Details of the ZPL Language

Specifics of the ZPL language are presented, together with some example applications.

1

Compute Value of Rows of Bits

Assume nx32 array of bits

```
  program Radix2;
    config var n : integer = 100;
    region     R = [1..n, 1..32];
    direction  E = [0,1];

  procedure Radix2();
    var Bits : [R] char;
         Val : [1..n,1] integer = 0;
            i: integer;
[1..n,1] begin
          read(Bits);
          for i := 1 to 32 do
            Val := 2*Val + Bits
     [R]   Bits := Bits@E;
          end;
          write ("Values of Bits: ", Val);
        end;
```

2

Regions

- Regions are named sets of index tuples
- Regions are declared with syntax

```
region <name> = [<ll>..<ul> {, <ll>..<ul>}* ]
```

- For example

```
region R = [1..n, 1..n];    -- Std 2-dim region
region V = [0..m-1];        -- 0-origin
```

- Short names common; caps by convention
- Specify stride with **by** following the limits,

```
region Evens = [0..n by 2]; -- 0, 2, 4, ...
```


Declaring Variables

- Variable declarations have the form of a list followed by colon (:) followed by a datatype

```
var x, y, z : double;
```

- The type of an array is a pair

```
[<region>] <data type>
```

- The region can be named or explicit

```
var A, B, C : [R] double;
  Small_data : [1..n] byte;
```

- Arrays passed as parameters must have this type given in the formal parameter

Regions Controlling Array Stmt Execution

Regions specify the indices over which computation will be performed

- Specify region in brackets as statement prefix

```
[1..n,1..n]    A := B;
```

- The n^2 elements of the region are replaced in **A** by their corresponding elements in **B**
- Regions are scoped

```
[1..n,1] begin -- Work on first column only
            A := 0;
            B := 2*C;
         end;
```

5

More About Regions

- With explicit indices leave a dimension blank to inherit from enclosing scope

```
[1..n, 1] begin
                X := Y; -- replace first column
     [ , 2]   X += X; -- double second column
          end;
```

- *Arrays must "conform" in rank and both define elements for indices of region*
- "Applicable region" for assignments are (generally) the most tightly enclosing region of the rank of the *left hand side*

6

Directions

- Directions are vectors pointing in index space
- Declare directions using

```
direction <name> = [ <tuple> ]
```

  where **<tuple>** is a sequence of indices separated by commas
- For example

```
direction northwest = [-1, -1];
                    right = [1];
```

- Short names are common and preferred

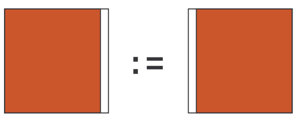
7

The @ Operator

The @ operator takes as operands an array variable and a direction, and returns an array whose values come from the given array offset from the prevailing region by direction

```
[1..n,1..n-1]  A := B@e;  -- assume e = [0,1]
```

- Assign **A[r,s]** the value **B[r,s+1]**
- That is, **B@e** contains the last n-1 columns of **B**, which are assigned to the first n-1 columns of **A**



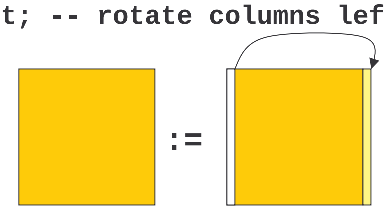
The @ must reference defined values

Wrap-@

The @-operator has (recently) been extended to automatically wrap-around an array rather than “falling off” -- excellent for “periodic boundaries”:

“Falling off” relative to the declared dimensions

```
var A : [1..n,1..n] double; -- array of doubles
…
  A := A@^east; -- rotate columns left
```



9

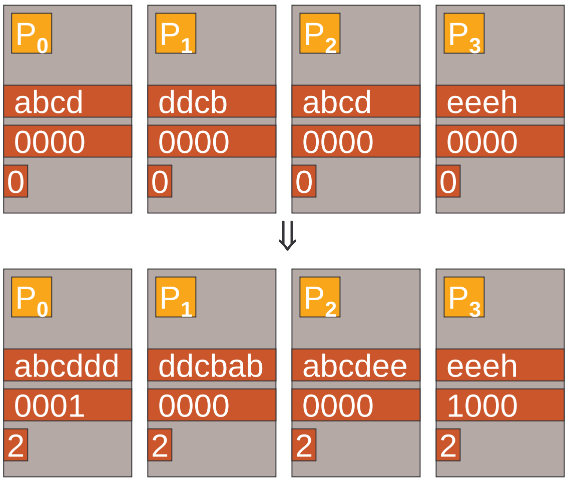
3 Identical Values In Sequence

```
region    V = [1..n];
var Letters : [V] char;
        Seq : [V] boolean;
    triples : integer;
direction r = [1]; r2 = [2];
…
[1..n-2] begin
            Seq     := (Letters = Letters@r)
                        & (Letters = Letters@r2);
            triples := +<< Seq;
         end;
```

10

What Happens

- Send left
- Compare +1
- Compare +2
- Local +
- Accum Tree
- Bdcast Tree



11

Conway's Life

```
program Life;
config var n : integer = 512;
region     R = [1..n, 1..n];

direction NW = [-1,-1]; N = [-1, 0]; NE = [-1, 1];
           W = [ 0,-1];              E = [ 0, 1];
          SW = [ 1,-1]; S = [ 1, 0]; SE = [ 1, 1];

var    Ncount : [R] byte; TW : [R] boolean;
procedure Life();
      [R] begin
          /* Read in the data */
           repeat

         Ncount := (TW@^NW + TW@^N  + TW@^NE
                  +   TW@^W         +  TW@^E
                  + TW@^SW + TW@^S  + TW@^SE );
             TW := (Ncount=2 & TW) | (Ncount=3 & !TW);
           until ! |<<TW;

           end;
end;
```

12

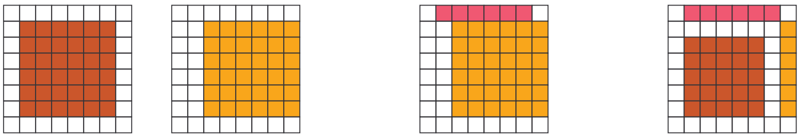
Region Operators

ZPL has region operators taking as operands a region and a direction, and producing a region

- at translates the region’s index set in the direction
- of defines a new region adjacent to the given region along direction edge and of direction extent

```
region R = [1..8,1..8];
       C = [2..7,2..7];
var X, Y : [R] byte;
```

```
Direction e = [ 0,1];
          n = [-1,0];
         ne = [-1,1];
```



`[C] X:=`   `[C at e] Y:=`   `[n of C] Y:=`   `[C] Y:=X@ne`

13

Index1 ...

- ZPL comes with “constant arrays” of any size
- Index*i* means indices of the ith dimension

```
[1..n,1..n]  begin
               Z := Index1; -- fill with first index
               P := Index2; -- fill with second index
               L := Z=P;    -- define identity array
             end;
```

- These array -- of arbitrary dimension -- are compiler created using no space

14

Scan

- Scan is the parallel prefix operation for associative operators: `+, *, min, max, &, |`
- Scan is like reduction, but uses `||`
- Prefix sum from the first lecture is `+||`

```
    A ⇔  2  4   6   8   0
 +||A ⇔  2  6  12  20  20
```

- Yes, "or scan" is `|||` as in

```
          B   ⇔ 0  0  0  1  1  0  1  1
 Run:=|||B   ⇔ 0  0  0  0  1  1  1  1
 [2..n] Run := (Run != Run@w)*Index1;
        pos := max<< Run;
```

Think globablly

15