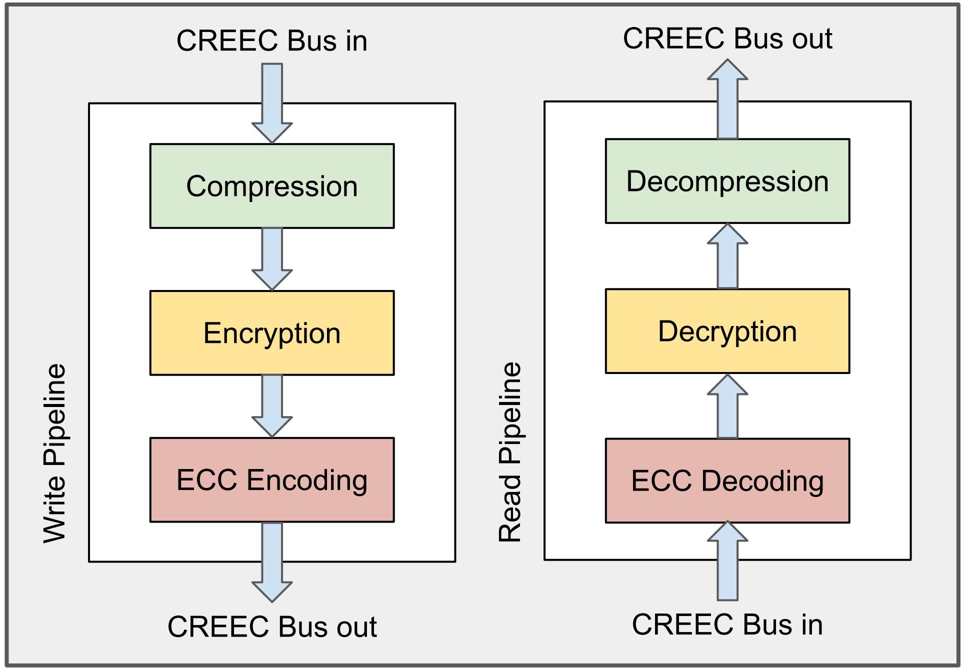
CREECelerator

Composable Hardware Generators for Disaggregated Storage Systems

Bob Zhou, Kyle Kovacs, Vighnesh Iyer, Tan Nguyen
EE290C Fall 2018
Dec 5, 2018
Final Presentation

What We Built

- Composable accelerators for storage in a disaggregated datacenter
- Compression
  - Increases SSD endurance
- Encryption
  - Ensures security of stored data
- ECC
  - Protects data against corruption
- Interconnect
  - Ties everything together with common abstractions and interfaces



Interconnect

- Minimal system bus that can bridge to existing testchipip BlockDevice, or a generic stream bus (AXI4-Stream), or any memory mapped bus (TL)
  - In hindsight, we should have just used AXI4-Stream and a modified architecture

```
case class BusParams(maxBeats: Int, maxInFlight: Int, dataWidth: Int)

class Header(val p: BusParams) extends Bundle {
  val len = UInt()
  val id = UInt()
}
class Data(val p: BusParams) extends Bundle {
  val data = UInt()
  val id = UInt()
}
class CREECBus(val p: BusParams) extends Bundle {
  val header = Decoupled(Header(p))
  val data = Decoupled(Data(p))
}
```

Compression

Problem: SSDs degrade over time

- Endurance is based on number of read/write cycles

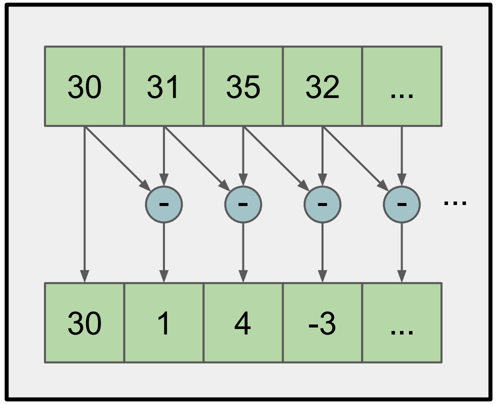
Goal: increase SSD lifetime by decreasing total physical I/O cycles

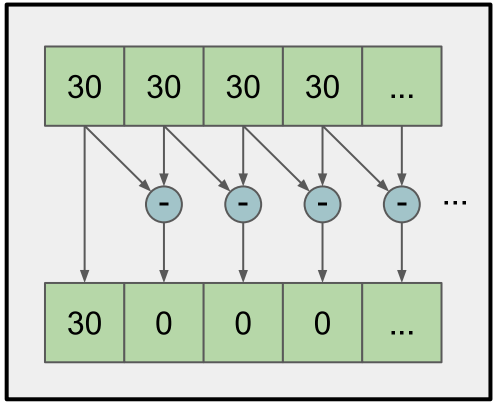
Method: Compress data before it hits the disk

- For the purposes of this project, a simple compression algorithm was used
    - Differential + run-length encoding

Differential Encoding

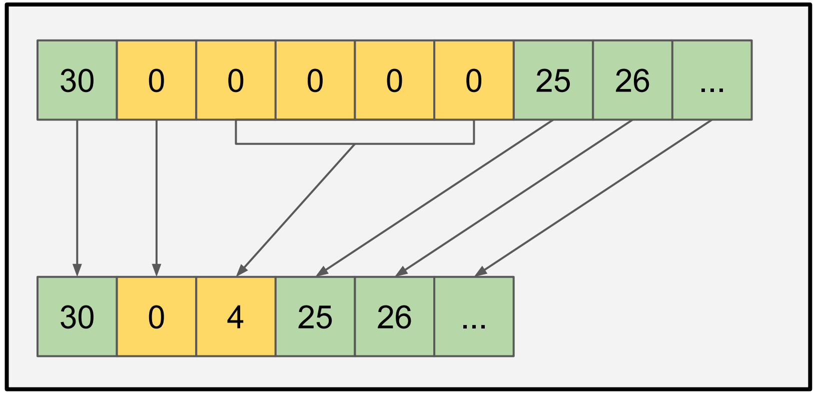
- Byte sequences are transformed into [initial value, differences]
- Reversible transformation
- Prepares the data for run-length encoding
- Converts any sequence of repeated bytes to a sequence of zeros





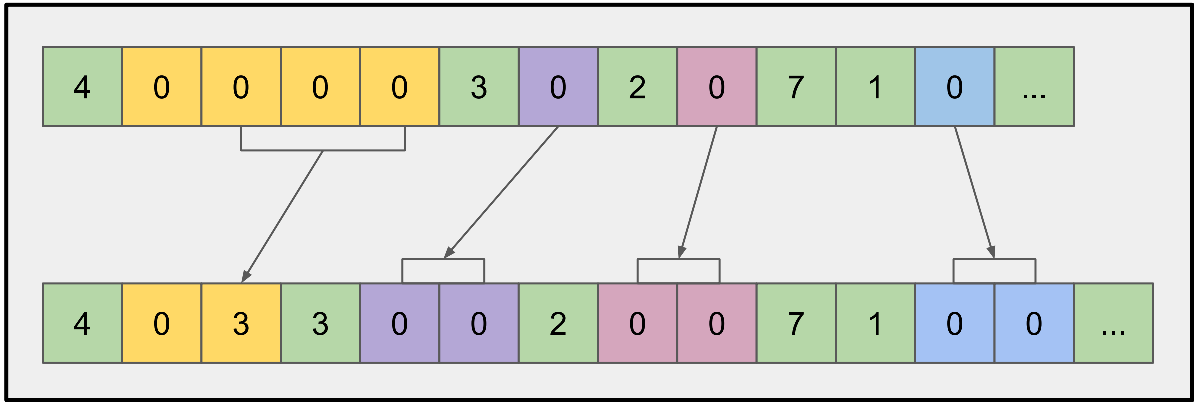
Run-Length Encoding

- “Runs” of zeros are transformed into [0, # additional zeros]
- Reversible transformation
- After differential, compresses runs of up to 256 repeated values into just 3 bytes



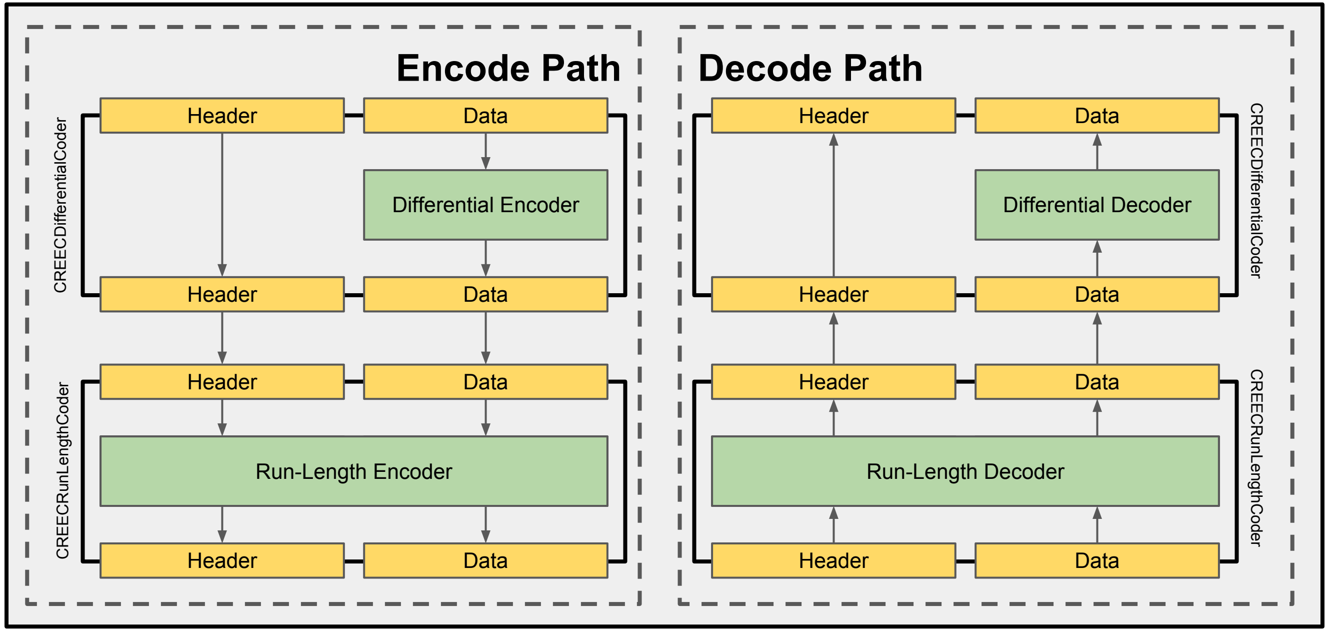
Problem With Run-Length Encoding

- A problem occurs when you have a single zero
  - It expands to 2 bytes
- In practice, this only works well for a specific type of data
  - Data like ASCII text, JPEG images, or other files do not compress well with this method



System Integration

- To interface with the system bus, the block must accept headers and data
- In the case of compression, the output header can only be sent after the data is compressed and the final length is determined



ECC: Reed-Solomon code

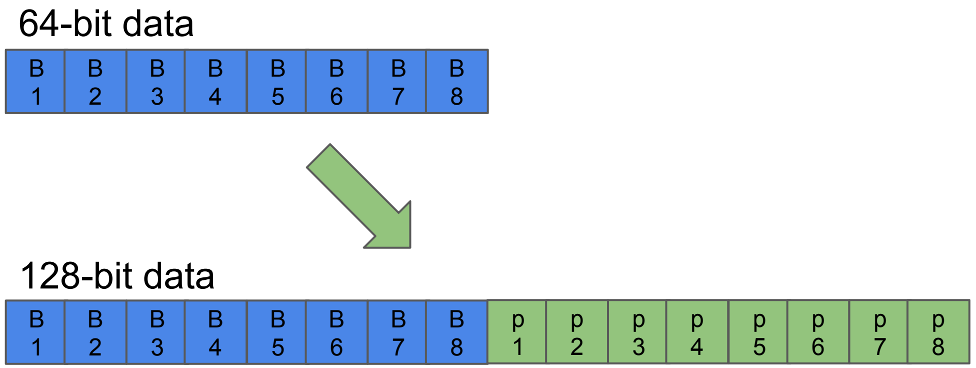
RS(n, k, b) Generator

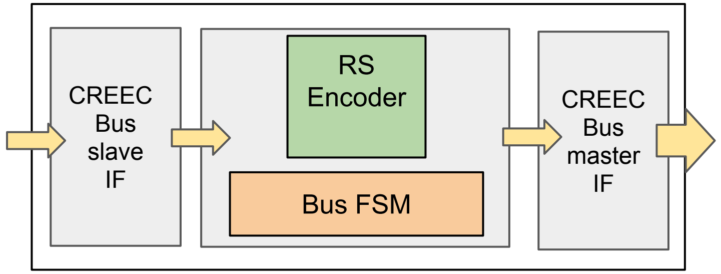
- n: number of output symbols
- k: number of input symbols
- b: bitwidth of a symbol

Note: **block-level** parity scheme!

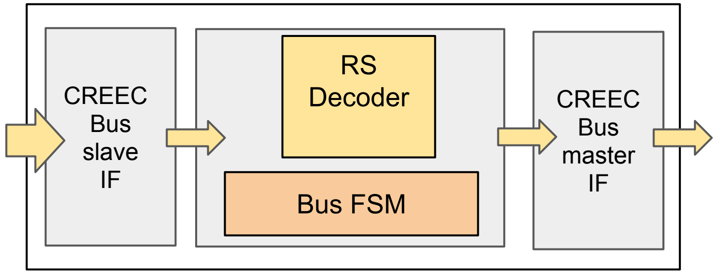
Capacity: correct up to (n-k)/2 symbols

Example: RS(16, 8, 8)



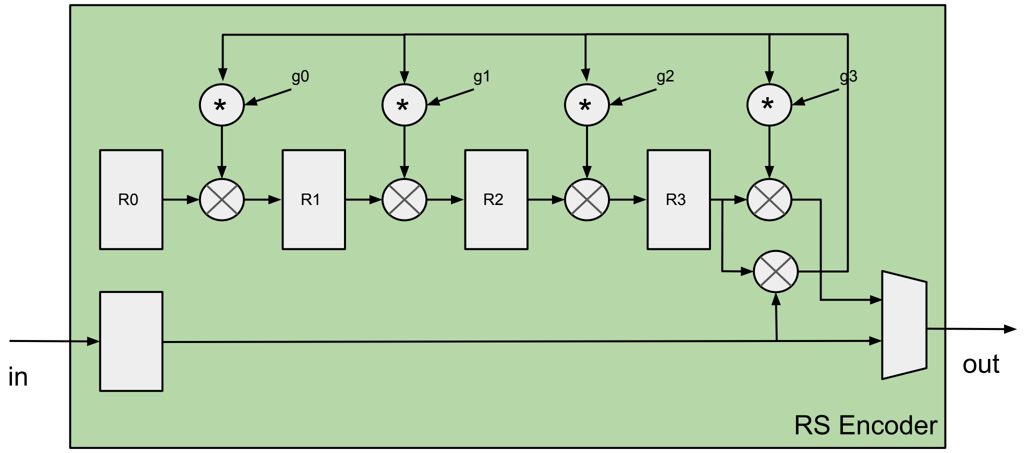


CREEC RS Encoder block



CREEC RS Decoder block

RS Encoder



in_1, in_2, …, in_k

in_1, in_2, … in_k,
p_1, p_2, …, p_(n-k)

Linear-Feedback Shift-Register

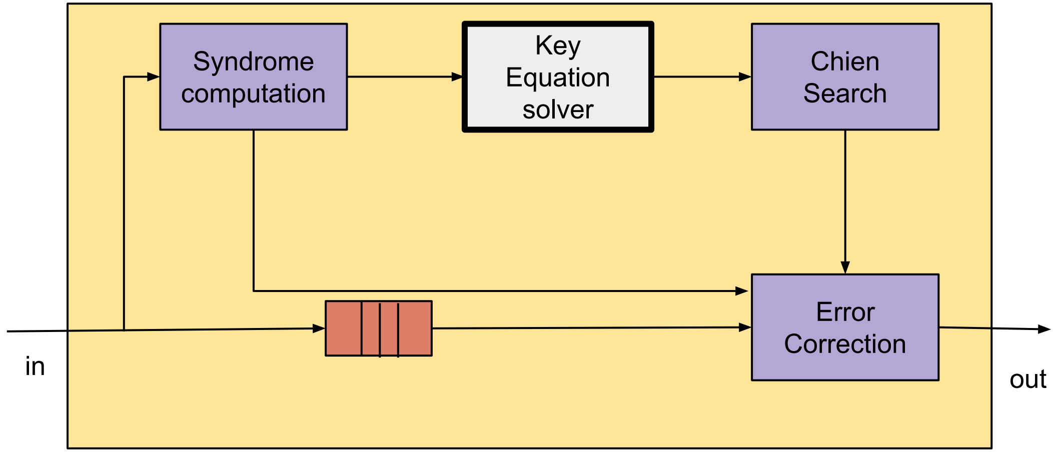
#Registers == #parity symbols

Arithmetic operations are performed in Galois Finite Field

- Addition: XOR operation
- Multiplication: XOR + bitmask + shift operations

No carry generated!

RS Decoder



in_1, in_2, …, in_n

in_1, in_2, … in_k

(drop the parity symbols)

> 20x expensive than Encoder!

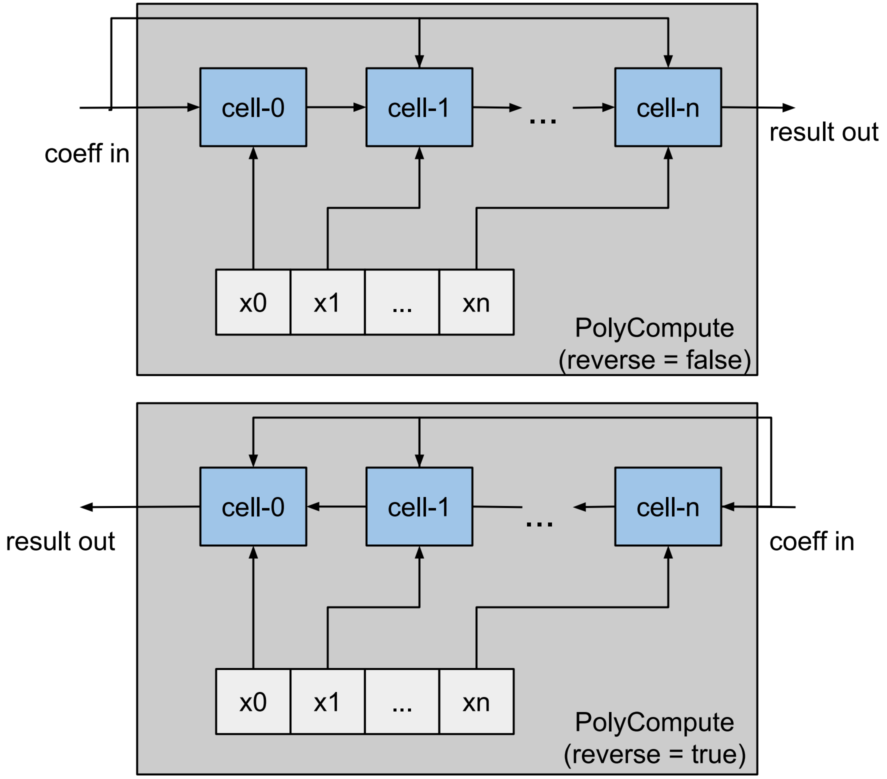
Fixes up to (n-k)/2 input symbols

Key Equation Solver: **polynomial division** (Modified Euclidean Algorithm)

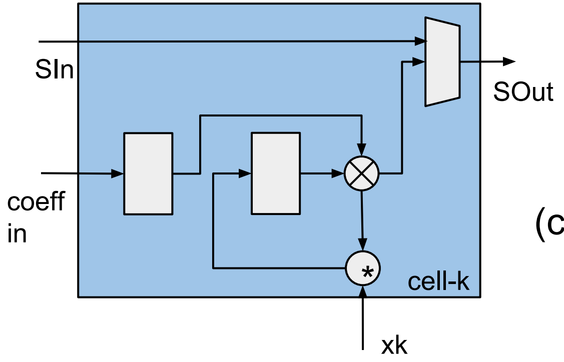
Syndrome computation, Chien Search, Error Correction: **polynomial evaluation**

Inversion operation (in Error Correction) is on the critical path

Polynomial evaluation (Horner's method)

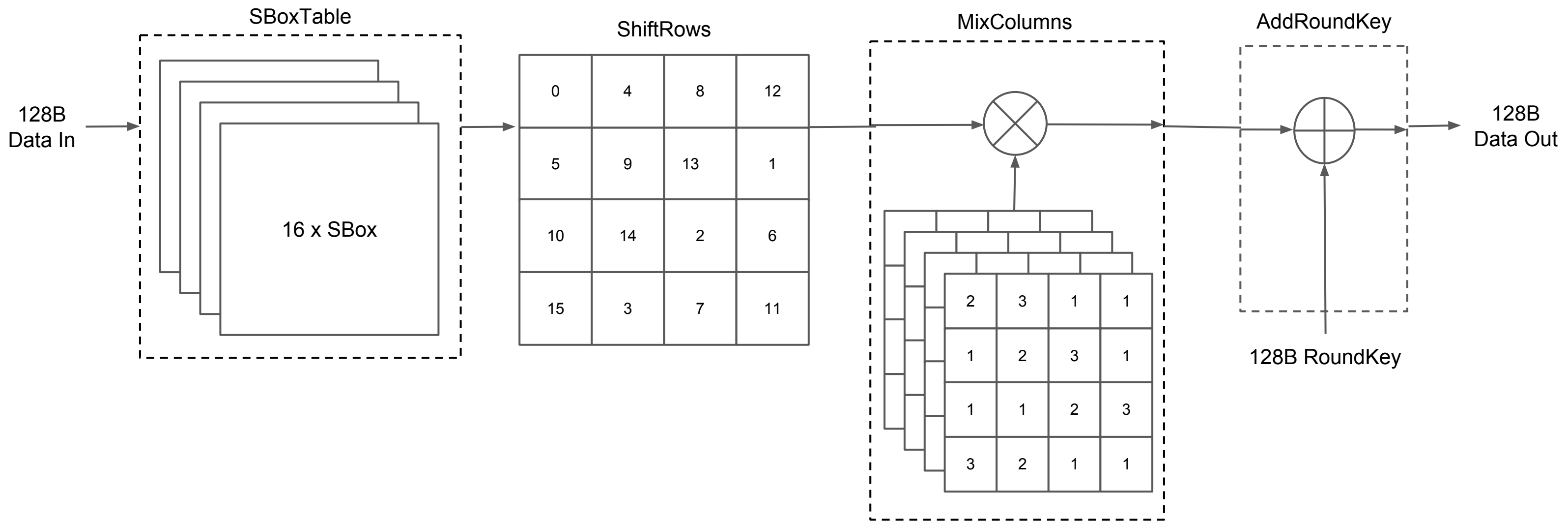


```
class PolyCompute(
   p: RSParams,
   numCells: Int,
   numInputs: Int,
   reverse: Bool
)
```



(c_1 * x + c_2) x + ..

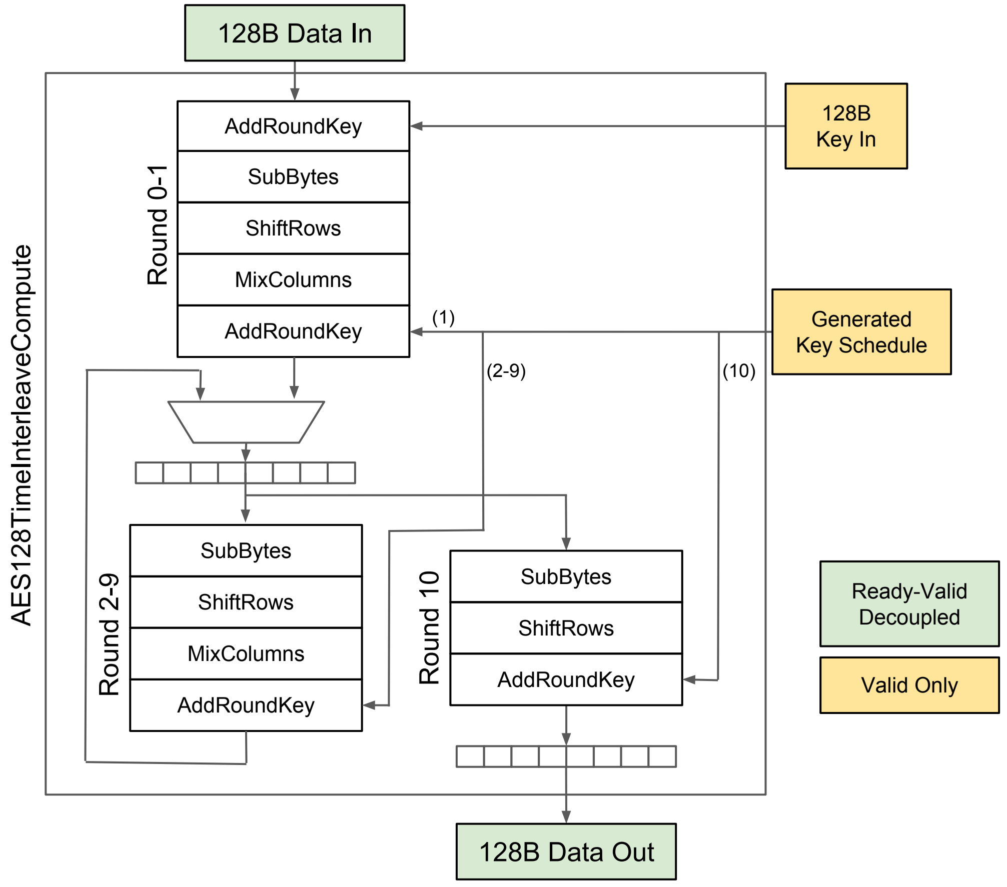
Encryption: AES Encryption



One AES Encrypt Stage
AES runs this 9 times, with a few extra substages before and after

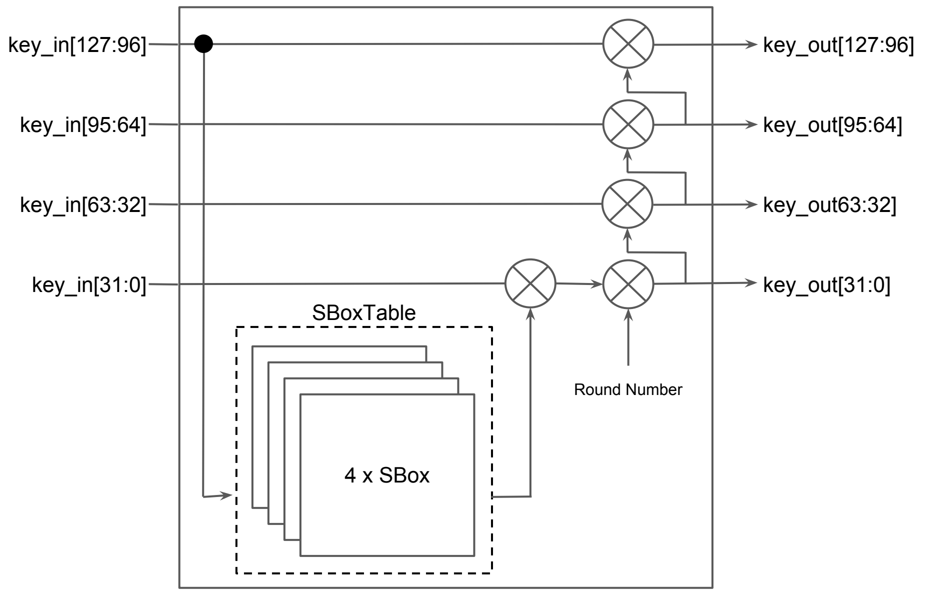
Encryption: AES 128 ECB

- Separate Compute blocks for encrypt and decrypt, with a shared key schedule generator
- Shared Key schedule generator reduces area and compute costs
- Iterative approach is easier to design and performance competitive with pipelined approached

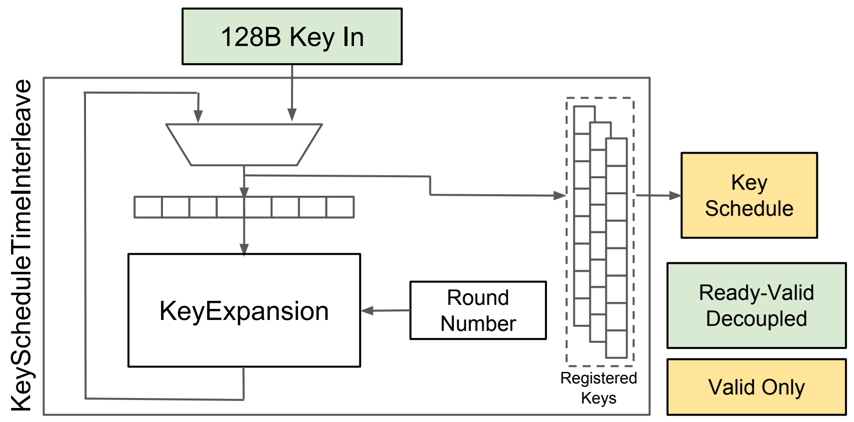


Encryption Block Diagram

Encryption: Key Generation

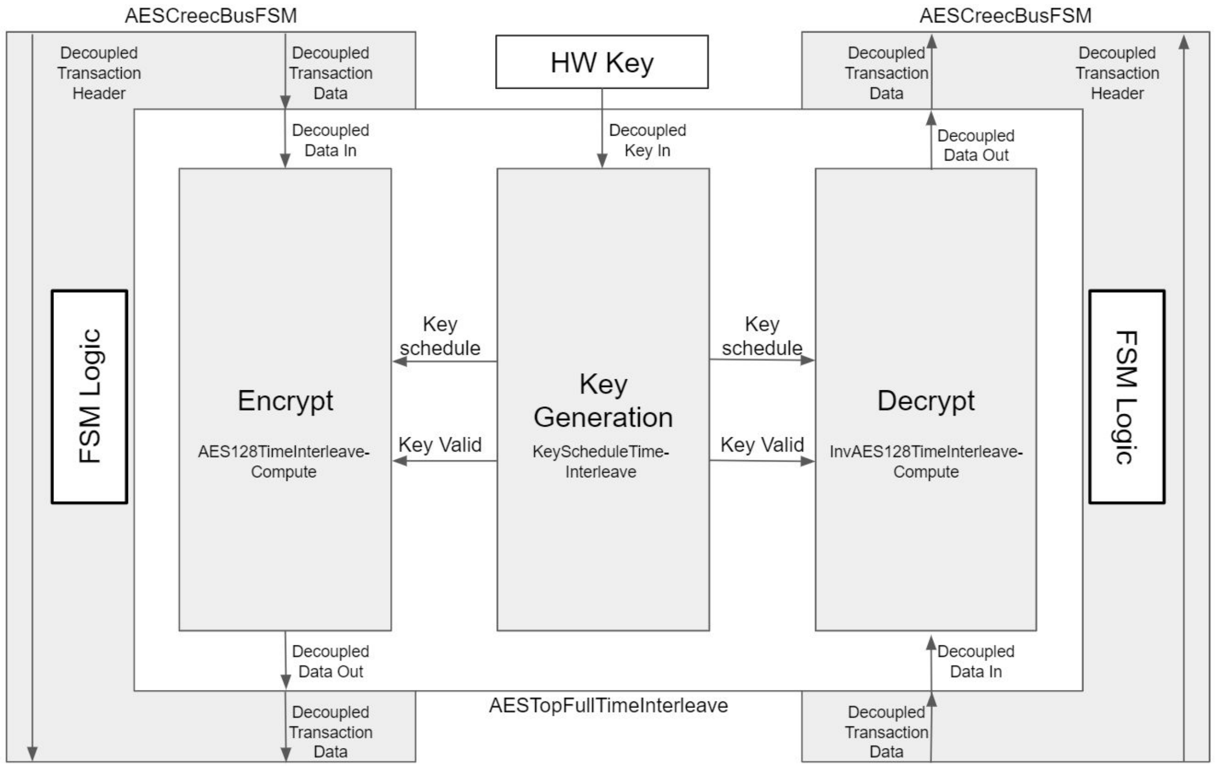


KeyExpansion



Iterative Key Generation

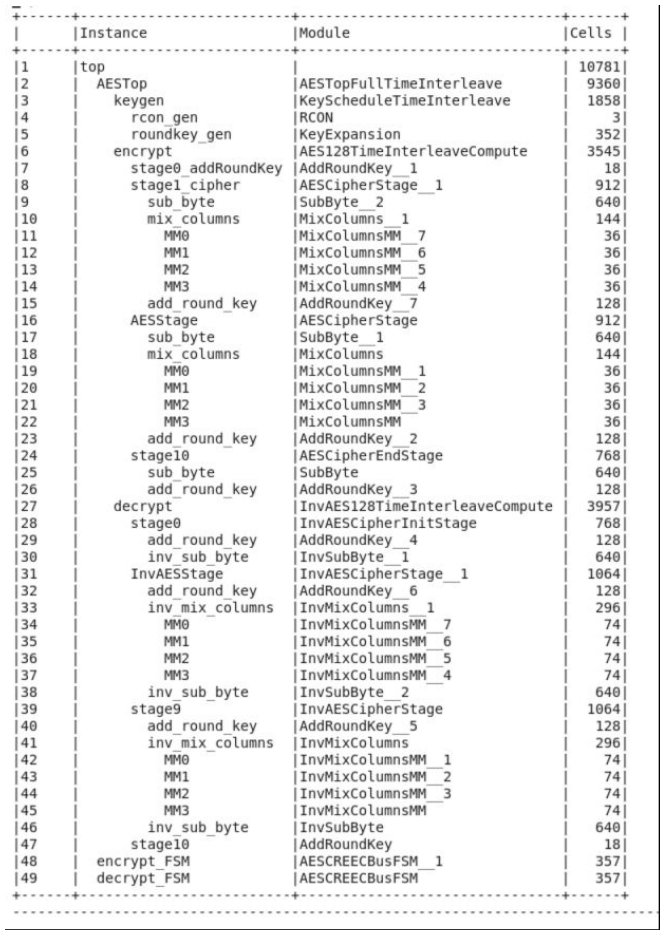
Encryption: Top Integration with CREECBus


AESCreecBusFSM
Decoupled Transaction Header
Decoupled Transaction Data
HW Key
AESCreecBusFSM
Decoupled Transaction Data
Decoupled Transaction Header
Decoupled Data In
Decoupled Key In
Decoupled Data Out
FSM Logic
Encrypt
AES128TimeInterleave-Compute
Key schedule
Key Valid
Key Generation
KeyScheduleTime-Interleave
Key schedule
Key Valid
Decrypt
InvAES128TimeInterleave-Compute
FSM Logic
Decoupled Data Out
Decoupled Data In
AESTopFullTimeInterleave
Decoupled Transaction Data
Decoupled Transaction Data

Encryption Block Diagram

Encryption: Brief Results

- Test latency and critical path are as expected
- Cipher area dominated by SBox Luts
- Key generation area dominated by key registers
- Future work: SBox reutilizations, explorating other operating modes, and time-interleaved performance

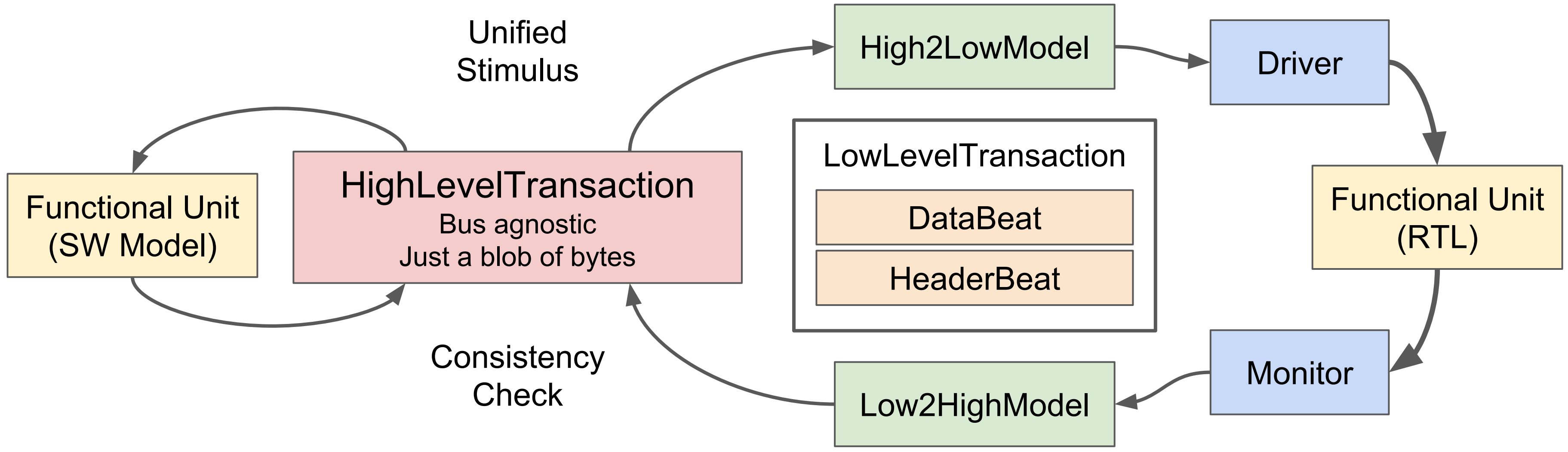


```
+------+----------------------+-------------------------------+------+
|      |Instance              |Module                         |Cells |
+------+----------------------+-------------------------------+------+
|1     |top                   |                               | 10781|
|2     |  AESTop              |AESTopFullTimeInterleave       |  9360|
|3     |    keygen            |KeyScheduleTimeInterleave      |  1858|
|4     |      rcon_gen        |RCON                           |     3|
|5     |      roundkey_gen    |KeyExpansion                   |   352|
|6     |    encrypt           |AES128TimeInterleaveCompute    |  3545|
|7     |      stage0_addRoundKey |AddRoundKey__1              |    18|
|8     |      stage1_cipher   |AESCipherStage__1              |   912|
|9     |        sub_byte      |SubByte__2                     |   640|
|10    |        mix_columns   |MixColumns__1                  |   144|
|11    |          MM0         |MixColumnsMM__7                |    36|
|12    |          MM1         |MixColumnsMM__6                |    36|
|13    |          MM2         |MixColumnsMM__5                |    36|
|14    |          MM3         |MixColumnsMM__4                |    36|
|15    |        add_round_key |AddRoundKey__7                 |   128|
|16    |      AESStage        |AESCipherStage                 |   912|
|17    |        sub_byte      |SubByte__1                     |   640|
|18    |        mix_columns   |MixColumns                     |   144|
|19    |          MM0         |MixColumnsMM__1                |    36|
|20    |          MM1         |MixColumnsMM__2                |    36|
|21    |          MM2         |MixColumnsMM__3                |    36|
|22    |          MM3         |MixColumnsMM                   |    36|
|23    |        add_round_key |AddRoundKey__2                 |   128|
|24    |      stage10         |AESCipherEndStage              |   768|
|25    |        sub_byte      |SubByte                        |   640|
|26    |        add_round_key |AddRoundKey__3                 |   128|
|27    |    decrypt           |InvAES128TimeInterleaveCompute |  3957|
|28    |      stage0          |InvAESCipherInitStage          |   768|
|29    |        add_round_key |AddRoundKey__4                 |   128|
|30    |        inv_sub_byte  |InvSubByte__1                  |   640|
|31    |      InvAESStage     |InvAESCipherStage__1           |  1064|
|32    |        add_round_key |AddRoundKey__6                 |   128|
|33    |        inv_mix_columns |InvMixColumns__1             |   296|
|34    |          MM0         |InvMixColumnsMM__7             |    74|
|35    |          MM1         |InvMixColumnsMM__6             |    74|
|36    |          MM2         |InvMixColumnsMM__5             |    74|
|37    |          MM3         |InvMixColumnsMM__4             |    74|
|38    |        inv_sub_byte  |InvSubByte__2                  |   640|
|39    |      stage9          |InvAESCipherStage              |  1064|
|40    |        add_round_key |AddRoundKey__5                 |   128|
|41    |        inv_mix_columns |InvMixColumns                |   296|
|42    |          MM0         |InvMixColumnsMM__1             |    74|
|43    |          MM1         |InvMixColumnsMM__2             |    74|
|44    |          MM2         |InvMixColumnsMM__3             |    74|
|45    |          MM3         |InvMixColumnsMM                |    74|
|46    |        inv_sub_byte  |InvSubByte                     |   640|
|47    |      stage10         |AddRoundKey                    |    18|
|48    |  encrypt_FSM         |AESCREECBusFSM__1              |   357|
|49    |  decrypt_FSM         |AESCREECBusFSM                 |   357|
+------+----------------------+-------------------------------+------+
```

FPGA Synthesized usage (no flattening or optimization)

Software Modeling / RTL TLM

- Transaction-level modeling (TLM) enables unified stimulus to be used to drive and compare software and RTL algorithm implementations
- We designed a TLM framework in Scala and implemented SW models
- We integrated Chisel testers2 to create CREECBus drivers and monitors



Transaction-Level Modeling

- **Transactions** are abstract; they represent a chunk of data (and/or control)
- **High-level** transactions are
  - Bus-**agnostic** (can work on AXI4-Stream or CREECBus)
  - Bus-parameterization **agnostic** (can be driven on a 64-bit/128-bit bus)

```
trait Transaction
abstract class CREECT extends Transaction
case class CREECHighT(data: Seq[Byte]) extends CREECT
```

- All our software models operate on high-level transactions
- RTL doesn’t see high-level transactions directly

Transaction-Level Modeling

- **Low-level** transactions are
    - Bus-**specific** (represents bus-specific transactions)
    - Bus-parameterization **specific** (bus data width is fixed)

```
abstract class CREECLowT extends CREECT
case class HeaderBeat(len: Int, id: Int)(p: BusParams) extends CREECLowT {
  require(len <= (p.maxBeats - 1))
  require(id <= p.maxInFlight)
}

case class DataBeat(data: Seq[Byte],id: Int)(p: BusParams) extends CREECLowT {
  require(data.length == p.bytesPerBeat)
  require(id <= p.maxInFlight)
}
```

Software Models

- SW models have 1 input and 1 output port and are untimed
  - Concrete models implement **process()**
- High2Low and Low2High are implemented as SW models

```
abstract class SoftwareModel[I <: Transaction, O <: Transaction] {
  def process(in: I) : Seq[O]
  def compose[O2 <: Transaction](s: SoftwareModel[O, O2]): SoftwareModel[I,
O2]
}

class EncryptModel extends SoftwareModel[CREECHighT, CREECHighT] {
  override def process(in: CREECHighT) = {
    val encryptedData = AESEncryption.encrypt(key, in.data)
    Seq(in.copy(data = encryptedData))
  }
}
```

Software Model Composition

- Models can be composed to create data pipelines
- Models are tested by hand, then are used to produce golden reference output for RTL

```
val model =
   CompressModel ->
   PadderModel(16) ->
   EncryptModel ->
   ECCEncoderModel(RS16_8_8) ->
   CommChannel(RS16_8_8, noiseByteLevel=4) ->
   ECCDecoderModel(RS16_8_8) ->
   DecryptModel ->
   StripperModel ->
   DeCompressModel
 val out = model.processTransactions(dataIn)
 assert(out == dataIn)
```

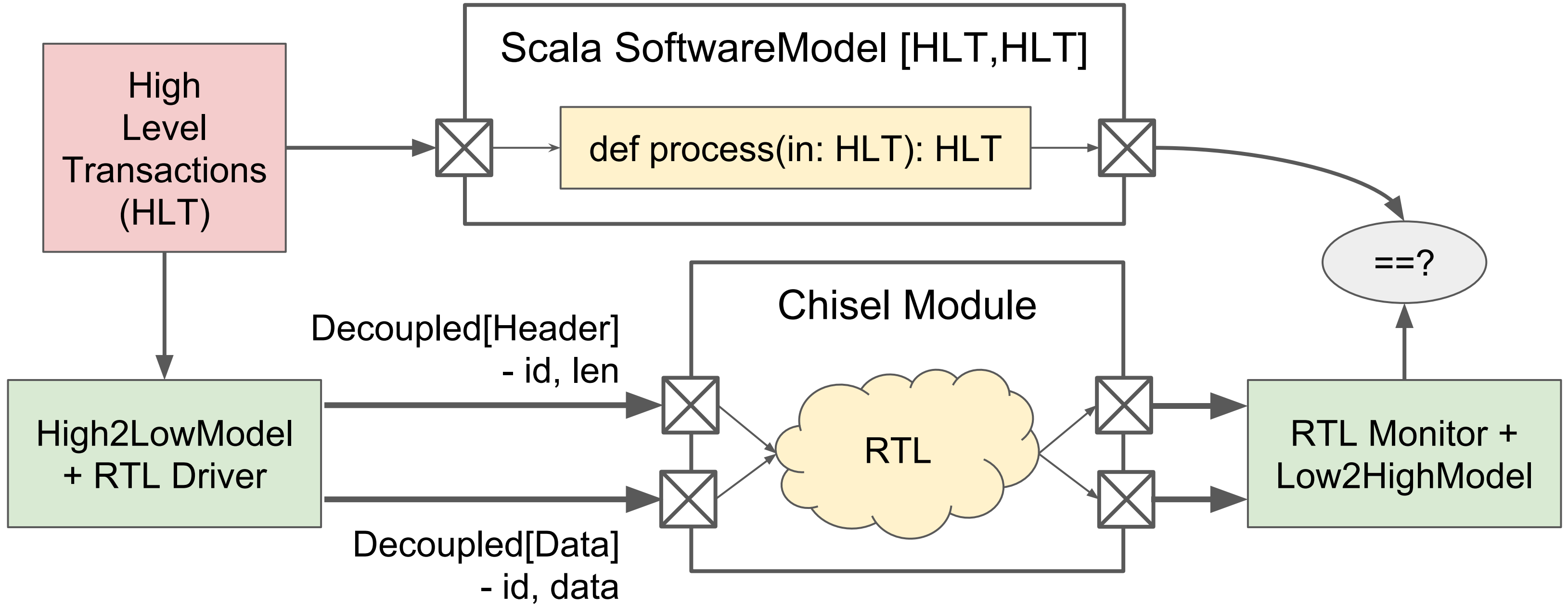
RTL Simulation and Verification

- **Driver/monitor** abstract away bus interaction from core test logic
- Test assertions operate on high-level transactions

```
val tx = CREECHighLevelTransaction(Seq(1, 2, 3, 4, 5, 6, 7, 8))
"the Compressor module" should "compress" in {
    test(CompressorRTL) { c =>
      val model = CompressorModel
      val outGold = model.processTransactions(tx)
      val driver = CREECDriver(c.io.in, c.clock)
      val monitor = CREECMonitor(c.io.out, c.clock)

      driver.pushTransactions(tx)
      c.clock.step(100)
      val out = monitor.receivedTransactions.getAll
      assert(out == outGold)
  }
}
```

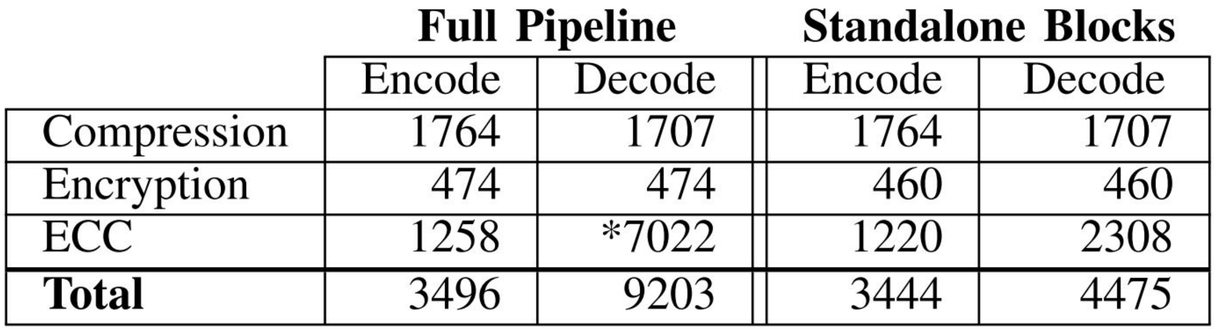
Putting It All Together


High Level Transactions (HLT)
Scala SoftwareModel [HLT,HLT]
def process(in: HLT): HLT
==?
Chisel Module
Decoupled[Header]
- id, len
RTL
High2LowModel + RTL Driver
RTL Monitor + Low2HighModel
Decoupled[Data]
- id, data

- We used this scheme to test our entire CREECelerator pipeline

Results: Cycle Counts

- Cycle counts measured based on 512 bytes of ASCII text (one transaction)
- Full pipeline tests push data through each block in succession
  - The starred number is high because we corrupted the data before ECC decoding
- The standalone tests were run with each block in isolation

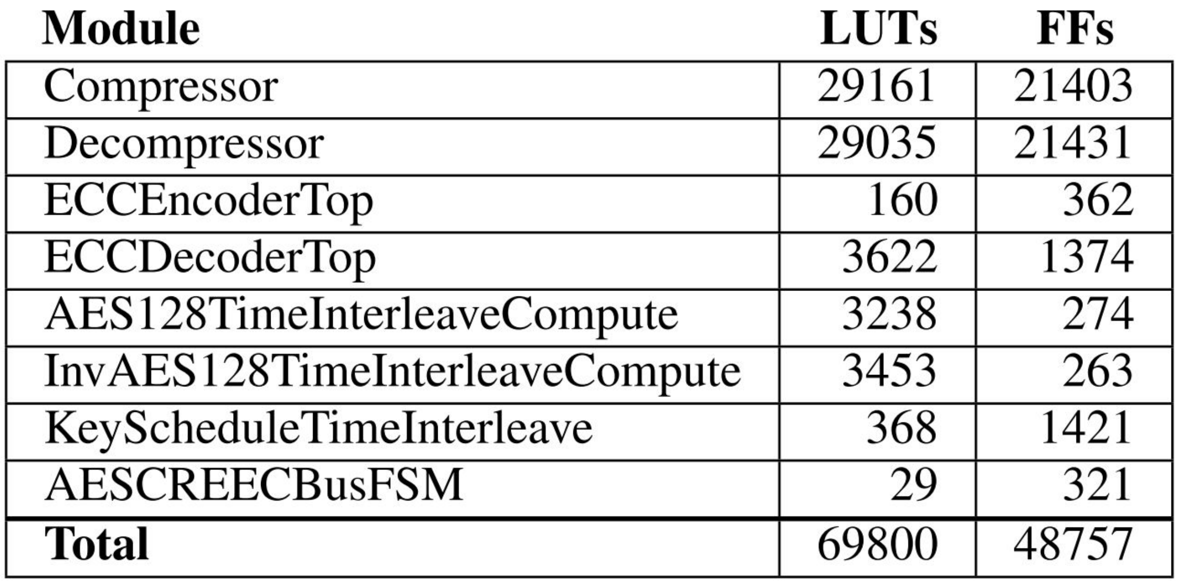


	Full Pipeline		**Standalone Blocks**	
	Encode	Decode	Encode	Decode
Compression	1764	1707	1764	1707
Encryption	474	474	460	460
ECC	1258	*7022	1220	2308
Total	3496	9203	3444	4475

- High numbers in compression are due to the bus semantics
  - Header length is not known until the end, but the header must be sent first

Results: Area (Vivado Synthesis with VC707)

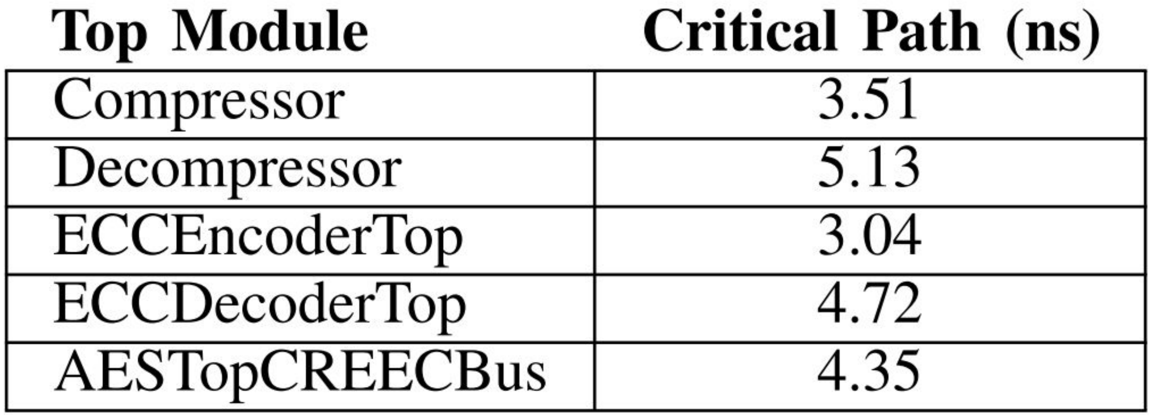
- No Block RAMs or DSPs
- Compressor buffers entire transaction in memory



Module	LUTs	FFs
Compressor	29161	21403
Decompressor	29035	21431
ECCEncoderTop	160	362
ECCDecoderTop	3622	1374
AES128TimeInterleaveCompute	3238	274
InvAES128TimeInterleaveCompute	3453	263
KeyScheduleTimeInterleave	368	1421
AESCREECBusFSM	29	321
Total	69800	48757

Results: Critical Path (Vivado Synthesis with VC707)

- Post-synthesis timing results
- All blocks were able to meet a target of 190 MHz
- The critical path in the decompressor can be cut at the cost of area
- Corresponds to a write/read throughput of 33.7 MB/s / 28.5 MB/s (very bad)

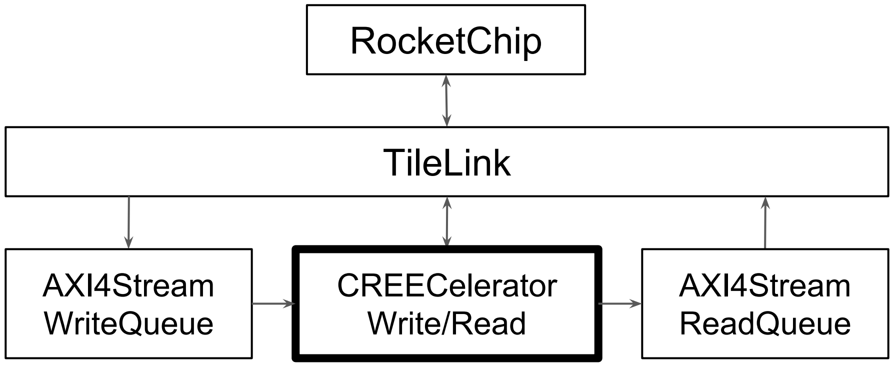


Top Module	Critical Path (ns)
Compressor	3.51
Decompressor	5.13
ECCEncoderTop	3.04
ECCDecoderTop	4.72
AESTopCREECBus	4.35

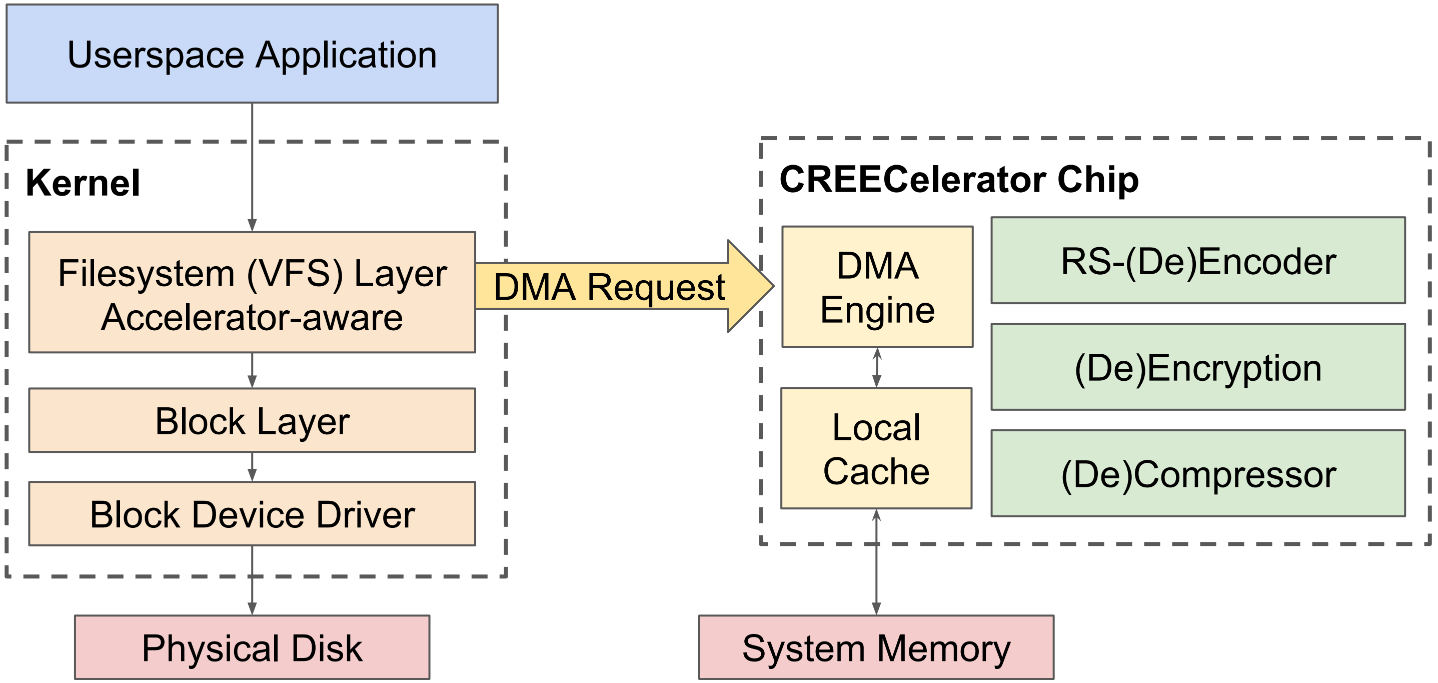
Rocket-chip Integration (it works!)

```
./simulator-freechips.rocketchip.system-CR
EECWConfig ../tests/creec.riscv
Sending data 5784376957523072O1
...
Received header: 8 1 1 1 4 8 0
OUT: (i=0, j=0) creecW=31, gold=31
OUT: (i=0, j=1) creecW=30, gold=30
...
OUT: (i=7, j=7) creecW=-99, gold=-99
creecW PASSED!
```

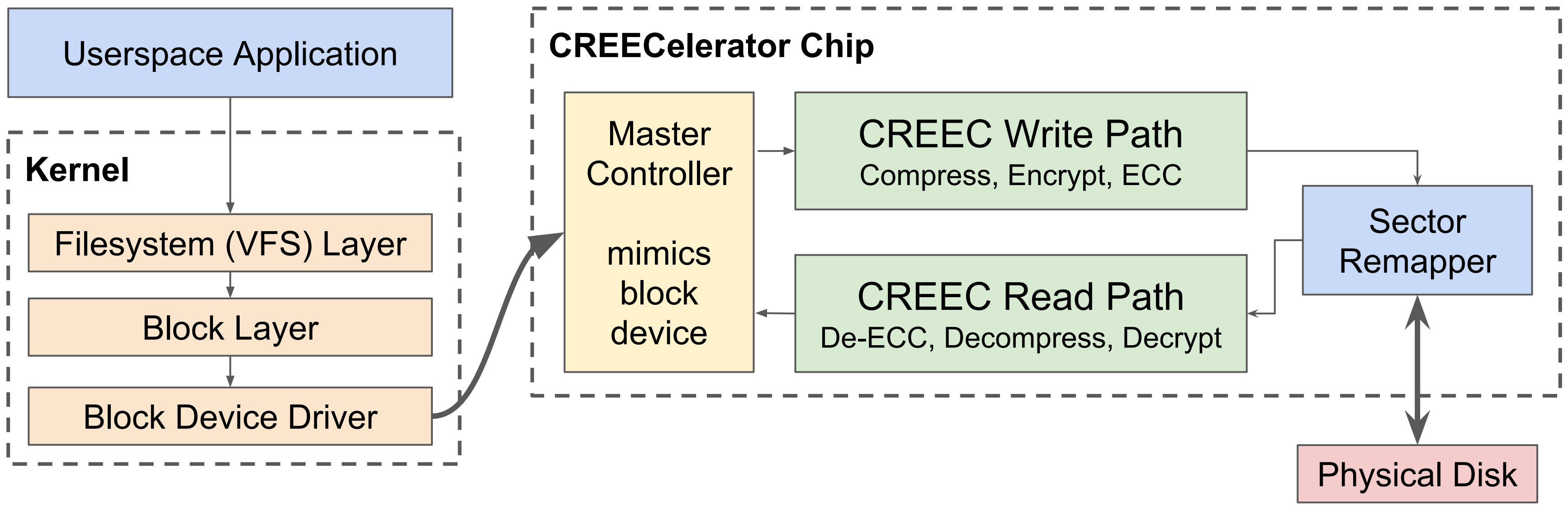
- The CREECelerator write and read paths were attached to Rocket just like CORDIC
- creec.c pushes data into the pipeline and reads the response



Application: Filesystem-Aware Accelerator

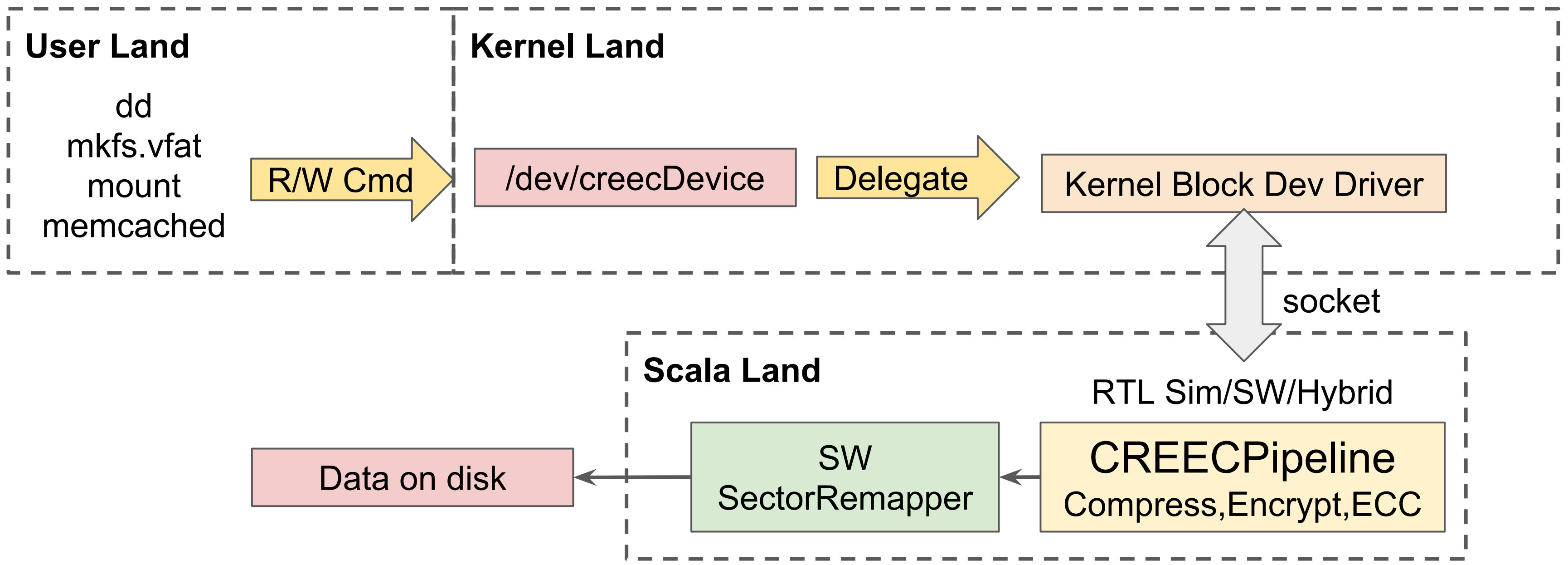


Application: Transparent Storage Utility Accelerator

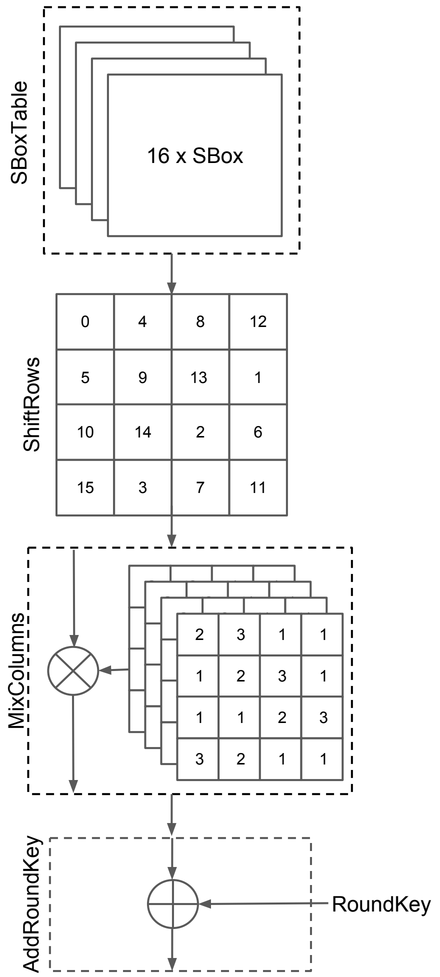


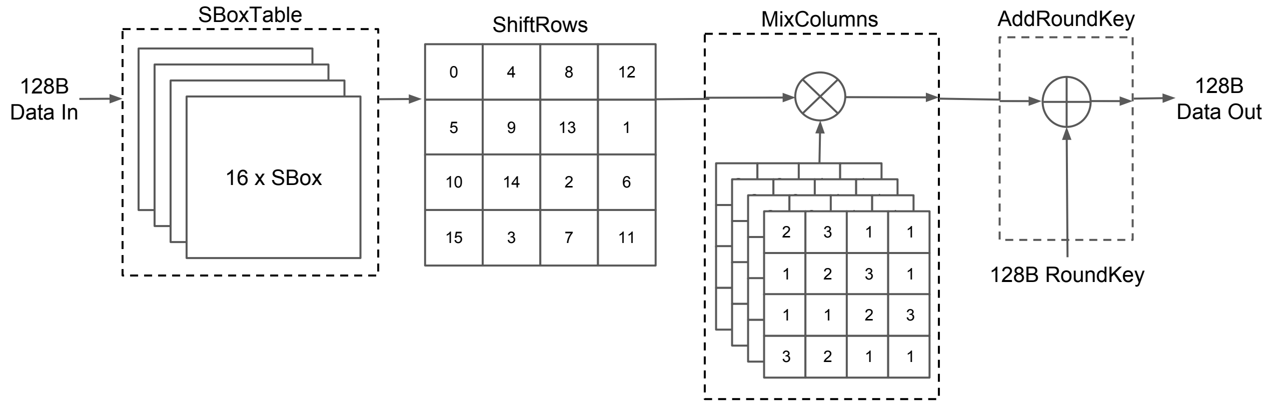
Future Work: System-Level Testing

- Generate real block device traffic and funnel to software and RTL models of the CREEC pipeline; implement the SectorRemapper in software

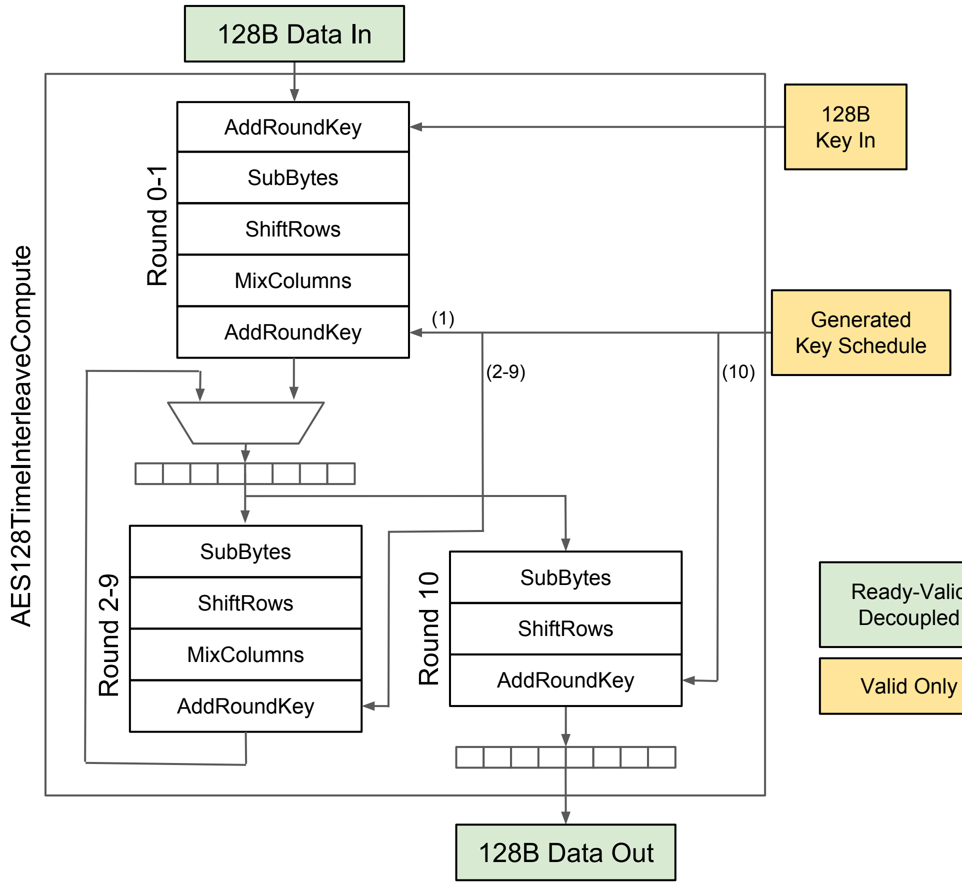


Extra slides


SBoxTable
16 x SBox
ShiftRows
0 4 8 12
5 9 13 1
10 14 2 6
15 3 7 11
MixColumns
2 3 1 1
1 2 3 1
1 1 2 3
3 2 1 1
AddRoundKey
RoundKey

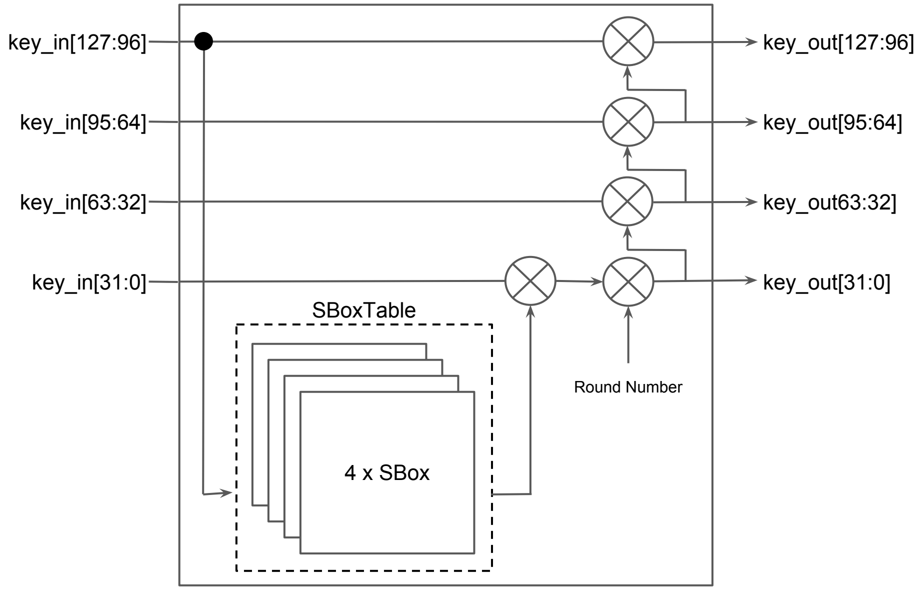

128B Data In
SBoxTable
16 x SBox
ShiftRows
0 4 8 12
5 9 13 1
10 14 2 6
15 3 7 11
MixColumns
2 3 1 1
1 2 3 1
1 1 2 3
3 2 1 1
AddRoundKey
128B RoundKey
128B Data Out

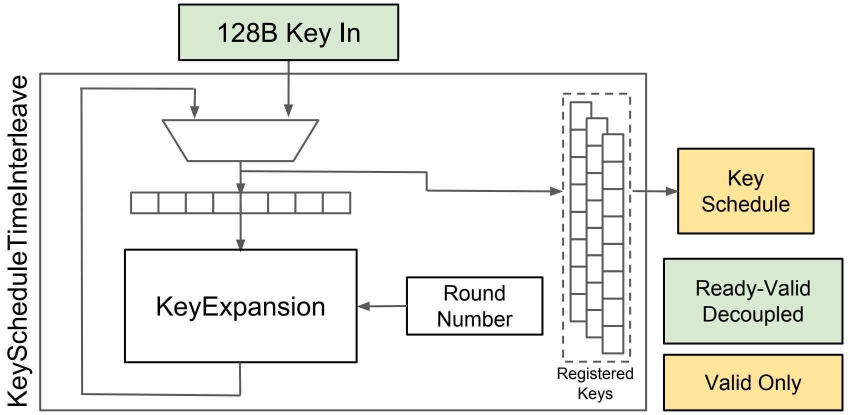
AESEncrypt



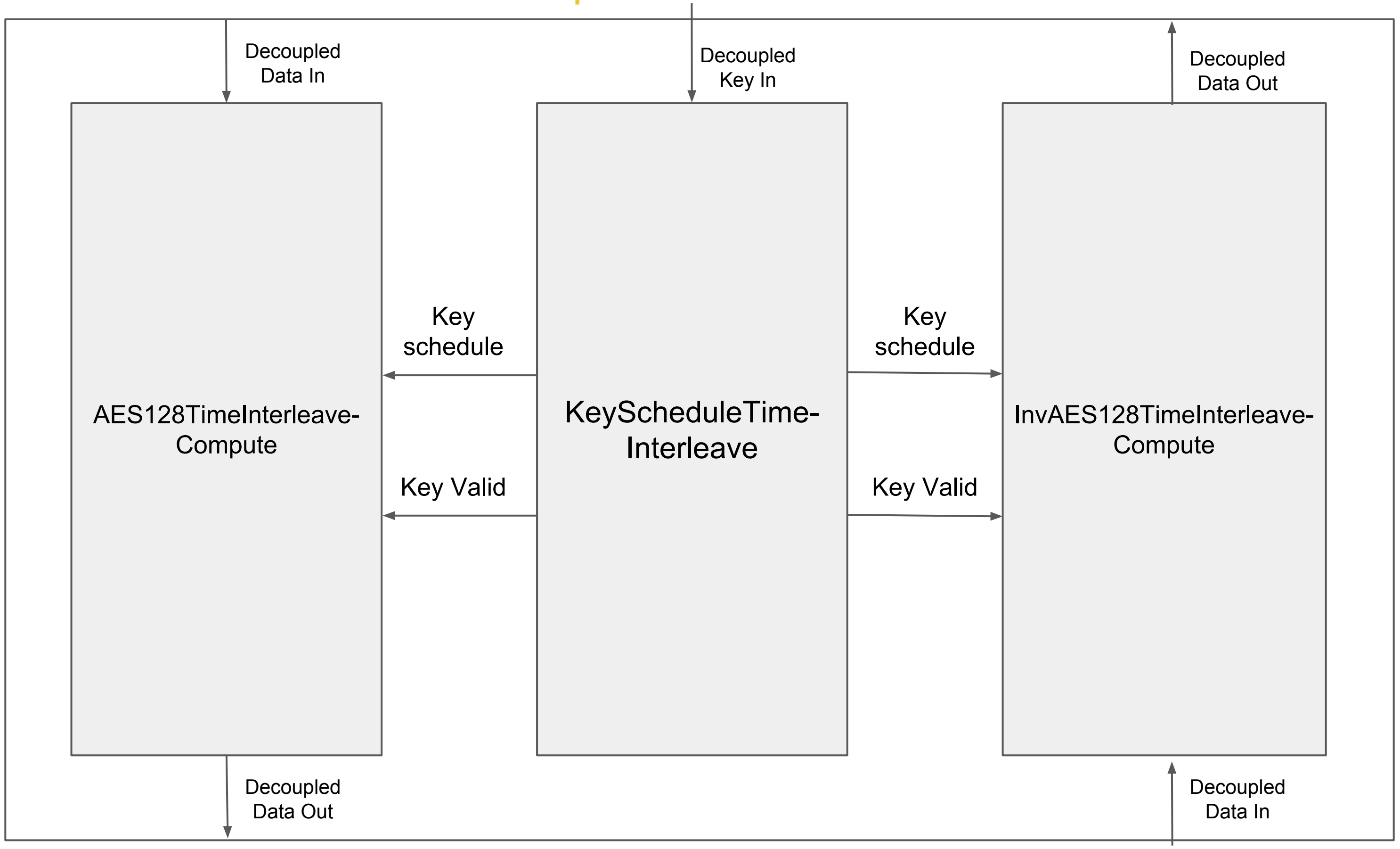
🟩 Ready-Valid Decoupled

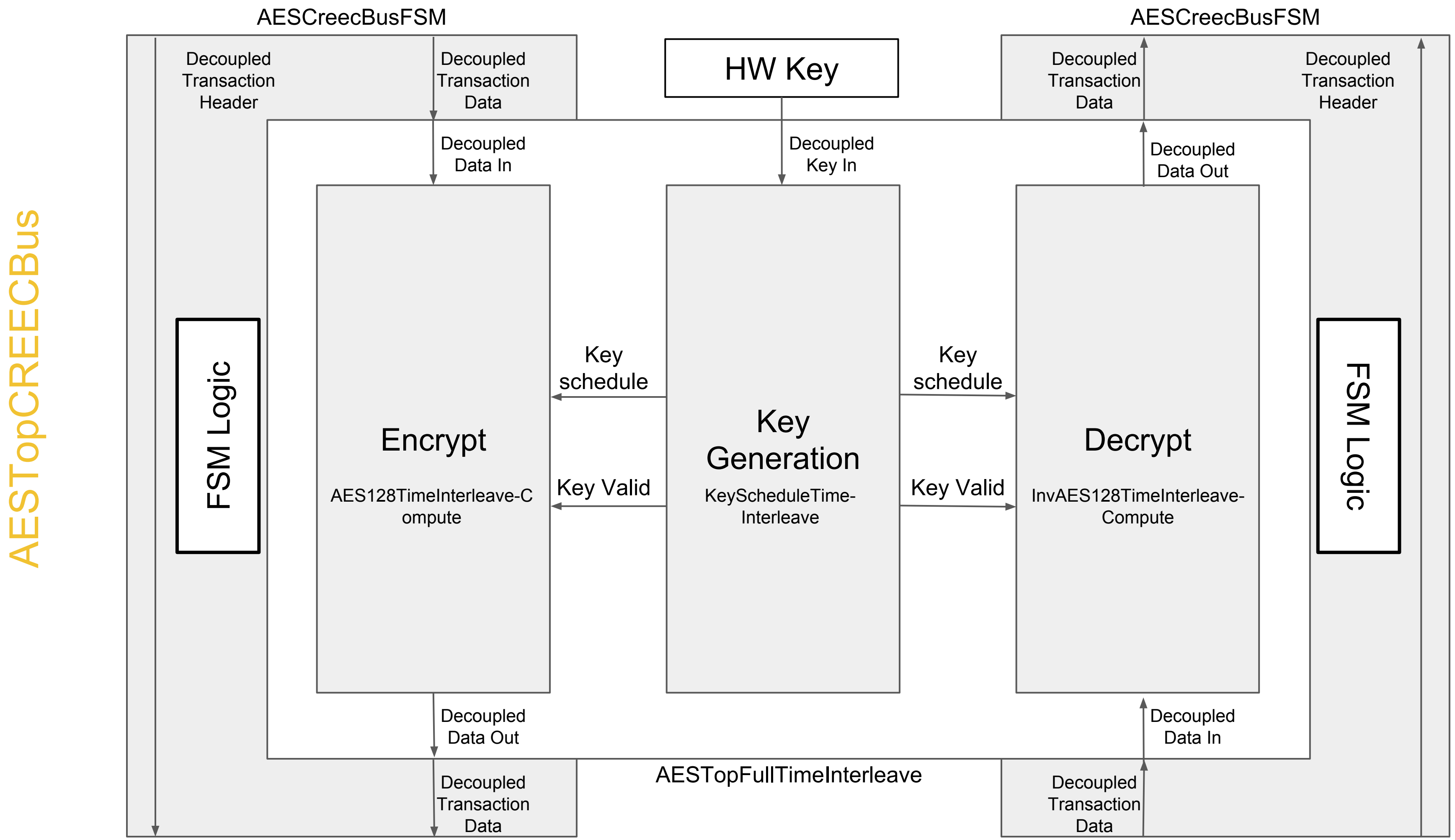
🟨 Valid Only

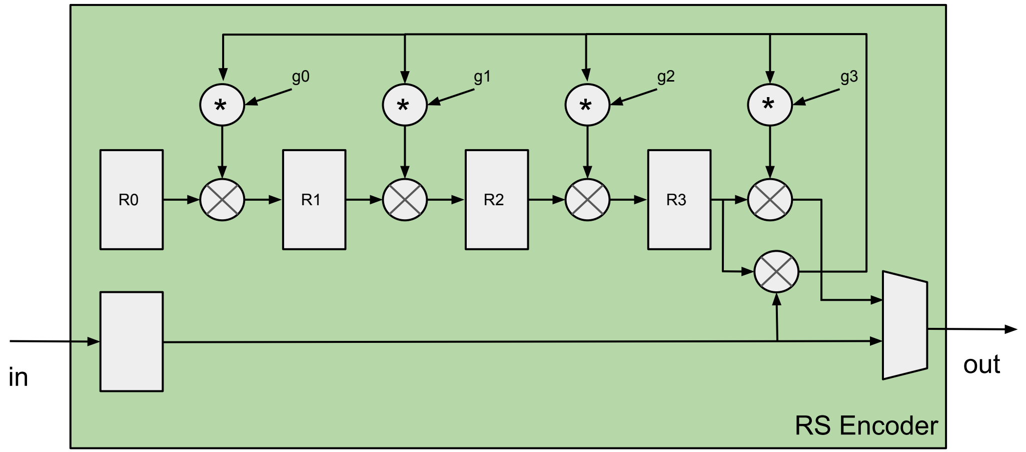

key_in[127:96]
key_in[95:64]
key_in[63:32]
key_in[31:0]
key_out[127:96]
key_out[95:64]
key_out63:32]
key_out[31:0]
SBoxTable
4 x SBox
Round Number

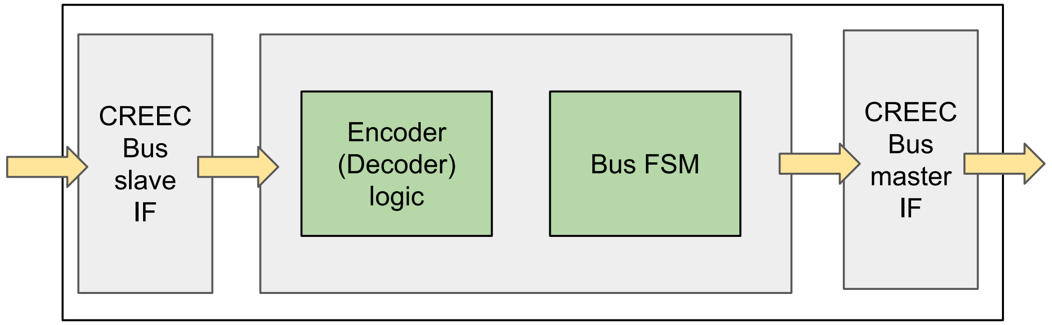

128B Key In
KeyScheduleTimeInterleave
KeyExpansion
Round Number
Registered Keys
Key Schedule
Ready-Valid Decoupled
Valid Only

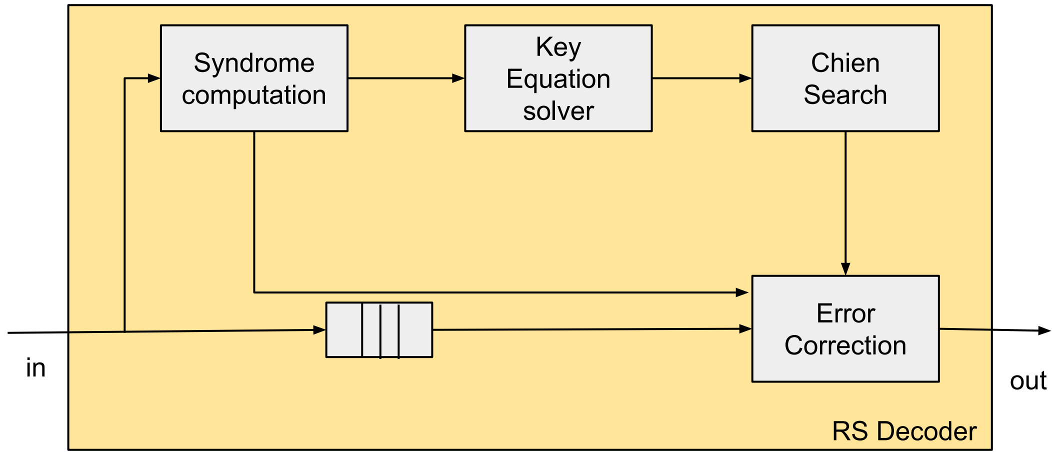
AESTopFullTimeInterleave

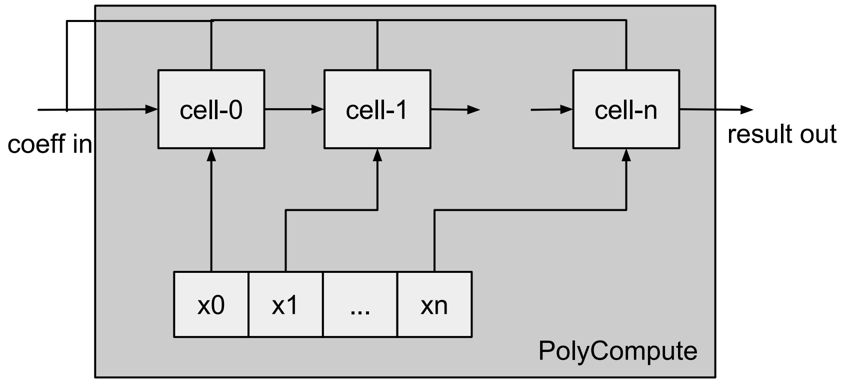


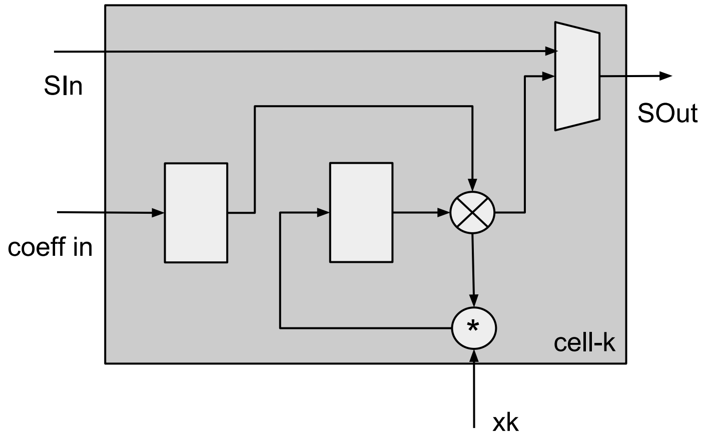

AESCreecBusFSM
AESCreecBusFSM
Decoupled Transaction Header
Decoupled Transaction Data
HW Key
Decoupled Transaction Data
Decoupled Transaction Header
Decoupled Data In
Decoupled Key In
Decoupled Data Out
AESTopCREECBus
FSM Logic
Encrypt
AES128TimeInterleave-Compute
Key schedule
Key Valid
Key Generation
KeyScheduleTime-Interleave
Key schedule
Key Valid
Decrypt
InvAES128TimeInterleave-Compute
FSM Logic
Decoupled Data Out
Decoupled Data In
AESTopFullTimeInterleave
Decoupled Transaction Data
Decoupled Transaction Data

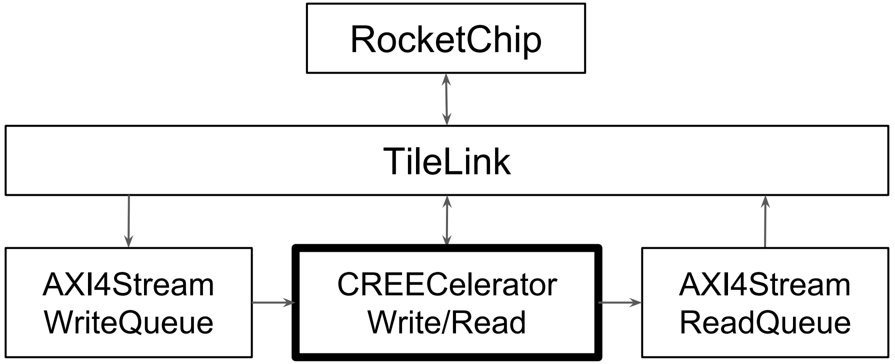

g0
g1
g2
g3
R0
R1
R2
R3
in
out
RS Encoder


CREEC Bus slave IF
Encoder (Decoder) logic
Bus FSM
CREEC Bus master IF


Syndrome computation
Key Equation solver
Chien Search
Error Correction
in
out
RS Decoder


cell-0
cell-1
cell-n
coeff in
result out
x0
x1
...
xn
PolyCompute


SIn
SOut
coeff in
*
cell-k
xk


RocketChip
TileLink
AXI4Stream WriteQueue
CREECelerator Write/Read
AXI4Stream ReadQueue