Additional ASC Programming Comments

• NOTE: These are additional notes to be added to “ASC Programming” slides by Michael Scherger.

• Comparison of logical parallel and index parallel
  – A index parallel variable selects a single scalar value from a parallel variable.
  – A logical parallel variable L is normally used to store the result of a search such as
  – ASC implementation simplifies usage by not formally distinguishing between the two.
    • The correct type should be selected to improve readability.

• Mixed mode operations are supported and their result has the “natural” mode. For example, if
  
  ```
  int scalar a, b, c;
  int parallel p[S], q[S], r[S], t[S,4];
  index parallel x[S], y[S];
  then
  c = a + b   scalar integer
  q[S] = a + p[S]   parallel integer variable
  a + p[x]   integer value
  r[S] = t[x,2]+3*p[S]   parallel integer variable
  x[S] = p[S] .eq. r[S]   index parallel variable
  ```

• Array Dimensions
  – Int parallel can have up to 3 dimensions
    • First dimension is “S”, the parallel dimension
  – The array numbering is zero-based, so the declaration
    ```
    int parallel A[S,2]
    ```
  creates the following 1-dimensional variables:
    \[ A[S,0], A[S,1], A[S,2] \]

• Dynamic Storage allocation
  – allocate is used to activate a cell to store a new association record
    • Creates a parallel index that points to the new cell
  – release is used to de-allocate storage of specified records in association
    • Can release multiple records simultaneously.
  – Example:
    ```
    char parallel   node[S], parent[S];
    logical parallel   tree[S];
    index parallel   x[S];
    associate node[S], level[S], parent[S] with tree[S];
    ......
    allocate x in tree[S]
    node[x] = 'B'
    endallocate x;
    release parent[S] .eq. 'A' from tree[S].
    ```

• Parallel IF-THEN-ELSE Example and Mask Trace
  ```
  if A[S] .eq. 2
    then  A[S] = 5;
    else   A[S] = 0;
  endif;
  ```

  ```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
  ```

• any – elsenany statement
  – All active cells execute statements inside the any-block if there is one responder.
  – If there are no responders, then all active cells execute the statements inside the elsenany block
  – any can be used alone (without the elsenany)
  – Example
    ```
    any A[S] .eq. 10
    B[S] =11;
    elsenany
    B[S] = 100;
    endany;
    ```

• for construct
  – Often used when a process must be repeated for each cell that satisfies a certain condition.
  – The index variable is available throughout the body of the for statement
  – The index value of for is only evaluated initially
  – Example:
    ```
    sum = 0;
    for x in A[S] .eq. 2
    sum = sum + B[S];
    endfor x;
    ```

  ```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1st</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2nd</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
<td>3rd</td>
<td>10</td>
</tr>
</tbody>
</table>
  ```

• Loop-Until Construct for sequential repetitions
  – Used for sequential type repetitions
  – See earlier slide and primer for details
• **while** construct
  - Unlike the *for* statement, this construct re-evaluates the logical conditional statement prior to each execution of the body of the *while*.
  - The bit array resulting from the evaluation of the conditional statement is assigned to the index parallel variable on each pass.
  - The index parallel array is available for use within the body each loop.
  - The body of the *while* construct will continue to be executed until there are no responders (i.e., all zeros) in the index parallel variable.
  - Study example and trace in ASC Primer carefully to make sure you understand *while*.

• **get** statement
  - Used to retrieve a value from a specific field in a parallel variable satisfying a specific conditional statement.
  - Example:
    ```asc
    get x in tail[$].eq. 1
    val[x] = 0;
    endget x;
    ```
  - Read trace of this example in on page 24 of ASC Primer to make sure its action is clear.

• **next** statement
  - Similar to *get* except *next* updates the set of responders each time it is called.
  - Unlike *get*, two successive calls to *next* is expected to select two distinct cells (and two distinct association records).
  - Can be used in loops to sequentially process each responder.
  - See page 22-23 of ASC Primer for more details.

• **maxval** and **minval** functions
  - *maxval* returns the maximum value of the specified items among the active responders.
  - Similarly, *minval* returns the minimum value.
  - Example:
    ```asc
    if (tail[$].neq. 1) then
      k = maxval(weight[$]);
    endif;
    ```
  - See trace of example on pg 27 of Primer.

• **maxdex** and **mindex** functions
  - They return the index of an (association) entry where a maximum or minimum occurs.
  - If maximum/minimum value occurs at more than one location, an arbitrary selection is made as to which index is returned.

• **setscope/endsetscope**
  - *setscope* jumps out of current mask setting to another mask setting.
  - One use is to reactivate currently inactive processors.
  - Also allow immediate return to a previously calculated mask, such as an association.
  - is an unstructured command such as go-to and jumps from current environment to a new environment.
    - Use sparingly
  - **endsetscope** resets mask to preceding setting.

• **Scalar variable input**
  - Static input can be handled in the code.
  - Also, *define* or *deflog* statements can be used to handle static input.
  - Dynamic input is currently not supported directly, but can be accomplished as follows:
    - Reserve a parallel variable *dummy* (of desired type) for input.
    - Reserve a parallel index variable *used*.
    - A value to be stored in scalar variables is first read into *dummy* using a parallel-read and then transferred using *get* or *next* to the appropriate scalar variable.
  - Example:
    ```asc
    read dummy[$] in used[x];
    get x in used[$]
    scalar-variable = dummy[x];
    endget x;
    ```
  - NOTE: Don’t need to use *associate* statement to associate *dummy* with *used*. Omission causes no problems as no check is currently made.