Answer 1. Other than the tools used within the design flow itself, (**Synthesizers, Simulators**) several tools exist that help system designers achieve further reductions in design time. This reduction is achieved as a result of increased convenience, or potential for better design decision-making that they provide. Examples of such tools are:

- **Estimation tools:** Tools that estimate the metrics of a design (i.e. power, performance, area, etc.) from a higher level description, allowing a better understanding of the design's performance and tradeoffs at an early stage in the design cycle. This provides for design time reduction by minimizing the risk of a design being selected that may not eventually meet the design objectives, or satisfy its constraints.
- **Rule-checkers:** These are primary used at the Layout level of the design flow, and are usually embedded as background tasks in the Layout Editor. Examples include:
  1. **Geometrical Design-Rule Checkers:** Geometrical design rules, or just design rules, are layout restrictions that ensure that the manufactured circuit will connect as desired with no short-circuits or open paths. The rules are based on known parameters of the manufacturing process, to ensure a margin of safety and to allow for process errors.
  2. **Electrical Rule Checkers:** Electrical rules are those properties of a circuit that can be determined from the geometry and connectivity without understanding the behavior. For example, the estimated power consumption of a circuit can be determined by evaluating the requirements of each device and trying to figure out how many of the devices will be active at one time. From this information, the power-carrying lines can be checked to see whether they have adequate capacity. In addition to power estimation, there are electrical rules to detect incorrect transistor ratios, short-circuits, and isolated or badly connected parts of a circuit. All these checks examine the network and look for inconsistencies. Thus, whereas design-rule checking does syntax analysis on the layout, electrical-rule checking does syntax analysis on the network.
- **Verification tools:** tools that check for the correctness of a design description. The essential difference between verifiers and rule checkers is that verifiers are given operating requirements to check for each different circuit. Thus any of the previously described rule checkers could be called a verifier if the conditions being checked were explicitly specified as part of the design. Two examples of types of verifiers are:
  1. **Timing verifiers** determine the longest delay path in a circuit to optimize performance and to make sure that the clock cycles are correct.
  2. **Functional verifiers** compare symbolic descriptions of circuit functionality with the derived behavior of the individual parts of the circuit, also described symbolically.

{Reference: http://www.rulabinsky.com/cavd/index.html "Computer Aids for VLSI Design"}

Answer 2: **Relation Ship between Design Styles and Timing Performance:**

Full-Custom Design: In this Design Style, layout elements are handcrafted, and placement and routing are both done manually by expert Layout Designers.

Although time-consuming and difficult, this approach gives human designers full control of the sizing, placement and routing of all circuit elements, thereby allowing the incorporation of human intelligence and intuition into the design. This Layout Style generally provides the finest timing performance for circuits, and is used in environments where there is usually no well-defined constraint on required performance (therefore designers aim as high as possible), i.e. in highly competitive mass-production markets, such as those of General Purpose Processors.

Standard-Cell Layout Style: In this Design Style, Layout elements are logic blocks that perform basic functions, e.g.: logic gates, flip-flops etc.). These logic blocks are provided as part of a 'Cell Library' that has been standardized in some manner (e.g.: all cells have the same height.). Also, constraints are imposed on the placement of these blocks: e.g.: blocks may only be arranged in rows, with room for routing in between rows.

Due to this standardization, only placement and routing of cells are now of concern. Furthermore, these constraints reduce the complexity of the task enough that automated design tools can be used to achieve acceptable results. Design time is therefore considerably reduced. However, the reduction in flexibility of cell-design, placement and routing have the drawback that circuits cannot be optimized for performance beyond a certain point. Thus equivalent designs in Full-custom layout would generally provide much greater performance. Therefore, this design style is restricted to application areas where lower-bounds specifying acceptable timing performance are specified. Examples are embedded devices whose functionality is more important than performance.

Field Programmable Gate Arrays: FPGAs are Gate Array devices that have pre-routed, programmable interconnection resources, distributed among programmable Logic Resources (CLBs). These devices have the advantage that they are not customized at fabrication time, and as such can be mass produced as identical devices, thereby considerably lowering their cost.

The draw back is that timing performance characteristics are heavily dependent on the design being mapped, the placement, and the routing (both generated by automated tools). Due to these characteristics, FPGAs are generally used only in prototyping applications, as well as applications where programmability and adaptability are of critical importance, e.g.: in Satellites. Only recently have FPGAs been considered for high performance computing applications, under a new computing paradigm known as Configurable Computing.

Answer 3: Given below is the code for the required program, "Connecti.exe". The program parses 'netlist.txt', and writes the Connectivity Matrix to 'cmat.txt'.

```
#include "stdio.h"
#include "stdlib.h"
#include "conio.h"
#include "string.h"
#include "process.h"

void main (void)
{
     FILE *netlist, *matrix;
     char *string, *temp1, **nodelist, **distinct, *temp2;
     int **connectivity, count=0, count1=0, count2=0, count3=0, unique=0;

                         //GLOBAL NOTES: 'count' will hold the actual length of 'nodelist'
                         //              'unique' will hold the actual no. of nodes (i.e. the size of 'distinct')


     if ((netlist = fopen("c:\\tc\\bin\\netlist.txt", "r")) == NULL)
     {                                                                                    //open the netlist file
          printf("File not found\n");
          exit(0);
     }

     nodelist = (char**) calloc (30, sizeof(char*));
     for (count = 0;count < 30; count++)                              //allocate memory to hold nodes
     {
          *(nodelist+count) = (char*) calloc (5 ,sizeof(char));
     }

     count = 0;
     //this loop will extract nodes and put them in an array

     string = (char*) calloc(40, sizeof(char*));
     while (!feof(netlist))
     {
          fgets(string, 40, netlist);                                      //get the first line from file
          temp1 = string;                                             //temporary variable
          if (*(string) == '(')                                       //skip the first bracket
               temp1++;
          strtok(temp1, ",");                                              //get first token (first half of 2pt net)

          *(*(nodelist+count)) = *(temp1);                            //write first character
          if(*(temp1+4) == '.')                                       //if gate, write next 2 chars
          {
```

```
          *(*(nodelist+count)+1) = *(temp1+1);
          *(*(nodelist+count)+2) = *(temp1+2);
          *(*(nodelist+count)+3) = *(temp1+3);
          *(*(nodelist+count)+4) = '\0';
     }
     count++;                                                    //increment string array ptr to next string
     temp1 = strtok(NULL, ")");                                 //get second token
     *(*(nodelist+count)) = *(temp1+1);                         //skip pace and write first character of 2nd token
     if(*(temp1+5) == '.')                                      //if gate, write next 2 chars
     {
          *(*(nodelist+count)+1) = *(temp1+2);
          *(*(nodelist+count)+2) = *(temp1+3);
          *(*(nodelist+count)+3) = *(temp1+4);
          *(*(nodelist+count)+4) = '\0';
     }
     count++;
}
fclose(netlist);
printf("Total Node Count: %d\n", count);             //here we print the entire node-list that we have extracted.
/*for (count2=0; count2 < count; count2++)
{
     printf("%d:\t", count2);
     puts(*(nodelist+count2));
     getch();
}*/
//at this point the nodes have been extracted...
//we now need to identify the distinct nodes...and assign them integer values

distinct = (char**) calloc(count, sizeof(char*));
for (count2=0; count2 < count; count2++)                  //allocate memory to hold distinct nodes
{
     *(distinct+count2) = (char*) calloc (4, sizeof(char));
     //*(*(distinct+count2)) = '\0';                          //initialize all strings to \0
}

/*for (count2=0; count2 < count; count2++)                //testing the contents of distinct
{
     puts(*(distinct+count2));
     printf("\t: %d\t", count2+1);
     getch();
}*/


//this nested loop will generate a list of distinct nodes from nodelist

for (count2=0; count2 < count; count2++)                  //counter for nodelist
{
     for (count3=0; count3 <= count2; count3++)           //counter for distinct
     {
          if( *(*(distinct+count3)) == '\0')
```

```
          {
               strcpy (*(distinct+count3), *(nodelist+count2));
               unique++; //holds the number of distinct nodes
          }
          if( strcmp(*(nodelist+count2), *(distinct+count3)) == 0)
               break;
     }
}

for (count2=0; count2 < unique; count2++)                 //testing the contents of distinct
{
     printf("\n%s\t: %d of %d", *(distinct+count2), count2+1, unique);
}

// at this point, we have a list of Distinct nodes, the number
// of which indicates the 'n' of the "n*n" Connectivity Matrix


connectivity = (int **) calloc (unique, sizeof(int*));     //allocate memory to hold Matrix
for (count2 = 0; count2 <= unique; count2++)
     *(connectivity+count2) = (int*) calloc (unique, sizeof(int));

for (count1 = 0; count1 < unique; count1++)
{
     for(count2 = 0; count2 < unique; count2++)
          connectivity[count1][count2] = 0;
}

count1 = 0;
for (count1 = 0; count1 < count; count1+=2)
{
     temp1 = *(nodelist+count1);
     temp2 = *(nodelist+count1+1);

     count2 = 0;
     count3 = 0;
     while (strcmp(*(distinct+count2), temp1) != 0)
          count2++;
     while (strcmp(*(distinct+count3), temp2) != 0)
          count3++;

     connectivity[count2][count3] = 1;
     connectivity[count3][count2] = 1;
}

printf("\n");
for (count1 = 0; count1 < unique; count1++)
     printf("\t%s", *(distinct+count1));
for (count1 = 0; count1 < unique; count1++)
{
```

```
            printf("\n%s\t", *(distinct+count1));
            for(count2 = 0; count2 < unique; count2++)
                  printf("%d\t", connectivity[count1][count2]);
      }

      matrix = fopen("c:\\tc\\bin\\cmat.txt", "w+");
      for (count1 = 0; count1 < unique; count1++)
            fprintf(matrix, "\t%s", *(distinct+count1));
      for (count1 = 0; count1 < unique; count1++)
      {
            fprintf(matrix, "\n%s\t", *(distinct+count1));
            for(count2 = 0; count2 < unique; count2++)
                  fprintf(matrix, "%d\t", connectivity[count1][count2]);
      }
      getch();
}
```

Given below are the contents of the input file ‘netlist.txt’:

```
(A, a2s1.in1)
(B, a2s1.in2)
(A, a2s2.in1)
(C, a2s2.in2)
(C, a2s3.in1)
(B, a2s3.in2)
(a2s1.out, o3s1.in1)
(a2s2.out, o3s1.in2)
(a2s3.out, o3s1.in3)
(o3s1.out, Z)
```

Given below are the contents of the output file ‘cmat.txt’:

```
     A    a2s1 B    a2s2 C    a2s3 o3s1 Z
A    0    1    0    1    0    0    0    0
a2s1 1    0    1    0    0    0    1    0
B    0    1    0    0    0    1    0    0
a2s2 1    0    0    0    1    0    1    0
C    0    0    0    1    0    1    0    0
a2s3 0    0    1    0    1    0    1    0
o3s1 0    1    0    1    0    1    0    1
Z    0    0    0    0    0    0    1    0
```