Details of the ZPL Language

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

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

- Regions are named sets of index tuples
- Regions are declared with syntax
  \[ \text{region } \langle \text{name} \rangle = [\langle ll \rangle..\langle ul \rangle \{, \langle ll \rangle..\langle ul \rangle\}^* ] \]
- For example
  \[ \text{region } R = [1..n, 1..n]; \quad \text{-- Std 2-dim region} \]
  \[ \text{region } V = [0..m-1]; \quad \text{-- 0-origin} \]
- Short names common; caps by convention
- Specify stride with \textbf{by} following the limits,
  \[ \text{region } \text{Evens} = [0..n \text{ by } 2]; \quad \text{-- 0, 2, 4, ...} \]

Declaring Variables

- Variable declarations have the form of a list followed by colon (:) followed by a datatype
  \[ \text{var } x, y, z : \text{double;} \]
- The type of an array is a pair
  \[ [\langle \text{region} \rangle] <\text{data type}> \]
- The region can be named or explicit
  \[ \text{var } A, B, C : [R] \text{ double;} \]
  \[ \text{Small_data : [1..n] byte;} \]
- 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
  \[ [1..n, 1..n] \quad A := B; \]

- The \( n^2 \) elements of the region are replaced in \( A \) by their corresponding elements in \( B \)

- Regions are scoped
  \[ [1..n, 1] \quad \text{begin} -- \text{Work on first column only} \]
  \[ \quad A := 0; \]
  \[ \quad B := 2*C; \]
  \[ \quad \text{end}; \]

More About Regions

- With explicit indices leave a dimension blank to inherit from enclosing scope
  \[ [1..n, 1] \quad \text{begin} \]
  \[ \quad X := Y; -- \text{replace first column} \]
  \[ \quad [*, 2] \quad X += X; -- \text{double second column} \]
  \[ \quad \text{end}; \]

- 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
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

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”:

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

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;
```
What Happens

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

Conway’s Life

```haskell
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;
```
Region Operators

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

- \text{at} translates the region's index set in the direction
- \text{of} defines a new region adjacent to the given region along direction edge and of direction extent

\begin{align*}
\text{region } R &= [1..8,1..8]; \\
C &= [2..7,2..7]; \\
\text{var } X, Y : [R] \text{ byte};
\end{align*}

\begin{align*}
\text{Direction } e &= [0,1]; \\
\text{of } n &= [-1,0]; \\
\text{of } e &= [-1,1];
\end{align*}

\begin{align*}
[C] \ X : &= [C] \ X : \\
[C \text{ at } e] \ Y : &= [C \text{ at } e] \ Y : \\
[n \text{ of } C] \ Y : &= [n \text{ of } C] \ Y : \\
[C] \ Y : &= X @ ne
\end{align*}

Index1 ...

- ZPL comes with “constant arrays” of any size
- \text{Index}\ i \text{ means indices of the } i^{\text{th}} \text{ dimension}

\begin{align*}
[1..n,1..n] \begin{align*}
\text{begin} \\
Z := \text{Index1}; &\quad \text{fill with first index} \\
P := \text{Index2}; &\quad \text{fill with second index} \\
L := Z=P; &\quad \text{define identity array} \\
\text{end}; \\
\end{align*}
\end{align*}

- These array -- of arbitrary dimension -- are compiler created using no space
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 + ||
  \[ A \leftrightarrow 2 4 6 8 0 \]
  \[ + || A \leftrightarrow 2 6 12 20 20 \]
- Yes, “or scan” is ||| as in
  \[ B \leftrightarrow 0 0 0 1 1 0 1 1 \]
  \[ Run := ||| B \leftrightarrow 0 0 0 0 1 1 1 1 \]
  \[ [2..n] \text{ Run := (Run } \neq \text{ Run@w})*\text{Index1;} \]
  \[ \text{ pos := max} << \text{ Run}; \]