AES – Advanced Encryption Standard:

The Advanced Encryption Standard (AES) is a widely adopted symmetric block‐cipher algorithm. It operates on fixed 128-bit blocks of data and supports three key lengths—128, 192, or 256 bits—which correspond to 10, 12, or 14 rounds of processing, respectively. Each round applies a series of transformations—SubBytes (a non-linear byte substitution using an S-box), ShiftRows (a transposition step that cyclically shifts each row of the state), MixColumns (a linear mixing operation over each column), and AddRoundKey (an XOR with a portion of the expanded key)—to thoroughly diffuse and obscure the plaintext. The advantages of AES is a combination of strong security, high performance in both hardware and software, and straightforward, regular structure, making it the backbone of modern secure communications, disk encryption, and countless cryptographic protocols.

GF – Galois Field:

Each byte in the encryption process is treated as an element in the finite field GF . The byte is observed in a polynomial representation, such as:

, bᵢ ∈ {0,1}

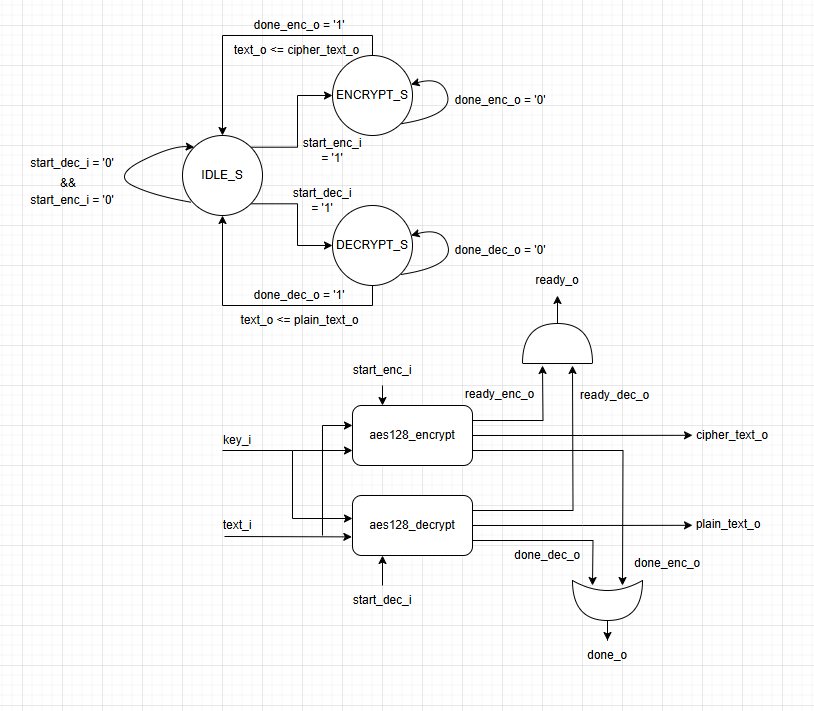
Over this field 2 operation are relevant to the encryption process, addition and multiplication. Addition is a bitwise XOR. Multiplication is multiplying 2 bytes then using the modulo operation using the AES irreducible polynomial:  
 = 0x1B

Practically, Multiplication is implemented in hardware using a “Multiply by 2” base block, which involves hardware cheap operation: shift and XOR. AES uses this finite field to gain diffusion and non-linearity.

Modules:

aes128\_core:

The aes128\_core module serves as the top-level controller for AES-128 cryptographic operations, integrating both encryption and decryption capabilities into a single cohesive unit. It receives a 128-bit input (text\_i) along with a 128-bit key (key\_i) and delegates processing to either the aes128\_encrypt or aes128\_decrypt submodule, depending on which start signal is asserted. Internally, the module uses a simple FSM to toggle between idle, encryption, and decryption states. Once an operation completes, the corresponding output (enc\_text\_o or dec\_text\_o) is routed to text\_o, and a done\_o pulse is generated. The ready\_o signal ensures that no operation starts unless both encryption and decryption units are idle. This modular architecture allows for clean integration into systems requiring runtime AES mode selection, while maintaining clear separation of encryption and decryption logic under a unified interface.



aes128\_encrypt:

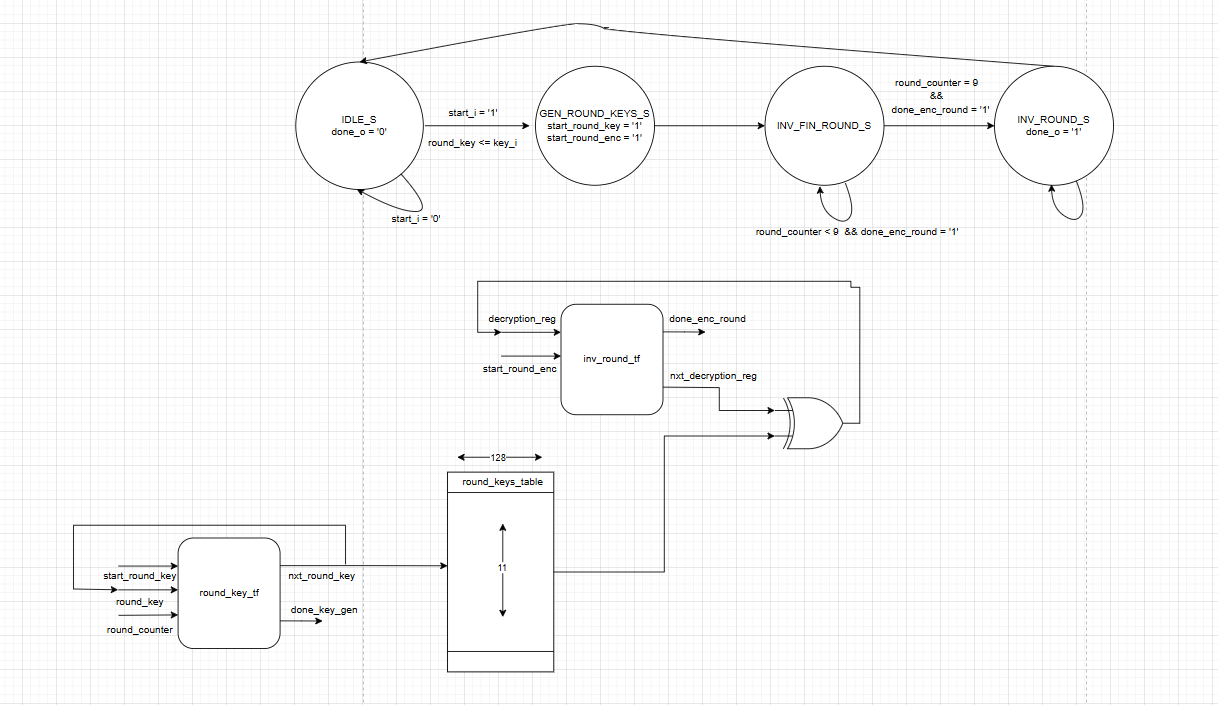
The aes128\_encrypt module orchestrates the full AES-128 encryption process, handling the sequencing and control of each encryption round. It accepts a 128-bit plaintext and a 128-bit secret key and outputs the corresponding 128-bit ciphertext. Internally, the design is driven by a finite state machine (FSM) that transitions through four states: IDLE, INIT, ROUND, and FIN, corresponding to the stages of AES encryption. Initially, the plaintext is XORed with the first-round key (AddRoundKey). For rounds 1 to 9, the module invokes two submodules: round\_tf (handling SubBytes, ShiftRows, and MixColumns) and round\_key\_tf (generating the next round key). The final round omits the MixColumns step as per the AES standard. Data and key registers are updated synchronously, with precise control of start and done signals to manage pipeline timing. This module ensures full compliance with the AES-128 specification while maintaining clean control logic and extensibility for integration into higher-level systems.

A diagram of a computer

AI-generated content may be incorrect.

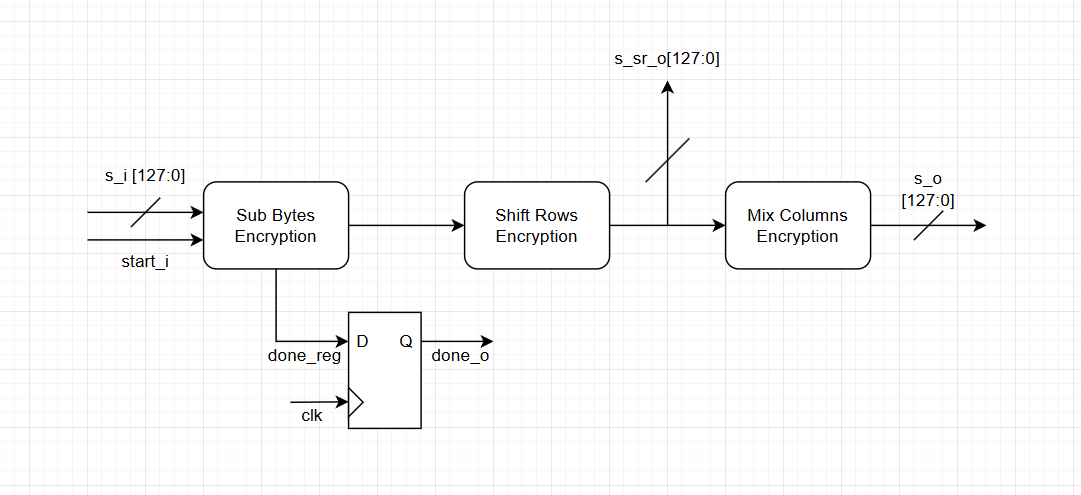
Aes128\_decrypt:

The aes128\_decrypt module implements the inverse AES-128 algorithm to recover plaintext from a 128-bit ciphertext using the original encryption key. It begins by generating all 11 round keys (from round 0 to 10) using the same key schedule logic as the encryption path. These keys are stored in a local register file, allowing direct access in reverse order for decryption. The decryption process then follows the AES standard in reverse: the final round is handled first (applying InvShiftRows, InvSubBytes, and AddRoundKey, but bypassing InvMixColumns), followed by rounds 9 to 1 which include the full inverse transformation including InvMixColumns. The last round (r=0) completes the inverse operation, yielding the original plaintext. Control is handled via a finite state machine with states to manage key expansion, round execution, and result delivery. Key design choices such as precomputing round keys and selectively bypassing MixColumns ensure accurate and efficient hardware decryption flow, fully compliant with the AES specification.



Round\_tf:

The round\_tf module performs a single standard AES-128 encryption round, applying the three core transformations: SubBytes, ShiftRows, and MixColumns. It receives a 128-bit input state (s\_i) and produces two outputs: the intermediate state after the ShiftRows operation (s\_sr\_o) and the final state after MixColumns (s\_o). Internally, the module first uses the sub\_bytes unit (parameterized for encryption) to substitute each byte via the AES S-box. The result is passed through the shift\_rows module, which cyclically shifts the rows of the state matrix to the left. Finally, the mix\_columns transformation mixes each column using Galois Field arithmetic. A done\_o signal is pipelined from the S-box unit to signal completion, ensuring alignment with clock timing. This modular structure facilitates clean round-level reuse in both encryption and decryption contexts.



Round\_key\_tf:

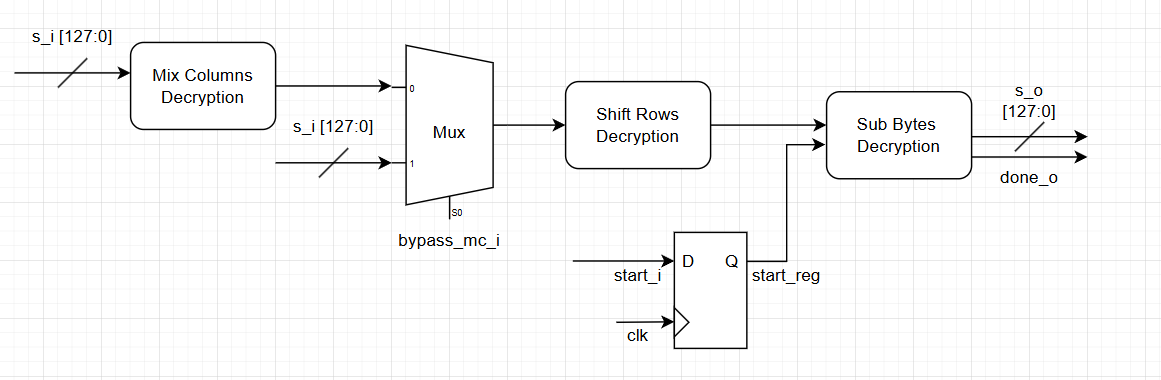
The round\_key\_tf module performs the key schedule operation for AES-128, generating the next round key from the current one during encryption or decryption. It accepts a 128-bit input key (key\_i) and a 4-bit round index (round\_count\_i), producing the corresponding next 128-bit round key (key\_o). The module follows the AES key expansion algorithm: it rotates the last word (w[3]), applies an S-box substitution to the rotated word, and then XORs it with the first word and the round constant (RCON). The remaining words are computed as chained XORs of previous results. To ensure synchronization with surrounding logic, the S-box substitution is handled in a pipelined manner with a done\_o flag signaling completion. It forms a key component of both encryption and decryption flows by enabling dynamic round key generation on a per-cycle basis.

A diagram of a computer program

AI-generated content may be incorrect.

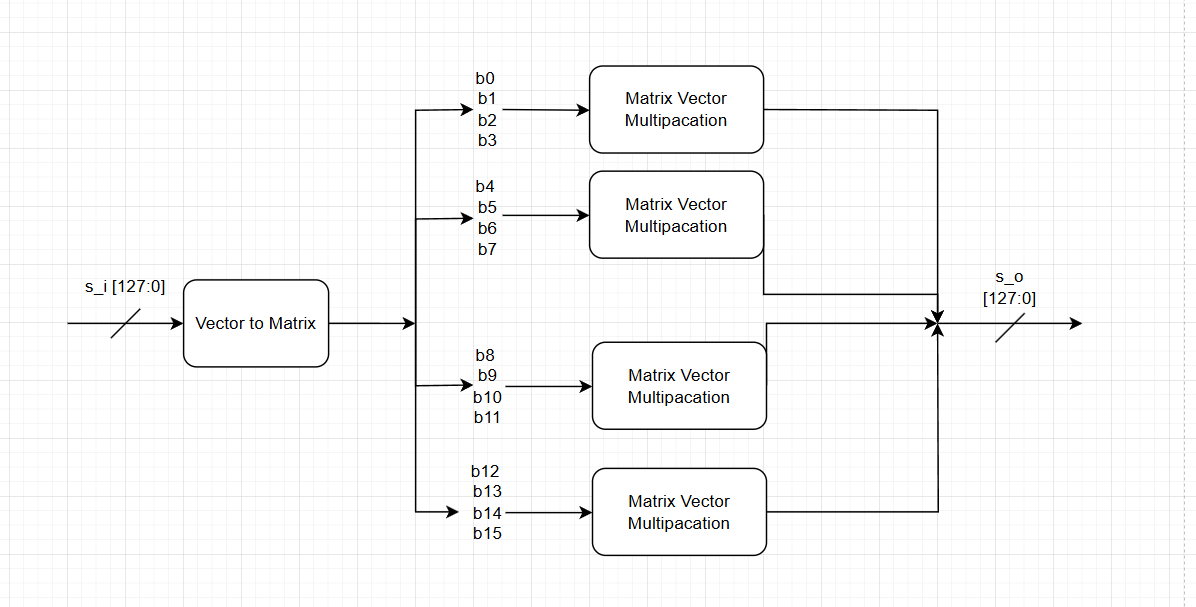
Inv\_round\_tf:

The inv\_round\_tf module implements one full decryption round of the AES-128 algorithm, applying the inverse transformations of MixColumns, ShiftRows, and SubBytes. It accepts a 128-bit state (s\_i) and produces the transformed output (s\_o) along with a done\_o flag to signal completion. A control input bypass\_mc\_i enables skipping the inverse MixColumns step, which is required during the final AES round. If bypass\_mc\_i is asserted, the state bypasses the inverse MixColumns unit and proceeds directly to inverse ShiftRows. The shift\_rows and sub\_bytes components are both configured for decryption (OP=0), ensuring correct application of inverse permutations and substitutions. The S-box operation is pipelined using a start\_reg to align timing with start\_i. This modular structure is critical for the decryption flow in AES-128, where each round applies the inverse transformations in reverse order compared to encryption.



Mix\_columns:

The mix\_columns module implements the MixColumns operation of AES, responsible for inter-column diffusion in the 4×4 byte state matrix. It is parameterized by OP, where OP=1 enables encryption mode and OP=0 enables decryption mode. Internally, the 128-bit input state (s\_i) is first unpacked into an array of 16 bytes. Each of the four 4-byte columns is then processed independently using Galois Field multiplications. In encryption mode, each output byte is calculated using the fixed matrix {2,3,1,1}, while in decryption mode, it uses the inverse matrix {14,11,13,9}. The appropriate finite field multiplications (gfmul2, gfmul3, gfmul9, etc.) are imported from an external function package. The resulting bytes are repacked into the 128-bit output state (s\_o). This module ensures accurate column mixing as required by the AES specification and supports both forward and inverse transformations through a single unified implementation.

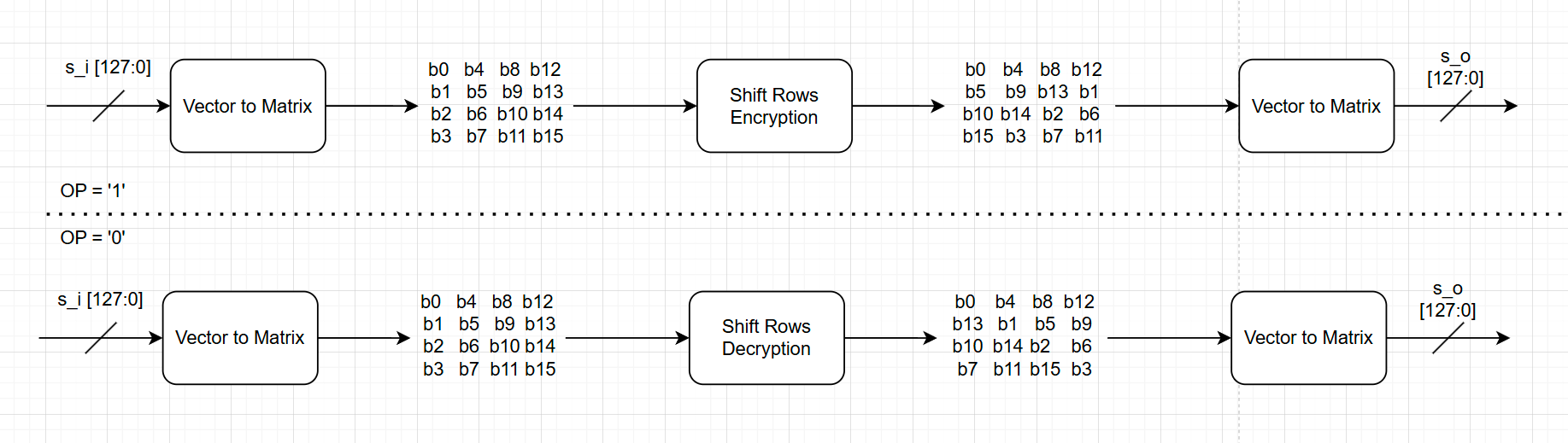


A diagram of a computer program

AI-generated content may be incorrect.

Shift\_rows:

The shift\_rows module performs the row-wise permutation step of AES, which cyclically shifts the bytes within each row of the 4×4 state matrix. This operation enhances diffusion by disrupting the byte alignment across columns. The module is parameterized by OP, where OP=1 applies the standard **left shift** used during encryption, and OP=0 applies the **right shift** used in decryption. The 128-bit input (s\_i) is first unpacked into 16 individual bytes, preserving their matrix order. Depending on the mode, the module rearranges these bytes according to the AES specification: encryption shifts rows by offsets {0, 1, 2, 3}, while decryption shifts them in reverse {0, -1, -2, -3}. The transformed bytes are then packed back into the output state (s\_o). This module is purely combinational and efficiently implements a critical structural transformation in AES processing.



Sub\_bytes:

The sub\_bytes module implements the SubBytes transformation of AES, a non-linear substitution that replaces each byte of the input state using a substitution box (S-box). Parameterized by WIDTH (typically 128 bits) and OP (operation mode), it supports both encryption (OP=1, using the forward S-box) and decryption (OP=0, using the inverse S-box). The S-box values are loaded from external memory files (sbox\_table.mem or inv\_sbox\_table.mem) into a 256-entry ROM. Internally, the module uses a simple FSM with states IDLE, RUN, and DONE, processing one byte per clock cycle in the RUN state. Each byte of the input (s\_i) is substituted and stored into a temporary result register (temp\_res), which is then output as s\_o once all bytes are processed. The done\_o signal marks the end of substitution. This design prioritizes simplicity and timing compatibility with pipelined systems, while still allowing optional substitution with a combinational implementation if desired.

A diagram of a flowchart

AI-generated content may be incorrect.

Aes\_pack\_functions:

The hea\_func\_pack package defines a suite of utility functions for performing finite field (GF(2⁸)) multiplications, which are fundamental to the AES MixColumns and InvMixColumns transformations. It provides optimized implementations for multiplying a byte by the constants used in AES: {2,3,4,8,9,11,13,14}. These functions exploit the properties of GF(2⁸) arithmetic, using bitwise shifts and conditional XORs with the AES irreducible polynomial (0x1B) to efficiently compute the results. For instance, gfmul2 performs a left shift with conditional reduction, while gfmul3 is a combination of gfmul2 and the original byte. Composite multipliers like gfmul14 chain together basic operations to reduce redundant logic. By centralizing these operations, the package promotes code reuse, modularity, and readability across all AES-related modules requiring Galois field arithmetic.

A diagram of a graph

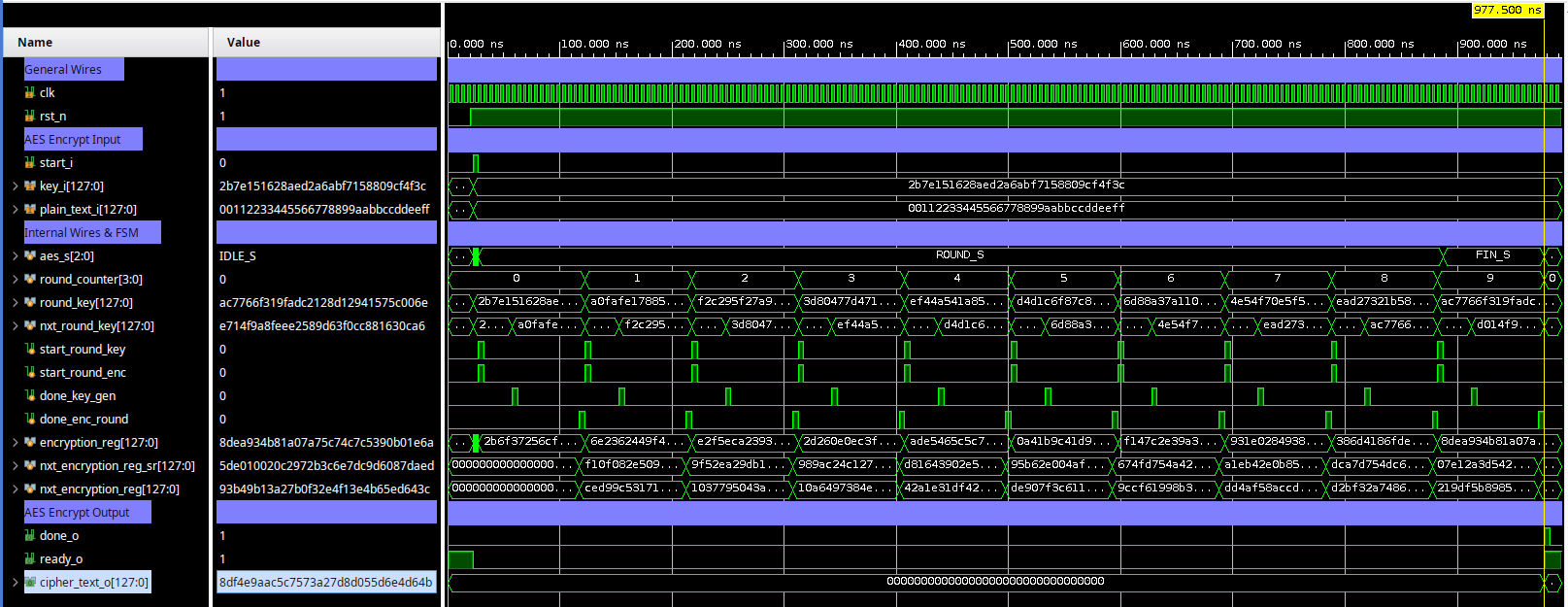
AI-generated content may be incorrect.

A diagram of a graph

AI-generated content may be incorrect.

Simulation examples:

Aes128\_encrypt test bench:



This waveform captures the full AES-128 encryption process as simulated in the testbench. The input plaintext (plain\_text\_i) is 00112233445566778899aabbccddeeff, and the key (key\_i) is 2b7e151628aed2a6abf7158809cf4f3c, both consistent with the AES FIPS-197 test vector.

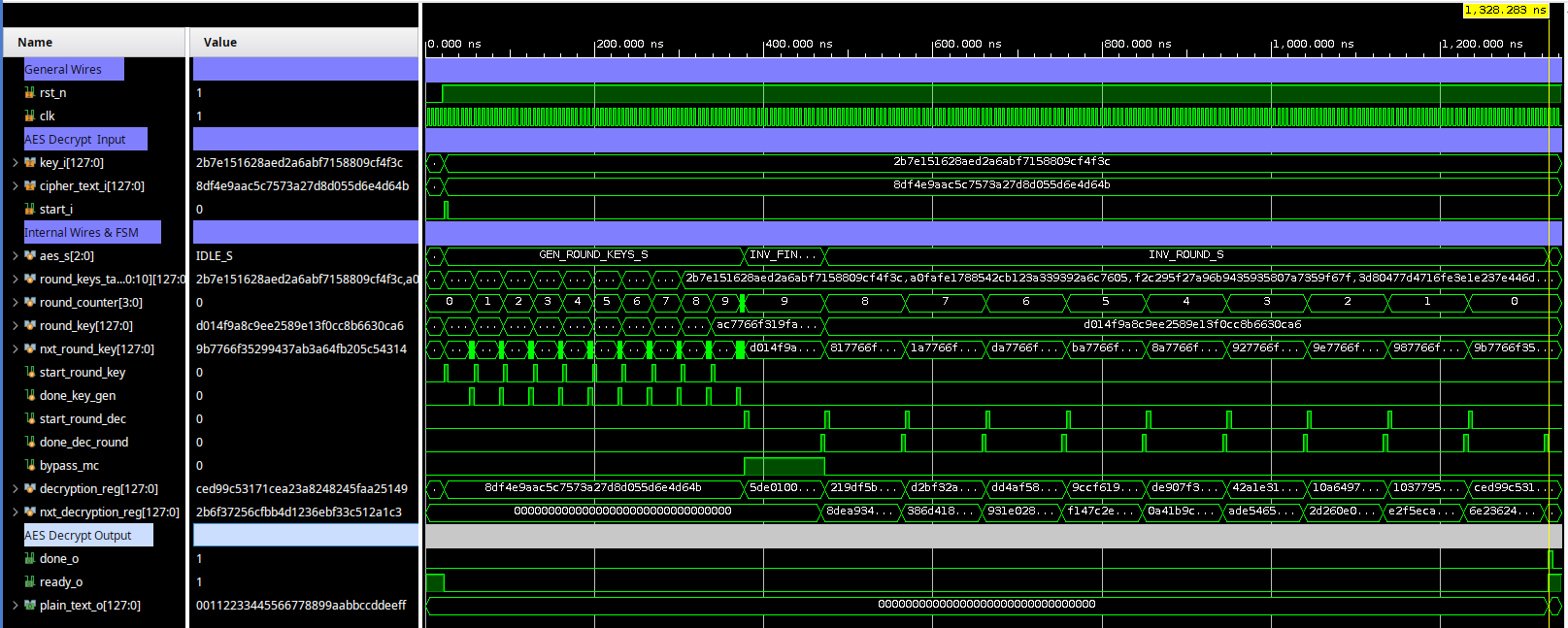
The simulation begins with a reset (rst\_n = 0 → 1) and triggers encryption via start\_i. The FSM (aes\_s) transitions from IDLE\_S to INIT\_S, then into ROUND\_S as it progresses through the 10 AES rounds. Each round involves a new round\_key, generated and shown per cycle, and a new internal state held in encryption\_reg.

* **round\_counter** tracks the current AES round (0–9).
* **start\_round\_enc** and **start\_round\_key** pulse each time a new encryption or key schedule step starts.
* **done\_enc\_round** and **done\_key\_gen** indicate completion of round logic and key generation respectively.

During the FIN\_S state, the final round executes (without MixColumns), and the ciphertext output (cipher\_text\_o) stabilizes at 8d4e9aac5c7573a27d8d055d6e4d064b, matching the expected AES result. The done\_o and ready\_o signals assert, indicating completion and readiness for the next operation.

This simulation confirms functional correctness of the AES encryption core, including key expansion, round processing, and output matching reference vectors.

Aes128\_decrypt test bench:



This waveform captures the simulation of the AES-128 **decryption process**, verifying that the ciphertext is correctly decrypted back to the original plaintext. The ciphertext input (cipher\_text\_i) is 8df4e9aac5c7573a27d8d055d6e4d064b, and the key (key\_i) is 2b7e151628aed2a6abf7158809cf4f3c, both matching the FIPS-197 test vector for AES-128.

The module begins in the IDLE\_S state. Once start\_i is asserted, the FSM enters GEN\_ROUND\_KEYS\_S, where round keys are generated sequentially (0 through 10), stored in round\_keys\_table, and indexed by round\_counter. Once key generation is complete, the FSM transitions to INV\_FIN\_ROUND\_S for the final round (which skips InvMixColumns using bypass\_mc = 1), followed by the remaining 9 inverse rounds in INV\_ROUND\_S.

* **decryption\_reg** and **nxt\_decryption\_reg** show the internal state transformation over time.
* **start\_round\_dec** and **done\_dec\_round** control the execution of each decryption round.
* At the end of the process, the decrypted output (plain\_text\_o) matches the original input message:  
  00112233445566778899aabbccddeeff

The handshake signals done\_o = 1 and ready\_o = 1 indicate successful decryption and readiness for the next operation. This confirms that the AES decryption logic, including key schedule and inverse round transformations, is functioning correctly.