e. one constructed and implemented in compliance with SP 800-90A,B,C and AIS31). The TRNG mode is provided directly by the entropy source.
* Compliant with NIST and BSI recommendations for random number generation.
* Hardware peripherals and software applications issue commands to dedicated RNG instances through a common application interface.
* In deterministic mode, meets the requirements given in AIS31 for a DRG.3 class deterministic random number generator (DRNG) meaning it provides Forward Secrecy and Enhanced Backward Secrecy.
* Utilizes the CTR\_DRBG construction specified in NIST SP 800-90A, qualifying it as a NIST-approved deterministic random bit generator (DRBG).
  + Operates at 256 bit security strength.
* Support for multiple separate CSRNG instances per IP block.
  + Each instance has its own internal state, control, reseed counters and IO pins.
  + The number of CSRNG instances is set via a module parameter.
* Software access to a dedicated CSRNG instance.
  + One instance, Instance N-1, is always accessible from the bus through device registers,
  + All other instances route to other hardware peripherals (e.g. the key manager, obfuscation engines, etc.) and in normal operation these other instances are inaccessible from software.
  + The IP may be configured to support "debug mode" wherein all instances can be accessed by software. For security reasons this mode may be permanently disabled using one-time programmable (OTP) memory.
* The IP interfaces with external entropy sources to obtain any required non-deterministic seed material (entropy) and nonces.
  + Requires an external entropy source which is compliant with NIST SP 800-90B, and which also satisfies the requirements for a PTG.2 class physical non-deterministic random number generator as defined in AIS31.
  + Dedicated hardware interface with external entropy source satisfies requirements for get\_entropy\_input() interface as defined in SP 800-90A.
  + This block does not use a derivation function and requires full entropy from the entropy source.
* Also supports the optional use of personalization strings or other application inputs (e.g. OTP memory values) during instantiation.
* Assuming a continuously-live entropy source, each instance can also optionally be used as a non-deterministic TRNG (true random number generator, also called a non-deterministic random bit generator or NRBG in SP 800-90C).
  + In this mode, an instance also meets the requirements laid out for a PTG.3 class RNG, the strongest class laid out in AIS31.
  + Implementation follows the NRBG "Oversampling Construction" approved by SP 800-90C, to meet both [Common Criteria (CC, ISO/IEC 15408)](https://www.iso.org/standard/50341.html) and FIPS TRNG constructions.
* In addition to the approved DRNG mode, any instance can also operate in "Fully Deterministic mode", meaning the seed depends entirely on application inputs or personalization strings.
  + This provides an approved means of seed construction in FIPS 140-2 as described in the [FIPS 140-2 Implementation Guidance](https://csrc.nist.gov/csrc/media/projects/cryptographic-module-validation-program/documents/fips140-2/fips1402ig.pdf), section 7.14, resolution point 2(a).

**Description**

Though the recommendations in AIS31 are based around broad functional requirements, the recommendations in SP 800-90 are very prescriptive in nature, outlining the exact constructs needed for approval. Thus the interface and implementation are largely driven by these explicit constructs, particularly the CTR\_DRBG construct.

The CSRNG IP consists of four main components:

1. An AES primitive
2. The CTR\_DRBG state-machine (ctr\_drbg\_fsm) which drives the AES primitive, performing the various encryption sequences prescribed for approved DRBGs in SP 800-90A. These include:
   1. **The Derivation Function:** Part of the instantiation and reseed routines, this routine assembles the previous seed material (on reseed only), application inputs, and entropy.
   2. **The Instantiation Routine:** Combines application inputs, external entropy and nonce (more entropy) via the derivation function.
   3. **The Reseed Routine:** Combines the previous seed material with external entropy to generate a new seed.
   4. **The Generate Routine:** Generates up to CSRNG\_MAX\_GENERATE random bits. If called with prediction\_resistance\_flag, forces a reseed.
   5. **The Update Routine:** Updates the internal state of the DRNG instance after each generate call.
3. State vectors for each DRNG instance.
4. Interface logic and access control for each instance.

**Note on the term "Entropy"**

Every DRNG requires some initial seed material, and the requirements for the generation of that seed material varies greatly between standards, and potentially between Common Criteria security targets. In all standards considered, DRNGs require some "entropy" from an external source to create the initial seed. However, the rules for obtaining said entropy differ. Furthermore the required delivery mechanisms differ. For this reason we must make a clear distinction between "Physical" (or "Live" or "True") entropy and "Factory Entropy". This distinction is most important when considering the creation of IP which is both compatible with both the relatively new SP 800-90 recommendations, as well as the well-established FIPS 140-2 guidelines.

* Physical entropy is the only type of "entropy" described in SP 800-90. The means of generation is described in [SP 800-90B](https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90B.pdf). One statistical test requirement is that physical entropy must be unique between reboot cycles, ruling out sources such as one-time programmable (OTP) memories. In SP 800-90A, the delivery mechanism must come through a dedicated interface and "not be provided by the consuming application".
* Factory entropy is a type of entropy described in the [FIPS 140-2 implementation guidance (IG)](https://csrc.nist.gov/csrc/media/projects/cryptographic-module-validation-program/documents/fips140-2/fips1402ig.pdf) section 7.14, resolution point 2(a). It can be stored in a persistent memory, programmed at the factory. In some use cases, the consuming application needs to explicitly load this entropy itself and process it to establish the expected seed.

This document aims to make the distinction between physical entropy and factory entropy wherever possible. However, if used unqualified, the term "entropy" should be understood to refer to physical entropy strings which are obtained in accordance with SP 800-90C. That is either physical entropy, or the output of a DRNG which itself has been seeded (and possibly reseeded) with physical entropy. In AIS31 terms, "entropy strings" (when used in this document without a qualifier) should be understood to come from either a PTG.2 or PTG.3 class RNG.

**Security**

All module assets and countermeasures performed by hardware are listed in the hjson countermeasures section. Labels for each instance of asset and countermeasure are located throughout the RTL source code.

The bus integrity checking for genbits is different for software and hardware. Only the application interface software port will have a hardware check on the genbits data bus. This is done to make sure repeated values are not occurring. Only 64 bits (out of 128 bits) are checked, since this is statistically significant, and more checking would cost more silicon. The application interface hardware port will not have this check. It is expected that the requesting block (EDN) will do an additional hardware check on the genbits data bus.

**Compatibility**

This block is compatible with NIST's SP 800-90A and BSI's AIS31 recommendations for Common Criteria.

**Theory of Operation**

The CSRNG block has been constructed to follow the NIST recommendation for a DRBG mechanism based on block ciphers. Specifically, it is a CTR\_DRBG that uses an approved block cipher algorithm in counter mode. As such, the block diagram below makes reference to hardware blocks that either directly or closely follow NIST descriptions for the equivalent functions.

There are two major hardware interfaces: the application interface and the entropy request interface. The application interface, which is described in more detail later, is provided for an application to manage an instance in CSRNG. Once setup, the application interface user can request for entropy bits to be generated, as well as other functions. The application interface supports up to 15 hardware interfaces, and one software interface.

A walk through of how CSRNG generates entropy bits begins with the application interface. An instantiate command is issued from one of the application interfaces. This request moves into the cmd\_stage block. Here the request is arbitrated between all of the cmd\_stage blocks. The winner will get its command moved into the command dispatch logic. A common state machine will process all application interface commands in order of arbitration. At this point, some seed entropy may be required depending on the command and any flags. If needed, a request to the entropy source hardware interface will be made. This step can take milliseconds if seed entropy is not immediately available. Once all of the prerequisites have been collected, a CTR\_DRBG command can be launched. This command will go into the ctr\_drbg\_cmd block. This ctr\_drbg\_cmd block uses two NIST-defined functions, the update and the block\_encrypt functions. If the command is a generate, the ctr\_drbg\_cmd block will process the first half of the algorithm, and then pass it on to the ctr\_drbg\_gen block. Additionally, the ctr\_drbg\_gen block also uses the update block and the block\_encrypt block. To keep resources to a minimum, both of these blocks have arbiters to allow sharing between the ctr\_drbg\_cmd and ctr\_drbg\_gen blocks. The command field called ccmd (for current command) is sent along the pipeline to not only identify the command, but is also reused as a routing tag for the arbiters to use when returning the block response.

Once the command has traversed through all of the CTR\_DRBG blocks, the result will eventually land into the state\_db block. This block will hold the instance state for each application interface. The specific state information held in the instance is documented below. If the command was a generate command, the genbits data word will be returned to the requesting cmd\_stage block. Finally, an ack response and status will be returned to the application interface once the command has been completely processed.

**Design Details**

**Non-blocking Commands**

Regarding command processing, all commands process immediately except for the generate command. The command generate length count (glen) is kept in the cmd\_stage block. When the state\_db block issues an ack to the cmd\_stage block, the cmd\_stage block increments an internal counter. This process repeats until the glen field value has been matched. Because each request is pipelined, requests from other cmd\_stage blocks can be processed before the original generate command is completely done. This provides some interleaving of commands since a generate command can be programmed to take a very long time.

When sending an unsupported or illegal command, CMD\_STAGE\_INVALID\_ACMD\_ALERT will be triggered. Furthermore, CSRNG will respond with the appropriate error status response on the relevant interface.

**AES Cipher**

The block\_encrypt block is where the aes\_cipher\_core block is located. This is the same block used in the AES design. Parameters are selected such that this is the unmasked version.

**Software Support**

The software application interface uses a set of TL-UL registers to send commands and receive generated bits. Since the registers are 32-bit words wide, some sequencing will need to be done by firmware to make this interface work properly.

**Application Interface**

This section describes the application interface, which is required for performing any operations using a CSRNG instance (i.e. instantiation, reseeding, RNG generation, or uninstantiation). Each CSRNG instance corresponds to a unique application interface port, which implements the application interface described here. Any hardware peripherals which require complete control of an instance may connect directly to a dedicated interface port. Meanwhile peripherals without any special requirements (i.e. personalization strings or non-FIPS-approved, fully-deterministic number sequences) may share access to an instance via the entropy distribution network (EDN) IP. The EDNs manage the instantiation and reseeding of CSRNG instances for general use-cases, providing either on-demand or timed-delivery entropy streams to hardware peripherals. Firmware applications can obtain access to random bit sequences directly through application interface port 0, which is directly mapped to a set of TL-UL registers.

The total number of application interface ports (for TL-UL, directly attached peripherals or EDN instances) is determined by the NHwApp parameter.

The command bus operates like a FIFO, in which a command is pushed into the interface. An optional stream of additional data may follow, such as seed material for an instantiate application command. For the generate application command, the obfuscated entropy will be returned on the genbits bus. This bus also operates like a FIFO, and the receiving module can provide back pressure to the genbits bus. There is one instance of a firmware application interface, and it uses the TL-UL registers. For more details on how the application interface works, see the Theory of Operations section above.

In general, users of the application interface are either firmware or some hardware module entity. For hardware, a module can either directly control the application interface, or it can connect to an EDN module. Attaching to an EDN module allows for a simpler interface connection to a more layout-friendly distributed-chip network.

**General Command Format**

The general format for the application interface is a 32-bit command header, optionally followed by additional data, such as a personalization string, typically twelve 32-bit words in length. Depending on the command, these strings are typically required to be 384-bits in length, to match the size of the seed-length when operating with 256-bit security-strength. The exact function of the additional data field depends in the command. However, in general, the additional data can be any length as specified by the command length field. The command header is defined below.

**Command Response**

Once a command has been completed, successfully or unsuccessfully, the CSRNG responds with a single cycle pulse on the csrng\_rsp\_ack signal associated with the same application interface port. If the command is successful, the csrng\_rsp\_sts signal will indicate the value 0 (SUCCESS) in the same cycle. Otherwise the application will receive an error value on the csrng\_rsp\_sts signal as described in [SW\_CMD\_STS.CMD\_STS](https://github.com/lowRISC/opentitan/blob/master/hw/ip/csrng/doc/registers.md#sw_cmd_sts). A number of exception cases to be considered are enumerated in NIST SP 800-90A, and may include events such as:

* Failure of the entropy source
* Attempts to use an instance which has not been properly instantiated, or
* Attempts to generate data when an instance has exceeded its maximum seed life. In such cases, a 32-bit exception message will be propagated to firmware via the hw\_exc\_sts register, and a cs\_hw\_inst\_exc interrupt will be raised.

**Generated Bits (genbits) Interface**

In addition to the command response signals there is a bus for returning the generated bits. This 129-bit bus consists of 128-bits, genbits\_bus, for the random bit sequence itself, along with a single bit flag, genbits\_fips, indicating whether the bits were considered fully in accordance with FIPS/CC standards.

There are two cases when the sequence will not be FIPS compliant:

* Early in the boot sequence, the ENTROPY\_SRC generates a seed from the first 384 bits pulled from the noise source. This initial seed is tested to ensure some minimum quality for obfuscation use- cases, but this boot seed is not expected to be full-entropy nor do these health checks meet the 1024-bit requirement for start-up health checks required by NIST 800-90B.
* If flag0 is asserted during instantiation, the resulting DRBG instance will have a fully deterministic seed. If this seed is determined only by user input data, it may lack the physical-entropy required by NIST SP 800-90A, and thus this DRBG instance may not be FIPS/CC compliant. If the seed has been derived by firmware in a FIPS/CC complaint way, the DRBG instance can still be FIPS/CC compliant. To this end, CSRNG allows forcing the FIPS flag using the [CTRL.FIPS\_FORCE\_ENABLE](https://github.com/lowRISC/opentitan/blob/master/hw/ip/csrng/doc/registers.md#ctrl) field and the [FIPS\_FORCE](https://github.com/lowRISC/opentitan/blob/master/hw/ip/csrng/doc/registers.md#fips_force) register. This feature may also be useful for known-answer testing through entropy consumers accepting FIPS/CC compliant entropy only.

**Handshaking signals**

The application command signal csrng\_req\_bus is accompanied by a csrng\_valid\_signal, which is asserted by the requester when the command is valid. CSRNG may stall incoming commands by de-asserting the csrng\_req\_ready signal. A command is considered received whenever both csrng\_req\_valid and csrng\_req\_ready are asserted in the same clock cycle.

Likewise a requester must only consider data on the genbits bus to be valid when the genbits\_valid signal is asserted, and should assert genbits\_ready whenever it is ready to accept the genbits data. The genbits data is considered successfully transmitted whenever genbits\_valid and genbits\_ready are asserted in the same clock cycle.

**Interrupts**

The cs\_cmd\_req\_done interrupt will assert when a CSRNG command has been completed.

The cs\_entropy\_req interrupt will assert when CSRNG requests entropy from ENTROPY\_SRC.

The cs\_hw\_inst\_exc interrupt will assert when any of the hardware-controlled CSRNG instances encounters an exception while executing a command, either due to errors on the command sequencing, or an exception within the ENTROPY\_SRC IP.

The cs\_fatal\_err interrupt will assert when any of the CSRNG FIFOs has a malfunction. The conditions that cause this to happen are either when there is a push to a full FIFO or a pull from an empty FIFO.