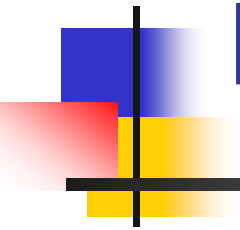
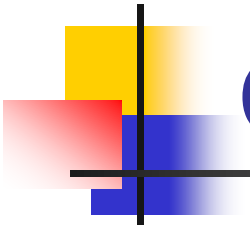


Advanced Infosec Machine Model and DSLs

John Launchbury

Thomas Nordin

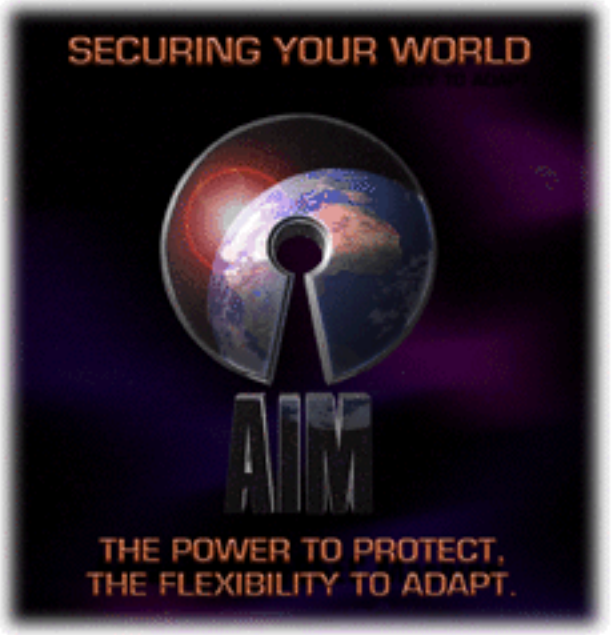
Oregon Graduate Institute



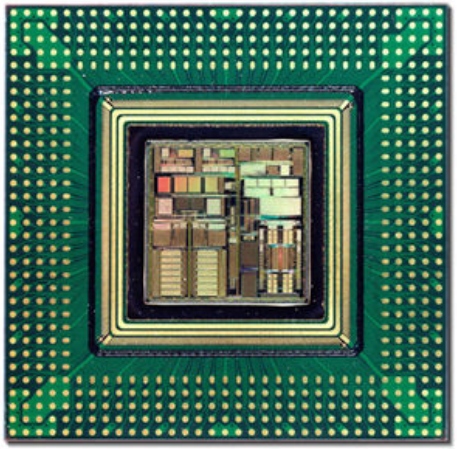
Overview of this talk

- AIM overview
- Introduction to OGI's AIM project
- Modeling permutations
  - Permutation building blocks
  - Building complex permutations from parts
- Modeling Sbox functions
- Summary

AIM



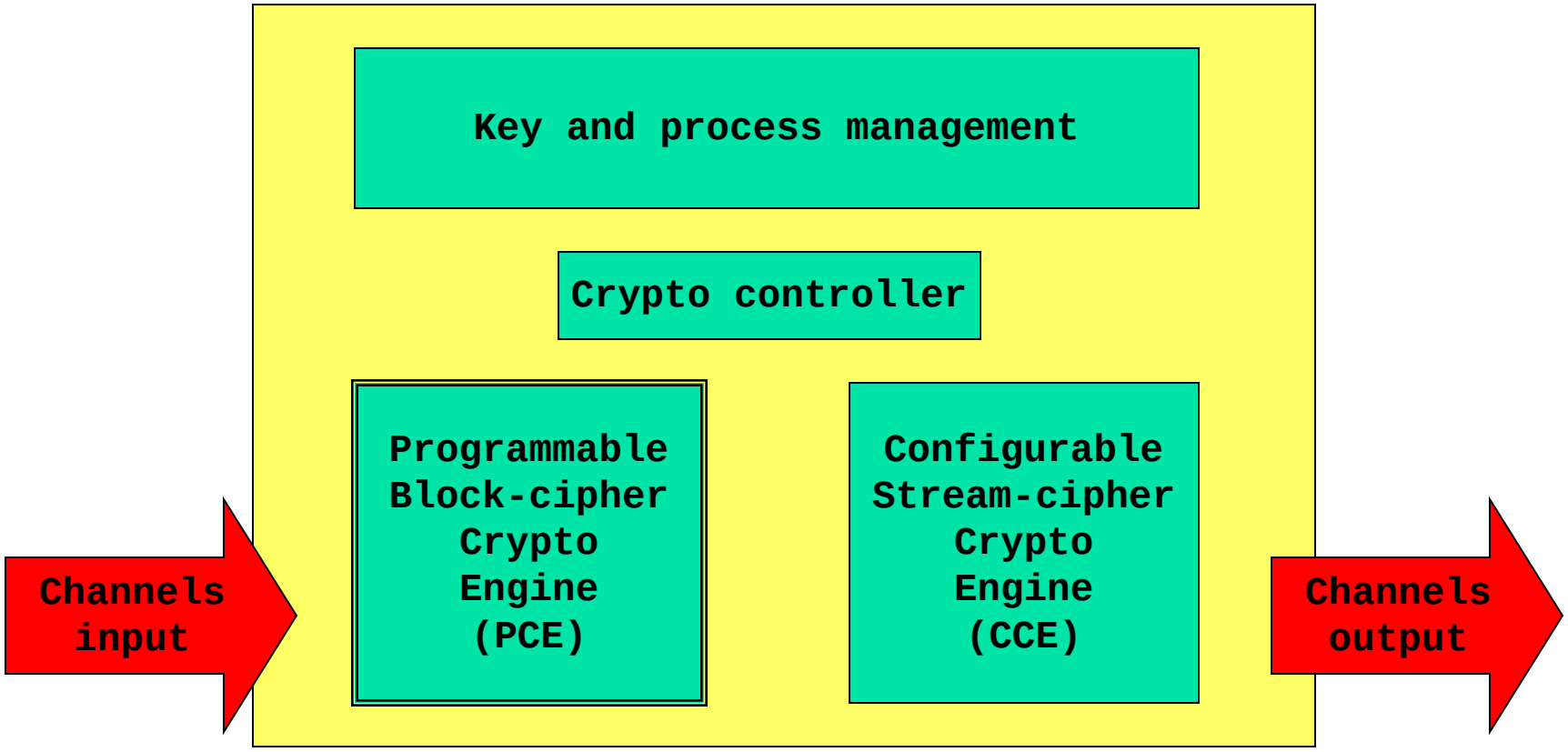
- Motorola AIM
  (Advanced INFOSEC Machine)



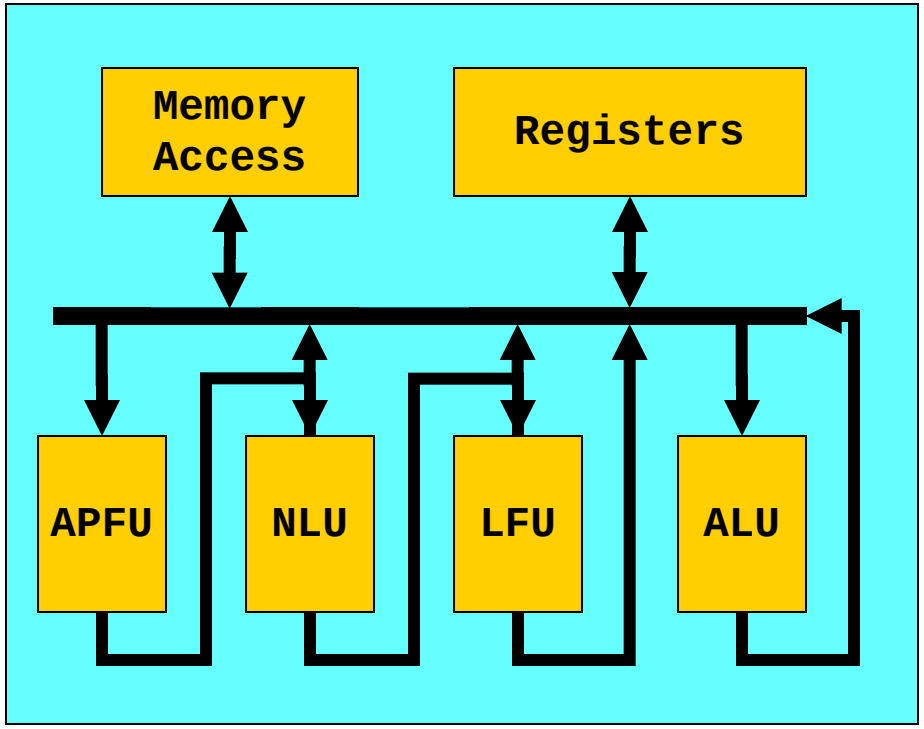
- On-board encryption engines
- MASK technology
  (Mathematically Assured Separation Kernel)
- Physically tamper-proof

www.motorola.com/GSS/SSTG/ISSPD/INFOSEC/Embedded/AIM/

AIM overview



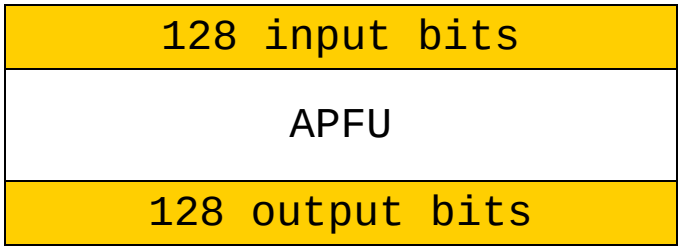
(Simplified) PCE Internal Structure



- Execution components
  - APFU (Permutation Function Unit)
    - 16 predefined permutations
  - NLU (Non-Linear Unit)
    - 16 one-bit memories
    - Independently addressable
  - LFU (Linear Function Unit)
    - XOR unit
  - ALU
- PCE microcode
  - Each component's in-parallel operation specified each cycle
  - Visible pipeline delays when accessing registers
  - Memory accessed via external requests

OGI's Project

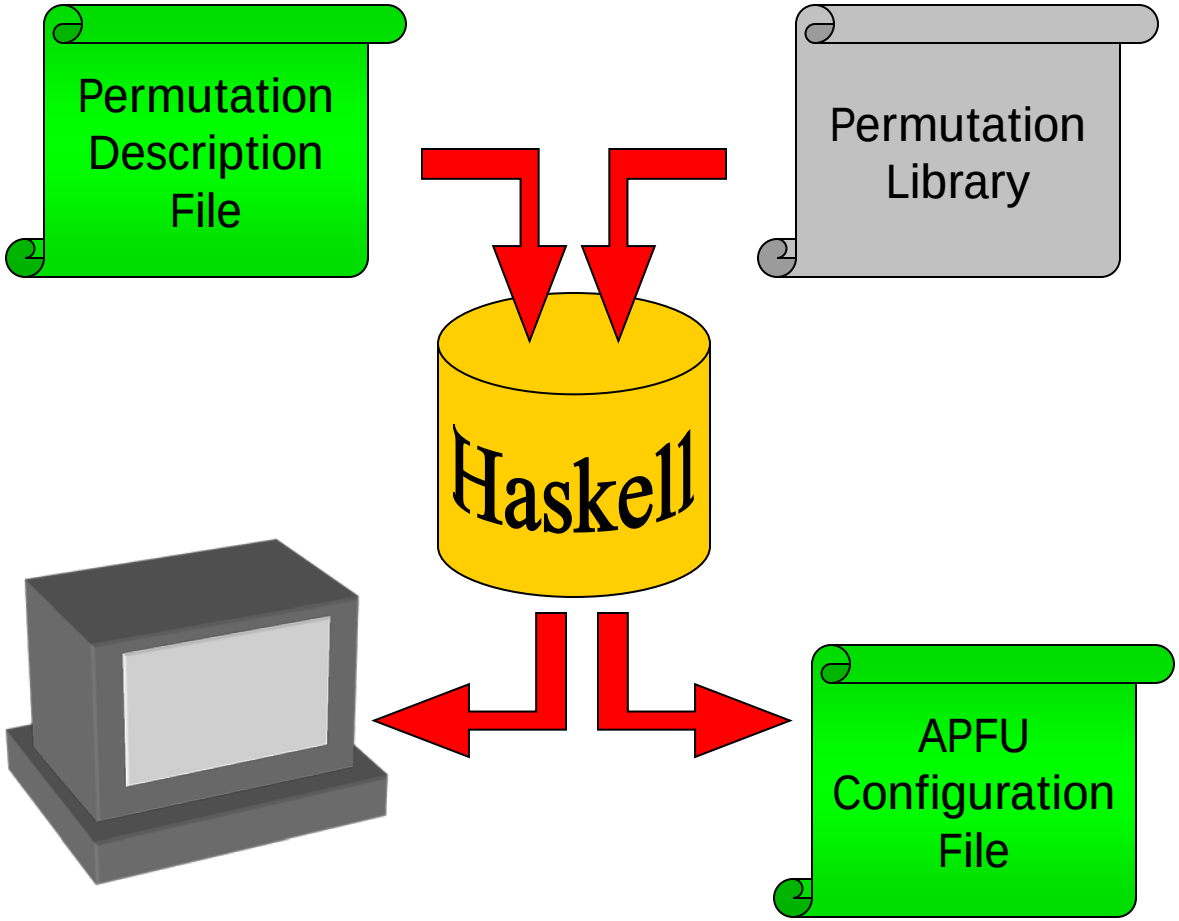
- Semantic analysis of AIM microcode
  - How do the components of the AIM crypto-processor behave?
  - Does a piece of PCE microcode meet its specification? Where does it go wrong?
  - Automatic generation of AIM microcode



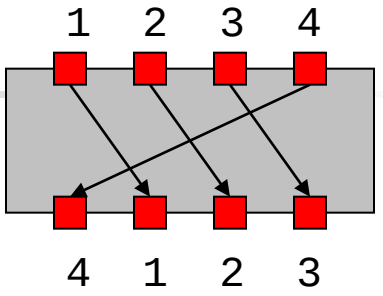
- In the process we have built the core of some potentially useful tools
  - Generating configuration files for permutation and Sbox specifications
  - AIM microcode single stepper

Permutation Tool

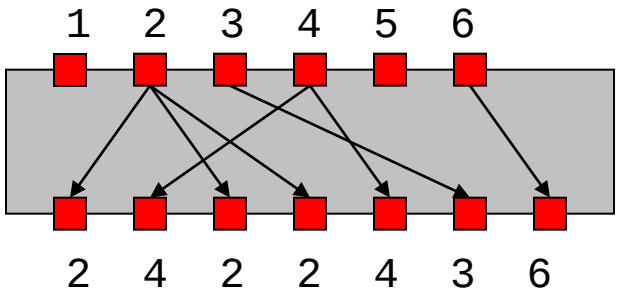
- Permutations are described naturally
- Permutations can be explored to check they behave as predicted
- Permutations can be used to generate APFU configuration files



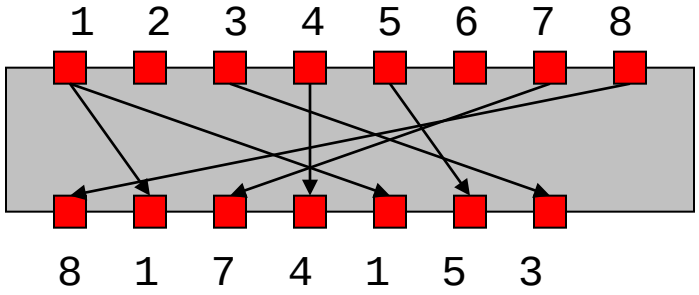
Permutations

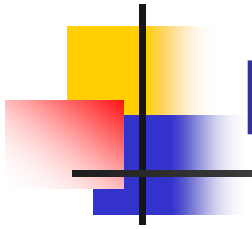


- Lists of numbers
  - Numbered left to right
  - Beginning at 1
- Examples
  - `[4,1,2,3]`
  - `[2,4,2,2,4,3,6]`
  - `[8,1,7,4,1,5,3]`



- Permutations can be any size
  - 16 or 32 bits is common





List definitions

- At their simplest, permutations can be defined by just giving a list of bit positions
- Examples

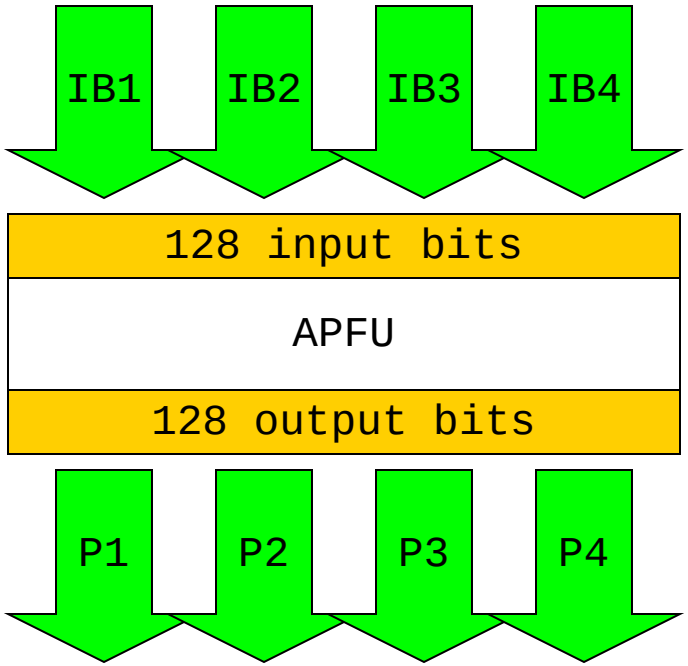
```
desP =
   [16, 7,20,21,29,12,28,17,
     1,15,23,26, 5,18,31,10,
     2, 8,24,14,32,27, 3, 9,
    19,13,30, 6,22,11, 4,25]
```

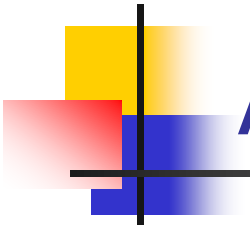
```
desIP =
  [58,50,42,34,26,18,10, 2,
   60,52,44,36,28,20,12, 4,
   62,54,46,38,30,22,14, 6,
   64,56,48,40,32,24,16, 8,
   57,49,41,33,25,17, 9, 1,
   59,51,43,35,27,19,11, 3,
   61,53,45,37,29,21,13, 5,
   63,55,47,39,31,23,15, 7]
```

AIM Input Permutations

- Input buffer permutations
  - Each selects the appropriate Permutation Unit input bits
    - Maps down to 32 output bits

```
ib1 = [1..32]
ib2 = [33..64]
ib3 = [65..96]
ib4 = [97..128]
```

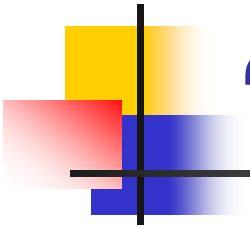




APFU definition

- APFU specification is a record containing
  - Permutation number
  - Four 32-bit permutations p1, p2, p3, p4
- Example

```
apfu5 = APFU {perm = 5,
              p1 = ib4 `into` desP,
              p2 = ib2,
              p3 = ib3,
              p4 = ib1}
```



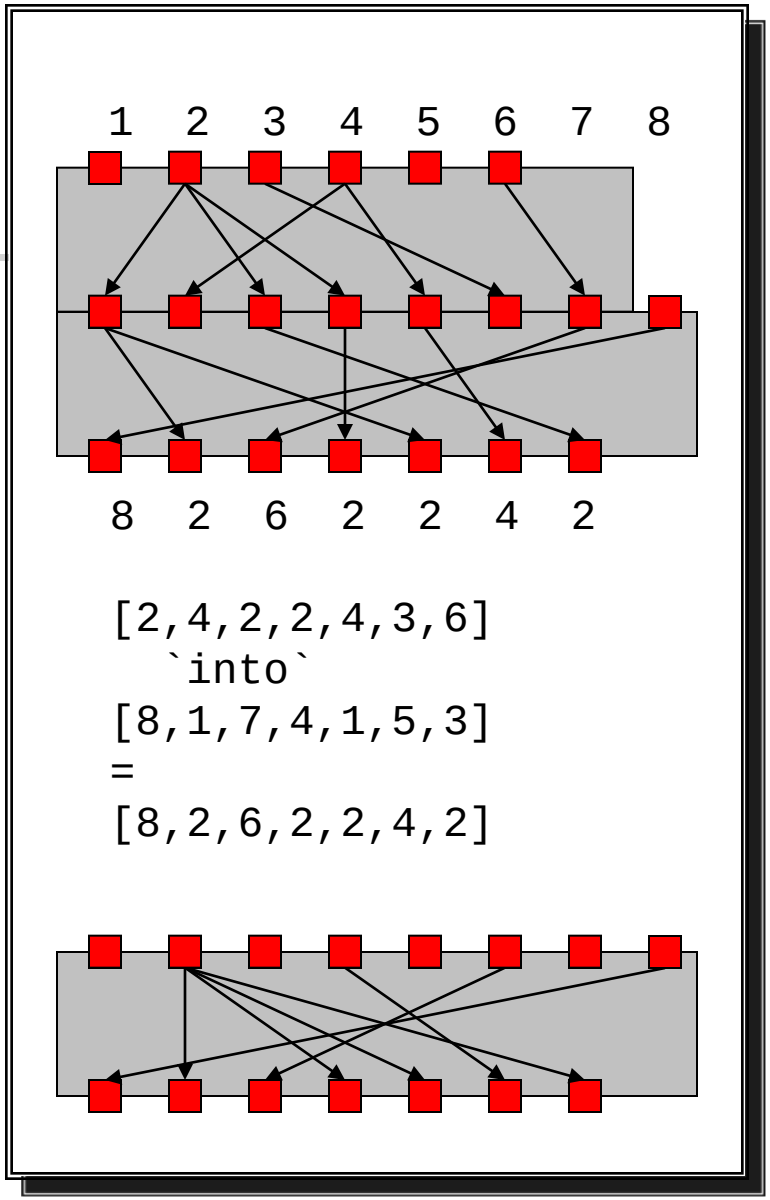
“Lego Block” Permutations

- Permutations are “values”
  - Like integers, complex numbers, polynomials, matrices etc.
  - Don’t think about storage
  - Think about operators
- Question
  - What operators are needed to produce new permutations from old?

`into`

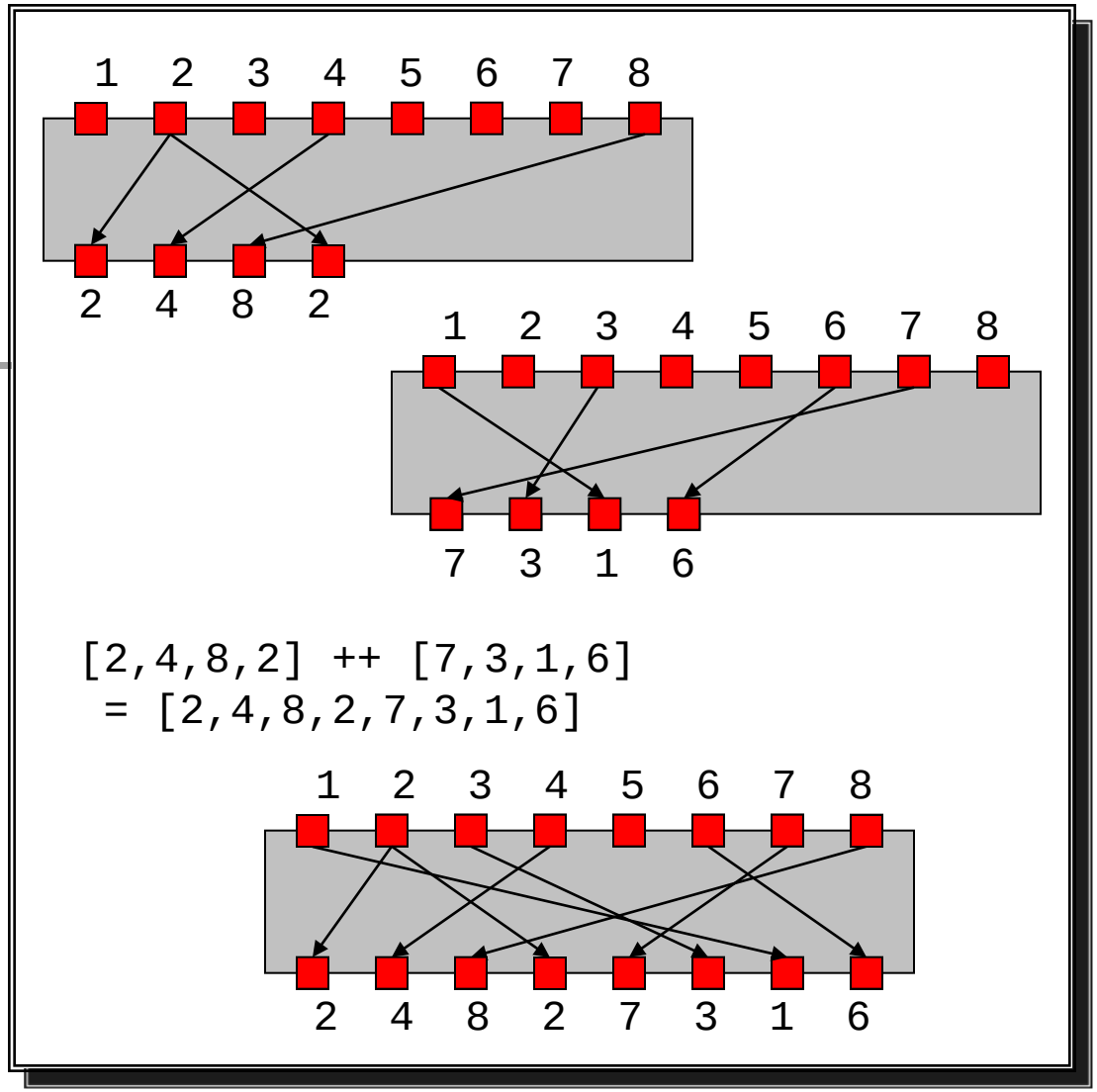
- Pipe the output of one permutation into the input of another
- Calculate the resulting composite permutation
- Example

```
ib4 `into` desP
=
[112,103,116,117,125,108,124,113,
  97,111,119,122,101,114,127,106,
  98,104,120,110,128,123, 99,105,
 115,109,126,102,118,107,100,121]
```



++

- Joins two permutations together, side by side
  - Each permutation draws from the same input bits
  - Obtained simply by appending the two lists together
- Example

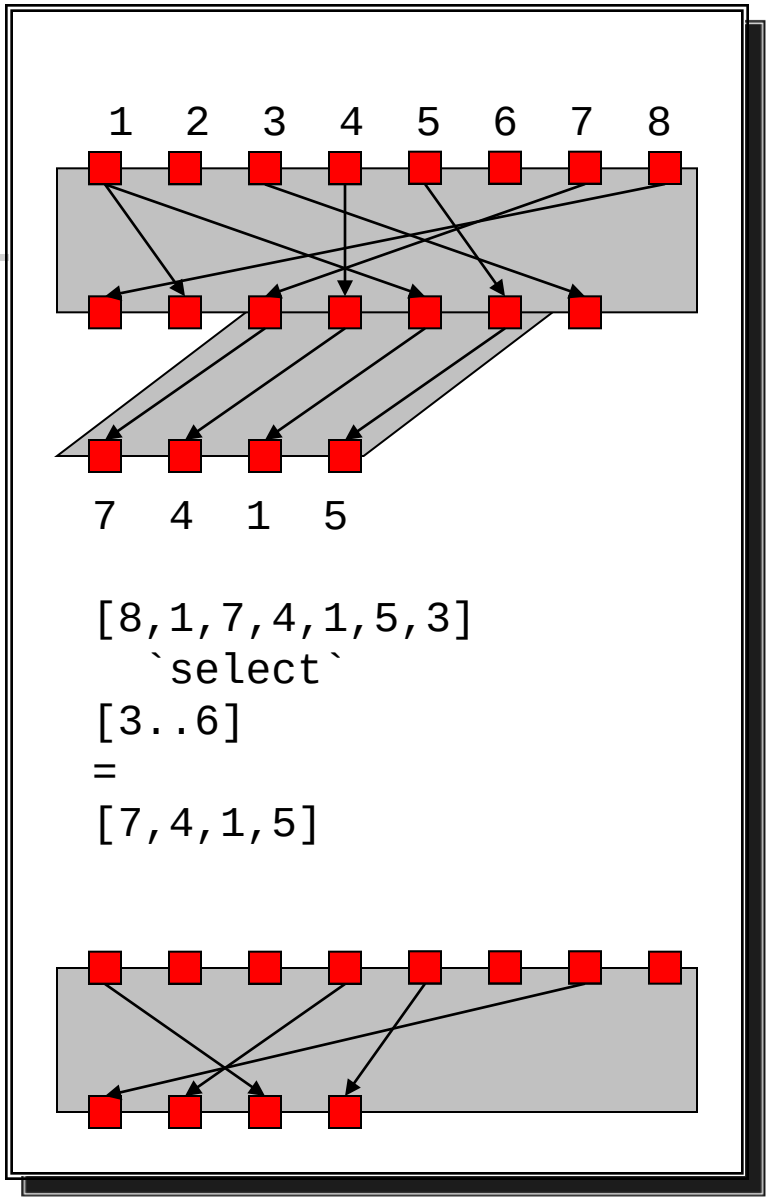



```
ib4 ++ ib3
= [97, 98, 99,100,101,102,103,104,105,106,107,108,109,110,111,112,
  113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,
   65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
   81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96]
```

`select`

- Selects output bits from a permutation
  - Requires a list of contiguous output bits
- Example

```
(ib3 ++ ib2) `select` [17..64]
=
[81,82,83,84,85,86,87,88,
 89,90,91,92,93,94,95,96,
 33,34,35,36,37,38,39,40,
 41,42,43,44,45,46,47,48,
 49,50,51,52,53,54,55,56,
 57,58,59,60,61,62,63,64]
```

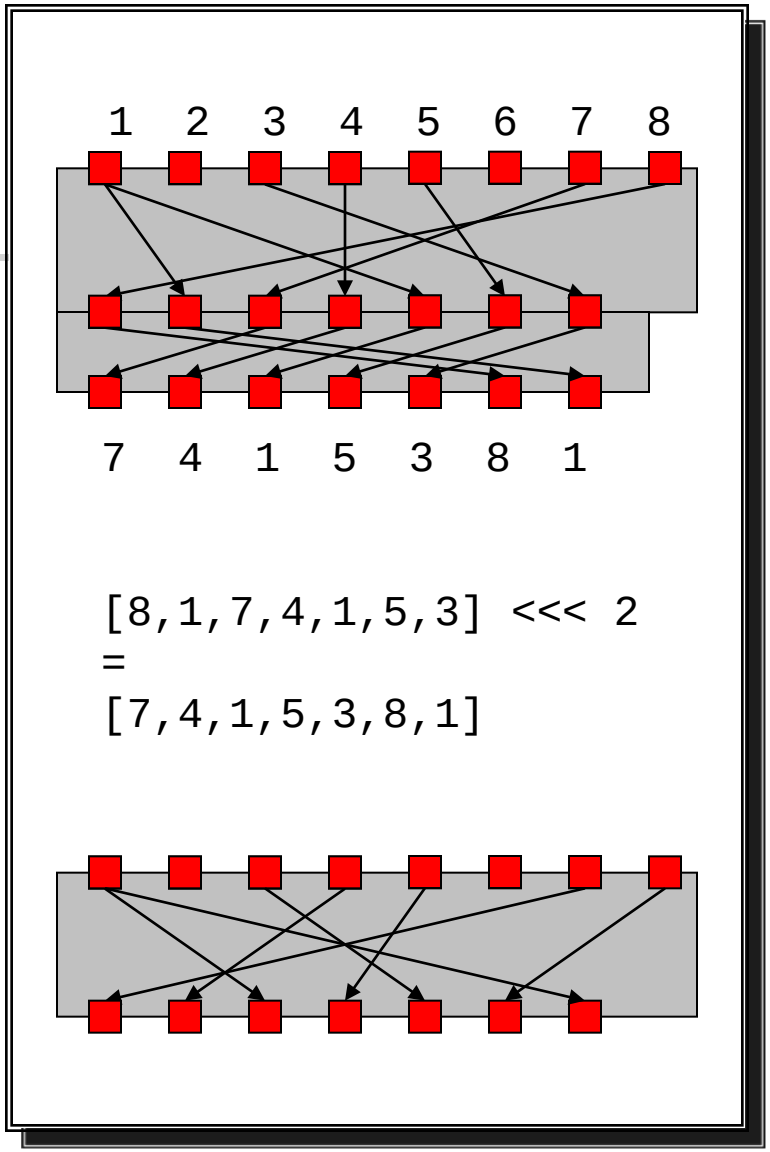


<<<

- Left rotation of a permutation
- Example

```
[5..32] <<< 2
=
[ 7, 8, 9,10,11,12,13,14,
 15,16,17,18,19,20,21,22,
 23,24,25,26,27,28,29,30,
 31,32,5,6]
```

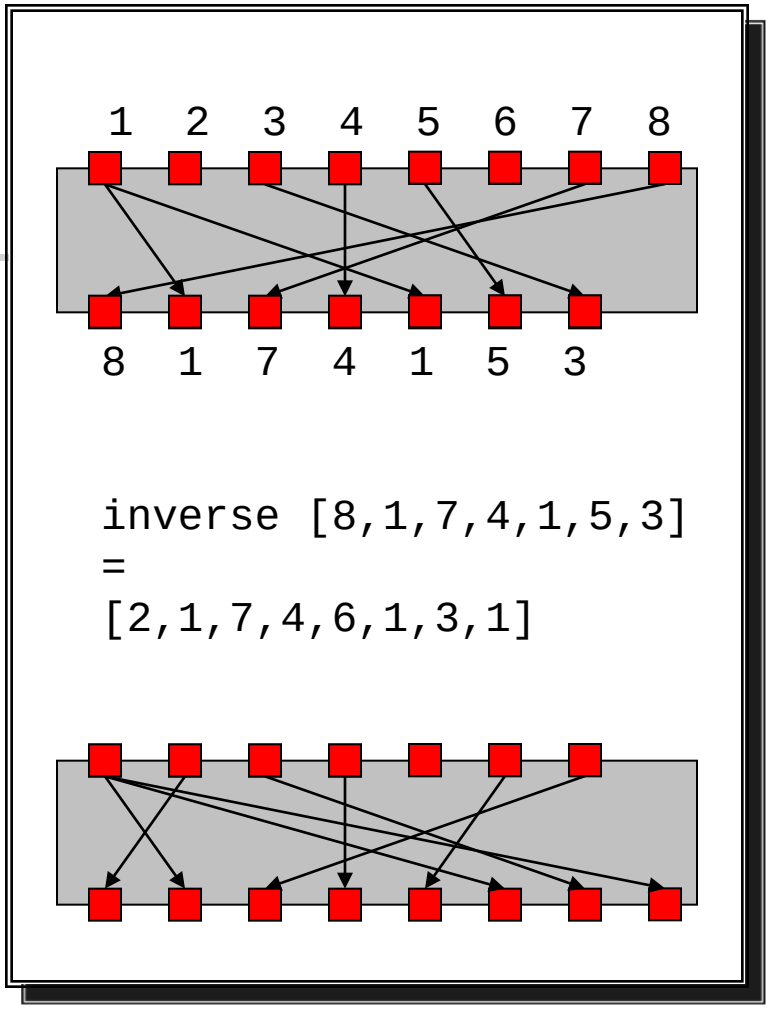
- Right rotation (>>>) is the converse



inverse

- Invert a permutation
- Example

```
inverse desIP
= [40,8,48,16,56,24,64,32,
   39,7,47,15,55,23,63,31,
   38,6,46,14,54,22,62,30,
   37,5,45,13,53,21,61,29,
   36,4,44,12,52,20,60,28,
   35,3,43,11,51,19,59,27,
   34,2,42,10,50,18,58,26,
   33,1,41, 9,49,17,57,25]
```

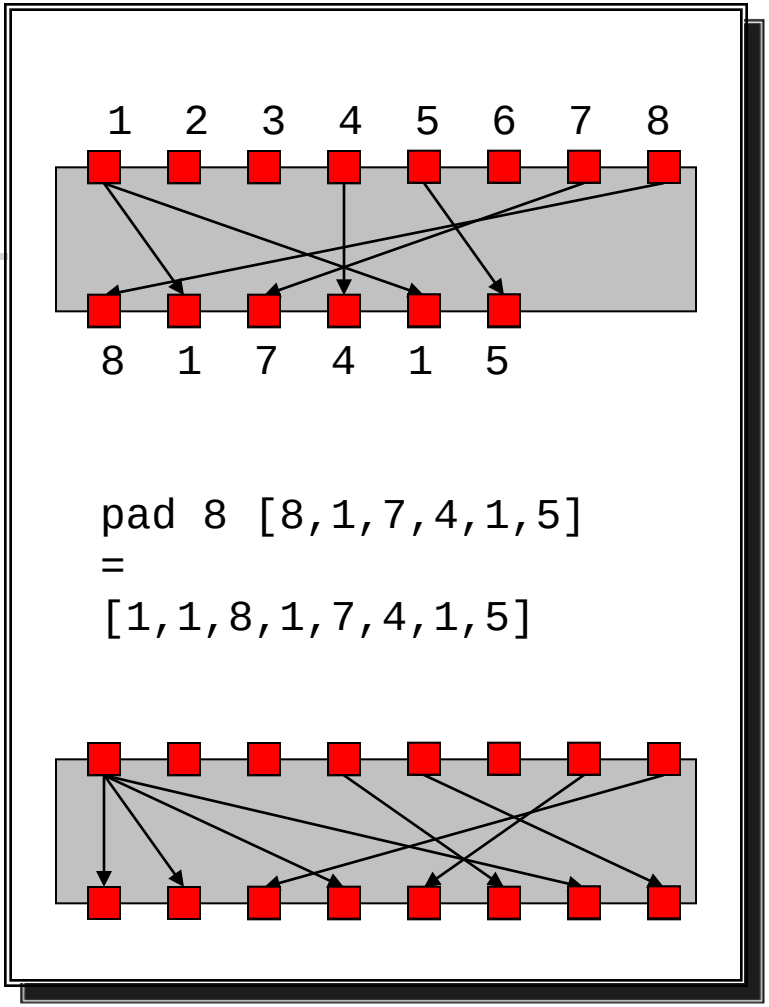


- If p is a true permutation (no duplication, no losses) then
  - `inverse p `into` p = id = p `into` inverse p`

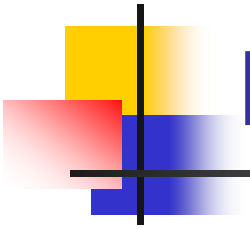
pad

- Pad the output of a permutation to make its output the desired width
- Example

```
pad 32 ([5..32] <<< 1)
= [ 1, 1, 1, 1, 6, 7, 8, 9,
   10,11,12,13,14,15,16,17,
   18,19,20,21,22,23,24,25,
   26,27,28,29,30,31,32, 5]
```

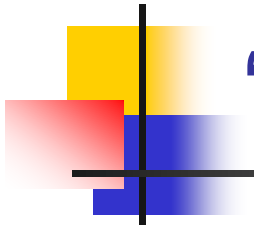


- NB. It is an error to pad less than the size of the permutation
  - `pad 4 [1,2,3,4,5]  -->  Error`



Example Definitions

```
apfu1 = APFU {perm = 1,
              p1 = pad 32 (expansion `select` [1..16]),
              p2 = expansion `select` [17..48],
              p3 = ib1,
              p4 = initialPerm `select` [33..64]}
   where
     initialPerm = (ib3 ++ ib4) `into` desIP
     expansion = (initialPerm `select` [33..64]) `into` desE


desE  = [32, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9
        , 8, 9,10,11,12,13,12,13,14,15,16,17
        ,16,17,18,19,20,21,20,21,22,23,24,25
        ,24,25,26,27,28,29,28,29,30,31,32, 1  ]
```



“Value” of APFU1

```
DES> apfu1

APFU{perm=1,
     p1=[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
         71,121,113,105,97,89,97,89,81,73,65,123,65,123,115,107],
     p2=[99,91,99,91,83,75,67,125,67,125,117,109,101,93,101,93,
         85,77,69,127,69,127,119,111,103,95,103,95,87,79,71,121],
     p3=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
         17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32],
     p4=[121,113,105,97,89,81,73,65,123,115,107,99,91,83,75,67,
         125,117,109,101,93,85,77,69,127,119,111,103,95,87,79,71]
    }
```

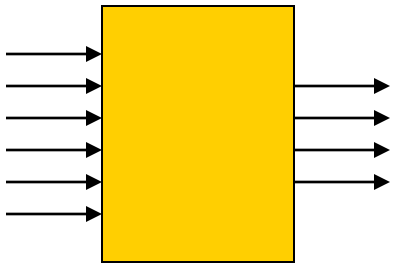
Displaying APFUs

NB. All indices have been automatically converted to be AIM compliant (numbered right to left, starting at 0)

```
Main> viewCode apfu5
PERM5:
        P1_31(IB4_16)|  P2_31(IB2_31)|  P3_31(IB3_31)|  P4_31(IB1_31)|
        P1_30(IB4_25)|  P2_30(IB2_30)|  P3_30(IB3_30)|  P4_30(IB1_30)|
        P1_29(IB4_12)|  P2_29(IB2_29)|  P3_29(IB3_29)|  P4_29(IB1_29)|
        P1_28(IB4_11)|  P2_28(IB2_28)|  P3_28(IB3_28)|  P4_28(IB1_28)|
        P1_27(IB4_3)|   P2_27(IB2_27)|  P3_27(IB3_27)|  P4_27(IB1_27)|
        P1_26(IB4_20)|  P2_26(IB2_26)|  P3_26(IB3_26)|  P4_26(IB1_26)|
        P1_25(IB4_4)|   P2_25(IB2_25)|  P3_25(IB3_25)|  P4_25(IB1_25)|
        P1_24(IB4_15)|  P2_24(IB2_24)|  P3_24(IB3_24)|  P4_24(IB1_24)|
        P1_23(IB4_31)|  P2_23(IB2_23)|  P3_23(IB3_23)|  P4_23(IB1_23)|
        P1_22(IB4_17)|  P2_22(IB2_22)|  P3_22(IB3_22)|  P4_22(IB1_22)|
        P1_21(IB4_9)|   P2_21(IB2_21)|  P3_21(IB3_21)|  P4_21(IB1_21)|
        P1_20(IB4_6)|   P2_20(IB2_20)|  P3_20(IB3_20)|  P4_20(IB1_20)|
        :
        :
        P1_2(IB4_21)|   P2_2(IB2_2)|    P3_2(IB3_2)|    P4_2(IB1_2)|
        P1_1(IB4_28)|   P2_1(IB2_1)|    P3_1(IB3_1)|    P4_1(IB1_1)|
        P1_0(IB4_7)|    P2_0(IB2_0)|    P3_0(IB3_0)|    P4_0(IB1_0);
```

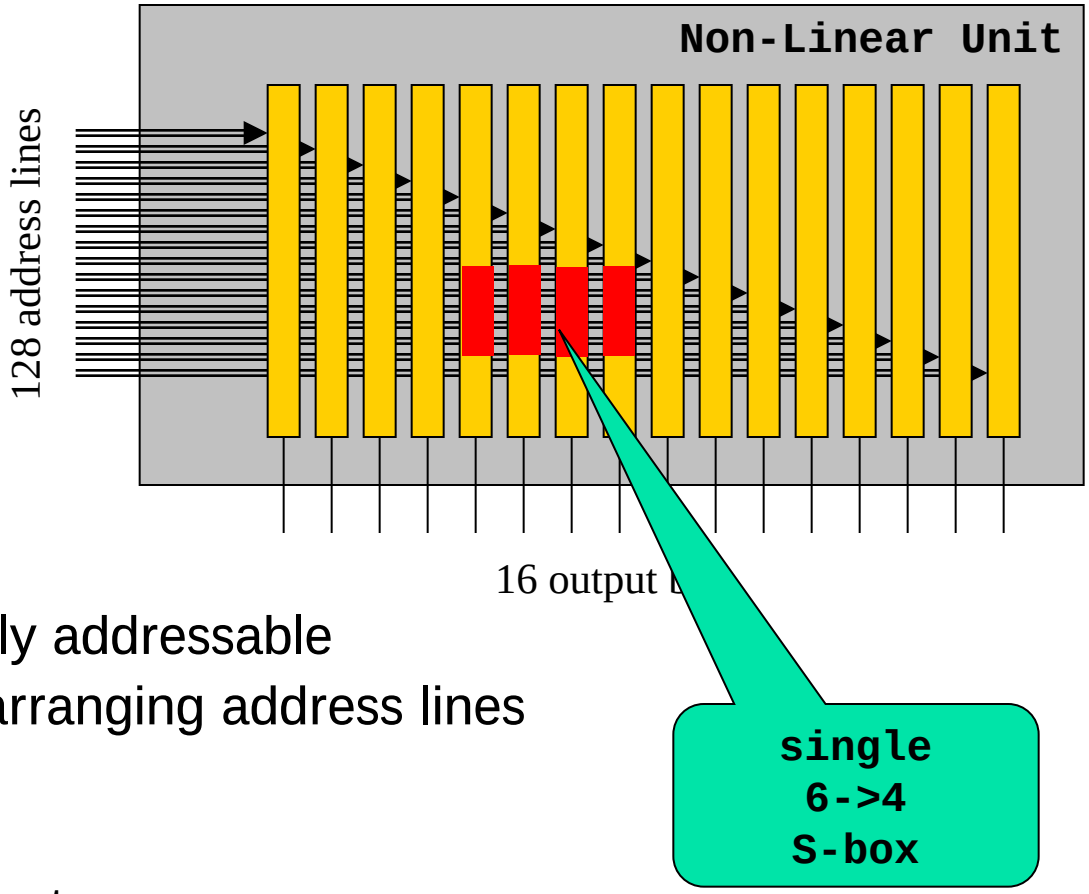
S-boxes

- Every crypto-algorithm needs non-linear components
  - Multiplication (RC6)
  - Galois field inversion (Rijndael)
  - Arbitrary functions, S-boxes
    - DES has 8 separate S-boxes
    - Each 6-bit in, 4-bit out

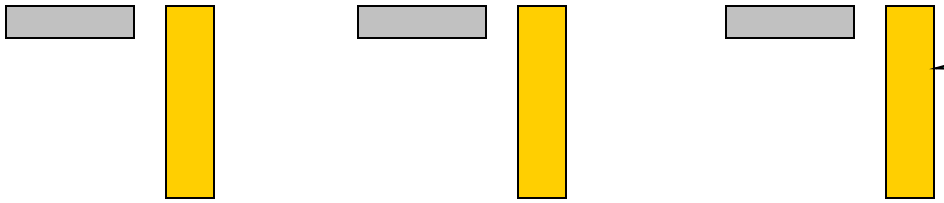


Implementing S-boxes

- NLU designed for S-box functionality
- Lookup tables allow arbitrary functions
  - Sixteen 256x1 tables
  - Each table independently addressable
  - Permutations used for arranging address lines
  - Examples
    - 256 x 16-bit words
    - 512 bytes, dual read ports



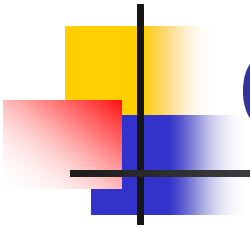
Modeling S-boxes



Permutation selects address lines for indexing its bit-list

Bit-list entries. |bit-list| = 2 ^ |perm|

- Underlying model
  - Sequence of permutation/bit-list pairs
  - Component permutations will be joined to build the final addressing permutation
  - Component bit-lists will be joined to construct the values of each look-up table
- What are the compositional operations?
  - Primitive construction of a single S-box from a list of values
  - Joining S-boxes together side-by-side, or vertically
  - Rearranging addressing lines
  - Computing new S-boxes from old



Operators

- Build a primitive S-box given an addressing permutation, the number of output bits, and a table of data

```
sbox :: Perm -> Int -> [Integer] -> Sbox
```

- Connect multiple S-boxes together vertically, given an addressing permutation which distinguishes which S-box is required

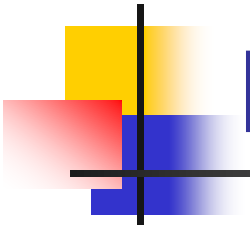
```
pack :: Perm -> [Sbox] -> Sbox
```

- Connect multiple S-boxes together horizontally

```
extend :: [Sbox] -> Sbox
```

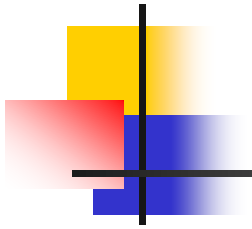
- Pre-compose a new addressing permutation with the S-box permutations

```
intoS :: Perm -> Sbox -> Sbox
```



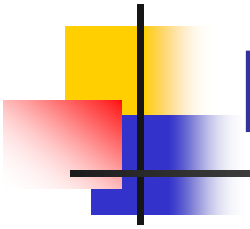
Example definitions

```
sbox1 = pack [1,6] [sbox1a, sbox1b, sbox1c, sbox1d]

sbox1a = sbox [2,3,4,5] 4
         [14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7]
sbox1b = sbox [2,3,4,5] 4
         [0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8]
sbox1c = sbox [2,3,4,5] 4
         [4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0]
sbox1d = sbox [2,3,4,5] 4
         [15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13]
```



Introducing a little abstraction

```
desBox xss = pack [1,6] (map (sbox [2,3,4,5] 4) xss)

sbox4 = desBox
        [[7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15]
        ,[13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9]
        ,[10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4]
        ,[3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14]
        ]
```



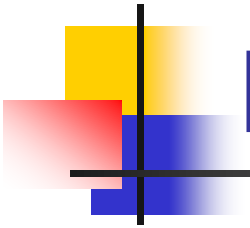
Packing the table

```
layer1 = extend [                    sbox1,
                  [9..14] `into` sbox2,
                 [17..22] `into` sbox3,
                 [25..30] `into` sbox4]

layer2 = extend [                    sbox5,
                  [9..14] `into` sbox6,
                 [17..22] `into` sbox7,
                 [25..30] `into` sbox8]

allDES = pack [7] [layer1, layer2]
```

Generating AIM configurations

- All the information can be calculated for the AIM configuration file
  - Format requires rearranging NLU data as 16-bit words
- Use the `viewSbox` command

```
> viewSbox allDES
NLF_BLOCK:
 F0(0xefa7) |
 F1(0x410d) |
 F2(0xd89e) |
 F3(0x1ee3) |
 F4(0x2660) |
 :
 F124(0xa6e3) |
 F125(0x4025) |
 F126(0x5836) |
 F127(0x3dcb);
```



Laws

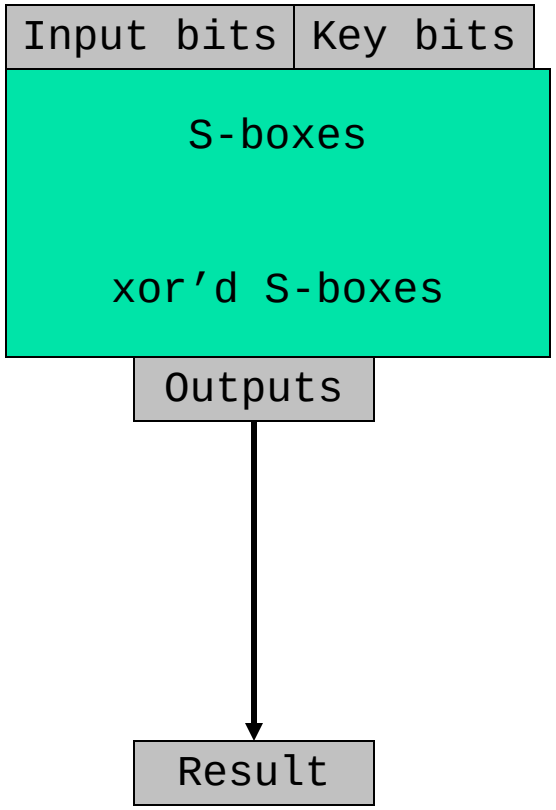
- Formal basis means that we should look for natural algebraic laws
  - Helps provide understanding of the operators

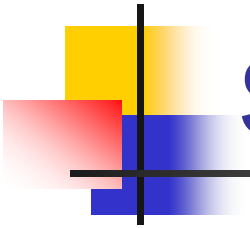
- Pre-composition law

```
p `intoS` (sbox q n xs)
 =
sbox (p `into` q) n xs
```

Exploiting the power

- DES S-boxes only require half NLU tables
  - What should the rest of the space be used for?
- Usually, table filled out with duplication
- Instead, pre-compute xor with key material





Summary

- Formal modeling
  - Family of operators for building permutations and S-boxes
  - Formal semantics for the operators
  - Commands for generating AIM configuration files
- Lessons
  - Separation of model from display
  - Power of little domain-specific languages
    - Made cheap by embedding in clean host language