GALOISCONNECTIONS

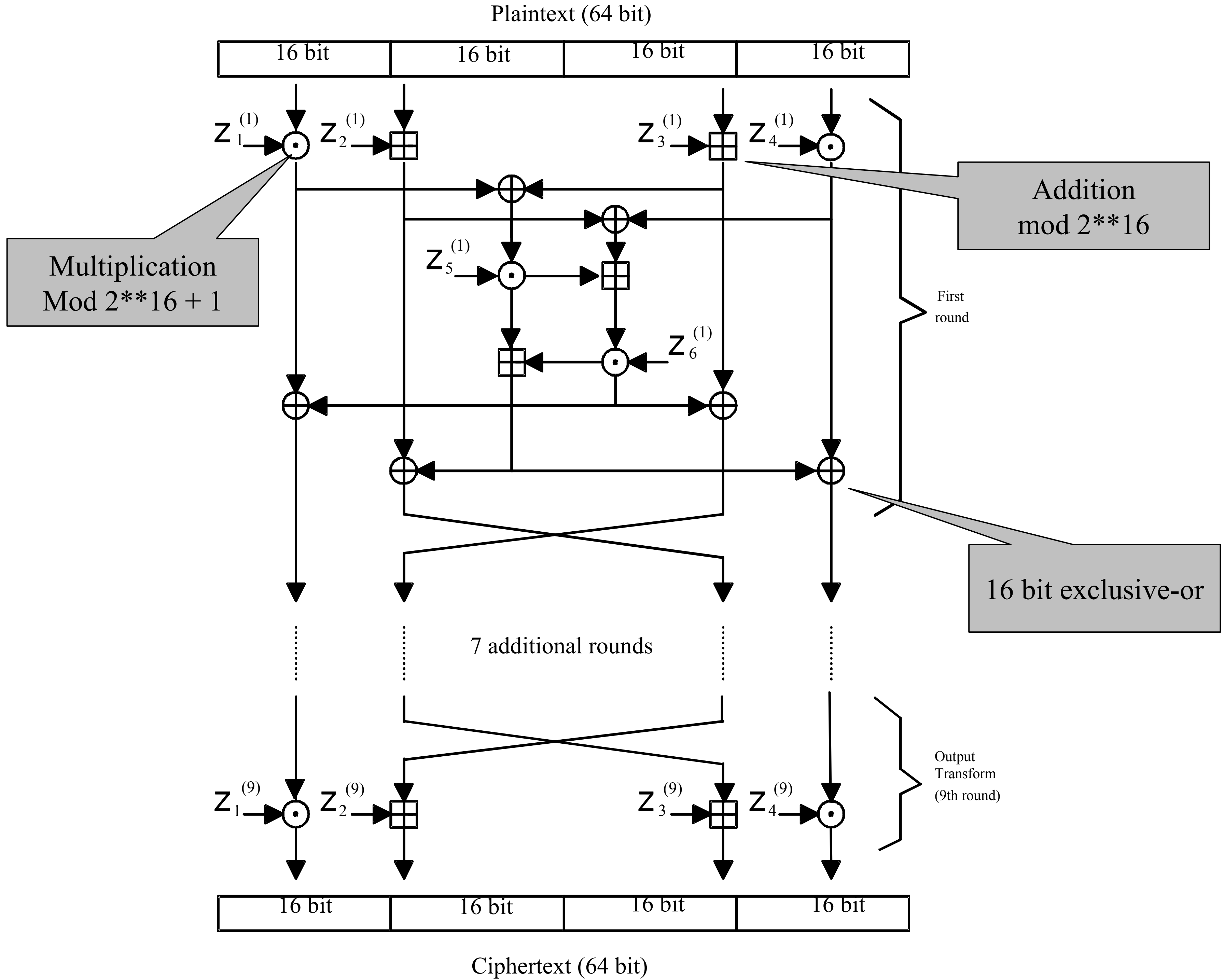
purely functional

Cryptol Tutorial

Worked Example

IDEA

- 64-bit blocks
  - Viewed as four 16-bit blocks
- 128-bit key
  - Expanded to fifty six 16-bit keys
  - Eight 16-bit keys from the key itself
  - Successive rotation to the left by 25 bits
- 16-bit operations
  - Addition, modulo 2**16
  - Multiplication, modulo 2**16+1
    - Treat 0 as 2**16
- 8 rounds, plus post-whitening


Plaintext (64 bit)
16 bit
16 bit
16 bit
16 bit
Z1(1)
Z2(1)
Z3(1)
Z4(1)
Addition mod 2**16
Multiplication Mod 2**16 + 1
Z5(1)
Z6(1)
First round
16 bit exclusive-or
7 additional rounds
Output Transform (9th round)
Z1(9)
Z2(9)
Z3(9)
Z4(9)
16 bit
16 bit
16 bit
16 bit
Ciphertext (64 bit)

Multiplication

```
ideaMul': ([16],[16]) -> [16];
ideaMul' (x,y) = z
 where {
  a,b,c,m: [33];        // worst case needs 33 bits
  a = if (x == 0)
      then 2**16        // if 0 use 2**16
      else 0 # x;       // pad to 33 bits
  b = if (y == 0)
      then 2**16        // if 0 use 2**16
      else 0 # y;       // pad to 33 bits
  c = (a * b) % m;      // multiply modulo 2**16 + 1
  m = 2**16 + 1;        // the modulus
  z = c @@ [17..32];            // return least sig 16 bits of product
 };
```

Multiplication (More Efficient)

```
ideaMul: ([16],[16]) -> [16];
ideaMul (x,y) = z
 where {
  a,b,c,m : [32];
  a = 0 # x;                          // pad to 32 bits
  b = 0 # y;                          // pad to 32 bits
  c = if (a == 0)
      then m - b                // 2**16 * b = m - b (mod m)
      else if (b == 0)
          then m - a  // 2**16 * a = m - a (mod m)
          else (a * b) % m;
  m = 2**16 + 1;                // the modulus
  z = c @@ [16..31];   // return least sig 16 bits of product
 };
```

Key Schedule

```
encryptionKeySchedule: [128] -> [52][16];
encryptionKeySchedule key = sks'
 where {
  ks = [key] # [| k <<< 25 || k <- ks |];
  ks': [7][128];
  ks' = ks @@ [0 .. 6];

  sks: [56][16];
  sks = join [| splitBy (8,k) || k <- ks' |];
  sks': [52][16];
  sks' = sks @@ [0..51];      };
```

Block Encryption

```
encryptBlock: ([64],[52][16]) -> [64];
encryptBlock (pt,sks) = join ct
 where {
  ks': [48][16];
  ks' = sks @@ [0 .. 47];
  ks: [8][6][16];
  ks = split ks';
  pt': [4][16];
  pt' = split pt;
  ps: [9][4][16];
  ps = [pt'] # [| ideaRound (p,k) || p <- ps
                                  || k <- ks |];
  ct' = ps @ 8;   // more...
```

Encryption continued

```
x0,x1,x2,x3: [16];
x0 = ct' @ 0;
x1 = ct' @ 1;
x2 = ct' @ 2;
x3 = ct' @ 3;
k0 = sks @ 48;
k1 = sks @ 49;
k2 = sks @ 50;
k3 = sks @ 51;
y0 = ideaMul (x0,k0);
y1 = x2 + k1;
y2 = x1 + k2;
y3 = ideaMul (x3,k3);
ct = [y0 y1 y2 y3];          };
```

Order of x's is switched

Each Round

```
ideaRound: ([4][16],[6][16]) -> [4][16];
ideaRound ([x0 x1 x2 x3],[k0 k1 k2 k3 k4 k5]) = [y0 y1 y2 y3]
 where {
  t0,t1,t2,t3,t4,t5,t6,t7,t8,t9: [16];
  t0 = ideaMul (x0,k0);
  t1 = x1 + k1;
  t2 = x2 + k2;
  t3 = ideaMul (x3,k3);
  t4 = t0 ^ t2;
  t5 = t1 ^ t3;
  t6 = ideaMul (t4,k4);
  t7 = t5 + t6;
  t8 = ideaMul (t7,k5);
  t9 = t6 + t8;
```

```
y0,y1,y2,y3: [16];
  y0 = t0 ^ t8;
  y1 = t2 ^ t8;
  y2 = t1 ^ t9;
  y3 = t3 ^ t9;
 };
```

Style Critique

- Breaking down into tiny steps
  - Can help debugging
  - Gives an “assembly code” result
- Abstraction
  - Name important concepts
  - Keep separate concepts separate
- Redo IDEA using more abstraction

Multiplication

```
ideaMul': ([16],[16]) -> [16];
ideaMul' (a,b) = to16 ((to33 a * to33 b) % modulus);


modulus = 2**16 + 1;          // Modulus for multiplication

to33 : [16] -> [33];
to33 x = if (x == 0)
         then 2**16           // if 0 use 2**16
         else 0 # x;   // otherwise pad with 0s

to16 z = z @@ [w-16 .. w-1]
  where {w = width z};
```

Multiplication (more efficient)

```
ideaMul: ([16],[16]) -> [16];
ideaMul (a,b)                  // 2**16 * b = m - b (mod m)
  = to16 (if (a == 0) then modulus - to32 b
          else
          if (b == 0) then modulus - to32 a
          else
            (to32 a * to32 b) % modulus
          );


to32 : [16] -> [32];
to32 x = 0 # x;                // pad to 32 bits
```

Key Expansion

```
encryptionKeySchedule: [128] -> [52][16];
encryptionKeySchedule key = sks @@ [0..51]
 where {
  ks: [inf][128];
  ks = [key] # [| k <<< 25 || k <- ks |];

  sks: [inf][16];
  sks = join [| splitBy (8,k) || k <- ks |];
 };
```

Block Encryption

```
encryptBlock: ([64],[52][16]) -> [64];
encryptBlock (pt,sks) = join ct
 where {
  ks: [8][6][16];
  ks = split (sks @@ [0 .. 47]);
  pt': [4][16];
  pt' = split pt;

  ps: [9][4][16];
  ps = [pt'] # [| ideaRound (p,k) || p <- ps
                                  || k <- ks |];
  ct = whiten (switch (ps @ 8), sks@@[48..51]);
 };
```

Helper Functions

```
whiten : ([4][16],[4][16]) -> [4][16];
whiten ([x0 x1 x2 x3], [k0 k1 k2 k3])
  = [ (ideaMul(x0,k0))     (x1 + k1)
        (x2 + k2)      (ideaMul(x3,k3)) ];


switch [a b c d] = [a c b d];
```

Each Round

```
ideaRound: ([4][16],[6][16]) -> [4][16];
ideaRound (x,k) = switch t ^ [r r s s]
 where {
  t = whiten (x,k@@[0..3]);

  p = ideaMul (t@0 ^ t@2, k@4);
  q = p + (t@1 ^ t@3);
  r = ideaMul (q,k@5);
  s = p + r;
 };
```

Cryptol Idiom: For Loops

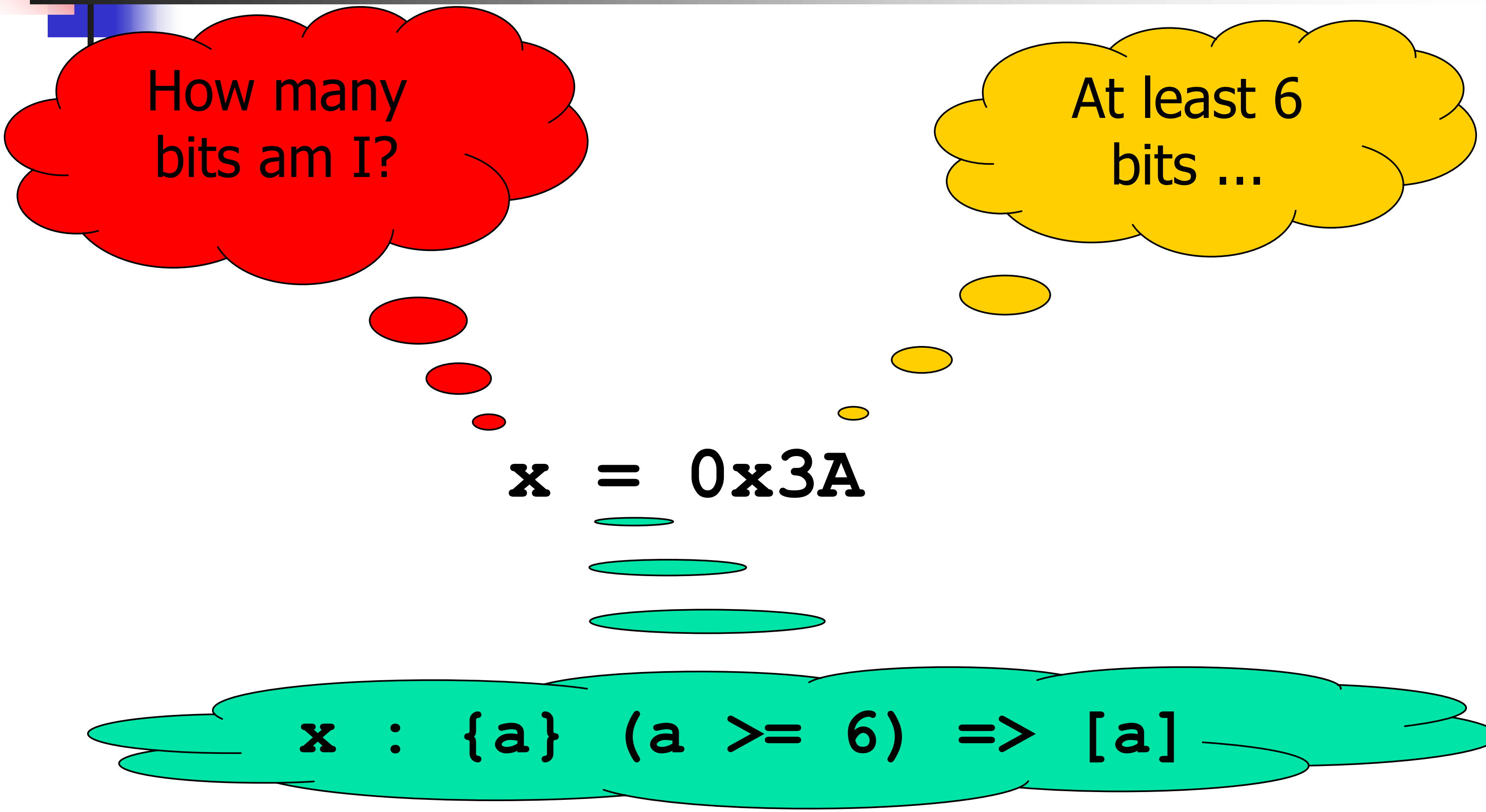
- Factors
  - Capture the body of the for-loop as a function
  - Identify the state variables
  - Define a recurrence
- Example: Sum the elements of a matrix:

```
sum xs = sums @ (width xs - 1)
  where
    { sums = [|  x + y || x <- xs
                      || y <- [0] # sums |];
    };
```

Size Polymorphism


```
add32 0xB4 0x3A
```

Size Polymorphism



Shape Polymorphism

What types do I handle?

Four of something to four of the same thing...

```
swab [a b c d] = [d c b a]
```

```
swab : {a} [4]a -> [4]a
```

Controlling Polymorphism

```
xor : {a b c}
    ([a]b,[c]b) -> [min(a,c)]b
```

```
xor(xs, ys) = [| (x & ~y) | (~x & y)
              || x <- xs
              || y <- ys |]
```

Controlling Polymorphism

```
xor : {a} ([a], [a]) -> [a]
xor(xs, ys) = [| (x & ~y) | (~x & y)
                || x <- xs
                || y <- ys |]
```

A Cryptol Idiom: Padding

- Key padding for MD5:

```
pad : {a} (6 >= a) =>
     [a] -> [512*((a+65+511)/512)]

pad key = key # [True] # 0 # size
  where
     size : [64]
    size = width key
```

> 0 can have any size, so fills out to satisfy the type constraint

Split

- Common task
  - Treat a 32 bit word as 4 bytes

```
toBytes : [32] -> [4][8];
toBytes x = split x;
```

- The split function has a very general type
  - Can be used to perform any split

```
split : {a b c} [a*b]c -> [a][b]c
```

Join

- Converse task
  - Treat 4 bytes as a 32 bit word

```
fromBytes : [4][8] -> [32];
fromBytes x = join x;
```

- The join function has a very general type
  - Can be used to perform any join

```
join : {a b c} [a][b]c -> [a*b]c
```

General reshaping

- Composite task
  - Treat four 12-bit words as three 16 bit words

```
change : [4][12] -> [3][16];
change x = split (join x);
```

- The general version of this has a very general type
  - Can be used to perform any regular rearranging

```
reshape : {a b c d e}
        (d*e == a*b) => [a][b]c -> [d][e]c;
reshape x = split (join x);
```

Transpose

- Transpose an nxm structure into an mxn structure
  - From [[1 2 3] [4 5 6]]
  - To   [[1 4] [2 5] [3 6]]

```
transpose xss
    = [| col (j,xss)
     || j <- [0 .. (width (xss @ 0) - 1)] |];

col (j,xss)
    = [| (xss @ i) @ j
     || i <- [0 .. (width xss - 1)] |];
```

Executing Cryptol

- A Cryptol interpreter
- A compiler that translates Cryptol specifications into executable code
  - Reference implementations currently
- Cryptol implementations of the AES algorithms shall be fast enough for checking the AES Known Answer Tests and Monte Carlo Tests

One Specification, Multiple Implementations

- Fundamental DSL concept:

  Distinguish between model and rendition

- Cryptol specifications are designed to be independent of the target language
  - Interpret specification
  - Reference implementation
  - Generate C code or Java
  - Machines with alternate word sizes, endian-ness